Efficient Modern C++ Data Structures

Container and Algorithm Optimizations

David Spuler

Aussie AI Labs

Published by Aussie AI Labs Pty Ltd, Adelaide, Australia.

https://www.aussieai.com

First published: May 2025.

About the Author

David Spuler is a C++ expert and serial technology entrepreneur who has combined his love of writing with AI technology in his latest venture: Aussie AI is a suite of tools for writing and editing, with a focus on fiction from short stories to full-length novels. His published works include two advanced C++ books (low latency and safety), two generative AI books, two CUDA C++ books, four non-fiction textbooks on C++ programming covering introductory and advanced C++ programming, efficiency, code optimization, debugging, testing, and software development tools, and one application management book.

Other than writing, he's an avid AI researcher with a Ph.D. in Computer Science and decades of professional experience. Most recently, Dr. Spuler has been founding startups, including the current Aussie AI startup and multiple high-traffic website platforms with millions of monthly uniques, including an e-health startup acquired by HealthGrades, Inc. Prior roles in the corporate world have been as a software industry executive at BMC Software, M&A advisor, strategy consultant, patent expert, and prolific C++ coder with expertise in autonomous agents, compiler construction, internationalization, ontologies and AI/ML. Contact by email to `research@aussieai.com` or connect via LinkedIn.

About the Contributors

Michael Sharpe is an experienced technologist with expertise in AI/ML, cybersecurity, cloud architectures, compiler construction, and multiple programming languages. He is currently Senior Software Architect at PROS Inc., where he is a member of the Office of Technology focusing on developing and evangelizing AI. His AI expertise extends to monitoring, observability, devops, MLOps, ITSM, low-resource LLM inference, Retrieval Augmented Generation (RAG) and AI-based agents.

In a long R&D career, Michael has been coding C++ for almost 30 years, with prior roles at BMC Software, Attachmate (formerly NetIQ) and IT Involve. Michael has a Bachelor of Science with First Class Honors in Computer Science from James Cook University and holds several registered patents. He made major contributions to this book, especially in the chapters on GPU hardware acceleration, LLM training, and RAG architectures, not to mention that he also technically-reviewed the book in its entirety!

Cameron Gregory is a technology entrepreneur including as co-founder of fintech bond trading startup BQuotes (acquired by Moody's), co-founder and Chief Technology Officer (CTO) of Trademark Vision with an AI-based image search product (acquired by Clarivate), and founder of several image creation companies including FlamingText.com, LogoNut, AddText, and Creator.me. Currently a Senior Data Scientist focused on "big data" for hedge funds at fintech startup Advan Research Corporation, he is used to working with real-world data at scale.

Cameron has been making code go fast since the 1990s at AT&T Bell Laboratories in New Jersey, and is proficient in multiple programming languages, including C++, plus Java and JavaScript. He holds a Bachelor of Science with First Class Honors in Computer Science from James Cook University. His contributions to the book included detailed suggestions for scaling a high-traffic cloud architecture underpinning AI engines, and overall software development practices and tools.

Preface

Why a Book on Efficient Data Structures?

There's nothing more important than data structures! The state of the art has advanced so much in the last few years that it's hard to know them all. Even more difficult is knowing how to implement them with optimal efficiency in modern C++.

Please Leave a Review

I hope you enjoy the book! Please consider leaving a review on the website where you purchased the book. Since few readers do this, each review is important to me, and I read them all personally.

Feedback and Contacts

Feedback from readers is welcome. Please feel free to tell us what you think of the book, the literature review, or our Aussie AI software. Contact us by email via `support@aussieai.com`.

Other Books by the Author

If you want fast code, here are a number of other books with a particular focus on AI and fast LLM backends:

- C++ Low Latency: Multithreading and Hotpath Optimizations
- Generative AI Applications: Planning, Design, and Implementation
- Generative AI in C++: Coding Transformers and LLMs
- CUDA C++ Optimization: Programming Faster GPU Kernels
- CUDA C++ Debugging: Safer GPU Kernels
- Safe C++: Fixing Memory Safety Issues

About Aussie AI

Aussie AI is a platform for the development of consumer AI applications, with a special focus on AI-based writing and editing tools for fiction. Our premier applications offer an extensive range of reports and error checks for both fiction and non-fiction writing, from a full-length novel to a short report. Please try it out and let us know what you think: https://www.aussieai.com

Our AI Research

The primary focus of research at Aussie AI is on optimizing LLM inference algorithms (i.e., "running" the model after training or fine-tuning), and our research is toward the following aims:

- Fast on-device model inference algorithms, specifically for smartphones and AI PCs.
- Scaling inference algorithms to large volumes of requests.
- Efficient GPU inference algorithms (hardware acceleration).
- Non-GPU inference optimization algorithms (i.e., software methods).

Disclosure: Minimal AI Authorship

Despite my being involved in the AI industry, there was almost no AI engine usage in creating this book's text or its coding examples. Some text has been analyzed and reviewed using Aussie AI's editing tools, but not even one paragraph was auto-created by any generative AI engine. All of the CUDA C++ code is also human-written, without involvement of any AI coding copilot tools. I mean, who needs them?

However, AI was used in several ways. AI-assisted search tools, such as "Bing Chat with GPT-4", were very useful in brainstorming topics and researching some of the technical issues. The main cover art image was AI-generated, followed by human editing.

Disclaimers

Although I hope the information is useful to you, neither the content nor code in this work is guaranteed for any particular purpose. Nothing herein is intended to be personal, medical, financial or legal advice. You should make your own enquiries to confirm the appropriateness to your situation of any information.

Many code examples are simplistic and have been included for explanatory or educational benefit, and are therefore lacking in terms of correctness, quality, functionality, or reliability. For example, some of the examples are not good at handling the special floating-point values such as negative zero, `NaN`, or `Inf`.

Oh, and sometimes I'm being sarcastic, or making a joke, but it's hard to know when, because there's also a saying that "Truth is often said in jest!" Your AI engine certainly won't be able to help you sort out that conundrum.

Third-Party License Notices

Except where expressly noted, all content and code is written by David Spuler or the contributors, with copyright and other rights owned by David Spuler and/or Aussie AI.

Additional information, acknowledgments and legal notices in relation to this book, the C++ source code, or other Aussie AI software, can be found on the Aussie AI Legal Notices page: https://www.aussieai.com/admin/legal-notices.

Table of Contents

Part I: Optimization Techniques

"Learning to fly is not pretty but flying is."

— Satya Nadella, *Hit Refresh*, 2017.

1. Data Structures Overview

What Are Data Structures?

Data structures are the different ways of organizing data in memory. For example, we can keep all the data together in memory, such as an array. Alternatively, we can spread it out all over different parts of the memory, such as a linked list.

Data Structures versus Algorithms. These two go together very commonly, and the understanding tends to be:

- Data structures — the structure of the data (aha!).
- Algorithms — the code that makes it so.

But it's not a full definition, since there are many algorithms that exist separately from particular data structures. Nevertheless, let's run with that definition for now.

Data Structures versus Databases. Generally speaking, the distinction between a data structure and databases goes like this:

- Data structures — small amount of data stored in memory.
- Databases — huge amount of data on disk.

It's all relative, though, because a "small amount of data" can actually be gigabytes if you're talking about an AI model. And there are exceptions to this rule in both directions:

- On-disk data structures — especially the B-tree.
- In-memory databases — various types exist, even with SQL and all that jazz.

So, let's pretend we now know what a data structure is.

Types of Data Structures

There's a lot of ways to categorize data structures, and all will fail abjectly. If you look at the Wikipedia page for data structures, there are literally hundreds. Every categorization has exceptions, but nevertheless, I'm going to try a few ways.

Contiguous versus Linked Data Structures. When you store data next to each other, that's called a contiguous data structure. The main example is any type of array, or the containers `std::array` and `std::vector`.

Linked data structures are those where the data is spread out more. The different pieces of data are "linked" to each other using pointers. Examples of these include linked lists and binary trees.

Linked lists can be either singly or doubly linked. The reason that the default is two pointers per node is that a doubly-linked list is easier for insertions and deletions, at the cost of some extra space. Modern C++ has both types of linked lists:

- `std::list` — doubly-linked list.
- `std::forward_list` — a singly-linked list.

The standard "binary tree" data structures include `std::map` for a dictionary and `std::set` for set membership. These are both implemented as balanced red-black binary search trees, which we could call BRBBST's, but we don't.

Dictionaries versus Sets. Dictionaries are a data structure category where you can look up something (e.g., a word in a dictionary), and then you find the result from the search (e.g., the definition of a word in a dictionary). The formal name for this class of data structures is "associative data structures." Other common names for dictionary data structures include:

- Maps
- Key-value stores
- Symbol tables
- Hashmaps

All of these mean the same thing: a pair of data items, where one type of data is the "key" that you search for, and the other item is the "value" that you return when you find the key.

Sets are like half a dictionary. With a set, we just look up the key, but there's no value to return. We only want to know whether our "key" is part of the set or not. There's no extra data to return.

Dictionaries and sets are very similar and are typically implemented in the same types of data structures. In both cases, we want to search for a key, and the same methods can be used. The only difference is at the end, whether there's a payload (dictionaries) or nothing (sets).

In modern C++ containers there's a close correlation between those that do dictionary lookup and those doing set membership. There are two container versions that use a "red black tree":

- `std::map` — dictionary in a red-black tree.
- `std::set` — set version.

And there are two classes that are hash tables, which are called "unordered" because the data is not sorted:

- `std::unordered_map` — dictionary hash table.
- `std::unordered_set` — set in a hash table.

I suppose that hashing being "unordered" means that the data in red-black trees is "ordered," but it really doesn't seem that way. They're a linked tree data structure, with the data all around the place. Sure, it's ordered, whatever.

Hash tables. What are hash tables? Well, they enjoy a special type of distinction as they don't really fit well into descriptions of other data structures, such as contiguous versus linked. A hash table can be one or the other, or a kind of hybrid of contiguous and linked data.

But we forgive them because hash tables are super-fast. In fact, they are usually constant-time cost for searches, insertions, and deletions. That's hard to beat, and we'll talk a lot more about modern C++ hash table containers.

Stacks versus Queues. These two data structures are not related to dictionaries or sets. We don't look up anything, but instead we process the data in and out in a particular pattern:

- Stacks — Last-In-First-Out (LIFO)
- Queues — First-In-First-Out (FIFO)

LIFO means that when we put something on a stack, it'll be the first thing we pull off the stack. It's like how restaurants put your receipt on a sharp pointy thing. I always wonder how many times the waiters stab themselves before they learn to avoid it.

Anyway, the FIFO method in queues is the reverse, where whatever we put on the queue will be the last thing we process, after all the other stuff already in our work queue. It's kind of like, you know, a queue that you stand in, where you are actually the job that someone else needs to process.

Modern C++ has containers for both stacks and queues, which are imaginatively named:

- `std::stack`
- `std::queue`

There's also `std::dequeue`, which means a "double-ended queue," since the default queue data structure is a single-ended queue. Paradoxically, a single-ended queue actually has two ends:

- Head — remove new work here.
- Tail — insert here.

And if you want a double-ended stack, don't worry, because it's literally the same thing as a dequeue. There's no difference.

And to further confuse things, there are two different ways to code them up. Both stacks and queues can be implemented in two main ways:

- Contiguous arrays — fixed-size stack or queue.
- Linked lists — unlimited size (dynamic).

However, modern C++ doesn't have any standard containers for the array versions, but only the dynamic linked list versions of stacks and queues.

Heaps and Priority Queues. Priorities queues are a generalization of queues, where each job can have a "priority" and higher-priority items come off the queue first. Hence, it's no longer about FIFO, but about what priority the jobs have, except it's still FIFO for jobs that have the same priority.

Priority queues are also known as "heaps" but you don't need to implement them using the heap, because that's a different thing. Two types of heaps makes a lot of sense, especially since they're completely unrelated uses of the word. Heaps of fun!

Recursive Data Structures. Some data structures are "recursive" in that a sub-part of the data structure looks the same structure as the whole thing. The main examples are binary trees, but other data structures are also technically recursive, such as linked lists.

As you'd expect from the name, you can code algorithms for these data structures using recursion in your function calls. But you probably shouldn't! This is a discussion of efficient coding, after all, and recursive function calls are not that. It's better to use a loop.

Static versus Dynamic Data Structures. Static data structures are ones that don't change, whereas dynamic ones can. The general idea is that dynamic data structures can grow and shrink, whereas static versions are read-only, and don't have any insertions or deletions. An example of a static data structure is "perfect hashing." A good example of a dynamic data structure is everything else.

Fastest Data Structures

Which one is the fastest data structure in the world? And I don't mean a tribute.

My vote for the fastest data structure goes to the hash table. I've hand-coded numerous hash tables in both C and C++, although these days I use the modern C++ container versions. In modern C++, the containers that are hash tables are:

- Dictionary lookup — `std::unordered_map`
- Set lookup — `std::unordered_set`

Unfortunately, you can't use a hash table for everything, although I've certainly tried! For example, a hash table doesn't work for coding a stack or queue data structure.

Complexity vs Practical Performance

How do you assess which data structure is the fastest? The "big-O" mathematical notation for algorithmic complexity is a useful way to think about efficiency. The "O" actually stands for "order" of the mathematical function.

The basic levels of complexity in order of speed include:

- Constant time — O(1)
- Logarithmic — O(log n)
- Linear time — O(n)
- Linear-log — O(n log n)
- Quadratic — O(n^2)

There's levels beyond these like cubic and quartic, such as dense matrix-matrix multiplication (known as "MatMul" or "GEMM"), which is cubic in the size. However, once you get to quadratic complexity, your code is slow already. A lot of the current AI research is about speeding up some parts of LLM algorithms that are horrifyingly quadratic or cubic in cost.

There are actually three different measures, which a data structure can have different orders of complexity:

- Best case
- Average case
- Worst case

For example, as hash table has terrific O(1) average search cost, but a horrible O(n) worst-case search performance. In practice, most code will get the average case performance, but we have to be careful of worst-case performance characteristics for time-critical applications, such as embedded programming or low-latency programming.

Practical performance. Big-O notation refers to *asymptotic* performance of an algorithm or data structure, as *n* gets very large. Hence, it's not everything in terms of speeding up C++ code. There are a lot of algorithms that are slow for large *n* but are the best choices for speed when *n* is small.

A related issue is the "constant of proportionality" that's hidden in the big-O notation. The complexity order only refers to the overall characteristics of the curve, and there's a big different in practical performance between *2n* and *2,000n*, but both are O(n) in terms of complexity order. There are some famous algorithms that have great complexity characteristics, but with a constant so high that they are impractical for actual usage. One well-known example is Strassen's matrix multiplication algorithm, which is *O(n^2.7)* and theoretically better than the cubic *O(n^3)* cost of standard matrix multiplication, but is too costly to program in practice.

Throughput Versus Latency

Analyzing the speed of data structures actually has multiple metrics. There's a major difference between optimizing for these two measures:

- Latency — also known as "response time" or "round trip."
- Throughput — total work pushed through the data structure.

These are both important concepts and it's not easy to understand the difference. Don't you want both? The basic distinction is:

- Single job — measure latency.
- All jobs — measure throughput.

The simplest way to decide which metric is more important is to consider the uses:

- Interactive jobs — low latency required for user responsiveness.
- Batch jobs — high throughput more important.

But yes, it's correct that low latency and throughput are often inter-related. Coding a better software algorithm or buying a faster CPU or GPU will improve both latency and throughput.

But there are cases where latency and throughput diverge. A workload that runs a long time, but does many jobs in parallel, will have high throughput in total (good), but each job has poor latency (bad for user responsiveness). Similarly, a workload that runs jobs quickly will have a low latency (for each job), but if it has to serialize the work, then it has a poor throughput (for all jobs).

In terms of system design, you have to consider your main processor. A little known fact is that CPUs are faster than GPUs. It's easy to buy a CPU with a 5 GHz clock speed, but a top-line GPU is only about 1.6GHz clock speed, because it does so much in parallel that it would overheat at faster clock speeds.

Hence, this seems like the choice:

- CPU — low latency
- GPU — high throughput

But as everyone who hasn't been hiding under a rock the last couple years will know, GPUs are super-fast at running AI workloads with low latency for users.

The reason for this is, when you have a lot of data to crunch through (e.g., a trillion-parameter AI model), then the equation is:

Throughput + parallelization = low latency

Hence, the answer is: yes, you can have both.

Hybrid Data Structures

The best thing about data structures: you can use more than one. There are various ways to combine two data structures into a hybrid beast.

Some common examples:

- Hashed chaining — a hash table and a linked list (as in `std::unordered_map`).
- Bloom filters — a probabilistic set combining bit vectors and hash functions.
- Skip list — hybrid of a sorted array and a linked list.

My favorite combination that I've used in real-life programming is to combine a hash table with a better data structure than chaining for collision resolution.

There's nothing to say that you have to hang a linked list off your hash table. Personally, I've used binary trees instead of chained hashing, which guarantees logarithmic worst case complexity if you use balance trees like AVL trees or red-black trees. Note that `std::map` uses red-black trees, but it's linear chaining in the hashed containers.

Another flexible approach is to simply insert the same key into two different data structures, independent of each other. There are two basic strategies:

- Create one data structure (insertions), then export the data at the end, or
- Incrementally maintain two separate data structures.

The first strategy is appropriate when the data structure has a two-phase sequence, which is very common in real-world usage:

- Initialization phase — lots of insertions.
- Processing phase — searches endlessly.

In this case, it can make sense to store the data in an insert-friendly data structure while building it, and then export the data to a more search-efficient data structure once the initialization phase is finished.

For example, you could build the data structure with an unsorted array (std::vector or std::array), a hash table (`std::unordered_map`) or a red-black tree (`std::map`), all of which offer fast insertions (on average).

Once finished, you can export them to an array data structure, and/or use `std::sort` for ordering. In various applications, the contiguous storage has better cache locality properties for lots of fast searches or linear scans across all the data.

Incremental processing means you insert twice, as new data arrives. For example, when you want fast accesses using a hash table (e.g., `std::unordered_map`), but if you also want fast ordered processing, use a red-black tree (e.g., `std::map`). You can simply insert every key into two independent data structures. It's twice the insertion cost and twice the memory space, but double the fun.

Modern C++ Versions

Modern C++ has evolved into a massive and amazing language, complete with a suite of already-coded container libraries. This has made C++ programming into a much higher-level abstraction for all but the most latency-critical of tasks. The overall version updates with their key points are:

- C++98 — formalized several language features.
- C++03 — considered mostly a "bug fix" version.
- C++11 — a huge update with so many goodies (hooray!).
- C++14 — mostly a relatively minor update.
- C++17 — added parallel execution modes.
- C++20 — added coroutines for async coding.
- C++23 — several major features like concepts, reflection, etc.
- C++26 — already looking great!

Every one of these releases also had a large set of smaller language feature updates. I'm not going to list them all, because we'd be here till next Sunday, but here's some more details on the versions.

C++11. This version of C++ in 2011 was such a massive release that it's hard to summarize everything. Some of the main advances were:

- Standard containers
- Iterators and range loops
- Move semantics
- Standardizing `std::string`
- Multithreading classes (e.g., `std::thread`)
- Compile-time metaprogramming (e.g., `constexpr`)
- Template capabilities enhanced (e.g., variadic templates)
- Lambda functions
- Function objects (functors)
- Asynchronous programming (futures and promises)

C++14. This wasn't a big release, but was mostly fixing a lot of minor issues in C++11 release. However, some performance-related improvements included:

- `constexpr` capabilities increased (relaxed limitations)
- Multithreading: `std::shared_lock`, `std::shared_timed_mutex`

C++17. New features related to efficiency and also data structures and containers included:

- Parallel execution modes in `<execution>`
- Standard container parallel execution support
- `constexpr` further unleashed
- `extract()` and `merge()` container member functions.
- Fold expressions
- Parameter packs
- Structured bindings
- `std::filesystem`
- Multithreading: `std::shared_mutex`

C++20. Some of the major features in C++20 included:

- Coroutines and asynchronous programming
- `constinit` and `consteval`
- The "spaceship" <=> three-way operator (I just love it!)
- Multithreading: `<latch>`, `<semaphore>`, `std::jthread`
- Concepts ("`requires`" syntax)

C++23. New containers in C++23 included:

- `std::flat_set`
- `std::flat_map`
- `std::flat_multiset`
- `std::flat_multimap`

Other major features of C++23 include:

- Reflection
- Contracts
- Concepts (extended)
- Modules

C++26. Features coming in 2026 include:

- `std::hive`
- `std::inplace_vector`
- Safe "checked" integer class `<stdckdint>`
- Multithreading: hazard pointers, `atomic_fetch_max`, `atomic_fetch_min`
- `<rcu>` — Read-Copy-Update
- `<simd>` — CPU vectorization primitives
- `<linalg>` — linear algebra operations
- Uninitialized memory block extensions
- Hardened standard C++ library (safety)
- Debugging support
- Placeholder variables

C is not C++. Finally, if you're a C programmer, too bad, you're missing out on all of the above! Well, not all, since newer C standards use some C++ capabilities:

- `restrict` (non-aliased pointers)
- `static_assert`
- `memccpy`
- `bool/true/false`
- `alignas/alignof`
- `typeof`
- `thread_local`
- `nullptr`

Advanced Data Structures

The best thing about studying data structures is that there's a never-ending stream of new ones. Some bright spark writes a research paper, and suddenly there's a whole new idea for everyone to code up.

There's already quite a few data structures, and nobody knows them all! Some of the lesser-known data structures include:

- Bloom filters — a fast hybrid of bit vectors and hash functions.
- Bucket Arrays — like a dynamic-size memory pool, but not re-using deleted space.
- Hives — a generalization of bucket arrays and memory pools (`std::hive` in C++26).
- Union-find data structure — I've never seen this actually used except for job interviews.

Some other specialized data structures for text processing include:

- Rope data structure — organizing subsequences of text as a binary tree.
- Directed Acyclic Word Graph (DAWG) — fast lookup of words in texts.
- Automata — text sequence lookup, like an unfolded trie.
- Suffix trees — good for text analysis.

Multi-dimensional data structures include:

- Tensors — generalized multi-dimensional data used in AI.
- Vector hashing — searching in a multi-dimensional space.
- Quad-trees and KD-trees — organizing multi-dimensional spatial data.

And if you want to get even more specialized, there's always more:

- Rewind and replay data structures
- Undo and redo trees
- Scatter-gather multi-buffering (networking)
- LLM Transformer architecture (in AI engines)

Don't worry, that's not the full list. Feel free to ask AI for a "list of data structures" on your own time.

Performance Tuning Practices

How should the huge number of methods of improving program efficiency be applied to a program? The code transformations that improve the program by a significant amount should be tried first, and the smaller optimizations used only when it is important to squeeze out that last bit of extra speed in bottlenecks. Hence, I suggest the following steps for improving the efficiency of a program:

1. Time your program to get a baseline
2. Invoke the C++ compiler's built-in optimizer.
3. Profile the code and find the "hot spots."
4. Consider a better data structure or algorithm.
5. Use the major code transformations.
6. Use smaller code transformations, if speed is crucial.

The first step is to measure your code's time cost. Otherwise, how will you know whether anything made it better?

The next step is easy: turn on your optimizer. All modern C++ compilers have an option to invoke an optimizer on the code. The optimizer, although it may not always yield a major increase in speed, has one very important advantage — the programmer need not change the code. Hence, if a small improvement is desired, the optimizer can often provide it without much effort.

Software tuning. Assuming you're done with all the non-code changes to the system (e.g., hardware, networking), it's time to examine the C++. You can either start high by looking at the data structures, or start low by optimizing the busiest low-level kernels.

The choice of a better algorithm (usually with different data structures) for a program is not an easy method of program improvement. Simply identifying what would be a better algorithm is a difficult problem! And once identified, the new algorithm must be implemented by the programmer, costing precious man hours. However, this is the best method to achieve an order-of-magnitude increase in the program's performance.

The next step is to profile in detail the C++ code to determine which functions (or statements) are accounting for most of the program's time; these are the "hot spots" of the program.

This identification of costly statements is best achieved by a profiler. Identifying frequently called functions and deeply nested loops is often adequate.

Once the hot spots are identified, all efficiency measures, large and small, should be applied to this code. Any improvement to the efficiency of a statement, no matter how small, will improve the overall efficiency greatly if that statement is executed often.

Once the most costly functions and loops have been optimized, other statements can also be optimized, although the increase in speed will not be as noticeable. Some of the better code transformations to apply are parallelization, loop optimizations (vectorizations), using pass-by-reference for passing structures or objects to functions, and replacing small functions with macros or `inline` functions.

Make it right first? The speed improvement techniques in C++ can be applied either as the programmer is writing the code, or after the development and debugging of the program.

The second approach is often referred to as the "make it right first" rule. However, I believe that the first method is preferable simply because optimizing your program once it is working is a dangerous practice, and often introduces new bugs. Deferring efficiency improvement to the final development stage can also waste programmer time in improving the basic algorithms used in a program.

Using efficiency techniques during the development of the program is a much sounder method of improving efficiency.

Tuning Trade-offs

Tuning a program is not always a clear-cut gain. There are numerous other quantities that efficiency may affect:

- Space versus time-efficiency.
- Robustness of a program.
- Readability and maintainability of a program.
- Portability of a program.

There is almost always a trade-off between time and space when making programs run faster. Many of the algorithm improvements sacrifice space for extra speed, such as caching and precalculation.

An often overlooked trade-off is between program efficiency and a programmer's time in making the changes.

Changing a program for efficiency can introduce extra bugs into a program (although you could argue that it might remove bugs, too). If a piece of code has already been debugged, improving its efficiency may not be worth the risk to the robustness of a program.

Many of the program transformations used for efficiency can reduce the readability of a program. Naturally, this also makes it more difficult for a program to be maintained, and since the major cost in a program's development cycle is usually maintenance, improving efficiency may not be worth it in the long run.

Perhaps surprisingly, the efficiency of a program can usually be increased significantly without affecting portability. There are data structure efficiency techniques in this book that are generic methods that work across all modern C++ code.

Almost all of the dangers of improving efficiency are dangers for the programmer. On the other hand, the users of a program will be well pleased by extra responsiveness, and this alone makes choosing an efficient data structure a worthwhile exercise.

2. Modern C++ Containers

Standard C++ Containers

Contiguous data containers. The general-purpose containers with contiguous data are called "sequence containers" and include several that are well-known and often used:

- `std::string` — dynamic character arrays.
- `std::vector` — dynamic everything arrays.
- `std::array` — static fixed-size arrays.
- `std::bitset` — fast bit vectors.

Associative containers and sets. The associative key-value data structures are more commonly called a "map," "dictionary," or "symbol table" design pattern. Note that the "set membership" idiom is usually very similar to the associative containers, because the search is the same, but the sets don't have a payload at the end.

The main types of modern C++ containers for searching include the choice between two main types of underlying data structures:

- Red-black balanced binary trees — logarithmic complexity for search, insert and delete.
- Hash tables (with chaining) — constant-time average complexity (fast!), but linear worst-case (slow!).

The containers include these red-black tree versions:

- `std::map` — key-value lookup (dictionary idiom).
- `std::set` — key-only set membership lookup.

And these are the hash tables (my favorite data structure!):

- `std::unordered_map` — dictionary hash table for key-value pairs.
- `std::unordered_set` — hash table for set membership.

There are also variants that allow duplicates, which means multiple copies of the same key stored separately in the container. Examples include:

- `std::multiset`
- `std::multimap`
- `std::unordered_multiset`
- `std::unordered_multimap`

Linked list containers. Some of the containers to manage data in dynamically-allocated linked lists include:

- `std::list` — double-linked list
- `std::forward_list` — singly-linked list

Note that the hash table containers (e.g., `std::unordered_map`) also belong on this list because they use "chaining" for collision resolution. This approach effectively hangs linked lists off every bucket of the hash table.

Sorted "flat" containers. There are some newer containers in C++23 that are "flat" in the sense that they maintain data in sorted order. These classes include:

- `std::flat_set`
- `std::flat_map`
- `std::flat_multiset`
- `std::flat_multimap`

Special semantics containers. Some of the general-purpose containers with different semantics to searching include:

- `std::stack` — dynamic FIFO structure.
- `std::queue` — queue data structure (single-ended).
- `std::dequeue` — double-ended queue.
- `std::priority_queue` — implements the "heap" data structure.

View containers. The various types of "view" containers include:

- `std::string_view`
- `std::span`
- `std::mdspan` — multidimensional view class.

Bit-level data structures. Modern C++ supports both class libraries and utility functions for a variety of low-level bit manipulation tasks. Some examples include:

- `std::bitset`
- Bit manipulation utilities in `<bit>`

Small utility data structures. Some of the more generic types of "mini-data structures" include:

- `std::pair`
- `std::tuple`
- Ranges
- `std::optional`
- Permutations

Multithreading data structures. Parallel coding with synchronization and locking is supported in modern C++ with libraries such as:

- `std::thread`
- `std::mutex`
- `std::lock`
- `std::condition_variable`
- `std::atomic`
- `std::latch`
- `std::barrier`

And that's not the full list of primitives available in the concurrency library. Many of these multithreading capabilities have been available since C++11.

Upcoming C++26 containers. Some of the upcoming containers include:

- `std::hive` (C++26)
- `std::inplace_vector` (C++26)

What's missing? I feel ungrateful to even be writing this list, given the amazing amount of work that's gone into coding up all the above data structures in the standard C++ library. Nevertheless, some of my favorites aren't on the list yet!

Data structures that are missing from the standard C++ containers library include:

- Sorted array — indirectly supported only (e.g., `std::sort`).
- Tries — 26-way tree for storing text keys based on letters.
- B-tree — multi-way tree data structure good for disk storage.
- Graphs — depth-first search, breadth-first search, topological sort.
- Tri-state Boolean — indirectly supported via `std::optional`.

General Container Optimizations

Containers have a lot of commonalities in their performance patterns. Some general comments apply to multiple types of container classes, and making them run faster. Consider the following when implementing the usage patterns of your containers:

- Choose an initial size — avoid container auto-resizing slowdowns.
- Minimize insertions and deletions — yeah, right, those actions are why we use containers!
- Auto-resizing of containers — watch out for silent slugs!
- Remove all elements with `clear()` rather than a loop.
- Container destruction can be slow.

Choose your containers wisely:

- Prefer hash tables for fast searching (e.g., `std::unordered_map`).
- Don't use a key-value associative container if you only need a set.
- Consider whether you need sorted or unsorted scanning of all elements.
- Prefer contiguous memory containers that are available in the standard C++ library: `std::array` and `std::vector` have good cache locality.

Optimizations in relation to the types of data to use in containers:

- Choose scalar types — objects have more risks of slowdowns from calls to constructors, destructors, move operators, etc.
- Prefer integer keys — faster than `std::string` or `char*` in key-value pairs.
- Reduce the sizes of keys and values — minimizes overall container memory size and improves cache locality.

Choosing Containers

This should be a short section. It's easy: use a vector with `std::vector` or maybe a faster hash table with `std::unordered_map`, and forget the rest. Oh, maybe `std::queue` and `std::stack` if you must.

I'm only half joking, because there are two things that you often want to do quickly:

- Scanning — `std::vector` is an array with contiguous data (cache locality).
- Searching — `std::unordered_map` is a hash table with *O(1)* average complexity for search, insert, and delete.

So, that's covered most of the basic data processing requirements. You're either scanning through a set of data to work on it repeatedly. Or you're looking a key up in a dictionary, so you need search to be fast.

What about the other dynamic classes? Somebody's spent a whole lot of time on them, so surely they're useful for something?

There are situations where you might want to consider alternatives to arrays and hash tables. For example, there's `std::map`, which uses red-black trees and has logarithmic complexity for searching, inserting and deletion. But this is not as good as *O(1)* of a hash table.

The situations where a hash table might not be the best include:

- Scanning sorted data — neither the vector class `std::vector` nor the hashmap `std::unordered_map` are good at this.
- Real-time latency-critical situations — where the worst-case linear performance of searching a hash table is too risky.

But if you ask me, you can still use only arrays and hash tables in combination. Hash tables aren't great at scanning because it's a non-contiguous linked list scan. Here's a funny thought:

1. Insert repeatedly into the hash table, and then

2. Linearize the hash table in an array.

Those are your main trade-offs. Beyond that, if you're only searching a set of keys, but don't need to map the key to any other data, then use a set rather than a dictionary (officially called an "associative container"). There's a (slow) red-black binary tree in `std::set`, but fortunately there's a (faster) hash table for that called `std::unordered_set`.

Linearizing Containers

One common optimization is to perform some "preprocessing" before doing a lot of sequential processing of the data. This applies when the startup does a lot of insertions, but the main processing is mostly about scanning the data. In this case, we can switch to a linearized version of a dynamic container for faster scanning. Here's example code for linearizing a linked list:

```
// Linearize linked list to vector
std::list<int> mylist;
std::vector<int> vec;
// ....
int n = mylist.size();
vec.reserve(n);
for (auto& iter : mylist) {
    vec.push_back(iter);
}
```

This code to linearize is not particularly efficient, because it's forced to linearly scan the linked list, and then insert into the vector one-at-a-time. However, I can't see a way to do a bulk-insert out of a linked list.

As an alternative, if we no longer needed the linked list version, we could use the `merge()` member function (C++17) to transfer items from the list container to the vector. This is particularly effective because `merge()` changes the internal container pointers, but doesn't call any copy or move methods.

Changing Containers

Another idea is to convert our insertion-friendly container to one that's best for fast searches. One idea that goes from binary trees to hash tables is this:

- Handle the insertion phase with `std::map` — logarithmic insertion complexity with red-black trees.
- Convert to `std::unordered_map` (hash table) for faster searches.

Note that C++17 has the `std::merge()` member functions for splicing one container into another. There's also `extract()` to remove a single item.

Note that these routines don't move or copy any user data, but only update container internal pointers. This avoids the need for erasing data from one container and re-inserting all the data into the other container.

On the other hand, hash tables also have fast insertion with constant time on average, which is better than logarithmic (on average), so why do we need the red-black trees at all? One reason is that hash tables can degrade to linear performance in the worst case. Another reason is that the trees are good at fast processing of the data in sorted order, whereas hash tables have unsorted data.

Maybe we should do the reverse, handling insertions with our hash table, and then converting to a red-black tree for scanning in sorted order. No, not really. If we want sorted scanning of data, we'd probably do better to export the hash table to a `std::array` or `std::vector`, and then use `std::sort()` on the array or vector.

So many choices, so little time!

Useful Member Functions

Optimizing containers is about choosing the best one for your requirements, and then making the best usage of the interfaces that are provided. You don't need to write your own if you can do better with the standard containers.

Memory management of the various standard containers can be optimized in a number of ways. Firstly, you can consider things like whether the container is "full" and what "capacity" it has.

The main member functions include:

- `size()` — number of elements in the data structure.
- `capacity()` — maximum allowed with current memory.
- `reserve()` — request an amount of memory.
- `resize()` — reorganize to a bigger or smaller size.
- `clear()` — quickly remove all elements.
- `shrink_to_fit()` — request a smaller memory size.

The hash table containers, such as `std::unordered_map`, also have member functions to control the number of buckets and the resizing policies:

- `bucket_count()` — size of the hash table array.
- `bucket_size()` — length of a chain at an index.
- `load_factor()` — number of keys divided by hash table size.
- `max_load_factor()` — read or set the load factor that will trigger a rehash.
- `rehash()` — manually trigger a hash table size change and rehash (at your discretion).

You can use these member functions to track how effective the hash table is performing. This also allows taking control of the policy of when it will auto-resize and rehash into a bigger hash table with more buckets.

These C++17 member functions are useful sometimes for removing or moving multiple elements in a container:

- `extract()` — pulls a node out of the container data structure.
- `merge()` — efficiently combines two containers.

Hidden Auto-Resize Slugs

The auto-resizing capabilities of many C++ containers makes them dynamic and easy to use. However, it also hides a common efficiency that has existed since the earliest days of the STL: hidden calls to special functions. In fact, there are multiple reasons that you might want to avoid container auto-resizing:

- Slow performance — every object might get moved.
- Iterator invalidation — all objects could be at new addresses.

Auto-resizing of a container is probably something you want to avoid for performance reasons. In the worst case, it can trigger a significant delay when you're inserting into a container. The cost of an auto-resize may include:

- Memory allocation — e.g., allocating a new memory block or a hash table array.
- Move assignment calls — not for all container classes.
- Re-hashing — re-computing this for all the objects.

Note that some containers will call the move assignment operators, whereas others will resize the container without actually putting the stored objects in new locations. Here's how it works for some:

- `std::vector` — calls move assignments if the allocated block changes.
- `std::unordered_map` — zero move operator calls.

The situation with the hash table is complex, but basically it moved internal pointers around, but not your objects. The hash container doesn't need to move the objects inside the nodes on the chained linked list, so doesn't call move operators for the user's objects on those nodes.

However, it does have to do other container-internal computations:

- Re-compute the hash function for every node's key, and
- Re-attach the node to a different chained linked list.

There's no overall mechanism to control the resizing properties of all containers, but we can use various different methods. The main solutions are:

- Reserve maximum memory, or
- Manually manage the resizing process.

Initialization with maximum size. The first idea for avoiding auto-resizing is to guess the maximum number of elements we could possibly need to store in the containers, and call the `reserve()` function at the definition of the container object. For example, the code could be:

```
std::vector<int> v;
v.reserve(1000);
```

But not this, which will run 1,000 default constructors in a vector of non-scalar type:

```
v.resize(1000);  // Slow!
```

And this also would create 1,000 new objects and run their constructors:

```
std::vector<int> v(1000);  // Slug!
```

This reservation of memory is a type of "preallocation" optimization. We ensure that all memory that could be required is allocated during the initialization phase, which ensures that no memory allocations are performed later in the hotpath.

Detecting auto-resizing. Alternatively, we can detect when an insertion is likely to trigger an auto-resize. The standard container interfaces allow us to know this, before we do an insertion:

```
if (v.size() + 1 > v.capacity()) {
    // Resizing likely on insertion!
}
```

Unfortunately, there's not a lot that we can do in this situation. I mean, we could just "not insert" as a strategy, but that doesn't sound great.

Deferring container auto-resizing. Alternatively, we could detect the situation 10 insertions ahead of time, still insert the single item, and then do something later to manage the resizing, perhaps in a lower-priority thread.

```
const int n_lookahead = 10;
if (v.size() + n_lookahead > v.capacity()) {
    // Resizing will be soon!
}
```

In classes that are more dynamic than the vector class `std::vector`, such as the hashmap `std::unordered_map`, we can also defer the auto-resizing to a more convenient time.

This is only possible for the dynamic classes based on linked lists or binary trees. Note that the hash table classes actually used linked lists, because of linear chaining as the collision resolution mechanism.

We can initialize the hash table to a particular size in the constructor. The bucket count is an optional integer parameter to the constructor.

```
std::unordered_map<std::string, int> hmap(1000);
```

This only works well if we know the maximum size that we need. For more dynamic handling, we can also use the bucket management functions in the hashmap `std::unordered_map` interface to detect when the hash table is getting full, and take appropriate action.

The "load factor" is the number of elements stored in the container, divided by the hash table array size (i.e., the number of "buckets"). There's no target load factor in the standard definition, but an implementation will typically aim for a load factor around 0.5 to 1.0.

The container implementation also has a "maximum load factor" that will trigger a rehash into a bigger hash table when it's exceeded.

When the load factor is near the maximum value, this means the class will soon be increasing the hash table size, and possibly re-hashing every single element. Here's the idea coded up:

```
// Detect rehash risk
std::unordered_map<int,string> h;
int n_lookahead = 10;
float load_estimate = (h.size() +
        n_lookahead) / (float) h.bucket_count();
if (load_estimate >= h.max_load_factor()) {
    // Rehash is likely!
}
```

In the case of a hash table, we can actually ensure that it won't rehash by manipulating the maximum load factor setting. The `max_load_factor` method has overloads allowing us to both get and set the value.

Hence, a solution that defers rehashing: increase the maximum load factor setting, insert our new object, and then reset the maximum load factor:

```
float old_load_factor = h.max_load_factor();
h.max_load_factor(old_load_factor * 2.0f);  // Avoid rehash
h.insert({ x, s });  // Insert the object without fear!
h.max_load_factor(old_load_factor);  // reset
```

Note that we have to be careful, lest we introduce another hidden slug: never-resizing our hash tables. Don't defer it forever!

If you forget to ever rehash your hash table, it won't crash, but becomes a hidden slowdown. The use of chaining means that the standard hash table containers won't fail if they never get auto-resized, but they will degrade to the linear performance of a linked list for all operations.

Hand-Coding Containers

The standard containers are elegant and beautiful, but they are designed to be very general. Hence, they can sometimes be slower than you could achieve on your own. Some of the problems with standard container performance include:

- Too many allocations and deallocations with `new` and `delete`.
- Non-contiguous storage in dynamic containers (e.g., linked lists, binary trees).
- No way to change the underlying container algorithm — e.g., you can't change `std::unordered_map` to not use linked list chaining for collision resolution.
- General containers may not meet the requirements of your specific application.

In short: sometimes you can do better!

3. Move Semantics

What are Move Semantics?

Whoever invented move semantics deserves the Nobel prize. Move semantics refers to a beautiful and elegant addition to C++ class definitions added in C++11. The syntax is concise and the internal definition is semantically consistent in many ways. But the most beautiful part of move semantics: it's all about making C++ even faster!

Move semantics were about making C++ more efficient at a very high level. The issues were unnecessary calls to class constructors and copy assignment operators in a number of situations, such as:

- Temporary object creation
- Returning a class type from a function
- Overloaded operator return types

Most of the changes in C++11 that brought in move semantics were done in a way that maintained backward compatibility.

The new features available in classes included:

- Move constructors
- Move assignment operators

Whereas the new special members needed to be added to existing classes, there were also a number of automatic compiler optimizations that were enhanced to take advantage of move semantics:

- Copy elision
- Return Value Optimization (RVO)
- Named Return Value Optimization (NRVO)

Some parts of copy elision rely on move operations, whereas other cases of copy elision and RVO are actually independent of move semantics, and can be used without move special functions. The optimal choice is to use all of them together.

Copy Elision

Copy elision is an automatic C++ compiler optimization that "elides" (removes) various "copy" operations on objects. I guess "copy removal" just didn't have the same ring to it?

Copy elision works in particular situations in the C++ language. These situations include:

- Class-type `return` statements — the main situation.
- throw expressions (and handlers)
- Coroutines

The effect of copy elision is to avoid a full object copy. Instead, the place where the new object is used simply refers to the old object, which would have been copied without this optimization.

Technically, there are other unusual situations, and there are two variants of copy elision:

- Removal of copying, or
- Downgrading copying to a move operation.

You don't need to modify your code to get the benefits of copy elision. In fact, you also don't need to turn the optimizer up to eleven. Copy elision is a normal part of the C++ standard.

Return Value Optimization

Returning an object type is a special case where the old code used to be inefficient. The good news:

- Return Value Optimization (RVO) is an automatic compiler optimization.
- Nothing you need to do!

Well, actually you do need to declare a move constructor and a move assignment operator to get the full benefits, but you were doing that already, right?

Why was RVO needed? Because return statements used to cause lots of copying in objects. This could be worked-around by declaring a reference object parameter, which was returned back, instead of having an object return type.

But that's inconvenient, and there are also cases where it's not possible:

- Binary operator overloads (non-assignment) — e.g., binary "+" operator.
- Unary operator overloads (non-increment/decrement) — e.g., unary "-" operators.
- Postfix increment/decrement operators — must return the old object (not the current one).

Operator overloading was one of the most beautiful parts of C++ signatures. Shame that it used to be inefficient, but now it's not.

Any function can return a class object, rather than a pointer or reference, but the effect is that the function itself needs to declare a local object to be returned. Consider this code:

```
MyClass func(int x)
{
    MyClass ret(x);  // Create object
    return ret;   // Copy object
}
```

And then it gets copy constructed again when we call the function:

```
MyClass m = func(3);
```

Move semantics solve this problem, in combination with copy elision. This special case is called Return Value Optimization (RVO), and allows the compiler to do "one-two-skip-a-few" for object copying.

To get even more technical, this situation is called Named Return Value Optimization (NRVO), when a function returns a named local variable (i.e., "`ret`" here). The non-named version of RVO occurs when the function returns an unnamed object, such as a temporary object created as the result of a construction or operator.

Some types of RVO are implementation-specific and optional for the compiler to do. However, NRVO is "mandated" by the C++17 standard when returning a named local object variable. I guess unnamed RVO will be mandated at some time in the future, too.

RVO is very efficient in that it doesn't just convert copying to moving, but can in fact avoid the complete creation of temporary variables. The compiler can optimize the above code so that the `return` statement constructs or moves the object directly into the place where it was called from.

This means not only we avoid various copies/moves, but also the avoidance of that temporary object's constructor and destructor, too.

Moving Multiple Objects

Moving multiple objects arises as an inefficiency in C++ because there's no multi-move semantics. Some examples where you want to move multiple contiguous objects to a different memory location include:

- Move capabilities for a custom multi-object container.
- Shuffling objects along in a sorted array on insertion or deletion.
- Auto-resizing a `std::vector` container (bigger or smaller).

There's no multi-move constructors or assignment operators in the standard C++ language, so there's only single object moving methods.

In practice, you can move multiple objects in various ways, such as:

- Moving them one-by-one
- `std::move(begin, end, dest)` overload

Note that this is the `std::move` overload that does real runtime work, not the simpler version that's just a type-cast to an R-value reference.

Unfortunately, all of these ideas are calling the move constructors for every single object. This is fine for scalar types or classes with simple inlined versions, but it's still not optimal.

The workaround for your own class is simply to define a non-special member function to do fast moving, which you can call explicitly. But this doesn't solve the general problem of using your new class in a container that may need to bulk-move your objects at some point.

Generic Move Operator

Some types of objects are "relocatable" and can used an optimized move method. The basic ideas of move semantics refer to the difference between a "shallow copy" (also called a "bitwise copy" or a "byte copy") versus a "deep copy". The basic idea is this:

- Copy assignment or constructor — deep copy
- Move assignment or constructor — shallow copy

The copy constructor has to make a full copy of every data member of the other object to create a new object. The old object is unchanged.

The move constructor needs to transfer all of the data members from the old object to the new object. And then the old object needs to be "cleared" in some way, which leaves it in a "valid" state (so that its destructor doesn't crash or deallocate memory it no longer owns). Hence, why not do these steps in general as an optimization:

- Shallow move old data members to new object — bitwise copy of all bytes.
- Clear old object's data members — zero the old bytes.

This idea of a relocatable object is similar to the type trait:

`std::is_trivially_move_constructible` (C++11).

However, this isn't quite what we want, which is a way to specify that our object is relocatable. The type trait instead only detects some cases where this is true. Perhaps we could set this type trait to "true" for our own class, and the standard container classes will honor this type trait setting, but I have my doubts.

Instead, let's think about generalizing the idea to all relocatable class types. We can even code up the idea:

```
template<typename T>
T& generic_move_assignment_buggy(T& newobj, T& oldobj)
{
    memcpy(&newobj, &oldobj, sizeof(T));  // Move (bit copy)
    memset(&oldobj, 0, sizeof(T));
    return newobj;
}
```

Well, that has an aliasing bug if the new and old object are the same. So, let's fix that first:

```
template<typename T>
T& generic_move_assignment_safer(T& newobj, T& oldobj)
{
    if (&newobj != &oldobj) {  // Avoid aliasing
       memcpy(&newobj, &oldobj, sizeof(T)); // Move (bit copy)
       memset(&oldobj, 0, sizeof(T));
    }
    return newobj;
}
```

Does this idea work?

The short answer is: yes and no. Yes, this idea can be used very often, and is efficient.

Let's look at the good news first. This approach works for all these situations:

- Scalar types — moving an integer is a bitwise copy anyway.
- Simple object data members — if this move approach also works for the sub-object.
- Virtual functions — yes, the hidden "`vptr`" pointer in the old object is also moved by the bitwise copy.

However, technically the full answer is "no," because there are some problem areas when using this approach:

1. Self-referring pointer data members.

2. Virtual function problems — `vptr` is nulled in the old object.

3. Virtual destructor problems — a problematic special case.

4. External pointers into the old object (invalidated).

5. Obscure portability problems with zero byte representations.

Self-referencing data member problems. This is a problem when the object is relying internally on its own address. Self-referring internal pointers (or references) are data members inside the object that point to another part of the object. These are uncommon, and seem like bad programming style anyway.

Note that pointers pointing outside of the object are just fine. In fact, that's why this copy-and-zero approach is efficient, because we don't need to copy and reallocate any pointer data members. A bitwise copy of a pointer or reference is still pointing to the right place.

Virtual function problems. The `memset()` function has cleared every byte to zero, including any of the hidden "`vptr`" pointers to the virtual function table. When there's any virtual function in a class, then it has a hidden pointer inside the object. There are also other places that may have another `vptr`, including:

- Base class — but it usually shares a single `vptr` with the derived class.
- Multiple inheritance — requires multiple `vptr`'s in the object.
- Subobject data members — if they are of a class that has its own virtual functions.

If your code calls any of these virtual functions after it's been nulled, I'm betting against you. Nevertheless, we might be able to work around this by simply not calling any virtual functions after this move sequence.

Virtual function problems. Destructors make it a little more difficult, because it's hard to stop the C++ compiler from calling them. And every class with any other virtual function is supposed to make its destructor also virtual. Just ask Scott Meyers in the very first edition of his *Effective C++* book, which was good advice in the 1990s, and still remains so.

Hence, if our object has a virtual destructor, it may try to access the null `vptr` at some point. There's no simple workaround to "just avoid calling the destructor," since it's called implicitly.

External pointers into the object. I feel like we can live with this idea. If there are any pointers or references to refer to the old object's internal data, they are now invalidated. But that's true anyway, because the whole idea of a moved object is that it's going away.

All bytes zero portability. There's a theoretical portability problem when using `memset` to clear an object to have all its bytes equal to zero. I'm not sure it even applies anymore, as I don't know of any platform where this is a real problem. The concern is whether clearing all the individual bytes to zero will actually clear multi-byte data to its equivalent zero or null value.

In practice, these are all true:

- Characters — byte zero is always character zero.
- Integers (signed and unsigned) — all bytes zero is integer zero.
- Floating-point — all bits zero is floating-point positive zero in the IEEE 754 standard.
- Pointers — all bytes zero is the `nullptr` in any platform I know.

Hence, I'm not sure it's a real problem, but every book on C++ portability I've read has mentioned it, so now I have, too.

Workaround for fast move problems. I hate to give up on a really efficient idea, so we can point to the limitations where we need to ensure:

- "Relocatable objects" with no internal pointers or references.
- No virtual functions
- No virtual destructor

Maybe we can work around the virtual function problems by not clearing the `vptr`. Here's the idea:

```
memset((char*)&oldobj+sizeof(void*),0,sizeof(T)-sizeof(void*));
```

This assumes that there's only one `vptr`, and it's shared by the base class and derived class. Unfortunately, this idea still fails for subobjects with their own virtual functions and multiple inheritance where objects can have more than one hidden `vptr`. Anyway, it's a worthy try, and we could always ban virtual functions, which aren't that efficient anyway!

Multi-move generic function. This idea can be generalized to moving a contiguous array of multiple objects at once. The need for such a "multi-move" capability is less often required, but can arise when containers resize, and we also need it to implement sorted array insertions and deletions.

The above "generic" version only works for one object. Let's think about generalizing the idea of bytewise moves and then clearing to zero:

1. The idea still generally works on a mult-object block, because it's similar to moving one object at a time.

2. Overlapping ranges of objects are a problem, because the `memset` will wrongly clear some of the newly moved objects.

Amusingly, note that we did deal with the "overlapping blocks" problem in the single-object generic move. It's the same as the "aliasing" check!

Detecting overlapping ranges more generally is a bit more intricate to code. Here's my attempt at updating the generic move method to support multiple objects:

```
template<typename T>
T&generic_multimove_assignment(T * destarr, T* srcarr,int n)
{
    if (destarr == srcarr) {  // Same exact block
        // Nothing to do
    }
    else {
        T* enddest = destarr + n;
        T* endsrc = srcarr + n;
        if (enddest > src && enddest < endsrc) {
            // Overlapping (moving left)
            memmove(destarr, srcarr, n * sizeof(T));
            int num_overlap = enddest - src;  // Ptr arith
            memset(enddest, 0, (n - num_overlap)
                * sizeof(T));  // Clear non-overlap part
        }
        else if (endsrc > dest && endsrc < enddest) {
            // Overlapping (moving right)
            memmove(destarr, srcarr, n * sizeof(T));
            int num_overlap = endsrc - dest; // Ptr arith
            // Clear non-overlapping part
            memset(src, 0, (n - num_overlap) * sizeof(T));
        }
        else {
            // Non-overlapping blocks
            memcpy(destarr, srcarr, n * sizeof(T));
            memset(srcarr, 0, n*sizeof(T)); // Clear old
        }
    }
    return newobj;
}
```

Compiler support? Even with the restrictions to scalar and relocatable objects, and other problems listed above, this idea of just moving memory blocks around is so efficient that maybe the compiler should provide this as an option automatically? Is this the default assignment operator? No, not quite, because the default move constructor or assignment operator is a "member-wise move" of all of the data members.

This is the same as a bitwise move if all data members are trivial, but any complex classes as subobjects will need their own move constructors called.

I like this whole idea a lot more than the normal move member functions, where you have to fiddle endlessly with every single data member. Come on, the single object version is only two statements!

Hence, I'm hereby recommending to the standards committee that, like the "`=default`" specifier, there needs to be a new "`=fast`" specifier added to the C++26 language.

4. String Optimizations

Efficient Strings

The C++ `std::string` class is a beautiful and elegant class that has been well-designed and near-optimally implemented. Its main advantages include:

- High-level abstraction of string coding
- Automates management of memory buffer allocation
- Safety (e.g., no buffer overflows when appending or concatenating)
- Moderately efficient

Note that I only said efficiency was "moderate"! As classes go, it's one of the most efficient, with lots of inline member functions and implementations super-optimized by compiler engineers.

Some of the fast parts of the standard string class include:

- Small String Optimization (SSO)
- Fast to copy
- Fast move semantics

But it's still not as efficient as bypassing the string interfaces and doing low-level string processing directly with `char*` pointers and arrays.

So, here we have a perfect example of the maxim: *don't optimize prematurely!*

I'm not advocating to replace all strings with C-style string operations, but if your profiler finds a hot-spot in a C++ string operation, you can do better. Furthermore, if you're doing a very string-intensive application, such as text processing, the lowest level kernels that spin through the document probably shouldn't use the string class.

Common String Operations

If you have a string, and you want to do some work on that string, the standardized `std::string` class is often very fast. In the situations where it's not, you can also revert to optimization using old-style coding on `char*` pointers by using the `data()` or `c_str()` methods to get to the raw character array.

String length. The `length()` method is extremely fast, and always so. The comparison goes like this:

- `length()` — always blazingly fast.
- `strlen()` — slow on very long strings.

Since the string class maintains the string length incrementally as a data member, it's already been precalculated. Hence, it's an inlined access to an already-computed integer.

In comparison, C-style null-terminated strings must scan for the null byte. Hence, `strlen()` is slow on very long strings, whereas `length()` is still fast.

String Equality Comparisons. Which method is faster is unclear, depending on the implementation of `operator==`, but my money's on the string class. In particular, it can compare the lengths quickly, since it has that precomputed for both strings. The full list of ways to compare strings:

- `operator==()` — fast version.
- `compare()` — explicit method version.
- `strcmp()` — old-style string comparisons.

Case-Ignoring String Equality Comparisons. There's not a standard case-ignoring version of the `compare()` method. However, there are non-standard implementations:

- `stricmp()` — Windows (MSVS)
- `strcasecmp()` — Linux (GCC)

String Search. This is a very simple and long-standing requirement. Your options are pretty obvious:

- `find()` — simple and fast!
- `strstr()` — the old C function.

Case-Ignoring String Search. There's not a standard method function named "`ifind`" or "`stristr`", but there are ways to get there:

- `strcasestr()` — Linux
- `StrStrIA()` on Windows in `shlwapi.h`

Reverse String Search. There the string class method `rfind()` for reverse string searching. There's not really a good alternative in the older C-style libraries.

Character Search. Searching a string for the first occurrence of a string characters. The options include:

- `find(char)` — string class overload.
- `strchr()` — old-style C function.

Reverse Character Search. The options here are:

- `rfind(char)` — another class overload.
- `strrchr()` — reverse long-standing C function.

Note that the `rfind()` version is likely faster than the older function on very long strings, because it has the string length precalculated in the string object and can jump straight to the end, whereas `strrchr()` has to scan from the very beginning of the string.

Multi-Character Search. If you want to search for a prefix or suffix of a set with characters, rather than just one, then the C++ string class has what you need:

- `find_first_of()` — first character from a set.
- `find_first_not_of()` — first character not in the set.

The suffix versions are:

- `find_last_of()`
- `find_last_not_of()`

Prefix and Suffix Tests. The standard C++ methods on the string class are:

- `starts_with()` (C++20)
- `ends_with()` (C++20)

Other options include:

- `string::find()` — search forwards
- `string::rfind()` — reverse search
- `LastIndexOf` — Win32 version

There's also some other options:

- `remove_prefix()` in `string_view` (C++17)
- `remove_suffix()` in `string_view` (C++17)

You can always code your own versions:

```
inline bool STRPREFIX(const char *s, const char *prefix) {
    return strncmp(s, prefix, strlen(prefix)) == 0;
}
```

Here's a modern C++ style version:

```
inline bool string_prefix(const std::string& str,
                          const std::string& prefix)
{
    return str.find(prefix) == 0;
}
```

And here's the same idea for suffix, using the "reverse find" method:

```
inline bool string_suffix(const std::string& str,
                          const std::string& suffix)
{
    return str.rfind(suffix)
         + suffix.length() == str.length(); // Buggy!
}
```

Actually, that's a bit careless of the failure return `-1` from `rfind()`. Here's a fixed version:

```
inline bool string_suffix(const std::string& str,
                          const std::string& suffix)
{
    int offset = str.rfind(suffix);
    if (offset == -1) return false;  // not found
    return offset + suffix.length() == str.length();
}
```

Note that `rfind` is needlessly inefficient here if the string is very long and the suffix is not present. It keeps on scanning all the way to the start of the string, rather than quitting early.

There's certainly a faster way to do it, such as comparing the two lengths, using them to compute the address of where the suffix would be, and then use basic string equality testing.

Case-Ignoring Prefix and Suffix Tests. There's not much help with this in the standard libraries, so you'll have to roll your own with `strnicmp` (Windows) or `strncasecmp` (Linux).

Here's an example:

```
inline bool STRIPREFIX(const char *s, const char *prefix) {
    return strncasecmp(s, prefix, strlen(prefix)) == 0;
}
```

Here's my attempt at a fast suffix version, which mixes C++ and C coding, but won't be slow on a long string:

```
inline bool string_strisuffix(
     const std::string& str,
     const std::string& suffix)
{
    int strlen = str.length();
    int suffixlen = suffix.length();

    if (suffixlen > strlen) return false;
    int offset = strlen - suffixlen;

    const char* raw = str.c_str();
    raw += offset;

    // Find the suffix
    const char* suffixraw = suffix.c_str();
    return stricmp(raw, suffixraw) == 0;
}
```

I'm sure that you could do better!

String Class Inefficiencies

What's so bad about the standard string class? Nothing, unless you want to do a lot of processing of strings.

Here's a list of some of its problems:

1. It's a large object (e.g., 40 bytes).

2. Sequences of binary + operators.

3. Too many calls to `new` and `delete`.

4. No way to use a larger non-allocated buffer.

5. Cannot use reference counting and copy-on-write.

A lot of these concerns can be summarized: *it's too easy to use!*

Programmers tend to get comfortable with the very convenient ways that `std::string` can be used in C++ programs. In comparison, doing C-style string processing with low-level character buffers is painful! Hence, there's a tendency to forget that C++ strings are significant objects that invoke memory allocation on all but the smallest of text strings.

String Memory Layout

The `std::string` class creates objects of a reasonable size, unlike C-style `char*` The string class is quite complicated, although great compiler engineers have made it look easy. Some of the main points about string efficiency are:

- Small String Optimization (SSO) is standard (with a small internal buffer).
- Reference counting is not enabled (and nor is Copy-On-Write).

The use of SSO makes sense because otherwise even just declaring an empty string object would cause a memory allocation call to the `new` operator:

```
std::string s1;   // No memory allocation!
```

We can interrogate the string objects about their features using standard member functions such as `data()`. If the pointer to the data is inside the object itself, then we're using SSO. And if two objects created from each other (via copy constructor and/or assignment operator) have the same data buffer address, then reference counting is enabled.

Here is some code that uses standard string member calls to determine some details about the layout of a string object.

```
void print_string_details()
{
    std::string str;
    cout << "Sizeof std::string = "
         << sizeof(std::string) << " bytes" << endl;
    int bytes = str.capacity() + 1;
    int header = (sizeof(str) - bytes);
    cout << "Capacity std::string = "
         << str.capacity() << " characters ("
         << bytes << " bytes)" << endl;
    const char* datastr = str.data();
    char* saddr = reinterpret_cast<char*>(& str);
    bool is_sso = datastr >= saddr
               && datastr < saddr + sizeof(std::string);

    cout << "Short String Optimization (SSO): "
         << (is_sso ? "yes" : "no") << endl;
    cout << "Reference counting: "
         << (string_is_reference_counted(bytes*100) ?
             "yes" : "no") << endl;
    int offset = (int)(datastr - saddr);
    if (offset == 0) {
        cout << "Char buffer at start offset=0" << endl;
    }
    else if (offset + bytes == sizeof(std::string)) {
        cout << "Char buffer at end (offset = "
             << offset << ")" << endl;
    }
    else {
        cout << "Char buffer in middle (offset = "
             << offset << ")" << endl;
    }
    cout << "Header block bytes = " << header << " ("
         << offset << " before buffer, "
         << (header - offset) << " after buffer)" << endl;
}
```

And here are the results in MSVS on my Windows laptop:

```
 Sizeof std::string = 40 bytes
Capacity std::string = 15 characters (16 bytes)
Short String Optimization (SSO): yes
Reference counting: no
Character buffer in middle of string (offset = 8)
Header block bytes = 24 (8 before buffer, 16 after buffer)
```

As to the 24 header bytes here, that could be 3 pointers (8 bytes or 64-bits each), or maybe it's 1 pointer to the buffer and 2 different 64-bit integers for length and capacity. We can go exploring in the memory layout of the header block inside a string object to try to answer that question. It's non-standard coding that is implementation-specific, but plenty of people have done it!

5. Object Instrumentation

What is Object Instrumentation?

The idea of using an "instrumented object" is for both debugging and performance tuning. Performance profiling tools are more capable than this idea, but using your own hand-coded instrumentation can be valuable:

- Dummy "Tester" class to track special functions.
- Intercepted new and delete operators to track memory usage.

These methods can be more effective than running profilers interactively in some special cases. They also have the advantage that we can put it into unit tests that validate the code in every nightly build.

Tester Class

One way to instrument is to create a "Tester" class that can be used with both standard containers and any of your other templated classes or methods. We'll discuss the class here, and the full source code is in the Appendix.

The efficiency gain from using such a testing class is mostly about detecting excessive calls to constructors and destructors, or too much copying or moving of objects. These calls are too often hidden behind container operations or complex templated functions.

The concept for this tool is to create a dummy class "`Tester`" that has these features:

- Traces calls to the special functions
- Counts total calls to these functions

This can be helpful with both standard containers and our own complex classes. It's useful for multiple types of programming.

In terms of improving efficiency, the idea is useful for:

- Standard container resizes — detecting hidden large-scale object moves and copies.
- Custom containers — checking your code is running move functions rather than copying.

For more details, reference to the Appendix for full source modern C++ code of the `Tester` class.

Link-Time Interception: new and delete

The idea of a tool to test memory allocations is to shine light on the hidden calls that create and destroy allocated memory. This helps examine how containers are using allocated memory, and it's not usually pretty!

Macro interception does not work for the `new` and `delete` operators, because they don't use function-like syntax. Fortunately, you can use link-time interception of these operators instead, simply by defining your own versions. This is a standard feature of C++ that has been long supported.

Note that defining class-level versions of the `new` and `delete` operators is a well-known optimization for a class to manage its own memory allocation pool, but this isn't what we're doing here.

Instead, this link-time interception requires defining four operators at global scope:

- `new`
- `new[]`
- `delete`
- `delete[]`

There's a pitfall in implementing our intercepted versions. You cannot use the real `new` and `delete` inside these link-time wrappers. They would get intercepted again, and you'd have infinite stack recursion.

However, you can call `malloc` and `free` instead, assuming they aren't also macro-intercepted in this code.

Here's the simplest versions:

```
void * operator new(size_t n)
{
    return malloc(n);
}

void* operator new[](size_t n)
{
    return malloc(n);
}

void operator delete(void* v)
{
    free(v);
}

void operator delete[](void* v)
{
    free(v);
}
```

This method of link-time interception is an officially sanctioned standard C++ language feature since the 1990s. Be careful, though, that the return types and parameter types are precise, using `size_t` and `void*`, as you cannot use `int` or `char*`. Also, declaring these intercept functions as `inline` gets a compilation warning, and is presumably ignored by the compiler, as this requires link-time interception.

Memory Error Detection. I've always used this method for some self-testing debug wrappers. Here's an example of some ideas of some basic possible checks you can do in these intercepted operators:

```
void * operator new(size_t n)
{
    if (n == 0) {
        AUSSIE_ERROR("new operator size is zero\n");
    }
    void *v = malloc(n);
    if (v == nullptr) {
        AUSSIE_ERROR("new: allocation failure\n");
    }
    return v;
}
```

Note that you can't use `__FILE__` or `__LINE__` as these are link-time intercepts, not macros. However, you could use `std::backtrace` in C++23 instead.

Memory Performance Analysis. We can also use the idea of link-time interception to do performance improvement on memory allocation. This helps us find the slugs in both standard containers and our own code.

The modified version of these link intercepts is shown in the Appendix, with full source code. The idea is that you can examine the behavior of code by wrapping memory debug calls around it:

```
memory_reset_counters();
std::vector<int> v;
memory_report();
```

This allows investigation of the memory characteristics of any sequence of code. It's quite enlightening to investigate what sort of actions in the standard C++ libraries will trigger memory allocations.

Unit Testing of Memory Allocation. Another useful idea is to add unit tests to your build, so as to ensure that nobody's accidentally added some memory allocations to the code.

```
memory_reset_counters();
std::vector<MyClass> v;
TEST(s_new_count == 0);  // No memory allocations!
```

You know what I mean: trust but verify!

6. Timing and Benchmarking

Timing C++ Code

There are a number of reasons why it can be useful to time the execution of a program. Timing C++ code can be useful in determining which statements should be optimized whereas profilers may only indicate which functions are consuming time. Timing code can also determine the relative efficiency of various operations and give you valuable information about writing code for your machine (e.g., is shifting faster than integer multiplication?).

There are several ways to time your C++ code, some of which have existed for decades, and some that are newer and standardized.

Here's a list of some options:

- `time` shell command
- `time` C++ function
- `clock` C++ function
- `<chrono>` standard C++ class

Another way to examine the efficiency of a C++ operation is to look at the assembly code. This is examined later in the chapter.

If the full execution time for a program is all that is needed, the Linux `time` command can be used to calculate the time required by a program. There are two versions — a stand-alone utility in `/bin` and a command built into `csh`.

The command to run is usually:

```
time a.out
```

A different executable name could also be used and command line arguments can also be specified.

The Chrono Class

The `std::chrono` library is an awesome piece of work, and has many features. It's been part of the C++ standard since C++11. I'm only going to touch on a handful of basic measurements here.

Here's an example of how to measure the duration between two events:

```
auto before = std::chrono::high_resolution_clock::now();
// ... Do something
auto now = std::chrono::high_resolution_clock::now();
auto diff =
       std::chrono::duration_cast<std::chrono::microseconds>
       (now - before).count();
std::cout << "Time: " << diff << " ms" << std::endl;
```

There are other ways to do this, as the library is very flexible, with many capabilities. Reading the documentation for this class is enough to make my head spin. Someone had a lot of time to spend on time! Kudos to them.

But one way is good enough for timing our C++ code, so let's move on and leave the rest as an exercise for the reader (LOL!).

The Clock Function

If a more detailed speed analysis is needed, it is possible to add C++ self-instrumentation code to your program to monitor its own performance. The basic idea is to use the standard library functions to monitor the time before and after an action.

The advantages of the standard older-style `clock` function over the new-fangled modern `std::chrono` library:

- Measures CPU clock ticks, not wall clock time.
- Works in C, if you need it, not only C++.
- Only have to remember one function name!

The oldest useful function is the "`clock`" function which has existed since the C programming language. The `clock` function counts the number of clock ticks since the program began executing.

The "time" function, which keeps track of the real calendar time could also be used, but it is not a true indication of processor time on a large multi-user system. The clock function is correct for both single user and multi-user systems.

The clock function returns a value of type clock_t (typically long or int) that counts the number of clock ticks. This value can be converted to seconds by dividing by the constant CLOCKS_PER_SEC, also declared in <time.h>.

The basic idea of timing C++ code blocks is to call the clock function before and after an operation and examine the difference between the number of clicks. The code below examines the relative speed of shift and multiplication operations on int operands.

```
void profile_shifts()
{
    const int MILLION = 1000000;
    const int ITERATIONS = 100 * MILLION;

    int x = 1, y = 2, z = 3;

    clock_t before = clock();
    for (int i = 0; i < ITERATIONS; i++)
        x = y << z;
    printf("%d Shifts took %f seconds\n", ITERATIONS,
        (double)(clock() - before) / CLOCKS_PER_SEC);

    before = clock();
    for (int i = 0; i < ITERATIONS; i++)
        x = y * z;
    printf("%d mult took %f seconds\n", ITERATIONS,
        (double)(clock() - before) / CLOCKS_PER_SEC);
}
```

Clock Problems

clock Portability Pitfall. Note that some implementations on older Unix versions don't conform to the C++ standard and return the number of clock ticks since the *first call* to the clock function. This means that a single call to clock at the end of the program would always return zero. Hence, it is more portable to measure the number of clock ticks between two calls to clock, one at the start and one at the end. Obviously, you can also put the first call to "clock" at the start of the "main" function to avoid this rare glitch.

Note that on implementations that are correct, a call at the start of "main" may be non-zero due to the overhead of global and static C++ object instantiations (i.e., constructors for global objects), which occurs before entering main.

Clock Tick Integer Division Pitfall. Note that the standard C++ clock_t type and CLOCKS_PER_SEC constant are both integers. Hence, here's a bug:

```
clock_t diff = clock() - before;
double seconds = diff / CLOCKS_PER_SEC; // Bug!
```

The problem is that it's integer division, so it inaccurately truncates to an integer. You need a typecast to float or double on either side of the division operator.

```
clock_t diff = clock() - before;
double seconds = diff/(double)CLOCKS_PER_SEC; // Okay
```

Clock Tick Overflow Pitfall. The clock function also has a problem with wraparound on some implementations. Because of its high resolution, the number of clock ticks can quickly overflow the maximum value that can be stored by the type clock_t.

On one system the clock function will wrap around after only 36 minutes. If the program being timed runs for longer than this period, the use of clock can be misleading.

One solution is to use the "time" function rather than "clock" when executions are longer, but this usually only has resolution to the nearest second.

Benchmarking

Benchmarking is a slightly different concept to tuning, and refers to testing the efficiency of certain operations, such as low-level operators, to find a more efficient way to do an operation. For example, if you want to compare multiplication versus addition, you write a program to run these operations a few million times.

When changing a program to increase efficiency, you shouldn't assume that a certain operation is clearly faster, but you should benchmark whether the changes have noticeably increased the operation's efficiency (or even decreased it!).

Techniques for measuring program efficiency range from the stop-watch method to the use of sophisticated profiler software tools.

If no profiler is adequate, the programmer can gain timing information by adding instrumentation statements to the program, although there are many pitfalls in attempting to determine the time taken by a sequence of statements.

The measurement of the memory usage and space-efficiency of a C++ program is a slightly more difficult problem. There are several types of memory: instruction code, static memory, read-only string literals, initialization data, global/static variables, the stack, and the heap.

Measuring the memory usage of the stack and heap is somewhat difficult because of their dynamic nature. However, various tools exist to measure the different types of memory, and clever use of C++ programming constructs can also yield reasonable data.

Benchmark programs attempt to examine how quickly your machine executes certain instructions, which is more useful for examining a single multiplication operation. You mainly use benchmarking for code that's running in low-level kernels, such as CPU speedups (e.g., AVX intrinsics) or examining the advanced use of different GPU primitives.

Consider benchmarking for timing of low-level arithmetic operations on your platform. For example, how would you determine whether the integer multiplication operation `x*2` could be more efficiently replaced by `x<<1`?

How can you time these instructions? You obviously cannot just time a single operation of each with the "`clock`" function, because a single click tick contains many CPU cycles. So, you have to time thousands or even millions of such operations.

```
for (int i = 0; i < 100 * MILLION; i++) {
    x << 1;
}
```

We've already noted one problem: there's all this extra loop overhead time for the for loop conditional test (the "<" operator) and its incrementer (`i++`). The loop actually has three operations that are all about the same order-of-magnitude cost (i.e., <, ++, <<).

To get at the operator cost, we'd need to subtract out the loop overhead. We could, for example, try to time an empty loop without any loop body, and subtract that from our final cost.

Benchmarking Problems

Null effect problems. Another problem is that we cannot easily time the operators with these statements in the loop body:

```
x << 1;
x * 2;
```

The compiler is clever enough to notice that the `x<<1` and `x*2` statements have no effect in the program above (and gives "null effect" warnings). The built-in optimizer may even remove them completely. So, they won't get timed properly, or at all, even in a loop.

Add volatility? One possible solution is that maybe the compiler can be forced to avoid this optimization on the original expressions by declaring `x` as a "`volatile`" variable.

```
volatile int x = 0;
```

The `volatile` qualifier tells the compiler that all accesses to `x` are important, and that it should not remove any. The intended purpose of volatile is to allow the declaration of addresses for memory-mapped I/O, debugger-modified variables, or for variables modified by other programs (e.g., a semaphore modified by another program running concurrently). However, we can use it here to force all accesses to `x` to occur even if they appear pointless.

On the other hand, by doing this, we've lost the ability to see the "real" time cost of these operations when they're running in normal code. Most variables aren't `volatile`.

Anyway, it doesn't even work properly. Unfortunately, the computations of the `<<` and `*` operators in `x<<1` and `x*2` are not being assigned anywhere, so the computations themselves could be optimized out, even though the actual read operations on `x` must occur because `x` is `volatile`.

To force the `<<` and `*` operations to occur, it is necessary to use their result somehow, such as by assigning it to the (`volatile`) variable `x`:

```
x = x <<  1;
```

Although all of the above improvements will enhance the previous version, a far better method of improvement is to time a loop that runs a huge number of the operations,. Hence, we have to use these assignment expressions inside a loop:

```
x <<= 1;
x *= 2;
```

The code given here examines the relative speed of 10,000 shift and multiplications:

```
volatile int x = 0; // volatile prevents optimizations
clock_t before  = clock();
for (int i = 0; i < ITERATIONS; i++) x = x << 1;
printf("%d Shifts took %f seconds\n", ITERATIONS,
    (double)(clock() - before) / CLOCKS_PER_SEC);
before = clock();
for (int i = 0; i < ITERATIONS; i++) x = x * 2;
printf("%d Mult took %f seconds\n", ITERATIONS,
    (double)(clock() - before) / CLOCKS_PER_SEC);
```

Loop Unrolling

Unfortunately, the above method of measuring the speed of operations is not completely accurate, because it also includes the loop overhead (incrementing i from 1 to 10,000) and the cost of the assignment of the result to x. The loop overhead can be minimized by placing many operations within the loop, as below:

```
volatile int x = 0; // volatile to prevent optimizations
clock_t before = clock();
for (int i = 0; i < ITERATIONS; i++) {
    x = x << 1; x = x << 1; x = x << 1; x = x << 1;
    x = x << 1; x = x << 1; x = x << 1; x = x << 1;
    x = x << 1; x = x << 1; x = x << 1; x = x << 1;
    x = x << 1; x = x << 1; x = x << 1; x = x << 1;
    x = x << 1; x = x << 1; x = x << 1; x = x << 1;
}
printf("%d Shifts took %f seconds\n", ITERATIONS * 20,
    (double)(clock() - before) / CLOCKS_PER_SEC);
before = clock();
for (int i = 0; i < ITERATIONS; i++) {
    x = x * 2; x = x * 2; x = x * 2; x = x * 2;
    x = x * 2; x = x * 2; x = x * 2; x = x * 2;
    x = x * 2; x = x * 2; x = x * 2; x = x * 2;
    x = x * 2; x = x * 2; x = x * 2; x = x * 2;
    x = x * 2; x = x * 2; x = x * 2; x = x * 2;
}
printf("%d Mult took %f seconds\n", ITERATIONS * 20,
    (double)(clock() - before) / CLOCKS_PER_SEC);
```

Unfortunately, the assignment operations are needed to prevent the optimizer removing the computations, as discussed above. The only truly effective method of removing the cost of the assignment from the measurement is to time another separate loop, and subtract its time from that of the other loops, as below. This method also automatically accounts for the loop overhead cost, so the multiple operations inside each loop are not needed (and in fact would be incorrect). Our final version of the benchmark program is also made more sophisticated to output the relative magnitude of the two operations:

```
void profile_shifts4()
{
    const int MILLION = 1000000;
    const int ITERATIONS = 1000 * MILLION;
    volatile int x = 0; // volatile to prevent optimizations
    double time1, time2;

    // Time the loop overhead
    clock_t before = clock();
    for (int i = 0; i < ITERATIONS; i++)
      x = 1;
    clock_t loop_cost = clock() - before; // overhead
    double ovtime = (double)(loop_cost) / CLOCKS_PER_SEC;
    printf("%d overhead: %f seconds\n", ITERATIONS, ovtime);

    // Shifts
    before = clock();
    for (int i = 0; i < ITERATIONS; i++) {
        x = x << 1;
    }
    time1 = (double)(clock()-before-loop_cost)/CLOCKS_PER_SEC;
    printf("%d Shifts took %f seconds\n", ITERATIONS, time1);

    // Multiplications
    before = clock();
    for (int i = 0; i < ITERATIONS; i++) { x = x * 2; }
    time2 = (double)(clock()-before-loop_cost)/CLOCKS_PER_SEC;
    printf("%d Mults took %f seconds\n", ITERATIONS, time2);

    // Compare both times, and print percentage difference
    const float ACCURACY = 0.00001f; // maximum error
    if (fabs(time1 - time2) < ACCURACY) // (almost) equal?
        printf("Shift and multiplications: same time\n");
    else if (time1 < time2) {
        printf("Shifts faster by %5.2f percent\n",
                (time2 - time1) / time2 * 100.0);
    }
    else {
        printf("Multiplications faster by %5.2f percent\n",
            (time1 - time2) / time1 * 100.0);
    }
}
```

Limitations of Benchmarking

Benchmarking of C++ using these timing methods is not perfect, but I've always found it useful. There are various reasons why this type of benchmarking timing results may not be fully correct.

- Hard to account for parallelism (e.g., GPU throughput)
- Single-threaded code is not always a true representation.
- Pipelining speedups often differ in production code (even for sequential CPU code, such as AVX intrinsics).
- Loop overhead is hard to separate from the raw operations (as seen above!)
- Compiler optimizations might modify or even remove the operations being benchmarked.
- Memory cache hit rates are too high because you're running tight code accessing only a few addresses.
- Optimization levels in test mode might not match your production version.
- Debug modes might not match production (e.g., if running in a debugger).
- Pipelining by the CPU of many instructions makes it appear better than reality.
- Unrealistic non-production conditions are being tested.

Compiler optimizations. In this day and age of amazing optimization algorithms, note that on some platforms the benchmarking code above may indicate that shifts and multiplications cost exactly the same. This is most likely an indication that the compiler automatically optimizes any multiplications by powers of two into left shifts.

To get the true cost of a multiplication, the expression should be:

```
x = x * x;
```

But even this might be optimized algebraically by a compiler. The only way to know for sure what's actually being benchmarked is to examine the assembly language.

Examining Assembly Output

Another way of examining the relative costs of particular operations for a particular compiler is to examine the assembly language produced by the compiler. Many compilers have an option to produce assembly language output.

For example, under Linux the command may be:

```
gcc -S main.cpp
```

This will produce the assembly language listing for the C++ source file and store it in a new file "`main.s`" as a human-readable text file. Without the `-S` option, the assembly output would have been passed to the assembler to create the machine code executable. GCC also has a "`-masm`" option that controls the different "dialects" of assembly language (e.g., "`intel`" or "`att`"). GCC also has a verbosity control on assembly output via "`-fverbose-asm`" and "`-fno-verbose-asm`" options.

Another way to generate assembly with GCC is the "`-save-temps`" option. This option tells GCC to save the temporary assembly language file that it used for the real compilation.

Hence, this option can be used with the normal compilation mode to both build the code as normal and also output a "`.s`" assembly file. The advantage of this GCC "`-save-temps`" option over "`-S`" is that you don't need to create a separate build path for generating assembly text files.

Reviewing assembly code. Examining assembly language instructions produced for C++ operations can be very enlightening. For example, you can determine whether the compiler uses a special increment instruction for the ++ operator. Whether or not the compiler is performing various optimizations can also be examined.

Counting the number of assembly instructions is a simple measure and gives a reasonable indication of how efficiently an operation will be performed. A better method is to determine the number of cycles used by each instruction, but this requires a rather more intimate knowledge of the assembly language being used.

Many useful things can be discovered by examining assembly output. For example, does the expression `x*2` generate a multiply instruction or a shift instruction (or an addition instruction to do "`x+x`")? Does the compiler notice that `x=x+1` can be replaced by `x++`? Is the integer `%` remainder operator implemented by a sequence of instructions?

Consider the use of the relational operators (e.g., >, <) in expressions such as:

```
flag = x > y;
```

This will often produce a sequence of instructions because of the need to assign flag the value either 0 or 1. The instructions may well look like the following pseudo-assembly language:

```
LOAD 10($sp) # Load x (from stack)
CMP 12($sp) # Compare with y (on stack)
BGT $1 # Branch if greater than
LOAD 0 # Result of > operation is 0
JUMP $2
$1:
LOAD 1 # Result of > operation is 1
$2:
STORE 14($sp) # Store in flag (on stack)
```

However, review the assembler for the similar test in `if` statements, such as:

```
if (x > y) ...
```

For an `if` statement, the instructions need not be as complex, because there is no need to store the value 0 or 1 anywhere. The assembly language could be similar to branches without computations:

```
LOAD 10($sp) # Load x (from stack)
CMP 12($sp) # Compare with y (on stack)
BLE $1 # Branch if NOT greater than
... # Code for if statement body
$1:
... # Statements after if statement
```

Examining Object Files

The `objdump` command is another useful tool on Linux for analyzing binary object files. `DUMPBIN` is the comparable tool on Windows for MSVS (or you can use the `LINK` command with the "`/DUMP`" option). These tools can get to the assembly language text in reverse, by disassembling the binary instructions that are in the object file, in combination with the various symbolic information.

`objdump` can be used to examine object files in various ways and there are various useful options. The "`-d`" and "`-D`" options provide disassembly where you can examine a full dump of the assembly code in printable form (as an alternative path to the "`-S`" option). The "`-h`" option shows the headers of the object file and "`-g`" shows debugging information in the file. There are numerous other options and the "`--help`" option can be used to list all options.

The `objdump` command is part of Gnu Binutils, which also includes other useful binary file tools such as `nm`, `size`, `strip`, and `strings` utilities.

`DUMPBIN` also has various options that can be used on the DOS command-line. The default is "`/SUMMARY`" for a summary of the information about the object file. The "`/DISASM`" command shows the disassembly of the object file, which is in assembly language. Also useful is "`/SYMBOLS`" to show the symbolic names.

References

1. Linux Code, December 27, 2023, *Measuring Execution Time with Microsecond Resolution in C++*, https://thelinuxcode.com/cpp-microseconds/

7. AVX SIMD Vectorization

What are AVX Intrinsics?

Hardware-assisted vectorization is a powerful optimization to processing contiguous data structures. AVX intrinsics are SIMD parallel instructions for x86 and x64 architectures. They are actually machine opcodes supported by the x86/x64 CPU, but are wrapped in the intrinsic prototypes for easy access from a C++ program.

The main advantage of SIMD instructions is that they are CPU-supported parallel optimizations. Hence, they do not require a GPU, and can even be used on a basic Windows laptop. The main downside is that their level of parallelism is nowhere near that of a high-end GPU.

There are multiple generations of AVX intrinsics based on x86/x64 CPU instructions. Different CPUs support different features, and exactly which intrinsic calls can be used will depend on the CPU on which your C++ is running. The basic AVX types are:

- AVX — 128-bit registers = 4 x 32-bit `float` values
- AVX-2 — 256-bit registers = 8 x 32-bit `float` values
- AVX-512 — 512-bit registers = 16 x 32-bit `float` values
- AVX-10 — 512-bit registers (with speedups)

The AVX intrinsics use C++ type names to declare variables for their registers. The `float` types used to declare the registers in AVX using C++ all have a double-underscore prefix with "`__m128`" for 128-bit registers (4 `floats`), "`__m256`" for 256 bit registers (8 `floats`), and "`__m512`" for 512 bits (16 `floats`). Similarly, there are also register type names for `int` types (`__m128i`, `__m256i`, and `__m512i`), and types for "`double`" registers (`__m128d`, `__m256d`, and `__m512d`).

AVX intrinsic functions and their types are declared as ordinary function prototypes in header files. The header files that you may need to include for these intrinsics include `<intrin.h>`, `<emmintrin.h>`, and `<immintrin.h>`.

Useful AVX SIMD vector intrinsics for `float` types include:

- Initialize to all-zeros — `_mm_setzero_ps`, `_mm256_setzero_ps`
- Set all values to a single `float` — `_mm_set1_ps`, `_mm256_set1_ps`
- Set to 4 or 8 values — `_mm_set_ps`, `_mm256_set_ps`
- Load arrays to AVX registers — `_mm_loadu_ps`, `_mm256_loadu_ps`
- Store back to `float` arrays — `_mm_storeu_ps`, `_mm256_storeu_ps`
- Addition — `_mm_add_ps`, `_mm256_add_ps`
- Multiplication — `_mm_mul_ps` (SSE), `_mm256_mul_ps` (AVX-2)
- Vector dot product — `_mm_dp_ps`, `_mm256_dp_ps`
- Fused Multiply-Add (FMA — `_mm_fmadd_ps`, `_mm256_fmadd_ps`
- Horizontal addition (pairwise) — `_mm_hadd_ps`, `_mm256_hadd_ps`

Note that the names of the intrinsic functions have meaningful suffixes. The "`_ps`" suffix means "packed-single-precision" (i.e., `float`), whereas "`_pd`" suffix means "packed-double-precision" (i.e., `double`).

AVX Operations

The main SIMD instructions are called "vertical" instructions, by convention. They take one vector and a second vector (e.g., both are 128-bit), apply an operation element-wise in parallel, and put the result into a third register. In other words, they return the result of a "pair-wise" or "element-wise" operation on two vectors into a third vector.

For example, vertical addition requires two input vectors and will output a third vector with the sums. AVX-512 SIMD addition will add two 512-bit registers full of `float` values on a paired element basis (i.e., adds 16 pairs of 32-bit `float` values), yielding a third 512-bit vector with the result (16 `float` values).

Binary operations. The full list of binary AVX operations is very long. Supported AVX operations include:

- Multiplication
- Addition
- Subtraction
- Division
- Maximum
- Minimum
- Fused Multiply-Add (FMA)
- Bitwise operations

Unary operations. AVX unary intrinsics apply a particular function to all elements of an AVX register in parallel, and return the resulting register. Supported AVX unary operations include:

- Clear to zero
- Set to a constant
- Casts
- Conversions
- Popcount (POPCNT)
- Leading-zero count (LZCNT)

Mathematical Functions. Simple float-to-float mathematical functions are effectively a type of unary operator. AVX supports a variety of functions with vector hardware instructions, such as:

- Absolute value: `abs`
- Error function: `erf`
- Reciprocal
- Rounding, ceiling, floor
- Roots: `sqrt` (square root), cube root
- Inverted roots (e.g., `invsqrt`)
- Exponential: `exp`, `exp10`
- Logarithm: `log`, `log10`
- Trigonometric functions
- Hyperbolic functions
- Statistics (e.g., Cumulative Distribution Function)

AVX Horizontal Intrinsics

Horizontal operations refer to arithmetic across the values within one vector. AVX intrinsics exist to do "horizontal" operations across the same vector, such as adding horizontal elements of a vector, or finding the maximum of pairs of elements within a vector.

Horizontal SIMD instructions are typically designated with a "h" prefix (e.g., "horizontal add" is "hadd"). More specifically, the intrinsic for 128-bit horizontal add is "`_mm_hadd_ps`" and it is "`_mm256_hadd_ps`" for 256-bits.

However, do not make the mistake of assuming that these horizontal AVX intrinsics are a "reduction" of a vector down to a single float (i.e., vector-to-scalar). I mean, they really should do exactly that, but that would be too good to be true.

The horizontal intrinsic functions are still effectively "pairwise" operations for AVX and AVX-2, except the pairs are within the same vector (i.e., horizontal pairs). If you want to add all elements of a vector, or find the maximum, you will need multiple calls to these intrinsics, each time processing pairs of numbers, halving the number of elements you are examining at each iteration. Hence, for example, summing all the `float` values in a vector with AVX or AVX-2 uses a method of "shuffle-and-add" multiple times.

Thankfully, AVX-512 actually does have horizontal reductions that process all the elements in their 512 bit registers. Hence, the 512-bit horizontal add uses a different naming convention and uses the prefix of "reduce add" in the intrinsic name (e.g., `_mm512_reduce_add_ps` is a summation reduction). In other words, this reduction operates in parallel on all 16 `float` values in an AVX-512 register, and the `_mm512_reduce_add_ps` intrinsic can add up all 16 `float` values in one operation. This horizontal reduction summation is useful for vectorizing functions such as average, and could be used for vector dot products (i.e., do an AVX-512 SIMD vertical multiplication into a third vector of 16 `float` values, then a horizontal reduction to sum those 16 `float` values), although there's an even better way with FMA intrinsics.

Supported AVX horizontal operations for pairwise horizontal calculations (AVX or AVX-2) or vector-to-scalar reductions (AVX-512) include floating-point and integer versions, with various sizes, for primitives, such as:

- Addition
- Maximum
- Minimum
- Bitwise operations

Portability Checking of AVX Versions

The power of AVX support has changed over the years, with different CPUs having different capabilities, not only with AVX, AVX-2 and AVX-512, but also their sub-releases. And it's also a little unclear into the future, with reports that some of the newer Intel chips have AVX-512 disabled.

If you write some code using AVX-512 intrinsics, and compile your C++ into an executable with the AVX-512 flags on, and then it runs on a lower-capability CPU without AVX-512, what happens? Do the AVX-512 intrinsics fail, or are they simulated somehow so that they're slower but still work? Answer: kaboom on MSVS. In the MSVS IDE, if you try to call these intrinsics on a CPU that doesn't support it, you get "unhandled exception: illegal instruction."

In other words, the C++ compiler still emits the AVX-512 instruction codes, but they aren't valid, so it excepts at runtime.

Hence, the calls to AVX-512 are not emulated at run-time on lower-capability CPUs. And they aren't checked, either. That's up to you!

Dynamic test required: Firstly, you cannot use the preprocessor. You can't test `#if` or `#ifdef` for whether you've got AVX-512 in the CPU or not. You can use the preprocessor to distinguish between different platforms where you'll compile a separate binary (e.g., ARM Neon for phones or Apple M1/M2/M3 chipsets). But you cannot choose between AVX/AVX-2/AVX-512 at compile-time, unless you really plan to ship three separate binary executables. Well, you probably could do this if you really, really wanted to.

The other thing you don't really want to do is low-level testing of capabilities. You don't want to test a flag right in front of every AVX-512 intrinsic call. Otherwise, you'll lose most of the speedup benefits. Instead, you want this test done much higher up, and then have multiple versions of the higher-level kernel operations (e.g., vector add, vector multiply, vector dot product, etc.)

What this means is that you have to check in your runtime code what the CPU's capabilities are, at a very high level in your program. Hence, it is important to check your platform has the AVX support that you need, such as via the "`cpuid`" intrinsic at program startup. Then you have a dynamic flag that specifies whether you have AVX-512 or not, and you can then choose between an AVX-2 dot product or an AVX-512 dot product, or whatever else, during execution. Obviously, it gets a bit convoluted when you have to dynamically choose between versions for AVX, AVX-2 and AVX-512 (not to mention all the AVX sub-capabilities and also AVX-10 coming soon).

Example: Basic AVX SIMD Multiply

Let us do a basic element-wise SIMD multiply using AVX (version 1) and its 128-bit registers. This will do a paired vector multiply an array of 4 `float` numbers (i.e., 4 x 32-bit `float` = 128 bits). Each `float` in the resulting array is a pairwise multiplication of the elements in the two operands.

This is how SIMD instructions work, by operating on each element of the array (i.e., "pairwise" or "element-wise"). For example, a "vertical" multiply will take the 4 `float` values in one input array, and multiply each of them by the corresponding `float` in the other input array of 4 `float` numbers, and then will return a resulting output array with 4 `float` values.

For testing, let us assume with want to create an AVX function that multiplies 4 `float` values element-wise. The test code looks like:

```
float arr1[4] = { 1.0f , 2.5f , 3.14f, 0.0f };
float arr2[4] = { 1.0f , 2.5f , 3.14f, 0.0f };
float resultarr[4];
// Multiply element-wise
aussie_multiply_vectors(arr1, arr2, resultarr, 4);
```

Testing the results of the multiply as an element-wise multiply of each pair in the 4 `float` values (using my home-grown "`aussie_testf`" unit testing function that compares `float` numbers for equality):

```
aussie_testf(resultarr[0], 1.0f*1.0f); // Unit tests
aussie_testf(resultarr[1], 2.5f * 2.5f);
aussie_testf(resultarr[2], 3.14f * 3.14f);
aussie_testf(resultarr[3], 0.0f * 0.0f);
```

Here's the low-level C++ code that actually does the SIMD multiply using the "`_mm_mul_ps`" AVX intrinsic function:

```
#include <xmmintrin.h>
#include <intrin.h>

void aussie_avx_multiply_4_floats(
    float v1[4], float v2[4], float vresult[4])
{
    // Mult 4x32-bit float in 128-bit AVX registers
    __m128 r1 = _mm_loadu_ps(v1);   // Load floats
    __m128 r2 = _mm_loadu_ps(v2);
    __m128 dst = _mm_mul_ps(r1, r2); // SIMD Multiply
    _mm_storeu_ps(vresult, dst);  // Convert back
}
```

Explaining this code one line at a time:

1. The header files are included: `<xmmintrin.h>` and `<intrin.h>`.

2. The basic AVX register type is "`__m128`" which is an AVX 128-bit register (i.e., it is 128 bits in the basic AVX version, not AVX-2 or AVX-512).

3. The variables "`r1`" and "`r2`" are declared as `_mm128` registers. The names "`r1`" and "`r2`" are not important, and are just variable names.

4. The intrinsic function "`_mm_loadu_ps`" is used to convert the arrays of 4 `float` values into the 128-bit register types, and the result is "loaded" into the "`r1`" and "`r2`" 128-bit types.

5. Another 128-bit variable "`dst`" is declared to hold the results of the SIMD multiply. The name "`dst`" can be any variable name.

6. The main AVX SIMD multiply is performed by the "`_mm_mul_ps`" intrinsic function. The suffix "`s`" means "single-precision" (i.e., 32-bit `float`). This is where the rubber meets the road, and the results of the element-wise multiplication of registers "`r1`" and "`r2`" are computed and saved into the "`dst`" register. It is analogous to the basic C++ expression: `dst = r1*r2;`

7. The 128-bit result register variable "`dst`" is converted back to 32-bit `float` values (4 of them), by "storing" the 128 bits into the `float` array using the "`_mm_storeu_ps`" AVX intrinsic.

AVX Memory Alignment Issues

The above example glosses over the issue of managing "alignment" of memory addresses on byte boundaries with the "`alignas`" specifier. Some of the AVX SIMD intrinsic calls require that addresses are 16-byte aligned (i.e., this is effectively 128-bit alignment), which is not guaranteed by the C++ compiler. However, we've tolerated non-aligned addresses by using the "`_mm_storeu_ps`" intrinsic, which works with either aligned or non-aligned addresses.

Note that alignment restriction requirements of AVX are somewhat in flux. Not all AVX intrinsics require alignment, and they are "relaxed" in many cases. There have also been some bugs in compiler toleration of non-aligned addresses in C++ intrinsics. Where required, the alignment needs are:

- AVX-1 — 16-byte alignment (128-bit).
- AVX-2 — 32-byte alignment (256-bit).
- AVX-512 — 64-byte alignment (512-bit).

Since we can sort out alignment at compile-time using the C++ "`alignas`" specifier and "`aligned`" type attributes, there is no performance penalty (except in terms of space) for ensuring greater compatibility across CPU platforms and compiler versions by preferring aligned addresses.

You can create your own macros to easily test pointer addresses for alignment by checking their remainder with the % operator. These examples use bitwise-and to replace the slow remainder operator:

```
#define aussie_is_aligned_16(ptr) \
     ((((unsigned long)(ptr)) &15ul) == 0)
#define aussie_is_aligned_32(ptr) \
     ((((unsigned long)(ptr)) &31ul) == 0)
```

Although our code to multiply 4 `float` values tolerates non-alignment, it's a minor slug. The "`_mm_storeu_ps`" AVX intrinsic is slower if the addresses are not aligned, so we should fix the alignment for performance reasons. There's also another "store" intrinsic to convert from 128-bits to 4 floats called "`_mm_store_ps`" (without the "u") that runs faster, but does not tolerate non-aligned `float` arrays. Actually, "`_mm_storeu_ps`" is supposed to be equally as fast as "`_mm_store_ps`" if the address is correctly aligned, so we can still use that intrinsic if we prefer safety, but we need to change the variables to be aligned on 16-byte boundaries for a speedup.

To ensure alignment in C++, there is an "`alignas`" specifier for variable declarations. We can use "`alignas(16)`" to force C++ to create the variables with 16-byte alignment of the address where they are stored. For example, our unit test harness code could ensure 16-byte alignment of all memory addresses via:

```
alignas(16) float arr1[4] = { 1.0f, 2.5f, 3.14f, 0.0f };
alignas(16) float arr2[4] = { 1.0f, 2.5f, 3.14f, 0.0f };
alignas(16) float resultarr[4];
```

There are various non-standard alternatives to "`alignas`" in the various compilers. For example, MSVS has "`__declspec(align(16))`" with two prefix underscores, and GCC supports "`decltype(align(16))`".

The AVX code for an alignment-requiring version is not much different, with minor changes to the names of the C++ intrinsics:

```
void aussie_avx_multiply_4_floats_aligned(
    float v1[4], float v2[4], float vresult[4])
{
  // Use 128-bit registers to mult 4x32-bit floats
  __m128 r1 = _mm_loadu_ps(v1); // Load floats
  __m128 r2 = _mm_loadu_ps(v2);
  __m128 dst = _mm_mul_ps(r1, r2);   // Multiply
  _mm_store_ps(vresult, dst);  // Aligned version
}
```

Ideally we'd like to ensure that the function is only called with aligned addresses at compile-time. The first attempt is to declare the array "vresult" above as "alignas(16)" for type checking of alignment issues, but it fails for function parameters. Fortunately, there's another way using type attributes:

```
__attribute__((aligned(16)))
```

Another method is to define our own assertion that uses bitwise tests on the address instead:

```
#define is_aligned_16(ptr) \
  ((((unsigned long int)(ptr)) & 15) == 0)
```

This tests the address is a number that is a multiple of 16 using bitwise-and with 15, but this is at runtime and costs extra cycles.

AVX-2 SIMD Multiplication

Here is the AVX-2 version of pairwise SIMD multiply with intrinsics for 256-bit registers, which is eight 32-bit float variables.

```
void aussie_avx2_multiply_8_floats(
    float v1[8], float v2[8], float vresult[8])
{
    // Multiply 8x32-bit floats in 256-bit registers
    __m256 r1 = _mm256_loadu_ps(v1);   // Load floats
    __m256 r2 = _mm256_loadu_ps(v2);
    __m256 dst = _mm256_mul_ps(r1, r2);  // Multiply
    _mm256_storeu_ps(vresult, dst); // Back to 8xfloat
}
```

This is similar to the basic AVX 128-bit version, with some differences:

- The type for 256-bit registers is "__m256".
- The AVX-2 loading intrinsic is "_mm256_loadu_ps".
- The AVX-2 multiplication intrinsic is "_mm256_mul_ps".
- The conversion to float uses AVX-2 intrinsic "_mm256_storeu_ps".

AVX-512 SIMD Multiplication

Here is the 16 `float` SIMD vector multiplication using 512-bits in AVX-512.

```
void aussie_avx512_multiply_16_floats(
    float v1[16], float v2[16], float vresult[16])
{
    // Multiply 16x32-bit floats in 512-bit registers
    __m512 r1 = _mm512_loadu_ps(v1); // Load 16 floats
    __m512 r2 = _mm512_loadu_ps(v2);
    __m512 dst = _mm512_mul_ps(r1, r2); // Multiply
    _mm512_storeu_ps(vresult, dst);  // Back to floats
}
```

Note that AVX-512 will fail with an "unhandled exception: illegal instruction" (e.g., in MSVS) if AVX-512 is not supported on your CPU.

Example: AVX 128-Bit Dot Product

The AVX instruction set has a vector dot product intrinsic that wraps an x86 dot product instruction. There are versions of the dot product intrinsic for AVX (128-bit), AVX-2 (256-bit) and AVX-512 (512-bit).

For basic AVX (128 bits), this is a full vector dot product of two vectors with 4 x 32-bit `float` numbers in each vector. One oddity is that although the result is a floating-point scalar (i.e., a single 32-bit `float`), it's still stored in a 128-bit register, and must be extracted using the "`_mm_cvtss_f32`" intrinsic. The example code looks like:

```
float aussie_avx_vecdot_4_floats(
     float v1[4], float v2[4])
{
    // AVX dot product: 2 vectors of 4x32-bit floats
    __m128 r1 = _mm_loadu_ps(v1);   // Load floats
    __m128 r2 = _mm_loadu_ps(v2);
    __m128 dst = _mm_dp_ps(r1, r2, 0xf1); // Dot product
    float fret = _mm_cvtss_f32(dst);  // Extract float
    return fret;
}
```

Example: AVX-2 256-Bit Dot Product

Here is my attempt at the 256-bit version of a vector dot product of 8 `float` values using AVX-2 instructions, which seems like it should work:

```
float aussie_avx2_vecdot_8_floats_buggy(
    float v1[8], float v2[8])
{
    // AVX2 dot product: 2 vectors, 8x32-bit floats
    __m256 r1 = _mm256_loadu_ps(v1); // Load floats
    __m256 r2 = _mm256_loadu_ps(v2);
    __m256 dst = _mm256_dp_ps(r1, r2, 0xf1); // Bug!
    float fret = _mm256_cvtss_f32(dst);
    return fret;
}
```

But it doesn't! Instead of working on 8 pairs of `float` numbers, it does the vector dot product of only 4 pairs of `float` values, just like the first AVX code.

The problem wasn't related to alignment to 256-bit blocks, because I added "`alignas(32)`" to the arrays passed in. It seems that the "`_mm256_dp_ps`" intrinsic doesn't actually do 256-bit dot products, but is similar to the 128-bit "`_mm_dp_ps`" intrinsic that does only four `float` numbers (128 bits). These are based on the `VDPPS` opcode in the x86 instruction for 32-bit `float` values and there is `VDPPD` for 64-bit `double` numbers.

However, it seems that "`_mm256_dp_ps`" is not using the 256-bit version. Or maybe my code is just buggy!

References

1. Intel (2023), *Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1*, August 2023, 248966-Software-Optimization-Manual-V1-048.pdf
2. Agner Fog (2023), *Optimizing subroutines in assembly language*, https://www.agner.org/optimize/optimizing_assembly.pdf
3. Félix Cloutier (2023), *x86 and amd64 instruction reference*, https://www.felixcloutier.com/x86/
4. Microsoft (2023), *x86 intrinsics list*, https://learn.microsoft.com/en-us/cpp/intrinsics/x86-intrinsics-list
5. Intel (2023), *Intel Intrinsics Guide, Version 3.6.6*, May 10th, 2023, https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
6. Intel (2023), *Intel C++ Compiler Classic Developer Guide, version 2021.10*, https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/overview.html, PDF: https://cdrdv2.intel.com/v1/dl/getContent/781922?fileName=cpp-compiler_developer-guide-reference_2021.10-767249-781922.pdf

8. Memory Optimizations

Memory Reduction in C++

This chapter discusses the general techniques for reducing the memory requirements of a C++ program. The more general AI memory management issues of reducing the size of an AI model (e.g., model compression, quantization, pruning, etc.) or improving the memory access bottleneck in AI models (e.g., pipelining and marshaling data for a GPU) are discussed in a separate chapter.

These techniques herein aim to reduce memory usage of a program so that:

> (a) your C++ does not waste too much time on memory management activity, such as allocating too much memory, and
>
> (b) your C++ code can execute on a low-memory platform, such as an IoT embedded device.

In these days of cheap gigabytes of memory in every PC, memory reduction techniques are perhaps not as important as those for increasing speed. However, there are certainly situations when reducing space requirements is far more important than increasing the speed of a program. This section discusses a number of general techniques for reducing C++ memory requirements.

Unfortunately, reducing space requirements can also lead to loss of speed. There is often a trade-off between space efficiency and time efficiency. Every C++ program uses memory for a number of different purposes, and each of these areas needs to be attacked separately.

The memory usage of the program can be divided into the following memory sections:

- Executable instructions
- Static storage
- Stack storage
- Heap storage

The executable instructions for a program are usually stored in one contiguous block of memory. Static storage refers to the persistent memory used by global and local `static` variables, string constants and (possibly) floating-point constants. Stack storage refers to the dynamic storage of non-`static` local variables. Heap storage refers to the memory that is dynamically allocated using the `new/delete` operators and the `malloc/calloc/free` standard library functions.

The memory requirements for the executable instructions are largely independent of the other memory areas, whereas the techniques for reducing the memory required for the other three areas are often similar. However, care must be taken that applying a technique to reduce data space does not increase the amount of C++ code too greatly, thus increasing the executable size.

Compact Data Representation

Different algorithms may store data differently and thereby reduce memory requirements. There are many ways to represent data, and all have varying space usage. For example, storing all the primes less than 1000 can be done with a list of integers, a list of the incremental differences between successive primes, or a bit vector with one bit for each integer up to 1000.

Different data structures. The program should be examined to determine if a large space reduction can be achieved by changing to different data structures. For example, the program could use arrays instead of linked lists or binary trees to avoid the extra space due to pointer storage. However, this also wastes more space if the array is not full, and it is even better to use dynamic arrays, which do not waste any storage, as exactly the right amount of memory is allocated. Unfortunately, using different data structures can sometimes reduce the time-efficiency of programs.

Data compression. Compressing data can reduce space requirements when large amounts of data are involved. Hmm, let's pause for a moment and try to think of an example application with lots of data. Just jump in whenever you're ready. Billions or trillions of weights in an LLM are a good candidate.

Model compression is the theoretical term and involves either using smaller data sizes (e.g., 8-bit integer weights instead of 32-bit `float` data) or "pruning" of weights we don't need. More generally, data compression algorithms have been used in research on AI models, such as sparsity, run-length encoding and Huffman encoding.

Proceduralization. Another data representation technique is to use a function to represent data. Instead of a list of the first 1,000 primes, you could create an "`is_prime`" function that contains a big C++ `switch` statement, with all the primes as `case` values, which return true. You could also write a piece of code to create this source code automatically.

Recomputation. Another example of proceduralization, consider the storage of several images generated by a fractal algorithm: the simplest method of storing the images is to store them as large image files. But a much more space-efficient method is simply to store the values of any arguments passed to the function creating the fractal images. This way, the images can be recreated by calling the fractal generation function with the correct arguments. The only space used is a small block of values containing the arguments and the code instructions for the function. However, the recalculation of an image by this method is extremely time-inefficient.

Reducing Data Size

There are many techniques for reducing the size of program data. These techniques apply to all three types of memory — static, stack and heap storage. In some cases, a method may increase the memory storage in one area to decrease the memory usage in another, which is valid only if the total storage requirements decrease.

Use `char` arrays not `std::string`. The use of `std::string` is very convenient, but if your program has many strings, the extra storage used by the `string` objects can add up. Consider managing your own raw `char` arrays as C-style strings if you really need the space.

Avoid max-size arrays or buffers. When using an array data structure or buffer, there is temptation to be lazy and just make it bigger than it will need to be. Avoid this temptation and optimize the memory usage properly. Change an oversize array into a dynamically allocated array, if size can be determined easily at runtime.

Smart buffers or smart array classes. An alternative to using an oversize array or buffer is to create "smart" classes that manage this, by automatically extending the array or buffer if more elements are needed. The `std::vector` class is a good way to do this.

Bit vectors. These can be used where information can be reduced to a single Boolean value, such as bit flags or masks. The use of bit vectors is very compact in terms of space, and there are standard C++ libraries to implement these efficiently.

Unions. When using a lot of structures, space can be reduced by overlaying the data fields. This can only be done if the fields to be overlayed are mutually exclusive (i.e., they never have active data in them at the same time). There is a special C++ data type for this purpose: the `union`.

Linearize multi-dimensional dynamic arrays. Use the simpler and smaller size of a one-dimensional array, with the two-dimensional structure mapped onto it with index calculations. This adds more runtime cost, but saves space over multiple levels of dynamic array allocations.

Reusing space. One way to conserve memory is to reuse the space used by a variable. The `union` data type is an example of this general idea, and another is reusing variables for different purposes. For example, rather than letting several functions each have a local temporary buffer, they could all use the same global variable (although this is a very dangerous practice). As another example, if a program uses two similar arrays, examine whether the two arrays can share the same storage (possibly as a `union`). Note that I don't recommend any of these approaches: too dangerous!

Small data types: `short`, `char`. Instead of using arrays of `int`, use arrays of `short`, `char` or `unsigned char`. There is no problem with this method, provided large integer values are not being stored (e.g., larger than 127 for `char`, or larger than 255 for `unsigned char`).

This technique is also worthwhile when applied to `int` fields in objects although alignment restrictions may limit the improvement — use the `sizeof` operator to determine if the size of the object has been reduced.

Smaller local variables could also be declared as a smaller type, but this may increase the executable size due to type conversions.

Note that speed can be compromised by using smaller data types because of the type conversions that often result. Similarly, use `float` instead of `double`, where the greater precision of results is not important (e.g., an AI model).

Bit-fields in objects. When storing small integers in objects or structures, there is a way to specify exactly the number of bits required. These types are called "bit-fields" and can only be used for fields inside objects, structures or unions. You cannot declare a local variable with a bit-field type. When using bit-fields, small integers or Boolean flags are automatically packed into a `struct` or `union`. This reduces storage requirements significantly, but reduces speed because it is necessary to pack and unpack bits.

Parallel arrays versus arrays of objects or structures. Because of alignment restrictions, an object or structure may have unusable extra padding bytes. The number of padding bytes can be determined by using the `sizeof` operator, and subtracting the sizes of each individual field from the size of the object. If there are padding bytes, replacing an array of `struct` with a number of "parallel" arrays removes the need for this padding.

Packing. When dealing with large arrays of small integers, it can be more efficient to pack them together (i.e., more than one value per word), particularly when the information is binary (true or false), because only one bit per value is needed. The easiest way in C++ is to use `std::bitset`. Note that bit-fields are also packing provided by the compiler that can support more than one bit. They are also much easier to use than coding it yourself.

Packing object arrays with `#pragma pack`. Microsoft compilers support the "`#pragma pack`" preprocessor directive, which can specify the packing or alignment characteristics of an object. This can allow arrays of these objects to be packed more closely into storage.

Reordering fields in objects and structures. Because of the word alignment on some machines, the order of fields in an object or structure can change the size of the object. This only applies to objects containing different size fields. A general rule for minimizing the space is to order the fields from largest to smallest.

This heuristic may not give the best ordering — examine the size of a few different orderings using the `sizeof` operator, if space is crucial. This is a machine-dependent optimization, and may not work well on some machines.

Store integer codes instead of string names. If you're storing a string to represent some particular type or a limited set of names, or something with a finite set, then you can use an enum instead. If you need to generate the actual string name, use an array lookup or a `switch` statement to return the equivalent string constant.

For example, when dealing with AI word tokens, which are indeed fixed and finite, use the integer token code without storing the word as a string, while maintaining a single copy of the vocabulary strings (which you need anyway for the tokenizing algorithm).

Measuring Code Size and Static Storage

In general, it is more difficult to measure how much space a program is using than to measure how much time it is using. However, most environments provide some means of determining the size of instructions and static data in an executable program. If nothing else, the size of the executable file in overall bytes can be a reasonable guide.

The `size` command. Under Linux and UNIX, a useful command is the "`size`" command, which examines an executable program and reports the memory used by its instructions and its global or local `static` variables. However, it does not (and cannot) report the stack or heap usage because the amount of such memory used is dynamic, and hence cannot be found by analyzing the executable. The command is simply:

```
size a.out
```

This produces output similar to the following:

```
text data bss dec hex
20480 8192 0 28672 7000
```

The "text" value refers to the machine code instructions for the program code. Both the "data" and "bss" areas refer to global and local `static` variables.

The "data" area refers to variables which have been explicitly initialized with values (e.g., string literals or initialized global variables); the "bss" area refers to variables with implicit initialization which defaults to zero (e.g., global variables or arrays without non-zero initializers).

Function Code Sizes: If the code size is needed on a per-function basis, Linux and most other UNIX environments support the "`nm`" command. Windows also supports the `nm` command.

```
nm a.out
```

The `nm` command differs slightly across older UNIX variants, but will usually print out information including the start and end address of a function, from which the size of a function can be trivially computed.

Link Maps: Window users may be able to use a "link map" report. This allows to find out about executable size by examining the output produced by some C++ compilers at the link stage (although not all compilers will produce useful output). For example, the DOS "`link`" command with the "`/map`" option can be used when linking the object files:

```
link /map *.obj
```

Code Bloat

The size of the executable depends on the size of your C++ source code. Hence, the obvious way to reduce executable size is to go to the beach. Take a day off! Stop writing code, for goodness sake!

Remove unnecessary code. Methods to reduce the number of executable statements in your program could involve deleting non-crucial functions from the program, and eliminating any dead code or old redundant code that has been "left in" for various reasons. The use of compile-time initialization of global and `static` variables instead of assignment statements is another means for reducing code size. Turning off debug code such as assertions, debug tracing, and self-testing code can also work, but this loses the supportability benefit of shipping a fully testable version.

Compile-for-space options. Another possibility is that your compiler may support an option that causes the optimizer to focus on space reduction. This causes it to generate executable instructions that are as compact as possible, rather than being as fast as possible.

Avoid using large libraries. Pay attention to what code libraries you are linking with. Some of them are quite extensive, and may be much more than you need. Try to use the basic standard libraries as much as possible.

Template overuse. Templates are a common cause of "code bloat" and their usage should be reviewed. This is particularly true if you are using an integer-parameterized template in order to gain compile-time efficiency, or an approach such as Template Meta-Programming (TMP).

If these templates are used with a large number of constant values, many copies with the template's executable code will be generated.

Avoid large `inline` functions. Overuse of `inline` functions has the potential to create more executable code. Try to limit your use of `inline` to small functions where the overhead of the function call is significant compared to the relatively low runtime cost of the function body. Don't inline large functions that do lots of processing each call.

Inline tiny functions. Although inlining large functions can cause code bloat, the reverse is usually true for very small functions. All of those getter and setter member functions have about one instruction. The code generated from an inlined call to these tiny functions may be much smaller than the instructions to call a real function.

`constexpr` is `inline`, too. Remember that `constexpr` functions are also effectively a type of `inline` function. Again, try to limit these to relatively small functions. If a `constexpr` function is called with non-constant values, or is beyond the compiler's ability to properly inline, then multiple copies of the executable code may result.

Library linkage. The size of the executable depends not only on the C++ code, but also on the extra library functions that are linked by the linker. Although it may seem that the programmer has no control over this, there are some techniques for reducing the amount of linked code. The techniques depend largely on how "smart" your linker is — that is, whether the linker links only the functions you need.

Use DLLs for common libraries. Dynamic link libraries (DLLs) are one way to reduce the size of the executable, because the library executable code is loaded at runtime. If the DLL is a commonly used library, such as the standard C++ runtime libraries, not only will your executable smaller, but it's also efficient at runtime because it will be loaded only once into memory, even if many programs are using the code. However, making your own special code into a DLL isn't likely to offer much memory benefit at runtime, since it will simply be loaded dynamically rather than immediately at load-time. However, if it's a library that isn't needed in many invocations of your program, you can save memory by deferring loading of the library until you can determine whether it will be required.

Remove executable debug information. Executable size can be reduced by avoiding generation of the "debug" information and symbol table information. For example, with GCC don't use the "`-g`" debugging information or "`-p`" profiling instrumentation options. Linux programmers can also use the "`strip`" utility which strips symbol table information from the executable after it has been created. However, the extra symbol table information is more relevant to the amount of disk space the executable file uses than to the amount of memory it uses during runtime execution.

Reducing Static Storage

Static storage refers to the memory for global and local `static` variables, string constants and floating-point constants. All of the general size-reduction above can reduce the size of the global and `static` variables.

String literal static memory. The space requirements for string constants can be reduced if the compiler has an option to merge identical string constants (which arise quite frequently). If there is no such option, or the option does not merge string constants across object files (which is quite likely), merging string constants can be achieved by the programmer, although the method is far from elegant. For example, including this variable in a header file and using it in multiple files may create multiple copies of the string literal:

```
#define TITLE "A very long string ... "
```

Instead, a global variable can be declared to hold the string constant and the name of this char array is used instead of the string constant. In modern C++ you can use "`inline` variables" to avoid linker problems with multiple definitions.

```
inline const char TITLE[] = "A very long string ... ";
```

This change is unlikely to reduce the speed of the program, nor does it increase memory requirements even if `TITLE` is used only once (there may seem to be an extra 4 bytes to hold a pointer value pointing at where the string of characters is stored, but this is not so).

Large global variables. If there is a large global or `static` variable or array, the amount of static storage can be reduced by allocating the memory on the heap using `malloc` or the `new` operator, or by making it an automatic variable.

This is particularly useful if the object has a short "lifetime", in the sense that it is used only briefly (e.g., the array is used as temporary storage inside a function).

If the variable is used all the time, this change doesn't reduce the overall space problem, but simply moves the problem to another area.

Stack Usage

Stack storage refers to memory storage used for function calls, and includes (non-static) local variables, function parameters and system information used to keep track of function calls. Hence, the basic methods of reducing stack storage are:

- Use fewer and smaller automatic local variables.
- Use fewer and smaller function parameters.
- Use "`const&`" to pass objects by reference.
- Use global or `static` local variables instead.
- Reduce the depth of function call nesting.
- Avoid recursion (always).

Data sizes. The size of parameters and local variables can be reduced using the general methods of using smaller data types. Another method is to avoid passing large objects and to only large objects by reference (which is faster anyway). Don't use large arrays or buffers as local variables, but prefer allocated buffers or global buffers, or declare them as local static variables.

Fewer parameters. The number of parameters can be reduced by using global variables, or by packing a number of parameters into an object and passing the whole object (which is often faster, too).

Fewer local variables. The number of local variables can be reduced by re-using local variables, although this can introduce bugs if not enough care is taken. Common examples of reusable variables are scratch variables, such as temporaries or `for` loop index variables. Another method of reducing the number of local variables is to use parameters as if they were local variables (this is safe because of call-by-value). Overall, most of these suggestions are minor improvements, unless you're using very large arrays or objects as local variables.

Flatten call hierarchies. Reducing the depth of function call nesting (especially by avoiding recursion) also reduces stack space requirements. This can be achieved by using preprocessor macros or `inline` functions (but this may increase code size). You can also refactor your code to avoid too many layers of wrapping functions in interfaces.

Naturally, recursion should be avoided as much as possible by using iterative loop algorithms or tail recursion elimination.

Reducing Heap Usage

Your C++ IDE should support tools that track heap or stack usage dynamically. For example, MSVS has a "heap profiler" tool that you can enable. Linux tools such as Valgrind can be very usual to examine heap memory usage.

The amount of heap storage used depends on the size of blocks, the number of blocks and how quickly allocated blocks are deallocated. The size of blocks can be reduced using the general techniques of reducing data sizes (e.g., small data types, packing, unions).

Fewer allocation calls. The number of heap blocks affects heap usage in the obvious way (more blocks means more memory) and because of the fixed space overhead of a few hidden bytes to store information about the block (so that `delete` or `free` can de-allocate it). When small blocks are used, it can be useful to pack more than one block together to avoid this fixed overhead.

Avoid small frequent allocations. If your frequently-used class allocates a small amount of memory in a constructor and then deallocates it in the destructor, consider ways to avoid this pattern. Small amounts of data could possibly be stored in extra fields of the object.

Memory leaks waste memory. Obviously, avoiding memory leaks which are never returned to the heap is important to reducing heap memory usage. There are many tools and debug libraries available to detect leaks, and ongoing use of these tools will reduce overall heap fragmentation.

Early deallocation of memory. It's a win if you have avoided leaking the memory, but that's not the end of the story. All allocated memory should be returned to the heap as early as possible. If memory is not deallocated, unused memory (called "garbage") can accumulate and reduce the available memory.

Avoid `realloc`. Measure and manage any calls to realloc, as they can be a significant cause of heap memory fragmentation. And they're also not time-efficient, so reducing them is a win-win.

Manage `std::vector` sizes via "`reserve`". The resize operations in `std::vector` can lead to extra unnecessary allocation requests.

Linearize multi-dimensional allocated arrays. One big allocation of a linear array is much more efficient on the heap than allocating separate blocks for rows or lower-dimensions of the array.

An array of pointers into the linearized large block is only one more allocation, and has the same efficiency as having each pointer be a separate dynamically allocated subarray.

Smart buffers. Use objects that contain a limited amount of memory, which is used for the typical cases. If a longer string, or larger array is required, it needs to allocate memory and manage that process. Overall, this can massively reduce the number of allocated blocks.

Memory fragmentation. Reduce memory fragmentation by reducing both allocations and deallocations. It's also important to manage the different sizes of allocations, as varying block lengths cause more fragmentation.

Per-class allocators. In severe situations, take control of your class's dynamic objects by defining your own per-class allocators. Since the allocators knows that all block requests will be the same size, it can not only be faster, but also better at reusing memory blocks and avoiding memory fragmentation. But this method can also be a big fail if coded lazily to first allocate one huge chunk of memory. These allocators should dynamically manage their requests for more storage, using some reasonable incremental block size, rather than attempting to guess their maximum requirements up front.

References

1. Ulrich Drepper (2007), *What Every Programmer Should Know About Memory*, November 21, 2007, http://people.redhat.com/drepper/cpumemory.pdf
2. Agner Fog (2023), *Optimizing software in C++: An optimization guide for Windows, Linux, and Mac platforms*, PDF: https://www.agner.org/optimize/optimizing_cpp.pdf
3. Kurt Guntheroth (2016), *Optimized C++: Proven Techniques for Heightened Performance*, O'Reilly Media, https://www.amazon.com/dp/1491922060
4. Wikibooks (2023), *Optimizing C++/Writing efficient code/Performance improving features*, Wikibooks, https://en.wikibooks.org/wiki/Optimizing_C%2B%2B/Writing_efficient_code/Performance_improving_features
5. Bjorn Andrist, Viktor Sehr (2020), *C++ High Performance: Master the art of optimizing the functioning of your C++ code, 2nd Edition*, Packt Publishing, Dec 2020, https://www.amazon.com/dp/1839216549, Code: https://github.com/PacktPublishing/Cpp-High-Performance-Second-Edition (Chapter 7 is on memory management.)

Part II: Contiguous Data Structures

"Life moves pretty fast.
If you don't stop and look around
once in a while, you could miss it."

— *Ferris Bueller's Day Off*, 1986.

9. Arrays

Arrays are wonderfully efficient! They're the most basic data structure known to humanity. The main features to note about an array include:

- Contiguous memory storage — great for cache locality.
- Single type of data — no need to be worried about the type.

In modern C++, there are several ways to create an array data structure:

- `std::array`
- `std::vector`
- `std::inplace_vector` (C++26)

There are also some older methods of using arrays that still work in modern C++ code:

- Fixed-size array variable: `int arr[10];`
- Allocated fixed-size array: `new int[10];`
- Old-style allocated array: `malloc(sizeof(int)*10);`

Note that the size of arrays in these examples don't need to be a compile-time constant in C++. They can be a variable, where the size of the declared array is sorted out at run-time.

Array Operation Complexity

There are two main types of arrays to store objects: sorted and unsorted. Well, actually, there's other types of arrays with different semantics (e.g., stacks, queues, heaps, ring buffers), but let's just look at searching and sorting for now.

Are they fast?

Here's the 10,000 foot view:

- Unsorted arrays — very fast insertions/deletions, but slow searches (linear) and even slower to sort the data.
- Sorted arrays — faster search (logarithmic), slower insertions/deletions, and great if you need sorted data.

In more detail, here's the overall complexity analysis of the basic searching methods:

- Searching — unsorted is *O(n)* (linear search) and *O(log n)* for sorted (binary search).
- Inserting — unsorted is *O(1)* (add to the end), but *O(n)* if sorted (shuffle required).
- Deleting — this is *O(1)* if unsorted (tricky swap method!), but *O(n)* if sorted (also shuffles).
- Print unsorted — both are *O(n)* with a linear scan of the array.
- Print sorted — unsorted is *O(n log n)* because it requires an array sort, but only *O(n)* if already sorted.

And some other algebraic operations:

- Maximum/minimum — unsorted is *O(n)* because it requires a scan, but only *O(1)* if already sorted (choose first or last element).
- Top-k elements — unsorted requires an *O(n log n)* sort or at least a "partial sort"; only *O(k)* for a sorted array.
- Sum or average — both are *O(n)* because the whole array must be scanned.

Modern C++ Arrays

We're going to implement our own sorted and unsorted arrays to examine the algorithms. Standard C++ already has two types of standard unsorted arrays in `std::array` and `std::vector`. We could just wrap around those types, but I'm going to use low-level raw arrays to show the algorithms in more detail.

Sorted arrays are trickier. Note that there's no "sorted array" class in the standard C++ library. However, there are some primitives we can use to achieve sorted arrays:

- `std::sort()` — modern C++ version with a hybrid quicksort/heapsort algorithm.
- `qsort()` — old quicksort with function pointers (not recommended).

There is also some builtins for "binary search" on a sorted array:

- `std::binary_search()` — modern C++ implementation for a sorted array.
- `std::equal_range()` — binary search that handles duplicate elements in the array.
- `bsearch()` — old-style binary search with function pointers (not recommended).

If we are inserting into a sorted array, we don't need binary search exactly, because we're assuming the element isn't already in the array. Instead, we need a "binary-like search" method of finding the index location to insert a new item. In other words, we need to find the spot where the item fits in the array, but do it logarithmically, rather than using a slow linear scan.

Writing a binary-like search algorithm to find the insertion point is very fiddly coding! Fortunately, the standard C++ library has two methods that code it for us:

- `std::lower_bound()` — generalizes binary search for use with insertions.
- `std::upper_bound()` — similar version that finds the location above.

Strictly speaking, `std::binary_search()` in the C++ standard only requires a "partitioned" array rather than a "sorted" array. But for a scalar type with well-defined comparisons, this is the same thing.

Custom Array Implementation

Anyway, let's look at some of the basic operations in our custom versions of array algorithms. We'll examine the unsorted array version, but the sorted version is almost identical. Here's the overall class members:

```
template<typename T, int N>
class UnsortedArray {
private:
    T arr_[N];
    int capacity_ = N;
    int count_ = 0;
    //...
};
```

Note that "capacity_" is somewhat redundant if we're templating based on a compile-time array size, but useful if dynamically constructing our arrays at runtime.

Here are some of the basic "getter" functions:

```
int size() { return count_; }
int count() { return count_; }
int capacity() { return N; }
```

And here are some of the basic utility functions:

```
bool empty() { return count_ == 0; }
bool full() { return count_ == N; }
```

Sorted Arrays

There is no standard C++ sorted array class, so we've got to implement our own. A sorted array has a good search lookup cost, being logarithmic in the number of elements, by using the "binary search" lookup algorithm. However, it's not as good as a hash table (e.g., `std::unordered_map`), which has *O(1)* average search cost.

Insertions and deletions have a poor *O(n)* theoretical complexity, although the first phase of finding where to insert or delete is also logarithmic, using an algorithm very similar to binary search. The linear cost arises because once they find the location, they then need to shuffle elements:

- Make a gap (insertion), or
- Close a gap (deletion).

If we're using a class object for our array, such as `std::array` or `std::vector`, we can use the `insert()` method. This is doing a shuffle behind the scenes.

The main advantage of a sorted array is that it's, well, sorted, so if we want to process the array elements in sorted order, then it's already done for us. That's desirable because sorting an unsorted array is expensive with an *O(n log n)* complexity (e.g., `std::sort` typically uses a quicksort-heapsort hybrid).

If we need sorted data, there are other options in C++ containers. The `std::map` container is implemented as a balanced binary tree, called a "red-black tree," and this has logarithmic complexity for all major operations: search, insertions and deletions. However, a sorted array has good memory cost because it used contiguous storage, so it should not be underestimated!

Shuffling Array Elements

Shuffling of array elements along by one location is required for both insertion and deletion in sorted arrays. Shuffle right to create a gap for a new insertion, and shuffle left to close a gap after deletion. We can also use this idea for unsorted arrays, but there are faster tricks, as examined later in this section.

In practice, shuffling of sorted arrays is quite efficient for scalar types via a memory block copy, using the `memmove()` standard function. Note that `memmove()` is an older function that does a bytewise copy of the memory that ignores object constructors and move operators. Presumably, the standard `insert()` method is using fast byte copies for scalar types.

Here's an obscure pitfall: we cannot use various other copying methods because the shuffle involves overlapping source and destination memory blocks. There does not seem to be a version of C++ copying that permits overlaps. These functions would be incorrect and lead to undefined behavior on overlapping memory blocks, which is definitely true of any array shuffle:

- `std::memcpy` (old C-style)
- `std::copy_n`

However, we can use the overloads of the `std::move` function that work on ranges of multiple objects. These version of `std::move` have a real runtime cost, unlike the basic version, which is a compile-time type-cast that converts to a movable R-value reference (with no runtime code generated). We also need to pay attention to whether we are shuffling to the left or right, because these functions don't work for all overlapping arguments.

- `std::move` or `std::copy` — moving or copying left (i.e., close a gap for deletion).
- `std::move_backward` or `std::copy_backward` — move or copy to the right (i.e., create a gap for insertion).

Note that using `std::copy` or `std::copy_backward` functions also work here, but copying is slower than moving for non-scalar types. Hence, the `std::move` versions are more general, but still have some downsides:

- Expensive for non-scalar objects.
- Iterators are invalidated on the array.
- Invalidates any pointers or references to specific objects.

Unfortunately, the shuffle cost is terrible for complex objects that will require their move operators called for every single object. I can't say that I recommended sorted arrays for those types. Note that there are also various types of objects where we could still use a memory block move to do a "shallow move" of the objects (i.e., "relocatable objects"), rather than individually moving each element. However, this needs tricks to prevent C++ from doing its move thing, such as raw array type.

Binary-Like Sorted Array Insertion

Sorted arrays are logarithmic for searches, but not quite as good for insertions and deletions. Inserting a new element into a sorted array is a three-phase algorithm:

1. Find the location to insert,
2. Shuffle elements to the right (create a gap), and
3. Insert the new element at the location.

There are three ways to find the location in a sorted array:

1. Linear search from the front.
2. Linear search from the back.
3. Binary-like search (faster!)

Linear search over a sorted array doesn't use equality, but finds the first element the bigger than the new element. Or to go in reverse, starting at the end.

The advantage of starting at the end is that we can shuffle as we go, but it'll have terrible cache locality problems in accessing memory addresses in reverse. CPU memory prefetch algorithms usually assume a forward access order.

Anyway, neither of the linear algorithms are fast and they aren't typically used. But binary-like search for the insertion point is faster, with logarithmic complexity.

Binary-like search for insertion involves splitting up the array into two intervals, and choosing between the two based on the midpoint value. This is not exactly the same as binary search, because we're assuming that the element is not already in the array. Hence, it's like binary search, but we're looking for smaller versus bigger elements in comparison to the new element, rather than seeking equality.

Sorted Array Deletion

Deletion of an element in a sorted array is easier than insertion. There are two major phases:

1. Find the element using binary search.

2. Shuffle the elements left to close the gap.

Note that we're using real binary search, not the binary-like search for insertion, because we assume the element is present. We can't delete an element that's not in the array. Hence, we can use `std::binary_search` to find the element.

The deletion phase is a left shuffle of all the array elements. As discussed above, we can do a byte copy such as `memmmove()` or `std::move`, which both are well-defined with overlapping memory blocks.

These methods can be efficient for scalar and other trivial types where bitwise shallow copying is allowed, but may trigger a cascade of move constructors or move assignments on complex classes. Thus, sorted arrays can be potentially inefficient for non-scalars because of the hidden costs of shuffling objects.

Unsorted Arrays

Unsorted arrays are not an all-star data structure, and don't get a lot of use for basic search requirements. The main features include:

- Slow search lookups in cases like associative arrays or sets (linear scan cost).
- Fast insertions and deletions (constant cost, without any "shuffle").
- Sorting an unsorted array is costly with *O(n log n)* complexity.

Unsorted arrays are very useful if we want fast insertions and deletions, but rarely need to search or sort the array. Insertion is very fast with constant time, just by adding the new element at the end of the array. Deletions can also be implemented in constant time, but only via a trick of swapping the to-be-deleted element with the last element.

Interestingly, we can always fix our unsorted array by sorting it, and that turns out to be a decent idea. Let's examine the two ways to get a sorted array:

- Build an unsorted array, then sort it, or
- Incrementally maintain a sorted array.

The first plan costs *O(n)* in total to do all the *n* insertions (unsorted), and then costs *O(n log n)* to sort it with `std::sort`. The second plan costs *O(n)* for every one of the *n* insertions into a sorted array, and so we get to *O(n^2)* quadratic complexity for the incremental sorted array approach. In summary, our analysis suggests:

- Unsorted array (sort it later) — complexity of *O(n log n)*.
- Sorted array (incremental) — quadratic *O(n^2)* complexity.

An unsorted array might be the way to go? However, as discussed above, it's not as bad as that sounds if we have scalar types in a sorted array, because the "shuffle" is a single memory block copy.

Note that an unsorted array is actually sorted in a weird way: by the order of insertions. Hence, if you have an ordered sequence of data, they are mapped into the array sequence according to the order in which they are processed. If these objects have an associated timestamp, your supposedly unsorted array may well be sorted implicitly according to the timestamp field.

Unsorted arrays are underestimated, and can be efficient in practice. An array that is unsorted functions as a list of items, but is stored in contiguous memory, which can make scanning the array efficient in terms of cache locality (e.g., faster than linked lists in `std::list` or red-black binary trees in `std::map`).

Unsorted arrays can be useful for semantics other than basic search lookups. An array can efficiently implement a fixed-size stack, but a fixed-size queue is better implemented using a ring buffer that progresses around the array in a circular fashion. You can also put a balanced binary tree or a heap data structure into an array, but we're getting far away from a basic unsorted array in doing that.

Linear Search of Unsorted Arrays

Linear search is the worst part of unsorted arrays. There's not really a better way to search an unsorted array. Here's a simple hand-coded linear search of the array to demonstrate the algorithm that's happening:

```
int find_linear_search(const T &item)
{
    for (int i = 0; i < count_; i++) {
```

```
            if (item == arr_[i])
                return i;  // found
        }
        return -1; // not found
    }
```

The above assumes we're stored our data in a raw array type as the data member. If we choose to store the data as `std::array` or `std::vector`, we could use standard member functions to search the array, such as `find()`.

Note that if we were doing a lot of searches of an array without many insertions or deletions, here's an idea: pre-sort the array! This gives us this approach:

1. Pre-sort the array with `std::sort`

2. Use binary search on our newly sorted array.

The use of binary search reduces our searches to logarithmic complexity, which is much faster than linear search.

Template Value vs Reference Parameters

Templating based on a type has a common conundrum about how to choose between passing function parameters by reference or value. The desirable efficient that we want is usually:

- Small integer types — pass-by-value.
- Large class types — pass-by-reference.

Which signature should we use?

```
int find_linear_search(const T &item)  // Const ref
int find_linear_search(T item)  // Pass-by-value
```

Which one we desire for larger non-class types, such as `long` or `double`, is somewhat implementation-dependent and you need to benchmark it!

Unfortunately, there's no way to alter the signature of a templated function declaration according to a compile-time setting. I don't think there's even a way to do it in type traits.

However, the most common modern C++ style is to use `const` reference parameters. The reasons are:

- Large class types — `const&` references are much faster.
- Small integer types — it's not much worse.

In one sense, I'm not sure about the last point, because:

1. It's a micro-optimization, and

2. The compiler may auto-optimize it anyway.

But there is a simple solution whereby you can use `const&` reference parameters for generic types, but use pass-by-value for small integers. Template specialization to the rescue! Just define specialized versions of templated functions for the handful of small integer types:

```
int find_linear_search(int item)  // Pass-by-value
{
    // etc...
}
```

Now you only have to define about 27 more versions for every single integral and floating-point type.

Fast Linear Search

You're thinking that this doesn't exist, and the heading is an oxymoron. But there are situations where linear search on an unsorted array can be faster than the alternatives:

- Small number of elements
- Sentinel search optimization
- Low-level support for searching
- Parallel linear search

Let's examine all of these techniques in turn.

Sentinel linear search optimization. This is an optimization attributable to Knuth (1973) in the Mix programming language. The idea is to remove the conditional test in the loop (i.e., removing "`i < count`") by guaranteeing a

successful search. The trick is to add an extra element at the end of the array, which equals what we're searching for.

Note that this requires that we declare our array data member with one more item than the capacity. We always need a spare element at the end, even if the array is full to capacity.

```
T arr_[N + 1];  // Extra dummy element
```

Sentinel-based searching is only good for arrays of scalar types, because it requires making a copy of the search element, which is created at the end. The sentinel search of an unsorted array still has linear complexity, but has a lower complexity constant because each loop iteration is faster in practice.

Low-Level Search Support

Some types of CPU have explicit instructions that support scanning a memory block for a value. If we're using an array of characters or bytes, there are these candidates:

- `std::find` — on an array, vector, or string type.
- `strchr` — old-style character strings (null-terminated)
- `memchr` — low-level memory blocks of bytes.

The modern C++ code using `std::find` looks something like this:

```
bool find_standard(const T& item)
{
    auto iter = std::find(arr_, item);
    return iter != arr_.end();
}
```

The version that returns the integer index of the element in the array is:

```
int find_standard_index(const T &item)
{
    auto iter = std::find(arr_, item);
    if (iter == arr_.end()) return -1;  // Fail
    return iter - arr.begin();  // Pointer arithmetic
}
```

Note that this idea only works for arrays of contiguous memory. Pointer arithmetic doesn't work well on general iterators for dynamic memory containers.

Parallel Linear Search

There are multiple ways that we could parallelize our linear search algorithm. It just depends on our budget! Here are some options:

- CPU SIMD instructions (e.g., AVX or ARM Neon)
- Multithreading (on CPU)
- GPU hardware

SIMD instructions allow use to test multiple values in parallel on a CPU. For example, an x86 CPU from Intel or AMD allows the AVX sets of instructions, of which there are a few versions:

- AVX — 128 bits (4 x 32-bit integers).
- AVX-2 — 256 bits (8 x 32-bit integers).
- AVX-512 — 512 bits (16 x 32-bit integers).
- AVX-10 — 1024 bits (32 x 32-bit integers).

CUDA C++ GPU linear search. If we have an NVIDIA GPU, the type of parallelism is much more extensive. In fact, we can create 1024 threads, and each thread can compare only a few elements with our search key. This sounds like an almost constant-time algorithm on the GPU, but it's not quite that good. In practice, there are two phases:

1. Compare each loop element in parallel, and

2. Collate the results.

The GPU can compare all the array elements 1024 at a time. Hence, it's not constant time, but it's still linear time divided by 1024.

Also, at the end we have a synchronization problem with detecting which of the threads had a successful result of the comparison. It's not quite as bad as a "horizontal reduction" of the array (e.g., max or sum), but we have to synchronize the results in shared memory or global memory. We could use "warp shuffle" instructions that coordinate via faster GPU registers, but these only work within each warp of 32 threads, so it ends up being like a horizontal reduction over each warp.

Unsorted Array Insertions

Inserting into an unsorted array is very fast because we can just insert it at the end. This is very efficient with constant time complexity. The code example for insertion at the end:

```
void insert_end(const T & obj)
{
    if (full()) {
        throw overflow_error("Insert on full array");
    }
    else {
        arr_[count_++] = obj;
    }
}
```

There's nothing much to this code: only one statement! It's very efficient to insert at the end of an array.

Insertion at an Index

Inserting in the middle of an unsorted array seems to be an *O(n)* operation. If we needed to insert into the middle, it would seem slower because of the need to shuffle the other elements out of the way. And that would certainly be true of a sorted array, where a shuffle is needed to maintain the sorted array.

But, no, we're talking about an unsorted array here. Let's ban the shuffle.

There's a move trick to insert into the middle of an unsorted array at a given index in *O(1)* time. The trick is to note that in an unsorted array we only need to move a single element out of the way.

The idea is two short phases:

1. Move the existing element "out of the way" and to the end.

2. Insert the element at that location.

Here's a coded version of the "move away to the end" optimization. One fast way is to use `std::move`, which is like a type cast with no runtime code, and this causes move assignment on a complex object (or simple byte copying on a scalar type).

Here's the code:

```
void insert_at_offset(const T & obj, int offset)
{
    if (full()) {
        throw overflow_error("Insert on full array");
    }
    else {
        // Move to end
        arr_[count_ + 1] = std::move(arr_[offset]);
        arr_[offset] = obj;  // Insert at location
        count_++;
    }
}
```

Note that this only works for an unsorted array, not a sorted array. If we wanted a sorted order, or we need the implicit order-of-insertion in an unsorted array, then this "move to end" idea cannot be used as it will ruin the ordering.

Fast Unsorted Array Deletion

There's a trick for deleting an arbitrary element from an unsorted array that is often missed in articles. Unsorted array deletion need not be *O(n)* complexity, but can be done in *O(1)* time.

Deletion of an item from an unsorted array is a two-phase operation: find and destroy. Here's the code to find the element, which uses linear search to find its offset, and is thus *O(n)* unavoidably:

```
void delete_key(const T& item)
{
    int offset = find_linear_search(item);
    if (offset == -1) {
        throw invalid_argument("Delete not found");
    }
    else {
        delete_offset_swap(offset);
    }
}
```

The naive idea for deleting from an unsorted array that we've found here is to remove the element and "shuffle" the rest of the elements downwards (to the left) so that there's no "gap" in the array. Doing a shuffle isn't so bad for scalar types, where it's probably just one call to `memmove` behind the scenes.

But for non-scalar objects, we're moving a lot of objects. Either way, our unsorted array deletion with a shuffle has cost complexity of *O(n)* time.

There is a faster way!

First, let's get rid of the special cases: if there's only one element in the array, just erase it, and set the count to zero. And if the erase location is the end-most object, just erase it there, and decrement the count. Otherwise, if the object we want to remove is at the front or middle of the array, we do a tricky swap with the end element:

- Swap `arr[i]` with `arr[n-1]`
- Erase at `arr[n-1]`
- Decrement `n`

This swap idea has changed our unsorted array deletion from *O(n)* time to the optimal *O(1)* complexity. There's no loops anywhere!

Note that we can use `std::swap` here, and we may need to explicitly run the destructor of objects being destroyed (optional for scalar types). Here's what the code looks like:

```
void delete_offset_swap(int offset)
{
    if (empty()) {
        throw underflow_error("Delete empty array");
    }
    else if (count_ == 1) { // ***
        if (!std::is_trivially_destructible<T>::value) {
            arr_[0].~T(); // Expl destructor if needed
        }
        count_ = 0;
    }
    else {
        if (offset != count_ - 1) {
            // Swap with the end element
            std::swap(arr_[offset], arr_[count_ - 1]);
        }
        if (!std::is_trivially_destructible<T>::value) {
            arr_[count_ - 1].~T(); // Expl dest (at end)
        }
        count_--;
    }
}
```

The above code uses "type traits" from modern C++ to detect whether or not we need to explicitly run the destructor when destroying an object in the array. This is very efficient because type traits are evaluated to compile-time constants, so the compiler should optimize out the path if not needed (i.e., using "dead code elimination").

There are several options available in the type traits library, depending on exactly what types we want to support in our array:

- `std::is_trivially_destructible<T>::value`
- `std::is_destructible<T>::value`
- `std::is_scalar<T>::value`

Actually, the above code has a minor inefficiency. The giveaway is that two code sequences with `is_trivially_destructible` are similar. Can you see it? We don't need to expressly test for `count==1` (marked with stars), because the general code in the else clause also works for that special case as well.

And also, what was I thinking? There's no need to swap the element to the end, only to destroy it there. That's two hidden moves inside `std::swap`, when we only need one moved element. The better idea than swapping is to destroy the object where it is, and then move the end element down:

```
if (!std::is_trivially_destructible<T>::value) {
    arr_[offset].~T(); // Destroy in place
}
if (offset != count_ - 1) {
    // Move down the end element
    arr[offset] = std::move(arr_[count_ - 1]);
}
count_--;
```

Note that `std::move()` here is only a compile-time type cast operation. It will ensure that the move assignment operator is used on complex class types, and is also efficient for scalar and other trivial types.

Yes, moving the end element to the middle of the unsorted array changes some addresses. It will certainly invalidate iterators over the container. But so would the shuffle of elements, so we're okay there.

Note that this only works for an *unsorted* array data structure. If we did this on a sorted array, we'd ruin the sorting order in the array by moving the biggest element into the middle of the sequence. Sorted arrays need to do the shuffle.

One final point is that this fast deletion trick with swapping will break the unofficial ordering of the array by its insertion order. If we have timestamps associated with our array elements, swapping the end element into the middle will ruin that implicit ordering.

Container Deletion Pitfalls

While we're on the topic of deletions, let's look at some common mistakes with deletions from C++ containers. There are at least two major pitfalls in using the `erase()` method to remove an object from a C++ container.

Here's the basic first attempt:

```
for (auto iter : container) {
    if (want_to_delete(*iter)) {
        container.erase(iter);  // Kaboom!
    }
}
```

This will crash with a big mushroom cloud. The problem is that we've assumed the iterator stays valid, whereas the `erase()` method actually returns an updated iterator that we need to use. We can't use a range `for` loop to do this, so we have to use `begin()` and `end()` manually:

```
for (auto iter = container.begin();
        iter != container.end(); ++iter) {
    if (want_to_delete(*iter)) {
        // Use return value
        iter = container.erase(iter);
    }
}
```

This is not a crash, but still a major bug. The iterator loop skips over the next item after the erased object. There are two increments in the deletion sequence:

1. `erase()` returns the next valid iterator (after the removed object), and

2. `++iter` skips to the next element (again!).

To be correct, we need to change the idiom to avoid `++iter` if we erase anything.

```
for (auto iter = container.begin();
     iter != container.end(); /*Not here!*/ ) {
    if (want_to_delete(*iter)) {
        // Use return value
        iter = container.erase(iter);
    }
    else {
        ++iter;  // Only if not erasing!
    }
}
```

And now the code finally works!

Bypassing Interfaces

The `std::array` and `std::vector` classes are designed to allow you to get access to the stored data via the `data()` member function. It's also guaranteed that the data is stored in contiguous memory locations. Note that this is also true of `std::string`, which has a `data()` member and also `c_str()`, which returns the same address.

The `data()` method allows direct access via pointers or low-level array types to the data in the standard array or vector containers. Whether doing this is any faster is unclear, and needs benchmarking, since many of the member functions are simple pass-through inlined functions that work on the internal data anyway.

But there's certainly a few pitfalls! The address returned by the `data()` member is not guaranteed forever. There are at least two major types of bugs:

- Object is destroyed, or
- Object is moved or modified.

Since you have a pointer to an object's data, you want that object to stick around. But the object can disappear in a few ways:

- Stack object goes out of scope (triggering the destructor and unwinding the stack).
- Allocated object is deallocated by the `delete` operator.
- Object is moved by a container (e.g., an auto-resize or other "iterator invalidation" situation).

Even if the object stays around to watch your skills, there's another problem. If the underlying object is modified, then the internal address of the data that you have may become invalid. The issues are very similar to the well-known "invalidated iterator" problems with containers.

Changes to the container that probably invalidate the `data()` pointer include:

- Insertions and deletions
- `reserve()`
- `resize()`
- `shrink_to_fit()`

Any of these members that modify the object are allowed to move the data. For example, they might allocate a different memory block, and move the whole array away from your pointer. But there are a huge number of other situations under which an iterator into a container may become invalidated, which presumably also invalidates an old address returned from the `data()` member function.

Watch out!

Extensions

1. Benchmark the unsorted array implementation above using a raw array type versus an alternative approach of using a `std::vector` member object to store the data.
2. Benchmark the sorted array implementation with a raw array versus using `std::vector` as the internal data array, especially to see if our hand-coded binary search is fast or not.
3. Explore the use of "shallow copying" on sorted arrays containing "relocatable objects" in the shuffle needed for insertions and deletions in a sorted array data structure.
4. Explore the efficiency of calls to move constructors in a "shuffle" for a sorted array implemented using `std::vector` or `std::array`.
5. Implement the binary-like search algorithm to find the insertion location in a sorted array. (Note that deletion is just the normal binary search to find the element.)
6. Benchmark inserting into an unsorted array and then sorting using `std::sort`, because incrementally maintaining a sorted array. Do the results differ for a scalar integer type versus arrays of an object like `std::string` (which has move operators)?
7. Implement a hybrid binary-linear search where the binary search reverts to linear search once the interval is small enough.

8. Implement an AVX SIMD version of linear search over integers that tests a number of integers in the array at once.
9. Implement a "cache-aware" binary search that chooses the middle index at the start of a cache line (where possible), and tests all values in that cache line immediately using an unrolled linear search.
10. Implement a binary search that is both cache-aware and uses AVX SIMD instructions to test all elements in the same cache line more efficiently.

10. Pointer Arithmetic

What is Pointer Arithmetic?

Pointer arithmetic is a tricky C++ optimization that allows us to do faster arithmetic on arrays and other contiguous data. Some of the key points are:

- Pointer arithmetic is fast with low-level raw array types.
- References are like pointers that cannot be nulled.
- Iterators are similar to pointers in some cases.

When do you use them in modern C++? Here are some situations:

- Contiguous containers like `std::vector` can also be optimized with pointer arithmetic.
- Iterator arithmetic on contiguous C++ containers is similar to pointer arithmetic (also fast).
- Text processing with low-level character arrays and pointer arithmetic can be faster than `std::string`.

One optimization is that pointer arithmetic can be used to get rid of incremented variables in loops. Instead, a pointer can be incremented each loop iteration. This changes an array access "`arr[i]`" into a pointer access "`*ptr`" and is usually faster.

Arrays and pointers are besties in C++ and there's a way that mathematical arithmetic operators can work on both. Consider the declarations:

```
int arr[10];  // Array
int *ptr;     // Pointer
```

To start with, we can cross over between pointers and arrays in both directions. You can set the pointer to point at the array, and C++ allows us to use index notation on a pointer:

```
ptr = arr;  // Use array like a pointer
x = ptr[3]; // Use pointer like an array
```

Here, x will get the value of `arr[3]` via `ptr[3]`. The pointer and array are equivalent. Note that the "`&`" address-of operator can be optionally used here. We could have written "`ptr=&arr`" to copy the address, but it's optional.

C++ allows array index accesses on pointers with "`ptr[3]`" as above. We can also do this using "pointer arithmetic" with the "+" operator and the "`*`" pointer de-reference operator:

```
x = *(ptr + 3);  // Same as ptr[3]
```

The expression "`ptr+3`" is the address of the third element in the array (i.e., `&arr[3]`), and the "`*`" dereference operator gets the value pointed to by the pointer (i.e., `arr[3]`).

Why does this work? If `ptr` is pointing to the start of an integer, shouldn't "`ptr+3`" be a weird address in the middle of an integer?

No, because C++ does "pointer arithmetic" on pointers. Because "`ptr`" is an "`int*`" type pointer, the compiler knows to work on "`int`" data. With pointer arithmetic, the "+" operation adds a multiple of the bytes of the size of `int` types. So "`ptr+1`" is not the address 1 more than `ptr`, it's actually 4 more than `ptr` for a 4-byte `int` (assuming 32-bit integers). And "`ptr+3`" is actually the address "`ptr+12`" in terms of bytes.

Details of Pointer Arithmetic

Which Operators Do Pointer Arithmetic? Pointer arithmetic works with a number of arithmetic operators:

- Increment — `ptr++` adds `1*size` bytes to `ptr`.
- Decrement — `ptr--` subtracts `1*size` bytes from `ptr`.
- Addition — `ptr + n` adds `n*size` bytes.
- Subtraction — `ptr-n` subtracts `n*size` bytes.
- Assign-Add — `ptr += n` adds `n*size` bytes to `ptr`.
- Assign-Subtract — `ptr -=n` subtracts `n*size` bytes from `ptr`.

Note that there's no pointer arithmetic multiplication or division. Actually, I was told that C++37 was going to have a C++ pointer multiplication operator that scanned down an array doing paired multiplications, adding them up as it went, and all in one CPU cycle, but then someone woke me up.

Pointer Comparisons: You can also compare pointers, which isn't really doing any special pointer arithmetic, but works as normal comparisons on their addresses:

- Equality tests — `ptr1 == ptr2` or `ptr1 != ptr2`
- Less than — `ptr1 < ptr2` or `ptr1 <= ptr2`
- Greater than — `ptr2 > ptr2` or `ptr1 >= ptr2`

Segmented Memory Model Pointer Comparisons: Note that there's a weird portability gotcha in relative pointer comparisons (i.e., less-than or greater-than). They're only guaranteed to work in very limited scenarios by the C++ standard, such as when the pointers are both operating over the same array data. Programmers tend to think of the address space as one huge contiguous range with addresses, where you can compare all of the pointers in the program against each other, and make some coding assumptions based on that. However, there are architectures where pointer addressing is more complicated, such as where pointers are a multi-part number pointing into different memory banks with a more convoluted segmented addressing scheme. For example, pointers to allocated heap memory might be separate from the pointers to global static data, and not easily comparable.

Pointer Differences: You can subtract two pointers using the normal "`-`" subtraction operator. The result is not the number of bytes between them, but the number of objects. Hence, the two pointers must be of the same type (i.e., pointing to the same type of object). Consider this code:

```
int arr[10];
int *ptr1 = &arr[1];
int *ptr2 = &arr[2];
int diff = ptr2 - ptr1;
```

The value of "`diff`" should be `1` in C++ (rather than 4 bytes), because the two pointers are one element apart (i.e., 1 integer difference). Note that "`diff`" is a signed integer here, and the value of subtracting two pointers can be negative (e.g., "`ptr1-ptr2`" above would be "`-1`" instead). Technically, the official type of the difference between two pointers is "`std::ptrdiff_t`" which is an implementation-specific integral signed type that you can use if you are the sort of person who alphabetizes their pantry.

Adding Pointers Fails: Note that adding two pointers with "`ptr1 + ptr2`" is meaningless and usually a compilation error. Also invalid are weird things like the "`+=`" or "`-=`" operators on two pointers. Even though "`-`" is valid on two pointers, "`ptr1-=ptr2`" fails to compile because the result of "`ptr1-ptr2`" is a non-pointer type.

Char Star Pointers (Size 1 Byte): Note that if you want to avoid pointer arithmetic, and see the actual numeric value of addresses, you can use a "`char*`" type pointer (or "`unsigned char*`"). Since `sizeof(char)` is 1 byte, then all of the pointer arithmetic will just add the expected number of bytes (e.g., `ptr++` on a `char*` pointer adds 1 to the address). If you want to know the actual number of bytes between two pointers, then cast them to "`char*`" type before doing the pointer subtraction.

```
int diffbytes = (char*)ptr2 - (char*)ptr1;
```

Stride of an Array. A useful piece of terminology when processing lots of AI model data in memory is the "stride" of an array. This means the number of bytes between adjacent array elements. We can try to compute it as follows:

```
int arr[100];
int stride = &arr[2] - &arr[1];  // Wrong
```

Nope, that's a fail. This isn't the stride, because it did pointer arithmetic. The addresses of array elements are really pointers, so the stride variable above is always 1 (the adjacent elements are 1 apart in pointer arithmetic). We need to convert to `char` pointers to get the stride in bytes.

```
int arr[100];
int stride = (char*)&arr[2] - (char*)&arr[1];
```

Can't we just use `sizeof` to get the stride? Isn't the stride above going to equal `4`, which is `sizeof(int)`? Yes, in the example above the use of `sizeof` is correct, but no, that is not true in general. The stride will often equal the element size, but may be larger. For a simply packed array of integers or other simple types, the stride is almost certainly the size of the array element type. But this is not always true, such as if it's an array of a larger object with an awkward size that requires padding bytes for address alignment considerations.

Loop Unrolling Stride. The term "stride" also has a secondary meaning when talking about array processing with loop unrolling. The stride of an unrolled loop is how long of a segment is being processed in each section of loop unrolling code. For example, if a loop is unrolled with AVX-2's 256-bit registers (equals 8 32-bit `floats`), then the stride when discussed in the literature is either 8 `floats` or 8x4=32 bytes.

Void Pointer Arithmetic Fails: Note also that pointer arithmetic on a generic "`void*`" pointer should be a compile error, because it points to unknown size objects. Some C++ compilers will allow pointer arithmetic on void pointers with a warning, and pretend it's a "`char*`" pointer instead.

Finally, I don't think you can increment a "function pointer" in valid pointer arithmetic, but you're welcome to try.

Pointers and Arrays

There is a close relationship in C++ between arrays and pointers. Array names are, in many ways, just pointers to the first element in the array. The array indexing operation is identical to a pointer expression involving address arithmetic. The following algebraic identities hold:

```
array[exp] == *(array + exp)
&array[exp] == array + exp
```

These relationships have a number of consequences. First, the commutativity of + means that `exp1[exp2]` is equivalent to `exp2[exp1]`, which leads to weird syntax tricks like "`n[ptr]`" instead of "`ptr[n]`".

Another consequence is that, in many situations, pointers can be used rather than arrays. For example, it is legal to apply the array indexing operator (i.e., square brackets) to a pointer. For example:

```
x = ptr[3];
```

Just like `arr[3]`, this sets `x` to equal the third element away from `ptr`, where `ptr` is pointing into an array.

Array Function Parameters: The array and function relationship is complicated when an array is a function parameter. When an array is passed to a function, the address of the first element of the array is passed. An array formal parameter is implemented as a pointer variable (i.e., a pointer pointing to the start of the array).

This explains why arrays are passed by reference, not by value. A local copy of the array is not used inside the function. Instead, a pointer to the original array is used. Hence, any change to an element of the local array variable is actually changing the original array (i.e., pass-by-reference instead of pass-by-value).

The differences between pointers and arrays are few. The main one is that an array name is not a variable, whereas a pointer is. Hence, an ordinary array name declared as a local variable cannot be assigned to, or incremented, whereas a local pointer variable can be. An array is similar to a constant pointer (e.g., `int *const ptr`). Note that this is untrue when the array is a function parameter, when it can be incremented or modified.

There are also the differences between pointers and array types in relation to their initializations. Consider the two initializations:

```
char *p = "hello";
char arr[100] = "hello";
```

For the pointer `p`, the string literal `"hello"` is stored in separate memory. Only the required number of bytes are allocated (six, because of the extra character zero added by the compiler to terminate the string). For the character array "`arr`", 100 bytes are allocated, but only the first six are filled.

Pointer Arithmetic Loop Optimizations

The main way that we use pointer arithmetic for optimization is to change a loop over an array into loop pointer arithmetic. Note that this is primarily a sequential code optimization, and does not change anything in terms of vectorization for parallel execution.

Pointer arithmetic is mainly used to get rid of an incrementer variable in sequential code. Here's a vector dot product with basic incremented loop variable `i++` and array index syntax `v1[i]` used inside the loop:

```
float aussie_vecdot_basic(
      float v1[], float v2[], int n)
{
    // Basic vector dot product
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += v1[i] * v2[i];
    }
    return sum;
}
```

And here's the same code when converted to pointer arithmetic:

```
float aussie_vecdot_ptr(float v1[], float v2[], int n)
{
    // Pointer arithmetic vector dot product
    float sum = 0.0f;
    float* endv1 = v1 + n;  // v1 plus n*4 bytes
    for (; v1 < endv1; v1++,v2++) {
            sum += (*v1) * (*v2);
    }
    return sum;
}
```

How does this work? We got rid of the temporary variable "`i`" by using pointer arithmetic "`*v1`" instead of array indices "`v1[i]`". We are also using the function parameters "`v1`" and "`v2`" as temporary local variables, as permitted in C++, so we don't need an extra temporary pointer variable.

The way this works with pointer arithmetic is `v1` and `v2` are treated as pointers, which works due to the near-equivalence of pointers and arrays in C++. Rather than using an array index "`i`" we increment both these pointer-array variables:

```
v1++,v2++
```

These for loop incrementers "`v1++`" and "`v2++`" are both adding 4 bytes (the size of a 32-bit `float`) to the pointers. Also note these two increment statements are separated by the C++ comma operator, not by a semicolon.

The "`endv1`" end marker is calculated as the address of "`v1[0]`" plus "`n*4`" bytes, because the "+" operator in "`v1+n`" is pointer arithmetic addition, which is auto-scaled by the size of the pointed-to object (i.e., 4 bytes for 32-bit float here), rather than normal integer addition.

Note that a further micro-optimization is possible. We can change the less-than test ("`v1 < endv1`") to an inequality test ("`v1 != endv1`"), because equality tests are slightly faster than less-than tests. Since this test is effectively inside the loop and done every iteration, this might be worth doing.

The trade-off is safety: it'll become an infinite loop if you get the pointer math slightly wrong, but hey, your code has no bugs, right?

Smart Pointers

Smart pointers are a programming idiom to make C++ pointers safer. They are not a speed optimization, and in fact, they are a wrapper that adds extra logic around the use of a raw pointer, and will be marginally slower. However, they avoid many C++ pointer pitfalls, thereby improving reliability, and will reduce total allocated memory usage by avoiding memory leaks. There may even be an indirect benefit to execution speed if overall memory management is improved.

Programmers have been defining their own smart pointer wrapper classes for decades, but there is now standard support for the idea in the C++ library. In the typical idiom, a smart pointer tracks the creation and destruction of the object it points to, which ensures that the destructor is called. This helps avoid "memory leaks" in standard C++ pointers where an object is allocated with "`new`", but is never deallocated by "`delete`".

The C++ standard libraries have various templates to support smart pointers, mostly since C++11, so they are longstanding features.

- `std::shared_ptr`
- `std::unique_ptr`
- `std::weak_ptr`

`std::shared_ptr` is a reference-counted shared pointer implementation. The idea is that it tracks the total number of pointers to an object, and then automatically destroys the object whenever there's no more pointers to it. This occurs when the last of the "`shared_ptr`" objects is itself destroyed, and then the reference count for the underlying object is zero.

`std::unique_ptr` is a one-to-one mapping of a smart pointer to an object. Whenever the `unique_ptr` object is destroyed (e.g., goes out of scope as a local variable), then both the smart pointer and its underlying object are destroyed or otherwise cleaned up. The `unique_ptr` object can refer to a single object allocated by "`new`" or a single array-of-objects allocated by the "`new[]`" operator.

`std::weak_ptr` is a less commonly used type of smart pointer that has relevance to `std::shared_ptr` in some complicated scenarios. It encodes the semantics where the pointer has a weak association with an object, but not strong enough to own it (i.e., as a shared pointer). However, later on, the weak pointer might get ambitious and want to upgrade to a full owner-operator. Usually, you should choose either of `std::unique_ptr` or `std::shared_ptr`, depending on how many pointers will point to the underlying object.

Pointers vs References

Overall, pointers are a good and bad feature of C++. They are low-level variables that allow efficient processing of memory addresses, so we can code some very fast methods with pointers. They allow us to get very close to the machine.

On the downside, there are pointer pitfalls. Pointers trip up novices and experienced programmers alike. There is an immense list of common faults with pointer manipulation, and coding problems with pointers and memory management are probably half of the causes of bugs in C++ (at least). There are some tools that mitigate against pointer problems (e.g., Linux Valgrind) but it is a never-ending battle against them.

Pointers and arrays were implemented very similarly, and came from the earliest designs of the original C language. Basically, arrays are treated as a specific type of pointer, with various differences depending on whether they are variables or function parameters.

Then came C++ to the rescue. References arrived with the new-fangled programming language (cleverly named as "C++") and were thoughtfully designed as a type of safe pointer that cannot be null. The part is that references are just as efficient as a pointer, with the constraints on references enforced at compile-time.

C++ allows two ways to indirectly refer to an object without creating a whole new copy: pointers and references. The syntax is either "`*`" or "`&`" for their declarations.

```
MyVector *myptr = &mv;  // Pointer to mv object
MyVector &myref = mv;   // Reference to mv object
```

Pointers and references are more efficient than spinning up a new copy of the object, especially when the underlying object is a complicated object. And when you have a function call, you should definitely avoid sending in a whole object.

```
void processit(MyVector v)  // Slow
{
    // ....
}
```

This is inefficient because the whole MyVector object will get copied, via whatever copy constructor you have defined, which is slow. And if you haven't defined a copy constructor, then the compiler uses default bitwise copy of a structure, which is not only slow, but also rarely what you want, and often a bug.

The faster reference version is to use a "`const`" reference (or non-`const` if you're modifying it inside the function):

```
void processit(const MyVector & v) // Reference param
{
    // ....
}
```

The pointer version is:

```
void processit(MyVector * v)  // Pointer param
{
    // ....
}
```

Which is faster in C++ — pointers or references? The short answer of "not any difference" is the general view, because references are implemented as pointers by the compiler behind the scenes. The two functions above are not going to be significantly different in terms of speed.

The slightly longer answer is that references can be faster because there's no null case. A reference must always be referring to an object for the duration of its scope. The C++ compiler ensures that references cannot occur without an object:

```
MyVector &v;           // Cannot do this
MyVector &v = nullptr; // Nor this
MyVector &v = 0;       // Nor this
```

A reference must be initialized from an object, and you cannot set references equal to pointers, because you actually have to de-reference the pointer with the "`*`" operator, which crashes if it's a null pointer:

```
MyVector &v = myptr;  // Disallowed
MyVector &v = *myptr; // Works if non-null
```

There's no way in C++ to get a zero value into a reference variable (we hope). For example, the address-of operator (`&`) applied to a reference variable returns the address of the referenced object, not the memory location of the reference itself.

Hence, references are always referring to something and they cannot be equivalent to the null pointer.

References are slightly faster: The guarantee of an object for a reference fixes all those null pointer core dumps, and also relieves the programmer of the burden of testing for null pointers. The compiler does this guarantee for references at compile-time, so there's no hidden null check being done by the compiler at run-time.

So, there's a minor speed improvement from using references, by not having to add safety checks for "`ptr!=nullptr`" throughout the function call hierarchy.

Pointers can be better than references if you need a "null" situation to occur. For example, you're processing an object that may or may not exist, and you need the pointer to be allowed to be "`nullptr`" if there's no object. This should occur rarely, and references should be preferred in many cases.

And finally, references aren't very useful when you're trying to scan through the data in vectors, matrices, or tensors in an AI engine. You can't do pointer arithmetic on a reference in C++.

Iterator Pointer Arithmetic

Modern C++ iterators are somewhat like pointers, but much more advanced. Nevertheless, you can use pointer-like arithmetic on C++ iterator types for some classes.

Generally, full pointer arithmetic only works properly when the objects are "random access iterators" that have the raw data stored in contiguous memory.

This means it works on classes such as:

- `std::vector`
- `std::array`
- `std::string`
- `std::inplace_vector` (C++26)

Note that some simple operators work properly on more general classes (e.g., the ++ operator on iterators). But we're talking about using integer subtraction to find an array offset, or integer addition to move around an array or vector.

Here's an example that uses pointer arithmetic to compute the index of an item in a vector or array, based on the iterator returned by the `std::find()` function:

```
int find_standard_index(const T& item)
{
    auto iter = std::find(arr_, item);
    if (iter == arr_.end()) return -1;  // Fail
    return iter - arr.begin();  // Pointer arithmetic
}
```

Note that `std::find` also works on scalar array types. Hence, we can use raw array types, but then it's actually doing raw pointer arithmetic rather than iterator arithmetic.

Restricted Pointers and Aliasing

Restricted pointer declarations help the compiler with advanced optimizations like loop unrolling and vectorization by telling the compiler to ignore potential "aliasing" of pointers, allowing much more powerful code transformations on loops. The possible benefit for memory block algorithms is that restricted pointer specifications might help the compiler do auto-vectorization of loops into parallel hardware-assisted code.

The portability of the "`restrict`" keyword is still somewhat lacking. The C99 standard added the "`restrict`" keyword to the C language, but it was not added to C++, and has not been widely supported by C++ compilers. The reason that `restrict` is standard in C and not C++ is not an oversight; there are a large number of problematic issues in merging it into various C++ language features.

Nevertheless, various non-standard features have been added to C++ compilers and can improve their levels of vectorization optimizations. GCC has supported two different keywords, `__restrict__` and `__restrict`, with the same intended meaning. MSVS has `__declspec(restrict)` which defines a restricted storage class, and also supports `__restrict`.

Some compilers have other related keywords for different types of aliasing. Microsoft also has `__declspec(noalias)` which has a slightly different meaning to restricted pointers, in that it tells the optimizer that a function does not modify global state in a hidden way, other than via its parameters, which is a different type of aliasing. GCC also has a "`mayalias`" attribute that permits a pointer to be aliased to suppress some warnings (and presumably also won't be optimized very much!).

11. Contiguous Memory Blocks

Why Contiguous Memory Blocks?

A critical part of optimizing low-latency engines is to store data in a contiguous memory block so that they have a sequential address space. Processing chunks with data in parallel is the main optimization used in both GPU and CPU SIMD acceleration. All of the vectors, matrices, and tensors need their underlying data in a block for efficiency.

Processing data that is in adjacent addresses is much faster than jumping all over the place. Vectors should obviously be stored in a simple contiguous array of memory. Less obviously, similar comments apply to the memory storage of matrices and tensors.

The use of contiguous memory is an important optimization for both sequential and parallel algorithms. The reasons that memory blocks are more efficient include:

- Data locality (cache hits)
- Data block GPU uploads (model weights from memory-to-cache)
- Predictive cache pipelining (in CPU sequential accesses)

Data locality refers to using data in the same or similar address locations. This is helpful for the cache hit rate because data that is already in the cache is much faster to access that a non-cached RAM memory address.

GPU uploads from CPU RAM to the GPU's Video RAM (VRAM) is done in blocks. Obviously, we don't want to be uploading random bits of data from different parts of the RAM.

Non-GPU architectures also benefit from the use of contiguous memory. This is obviously true of CPU SIMD instructions (e.g., AVX on x86), but even in sequential execution, the CPU has its own RAM caching methods and often has other optimizations of memory accesses. Predictive cache pipelining is where the CPU attempts to predict what the next memory location will be, and load it in a pipelined speedup, before being asked. This pipelining of memory accesses is much faster than doing completely sequential address lookups.

Typically, predictive cache pipelining uses the simple heuristic that the next address is the most likely next request, which assumes that data is being processed in order of the addresses. Hence, scanning an array in reverse is the worst possible order for these CPUs. Similarly, jumping around to different memory addresses, such as scanning the column of a matrix using a large "stride," is also inefficient.

Low-Level Memory Block Functions

Memory block operations in the standard C++ libraries are implemented using fast assembly language behind the scenes. The main functions in the standard C++ library that operate at a low level on binary bytes in a memory block are:

- `memset()`: set bytes to a value, usually used to clear bytes to zero.
- `memcpy()`: copy bytes.
- `memmove()`: copy bytes, but tolerates overlapping regions.
- `memcmp()`: compare a sequence of bytes.
- `memchr()`: search for a byte in a sequence.

These functions are lower-level than the modern C++ versions, such as `std::copy`, `std::move()`, and their "backward" versions. The above listed memory block functions are not aware of object-level semantics, and won't run any of the special functions on memory containing objects.

Note that unlike the standard string functions (such as `strlen`), these functions do not assume a block is null-terminated by a zero byte. Zero is simply a binary value, and these functions don't stop at a zero byte. All of these functions operate on a block of memory with a known maximum byte length.

Each compiler environment typically offers some extra non-standard byte-wise functions that are also fast. Some of the less standardized C++ intrinsics that operate on memory blocks include:

- `_memccpy()`: copy bytes up to a specified sentinel byte.
- `memicmp()` or `_memicmp`: compare bytes ignoring letter case.
- `bcopy()`: copy bytes
- `bzero()`: clear bytes to zero.
- `bcmp()`: compare bytes.
- `_byteswap_uint64()` (Microsoft intrinsic): Swap the bytes of an integer.
- `__builtin_bswap16()`: GCC function to swap the bytes in an integer. There are versions for 32-bit and 64-bit.

Fast Memory Block Operations

The slow way to do things in arrays is one element at a time. The faster way is to use the standard memory block functions on the whole array. There are a number of standard functions that operate on array data or memory blocks and they are very fast.

Initialize with `memset` byte fill. The `memset` function sets all of a memory block to a byte value. It is widely used as a fast way to initialize a block of memory to all zeros.

```
memset(&x, 0, sizeof(x));
```

Almost all usages of memset will be for the zero byte. The only other usage I've seen is to fill memory with a dummy non-zero byte as a form of mutation testing to catch uses of uninitialized memory.

```
memset(&x, 0x55, sizeof(x));
```

Fast array copying with `memcpy`. The fast way to copy an entire array is with `memcpy`. Rather than copy each element of an array, one at a time, in a loop, the `memcpy` standard library function can be used to copy the entire array in one statement:

```
memcpy(destarr, srcarr, sizeof(srcarr));
```

Note that this is a bitwise copy of the array intended for simple data types. For example, it won't run copy constructors if applied to an array of objects.

The `memcpy` function does a very fast memory block copy. It is like `strcpy` in that the destination is the first parameter. `memcpy` will copy everything, even null bytes and hidden padding bytes. It keeps going even if it finds a null byte, so it is not like `strcpy`, and will always copy a fixed number of bytes. `memcpy` is a super-fast byte copy, but is unsafe, because it does not have well-defined behavior if the source and destination blocks overlap.

Safer byte copy with `memmove`: The `memmove` function is a safer version of `memcpy`, which also works correctly if the memory blocks overlap. If the source and destination blocks don't overlap, it's the same as `memcpy`, except probably slightly slower. If they do overlap, then `memmove` conceptually will copy the source to a temporary area, and then copy it to the destination block.

Copying arrays using `struct` assignment. An alternative method of copying arrays is to make use of `struct` assignments. This is similar to how `std::array` works, which could also be used in a similar vein, but this example totally avoids any constructor, copying or move costs (and also works in C).

This method is not portable, is very unreadable and uses pointers incorrectly by converting between two different pointer types. However, it can be faster than `memcpy` because it makes use of the assignment operator rather than calling a function. On the other hand, `memcpy` is an intrinsic function that might be inlined to assembler instructions by the compiler, so this trick might be a waste of time. Benchmarking is recommended here.

To copy an array using this method it is necessary to declare a new dummy `struct` type that is the same size as the array that is to be copied. Then we use type casting to fool the compiler into thinking it is copying structures when really it is copying arrays. The method is illustrated below:

```
struct dummy_transfer { // The new struct type
    int a[MAX]; // This field gives the right size
};

int a[MAX], b[MAX]; // The array variables being copied
static_assert(sizeof(struct dummy_transfer) == sizeof(a));
*(struct dummy_transfer *)a = *(struct dummy_transfer *)b;
```

The assignment statement first type casts both "`a`" and "`b`" to be pointers to the new `struct` type, and then dereferences these pointers so that the compiler believes it is assigning between two structures. The assertion is an efficient compile-time safety net to ensure that the copying statement will work.

Of course, a better way entirely is probably to put the array inside a class object, with lovely encapsulation and modularity, and then we can simply copy the objects.

`memcmp` byte comparisons. The `memcmp` function does a byte-wise comparison of a memory block. Its return value is like `strcmp`, returning 0 for equality, and a negative or positive value otherwise.

Note that `memcmp` is not like `strcmp`, and will not stop when it finds a zero byte.

Memory Block Function Pitfalls

The standard memory block functions are fast, but they are not always safe. Here are some of the common pitfalls that commonly occur in everyday coding.

`memset` sizeof problem. Here's another glitch in using memset inside functions:

```
void zero_array(int arr[10])
{
    memset(&arr, 0, sizeof(arr));  // Bug
}
```

The problem is not memset, but the sizeof operator on function parameters. An array parameter in a function is like a hologram and isn't really there. It's not really an array, but a pointer, and `sizeof(int[10])` is the same as `sizeof(int*)`. Hence, `sizeof(arr)` is probably only 4 or 8, rather than 40 or 80, leaving most of the array uninitialized. Personally, I recommend a memset debug wrapper function to catch this kind of problem at runtime, or maybe a tricky preprocessor macro can detect it at compile-time with a `static_assert` somehow.

`memset` portability issue. Even though it's a fast zeroing method, the use of `memset` to zero bytes has an obscure portability problem on any architecture where all-bytes-zero is not the same as all data types zero. However, on most standard platforms, all-bytes-zero is correct for all types: integer zero (ignoring endianness), floating-point zero (positive zero is all bits zero), and the null pointer.

`memcpy` overlapping blocks error: The only downside with `memcpy` is that it can fail with overlapping ranges for the source and destination blocks, so if you are shuffling arrays up or down one element using `memcpy`, then you have to be careful, because the results on overlapping ranges are undefined. Here's a buggy example of using `memcpy` to remove the first character of a string in place:

```
memcpy(s, s+1, strlen(s+1)+1);  // Bug
```

The problem is that the blocks starting at "`s`" and "`s+1`" are overlapping. It is implementation-defined whether it will be correct. The fix is simply to use `memmove`, which always works correctly for overlaps:

```
memmove(s, s+1, strlen(s+1)+1);  // Correct
```

memcmp return value. A pitfall with `memcmp` is that you cannot assume that it returns `1` or `-1`, but must compare the return result to zero (like the `strcmp` function).

```
if (memcmp(&a, &b, sizeof(a)) == 1)  // Bug
if (memcmp(&a, &b, sizeof(a)) > 0)   // Correct
```

memcmp object equality testing. Looking at the `memcmp` function, you might think of it as an opportunity to do a fast equality/inequality test on large objects by simply doing a byte-wise test. You would not be the first to think that.

Consider if you have a complex number class:

```
class MyComplex {
    float real,imag;
    // .. etc
}
```

The brute-force equality test is:

```
bool is_equal(const MyComplex &a, const MyComplex &b)
{
  return (a.real == b.real && a.imag == b.imag);
}
```

Our idea to optimize this with `memcmp` looks like:

```
bool is_equal(const MyComplex &a, const MyComplex &b)
{
  return memcmp(&a, &b, sizeof(MyComplex)) == 0; // Bug!
}
```

Unfortunately, there are multiple obscure pitfalls with this approach:

- Padding bytes
- Two types of floating-point zero
- Multiple types of floating-point `NaN` (not-a-number)
- Bitfields

Padding byte problems. If float is 4 bytes, but the machine has 8-byte alignment, then the "`real`" and "`imag`" data members will be stored on 8-byte alignment addresses, and there will be another 4 bytes each of dummy padding.

It doesn't even have to be on a machine with alignment issue, but can occur with a bigger object if we've mixed different size objects (e.g., `char`, `int`, and pointers). The padding bytes will be uninitialized (e.g., for local objects or if allocated with "`new`"), in which case they can contain random values. Since `memcmp` does not skip the padding bytes, its test will fail. Now, we could possibly work around this portability issue via the use of `memset` in the constructor, or `calloc` memory allocation, to zero all of the bytes of an object including the padding bytes.

Negative zero problems. Unfortunately, the next problem is not a portability problem, but a fundamental issue with floating-point numbers. There are two zeros! There's the normal zero with all bits zero, and negative zero, with sign bit set, but other bits zero. Hence, bitwise testing of float numbers fails for a negative zero.

`NaN` problems. Similarly, but perhaps less seriously, the representation of `NaN` (Not-a-Number) in floating-point is also not fixed. There are multiple values of `NaN`, both positive and negative. So, `memcmp` would say the float values differ, even if both are `NaN`.

Bitfield problems. If our structure has bitfield data members, this `memcmp` idea fails too. Bitfields are a standard C++ feature that is defined with a colon syntax:

```
unsigned int myflag:1;  // Boolean bitfield with 1-bit
```

With bitfields it's implementation-defined how this is represented numerically, and there might be undefined bits in the same byte, or extra padding bytes again.

Still want your `memcmp` speedup? I've just shown you about 15 pitfalls, but maybe you still want to live on the edge and get that speedup? You can use `memcmp` to do fast array or object comparisons if you're really sure you have:

- Zero byte initializations. All allocated arrays or objects must be first zero'd by `memset` or `calloc`. You cannot rely on constructors, and it's hard to put a `memset` as the first action of the constructor due to initializer lists and the various defined base classes. You might also have to intercept all of the `new` and `new[]` allocation operators with your own wrapper that does `memset` on the block, rather than use constructor tricks. It's also unclear if you can actually rely on `static` or global variable initialization to carefully zero all the padding bytes in an array or object. Probably it works on most platforms, but I doubt it's fully portable. To be sure, use `memset` on the global variables during program startup.
- No bit-fields used. That's easy, at least.
- Floating point computations should avoid negative zero and `NaN`.

Raw Subarray Memory Blocks

Passing raw subarray types to functions can be a fast alternative to some of the modern C++ contiguous containers (i.e., `std::array`, `std::vector`). However, the passing of a container object by reference with "`const&`" parameters is also very fast, so don't assume that raw arrays are always faster.

If a function accepts a raw array type, it is possible to pass it any array as an argument, or any pointer of the right type. In this way, it is possible to pass memory blocks or "sub-arrays" to a function by passing the address of a particular array element. A function to operate on a particular type of array can be written, and used to operate on various arrays.

```
void clear(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        a[i] = 0;
}

void test_subarrays()
{
    int a[100];
    clear(a, 10); // clear first ten, 0..9
    clear(a + 50, 10); // clear 50..59
    clear(&a[50], 10); // clear 50..59 (equivalent)
}
```

Multidimensional subarrays. It is also legal to pass multi-dimensional arrays to functions. However, the sizes of all but the first dimension must be specified in the function receiving the array. For example, to pass a two-dimensional array to a function, the function header would look like:

```
void fn(int a[][SIZE2]);
```

The reason for this restriction is that the compiler cannot determine the address of an arbitrary array element if it does not know the sizes of all but one of the dimensions.

Because the sizes of most of the array dimensions must be specified in the function declaration it is very difficult to write a function to act on sub-arrays of multi-dimensional arrays. For example, this idea would be useful to define library functions to operate on matrices with different dimensions.

Ideally, we would like one function to calculate the determinant of a matrix for any dimension (i.e, an n-by-n matrix where n varies). Consider how we would like the determinant function to look:

```
double determinant(double matrix[][], int n); // No!
```

Ideally, the dimensions of the matrix are not specified at compile-time, but are specified at run-time by the n argument. This is not possible as a simple C++ declaration because the second dimension (i.e., n) needs to be specified in the definition of the two-dimensional array type. The best solution is to use dynamic multi-dimensional arrays.

Cache Warming

Cache warming is a specific type of prefetching optimization on memory blocks aimed at keeping the various memory caches fresh. It typically involves scanning through all the memory data required for the "hot path," even though there's no real intention to use the data (until later). The hot path maintains a warm cache, so that when the hot path is executed for real, then memory accesses are very fast.

There are multiple ways to trigger the prefetching of data needed to keep the cache warm:

- Low-level C++ prefetching primitives.
- Copy to `volatile` temporary variables.
- Explicit dry-run parameters in the code.

Unlike other types of CPU prefetching, cache warming is something your C++ code does directly, rather than a hardware-enabled feature. It's up to you to determine what data is needed the most in hot path computations, and then pre-load that data on every pass-through. You effectively do a "dry run" of the hot path, but access the memory to ensure it's maintained in the cache.

Note that cache warming is not always a guaranteed win. Using the "dry run" approach can end up with a lot of extra conditional tests:

```
if (!dry_run) {
    // Do something
}
```

This can negatively impact performance in two ways:

- Runtime cost of testing the flag, and
- Extra branches of code that slow down CPU branch prediction.

As with everything in coding, you really need to time it to see if these costs are less than the gain from faster memory cache accesses.

Memory Prefetch Primitives

Although you can "manually" prefetch data in basic C++ code, there are also some builtins that are convenient for larger amounts of data. Some of the C++ primitives to use for cache warming include:

- `__builtin_prefetch` (GCC)
- `_mm_prefetch` (GCC)

Prefetching is more effective on some data structures than others, with a general preference for contiguous data blocks. Cache locality issues with the "cache lines" of size 64-256 bytes are another reason. As a practical example, contiguous arrays are better than dispersed data structures liked links lists and trees. This means that `std::vector` contiguous memory layouts can be more effectively prefetched than the spread-out memory used by `std::list` objects.

Volatile Temporary Variables

Another approach for manual prefetching is the use of `volatile` specifier on temporary variables. By assigning data to a `volatile` temporary variable, the optimizer cannot remove an apparently unused assignment. For example, consider if we do this:

```
int temp = my_data[0];
```

The C++ compiler may notice that "`temp`" is not used anywhere else, so it can helpfully throw away that entire assignment statement. The solution is to use the `volatile` specifier:

```
volatile int temp = my_data[0];
```

The compiler is forced to load the data into memory even when it seems to be unused by the remainder of the code block, because assigning any data to a `volatile` variable is itself a side-effect.

Note that we only want to declare temporary variables as `volatile`, but not the shared global data arrays we're trying to prefetch. We don't want the main data structures to have this status. If our main global variables or arrays were declared as `volatile`, this would actually interfere with having them loaded from the memory caches. They would be uncached!

Pros and Cons of Cache Warming

The advantage of the use of cache warming is that all the various data structures are kept warm in the memory caches (i.e., the L1/L2/L3 CPU memory caches). The downside is extra processing that occurs whenever you're not using the memory again. In other words, there are extra computations done on the "cold path" every time, just to keep the "hot path" all snuggly and warm.

The code to traverse all the memory data structures can be a significant cost in itself, although it hopefully only occurs during the cold path. There are several advanced tweaks to optimize your cache warming code:

- Exploit cache line sizes for quicker loading of contiguous data.
- Limit cache warming to the total L1/L2/L3 cache size.

A further optimization of cache warming is to use "cache lines" to your advantage. The L1/L2 caches don't work on individual bytes, but on blocks of memory called "cache lines", which are usually sized between 64 bytes and 256 bytes (e.g., Intel is usually 64 bytes, Apple M2 is 128 bytes, some other CPUs are 256 bytes). Hence, to load a "cache line" of 64 bytes on an Intel CPU, you only need to load one byte from the 64-byte block. Your C++ code doesn't need to explicitly touch every element of a vector to have the entire vector hot as a fresh-baked oven loaf in the cache. Admittedly, this doesn't speed up the hot path itself, but only the preliminary cache warming code.

An important limitation of cache warming is the maximum sizes of the L1, L2, and L3 caches. If you're trying to warm up the CPU cache for your 7B AI model, that's 7 billion floating-point numbers, and trying to keep them all in the CPU cache isn't going to work. On the other hand, you can probably preload an entire 7B model into the CPU RAM (i.e., global memory, not the caches), or into the GPU's VRAM, but that's preloading not cache warming, and it's a slightly different story.

If you know your CPU's cache size, you can optimize your cache warming strategy by only trying to prefetch that much data. If you load more data than the cache size, the newly warmed data is just evicting other data from the cache that you prefetched earlier in the warming code. Hence, prefetching exactly the amount of data equal to your CPU cache size is the optimal cache warming strategy.

Dynamic Memory Management Pitfalls

Memory management is really not the strong suit of C++. If your program is crashing or behaving badly, it's highly likely to be some kind of memory problem. There are so many pitfalls in C++ dynamic memory management, and even in static or global (non-dynamic) memory, that it's hard to list them all.

C++ programs have access to a large block of free memory, called the heap. The actual size of the available memory depends on the system. This memory is available to a C++ program which can allocate itself chunks of memory from this heap. This is useful when a C program does not know beforehand how much data is being stored, and hence, how much memory is required. Instead of allocating a large array to cater for the worst case, the program can allocate itself blocks of memory as required.

Blocks of dynamic memory can be allocated in two main ways:

- The C++ style "`new`" or "`new[]`" operators
- The older style `malloc()` and `calloc()` functions (inherited from C)

Other ways to allocate dynamic memory include:

- `strdup()`: make an allocated copy of a string.
- `realloc()`: a companion to malloc/calloc that is rarely used.

Once the memory is no longer needed it is "freed" back to the heap. Again, there are two main ways:

- The C++ style "`delete`" and "`delete[]`" operators
- The older style "`free`" function

Some of the main memory problems in a C++ program can include:

> **Uninitialized new memory.** The `new` operator does not initialize the new chunk of allocated memory. Accidentally using it is a common bug.

Uninitialized malloc memory. The `malloc` function also does not initialize its allocated memory. Again, use of a memory block that is allocated by `malloc` but hasn't been properly cleared is a common bug. One of the mitigations is to use `calloc` instead, because `calloc` does zero the bytes of every block it allocates.

Mismatched new/delete with malloc/free. Memory allocated with `new` should be deallocated by `delete`, but `malloc`'d memory should be `free`'d. Never the twain shall meet, or else kaboom.

Mixing new/new[] and delete/delete[]. Memory allocated by `new` should be released by `delete`, but memory allocated by the array version "`new[]`" should be freed by the `delete[]` array version. Again, they're not supposed to mix.

free(nullptr) is harmless. If it's so harmless, why is it a pitfall? Sure, `free(nullptr)` is officially defined by the standard to do nothing. But if your coding is doing this, it sure walks and talks and quacks like a buggy duck.

`strdup(nullptr)` is not harmless. This is probably a crash, but even on systems where it's not, it's clearly a bug in your code if you're trying to duplicate a null pointer.

Pitfalls for Non-Dynamic Memory Blocks

There's so many pitfalls in management dynamic memory, with either new/delete or malloc/free, that surely we've run out? No, don't worry, it's comforting to know that there are still a bunch more insidious problems in other types of non-allocated memory.

Here's a list of some more fatal memory stomps that aren't about allocated blocks on the heap:

- Buffer overrun of a global, local, `static`, or stack buffer variable.
- Returning the address of an automatic local variable on the stack (i.e., non-`static` variable).
- Trying to write to addresses of string literals (often a crash if they're non-writable, but maybe worse behavior if it can be modified).
- Modifying `arr[10]` in an array of size 10 (raw arrays or `std::array`).
- Uninitialized local variables or local buffers on the stack (non-`static`).

- Using an uninitialized local pointer variable to access some random address in Timbuktu.
- Null pointer dereferences. Oh, well, at least you initialized it.
- Returning the address of a "`static`" local variable (aliasing problems).
- Using a negative array index.
- Modifying a string literal (they're in read-only memory on Linux).

The standard C++ library functions can also have problems:

- `strcpy()` on overlapping string arguments: `strcpy(s, s+1);`
- `strncpy()` can leave strings without a null byte terminator.
- `memcpy()` on overlapping memory blocks (use `memmove` instead).
- Trying to `free()` or `delete` a global, `static`, stack or instruction address will crash.
- Double `fclose()` on file pointers from `fopen`.
- Ignoring the return value of `erase()` in an iterator loop.

References

1. Paul Bilokon, Burak Gunduz, 8 Sep 2023, *C++ Design Patterns for Low-latency Applications Including High-frequency Trading*, https://arxiv.org/abs/2309.04259, Code: https://github.com/0burak/imperial_hft
2. Edelweiss Global Markets Oct 14, 2024, *Cache-Warming*, https://edelweissgm.github.io/hft/2024/10/14/CacheWarming.html
3. Ibrahim Essam, Jul 19, 2024, *Cache warming and memory access*, https://ibrahimessam.com/posts/cache/
4. Daniel Lemire, April 2018, *Is software prefetching (__builtin_prefetch) useful for performance?* https://lemire.me/blog/2018/04/30/is-software-prefetching-__builtin_prefetch-useful-for-performance/
5. Johnny's Software Lab, March 31, 2024, *The pros and cons of explicit software prefetching*, https://johnnysswlab.com/the-pros-and-cons-of-explicit-software-prefetching/
6. Katecpp, Oct 5, 2015, *Improve performance with cache prefetching*, http://katecpp.github.io/cache-prefetching/

12. Loop Optimizations

Sequential vs Parallel Loop Optimizations

Loop optimizations are the basic of many speedups to the processing of contiguous array data. Loops are often sources of inefficiency and can be optimized in numerous ways, such as:

- Cache locality — processing data in the fastest order for CPU caches (sequential).
- Parallelization — allowing vectorization via CPU SIMD instructions or a GPU.

Not all loop transformations are created equal. Some of them are best for sequential code optimizations, whereas other loop transformations are used to parallelize loops for vectorization.

Loop transformations that are good for both sequential and parallel loop optimization include:

- Loop unrolling — repeat the loop body to reduce loop test overhead and parallelize the loop body.
- Loop peeling — unroll the first few iterations.
- Loop coalescing — flatten nested loops.
- Loop splitting — split out subportions of the iteration range.
- Loop collapsing — another way to flatten nested loops.
- Loop interchange — switch the inner and outer loop iterators of nested loops.
- Loop reordering — change the ranges and arrangements of inner/outer nested loops.

Some loop transformations are mainly for sequential improvements, and are not parallelization in themselves. However, these techniques can sometimes help with parallelization if they enable another followup loop parallelization optimization.

Loop transformation optimizations which tend to be good for sequential code optimizations but not parallelization include:

- Loop fusion — combine or "fuse" the bodies of two loops.
- Duff's device — amusing but impractical coding trick for loop unrolling.
- Loop code motion — move or "hoist" loop-invariant calculations from the loop body to pre-loop initialization.
- Loop perforation — randomly skip a subset of loop iterations; it's really a thing.
- Loop sentinel — fake it till you make it.
- Loop iterator strength reduction — change "*" to "+" if you can.
- Loop reversal — going backwards, and yet, still making progress!

Parallelizing loop optimizations with a main goal of vectorization of the loop body include:

- Loop fission — opposite of loop fusion; split a single loop body into two loops.
- Loop tiling — process sub-parts of contiguous data in separate loops.
- Loop distribution — split two sub-parts of a loop body into two simpler separate loops.

Loop Fusion

Loop fusion is a well-known code optimization where two separate loops are merged into a single loop. This does not change the amount of in-loop computation in either loop body, but reduces the loop overhead of the exit test by half. There is also often a benefit from data locality that reduces data movement and temporary data storage, which can also improve overall speed.

Note that loop fusion is not great at vectorization, because complicated loop bodies are actually harder to parallelize. Most of the benefits arise in traditional sequential code execution, which is why its theory dates back many decades. For modern parallel execution on GPUs, loop fusion is often a poor choice, and more benefits may arise from loop fission (the opposite of fusion) and loop vectorization.

Example: Loop Fusion: The general idea is to combine the body of two loops into a single loop. Here is a simplistic example with the (non-fused) loops for initializing two vectors using two sequential loops:

```
for (i = 0; i < n; i++) v1[i] = 0;
for (i = 0; i < n; i++) v2[i] = 0;
```

And here is the version with loop fusion:

```
for (i = 0; i < n; i++) {
    v1[i] = 0;
    v2[i] = 0;
}
```

Note that the loop fusion version incurs the same number of assignments for initialization, but only half of the loop overhead cost (i.e., half of the "`i < n`" and "`i++`" operators have been optimized away).

For the sake of argument, let's pretend we don't know a fast way to clear a vector in C++ like `memset` or `calloc` or load-time `static` variable initialization.

Loop Perforation

The intentional introduction of randomness to code is known as a "stochastic" algorithm. Personally, I'm more familiar with unintentional introduction for randomness, otherwise known as a "bug," but now when it happens you can tell your boss that you were adding "stochastic functionality."

Code perforation is an optimization technique that trades accuracy for speed, by randomly (ahem, I mean, stochastically) skipping some computations. Essentially, using loop perforation is similar to an approximation with a random element, but in a generalized way for any iterative code.

It's kind of like how teenage children randomly skip their homework.

Loop perforation skips iterations of a loop in a probabilistic manner. Randomly skipping some percentage of the loop bodies doesn't sound like a good plan, but it has its merits. In some types of applications, such as an AI inference computation, there's so much going on that no-one's going to notice a few missed beats. Apparently it can even be useful.

Well, at least it's faster to do nothing.

Example: Loop Perforation: Here is an example of adding loop perforation to a vector dot product computation. This is an incredibly slow version, and is not recommended, but is just to give the idea of skipping a percentage of the iterations:

```
float aussie_vecdot_perf(
        float v1[], float v2[], int n, int pc)
{
    // Loop perforation -- vector dot product
    float sum = 0.0;
    for (int i = 0; i < n; i++) {
        if ( ( rand() % 100 ) + 1 <= pc) {
            // This iteration is perforated...
            continue; // Skip it...
        }
        sum += v1[i] * v2[i];
    }
    return sum;
}
```

Loop Unrolling

Loop unrolling is a code optimization where the body of a loop is repeated in sequential code. This speeds up the algorithm because the overhead of both the incrementer and the loop iteration test is avoided. In some cases, the entire loop can be unrolled, usually when the loop iterations are finite and known at compile-time. In other cases of partially unrolling, the loop body can be repeated multiple times, and thereby the loop test only occurs every few iterations.

Example: C++ Loop Unrolling of Vector Dot Product. Here is the basic C++ non-unrolled vector dot product code:

```
float aussie_vecdot_basic(float v1[], float v2[],int n)
{
     // Basic vector dot product
     float sum = 0.0;
     for (int i = 0; i < n; i++) {
         sum += v1[i] * v2[i];
     }
     return sum;
}
```

If we know the value of *n*, e.g., that *n=5*, then we can completely unroll it:

```
  return v1[0] * v2[0]
       + v1[1] * v2[1]
       + v1[2] * v2[2]
       + v1[3] * v2[3]
       + v1[4] * v2[4]
       ;
```

If we don't know the value of *n*, we can still unroll multiple iterations. Here's an example of 4-level loop unrolling of vector dot product in C++ by assuming that *n* is a multiple of 4:

```
float aussie_vecdot_unroll4(float v1[], float v2[],int n)
{
   // Loop-unrolled Vector dot product
   if (n % 4 != 0) {
       aussie_assert(n % 4 == 0);
       return 0.0; // fail
   }
   float sum = 0.0;
   for (int i = 0; i < n; ) {
       sum += v1[i] * v2[i]; i++;
       sum += v1[i] * v2[i]; i++;
       sum += v1[i] * v2[i]; i++;
       sum += v1[i] * v2[i]; i++;
   }
   return sum;
}
```

And here's a generalization of that 4-level unrolling with extra code to handle the leftover cases if *n* is not a multiple of 4. Although the extra cases look messy, they are not actually the main performance bottleneck.

```
   float aussie_vecdot_unroll4b(
      float v1[], float v2[], int n)
   {
       // Better loop-unrolled Vector dot product
       int i = 0;
       float sum = 0.0;
       if (n % 4 != 0) {
           // Handle the extra cases...
           switch (n % 4) {
           case 1:
               sum += v1[i] * v2[i]; i++;
               break;
```

```
            case 2:
                sum += v1[i] * v2[i]; i++;
                sum += v1[i] * v2[i]; i++;
                break;
            case 3:
                sum += v1[i] * v2[i]; i++;
                sum += v1[i] * v2[i]; i++;
                sum += v1[i] * v2[i]; i++;
                break;
            default: aussie_assert_not_reached(); break;
            } // end switch
            // Keep going with rest of the vector
        }
        for (; i < n; ) {  // Unrolled 4 times...
            sum += v1[i] * v2[i]; i++;
            sum += v1[i] * v2[i]; i++;
            sum += v1[i] * v2[i]; i++;
            sum += v1[i] * v2[i]; i++;
        }
        return sum;
    }
```

This code is just an example for explanation. There are various further code optimizations that can be done for production-level efficiency. For parallelization, the loop body should call an intrinsic function to vectorize the method.

For many applications, we could choose our data structure sizes as multiples of the loop unrolling factor, and thereby avoid ever having any of the "leftover" cases.

For sequential code, we could change it to use pointer arithmetic rather than array indices, we might try replacing the four `i++` operators with `i+=4`, change the integer modulo operator (`%`) to a bitwise-and operator test (i.e., use "`n&3`" not "`n%4`", which works since 4 is a power-of-two), and it also might be better to use "+" rather than the "+=" operator.

Finally, if we carefully code the leftover cases, the main loop could be unrolled to many more levels than just four.

Duff's Device for Loop Unrolling

There's a neat coding trick called "Duff's Device" for loop unrolling, which uses a `switch` with `case` fallthrough to mimic assembler coding style. However, it's not great for vectorization as it's likely to confuse the compiler, so may be mostly of theoretical interest.

```
float unroll4_duff(float v1[], float v2[], int n)
{
    // Unrolled dot product with Duff's Device
    int i = 0;
    float sum = 0.0;
    switch (n % 4) {
        for (; i < n; ) {
            case 0: sum += v1[i] * v2[i]; i++;
            case 3: sum += v1[i] * v2[i]; i++;
            case 2: sum += v1[i] * v2[i]; i++;
            case 1: sum += v1[i] * v2[i]; i++;
            default:;
        } // end for
    } // end switch
    return sum;
}
```

What's happening here? My brain hurts looking at this code! The trick is that the outside `switch` branches into a `case` that is inside the body of a `for` loop. This is not normal everyday coding, because there's a loop inside a `switch`, and the loop body control flow crosses over several of the `case` statements. Also, none in the `case` statements has a "`break`" statement and they instead rely on fallthrough semantics. Similarly, the "`default`" clause is mainly just to avoid getting a spurious compilation warning (i.e., "missing `default`"), and also has no "`break`" with only a lonely semicolon. Note also that the `case` labels are written in reverse order from top to bottom (3..2..1), except for 0 at the top.

How does this even work? The first point is that it *does*. This code performs the exactly correct number of iterations for any value of `n` (except `n==0`), and similar versions with an unrolling factor of more than 4 will also work (i.e., if you change "`n%4`" and add more `case` constants). The code looks like a hack, but actually uses standardized C++ semantics of `case` fallthrough and `switch` multi-way control flow and should work on all platforms. Branching into the middle of a loop with a switch is valid in C++ provided it doesn't bypass any local variable initialization (hence, don't put "`sum`" into the `switch`).

Also, the `case` fallthrough semantics (i.e., without a "`break`" ending each "`case`") are standard for C and C++ since inception. Finally, note that this code is buggy for `n==0`, because it incorrectly does 4 iterations, so it ideally needs a parameter validation assertion at the start.

Bug alert! Note that you cannot tweak the "`i++`" instruction using the standard idiom:

```
sum += v1[i] * v2[i++];  // Bug!
```

The obscure problem is that the "`*`" operator doesn't guarantee left-to-right evaluation of its operands. The code assumes evaluation order of: `v1[i]`, `v2[i]`, `*`, `i++`, starting from the left. However, the C++ optimizer can legally do this order of operations: `v2[i]`, `i++`, `v1[i]`, `*`, which is not what you intended and gets the wrong array element for `v1[i]`. This code might be unreliable across platforms, or it might work in the debugger mode, but fall over once you turn on high levels of optimization. So, there is an "order of evaluation" pitfall if you put "`++`" in an operand of the "`*`" operator or many other binary arithmetic operators.

Is Duff's Device any faster? The short answer is "not really," although it looks very appealing (or appalling). Firstly, note that this trick is not actually very useful for vectorization, because a `switch` cannot branch into the middle of a vectorized intrinsic (i.e., if you replace the loop body with a SIMD instruction). Furthermore, although I haven't tested it, I doubt many optimizers will be able to auto-optimize that complex control flow with SIMD instructions. In sequential code, this method also isn't much faster, as it doesn't really have any fewer operations than a basic unrolled loop (i.e., with extra cases handled separately before or after the main loop). The above example of Duff's Device can be further sped up using pointer arithmetic and "looping down to zero" optimizations, but so can the other unrolled versions. However, there is a minor speed advantage in terms of "instruction locality" because the above code is very concise.

The main advantage of Duff's Device is to bamboozle your colleagues. You can use Duff's Device with any unrolling factor, not just 4 as in the example shown above (e.g., change to 8 by using "`n%8`" and adding cases for 4, 5, 6, and 7, ordered from 7 down to 1, leaving 0 on top). Actually, the unrolling factor needn't be a power-of-two. Make it a prime number for extra bonus points. If you want more of this kind of coding trickery, also search up Jensen's device and Pigeon's device.

Loop Tiling or Blocking

When you hear about a "tiled MatMul" or a "blocked GEMM," this is the "tiling" or "blocking" optimization method it refers to. MatMul is matrix multiplication and GEMM is General Matrix Multiplication (i.e., the same thing). Tiling is the optimization that most applies to speeding up matrix or tensor multiplications.

This optimization is for two-dimensional data (e.g., matrices). When you hear "tiles" or "blocks," think squares or rectangles of data. For example, if you have a 512x512 matrix, then a tiled algorithm might act on 16x16 sized chunks, one at a time. Loop tiling is an optimization of two-dimensional or three-dimensional data such as matrices or tensors. The one-dimensional equivalent of processing sub-parts of a one-dimensional array is called "strip mining", "loop sectioning" or often simply "vectorization."

In other words, tiling means operating on small subsections of a matrix. If you hear "tiled tensor" that could mean two-dimensional data (i.e., just a fancy name for a matrix), or alternatively it might refer to three-dimensional data, in which case, don't think anything or else your head will hurt.

Loop tiling is a method of executing sub-parts of nested loops in a way that maximizes data locality, increases cache utilization, and improves parallel execution. This is also called "loop blocking" because it processes the data a "block" at a time, although the term "tiling" is more widely used in research. The two-dimensional sub-partitions of the data that are square or rectangular are called "tiles" or "blocks".

The same number of arithmetic operations are performed in a tiled versus non-tiled algorithm. However, there should be fewer loads of the data into memory with tiling. The downside is that tiling introduces additional loop overhead. In fact, rather than flattening nested loops over a 2-D array (e.g., 512x512), tiling often introduces additional levels of nesting! The two small loops that spin through the 16x16 square shape of a single "tile" or "block" are often newly added inner loops. So, loop tiling often adds two new layers of nested loops inside your already-nested loops. It makes you wonder how it can even be faster!

Example: Tiled Matrix Clear: For these examples, there is a type "`ymatrix`" type:

```
typedef float ymatrix[ROWS][COLUMNS];
```

If we forget about `memset`, here is the simple code to clear a matrix one element at a time in a brute-force nested loop (non-tiled):

```
void aussie_clear_matrix(ymatrix m)
{
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLUMNS; j++) {
            m[i][j] = 0.0;
        }
    }
}
```

Now we decide to add a 4x4 square tile optimization to this code. The result is an extra two levels of nested loops. Here is the basic code which assumes that the row and column dimensions are exact multiples of the tile size, so there's no extra leftover cases to handle:

```
void aussie_clear_matrix_tiled(ymatrix m)
{
    const int TILEX = 4; // 4x4 tile size
    const int TILEY = 4;
    static_assert(ROWS % TILEX == 0, "Exact X");
    static_assert(COLUMNS % TILEY == 0, "Exact Y");
    for (int i = 0; i < ROWS; i += TILEX) {
      for (int j = 0; j < COLUMNS; j += TILEY) {
          // Do the 4x4 tile...
          for (int tx=i; tx < i+TILEX; tx++) {
              for (int ty=j; ty < j+TILEY; ty++) {
                  m[tx][tiley] = 0.0f;
              }
          }
      }
    }
}
```

Unrolled Tiles. One followup optimization trick with a tiled loop algorithm is to apply loop unrolling to the two inner loops. This avoids the extra overhead of the two extra inner loops, but retains the data locality benefits of tiling. This optimization results in a fully "unrolled tile" computation without any extra inner loops. In the above example, the two inner loops of a 4x4 tile would be replaced with 16 unrolled computations in sequence. Or for a vectorized version, a fully unrolled tile would be 4 sequential calls to vectorized intrinsics that each do 4 operations in parallel (e.g., AVX intrinsics each do 4 `float` operations in parallel).

Example: Tiled Matrix Multiplication: Tiling techniques are widely used to improve the efficiency of MatMul's and thereby get better throughput of tensor calculations from a GPU. Matrix multiplication is a good candidate for this optimization because it has *O(n^3)* arithmetic calculations, but uses only *O(n^2)* data. Hence, a naive matrix multiplication algorithm that doesn't address cache locality will re-load the same data into memory many times, whereas a tiled algorithm can reuse the same data more efficiently.

A tiled version of MatMul processes "tiles" or "blocks" of each matrix one at a time (i.e., small square or rectangular sections), with the aim of keeping small subparts of the matrix in the memory cache while they are processed. The algorithm progresses across the matrix a tile/block at a time, rather than scanning all the way down one dimension (row or column). The same number of multiplication operations are performed as a non-tiled MatMul, but data locality and cache freshness should improve the overall speed.

Loop Fission

Loop fission is an optimization that is the opposite of loop fusion. Instead of fusing two loops into one, we take one loop and split parts of it into two loops. Loop fission also been called other names such as "loop splitting" or "loop distribution."

Loop fission can be more efficient for parallel execution (e.g., vectorization for GPUs), but is often slower for sequential execution. Whereas loop fusion aims to remove the overhead of one of the loops, loop fission tolerates an increased loop overhead in return for simpler loop bodies that can be parallelized. The kernel optimization of "kernel fission" is based on loop fission, and loop fission is one technique used to achieve vectorization for GPUs.

The main reason to use loop fission is hardware acceleration via loop parallelization. A complicated single loop can often run faster if split into two simpler loops, when hardware acceleration can be accessed. This is true even if the two resulting loops must run sequentially, because the iterations of each loop are parallelized, but there's a double benefit if the two whole loops can also run in parallel.

Example: Loop Fission in BatchNorm: A good example arises in part of the code for batch normalization. Each element of the vector needs to have two operations performed on it: subtract the mean (re-centering) and multiply by a variance factor (re-scaling).

The naive implementation of the second half of BatchNorm looks like this:

```
float denom = sqrtf(varc + eps); // Scale factor
for (int i = 0; i < n; i++) {
    // Normalize: re-center and scale
    v[i] = (v[i] - fmean) / denom;
}
```

This is difficult to hardware accelerate because it's unlikely that there's a combined "subtract-and-then-divide" operation to apply to all elements of a vector in parallel. The first point is that maybe there's an "add-and-then-multiply," in which case we can use the negative of the additive factor and the reciprocal of the scaling factor. However, assuming there's not, loop fission can be used to split the single complicated loop into two sequential loops.

```
float negmean = -fmean;  // Use negative for addition
float denom = sqrtf(varc + eps); // std. deviation
float recip = 1.0f / denom;  // reciprocal multiply
// Loop 1: Re-center using mean
aussie_vector_add_scalar(v, n, negmean);
// Loop 2: Re-scale by factor
aussie_vector_multiply_scalar(v, n, recip);
```

Each of the two loops is now easy to hardware accelerate, because they are both very simple vector operations: "multiply-by-scalar" and "add-scalar." Every platform is likely to have hardware acceleration APIs for those simpler operations. So, to summarize, we got an explosive boost to hypersonic rocket speed using atomic operations with loop fission. Isn't that just the bomb?

Loop Reversal

Loop reversal is the optimization of making the loops go backwards. It does the same number of arithmetic operations, but in reverse order, so there is no change in the total arithmetic operations.

This goal is a speedup by "looping down to zero" with a faster loop test, but it is often a de-optimization even for sequential execution. Typical CPU processors rely on ascending order of memory accesses for predictive cache pipelining, and reverse array access is a worst case for that.

Loop reversal is also not a useful parallelization method in itself. Vectorization for GPU computation doesn't really work in reverse.

However, reversing a loop can sometimes be useful as an initial transformation on nested loops if reversing the inner loop's direction allows another followup loop vectorization technique.

Example: Reversed Vector Dot Product: Loop reversal can be used on vector dot product, as below, but it probably shouldn't be. Here's the basic idea:

```
float vecdot_reverse(float v1[], float v2[], int n)
{
    float sum = 0.0;
    for (int i = n - 1; i >= 0; i--) {
        sum += v1[i] * v2[i];
    }
    return sum;
}
```

Note that there are several coding pitfalls to avoid. The loop variable "`i`" cannot be "`unsigned`" or "`size_t`" type, because the test "`i>=0`" would never fail, creating an infinite loop. Also, the reversed loop needs to start at "`n-1`" and must use "`i>=0`" (not "`i>0`") to avoid an off-by-one error. The above code also craters for "`n<=0`" and needs a safety test.

Loop Code Motion

Loop code motion is moving loop-invariant code from inside the loop body to the pre-initialization code for the loop. Any code that has the same value should not be performed inside the loop body. Instead, it should be pre-calculated before the loop, and stored in a temporary variable. This is sometimes called "hoisting" the code out of the loop.

Example: Loop Code Motion: One common example of unnecessary recalculation of loop-invariant values is in the loop test. The code in the Boolean test for the loop is actually part of the loop body.

An example of code that re-calculates the loop limit:

```
for (i = 0; i < vec.num_elements(); i++) {
   // ...
}
```

The "num_elements" call is probably loop-invariant, assuming the vector doesn't change size during processing. Maybe the "num_elements" function is declared "inline" and the C++ compiler will fix it anyway. Nevertheless, this is a candidate for loop code motion, using a temporary variable instead:

```
int n = vec.num_elements();  // Loop-invariant value
for (i = 0; i < n; i++) {
   // ...
}
```

Loop Distribution

Loop distribution is type of loop code motion that creates two loops from a single loop that contain an "if" statement. The hoisted code is a conditional test. Some early papers in the 1990s called it "loop unswitching." Some papers use the term "loop distribution" with the different meaning of splitting a loop into two loops, which we call "loop fission."

The goal of loop distribution is to move an "if" test out of the loop body, by creating two loops, and ends up creating two separate loops on two pathways. This sounds similar to loop fission, but loop distribution is a more general optimization that doesn't require parallelization to get a speed improvement (whereas loop fission does). Instead, loop distribution gets a benefit in ordinary sequential execution because it moves the if-test computation out of the loop body to a once-only pre-initialization test (i.e., "hoisted"). Note that only one of the two loops is executed each time, and these two loops are never executed in parallel, so this technique is not really a type of loop fission.

Example: Loop Distribution: Here's a dummy example of implementing an "add-or-subtract" function using a passed-in Boolean flag.

```
void aussie_vector_addition_slow(
    float v[], int n,
    bool do_add, float scalar)
{
    for (int i = 0; i < n; i++) {
        if (do_add)
            v[i] += scalar; // Add
        else
            v[i] -= scalar; // Subtract
    }
}
```

The problem is that the test "`if(do_add)`" is computed for every loop iteration, and yet "`do_add`" is a loop-invariant flag variable. The faster version is to use loop distribution to move the `if`-test into the loop initialization, and then split the two pathways inside the loop to instead have two separate loops. Here's the faster version:

```
void aussie_vector_addition_loop_distribution(
    float v[], int n,
    bool do_add, float scalar)
{
    if (do_add) { // Add scalar
        for (int i = 0; i < n; i++) {
            v[i] += scalar;  // Add
        }
    }
    else {  // Subtract scalar
        for (int i = 0; i < n; i++) {
            v[i] -= scalar; // Subtract
        }
    }
}
```

This example is still far from optimal. For starters, it should be using pointer arithmetic rather than array indices.

Loop Reordering

Loop reordering is the general class of optimizations that involves reordering loops or their iterations. In complex algorithms, there are many loops, and many ways for nesting them, or running them in sequence. Such optimizations can involve changing the ordering of two sequential loops or two nested loops.

The reordering optimization to reverse the inner and outer nested loops is more precisely called "loop interchange." A single loop can also be reordered with "loop reversal."

Loop reordering is an optimization that doesn't actually reduce the total computations, because it always executes the same number of iterations as the original version. However, loop reordering may have several benefits:

- Vectorization. Putting the loop in a different order may make it more vectorizable, or may allow other loop transformations to be applied before vectorization.

- Data locality. Reordering the loops may improve data locality and cache access speed by doing the operations in a different order. This reduces the cost of accessing the data into memory (or low-level caches), rather than the cost of the arithmetic. It is therefore related to memory/dataflow optimizations and pipelining optimizations.
- Reduced loop overhead. Both loop interchange and loop reversal can reduce the general overhead of loop testing. Loop interchange allows the shorter loop to be on the outside. Loop reversal allows "looping down to zero" which reduces overhead.

Loop Iterator Strength Reduction

Loop strength reduction is the arithmetic optimization of "strength reduction" applied to loop iteration variables. For example, strength reduction aims to replace multiplication with addition. Consider this loop:

```
for (int i = 0; i < n; i++) {
    a[i] = 10 * i;
}
```

This can be optimized to change the multiplication into an incremental addition:

```
for (int i = 0, x = 0; i < n; i++) {
    a[i] = x;
    x += 10;
}
```

Note that the loop strength reduction optimization isn't a good choice for loop parallelization. Although it would be desirable to change a vectorized multiplication to addition, this optimization has changed to an incremental algorithm. This makes each loop iteration dependent on the prior one, with the results dependent on the previous computation, so they cannot be done in parallel.

Loop Coalescing

Loop coalescing is a loop optimization that involves flattening two nested loops into one non-nested loop. Typically, loop coalescing will still operate on a 2-dimensional array, whereas flattening both the nested loops and the array is called "loop collapsing."

As a dummy example, consider a matrix initialization via nested loops:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        arr[i][j] = 0.0f;
    }
}
```

Loop coalescing involves changing to a single loop, but still using two indices i and j, which are calculated from the main linear index.

```
int maxx = n * m;
for (int x = 0; i < maxx; x++) {
    int i = x / n;
    int j = x % m;
    arr[i][j] = 0.0f;
}
```

The benefit in speed from loop coalescing can arise by simplifying the loop, which makes it easier to parallelize via hardware acceleration, and also maybe a different data access pattern which might improve data locality and cache freshness.

This optimization is not always possible, as nested loop logic is often quite complicated, and flattening a nested loop may actually worsen data locality in many instances. However, the linear nature of a simple loop can make the code to send off chunks to a GPU much easier.

Loop Collapsing

Loop collapsing is closely related to loop coalescing, since both aim to flatten nested loops, but loop collapsing is a special situation where the array is also flattened to one dimension.

Consider a matrix initialization via nested loops over a 2-dimensional array:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        arr[i][j] = 0.0f;
    }
}
```

The loop collapsed version has one big loop over a different one-dimensional array:

```
int maxx = n * m;
for (int x = 0; x < maxx; x++) {
    arr2[x] = 0.0f;
}
```

This loop transformation to a single loop is obviously more amenable to vectorization.

Loop Peeling

Loop peeling is a type of loop unrolling that involves unraveling only the first few iterations of a long loop. This is also similar to "loop splitting" with two sections, where the first section is over the early range, and the second range is the main section of all remaining iterations.

Loop peeling is beneficial to the overall loop efficiency if there is code in the loop body that is only required for one or two early iterations, which can then be removed from the main loop body. Similarly, there can be benefit in unraveling the last few iterations of a loop, which is a similar technique.

One common case of loop peeling is when the first iteration is different from the rest, so peeling off a single iteration is valuable.

```
for (int i = 0; i < n; i++) {
    arr[i] = (i == 0) ? 0.0f : 1.0f;
}
```

In this case, we can peel off the first "`i==0`" iteration into a single unrolled instruction, and change the main loop to start at 1. This is also a trivial form of "loop distribution," where we are hoisting an "`if`" conditional test out of the loop. The new code becomes:

```
arr[0] = 0.0f;  // Peeled
for (int i = 1 /*not 0*/ ; i < n; i++) {
    arr[i] = 1.0f;
}
```

This peeled version is faster in terms of both sequential or parallel execution. The loop body has less computation and is also more amenable to vectorization.

Loop Splitting

Loop splitting refers to splitting the sequential iterations of a loop into two loops, which each perform part of the original loop's iterations. Loop splitting is closely related to "loop sectioning" ("strip mining"), but often relates to more complex arithmetic in the loop body.

Note that "loop peeling" is a special case of loop splitting where the first section is a small range of a few initial iterations, but these few iterations are unrolled rather than looped.

Loop splitting takes a single loop and transforms it into at least two "split-out" loops, one for the early iterations, and one for the remainder. However, loops can also be split out into more than two loops.

In loop splitting, each split-out loop is shorter than the original loop. Unlike loop fission, the two loops operate over different subportions of the iterator variable range, executing the same number of total iterations, rather than double iterations as in loop fission.

Example: Loop Splitting: Here's some example code to "sqrtize" a vector, using a cached optimization for the numbers up to 100.

```
void aussie_vector_do_sqrt(float v[], int n)
{
    for (int i = 0; i < n; i++) {
        if (i < 100) { // Fast cases
            v[i] = aussie_sqrt_optimized(v[i]);
        }
        else {  // General case
            v[i] = sqrtf(v[i]);
        }
    }
}
```

However, we can use loop splitting to split this big loop into two shorter disjoint ranges. Instead of 0..n-1, we do 0..99, and then 100..n-1.

Each loop is over part of the range, and has a simpler loop body. Note that this code fails with an array bounds violation for small values of n less than 100.

```
void vector_do_sqrt_loop_splitting(float v[], int n)
{
    for (int i = 0; i < 100; i++) { // Fast cases
        v[i] = aussie_sqrt_optimized(v[i]);
    }
    for (int i = 100; i < n; i++) { // General cases
        v[i] = sqrtf(v[i]);
    }
}
```

The loop splitting optimization is beneficial if the loop body has different sections of code that only relate to a subset of the iterator range. Hence, the loop bodies of the two loops can be reduced to execute less code. Overall, there is still the same number of iterations performed in the two loops combined, but each loop performs only a proportion of the original iterations on a simpler loop body. This optimizes sequential execution and the simpler code in each loop body may make vectorization of one or both subloops easier. Furthermore, both subloops could run in parallel.

Loop Interchange

Loop interchange is an optimization of nested loops that switches the inner and outer loops. In a typical nested loop, the outer loop body and loop test is executed rarely, almost lazily, whereas the inner loop body is scrambling along in a frantic mess. Loop interchange simply switches them, reversing their roles.

Why is this an optimization? Although the same number of loop iterations still occur in total, and the newly-made inner loop body is also thrashed, various improvements can arise from reversing the iterator variables, usually to make the innermost loop the longest. Possible optimizations result from:

- Fewer outside computations. A shorter outside loop reduces the arithmetic operations of the outer loop, whereas the inner loop's number of computations is unchanged in either loop structure.
- Data locality. Another possible improvement is in data locality, which can reduce cache misses and speeds up the overall execution. Note that this benefit is not guaranteed just by switching loops, and sometimes loop interchange can worsen data locality; careful analysis is needed.
- Inner loop vectorization. Another important possibility is that reversing nested loops can create opportunities to apply other loop optimizations to the new inner loop, notably to vectorize the inner loop.

Shortest loop outside, longest innermost loop: One of the considerations of loop interchange is the optimization of putting the shortest loop on the outside, and making the innermost loop with the longest range of iterations. This is an optimization for both sequential or parallel execution. For sequential execution, there is less overhead from the outer loop, because it is shorter. For parallelization, there is improved vectorization of the inner loop, which now has a longer range.

Consider this example:

```
for (int i = 0; i < 1000; i++) {
    for (int j = 0; j < 50; j++) {
        // ...
    }
}
```

The current loop nesting has the longest loop (to 1000) on the outside, and the shorter loop (to 50) as the innermost loop. Loop interchange simply makes it the reverse nesting:

```
for (int j = 0; j < 50; j++) {
    for (int i = 0; i < 1000; i++) {
        // ...
    }
}
```

Considering sequential execution, the inner loop body is executed the same number of times, so there's no difference. This also includes the inner loop's conditional test and incrementer, which are different variables in the two examples, but also execute the same number of times (50,000 times). However, consider the different outer loops. The first example is 1000 iterations, whereas the second example's outer loop is only 50 times. Hence, the loop reordering optimization of "shortest outer loop" and "longest innermost loop" has saved 950 of the outer loop's calculations (i.e., loop test and incrementer). Any extra code that's in the outer loop, either before or after the inner loop, would also be executed fewer times.

There is also an advantage for vectorization. In the first example, we could possibly have 1000 vectorized operations of data size 50. In the interchanged loops, there are 50 operations on vectors size 1000. Hence, there is more opportunity for much larger vectorization gains in the second format with the longest inner loop.

Loop Sentinel

Loop sentinels are an optimization that removes the overhead of checking an array index or pointer scanning an array or pointer chain. The technique does this by adding a pretend extra element onto the end of the array, in a way that we can pretend to succeed. And since we're guaranteed to always succeed, we don't need to check for failure while scanning the loop.

This technique is not particularly useful for vectorization, but is quite powerful for long sequential scanning of arrays. It also has the downside of requiring at least one writeable array element, so it cannot run on read-only arrays.

Example: Check Vector Negatives: Here's a sentinel with a dummy in `v[n]`:

```
bool vector_has_negative_sentinel(float v[], int n)
{
     v[n] = -99.0;  // Dummy negative (BUG!)
     int i = 0;
     for ( ; /*GONE!*/; i++) {
         if (v[i] < 0.0) break;  // Found negative
     }
     if (i == n) return false;  // Fake success
     return true;  // Found a negative (for real)
}
```

However, this is actually buggy, since "`v[n]`" is potentially an array overflow. A better version can manipulate the last valid element "`v[n-1]`" instead of modifying "`v[n]`". Then, we have to remember to fix it before we leave town. We also have to check the last vector element that we temporarily overwrote wasn't a real success.

```
bool vector_has_negative_sentinel2(float v[], int n)
{
    float save = v[n - 1];  // Save it!
    v[n - 1] = -99.0;  // Dummy negative at end
    int i = 0;
    for ( ; /*GONE!*/; i++) {
        if (v[i] < 0.0) break;  // Found negative
    }
    v[n - 1] = save;  // Restore it!
    if (i == n - 1) { // At the dummy (fake success)
        if (save < 0.0) return true; // Must check
        return false;
    }
    return true;  // Found a negative (for real)
}
```

Loop Strip Mining (Loop Sectioning)

Loop strip mining is a loop optimization that scans or "mines" various "strips" of an array. It is related to "loop tiling" on arrays in two dimensions, but strip mining only applies to processing one-dimensional arrays. Loop strip mining is also called "loop sectioning" because it breaks an array up into sections that are operated on.

For a basic example, consider a simple array initialization:

```
for (int i = 0; i < n; i++) {
    arr[i] = 0.0f;
}
```

Let's assume we can parallelize this with 16 elements at a time (e.g., 512 bits total parallel processing, which is 16 separate 32-bit `float` variables). So, we want to process "strips" of length 16. For simplicity, let us assume that `n` is divisible exactly by 16, so there's no leftover work after the main loop.

```
for (int i = 0; i < n; i += 16) {
    // Initialize arr[i]...arr[i+15] in parallel
}
```

Obviously, this is a dummy example, where `memset` would do better for zeroing the array. Also, this really looks exactly like "vectorization" to me, where we are vectorizing 512 bits at a time (16 `floats`), and indeed the research mentions vectorization as one application.

But loop strip mining and vectorization are not exactly the same techniques, because loop strip mining is a more general idea with other applications.

Loop Spreading

Loop spreading is an optimization of two non-nested sequential loops that have different iteration ranges. Typically, this refers to where the end ranges differ significantly. If the loop ranges only differ by an off-by-one issue, then only loop normalization is required.

Loop spreading modifies one of the loops, so that part of this loop fully overlaps with the other loop (i.e., ideally one loop "spreads out" further to match the other loop's end bounds). Hence, after loop spreading has occurred, this subloop can be fused with the other loop, and possibly parallelized.

The remaining iterations that are not overlapping then have to be addressed in a followup partial loop (only for one of the loops).

Loop spreading mainly enables loop fusion as a followup optimization. For using loop fission on the two loops, it is not necessary to do loop spreading, since the two loops are already split apart, and each loop could already potentially be vectorized independently.

Loop Normalization

Loop normalization is not directly an optimization, but is a preliminary loop transformation that can make further loop optimizations easier. Followup optimizations might be to fuse the two loops with loop fusion, or to parallelize each loop, such as with loop fission or vectorization.

The goal of loop normalization is to make the loop iteration variables act across the same range. This applies to two sequential loops, rather than nested loops.

Hence, loop normalization is needed when two loops in sequence are starting at different offsets (e.g., one is `i=1` and another starts at `i=0`), or are finished at different endpoints (e.g., `n` versus `n-1`).

If two loops have the same number of computations, but with different ranges, then one loop can be changed with an offset. For example, these loops differ with ranges `0..n-1` and `1..n`:

```
for (int i = 0; i < n; i++) a[i] = 0;
for (int j = 1; j <= n; j++) b[j] = 0;
```

These can be adjusted to the same ranges with a "`j+1`" index offset, as follows:

```
for (int i = 0; i < n; i++) a[i] = 0;
for (int j = 0; j < n; j++) b[j+1] = 0;
```

If the two loops have a different number of iterations, typically off by 1 or 2, then "loop peeling" can be used to unroll and split off one or two iterations and shorten the longer loop, so that both loops have the same number of iterations over the same range.

For example, in this example, one loop is `0..n-1` and another is `0..n`:

```
for (int i = 0; i < n; i++) a[i] = 0;
for (int j = 0; j <= n; j++) b[j] = 0;
```

The way to normalize the loop ranges is to "peel" off the last iteration of the "`j`" loop:

```
for (int i = 0; i < n; i++) a[i] = 0;
for (int j = 0; j < n; j++) b[j] = 0;
b[n] = 0;  // Peeled
```

This example has peeled the longer loop to make it shorter.

An alternative would be "loop spreading" to lengthen the shorter loop, such as by adding an extra padding element into the array.

Normalizing two loops doesn't change the number of arithmetic computations. However, once two loops have normalized ranges, it becomes easier to see opportunities for further optimizations such as loop fusion or loop fission.

Loop Skewing

Loop skewing is a somewhat mind-bending method to change nested loops to make them more parallelizable. This technique applies to two nested loops, where the inner loop is difficult to parallelize because of a dependency on the outer loop.

The performance advantage from loop skewing is not directly its usage, but because skewing changes then make possible other loop optimizations, especially loop interchange, which reorders the inner and outer loop.

The loop skewing solution is far from obvious. The range bounds of the inner loop are changed by "skewing" them by a factor based on the outer loop variable. And then, by some magical potion, this somehow breaks the dependence on the outer loop, and then the inner loop can run fast on a GPU. Who knew?

As a simplistic example, consider two nested loops:

```
for (int i = 0; i < 1000; i++) {
    for (int j = 0; j < 50; j++) {
        arr[i][j] = something;
    }
}
```

We can skew the inner loop by adding a skew factor based on the outer loop variable (e.g., "`i`" or "`i+1`" or something similar). Add this skew factor to the ranges of `j`, but then subtract the skew factor ("`i`") from any usages of the index "`j`" inside the inner loop's body.

```
for (int i = 0; i < 1000; i++) {
    for (int j = i; j < 50 + i; j++) {
        arr[i][j - i] = something;
    }
}
```

Hence, `j` has changed from the range (`0...50`) to the skewed range (`i...i+50`), by adding the skew factor "`i`" to the start and end. The use of "`j`" in the inner loop body has changed from "`j`" to "`j-i`" (i.e., subtracting the skew factor "`i`").

The result is a kind of skewed and "triangular" shape of `i` and `j` indices, but the actual arithmetic calculations are unchanged.

This newly skewed code isn't any faster, does exactly the same calculations on the 50,000 elements of array `arr`, and indeed is actually worse because of the extra "`50+i`" and "`j-i`" computations.

However, in some cases, doing this weird skewing transformation then allows us to follow up with a loop interchange optimization, switching the inner and outer loops. And I'm not even going to pretend to understand this, but there are situations where the non-skewed inner loop cannot be vectorized or interchanged, but after we've skewed the loop, then we can interchange it, and then we get via hocus pocus a different inner loop that can then be vectorized.

Hopefully, the GPU knows what's going on.

References

1. Allen, F. E., and Cocke, J. 1972. *A catalogue of optimizing transformations*. In Design and Optimization of Compilers, Prentice-Hall, Englewood Cliffs, N.J., pp. 1–30.
   PDF: https://www.clear.rice.edu/comp512/Lectures/Papers/1971-allen-catalog.pdf
2. D. F. Bacon, S. L. Graham, and O. J. Sharp. 1994. *Compiler transformations for high-performance computing* . ACM Computing Surveys 26, 4 (1994), 345–420. https://dl.acm.org/doi/10.1145/197405.197406,
   PDF: https://people.eecs.berkeley.edu/~fateman/264/papers/bacon.pdf (Paper with extensive coverage of numerous compiler auto-optimizations of program code.)
3. Paul Bilokon, Burak Gunduz, 8 Sep 2023, *C++ Design Patterns for Low-latency Applications Including High-frequency Trading*, https://arxiv.org/abs/2309.04259,
   Code: https://github.com/0burak/imperial_hft
4. Eric LaForest, March 19, 2010, *Survey of Loop Transformation Techniques*, ECE 1754, http://fpgacpu.ca/writings/SurveyLoopTransformations.pdf
5. B Qiao, O Reiche, F Hannig, 2019, *From loop fusion to kernel fusion: A domain-specific approach to locality optimization*, 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), https://ieeexplore.ieee.org/document/8661176 (Theory of loop fusion generalized to graph kernel fusion for image processing.)
6. Kathryn S. McKinley, Steve Carr, Chau-Wen Tseng, 1996, *Improving data locality with loop transformations*, ACM Transactions on Programming Languages and Systems, Volume 18, Issue 4, pp 424–453, https://dl.acm.org/doi/10.1145/233561.233564
7. B Blainey, C Barton, JN Amaral, 2002, *Removing impediments to loop fusion through code transformations*, International Workshop on Languages and Compilers for Parallel Computing, LCPC 2002: Languages and Compilers for Parallel Computing pp 309–328, https://link.springer.com/chapter/10.1007/11596110_21

13. Parallel Vectorization

What is Vectorization?

Vectorization is the name given to transforming a software loop from running sequentially on an array of data to performing the same computation fully in parallel, by sending the data to a GPU or CPU SIMD extensions. This is a powerful way to optimize the processing of contiguous data structures such as arrays and vectors.

Vectorization uses techniques from loop optimizations to transform loops into faster parallelizable versions, such as "unrolling" a loop into all its element-wise actions, and loop distribution (also called "loop sectioning"), which breaks the array into segments that are the right size to fit in parallel into your GPU or CPU SIMD extensions.

In theory, a good optimizing compiler can do vectorization optimizations automatically for simple loops, but often you have to do it yourself.

A powerful way to do vectorization of contiguous data processing is to use the AVX SIMD instructions for CPU-based parallelism. The AVX intrinsics are C++ built-in functions that wrap around SIMD instruction codes in the x86 instruction set.

The basic AVX intrinsics are 128-bits (4 `float` values of size 32-bits), AVX-2 is 256 bits (8 `float` values), and AVX-512 is 512 bits (surprise!), which is 16 `float` numbers. The upcoming AVX-10 (announced in July 2023) is also 512 bits, but with extra capabilities.

Obviously, since the largest number of floating-point values that can be parallelized is 16, the AVX SIMD intrinsics cannot fully vectorize a larger vector with many `float` values, such as an AI model with dimension 1024. Instead, we can use AVX intrinsics on segments of vectors, and thereby vectorize chunks of the right size to get a speedup.

Example: AVX Vectorized Dot Product

Here is the basic non-vectorized dot product computation without any optimization attempts.

```
float vecdot_basic(float v1[], float v2[], int n)
{
    // Basic FLOAT vector dot product
    float sum = 0.0;
    for (int i = 0; i < n; i++) {
        sum += v1[i] * v2[i];
    }
    return sum;
}
```

To use AVX to vectorize it, we need to unroll the loop first. Here's a simple vector dot product with its inner loop unrolled 4 times. This version assumes that `n` is a multiple of 4 rather than handling odd cases:

```
float vecdot_unroll4(float v1[], float v2[], int n)
{
    // Loop-unrolled Vector dot product
    if (n % 4 != 0) {
        aussie_assert(n % 4 == 0);
        return 0.0; // fail
    }
    float sum = 0.0;
    for (int i = 0; i < n; ) {
        sum += v1[i] * v2[i]; i++;
        sum += v1[i] * v2[i]; i++;
        sum += v1[i] * v2[i]; i++;
        sum += v1[i] * v2[i]; i++;
    }
    return sum;
}
```

So, now we can change those 4 unrolled multiplications into one AVX computation of the vector dot product of 4 `float` numbers.

```
#include <intrin.h>

float vecdot_unroll_AVX1(float v1[], float v2[],int n)
{
    // AVX-1 loop-unrolled (4 floats) dot product
    if (n % 4 != 0) {
        aussie_assert(n % 4 == 0);
        return 0.0; // fail
    }
    float sum = 0.0;
    for (int i = 0; i < n; i += 4) {
       // AVX1: Vector dot product of 2 vectors
       //  ... process 4x32-bit floats in 128 bits
       __m128 r1 = _mm_loadu_ps(&v1[i]); // Load
       __m128 r2 = _mm_loadu_ps(&v2[i]);
       __m128 dst = _mm_dp_ps(r1, r2,0xf1); // Dot prod
       sum += _mm_cvtss_f32(dst);
    }
    return sum;
}
```

This basic AVX sequence of code to do the 4 float dot product has been analyzed in a separate chapter. The main dot product computation is "`_mm_dp_ps`" which is an AVX intrinsic and multiplies 4 pairs of 32-bit `float` numbers, and then sums them, all in one call to an intrinsic. Note that the loop now iterates 4 at a time through the array of `float` values (i.e., "`i+=4`") and then the AVX intrinsic does the rest.

Here's the benchmark analysis showing that the AVX-vectorized version is more than twice as fast:

```
FLOAT Vector dot product benchmarks:
Time taken: Vecdot basic: 2805 ticks (2.81 seconds)
Time taken: Vecdot AVX1 unroll (4 floats, 128-bits):
  1142 ticks (1.14 seconds)
```

Fused Multiply-Add (FMA) in AVX-2. The AVX-2 FMA intrinsic takes 3 vectors, each of size 256-bits, multiplies two of them pair-wise, and then adds the third vector. Both the multiplication and addition are done in element-wise SIMD style. At first blush this sounds like doing a vector multiply and then adding a "bias" vector, and hence doesn't sound like a good optimization for the vector dot product. The SIMD pairwise multiplication is the first step of dot products, but the vector addition seems the opposite of what we want, which is "horizontal" addition of the products that result from the multiplications.

The default idea is doing a dot product of 8 `float` values, and then another one, and then adding each individual sum at the end. With that idea, the vertical addition in FMA is not what we want, and it looks like using SIMD multiplication and an extra horizontal addition would be better than using a single FMA intrinsic. However, we can make like Superman III...

Reverse it!

If you think about FMA not as a multiplication and then addition, but as "adding multiplications" in the reverse order, then there is a eureka moment: put the addition first. The idea is that we can maintain a vector of running sums, and then only do a single horizontal addition at the very end. It's kind of mind-bending, but here's the code:

```
float vecdot_FMA_unroll_AVX2(float v1[], float v2[], int n)
{
   // AVX2 vecdot using FMA (Fused Multiply-Add) primitives
   if (n % 8 != 0) {
       aussie_assert(n % 8 == 0);
       return 0.0; // fail
   }
   __m256 sumdst = _mm256_setzero_ps(); // Zero accumulators
   for (int i = 0; i < n; i += 8) {
       // AVX2: process 8x32-bit floats in 256 bits
       __m256 r1 = _mm256_loadu_ps(&v1[i]); // Load floats
       __m256 r2 = _mm256_loadu_ps(&v2[i]);
       sumdst = _mm256_fmadd_ps(r1, r2, sumdst); // FMA 3 vect
   }
   // Add the final 8 accumulators manually
   float* farr = (float*)&sumdst;
   float sum = farr[0] + farr[1] + farr[2] + farr[3]
             + farr[4] + farr[5] + farr[6] + farr[7];
   return sum;
}
```

How does this work? Well, we declare "`sumdst`" as a vector of 8 `float` numbers that maintains the 8 parallel accumulators, which is first initialized to all-zeros via the "`_mm256_setzero_ps`" intrinsic. In the main loop, we use "`sumdst`" to maintain a running sum in all 8 of those parallel accumulators across multiple segments of the vector. One accumulator sums the products in array indices 0,8,16,..., and the next accumulator sums the products for indices 1,9,17,... We use the FMA intrinsic ("`_mm256_fmadd_ps`" in AVX2) to do the SIMD multiplication, but rather than trying to add the 8 resulting products together, we add each product to a separate accumulator. This works very neatly, because the AVX-2 FMA intrinsics does this all in SIMD parallelism with the combined FMA intrinsic. Only at the very end, after the main loop, we do a horizontal add of the 8 parallel accumulators to get the final sum.

This idea works surprisingly well, and is gratifying since I couldn't get the AVX-2 256-bit version with the dot product "_mm256_dp_ps" intrinsic to run correctly on 8 float values. Here's the benchmarking, which shows that AVX-2 using FMA on 8 float values in parallel runs much faster than the AVX1 unrolled vector dot product using the intrinsic "_mm_dp_ps" with 4 float values.

```
FLOAT Vector dot product benchmarks: (N=1024, Iter=1000000)
Vecdot basic: 2961 ticks (2.96 seconds)
Vecdot AVX1 unroll (4 float, 128-bit): 1169 ticks (1.17 sec)
Vecdot AVX1 FMA (4 float, 128-bit): 1314 ticks (1.31 seconds)
Vecdot AVX2 FMA (8 float, 256-bit): 783 ticks (0.78 seconds)
```

Note that we can improve on the horizontal addition at the very end. The example code just uses basic C++ with 7 additions and 8 array index computations. Instead, this last computation should really use some AVX "hadd" intrinsics instead (it needs 3 calls to horizontal-pairwise add 8 float values).

Example: AVX Vector Sum Reduction

Let us suppose we need to calculate the sum of all the elements of a vector. This is a "reduction" that has dimensions "vector-to-scalar." Here is a basic naive C++ version without any optimizations:

```
float vector_sum(float v[], int n)  // Summation
{
    float sum = 0.0;
    for (int i = 0; i < n; i++) {
        sum += v[i];
    }
    return sum;
}
```

AVX vector reductions have some issues in the early releases. Although AVX has SIMD instructions to add two vectors in parallel, it struggles to do a "reduction" operation like this. AVX and AVX-2 do have "horizontal add" ("hadd") intrinsics, but these only do pairwise additions within the single vector, rather than adding all elements. AVX-512 has a "reduce add" intrinsic ("_mm512_reduce_add_ps") for horizontally adds 16 float numbers, which works a lot better.

For AVX and AVX-2, are we stuck with doing multiple calls to the pairwise "hadd" intrinsics? No, there's a non-obvious way to use the "vertical add" intrinsics in parallel. We can do "in parallel" squared. It's almost like we're doing math inside a computer.

The trick is to use the AVX registers as a set of 4 parallel accumulators (AVX 128 bits) or 8 parallel accumulators (AVX-2's 256 bits). In this way, we can defer the "hadd" until the very end, and since it's not in the critical loop, its performance hardly matters. Here's the code for AVX-1 with 128-bit registers:

```
float vector_sum_AVX1(float v[], int n)
{
    // Summation (horizontal) of a single vector
    if (n % 4 != 0) {  // Safety
        aussie_assert(n % 4 == 0);
        return 0.0; // fail
    }

    __m128 sumdst = _mm_setzero_ps(); // Zero accums
    for (int i = 0; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]); // Load float
        sumdst = _mm_add_ps(r1, sumdst); // SUM=SUM+V
    }

    // Add the final 4 accumulators manually
    float* farr = sumdst.m128_f32;
    float sum = farr[0] + farr[1] + farr[2] + farr[3];
    return sum;
}
```

The AVX-2 version is faster, because it processes 8 `float` values at a time. This uses the same strategy of 8 parallel accumulators and a loop unrolling factor of 8 (i.e., the loop incrementer is now "`i+=8`"). Here's the C++ code:

```
float aussie_vector_sum_AVX2(float v[], int n)
{
    if (n % 8 != 0) { // Safety check (no extra cases)
      aussie_assert(n % 8 == 0);
      return 0.0; // fail
    }
    __m256 sumdst = _mm256_setzero_ps(); // Clear acc
    for (int i = 0; i < n; i += 8) {
      __m256 r1 = _mm256_loadu_ps(&v[i]); // Load 8
      sumdst = _mm256_add_ps(r1, sumdst); // SUM=SUM+V
    }
    // Add the final 8 accumulators manually
    float* farr = sumdst.m256_f32;
    float sum = farr[0] + farr[1] + farr[2] + farr[3]
              + farr[4] + farr[5] + farr[6] + farr[7];
    return sum;
}
```

I've been lazy not bothering to optimize the final horizontal addition. A small extra speedup is probably available using the "hadd" intrinsics 3 times in a row to drop it down from 8 accumulators to a single float. If this was AVX-512, we could use the horizontal reduction "`_mm512_reduce_add_ps`" intrinsic for summation at the end (for adding 16 partial sums of type float).

Loop Peeling Optimization: Another inefficiency with these AVX addition routines it that they needlessly perform an addition with zero in the first iteration. Effectively, we need to do "loop peeling" to handle the first loop iteration differently. This is the slow first iteration of AVX2 vector sum:

```
__m256 sumdst = _mm256_setzero_ps(); // Clear 8
for (int i = 0; i < n; i += 8) {
    // ...
}
```

Loop peeling says to replace the initialization with zero with loading the first 8 values from the vector. The loop starts its first iteration at `i=8` instead of `i=0`, skipping what had been the first addition:

```
// Get first 8 values
__m256 sumdst = _mm256_loadu_ps(&v[0]);
for (int i = 8 /*not 0!*/; i < n; i += 8) {
    // ... same
}
```

AVX Vector Max and Min Reductions

The need to find a minimum or maximum of a vector's elements is similar to a summation reduction. Again, AVX1 and AVX2 don't have proper "reduction" intrinsics for `max` or `min`, but we can compute them in parallel by keeping a running `min` or `max` value of 4 or 8 `float` values (i.e., analogous to parallel accumulators when doing summation).

The AVX intrinsics are:

- MIN: `_mm_min_ps`, `_mm256_min_ps`
- MAX: `_mm_max_ps`, `_mm256_max_ps`

Here is the AVX1 version of MAX vector reduction:

```
float aussie_vector_max_AVX1(float v[], int n)
{
    // Maximum (horizontal) of a single vector
    assert(n % 4 == 0);// Safety check (extra cases)
    __m128 sumdst = _mm_loadu_ps(&v[0]); // Init value
    for (int i = 4 /*not 0*/; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]); // Load float
        // dst = MAX(dst, r1)
        sumdst = _mm_max_ps(r1, sumdst);
    }
    // Find Max of the final 4 accumulators
    float* farr = sumdst.m128_f32;
    float fmax = farr[0];
    if (farr[1] > fmax) fmax = farr[1];
    if (farr[2] > fmax) fmax = farr[2];
    if (farr[3] > fmax) fmax = farr[3];
    return fmax;
}
```

And here is the analogous AVX2 version of MAX vector reduction:

```
float aussie_vector_max_AVX2(float v[], int n)
{
    // Maximum (horizontal) of a single vector
    assert(n % 8 == 0);// Safety check (extra cases)
    __m256 sumdst = _mm256_loadu_ps(&v[0]);   // Init
    for (int i = 8/*not 0*/; i < n; i += 8) {
        __m256 r1 = _mm256_loadu_ps(&v[i]); // Load
        // dst = MAX(dst, r1)
        sumdst = _mm256_max_ps(r1, sumdst);
    }

    // Find Max of the final 8 accumulators
    float* farr = sumdst.m256_f32;
    float fmax = farr[0];
    if (farr[1] > fmax) fmax = farr[1];
    if (farr[2] > fmax) fmax = farr[2];
    if (farr[3] > fmax) fmax = farr[3];
    if (farr[4] > fmax) fmax = farr[4];
    if (farr[5] > fmax) fmax = farr[5];
    if (farr[6] > fmax) fmax = farr[6];
    if (farr[7] > fmax) fmax = farr[7];
    return fmax;
}
```

The MIN versions are very similar. They use the "min" AVX intrinsics, and the final steps use "<" not ">" operations. Here's the AVX1 version of a MIN vector reduction:

```
float aussie_vector_min_AVX1(float v[], int n)
{
    // Minimum (horizontal) of a single vector
    assert(n % 4 == 0); // Safety check (extra cases)
    __m128 sumdst = _mm_loadu_ps(&v[0]); // Init
    for (int i = 4 /*not 0*/; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]); // Load
        // dst = MIN(dst, r1)
        sumdst = _mm_min_ps(r1, sumdst);
    }
    // Find Min of the final 4 accumulators
    float* farr = sumdst.m128_f32;
    float fmin = farr[0];
    if (farr[1] < fmin) fmin = farr[1];
    if (farr[2] < fmin) fmin = farr[2];
    if (farr[3] < fmin) fmin = farr[3];
    return fmin;
}
```

This is the AVX2 version of a MIN vector reduction:

```
float aussie_vector_min_AVX2(float v[], int n)
{
    // Minimum (horizontal) of a single vector
    assert(n % 8 == 0); // Safety check (extra cases)
    __m256 sumdst = _mm256_loadu_ps(&v[0]);   // Init
    for (int i = 8/*not 0*/; i < n; i += 8) {
        __m256 r1 = _mm256_loadu_ps(&v[i]); // Load
        // dst = MIN(dst, r1)
        sumdst = _mm256_min_ps(r1, sumdst);
    }
    // Find Min of the final 8 accumulators
    float* farr = sumdst.m256_f32;
    float fmin = farr[0];
    if (farr[1] < fmin) fmin = farr[1];
    if (farr[2] < fmin) fmin = farr[2];
    if (farr[3] < fmin) fmin = farr[3];
    if (farr[4] < fmin) fmin = farr[4];
    if (farr[5] < fmin) fmin = farr[5];
    if (farr[6] < fmin) fmin = farr[6];
    if (farr[7] < fmin) fmin = farr[7];
    return fmin;
}
```

These versions are not especially optimized. AVX-512 would allow us to further vectorize to 16 `float` values. Also, the final computation of the maximum or minimum of 8 `float` numbers is far from optimal.

The AVX horizontal `min/max` intrinsics could be used (pairwise, multiple times). Or we can at least avoid some comparisons by doing it pairwise sequentially.

Here's the alternative for AVX1 minimum computation:

```
    // Find Min of the final 4 accumulators
#define FMIN(x,y)  ( (x) < (y) ? (x) : (y) )
    float* farr = sumdst.m128_f32;
    float fmin1 = FMIN(farr[0], farr[1]);
    float fmin2 = FMIN(farr[2], farr[3]);
    float fmin = FMIN(fmin1, fmin2);
    return fmin;
```

These functions can also have their main loops further improved. Other basic optimizations would include using loop pointer arithmetic to remove the index variable "`i`" and also unrolling the loop body multiple times.

Vectorized Sum-of-Squares Reduction

The sum of the square of an element of a vector has various applications in our AI Engine. Firstly, it can be used to compute the magnitude of a vector.

Secondly, the sum-of-squares is used in various normalization functions, as part of computing the variance from the sum-of-squares of the difference between values and the mean. The RMS factor in RMSNorm is also the square root of the sum-of-squares.

The method to add up the sum-of-squares for a vector reduction to a single `float` is very similar to a simple summation reduction.

The idea for AVX1 and AVX2 is to keep 4 or 8 running sum accumulators, and then add them up at the final step.

Here is the AVX1 version of sum-of-squares of a vector:

```
float aussie_vector_sum_squares_AVX1(float v[], int n)
{
    // Summation of squares of all elements
    assert(n % 4 == 0); // Safety check (extra cases)
    __m128 sumdst = _mm_setzero_ps(); // Zero accum
    for (int i = 0; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]); // Load
        __m128 sqr = _mm_mul_ps(r1, r1); // Sqr (V*V)
        sumdst = _mm_add_ps(sqr, sumdst); // SUM+=V*V
    }
    // Add the final 4 accumulators manually
    float* farr = sumdst.m128_f32;
    float sum = farr[0] + farr[1] + farr[2] + farr[3];
    return sum;
}
```

And here is the AVX2 version of sum-of-squares:

```
float aussie_vector_sum_squares_AVX2(float v[], int n)
{
    // Summation of squares of all elements
     assert(n % 8 == 0); // Safety check (extra cases)
    __m256 sumdst = _mm256_setzero_ps(); // Zero accum
    for (int i = 0; i < n; i += 8) {
        __m256 r1 = _mm256_loadu_ps(&v[i]);  // Load
        __m256 sqr = _mm256_mul_ps(r1, r1);  // (V*V)
        sumdst = _mm256_add_ps(sqr, sumdst); // SUM+=V*V
    }

    // Add the final 8 accumulators manually
    float* farr = sumdst.m256_f32;
    float sum = farr[0] + farr[1] + farr[2] + farr[3]
              + farr[4] + farr[5] + farr[6] + farr[7];
    return sum;
}
```

Various optimizations can be further applied to these versions. Like the summation reduction, these loops needlessly add zero at the first iteration, and loop peeling should be used for split out the first iteration separately. The final horizontal addition of 4 or 8 `float` values should be optimized. AVX-512 should be used for greater parallelism to 16 `float` numbers. Finally, basic loop optimizations with pointer arithmetic and loop unrolling could be applied.

Vectorized Multiply Vector by Scalar

The requirement to multiply a vector by a scalar is common when using scaling vectors. Division by a scalar is also handled by multiplying by the reciprocal (e.g., needed for Softmax). Multiplication by a scalar is amenable to vectorization because the naive C++ version is very simple:

```
void vector_multiply_scalar(float v[], int n, float c)
{
    // Multiply all vector elements by constant
    for (int i = 0; i < n; i++) {
        v[i] *= c;
    }
}
```

Loop Pointer Arithmetic. First, we can try the basic C++ optimization of pointer arithmetic:

```
void vector_multiply_scalar_pointer_arith(
     float v[], int n, float c)
{
    // Multiply all vector elements by constant
    for (; n > 0; n--, v++) {
        *v *= c;
    }
}
```

AVX1 multiply-by-scalar: There is no special scalar multiplication opcode in AVX or AVX-2, but we can populate a constant register (128-bit or 256-bit) with multiple copies of the scalar (i.e., `_mm_set1_ps` or `_mm256_set1_ps`), and we need do this only once. We can then use the SIMD multiply intrinsics in the unrolled loop section. The AVX 128-bit vector multiplication by scalar becomes:

```
void vec_mult_scalar_AVX1(float v[], int n, float c)
{
    // Vector of scalars
    const __m128 rscalar = _mm_set1_ps(c);
    for (int i = 0; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]);   // Load floats
        __m128 dst = _mm_mul_ps(r1, rscalar); // Mul scalars
        _mm_store_ps(&v[i], dst);  // cvt to float (aligned)
    }
}
```

AVX2 multiply-by-scalar: Even faster is to use 8 parallel multiplications with AVX-2's 256-bit registers. The AVX-1 version is simply changed to use the "`__m256`" type and the analogous AVX-2 intrinsics. The new code looks like:

```
void vector_mult_scalar_AVX2(float v[], int n, float c)
{
    const __m256 rscalar = _mm256_set1_ps(c);  // Scalars
    for (int i = 0; i < n; i += 8) {
        __m256 r1 = _mm256_loadu_ps(&v[i]);   // Load floats
        __m256 dst = _mm256_mul_ps(r1, rscalar); // Multiply
        _mm256_store_ps(&v[i], dst); // cvt floats (aligned)
    }
}
```

Combining AVX-2 with pointer arithmetic. Finally, we can get a small extra benefit by adding pointer arithmetic optimizations to the AVX-2 parallelized version. The new code is:

```
void aussie_vector_multiply_scalar_AVX2_pointer_arith(
     float v[], int n, float c)
{
    // Multiply all vector elements by constant
    const __m256 rscalar = _mm256_set1_ps(c);  // scalars
    for (; n > 0; n -= 8, v += 8) {
        __m256 r1 = _mm256_loadu_ps(v); // Load floats
        __m256 dst = _mm256_mul_ps(r1, rscalar); // Multiply
        _mm256_store_ps(v, dst);  // cvt to floats aligned
    }
}
```

Benchmarking results. In theory, the AVX-2 intrinsics could parallelize the computation by 8 times, but benchmarking showed that it only achieved a 4-times speedup.

```
Vector-scalar operation benchmarks (N=1024, ITER=1000000):
Vector mult-scalar C++: 1412 ticks (1.41 seconds)
Vector mult-scalar pointer-arith: 995 ticks (0.99 seconds)
Vector mult-scalar AVX1: 677 ticks (0.68 seconds)
Vector mult-scalar AVX2: 373 ticks (0.37 seconds)
Vector mult-scalar AVX2 + ptr arith: 340 ticks (0.34 sec)
```

Vectorized Add Scalar

The code to vectorize an "add-scalar" operation is almost identical to "multiply-scalar" operations, except that "add" intrinsics are used. Here is the AVX-1 version with "`_mm_add_ps`":

```
void vector_add_scalar_AVX1(float v[], int n, float c)
{
    // Add scalar constant to all vector elements
    const __m128 rscalar = _mm_set1_ps(c);  // scalars
    for (int i = 0; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]);   // Load floats
        __m128 dst = _mm_add_ps(r1, rscalar); // Add scalars
        _mm_store_ps(&v[i], dst);  // store back to floats
    }
}
```

And this is the analogous AVX-2 version using the "`_mm256_add_ps`" intrinsic:

```
void aussie_vector_add_scalar_AVX2(
   float v[], int n, float c)
{
    // Add scalar constant to all vector elements
    const __m256 rscalar = _mm256_set1_ps(c); // scalars
    for (int i = 0; i < n; i += 8) {
        __m256 r1 = _mm256_loadu_ps(&v[i]); // Load floats
        __m256 dst = _mm256_add_ps(r1, rscalar); // Add
        _mm256_store_ps(&v[i], dst);  // convert to floats
    }
}
```

Vectorized RELU with Max Intrinsics

The RELU activation function is an important piece of code in AI engines. However, it's very simple, arithmetically converting negatives to zero, leaving positives unchanged. This is algebraically equivalent to *max(x,0)*, which can be implemented in AVX like a "max-scalar" operation.

To vectorize RELU applied to a whole vector of `float` elements, we are effectively doing a SIMD `max` operation with a scalar zero (i.e., `0.0`). Hence, the code is very similar to vectorization of add-scalar, but uses the "`_mm_max_ps`" intrinsic.

The AVX1 version of vectorized RELU looks like:

```
void aussie_vector_reluize_AVX1(float v[], int n)
{
    // Apply RELU to each element (sets negatives to zero)
    if (n % 4 != 0) {
        aussie_assert(n % 4 == 0);
        return; // fail
    }
    const __m128 rzeros = _mm_set1_ps(0.0f);  // zeros...
    for (int i = 0; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]);   // Load floats
        __m128 dst = _mm_max_ps(r1, rzeros);  // MAX(r1,0)
        _mm_store_ps(&v[i], dst);  // store back to floats
    }
}
```

And here is the AVX2 version doing 8 `float` elements at a time using the "`_mm256_max_ps`" intrinsic:

```
void aussie_vector_reluize_AVX2(float v[], int n)
{
    // Apply RELU to each element (sets negatives to zero)
    if (n % 8 != 0) {
        aussie_assert(n % 8 == 0);
        return; // fail
    }
    const __m256 rzeros = _mm256_set1_ps(0.0f); // zeros...
    for (int i = 0; i < n; i += 8) {
        __m256 r1 = _mm256_loadu_ps(&v[i]); // Load floats
        __m256 dst = _mm256_max_ps(r1, rzeros); // MAX(R1,0)
        _mm256_store_ps(&v[i], dst);  // store to floats
    }
}
```

Vectorization of Exponentiation

The `expf` function is very expensive to call, but exponentiation of entire vectors of `float` values are required in several parts of AI engines, such as activation functions and Softmax normalization. Surprisingly, in x86 there are CPU opcodes to do exponentiation in hardware, and there are matching AVX intrinsics for SIMD exponentiation operations on small vectors (i.e., 4 `float` values for AVX-1 and 8 `float` values for AVX-2).

The basic C++ version to apply `expf` to every element of a vector, and store the result in the original vector, looks like this:

```
void aussie_vector_expf(float v[], int n)
{
    // Apply EXPF (exponential) to each element
    for (int i = 0; i < n; i++) {
        v[i] = expf(v[i]);
    }
}
```

Loop Pointer arithmetic. Applying the basic C++ optimization of pointer arithmetic, the new code is:

```
void vector_expf_pointer_arith(float v[], int n)
{
    for (; n > 0; n--, v++) {
        *v = expf(*v);
    }
}
```

AVX1 SIMD exponentiation of 4 values: There is an AVX intrinsic called "`_mm_exp_ps`" to exponentiate 4 `float` values in parallel using the 128-bit registers.

Here's the new vector exponentiation code with loop unrolling every 4 elements and AVX1 vectorization:

```
void aussie_vector_expf_AVX1(float v[], int n)
{
    for (int i = 0; i < n; i += 4) {
      __m128 r1 = _mm_loadu_ps(&v[i]); // Load float
      __m128 dst = _mm_exp_ps(r1); // Exponent (expf)
      _mm_store_ps(&v[i], dst);  // convert to floats
    }
}
```

AVX2 SIMD exponentiation of 8 floating-point values: The AVX2 intrinsic is "`_mm256_exp_ps`" to exponentiate 8 elements in parallel using the 256-bit registers.

The new code with loop unrolling every 8 values and AVX-2 intrinsics becomes:

```
void aussie_vector_expf_AVX2(float v[], int n)
{
    // Apply EXPF (exponential) to each element
    for (int i = 0; i < n; i += 8) {
      __m256 r1 = _mm256_loadu_ps(&v[i]); // Load
      __m256 dst = _mm256_exp_ps(r1);  // Exponentiate
      _mm256_store_ps(&v[i], dst); // cvt to floats
    }
}
```

Benchmarking results. The results of optimization of exponentiation are striking! AVX1 is massively faster, cutting out 97% of the original computation time, and then AVX2 is faster still. It's almost like hardware is faster than software. Who knew?

```
Vector-exponentiation benchmarks (N=1024, ITER=100000):
Vector expf basic: 6695 ticks (6.70 seconds)
Vector expf pointer-arith: 6395 ticks (6.39 seconds)
Vector expf AVX1: 260 ticks (0.26 seconds)
Vector expf AVX2: 124 ticks (0.12 seconds)
```

Vectorization of Lookup Tables

The use of lookup-tables is already a powerful speed optimization, but we can double down by adding vectorization. The AVX SIMD instruction sets include a variety of "gather" intrinsics that perform parallel array lookups from a vector with integer indices, using a base address. The basic algorithm we're going to use for AVX SIMD optimizations of a LUT precalculation of some mathematical function is as follows:

- Offline: Precalculate a big LUT for 24 bits with 2^24 elements using non-AVX basic C++ methods.
- Input: vector of 4 `float` values (AVX-1) or 8 `float` values (AVX-2).
- Use a cast to treat these `float` arrays as arrays of integers.
- Load these "`int`" arrays into an AVX register.
- AVX shift right by 8 with the AVX-2 "`_mm_srli_epi32`" intrinsic, which shifts right and adds zero bits, so that they are now 24-bit numbers in 32 bits, with a zero sign bit (hence, all indices are positive integers).
- AVX "gather" with the LUT array, and scale of 4 (i.e., `float` byte size).
- Store the AVX register results back into an array of `float` values.
- Output: vector of 4/8 `float` values with the LUT-calculated function.

Note that we can use a smaller (or bigger) LUT than 24 bits simply by modifying the bitshift counts.

LUTs with AVX Shuffle. Another way to implement a LUT in AVX is to use "shuffle" operations on another register. This only works for small lookup tables, that have few enough elements to fit inside AVX registers. In other words, this can be fast, but only for 16 or 32 elements in the LUT for AVX-2, or more if you use AVX-512. This optimization is unlikely to be relevant to computing the massive 16-bit or 24-bit LUTs that we need for AI mathematical functions.

AVX SIMD Pointer Dereferences. A corollary to the AVX LUT "gather" functionality is they can possibly be used to vectorize arrays of pointers, where the pointers are directly aimed at the data without any intervening lookup-table. For example, suppose we have an array of pointers to float (i.e., rather than an array of integer indices), and we want to access these addresses to generate the corresponding array of float. This is analogous to using a lookup table, but with a base address of zero. Hence, we could potentially use AVX "gather" intrinsics with a zero base address, and the integer offsets equal to the address (i.e., the pointers converted to integer). The x86 platform has 64-bit pointers, so 64-bit integer index offsets are required in the "gather" intrinsic. For example, the AVX2 "`_mm256_i64gather_epi32`" and "`_mm256_i64gather_ps`" intrinsics seem to be along these lines with 64-bit indices. I haven't actually tested this approach to check if it works.

Auto-Vectorization and Restricted Pointers

Modern C++ compilers attempt to automatically vectorize simple loops. Basic loop structures can be unrolled by optimizers, either partially or fully, and then sent to hardware acceleration automatically.

One of the most important hints to the compiler is a "`restrict`" designation on pointer variables. Ironically, the benefit of `restrict` is to limit what you can code, but also to allow unrestricted use of the pointers by the optimizer.

The purpose of the `restrict` attribute is a type specifier to tell the C++ compiler that a given pointer or array variable is not an "alias" for any other pointer. There are various loop transformations and vectorization optimizations that cannot be performed if the compiler has to be conservative and assume that aliasing could occur.

One of the main uses of `restrict` is on pointer or array function parameters, because arrays are pointers in this context. For example, if we have two function parameters (e.g., vector addition), declaring both parameters as `restrict` tells the compiler that the two pointers will never point to the other vector.

Note that this use of the word "aliasing" refers to two pointers referring to the same object or array (i.e., the pointers are aliases of each other). There is another unrelated but similar use of the term in C++ "aliases" for declarations, which means one function or type with two alias names.

The "`restrict`" keyword is merely a hint to the optimizer, and recalcitrant C++ compilers are free to ignore the advice. In fact, "`restrict`" isn't even valid C++, because it's part of C, but not yet in the C++ standard. Nevertheless, various compilers support it or similar extensions like `__restrict__`, so it can be used in C++ programs.

Restricted pointers don't always need to be marked as such. In some usages, the use of "`const`" can allow the compiler to infer non-aliasing of parameters, but it probably doesn't hurt to declare it with "`restrict`" as well. Note also that the C++ compiler is free to assume non-aliasing of pointers of different types, because it is undefined behavior if they are aliases. This is known as the "strict aliasing rule" and this assumption can be disabled in GCC via the option "`-fno-strict-aliasing`".

The C++ compiler doesn't really check if you are lying (to yourself). If you tell the compiler that pointers are restricted, and then pass in two aliased pointers, the behavior of your program is "undefined" and there aren't likely to be any compilation errors or runtime warnings. So, don't do that.

The correct declaration of a "`restrict`" pointer is:

```
int * restrict ptr;  // Correct
```

This is actually incorrect:

```
int restrict * ptr;   // Wrong
restrict int * ptr;   // Also wrong
```

The syntax for array parameters has the keyword inside the square brackets:

```
void myfunc(int arr[restrict]);
```

Read-only functions. Note that read-only functions don't really need to use the `restrict` keyword. For example, the calculation of a vector dot product of two arrays doesn't really have an aliasing problem, since neither of the vectors are changed.

Restricted references. The "`restrict`" type specifier can be used on references, as well as pointers and arrays. This is helpful for some of the issues with aliasing between references in pass-by-reference function parameters. But this usage of `restrict` for references isn't very important for auto-vectorization optimizations.

Restricted "this" pointer. GCC also supports specifying that the class object "`this`" pointer is unaliased by marking the function body with the "`__restrict__`" keyword. This is placed after the closing right parenthesis of the function parameters (i.e., similar to a `const` member function declaration). The declaration looks like:

```
void MyClass::myfunc(int x) __restrict__;
```

Overall, it's unclear how much all these restricted pointer specifiers help the compiler to optimize, but it certainly won't harm the performance!

14. Lookup Tables & Precomputation

Precomputation with Lookup Tables

Look-up tables (LUTs) are a well-known simple data structure for optimizing code. They have been used to optimize algorithms in various ways. Some examples include:

- Precomputed activation functions
- Zero-multiplication networks
- Approximation of non-linear functions

Precalculation or precomputation is a code optimization where results are partially or fully calculated ahead of time. This method is similar to caching and computation reuse but refers to calculations being performed long before they are needed, often at program startup or compile-time, and stored in lookup tables. Like caching, this method trades extra space for time.

Vectorization of LUTs is possible with hardware acceleration primitives that support parallel memory accesses using integer indices. For example, the x86 CPU with AVX intrinsics has a set of "gather" instructions for doing indexed lookup that can be used to load from a LUT into the internal registers, and "scatter" instructions for storing the registers back to an indexed LUT.

Typical precalculations are those where the results are computed at program initialization or compile-time. The best methods generate the results at compile-time, and are simply loaded as data, such as numeric constants or pre-initialized data arrays. There are multiple ways to do this:

- Program startup initialization
- Lazy evaluation
- Binary data file
- Precompiled source code

One method for precomputation of larger amounts of data in an array or lookup table is to perform the initialization dynamically at program startup. A lookup table can be populated with the required results, before the main logic of the program begins. Or alternatively, the idea of "lazy evaluation" allows storing the precomputation into a lookup table only when the program first needs the data.

A faster alternative is to calculate all this data offline before program startup, and store the results in a binary data file. This data file can then be loaded into an array at program startup, without needing to perform any of the arithmetic computations. Whether this is beneficial depends on the cost of the computations versus the cost of file loading.

The logical extension of the precomputation method for a large number of numeric results is to write special C++ code that performs these calculations, but then outputs the results into a text file in the exact format of a C++ source code file (rather than a data file), that declares a global array name and the numeric values. This auto-created C++ code is then linked with your program.

Example: LUT Precomputation for sqrt

Let's say that you want to optimize a slow non-linear function like "`sqrtf`" (or "`expf`" or "`logf`"). These are good candidates for optimization because of their non-linearity.

The first point is that you'd better do a really good job, because there are actually hardware instructions for these common math functions, even in x86 architectures. So, you could easily optimize this into a table lookup, and find that your C++ code is still slower than the single CPU instruction that's called by the standard C++ library versions. Hence, investigate the C++ intrinsic functions for common math functions before you assume that you can do better than electrons zipping through silicon.

This example investigates precomputing "`sqrtf`" even though that may not be as fast as hardware-acceleration. However, the same ideas apply to precomputing more sophisticated derivative functions, such as Softmax and activation functions, which are not hardware-supported (or not yet, anyway). The same general ideas apply.

The basic method for table lookup optimization is:

- Declare a big array (the bigger the better).
- Run a loop sending every value to the real "`sqrtf`" function.
- Store each result in the big array.
- Now you have a precomputed table of all possible values.
- Later, use an array index lookup to compute the function fast.

How is than any faster? I mean, we've just called "`sqrtf`" a bazillion times with numbers that we probably won't ever need. Yes, there is extra cost, and we are running slower during program initialization. There are at least two ways to fix this:

1. Load the array values from a pre-built binary data file instead, or,

2. Precompile the array data into a C++ source code file.

However, this complaint underestimates just how many times the code may call these functions. Even with this startup cost, once that is all done and dusted, we have a big array of precomputed data that we can use to speed up the program execution, which is our main goal. And in a production environment, any extra startup cost is hopefully amortized over many executions.

Example: Precomputing sqrt of integer: For simplicity, we're going to first assume that we're computing a `float` square root of integers. The function we are precomputing is "`int`-to-`float`" type. This makes it easier, because the `int` can be used as an array index.

Here's my big array with about 65,000 entries:

```
#define AUSSIE_SQRT_PRECOMP_MAX (1u<<16)
float g_sqrt_precomp_table[AUSSIE_SQRT_PRECOMP_MAX];
```

Here's the unoptimized function "`int`-to-`float`" version of "`sqrtf`" that we are planning to precompute:

```
float aussie_sqrtf_basic_int(int x)
{
    return sqrtf((float)x);
}
```

Here's the initialization call to the precomputation routine, sending in the array, the size *N*, and the function pointer:

```
aussie_generic_precompute_int(
    g_sqrt_precomp_table,   // Big array
    AUSSIE_SQRT_PRECOMP_MAX,  // N
    aussie_sqrtf_basic_int    // Function pointer
);
```

And here's the code to run the big precomputation loop:

```
void aussie_generic_precompute_int(
  float arr[], unsigned int maxn, float (*fnptr)(int))
{
    for (unsigned int i = 0; i < maxn; i++) {
            arr[i] = fnptr(i);
    }
}
```

So, that's all there is to initialize the LUT. Once this function returns, we now have a big array full of data. Here's what the new optimized "`sqrtf`" looks like:

```
float aussie_table_lookup_sqrt(int i)
{
    return g_sqrt_precomp_table[i];
}
```

And we can make that function "`inline`" or use a C++ preprocessor macro:

```
#define AUSSIE_TABLE_LOOKUP_SQRT_BASIC(i) \
     ( g_sqrt_precomp_table[(i)] )
```

So, here are a few provisos about this code:

1. Might be slower than `sqrt` in hardware (needs benchmarking).

2. Unsafe array index accesses (e.g., crashes on negatives or large numbers).

3. `unsigned int` types can overflow and then spin infinitely for precomputing tables of size "`1<<32`" (change to `unsigned long`).

4. The memory size of the precomputed table for `1<<16` is already about 262k (65k times 4 bytes).

Float-to-Float Precomputation

Using a precomputed table lookup for a float-to-float function is more complicated than integers. However, this is also the main approximation needed for non-linear functions, or even the basic math library functions like `sqrtf` or `expf` or `logf`.

Why is it tricky? The reason that `float` inputs are more difficult is that we need to convert a `float` into an array index in order to look it up. For example, we could try type casts:

```
int offset = (int)f;
```

But that limits us to only precalculating values for 1.0, 2.0, 3.0, etc. Our approximation works poorly on any fractions, and we also haven't limited the array index to a fixed finite range, so it won't work for any negative values or very large positive values. And the type cast of a `float` is also slow!

Scaled Multiple: Another idea is that we could scale it upwards to get more decimals:

```
int offset = (int) (f * 1000.0f);
```

This approach at least gives us 3 decimal places: e.g., 1.234 or 23.456, or similar. We will still have to check for negatives and large values to bound it. But again, this is even slower!

Bitwise Floating-Point Truncations: The above truncation via a floating-point scaled multiple is not very fast. Twiddling the bits is much faster. For example, when we have a standard 32-bit `float` type, it has 1 sign bit, 8 exponent bits, and 23 mantissa bits. This is from left-to-right, with the sign bit as the most significant bit, and the low-end mantissa bits are the least significant bits. Remember that this is like Scientific notation:

- Number = Mantissa x 2 ^ Exponent

Also, the sign bit makes it all negative, if set. Note that exponent in 8-bits encodes the numbers -128 to +127, so that ranges from very small 2^-128 near-zero values, to very huge 2^127 sized values. If the mantissa was in decimal, and it was "1234567" and the exponent was "17":

- Number = 1.234567 x 10^17

If the mantissa was 23 bits, it's actually binary digits, with about 3 binary digits per decimal digit, so a 23-bit mantissa is about 7 or 8 decimal digits. Note that the mantissa is actually 24 bits, not 23, because there's an "implicit one" mantissa bit, not that it changes the calculation; you needed to know that for C++ trivia night.

So, if we think about it for a year or two, it becomes obvious that the rightmost bits of the mantissa are simply the rightmost digits in "1.234567", and if we truncate some of the rightmost bits, it's like truncating a very small fraction (e.g., "1.234567" becomes "1.2345" or whatever).

Hence, a first idea is just to cut off 2 of the 4 bytes of a 32-bit `float`. This leaves us with 1 sign bit, 8 exponent bits, and 7 mantissa bits (plus 1 implied bit makes 8 mantissa bits). In decimal, the 8-bit mantissa now encodes only about 2 or 3 decimal digits, as if we've truncated "1.234567" to "1.23".

Incidentally, congratulations, you've created "*bloat16*" type, which is what Google did with TPUs, making a 2-byte `float` format with 1 sign bit, 8 exponent bits, and 7 stored mantissa bits. So, now you can get into your blue telephone booth, time travel back a decade, file a patent, and retire on your royalties. If you're ever a contestant on *Wheel of Fortune* you probably won't need to know that the "b" in "bfloat16" stands for "brain float" and that is such a great name. But I digress.

Anyhow, this idea actually works for the precomputation. A 2-byte integer value in `bloat16` format is easy to extract from a 4-byte FP32 float (i.e., the uppermost two bytes). The trick for bitwise processing is to convert the `float` to `unsigned int`, because the bitwise shift operators don't work on `float` (it's planned for C++37, as I heard at my fungus collector's club trivia night).

```
float f32 = 3.14f;
unsigned u32 = *(unsigned int*)&f32;
```

Extracting the top-most 2 bytes (16 bits) is simply a right bitshift:

```
unsigned ubf16 = ( u32 >> 16 );
```

Note that here's a good reason that we had to use "`unsigned`" integer type. The right bitshift operator (>>) has undefined behavior on negatives, so "`int`" type wouldn't work predictably (or portably) if the floating-point sign bit was set.

The result is a 16-bit `unsigned` integer to use as the array index. Hence, there are only `1<<16=65,536` entries in our precomputation table. Assuming we store results as 4-byte `float` values, this makes the precomputation array's memory size about 262kb.

What's more, it works for negative `float` numbers, because the sign bit is still part of that shemozzle, and we also don't need to check any minimum or maximum bounds, because it works for all 32-bit float numbers.

Precomputing with 24-Bit Lookup Tables: Interestingly, none of the above code is especially tied to 16-bit sizes. The `bfloat16` version truncates 32-bit float to 16-bit by truncating the rightmost 16 mantissa bits. But we can actually choose to keep however many mantissa bits we like. The trade-off is that more mantissa bits increase accuracy, but at the cost of needing a much bigger precomputation array (doubling the storage size for each extra bit).

Let's try only cutting the rightmost 8 mantissa bits, leaving us with 24 stored bits total (i.e., 1 sign bit, 8 exponent bits, and 15 stored mantissa bits). The mantissa bits reduce from 23 to 15 (plus one implied bit makes 16), so this now stores about 5 decimal digits (e.g., "1.2345"), giving quite good precision on our results. When I tested the 16-bit version, it had some reasonably large errors of almost 0.1 in computing `sqrt`, whereas this 24-bit version has much lower errors, as expected.

Code changes are minor. The bitshift operations simply change from 16 bits to 8 bits (i.e., `32-24=8` bits). This is the precomputation loop for 24 bits:

```
void aussie_generic_precompute_24bit_float(
  float farr[], unsigned int maxn,
  float (*fnptr)(float))
{
   for (unsigned int u = 0; u < maxn; u++) {
      unsigned int unum = (u << 8u); // 32-24=8 bits!
      float f = *(float*)&unum;
      farr[u] = fnptr(f);
   }
}
```

And this is the call to the precomputation function in the startup phase:

```
aussie_generic_precompute_24bit_float(
  g_sqrt_float_24bit_precomp_table, // Big array
  (int)AUSSIE_SQRT_24bit_MAX,     // 1 << 24
  aussie_sqrtf_basic_float        // Function pointer
);
```

The table lookup routine also similarly shifts 8 bits, rather than 16, but is otherwise unchanged:

```
float aussie_table_lookup_sqrt_24bit_float(float f)
{
    unsigned u = *(unsigned int*)&f;
    u >>= 8;  // 32-24=8 bits
    return g_sqrt_float_24bit_precomp_table[u];
}
```

Note that this only works if we are sure that both "`float`" and "`unsigned int`" are 32-bits, so we should check that during startup with some assertions via `static_assert`. If we are sure of that fact, then not only will it work, but we don't also need to check the array bounds. It won't try a negative array index, and won't overflow no matter what bit pattern we send it in as a `float`.

But there is one problem. If we send the fast table lookup version the special `float` value of `NaN` ("not a number"), then the table lookup routine will actually return a valid numeric answer, which probably isn't what we want. Maybe we need to add a check for that special case, and this needs more testing.

The new size of the precomputation array is `2^24=16,777,216`, so we have about 16.7 million results available. If our results are 32-bit `float` values, our `bloat16` precomputed array above requires about 262kb, and the new size with 24-bits is a lookup table (array) of about 67 megabytes. It wouldn't have worked on my old TRS-80 CoCo in 1986, but it'll work nowadays.

Precalculating C++ Source Files

One way to improve on the precomputation of a big array is to skip it entirely during startup by writing a lot of code. It's like using an AI coding copilot, only it's not really. I mean, come on, the day an AI writes better code than me is the day that I retire to the hologram beach with my robot dog companions.

The idea here is to write a program to generate a C++ source file that contains the global precomputed lookup table. Yes, it's a C++ program that creates part of a C++ program, which is almost like your AI has become self-aware, only one step away from *Skynet*. Well, maybe not, it's just a dumb C++ program written by a dumb human creating some dumb data.

Anyway, this auto-generated C++ code can be compiled and linked into your C++ program, and used like a global array of data in other parts of the program. Zero calculations are required at runtime, and the data can be read-only.

The benefit is that this auto-generated code method does not even require the time cost of startup initialization for any precomputations. There's not even the cost of data file loading. Instead, the data is auto-loaded by the linker-loader during executable file instantiation (i.e., when the user starts the app). The only downsides for the user are that the size of the executable program increases, which means more disk space usage, and that application program startup may take longer and it will use more memory (regardless of whether it ever needs this precomputed data). Also, various offline tasks take longer for the software developers, such as compilation and linking for testing, which is why we bill per hour.

I tried this out for precalculating GELU with a 24-bit table. The C++ source file was size 514k for 24-bit precomputation table of size `1<<24`. This is what the auto-generated source code should look like:

```
// Precomputed table source code: GELU
// "gelu_precomp_24bits.cpp"
float g_gelu_table_precompute_24bits[] = {
0f,
1.79366203433576585078237386611092648039e-43f,
3.58732406867153170156474773322185296077e-43f,
5.38098610300729755234712159983327944116e-43f,
7.17464813734306340312949546644370592155e-43f,
...
...
};
```

Here's the code to generate the code to generate the code to generate the code:

```
// Print C++ of 24-bits GELU precomputed table
void aussie_generic_setup_table_FP32_24bits_PRINT_SOURCE(
     char* nickname,
     char* outfname,
     float (*fnptr)(float),  // e.g., GELU
     int maxn,  // eg. 1<<24
     float arrout[]  // array to store (can be nullptr)
 )
 {
     if (!fnptr) {
             aussie_assert(fnptr);
             return;
     }
     // Generate C++ source code so we can
     // pre-compile the precomputed GELU table (24-bits)
     // There are 2^24 = 16.7 million numbers...
```

```
        FILE* fp = stdout;
        bool writingfile = false;
        bool add_commented_number = true;
        if (outfname && *outfname) {
                fp = fopen(outfname, "w");
                if (!fp) {
                        aussie_assert(fp);  // file write failed
                        return;  // fail
                }
                writingfile = true;
                add_commented_number = false; // Not for files
        }
        unsigned int u = 0;
        fprintf(fp, "// Precomputed table source code: %s,
\"%s\"\n", nickname, outfname);
        fprintf(fp, "float g_gelu_table_precompute_24bits[] = {
\n");
        char numbuf[5000] = "";
        for (; u < maxn /*1<<24*/ ; u++) { // For 2^24=~16.7M
                // zeros in least significant 8 mantissa bits
                unsigned int uval = u << 8;
                float f = AUSSIE_UINT_TO_FLOAT(uval);
                float g = fnptr(f);  // Call GELU or whatever
                if (arrout) arrout[u] = g;  // Store data
                // Format: %g means the smaller of %e or %f
                // ... %e is exponent format (scientific-like)
                char* buf = numbuf;
                // Format %g (Number)
                // ... and suffix "f" (float constant type)
                sprintf(buf, "%40.40gf", g);
                if (strchr(buf, 'n')) {
                    // Nan or "-nan" ... Dummy value
                    strcpy(buf, "0.0 /*nan*/");
                }
                // Remove prefix padding spaces...
                while (buf[0] == ' ') buf++;

                // Remove suffix zeros ...
                int len = (int)strlen(buf);
                if (buf[len - 1] == 'f') len--; // skip suffix f
                if (buf[len - 1] == '0') {
                    while (len > 5) {
                        if (buf[len - 1] == '0'
                            && isdigit(buf[len - 2])) {
                           if (buf[len] == 'f') {
                                buf[len - 1] = 'f';  // leave f
                                buf[len] = 0;
                            }
                            else {
                                buf[len - 1] = 0;  // remove
                                buf[len] = 0;
                            }
                            len--;
```

```
                    }
                    else break;
                }
            }

            if (add_commented_number) {
                fprintf(fp, "%s // (%40.40f) [%u] \n", buf,f,u);
            }
            else {  // No comments...
                fprintf(fp, "%s,\n", buf);
            }

            // Progress update
            if (u % 100000 == 0 && u != 0) {
                if (writingfile) fprintf(stdout, "%u -- %s\n",
u, buf);  // Progress
                fprintf(fp, "// U= [%u]\n", u);  // Comment
            }
    }
    fprintf(fp, "}; \n");  // Close initializer...
    if (fp && fp != stdout) fclose(fp);
  }
```

Conclusions on Source Code Generation: Does it work? Yes and no. It builds the output file quite quickly, zipping through `1<<24` computations and writing to disk. But I can't get this 24-bit version with its 500k CPP source file to actually compile in the Microsoft Visual Studio IDE. Maybe it works on Windows command-line or Linux GCC, but I haven't tried.

Anyway, this self-generating code idea is certainly quite workable for table lookups of approximations for FP16 numbers (16-bit half-precision floating-point), because the lookup table needs to "only" contain 2^16=65,536 numbers. This is about a 200k C++ source file in plain text, and creates linked data of about 65k times 4 bytes equals about 256k space usage. This would use half that space if you also store the computation as 16-bit numbers rather than 32-bit floats or integers.

References

1. Nils Graef, 12 Mar 2024 (v3), *Transformer tricks: Precomputing the first layer*, https://arxiv.org/abs/2402.13388 Code: https://github.com/OpenMachine-ai/transformer-tricks (Because the first layer only depends on the embeddings, it can be precomputed.)
2. SZ Lin, YC Chen, YH Chang, TW Kuo, HP Li, 2024, *LUTIN: Efficient Neural Network Inference with Table Lookup*, ISLPED '24, August 5-7, 2024, Newport Beach, CA, USA, https://dl.acm.org/doi/pdf/10.1145/3665314.3670804

3. S Fanning, *Fixed Point Multiplication-Free Implementation of Deep Neural Networks for Embedded Systems*, Masters Thesis, School of Electrical and Electronic Engineering, University College Dublin 2018, https://seanfanning.eu/posts/projects/low-bitwidth-neural-networks/Thesis_SeanFanning_13360951.pdf
4. Mohammad Samragh Razlighi; Mohsen Imani; Farinaz Koushanfar; Tajana Rosing *LookNN: Neural network with no multiplication*, Design, Automation & Test in Europe Conference & Exhibition (DATE), 27-31 March 2017, https://ieeexplore.ieee.org/document/7927280 (Lookup-table based multiplication.)
5. Covell M, Marwood D, Baluja S, Johnston N., *Table-based neural units: Fully quantizing networks for multiply-free inference*, 2019, arXiv preprint arXiv:1906.04798, http://arxiv.org/abs/1906.04798
6. Joonsang Yu, Junki Park, Seongmin Park, Minsoo Kim, Sihwa Lee, Dong Hyun Lee, Jungwook Choi, Dec 2021, *NN-LUT: Neural Approximation of Non-Linear Operations for Efficient Transformer Inference*, https://arxiv.org/pdf/2112.02191
7. Neelesh Gupta, Narayanan Kannan, Pengmiao Zhang, Viktor Prasanna, 8 Apr 2024, *TabConv: Low-Computation CNN Inference via Table Lookups*, https://arxiv.org/abs/2404.05872
8. Darshan C. Ganji, Saad Ashfaq, Ehsan Saboori, Sudhakar Sah, Saptarshi Mitra, MohammadHossein Askari Hemmat, Alexander Hoffman, Ahmed Hassanien, Mathieu Léonardon, 18 Apr 2023, *DeepGEMM: Accelerated Ultra Low-Precision Inference on CPU Architectures using Lookup Tables*, https://arxiv.org/abs/2304.09049
9. Grigor Gatchev, Valentin Mollov, 4 Apr 2021, *Faster Convolution Inference Through Using Pre-Calculated Lookup Tables*, https://arxiv.org/abs/2104.01681
10. Han Guo, William Brandon, Radostin Cholakov, Jonathan Ragan-Kelley, Eric P. Xing, Yoon Kim, 15 Jul 2024, *Fast Matrix Multiplications for Lookup Table-Quantized LLMs*, https://arxiv.org/abs/2407.10960
11. Davis Blalock, John Guttag, 21 Jun 2021, *Multiplying Matrices Without Multiplying*, https://arxiv.org/abs/2106.10860
12. Gunho Park, Hyeokjun Kwon, Jiwoo Kim, Jeongin Bae, Baeseong Park, Dongsoo Lee, Youngjoo Lee, 10 Mar 2025, *FIGLUT: An Energy-Efficient Accelerator Design for FP-INT GEMM Using Look-Up Tables*, https://arxiv.org/abs/2503.06862

Part III: Multidimensional Data Structures

"Bad programmers worry about the code.
Good programmers worry about
data structures and their relationships."

— Linus Torvalds

15. Matrix Multiplication

Matrix-Vector Multiplication

Matrix multiplication by a vector gives another vector. Let us consider the simple case first, where the matrix is square with dimensions *NxN* and the vector is also of size *N*. The matrix has *N* rows and *N* columns, and the input vector has *N* elements. The resulting output vector will also have the same *N* elements.

Conceptually, in pseudocode:

```
MAT[N][N] * VIN[N] -> VOUT[N]
```

It's not immediately obvious, or at least, I don't remember my High School Math teacher mentioning it, but matrix-vector multiplication is a bunch of vector dot product computations. We need to do a vector dot product for each of the elements of the output vector. Each element is a dot product of a matrix row times the input vector.

Note that the dimensions match for a dot product, with *N* matrix rows and *N* elements in the input vector.

Rectangular matrices. The general case of a rectangular matrix multiplied by a vector is a little trickier, but not a lot. If our matrix is *MxN* and the vector is size *N*, then the output vector has size *M*. Note the two of the dimensions must match: the columns of the matrix and the elements of the input vector are both *N*. However, this dimension *N* "disappears" and the output vector has size only dependent on *M*. The pseudocode:

```
MAT[M][N] * VIN[N] -> VOUT[M]
```

The rectangular matrix-vector multiplication is almost identical to square matrix-vector computations. Each element of the output vector is a dot product of a matrix row with the input vector.

Again, we note that the dimensions of the matrix rows (*N*) must match the size of the input vector (*N*), or else we cannot compute it. I mean, we *could* still compute it with mismatched dimensions, such as by assuming that the shorter one (matrix row or input vector) had zeros in the missing elements, but that sounds a little buggy.

Complexity of Matrix-Vector Multiplication. The algorithmic complexity of matrix-vector multiplication is quadratic in *N*, whereas matrix-matrix multiplication is cubic in *N*. The basic matrix-vector multiplication scans *N* rows of the matrix, with each row element performing a computation against each of the *N* elements of the vector, giving two nested loops with an overall *O(N^2)* cost.

Memory layout: One important point for the efficiency of matrix-vector multiplication is that the default memory layout has contiguous addresses for both the matrix row and the vector. Obviously, a vector is just a sequence of memory with all the elements in series. Not so obviously, a row of a matrix, when stored as a C++ two-dimensional array, is also a contiguous set of data (i.e., a matrix row is like a vector).

Hence, the dot product multiplication of a matrix row and the input vector is simply scanning forward along contiguous addresses for both of its inputs, which makes it easy to vectorize.

Spot the Buggy MatMul

Have a look at this code for a matrix-vector multiplication using vector dot product. It took me a long time to realize what was wrong with this. Can you spot the bug?

```
void matmul_vector_basic1_buggy(ymatrix m, float v[], int n)
{
    // Basic matrix-by-vector using vector dot products..
    for (int i = 0; i < n; i++) {
        float* rowvector = &m[i][0];
        float sum = aussie_vecdot_basic(rowvector, v, n);
        v[i] = sum;
    }
}
```

The bug is a kind of aliasing problem here:

```
    v[i] = sum;  // Bug!
```

It looks correct, but it's wrong. The computation of `v[i]` is setting its value in the middle of the loop, and then going around for the next matrix row, which will then use that newly calculated `v[i]` value as if it was part of the input vector. Because I'm misusing "v" as both the input and output vector, parts of the output vector will get used as the input vector. It's a very insidious type of aliasing bug, and many of my simple unit tests with zero matrices and identity matrices were still succeeding. It's my fault for trying to do matrix-vector multiplication as an element-wise vector method.

The solution is simple: matrix-vector multiplication needs a third operand for the output vector.

Optimizing Matrix-Vector Multiplication

The fixed-up version of matrix-vector multiplication with row-wise vector dot products simply outputs to another separate destination vector operand.

```
void aussie_matmul_vector_basic_out1(
   const ymatrix m, const float v[], int n, float vout[])
{
    // Basic matrix-by-vector using vector dot products..
    for (int i = 0; i < n; i++) {
        const float* rowvector = &m[i][0];
        float sum = aussie_vecdot_basic(rowvector, v, n);
        vout[i] = sum;
    }
}
```

Nested Loop Matrix-Vector Version: The same matrix-vector multiplication algorithm in the form of two nested loops is below. This is flattening the call to the lower-level vector dot product function and putting its inner summation loop directly inside the outer main loop. The basic C++ code looks like:

```
void aussie_matmul_vector_basic_out2(
  const ymatrix m, const float v[], int n, float vout[])
{
    // Basic matrix-by-vector using nested loops..
    for (int row = 0; row < n; row++) {
            float sum = 0.0f;
            for (int col = 0; col < n; col++) {
                    sum += (m[row][col] * v[col]);
            }
            vout[row] = sum;
    }
}
```

Optimizations of matrix-vector multiplication. Various ways to optimize the naive nested loop matrix-vector multiplication suggest themselves:

- Hoisting loop-invariant code (also called loop code motion) of the "`m[row]`" expression.
- Loop pointer arithmetic for both loops.
- Loop unrolling of the inner loop to unroll 4, 8 or more iterations.
- Loop tiling to unroll a 2x2 tile/block.
- Vectorization using the AVX1/AVX2 vector dot product versions we already examined.

I tried coding several more of these optimizations and here are the benchmarks:

```
Matrix-Vector mul (MatMulVec) benchmarks (N=2048, ITER=300):
Matrix-vector nested loops: 3480 ticks (3.48 seconds)
Matrix-vector nested loops hoisted: 3489 ticks (3.49 seconds)
Matrix-vector nested ptr-arith: 3415 ticks (3.42 seconds)
Matrix-vector unrolled inner (4): 1166 ticks (1.17 seconds)
Matrix-vector unrolled inner (8): 938 ticks (0.94 seconds)
Matrix-vector nested tiled 2x2: 1995 ticks (2.00 seconds)
Matrix-vector vecdot AVX1 DP: 1414 ticks (1.41 seconds)
Matrix-vector vecdot AVX2 FMA: 929 ticks (0.93 seconds)
```

Interestingly, code hoisting and loop pointer arithmetic were a waste of effort. Loop tiling did better than the original, but probably its speedup is primarily from the effect of loop unrolling rather than data locality or cache hit rates, since simpler loop unrolling did better.

Note that the AVX1 version used the "dot product" intrinsic but AVX-2 used the FMA intrinsic, for reasons covered in Chapter 13. Simple loop unrolling also did as well as AVX2 hardware vectorization, probably because the versions of AVX1 and AVX2 were simply calling the vector dot product functions, so they still had function call overhead.

Hence, this algorithm can be further optimized by inlining to fix the AVX function call overhead, combining AVX intrinsics with unrolling of the inner loop, and then some minor final tweaks such as pointer arithmetic.

Tiled Matrix-Vector Multiplication

A more detailed analysis of the matrix-vector algorithm shows that it is not optimal in at least three areas:

- Data locality
- Pipelining AVX intrinsic arithmetic
- Redundant loads

The data locality of the 2x2 tiled version is better, but more improvement is possible, starting with the use of AVX intrinsics inside the "sub-kernel" for the tile.

The AVX instruction sequences of "load, calculate, store" in the earlier non-tiled AVX-optimized versions are not allowing for the natural instruction pipelining with the AVX intrinsics to calculate multiple sums or FMA operations with near-parallel pipelining. And the entire input vector is getting re-loaded repeatedly for every row of the matrix. So, we need to examine improvements on three aspects.

A tiled sub-kernel is the main way to fix data locality and pipelining. Improving data locality is somewhat inherent to tiling. The pipelining can be improved by unrolling the tiled sub-kernel and reordering the loads and stores so they don't block the arithmetic of AVX intrinsics.

Can we avoid redundant vector loads?

Since it's unavoidable to access every element of every row at least once, the redundant loads of the vector suggest that we should modify the algorithm so as to work on a subsection of the vector for each of the matrix rows.

This suggests an inversion of the main nested loops of the algorithm. However, that runs into the major problem that it destroys cache locality, by scanning down the column of the first matrix. I benchmarked this loop interchange idea, and it actually increased execution time. Maybe we should use the transpose of the first matrix, so that it's in column-major order when scanning its columns? No, that's actually just going back to the original algorithm without the loop interchange.

Anyway, a better plan seems to be to reduce the redundant loading by using temporary calculations inside the tile sub-kernel.

Here is what a basic tiled/blocked algorithm using 2x2 tiles looks like in basic sequential C++:

```
void aussie_matmul_vector_tiled_2x2_better(
    const ymatrix m, const float v[], int n, float vout[])
{
    // Tiled/blocked matrix-by-vector using 2x2 tiling..
    aussie_assert(n % 2 == 0);
    for (int row = 0; row < n; row += 2) {
      vout[row] = 0.0f;
      vout[row + 1] = 0.0f;
      for (int col = 0; col < n; col += 2) {
        vout[row] += (m[row][col] * v[col])  // row+0, col+0
           + (m[row][col + 1] * v[col + 1]) // row+0, col+1
              ;
        vout[row + 1] +=
             (m[row + 1][col] * v[col]) // row+1, col+0
          + (m[row + 1][col + 1] * v[col+1])  // row+1,col+1
                ;
        }
    }
}
```

One minor improvement would be to use memset to clear the whole output vector to zero, rather than individual assignments, which I added to the 4x4 tiled version. There is another minor improvement is removing the "common sub-expressions" of `v[col]` and `v[col+1]` and I tried this with no improvement noted in the 2x2 tiled version, but about 10% improvement in the 4x4 tiled version. The computations of `m[row]` and `m[row+1]`, etc., can also be hoisted out of the inner loop, giving another 10% gain for the 4x4 tiled version.

The C++ code for the 4x4 tiled version with a fully unrolled 4x4 sub-kernel now looks like:

```
void aussie_matmul_vector_tiled_4x4_CSE2(
     const ymatrix m, const float v[], int n, float vout[])
{
    // Tiled/blocked matrix-by-vector using 4x4 tiling
    aussie_assert(n % 4 == 0);
    memset(vout, 0, sizeof(float) * n);
    for (int row = 0; row < n; row += 4) {
        const float* rowvec = &m[row][0];
        const float* rowvec1 = &m[row + 1][0];
        const float* rowvec2 = &m[row + 2][0];
        const float* rowvec3 = &m[row + 3][0];
        for (int col = 0; col < n; col += 4) {
          float fcol0 = v[col];
          float fcol1 = v[col + 1];
          float fcol2 = v[col + 2];
          float fcol3 = v[col + 3];
          vout[row] +=
            (rowvec[col] * fcol0) // row+0, col + 0
            + (rowvec[col + 1] * fcol1) // row+0, col + 1
            + (rowvec[col + 2] * fcol2) // row+0, col + 2
            + (rowvec[col + 3] * fcol3) // row+0, col + 3
            ;
          vout[row + 1] +=
            (rowvec1[col] * fcol0) // row+1, col + 0
            + (rowvec1[col + 1] * fcol1) // row+1, col + 1
            + (rowvec1[col + 2] * fcol2) // row+1, col + 2
            + (rowvec1[col + 3] * fcol3) // row+1, col + 3
            ;
          vout[row + 2] +=
            (rowvec2[col] * fcol0) // row+2, col + 0
            + (rowvec2[col + 1] * fcol1) // row+2, col + 1
            + (rowvec2[col + 2] * fcol2) // row+2, col + 2
            + (rowvec2[col + 3] * fcol3) // row+2, col + 3
            ;
          vout[row + 3] +=
            (rowvec3[col] * fcol0) // row+3, col + 0
            + (rowvec3[col + 1] * fcol1) // row+3, col + 1
            + (rowvec3[col + 2] * fcol2) // row+3, col + 2
            + (rowvec3[col + 3] * fcol3) // row+3, col + 3
            ;
        }
    }
}
```

Matrix-Matrix Multiplication

Now let's look at matrix-matrix multiplication, whereas above we looked at matrix-vector multiplication. The proper MatMul and GEMM kernels are coded for full matrix-matrix multiplication.

Matrix multiplication results in another matrix as the output. For the simple case of two square matrices of the same size, the resulting output matrix is also of the same dimensions. In pseudocode:

```
M1[N][N] * M2[N][N] -> MOUT[N][N]
```

For multiplying two rectangular matrices, or sizes *MxN* and *NxP*, we get an output matrix of size *MxP* (i.e., the inner *N* dimensions disappear).

In pseudocode style:

```
M1[M][N] * M2[N][P] -> MOUT[M][P]
```

Note that *P=1* is the case of matrix-vector multiplication, because an *Nx1* matrix is actually a vector with *N* rows of a single element (i.e., one column).

Algorithmic Complexity. The naive implementation of a matrix-matrix multiplication via three nested loops is a cubic algorithm, with *O(N^3)* complexity. The well-known Strassen algorithm has complexity about *O(N^2.7)*, which looks like such a massive improvement.

Other algorithms such as the Coppersmith-Winograd algorithm and numerous sub-variants have better asymptotic complexity, but with a high constant overhead, making them impracticable for anything but very large values of *N*.

Basic Matrix-Matrix Multiplication. The basic algorithm for matrix multiplication is three nested loops. There is nothing fancy here: this is just coding up the basic matrix multiplication method that you forgot the second you finished your Senior math exam.

If you don't believe me, check it out on Wikipedia.

Here's the C++ code:

```
void aussie_matmul_matrix_basic(
      const ymatrix m1, const ymatrix m2,
      int n, ymatrix mout)
{
    // Matrix-Matrix mult basic naive n^3 algorithm...
    for (int row = 0; row < n; row++) {
        for (int col = 0; col < n; col++) {
            float sum = 0.0f;
            for (int k = 0; k < n; k++) {
                 sum += (m1[row][k] * m2[k][col]);
            }
            mout[row][col] = sum;
        }
    }
}
```

The two outer loops are scanning the rows of the first matrix, and the columns of the second matrix. The innermost of the three loops is doing a vector dot product computation over the "k" index variable. However, it's not a normal vector-vector dot product. Instead, it's the dot product of one "horizontal" vector, which is a row of the first matrix, and of a second "vertical" vector, which is a column of the second matrix. Hence, the number of rows in the first matrix must equal the columns of the second matrix, which is true here because we're assuming that both matrices are square. Hence, the "k" variable is spinning down the n elements of a row and a column at the same time. Every element of the *NxN* output matrix requires a vector dot product calculation like this.

Vectorization. None of these matrix multiplication algorithms are especially good, because they are all *sequential*, rather than parallel algorithms. Neither the naive cubic version nor the Strassen algorithm are what we need. What we need for GPUs and CPU SIMD intrinsics are vectorizable algorithms for matrix-matrix multiplication. Unfortunately, the above simple triple-nested matrix multiplication algorithm is *not* one of them, because non-contiguous storage of the second matrix hampers vectorization.

Memory layout problems for matrix-matrix multiplication: The layout for memory with matrix-matrix multiplications is not as fortuitous as it was for matrix-vector multiplications. Each computation in matrix-matrix multiplication is a vector dot product of a row of the first matrix with a column of the second matrix. Each row of the first matrix is happily stored in contiguous memory, but the columns in the second matrix are not. In fact, the "stride" between two elements of a column of a matrix is a very large number of bytes in the default memory layout.

The default storage of matrices and two-dimensional arrays in C++ is called "row-major" storage layout. Row-major storage has each row in contiguous memory. The rows are stored one at a time, top to bottom, and adjacent elements in a row are also adjacent memory addresses. Columns are a second-class citizen in row-major layout, and you have to jump around to find adjacent elements of a column vector.

The alternative storage method is "column-major" storage layout where the columns are stored in contiguous memory, and it's the rows that are in the smoker's carriage at the back of the train. However, column-major is not the default C++ storage mode.

Hence, to vectorize a matrix-matrix multiplication, we want to keep the first matrix in row-major storage, but we need to rearrange the storage of the second matrix to be column-major storage, rather than the default row-major storage. Column-major storage would help vectorize the columns with each column element in adjacent memory locations. The first matrix is fine, but we want the second matrix to be stored in a mirror image of itself.

Hmm, a mirror and a matrix. What does that sound like? A transposed matrix.

Pseudo-Transposed Second Matrix. The simplest way to get column-major order of a matrix (especially if square) is to use the transpose of the matrix, and modify the internals of the matrix multiplication function to pretend that the transpose is actually the column-major storage of the original second matrix. I call it the "fake transpose" method, which is a bit of a misnomer because it is the actual transposed matrix, but we modify the matrix multiplication code to access it with reversed logic indices.

Confusing? Yes, I felt the same way, but if you follow it through carefully, you can see that the transpose is really very similar to storing the original matrix in column-major order, where each column element is stored in adjacent memory. The columns of the original problematic matrix become fake rows in the fake transpose, stored in sequential memory addresses. So, for square matrices, we can take the transpose of a matrix, and it's like the matrix has been converted into column major storage. However, we also need to change the C++ code in the matrix multiplication kernel, because it assumes row-major order storage of both matrices, but now we've got row-major storage only for the first matrix, and column-major storage for the second one (our fake transpose).

The main point of optimization with a transpose is that the column becomes a contiguous vector from a row in the transposed matrix.

Here's what the matrix multiplication algorithm looks like when it's working on a "fake" transpose:

```
void aussie_matmul_matrix_fake_transpose(
   const ymatrix m1, const ymatrix m2, int n, ymatrix mout)
{
    // Matrix-Matrix naive n^3 algorithm on a TRANSPOSE...
    for (int row = 0; row < n; row++) {
        const float* rowvec = &m1[row][0];
        for (int col = 0; col < n; col++) {
            float sum = 0.0f;
            const float* colvec = &m2[col][0];  // Row!
            for (int k = 0; k < n; k++) {
                sum += (rowvec[k] * colvec[k]);
            }
            mout[row][col] = sum;
        }
    }
}
```

Note that the above code assumes the transpose has already been computed. However, it is viable to compute a new transpose matrix in a preliminary step and still be faster, because transposing a matrix only adds an extra *O(N^2)* time to compute the transpose (and *N^2* storage space to store it temporarily), whereas the main matrix multiplication is *O(N^3)* time.

Perhaps surprisingly, this transpose method is much faster even without any vectorization. Because the column vectors are accessed in sequential order from contiguous memory, there is much better data locality for the memory cache, and also for any predictive pipelining happening in the cache. Here's the benchmark comparison:

```
Matrix-Matrix (MatMul) benchmarks (N=2048, ITER=1):
Matrix-matrix mult basic: 69479 ticks (69.48 seconds)
Matrix-matrix fake transpose: 47469 ticks (47.47 sec)
```

The transpose method is 31% faster with an unchanged basic MatMul algorithm. And all we did was permute two indices in a two-dimensional array. This code does exactly the same arithmetic computations as the naive version, but accesses memory in a different order, giving us a cache speedup.

There are various other small coding optimizations that can improve the transposed MatMul method further. The loop body could be partially unrolled by 4 or 8 iterations (or more).

Here's the C++ code of the version with an unrolling factor of 8 iterations:

```
void aussie_matmul_matrix_fake_transpose_unrolled8(
  const ymatrix m1, const ymatrix m2, int n, ymatrix mout)
{
    // Transpose Matrix-Matrix mult with 8 iter unroll
    aussie_assert(n % 8 == 0);
    for (int row = 0; row < n; row++) {
        const float* rowvec = &m1[row][0];
        for (int col = 0; col < n; col++) {
            float sum = 0.0f;
            const float* colvec = &m2[col][0];
            for (int k = 0; k < n; k += 8) {
                sum += (rowvec[k] * colvec[k])
                    + (rowvec[k + 1] * colvec[k + 1])
                    + (rowvec[k + 2] * colvec[k + 2])
                    + (rowvec[k + 3] * colvec[k + 3])
                    + (rowvec[k + 4] * colvec[k + 4])
                    + (rowvec[k + 5] * colvec[k + 5])
                    + (rowvec[k + 6] * colvec[k + 6])
                    + (rowvec[k + 7] * colvec[k + 7])
                    ;
            }
            mout[row][col] = sum;
        }
    }
}
```

Here are the benchmark results:

```
Matrix-Matrix mul (MatMul) benchmarks (N=2048, ITER=1):
Matrix-matrix fake transpose unroll 4: 15221 ticks (15.22 s)
Matrix-matrix fake transpose unroll 8: 12151 ticks (12.15 s)
```

Further tweaks are possible. The internal loop could be fully unrolled for a known vector size. Also, the initialization "`sum=0.0f`" could be removed by peeling the first iteration and starting the loop at "`k=1`".

Pointer arithmetic could be used to avoid loop indices and the double bracket accesses.

However, these are small fry, and we're now on the hunt for the Spanish mackerel of MatMul optimizations: *vectorization*.

Vectorized MatMul

Cache speedup is not the only benefit of the transpose method. Once we have column-major storage for the second matrix, then both the rows of the first matrix, and the columns of the second matrix are in contiguous memory. The computation is a normal vector dot product again on two vectors stored as arrays in memory (i.e., "`rowvec`" and "`colvec`" in the C++ code above). Hence, we can use all our standard vector dot product speedups again, including vectorization and hardware acceleration.

As an example, here's the AVX-2 vectorization of the transpose method using the FMA 256-bit intrinsics to do the vector dot product in parallel (see Chapter 13 for this AVX vector dot product code). This parallelizes the dot product by 8 elements at a time:

```
void aussie_matmul_matrix_fake_transpose_vecdot_AVX2(
  const ymatrix m1, const ymatrix m2, int n, ymatrix mout)
{
    // AVX2 Matrix-Matrix multiplication
    aussie_assert(n % 8 == 0);
    for (int row = 0; row < n; row++) {
        const float* rowvec = &m1[row][0];
        for (int col = 0; col < n; col++) {
             const float* colvec = &m2[col][0];
             mout[row][col] = vecdot_FMA_unroll_AVX2(
                              rowvec, colvec, n);
            }
    }
}
```

Here are the benchmark results:

```
Matrix-Matrix mult (MatMul) benchmarks (N=2048, ITER=1):
Matrix-matrix fake transpose AVX1: 19522 ticks (19.52 s)
Matrix-matrix fake transpose AVX2: 12747 ticks (12.75 s)
```

If anything, these AVX results are disappointing. Basic loop unrolling techniques (in the prior section) did better than AVX1 and the same as AVX2 vectorization. However, we haven't used AVX optimally inside the sequential code here. The AVX intrinsic calls should be moved up into the loop body without any function call overhead (i.e., inlining the function manually). I coded up that idea, and it made almost zero difference! I guess the C++ compiler is already inlining it, or function call overhead is a tiny percentage.

Further parallelization speedups would include using AVX-512 or AVX-10 intrinsics for vectorizing 16 elements in parallel. Also desirable are various further optimizations of the sequential code around any AVX intrinsics. The inner "`col`" loop could be fully or partially unrolled with multiple AVX sequences and/or optimized with pointer arithmetic.

Loop Tiled/Blocked MatMul

The triple-nested MatMul version with the vectorized inner loop is still nowhere near what is possible. There are three more ways to increase throughput:

- Data locality within the matrices.
- Pipelining of the SIMD instructions.
- Avoiding repeated loads of the same data.

The data locality of the basic AVX transposed MatMul algorithm is still far from optimal, although we fixed the most egregious problem by using the transpose. The algorithm is simply scanning down all of the dimensions, without really any attempt to maintain data locality.

The method of calling AVX intrinsics is simply doing "load, FMA, store" repeatedly along blocks of 4 or 8 elements, which does not allow for the natural pipelining of the FMA instructions. The loads and stores are interrupting the flow of computation.

Secondly, if you look carefully at the "load" operations that are happening in the sequence, you realize that it is repeatedly loading the same regions of the matrices.

Tiling or blocking the MatMul loops are far more effective. The basic idea is that instead of scanning sequentially, we process smaller square or rectangular "tiles" or "blocks" of the data, one at a time. Refer to Chapter 12 for the basic idea of tiling/blocking optimizations of nested loops. Data locality is the main aim of a tiled algorithm, but it also helps us achieve better pipelining of SIMD instructions, because we can load all the data in, and then perform multiple arithmetic operations on it without any intervening loads or stores. And since a tiled MatMul is iterating more carefully over smaller blocks of data within the matrices, there's also less redundant loading of the data overall.

Fast Matrix Multiplication Theory

The main techniques for faster matrix multiplication of general matrices include:

- Strassen's algorithm
- Winograd's algorithm
- Fast Fourier Transform (FFT) methods

Matrix multiplications can also be sped up by restricting our algorithm to only use matrices that are of special types:

- Low-rank matrix factorization
- Sparse matrices
- Special matrix methods (e.g., Butterfly matrices, Monarch matrices, etc.)

Each of these specialized matrix types can have a faster matrix multiplication kernel than using the all-purpose GEMM kernel. For example, sparse matrices can be stored in a compacted permuted-tuple format, with parallelization of permutation arrays for computation.

Approximate Matrix Multiplication. Approximate Matrix Multiplication (AMM) refers to a variety of complicated model optimization techniques that replace matrix multiplications with various approximations that avoid the cost of arithmetic multiplication, trading off some accuracy. These methods are usually distinct from quantization methods, are not specific to certain subclasses of matrices, and evoke more advanced mathematics in the theory of matrices.

Note that these algorithms apply at the high-level of how matrices are multiplied with other matrices or with vectors (e.g., avoiding some vector dot products), whereas there are also low-level optimizations of the arithmetic operation when multiplying two numbers. These two classes of approximation research are not the same, and are actually orthogonal to each other.

Multiplying by Transpose

The transpose of a matrix is commonly used in matrix multiplications, both as part of the algorithms and as a speedup. For example, this occurs in AI engines with the QKV matrix computations inside the attention heads, where the transpose of K is used, usually denoted as K^T in the algebraic formula.

Note that this is the actual algebraic use of the *real* transpose, as opposed to the idea of using a "fake transpose" to get column-major storage of matrices for easier vectorization. The code to compute the transpose of a matrix is shown below for a square matrix:

```
void aussie_matrix_transpose_basic(
    const ymatrix m1, int n, ymatrix transpose)
{
    // Transpose: in output matrix (square matrix)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            transpose[j][i] = m1[i][j];
        }
    }
}
```

The funny thing is that if we want to multiply a "real" transpose as the second matrix in some computation, then the original non-transposed matrix is the "fake transpose" of the "real" transpose. How awkward! But it's actually good, because we usually already have the original matrix in memory, and we don't even need to compute the (real) transpose. Instead, to do a MatMul of a matrix with this real transpose, we can instead use the original matrix as the second operand in the kernel that is based on the column-major storage of a fake transpose. Oh, dear, I feel like it's all circular and I'm digging myself into a word pit here! But it all works out in the end, and it's fast, which is really the one and only thing.

References

1. Ulrich Drepper (2007), *What Every Programmer Should Know About Memory*, November 21, 2007, http://people.redhat.com/drepper/cpumemory.pdf
2. Kazushige Goto (2008), *Anatomy of High-Performance Matrix Multiplication*, ACM Transactions on Mathematical Software, Volume 34, Issue 3, Article No.: 12, May 2008, pp 1–25, https://doi.org/10.1145/1356052.1356053, PDF: https://www.cs.utexas.edu/~flame/pubs/GotoTOMS_revision.pdf
3. Harald Prokop (1999), *Cache-Oblivious Algorithms*, Masters Thesis, MIT, June 1999, http://supertech.csail.mit.edu/papers/Prokop99.pdf
4. Intel (2023), *Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1*, August 2023, 248966-Software-Optimization-Manual-V1-048.pdf
5. Agner Fog (2022), *Vector Class Library (VCL)*, https://www.agner.org/optimize/vcl_manual.pdf
6. Sergey Slotin (2022), *Matrix Multiplication*, Algorithmica, https://en.algorithmica.org/hpc/algorithms/matmul/ Code: https://github.com/algorithmica-org/algorithmica/blob/master/content/english/hpc/algorithms/matmul.md

16. Tensors

What are Tensors?

Tensors are terrifying at first! I avoided learning about them for ages. All those nested loops are scary. But eventually it dawned on me that they're just three-dimensional arrays, and the computations are nothing harder than multiplication and addition.

An important point is that "tensors" in Computer Science are much different to the mathematical forms used in Physics. AI tensors are used in "linear algebra" for LLMs and are much more basic than the 4-D space-time tensors in Einstein's theory of general relativity. Which may explain why all those brainy physicists are so smug, despite being unable to predict if it'll rain tomorrow.

Tensors are simply multi-dimensional arrays, and are usually 3-dimensional. Each slice of a 3-D tensor is a two-dimensional matrix. And like vectors and matrices, tensors have these basic properties:

(a) Each element stores a single number (i.e., no strings or objects).

(b) All elements have the same data type (e.g., `int` or `float`).

(c) Elements may be positive, negative or zero.

(d) There are no missing elements. The concept of "missing" can only be represented by zero in a normal tensor.

There are exceptions, of course. There are "sparse tensors" that can represent elements as missing. Also, you can technically store strings or objects in a C++ three-dimensional array, but then it's more of a misuse of a tensor. Numbers are where it's at.

Tensors are technically the superset of all of the computational structures, and the number of dimensions is called the "rank" or "dimension" or "axes" of a tensor. Matrices are rank-2 tensors, vectors are rank-1 tensors, and even scalars are rank-0 tensors.

Conceptually, there's a hierarchy of complexity for tensor operations:

- 3-D tensor operations break down into 2-D matrix multiplications.
- 2-D matrix multiplications break down into vector dot products.
- 1-D vector dot products break down to a single `float` number (a scalar).
- 0-D scalars are single numbers.

Another way to think about tensors is in terms of nested loops. Scanning a vector requires one loop, and a matrix needs two nested loops. Tensor operations require three or more nested loops to process all their data.

Neural Network Tensors

I'm not going to take you in detail through the theory of how neural networks function. But in broad strokes, there are "neurons" in layers, where each neuron has a "signal," and there are also connections between neurons that forward the strength of a signal on to the next layer of neurons. So, each neuron in connected to *every* neuron in the previous layer by an "arc" and on that arc is a "weight" that says how strong or weak to consider the incoming neuron's signal.

But how do we get to tensors from that? Not obvious.

Let's step back a little and be one with the neuron. So, we are just one neuron in a layer of 100 neurons. And the previous layer has 100 neurons, and we are "fully connected" with arcs from every one of those 100 prior neurons. With 100 neurons in the previous layer, our little lonely neuron has to consider the signals from all of the 100 neurons in the prior layer, with 100 weights on the arcs to help decide how much attention to pay to each of the 100 prior neurons.

If we consider the previous layer of 100 neurons as a "vector" of each neuron's computed values. What this means is that every one of the 100 prior neurons has a number of its computed signal, so we have a vector of 100 signal numbers from the prior layer (i.e., a vector full of 100 neuron computed values).

Again, our little neuron has to receive a computed signal value from every one of the 100 prior layer neurons, so we have 100 arcs coming into our little neuron, each with a different number, that is the "weight" of that arc. The computed value of a prior neuron is multiplied by the "weight" that's on each arc (i.e., there's 100 weights, one for each arc). So, every one of the arcs from the 100 neurons in the prior layer has a weight, and what does that sound like? A vector of weights.

So, we have a bunch of 100 prior-layer neuron's computed values in a vector, where each one of those 100 signal values is multiplied by a weight that's in a vector with 100 weights. Hence, we've got to pairwise multiplication, where we multiply 100 neuron values times 100 associated weights. Hence, we've got a bunch of element-wise multiplications of two vectors (100 values times 100 weights), which creates a vector of 100 multiplication computations.

But our little neuron cannot have 100 computed values, but can really only have one number, the total computed signal for our current neuron. There are various things we could do to "reduce" our interim vector of 100 multiplications, but the simplest is to add them all up, and this gives us one number. Now we have one number, and it's the computed signal value for our current neuron.

Umm, I remember that from High School. If we multiply two vectors together with the numbers in pairs, and then add it all up: *vector dot product*.

In summary, we have a vector dot product for our single neuron in the current layer, based on two vectors from the prior layer (the vector of 100 calculated neuron values, and the vector of 100 weights).

But this is just for our one lonely neuron. Except, it's not lonely, since it has 99 friends, because it's in a layer of 100 neurons itself. So, our neuron and its 99 friends in the current layer, all have to do a different dot product computation because the weights are different for each set of arcs into each neuron. We have a whole vector of 100 neurons in the current layer, for which we have to compute dot products with 100 values times 100 weights (i.e., using the prior layer). So, we have to do 100 vector dot products to calculate the result for our neuron and its 99 friends. If we do 100 repetitions of vector dot products, this sounds like...*matrix multiplication*.

But that's not all. There's a third dimension based on the "tokens" in the prompt, which is represented by an "embeddings" vector. And with this third dimension thrown in, well, then it's a whole vector worth of matrix multiplications, and we get to a 3-D operation called a "tensor product." Tensors are three-dimensional blocks full of numbers (i.e., cubes or rectangular prisms), which generalize two-dimensional matrices, which generalize one-dimensional vectors, which generalize zero-dimensional scalars. And if you have any common sense, you've stopped reading this section by now, so I'm not going to try explaining this mind-bending tensor stuff any further.

Tensor Arithmetic

Tensors are a convenient and efficient representation of multi-dimensional data. Since complex computations may involve a lot of matrix multiplications, it is useful to represent a sequence of matrix operations as a tensor operation.

Importantly, the arithmetic performed is the same. Using a tensor is computationally efficient for parallelization of algorithms, and also mathematically concise for theoretical analysis, but is not some fantastically amazing matrix algorithm. It's just crunching lots of numbers with the standard matrix multiplication methods. Usually, it's the same as an array of matrices, where you do matrix multiplication on each one.

In practice, tensor kernels will send out different chunks of that computation all over the place for parallel speedup, but it's still computing the exact same numbers as if you did it all brute-force in nested loops. You could even follow along with a pen and paper, except that the computer is better because it won't forget to carry the negative sign.

Tensor shape. Another point is the shape of a tensor. I'm sure you know that matrices may be square or rectangular in shape, but can't be a skewed parallelogram or a circle. Yes, you're right, there are triangular matrices, but now you're messing up my nice clean point.

Anyway, a 3-D tensor can have different sizes on each of its three dimensions. Hence, a 3-D tensor can be a cube if all three sizes are identical, but usually they have the shape of a more general rectangular prism. And it still has a brick-like shape, and can't really represent a triangle, cone, or sphere. Tensors are much less scary if you sing *Everything is Awesome* while you code the nested loops.

Unary Tensor Operations

Like a 2-D matrix, there are various simple operations we can define on a single tensor. The various element-wise operations apply individually to each tensor item.

- Clear or set to a value
- Add or subtract a scalar
- Multiply or divide by a scalar

Similarly, we could apply a particular unary mathematical function to each element separately: square root, exponentiation, natural logarithm, and more.

Binary Elementwise Tensor Operations

Adding two matrices means simply adding each pair of elements in the matrix, which only works if the two matrices have the same size and shape. The same idea generalizes to the addition of tensor elements of two tensors with the same size (i.e., all three dimensions are the same). Hence, we can do element-wise binary arithmetic on each element in two tensors to create a third tensor of the same size:

- Addition or subtraction
- Multiplication or division
- Maximum or minimum

Note that element-wise multiplication of tensor elements is not "tensor multiplication" in the same way that matrix multiplication isn't just paired multiplications of the elements in two matrices. Such an element-wise multiplication is called the "Hadamard product" of matrices, and is so useless that I don't think I was ever taught that in High School. The Hadamard product is not what is used by normal multiplication computations, but I've seen a few research papers where it was proposed as an optimization (probably unsuccessfully). Matrix multiplication is more complex, with its row-by-column vector dot product multiplications, and so is generalizing that to tensors.

That's how we get to "tensor product" of two tensors. It's really just nested loops doing matrix multiplications on slices of each tensor. And then matrix multiplications are just nested loops doing vector dot products. Like I said, tensors are just three-dimensional arrays doing multiplication and addition.

Sparse Tensors

Sparse tensors occur when most of the values are zero. These are a generalization of sparse vectors and sparse matrices, and offer the same advantages: compressed storage and faster arithmetic operations (by skipping operations involving zero).

The level of sparsity required for optimization usually means 80-90% of the weights are zero. With so few non-zero values, tensor arithmetic involves fewer operations and the memory requirements are low (i.e., store only the non-zero weights). Such sparsity is often the result of a "pruning" optimization, but there are also obscure theoretical means to get sparse tensors using tensor algebra (let's not even go there!).

When there is a high degree of sparsity, such as when 80-90% of the values are zero, it becomes more efficient to use alternative algorithms.

Sparse tensors can be stored in a permutation index format, where only the index locations of non-zero items are stored (e.g., storing a four-tuple with the non-zero value and the three indices at which it is located in the tensor). Operations on sparse tensors can use the alternative storage format to create much more efficient kernels that avoid most of the computations involving the missing zero values.

Parallelization of sparse tensor operations is a double optimization, because there are fewer operations (only on non-zero weights), and you can parallelize them as well. Although a permuted index data format is not the usual contiguous memory space amenable to vectorization, there are other methods to vectorize permutation indices, such as with "gather" and "scatter" SIMD operations.

Part IV: Advanced Data Structures

"Design is the fundamental soul of a human-made creation that ends up expressing itself in successive outer layers of the product or service."

— Steve Jobs.

17. Algorithm Speedups

Algorithm Optimization Techniques

Changing the underlying algorithms used by the program is often the only real way to gain a large speed increase. In particular, the algorithms and data structures used can often be modified to give a significant speed increase. Is there a better way to do what your program does? Is it doing too much unnecessary calculation? Although much depends on the programmer's ingenuity, there are some common techniques for improving performance of algorithms.

- Parallelization and vectorization
- Precomputation (save time by using space)
- Recomputation (save space by using time)
- Caching and computation reuse
- Greedy algorithms (immediate computation)
- Skipping algorithms
- Arithmetic strength reduction
- Integer arithmetic
- Change recursion to loops
- Incremental algorithms
- Choose a better data structure

The idea of "skipping" computations also has various sub-methods:

- Lazy algorithms (delay computation until needed)
- Common case first
- Simple case first
- Approximate tests first

Lookup Table Precomputation

Lookup tables are so widely used in latency-critical programs that they're usually abbreviated as LUTs. The aim is to precompute results and replace frequently called costly function evaluations with table lookup (i.e., array references). Note that this use of precalculation is only worthwhile if some calculations are repeated and computing the same result.

As an example, we can replace a call to "`sqrtf`" with a precalculated table of square roots. In the subsequent calculations where square root is needed, a call to the `sqrtf` function is replaced by a table lookup.

The precalculation uses two separate functions: one to perform the precalculation, and another to access the values by table lookup. The precalculate function must be called once via a global initialization routine for the class. Alternatively, every call to the square_root function could self-check a `static` Boolean flag indicating whether the values have been precalculated yet, and call the precalculate function if not, but this is needlessly slower for every access.

Even more efficient is to use "offline precomputation" before your program even runs. This is a more efficient method whereby the data is not precalculated during initialization of the program, but is done earlier in an "offline" mode (e.g., as part of your build process). For example, the precomputed results are either stored to a data file, or converted to a C++ source file that is linked.

Another good example of precalculation is the Boolean functions on characters (e.g., `isupper`). To improve performance, it is possible to implemented these functions as a precomputed array of 256 `bool` values, or 256 bytes with 0 if `isupper` is false, and 1 if `isupper` is true. Then `isupper` is evaluated by indexing the character into the precomputed table:

```
#define isupper(ch) ( precomputed_array[ch] )
```

In fact, many C++ compilers implement `isupper` and other functions in `<ctype.h>` as a table lookup over the 256 characters (plus an extra one for `EOF`), with a precalculated single bit flag per function — that is, one bit indicating `isupper`, another bit for `islower`, etc.

Lazy Evaluation

The idea of lazy evaluation is a slight amendment to precalculation or data structure augmentation. Full precomputation during program startup can be inefficient when only some of the values are needed.

Lazy evaluation works in a "lazy" manner, by only doing work when asked. Instead of precalculating every result, results are calculated only as needed. To use this method, some way is needed of indicating whether a result is already in the table. When seeking a result, it is necessary to check if the required value is already present. If so, table lookup is used to get the result. If not, the value must be calculated, stored in the table and that entry marked as present.

The precomputation of `sqrtf` can be modified to become lazy evaluation by adding another array of Boolean flags, indicating which of the square roots have been computed. When calculating a square root, the function checks if it has been computed, and calculates it if not.

```
float square_root_lazy_eval(int n)
{
    static float sqrt_table[NUM_PREC + 1]; // values
    static bool precalc[NUM_PREC + 1];     // flags

    if (!precalc[n]) { // precalculated?
        sqrt_table[n] = sqrtf((float)n); // real sqrt
        precalc[n] = true; // Mark as computed
    }
    return sqrt_table[n];
}
```

The use of lazy evaluation is slower than complete precalculation if all of the values are eventually calculated, because of the overhead of checking whether calculation is needed. Also, there's only an efficiency gain for values that are calculated twice or more. However, lazy evaluation can make the program faster overall if not all calculations are needed, but some are needed many times. Any unnecessary calculations are avoided.

How lazy!

Source Code Precomputation

The examples of the precomputation of square roots in the previous two sections are not particularly efficient because they must still call the `sqrtf` function a number of times. A far more efficient alternative is to use C++'s compile-time initialization of arrays to set up the precomputed `sqrt_table` array inside the C++ source code. Hence, the square_root function becomes a simple lookup into an array variable as follows. Note that the array is declared as "`static`" so that the initialization occurs at compile-time.

```
float square_root_precalc(int n)
{
    const int NUM_PRECALC = 100; // Precalc to 100
    static float sqrt_table[] = {
      0.000000f, 1.000000f, 1.414214f, 1.732051f,
      2.000000f, 2.236068f, 2.449490f, 2.645751f,
      2.828427f, 3.000000f, 3.162278f, 3.316625f,
      //... etc .....
    };
    if (n >= NUM_PRECALC) return sqrtf((float)n);
    return sqrt_table[n];
}
```

The simplest way to produce the values for the precomputed array is to write another program to produce them. Once the values are produced, this program could be discarded, or it could be left in the build process. The following program was used to produce the declaration of `sqrt_table` used in the square_root function given above. The output from the following program was copy-pasted into the source code for the program above.

```
void generate_sqrt_table()
{
    const int NUM = 100; // Precalculate to 100
    printf("static float sqrt_table[] = {\n");
    for (int i = 0; i < NUM; i++) {
        printf("%ff", sqrtf((float)i));
        if (i + 1 < NUM)
            printf(", "); // comma after all but last
        if (i % 4 == 3 && i + 1 < NUM)
            printf("\n"); // newline every 4 numbers
    }
    printf("\n};\n"); // finish off declaration
}
```

Source code precomputation should always be more efficient than lazy evaluation and run-time precomputation. However, source code precomputation is only applicable when the function can be computed at compile-time (e.g., any "`constexpr`" function).

If the computation involves any variables whose values are known only at run-time, either lazy evaluation or run-time precomputation may be needed.

Incremental Algorithms

It is often easier to modify what has already been done than to start from scratch. This idea can be used to write faster algorithms. However, changing an existing algorithm to use incremental calculations will usually require a total redesign of the algorithm.

A simple example of an incremental algorithm is counting the number of symbols in a hash table. The non-incremental way to count them is to traverse the hash table, counting the number of entries along each hashed chain. The incremental method is to keeping a running count — increment it when a symbol is inserted; decrement it when a symbol is deleted. The incremental method is better if the count will be required many times. If the count is not required, there has also been a small amount of unnecessary overhead.

Another good example appears in graphics animation when managing the buffers. When displaying a new screen, it is usually more efficient to change the existing screen buffer than to redraw the whole screen. The idea is to set only those pixels that need to be changed.

For another example, a chess-playing program uses a game tree and the minimax algorithm with a static evaluation function. This function usually analyses the material balance (i.e., how many pieces each side has), along with other chess strategy factors. A simple but inefficient method of computing the material value of a position is to add the values of each piece on the 64 squares. The efficient incremental algorithm is to subtract the value of the piece from a running count whenever any piece is captured by the opponent.

Common Case First

When testing for a number of different conditions, it is best to test the most common case first. If it is true, the other tests are not executed. When using multiple `if-else-if` statements, place the common case first. For example, consider the binary search function:

```
if (key > a[i]) {
    // ...
}
else if (key < a[i]) {
    // ...
}
else { // equality
    // ...
}
```

Equality is least likely of all the three conditions, and hence it goes last. Greater-than and less-than are more common, so they go first.

The idea of common case first also appears in Boolean expressions using `&&` or `||`. The short-circuiting of these operators makes them very efficient when the common case is first. For `||`, the most likely condition should be placed first (i.e., most likely to be true). For `&&`, the most unlikely condition should be placed first (i.e., most likely to be false).

Simple Case First

This method is similar to common case first — the idea is to test the simplest condition first. More complicated and time-consuming computations can be avoided if the first test succeeds (or fails, depending on the context). This idea appears in two main situations:

- `if-if` construct (nested `if` statements), and
- logical operators (`&&` and `||`).

The simplest test should be the first of a pair of nested `if` statements and should also be the first operand of a `&&` or `||` operator. In the examples below, the sub-expression "`x!=0`" is evaluated first because it is the simplest and hence the least expensive to evaluate.

This is the nested-`if` example:

```
if (x != 0) {
    if (expensive_fn(x) != 0) {
        // ...
    }
}
```

This is the `&&` short-circuiting method:

```
if (x != 0 && expensive_fn(x) != 0) {
    // ...
}
```

Special Solution of Simple cases

In addition to putting a simple case first, it can also be efficient to solve simple cases differently to the general case. When solving a problem, simple cases can often be solved by specially designed fast functions. These "special solutions" can involve table lookup of precalculated values (e.g., storing the first ten factorials in an array) or just a fast algorithm for small cases (e.g., sorting less than five numbers quickly).

In general, the special solution of simple cases will give some speed increase if the simple cases are fairly common. The advantage of simple case precalculation over full precalculation is flexibility — it is not limited to those values that can be stored in a fixed size table.

The use of table lookup for simple cases for the factorial function is shown below. The use of the method here gives speed increase for all cases, not just the simple ones, because the recursive definition of factorial eventually breaks the problem down to a simple case.

```
int factorial_precalc(int n)
{
    const int NUM_PRECALC = 5; // How many
    static int s_precalc[NUM_PRECALC + 1] =
        { 1, 1, 2, 6, 24, 120 };

    if (n <= NUM_PRECALC)
        return s_precalc[n];
    else
        return n * factorial_precalc(n - 1);
}
```

Approximate Tests

Many algorithms can be improved by avoiding complex calculations with a fast preliminary test that is often successful. This is a special type of common and simple case optimization combined. This method is only worthwhile when avoiding the complicated test is highly probable; if avoiding it is unlikely, the extra simple test reduces efficiency because it adds (slightly) to the run-time cost.

Zero skipping. A common example of an approximation is "zero skipping." A low-cost test of a weight against zero can avoid the complexity of computing vector and matrix operations with that weight.

Bounding Sphere Tests in Ray Tracing. As an example in 3D graphics, to implement a ray tracing algorithm for graphical image rendering, it is necessary to determine whether a ray strikes an object. Since the objects are often complex and more often than not the ray will miss an object by a large amount of space, a simple test can be used to quickly identify rays that are close enough to the object to intersect with it. A good simple test is to determine if the ray intersects with the bounding sphere of an object, as it is relatively efficient to determine this. If the ray does intersect the sphere, the more expensive tests are applied to determine if the ray intersects with the object. If the ray does not intersect with the sphere, the cost of the more expensive tests has been avoided. Interestingly, the simplicity of testing the intersection of a ray with a sphere helps explain why there are so many ray-traced images of spherical objects.

Bounding-box 2D collision detection. The similar idea of a bounding rectangle is useful for collision detection in coding 2D arcade games. Collision detection usually involves testing many pairs of objects in a two-dimensional setting, and the tests are complicated because of the different shapes of the objects. The more complicated tests can be avoided by examining whether the bounding rectangles of each object are intersecting. If they do intersect, then a closer examination of whether the objects have pixels that overlap is carried out.

Rectangle Shapes. For yet another example of using a simple test to avoid complicated tests, consider the problem of a GUI-based drawing program. Typically, the user can select a vertex (e.g., the end of a line segment) by clicking "close" to the vertex. In other words, the user must click the mouse within a specified radius of the point. Hence, when the mouse is clicked, the program must compare the mouse location with all the currently active vertices.

The obvious method is to use the distance formula for two points and apply the following test on the x and y coordinates of the mouse and all points:

```
const float DISTANCE = 2.0f;
float diffx = xMouse - xPoint;
float diffy = yMouse - yPoint;
float distance = sqrtf( diffx * diffx + diffy * diffy);
if (distance <= DISTANCE) {
     // clicked! ...
}
```

Firstly, the efficiency of this test can be improved simply by avoiding the calculation of the square root. Squaring both sides of the equation gives the equivalent test:

```
float distance_squared = diffx * diffx + diffy * diffy;
if (distance_squared <= DISTANCE * DISTANCE) {
     // clicked! ...
}
```

However, the multiplications involved in computing the squares of the two sub-expressions on the left are quite expensive, although the square on the right-hand side will be a compile-time constant. A simple test can be used to avoid the expensive multiplications in most cases. If the difference between either the x or the y coordinates is greater than DISTANCE, then the points cannot be close enough. Although the cost of these tests is quite high because the absolute value over the difference must be found, it should still cost less than two multiplications, and will be more efficient if there are many widely spaced points to be tested. The code using this idea is:

```
bool check_pt_clicked(int xm, int ym, int xp, int yp)
{
     const float DISTANCE = 2.0f;
     int xd = xp >= xm ? xp - xm : xm - xp;
     if (xd > DISTANCE) return false;
     int yd = yp >= ym ? yp - ym : ym - yp;
     if (yd > DISTANCE) return false;
     return xd * xd + yd * yd <= DISTANCE * DISTANCE;
}
```

Of course, algorithm improvements are more effective. The best way of improving the efficiency of this program is to avoid the need for multiplications entirely, by changing the program specifications (!) so that the definition of clicking "close enough" to a vertex with a mouse refers to clicking within a *square* around the point, instead of a circle. Squares don't need multiplication.

Augmenting Data Structures

An interesting type of caching is where the data is stored inside the main data structure, rather than in a separate cache. Instead of recalculating derivative data every time you need it, a faster way is to store the data in the data structure. This is a form of caching that saves the time of recalculation, which need be done only once. If the data ever changes, the calculations must be redone and stored again. Hence, this method works best where data is unchanging, but can also tolerate modifications.

As an example of augmentation, consider a struct defined to represent a line segment (e.g., in a CAD drawing program). The struct contains four fields, for the x and y coordinates of the start and end points:

```
struct line_segment {
    int x1, y1; // Start point
    int x2, y2; // End point
};
```

Consider the computation of the length of the line segment, using:

```
float flen = sqrtf((y2 - y1) * (y2 - y1)
                 + (x2 - x1) * (x2 - x1));
```

If the length is a common calculation, it can be beneficial to cache the length of the line segment as an extra field in the `struct`:

```
struct line_segment {
    int x1, y1; // Start point
    int x2, y2; // End point
    float length; // Length of line segment
};
```

Whenever this length is needed during calculation it is immediately available as a field member. However, it is important to be careful that there is no consistency problem (where the `length` field is not the true length of the line segment). The main danger is that the `length` field won't be recalculated every time one of the other fields change.

18. Vector Algorithms

Vector Dot Product

Vector dot product is an algorithm that has received a lot attention lately, because it's the most basic computation algorithm in an AI engine. All tensor operations and matrix multiplications break down into many dot product calculations.

The dot product is so-named because its mathematical notation is a dot. It is also known as the "scalar product" because its result is a scalar, rather than a vector.

The vector dot product takes two vectors as input, and computes a single `float` number. The algorithm is a product of the elements of each vector, added together. Here's the code:

```
float vecdot_basic(float v1[], float v2[], int n)
{
    float sum = 0.0;
    for (int i = 0; i < n; i++) {
        sum += v1[i] * v2[i];
    }
    return sum;
}
```

Properties of the dot product include:

- Two vectors as input.
- Scalar output (single number).
- Can be positive or negative.
- Is zero if either vector is all zeros.
- Can also be zero for two non-zero vectors (e.g., if the vectors are "perpendicular" in 2-D or 3-D space).
- Has a physical meaning related to the "angle" between the two vectors.
- Is an integer if both vectors contain integers.
- Dot product of a vector with itself is the square of the vector's magnitude (equivalently, the vector's L2-squared norm).
- Is very slooow. Dot product-based operations inside matrices and tensors are the main culprit for AI needing all those GPUs.

The dot product differs from the "vector product" of two vectors (also called "cross product") that returns a vector, and is a completely different mathematical operation. The vector cross product is interesting mathematically in that it computes a vector perpendicular in 3 dimensions, but it's not very useful in practical applications. The dot product is where the action's at in big tensors.

Vector Norms

Vector norms are measurements of vectors that indicate features of a vector. For example, we can measure if two vectors are "close" to each other. Again, these used to be obscure linear algebra algorithms, but are now widely used in various AI algorithms.

Vector norms map vectors to a single number. Note that vector norms are not the same thing as the "normalization" layer in a Transformer (i.e., LayerNorm or BatchNorm). Note also that a vector "norm" is not at all related to the similarly-named "normal vector" (a vector perpendicular to a surface). The *norm* is a number, whereas the *normal* is a vector, and they're not on speaking terms since that incident last summer.

L2 Norm: The basic norm of a vector is the level-2 (L2) norm, and you probably already know it. This is the length of the vector in physical space, also called the vector's "modulus" or "magnitude" in Mathematics 101. If you treat a vector as a "point" in space, the L2 norm is its straight-line distance from the origin.

The calculation of the L2 norm of a vector is a generalization of Pythagoras's Theorem: sum the squares of all the vector elements, and then take the square root. The code looks like:

```
float aussie_vector_L2_norm(float v[], int n)
{
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += (v[i] * v[i]);   // Square
    }
    return sqrtf(sum);
}
```

Because we square every element, they all get turned positive. Zero squared is still zero. Once we've summed all the squares, we usually get a big positive number, which we then square root to get a smaller positive number. Hence, the result of the L2 norm is compressing a whole vector down to a single positive floating-point number.

The properties of the L2 norm are:

- Floating-point number (e.g., 0.567 or 5.6789 or 3.0 or whatever)
- Positive number (not ever negative)
- Zero only if the whole vector is zero.
- Represents the "length" (or "modulus" or "magnitude") of a vector, called the "Euclidean distance".
- Usually a non-integer, even if the vector was all integers.

For a simple 2-D or 3-D vector in Senior Math, the L2 norm is the physical length of the vector in 2-D or 3-D space (or the length of the line from the origin to the equivalent point). For AI, which has vectors in 1024-dimensions, or N-dimensional vectors for whatever N is being used, there's not really a physical explanation of the L2 norm that's easy to visualize, but it's kind of a measure of the length of the vector in N-dimensional space. The value of the L2 norm can be zero, but only if all the vector's elements are zero.

Note that the value of the L2 norm is not unique. Two different vectors can have the same value for the L2 norm. In fact, an infinite number of vectors can have the same value, and those vectors are the set of vectors with the same length (magnitude), which will define a sphere in N-dimensional space.

L2-squared norm: A minor modification of the L2 norm is the "squared L2 norm", which is, as you may have guessed, the square of the L2 norm. To put it another way, it's just the L2 norm without the square-root at the end. The code looks like:

```
float aussie_vector_L2_squared_norm(float v[], int n)
{
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += (v[i] * v[i]);  // Square
    }
    return sum;  // NOT sqrtf(sum);
}
```

The value of the L2-squared norm is a positive number, but a much larger one. The physical meaning is the square of the physical/Euclidean length of the vector. The L2-squared norm also equals the vector's dot product with itself.

Why use the L2-squared norm? Because it's faster to skip the square-root operation, of course. Also, if the vector contains integers, then the L2-squared norm is also an integer, which can make it even faster to compute in integer-only mode.

The L2-squared norm is just as good as basic L2 for some uses. The properties of L2 and L2-squared norms are very similar except that one is a much larger number. Both are positive and related to Euclidean distance, and both increase monotonically the further the vector is away from the origin.

Level 1 Norm: As you can guess from my calling it the L2 norm, there's also an L1 norm, and there's L3 norms, and more. Let's look at the L1 norm, because it's even simpler, although it's *not* usually something that's covered when studying vectors in Math class.

The L1 norm is simply the sum of the absolute values of all the vector elements. We don't square them. We don't take the square root. We just make them positive and add them up. The code looks like:

```
float aussie_vector_L1_norm(float v[], int n)
{
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += fabsf(v[i]);   // Absolute value
    }
    return sum;
}
```

Using the absolute values of elements reverses any negative vector elements to positive. The absolute value ensures the whole total can't go negative, and any negative value also adds to the total. A zero element is fine in the vector, but does nothing. The result of the L1 norm is a single positive float number, which can be fractional or whole, ranging from zero to as high as it goes (i.e., if you have big numbers in the vector elements, then the L1 norm will also be large).

The properties of the L1 norm are:

- Floating-point number (fractional or whole).
- Positive number (never negative).
- Zero only if all vector elements are zero.
- Physical meaning is an obscure distance measure (the "Manhattan distance").
- Will be an integer if the vector elements are integers.

What does an L1 norm mean? It's kind of like the distance you'd travel if you walked the longest way by going along each element/dimension of the vector, one at a time, and not going backwards (no negatives).

So, the L2 norm was the fastest diagonal direct way to get to a point, but the L1 norm is going the scenic route, and the L1 norm is usually bigger than the L2 norm.

Like the L2 norm, the L1 norm is not unique. Multiple vectors can have the same L1 norm. For example, the vectors `(1,2)` and `(0.5,2.5)` will have L1 norm value of 3.0. I'm not really sure what the set of all the vectors with the same L1 norm means. Maybe it's this: all the points that you can walk to from the origin when you travel a certain distance (going forwards-only)?

L3 Norms and Above: The mathematical vector norms can be generalized to L3 and higher norms, even to infinity. For an L3 norm, you cube all the vector elements (made positive by absolute value), and take the cube root at the end. It's tricky to find the cube root in C++ until you remember that a cube root is exponentiation to the power of 1/3 (from Year 10 math), so we can use the "`powf`" function. Here's the code:

```
float aussie_vector_L3_norm(float v[], int n)
{
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += (v[i] * v[i] * v[i]);  // Cube
    }
    const float frac_third = 1.0f / 3.0f;
    return powf(sum, frac_third);
}
```

Can you guess what an L4 norm is? The higher order versions are really fun and interesting if you wear socks with your sandals, but not very useful in any practical applications of AI coding.

Matrix Norms

There are norms for matrices, but they're not really that often used. Taking a "measurement" of a matrix via a "norm" (or a "metric") to compare it to other matrices isn't a common task.

The silly ones are element-wise matrix norms. You can define an L1 or L2 norm on a matrix using the same algorithm over all its elements. You can also find the maximum element inside a matrix, and call that the "max norm" if you like to sound math-ish. The reason I say these are dumb? Because they ignore the structure in the matrix, so it's a kind of "pseudo-norm" of a matrix. It's really just treating a matrix like it's a big, flat vector, and to me it seems more like misusing a vector norm on a matrix.

More sensible matrix norms consider the rows or columns of the matrices as separate vectors. An NxN matrix has N column vectors or N matrix vectors, so there are N vector norms. Should we add them up? No, taking the *maximum* of the vector-wise L1 or L2 row/column vector norms has a more useful meaning as a matrix norm than the element-wise matrix L1 or L2 pseudo-norms. You can do this maximum-of-vector-norms either for rows or columns, but not both.

Vector Min and Max

Finding the maximum or minimum element of a vector is useful, and somewhat relevant to the L1/L2 norms. The maximum is a kind of "metric" of the size of a vector. Also, the maximum function over a vector is used in "greedy decoding" to pick the word with the highest predicted probability, which is then output. The minimum function would give us the least likely word, which might also be interesting if useless.

The simple linear code for vector max is:

```
float aussie_vector_max(float v[], int n) // Maximum
{
    float vmax = v[0];
    for (int i = 1 /*not 0*/; i < n; i++) {
        if (v[i] > vmax) vmax = v[i];
    }
    return vmax;
}
```

The vector minimum function looks similar in sequential C++ code:

```
float aussie_vector_min(float v[], int n) // Mininum
{
    float vmin = v[0];
    for (int i = 1 /*not 0*/; i < n; i++) {
        if (v[i] < vmin) vmin = v[i];
    }
    return vmin;
}
```

These simple linear functions are crying out for optimizations: loop unrolling, pointer arithmetic, etc. However, what they really need is vectorization. There are parallelized *max* and *min* primitives in GPUs and CPU-based AVX intrinsics that you can use.

Top-K Vector Algorithm

The top-k algorithm is more complicated than vector max or min: find the largest *k* elements in a vector. Note that "maximum" is the same as top-k with *k=1*. If you want the short version of the top-k story in C++, there's a `std::partial_sort` function that sorts the top k elements, and there's also `std::sort` for a full array sort.

However, let's hand-code some top-k algorithms for more clarity.

Note that the top-k algorithm is a somewhat obscure algorithm that used to be rarely used, but now it's a very important piece of code in AI engines. It gives us "top-k decoding" which is how to choose which word to output. The whole encoder-decoder computes a vector giving us the probability that each word should be output next. Using the maximum probability word gives us "greedy decoding" which always outputs the most likely word. But that's kind of boring and predictable, so top-k randomly chooses between the *k* most likely words (e.g., top-50), which is still accurate and more interesting because it has creative variation.

Example: Hand-coded top-2 algorithm: Since top-1 is the maximum of a vector, we can also find a fairly simple linear scan for *k=2*. The basic idea is to scan through and keep track of the two largest values as we go.

```
void aussie_vector_top_k_2(float v[], int n, float vout[])
{
    // Order the first 2 elements
    float vmax1 = v[0], vmax2 = v[1];
    if (v[1] > v[0]) {
        vmax1 = v[1];  // Reverse them
        vmax2 = v[0];
    }
    for (int i = 2 /*not 0*/; i < n; i++) {
        if (v[i] > vmax2) { // Bigger than the smallest
            if (v[i] > vmax1) {
                // Bigger than both (shuffle)
                vmax2 = vmax1;
                vmax1 = v[i];
            }
            else {  // In the middle (fix 2nd only)
                vmax2 = v[i];
            }
        }
    }
    vout[0] = vmax1;  // Biggest
    vout[1] = vmax2;  // 2nd biggest
}
```

Note that the above top-2 algorithm is still impractical for our word decoding algorithm. We need to know not only the top probabilities, but also which two indices in the vector had those probabilities, because that's how we know which words map to which probabilities. So, we'd need to modify the above code to track and return the two array indices as well (or instead).

Shuffle Top-K Algorithm

For a larger value of *k* the code becomes more complicated. The above code for *k=2* motivates the general idea for a brute-force algorithm: shuffle sort the first *k* elements, and then scan the rest, shuffling any larger items up into place. We can merge the two shuffling phases into one block of code that handles both the startup and ongoing scan.

```
void aussie_vector_top_k_shuffle(
    float v[], int n, int k, float vout[])
{
    aussie_assert(n >= k);
    vout[0] = v[0];
    int nout = 1;
    for (int i = 1 /*not 0*/; i < n; i++) {
        float fnew = v[i];
        int maxj;
        if (nout < k) {
            vout[nout++] = fnew;
            maxj = nout - 2;
        }
        else {
            maxj = nout - 1;
        }
        maxj = nout - 1;
        for (int j = maxj; j >= 0; j--) {
            if (fnew > vout[j]) { // Shuffle & insert
                if (j + 1 < k)  // Shuffle down
                    vout[j + 1] = vout[j];
                vout[j] = fnew;
                // Keep going
            }
            else {      // Done.. insert it
                if (j != maxj) {
                    if (j + 1 < k)
                        vout[j + 1] = vout[j];
                    vout[j] = fnew;
                }
                break;
            }
        } // end for j
    } // end for i
}
```

The above example is a simplistic and inefficient top-k algorithm, not to mention that it was horribly fiddly and failed my unit tests for hours (i.e., it's a special kind of "fun"). Several loop optimizations suggest themselves: loop sectioning for the outer `i` loop to do the first `k` iterations as a separate loop (avoiding lots of tests against `k`), and loop peeling of the first iteration of the inner `j` loop (i.e., `j==maxj`). This version also should be extended to track the indices from where the top-k values came.

Theoretical Top-K Algorithms

There's a lot of theory about computing the top-k function of an array for large *k* values. These theoretical top-k algorithm papers mainly consider sequential processing, rather than vectorization. Even so, it's not a simple linear scan like *max* or *min* functions, but doesn't need to be as slow as shuffling.

Example: Top-k with qsort sorting: The simplest method for large *k* is to sort the array with a fast method (e.g., the quicksort algorithm) and then pick off the top *k* elements from the sorted array. In C++ there are the `std::sort` methods or the older style `qsort` function. Here's an example using the C++ standard `qsort` function:

```
int aussie_top_k_qsort_cmp(
    void const* addr1, void const* addr2)
{
    float f1 = *(float*)addr1;
    float f2 = *(float*)addr2;
    if (f1 < f2) return +1;  // Reversed (descending)
    else if (f1 > f2) return -1;
    else return 0;
}

void aussie_vector_top_k_qsort(
     float v[], int n, int k, float vout[])
{
    // Top-k with general k (qsort algorithm)
    // Sort the array
    qsort(v, n, sizeof(vout[0]),
          aussie_top_k_qsort_cmp);
    // Copy top-k elements
    for (int i = 0; i < k; i++) vout[i] = v[i];
}
```

Top-k with qsort and permutation array: We really need a version that returns the indices of the probabilities, rather than just their values. So, I coded up a `qsort` version that sorts via a permutation array, and then returns the top-k of these permutation indices.

```
void aussie_permutation_identity(int permut[], int n)
{
    for (int i = 0; i < n; i++) permut[i] = i;
}

float* g_float_array_for_qsort = nullptr;

int aussie_top_k_qsort_permutation_cmp(
    void const* addr1, void const* addr2)
{
    int index1 = *(int*)addr1;
    int index2 = *(int*)addr2;
    float f1 = g_float_array_for_qsort[index1];
    float f2 = g_float_array_for_qsort[index2];
    if (f1 < f2) return +1; // Reverse (descending)
    else if (f1 > f2) return -1;
    else return 0;
}

void aussie_vector_top_k_qsort_permut(
    float v[], int n, int k,
    float vout[], int permut_out[]
)
{
    // Create a dynamic permutation array
    int* permut_arr = ::new int[n];
    // Identity permutation
    aussie_permutation_identity(permut_arr, n);

    // Sort the array (by permutation)
    g_float_array_for_qsort = v;
    qsort(permut_arr, n, sizeof(permut_arr[0]),
       aussie_top_k_qsort_permutation_cmp);
    // Copy top-k elements
    for (int i = 0; i < k; i++) {
            permut_out[i] = permut_arr[i];
            vout[i] = v[permut_arr[i]];
    }
    delete[] permut_arr;
}
```

Top-k without sorting: Sorting the whole array is somewhat wasteful if we only want the top 50 elements. There are various faster top-k algorithms that don't fully sort the array. These algorithms are called a "partial sort" and can achieve the top-k output with better performance

Standard C++ top-k libraries: As mentioned earlier, the standard C++ libraries have support for sorting algorithms in `std::vector`, such as with:

- `std::sort` — full array sort (simplest idea).
- `std::partial_sort` — partial sort of *k* elements (faster).

There is a top-k specialized version in the modern C++ libraries called `std::partial_sort`, which sorts the top *k* elements of an array, which can then be selected for the top-k algorithm.

Presumably, the `std::partial_sort` function is a faster algorithm than `std::sort`, by not fully sorting the whole array, but I haven't tested it. There is also `std::nth_element`, which is similar to top-k.

19. Perfect Hashing

What is Perfect Hashing?

Perfect hashing is the extreme of hashing, where we guarantee that there's no collisions. Hence, the hash function is "perfect" because no pair of two keys map to the same hash value. This makes for a super-fast hash lookup with guaranteed *O(1)* search performance, and no need to look up a second hash location ever.

Perfect hashing is faster than normal hash tables. Regular hashing is fast on average, with *O(1)* average search, but collision resolution mechanisms like linear chaining or probing can have worst case *O(n)* search cost. Perfect hashing has guaranteed *O(1)* search complexity for best, average, and worst case. In fact, we don't even code up a collision resolution method at all.

Unfortunately, the good news stops there, because this only works in a very special situation: where the set of keys is known at compile-time. This hash table can only contain a fixed set of keys that we know whenever we build the perfect hashing code.

If there are any insertions or deletions, this idea doesn't work at all, and may require us to re-run and re-compile our perfect hashing engine if they occur. Thus, we can tolerate insertions and deletions but only if they are *rare*. Some examples of rarely changing sets of strings we might want to look up with perfect hashing include:

- Special keywords in a programming language tokenizer (e.g., 100 reserved words).
- Common English words in a grammar checker (e.g., 1,000 basic words).
- Stock tickers on an exchange's market data feed (e.g., about 5,000).
- Vocabulary words of an AI model (often 50,000 to 100,000 words).

Yes, the last one is a bit tricky, because tickers might change daily, in which case we might need to re-run our perfect hashing in every overnight build. Also, finding a perfect hash function for 100,000 LLM vocabulary strings in a reasonable amount of time might be a struggle.

Disadvantages of Perfect Hashing

We already mentioned the main disadvantage of perfect hashing, which is that it requires a known set of keys, or at least a very rarely changing set of keys. Other disadvantages include:

- Cost to build — expensive to scan the search space for a perfect hash map.
- Scalability problems — cannot handle a large number of keys because the search space becomes too large.
- Static data — insertions and deletions invalidate the hash map.
- Recomputations — increasing the key set requires a total re-run of the whole shemozzle.

Perfect hashing also has some of the disadvantages of a basic hash map like `std::unordered_map`, such as:

- Unsorted data
- Scanning all data is somewhat inefficient (and in unsorted order)
- Cache locality issues because objects are stored randomly in the hash table.

Perfect hashing is not perfect for every case. Some alternatives data structures to consider for search lookup optimization include:

- Bloom filters
- Tries
- Automata (precomputed)

Or you could put all your keys in an array and use a GPU to check all in parallel.

Perfect Hash Functions

Special hashing algorithms can be used in any situation where the search data is known at compile-time. The most efficient solution is to use hashing with a specially developed hash function, designed to prevent all collisions. This is called a *perfect hash function* and can only be developed for unchanging data.

If a perfect hash function can be found, the symbol table can be searched with one computation of the hash function and one key comparison to determine if the key is actually there at the index.

The most difficult aspect of using this method is the search for a perfect hash function for a particular set of data. There are a few common methods of doing so:

- Inspired guesswork
- Brute-force computation
- Use a perfect hashing tool (e.g., GNU gperf)

In some cases, the programmer can work out a function that has no collisions by guessing at a function. For example, if the programmer notices that all keys have a different first letter then it is easy to compute a perfect hash function as a mapping from the 26 letters to a different unique integer, the hash value. There's a curious fact unknown to most AI engineers, that humans are very resourceful and this method of "guessing" the function works surprisingly well. However, the average human might have a little trouble with 100,000 distinct keys.

The brute-force approach involves trying to generate the hash function using a computer which tries a number of different hash functions of a particular meta-pattern, applies the hash function to each key, and reports when a function that produces no collisions is found.

Further Optimizations of Perfect Hashing

The general complexity of perfect hashing is *O(1)*, which is true of the best case, average, and worst case complexity. Hence, it's fast for large sizes, but we still might want to optimize it a little more! There are two places to try to speed up:

- Lookup function (online)
- Perfect hash function creation (offline)

The basic method of perfect hashing can be optimized so that lookup is even faster. Some of the ways that we might super-optimize the search phase of the perfect hash function include:

- Not checking the key is present.
- Using a power-of-two hash table size.
- Larger hash table size.

Avoiding string comparisons. The computation sequence for a perfect hash lookup goes like this:

1. Calculate the perfect hash function.

2. Find that location in the hash table.

3. Compare the string at that location with our search key.

But why are we doing this string comparison at the end? That's quite slow. Well, sometimes we don't need to, and it depends on context. For a grammar checker or LLM tokenizer, we need to detect whether or not the key is there, because multiple words could map to the same hash location.

On the other hand, a market data feed from a US stock exchange might only contain our set of ticket names, so we can *assume* that only one string could possibly be at the hash table location. In other words, we're assuming that every string is found, and there are zero failed searches, so our hash table is mapping of the string to a set of data structures (e.g., our order book for that stock). That's all fine, and it will go faster, but the code will break completely if the exchange adds a new stock ticker!

Another way we could avoid the string comparison is to use two or more perfect hash functions. This data structure is known as a Bloom filter, and combines multiple bit vectors with multiple hash functions. Bloom filters are a probabilistic data structure that can confirm 100% that a key is invalid, but can only confirm that a key is likely to be valid, but not with 100% certainty.

Power-of-two hash table size. The size of the array that is our hash table is one of the parameters for a perfect hash function, so we have some control over it. Note this basic point: the hash table size must be more than the number of keys, or else it's a little hard to avoid collisions! In fact, it's easier to find a perfect hash function if the size is significantly more than the number of keys, so that there are some empty slots.

But what size? For some reason lost in the mists of time, everyone wants to choose a prime number, preferably a Mersenne prime, because that supposedly makes hash maps more evenly spread. But in the case of perfect hashing, we are looking for exact mappings with zero collisions, so it's perhaps not so important to use a prime.

Instead, we should use a power-of-two hash table size, because that allows the arithmetic in our perfect hash function to be faster.

The reason is that most perfect hash functions look like this:

```
offset = some_big_number(key) % N;
```

The `%` remainder operator is extremely slow, even on integers. The only reason it is used here is to ensure that the hash function maps to between `0` and `N-1`, where `N` is the hash table size. We can use "strength reduction" to use a faster arithmetic operation, such as:

- Bitwise-and (&) operator — if `N` is a power-of-two (e.g., for `x%16` use the bitwise operation `x&15`).
- Type cast to `unsigned char` — if `N` is 256 (8 bits)
- Type cast to `unsigned short` — if `N` is 65,536 (16 bits).
- Overflow of `unsigned char` — if `N` is 256 (8 bits)
- Overflow of `unsigned short` — if `N` is 65,536 (16 bits).

We've already examined a lot of these optimizations to modulo arithmetic in detail for the discussion of ring buffers in Chapter 21.

Larger hash table size. An important point about hash table sizes is that bigger can be better. This is true for both the offline computation of the perfect hash function, and the online search lookup. Bigger hash tables have more "gaps" and are an easier search space to find a solution. In terms of online search performance, a bigger table worsens cache performance, but that's not likely to be great for a hash table anyway. Furthermore, these extra gaps also mean that unsuccessful searches will be faster on average, because those keys that map to a gap can avoid the string comparison at the end. And memory is cheap, after all.

Offline search optimizations. The search for a perfect hash function can be very expensive, and even impossible. Some of the ways to speed things up include:

- All of the hash function optimizations.
- Splitting up the search space (partitioning).

The first point is that any optimization to the perfect hash function computation applies a thousand-fold to the offline search. For example, we also get faster computations possible in the offline search for a hash function if we only look at power-of-two table sizes. In fact, our offline code does a lot more of those computations.

Search space partitioning optimizations. The search space is combinatorial and explodes with large key sets. One approach is to split the keys into multiple perfect hash tables, such as by partitioning the key sets. Some of the ways to consider partitioning include:

- First letter — we can use 26 different perfect hash tables.
- Two letters — this gives 26*26=676 separate hash tables.
- Length of keys — e.g., stock tickers are at most 5 letters long.
- Preliminary hash — a simple hash function to start with (e.g., first two letters modulo a size smaller than 676).

Note that this means running the perfect hash engine multiple times to find a different perfect hash function for each partitioned set of keys. However, running 26 searches for smaller sets of keys will often run faster overall than trying to find one super-perfect hash function for every single key.

Example: ANSI C Keywords

As an example of the various approaches, let us attempt to develop a perfect hash function for a set of C's 32 keywords for a programming language tool:

```
auto break case char
const continue default do
double else enum extern
float for goto if
int long register return
short signed sizeof static
struct switch typedef union
unsigned void volatile while
```

Using my own version of "inspired guesswork", involving a couple of hours of poring over ASCII tables, I managed to come up with a reasonable perfect hash function. The basic approach I took was to break up the words into groups of about five keys by using a test of the string length, and also by making single character comparisons on the larger groups of keys with the same length. Once the group was small enough I looked for letters in the keys that were unique, often the first or second letter, and then examined the ASCII binary values of these letters. This way, the hash function extracts certain bits from each letter, and generates a small integer, which is then mapped into an "interval" of values for that particular group.

The function, which produces hash values in the range 0..36, is as follows:

```
int my_hash(char* s)
{
    switch (strlen(s)) {
    case 2: // Only "if" and "do"
            return (s[0] & 01) + 2; // 2..3
    case 3:
            return (s[0] & 01) + 8; // 8..9
    case 4:
        if (s[1] == 'o') // goto, long, void
            return (s[0] & 03) + 26; // 26..29
        else // auto, case, char, else, enum
            return ((s[1] & 14) >> 1) + 30;
    case 5: // break, const, float, short, union, while
        // First letter is unique
        return (s[0] & 07) + (s[0] == 'c') + 10; // 10..16
    case 6: // signed,sizeof,static,struct,switch
        if (s[0] == 's')
                return (s[5] & 03) + ((s[5] & 8) >> 3)
                + ((s[5] & 16) >> 2) + 18; // 18..22
        else // First not 's' - double, return, extern
                return (s[0] & 03) + 23; // 22..24
    case 7: // "typedef", "default"
        return (s[0] & 16) != 0;
    case 8: // continue, register, unsigned,volatile
        // First letter is unique
        return ((s[0] & 04) >> 1) + (s[0] & 01) + 4; // 4..7
    default: // Can't be a C keyword
        return 0; // Pick any number
    }
}
```

The second approach is to make the computer perform a brute-force search for a perfect hash function. The following program takes a set of keys from a file and develops a hash function of the following form:

```
( Σ C[i] * key[i] ) mod N
```

The code attempts brute-force computations with many combinations of the constants C[i] and N. If one of these hash functions produces no collisions, a perfect hash function has been found.

The source code below implements this concept.

```
//-------------------------------------------------------
// PERFECT HASH FUNCTION BRUTE-FORCE SEARCH
//-------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define LEN 10 // Maximum length of a word

char words[MAX][LEN]; // words being hashed
int C[LEN]; // coefficients of hash function

#define MAX_MULTIPLIER 1 // Let C[i] range 0..MAX_MULTIPLIER
// 0 means skip, 1 --> use addition
#define MAX_MODULUS 1000
int G_MODULUS;
int G_MODULUS_START_MULTIPLIER = 5;
int G_MODULUS_TOP;

// Apply the hash function coefficients to a key
int compute_hash_perfect(char* s, int modulus)
{
    unsigned int hash = 0;
    for (int i = 0; i < LEN && s[i] != 0; i++) {
            hash += s[i] * C[i];
    }
    return hash % modulus;
}

//-------------------------------------------------------
// Try all the combinations of coefficients
// This function finds the perfect hash function!
//-------------------------------------------------------

void perfect_hash_find_best(int nwords, int nstart)
{
  bool done = false;
  bool flags[MAX_MODULUS]; // has a key hashed here yet?
  int modulus = nstart * G_MODULUS_START_MULTIPLIER;
  do {
      // Do one possible modulus (table size)
      for (int i = 0; i < LEN; i++) C[i] = 0; // Clear coef
      do {
          // Update C[i] coefficients for next attempt
          C[0]++;
          for (int i = 0; i < LEN; i++) {
              if (C[i] <= MAX_MULTIPLIER) break;
              C[i] = 0;
              if (i + 1 < LEN) { C[i + 1]++; }
          }
          memset(&flags, 0, sizeof flags);
          // Scan all strings to count collisions...
          bool collision = false;
          for (int num = 0; num < nwords; num++) {
              int val = compute_hash_perfect(words[num], modulus);
              if (flags[val]) {
```

```
                    collision = true;
                    break;
                }
                flags[val] = true;
            }
            if (!collision) { // report success!!
                printf("NO COLLISION: ");
                for (int i = 0; i < LEN; i++) {
                    printf("%2d ", C[i]);
                }
                printf(", MODULUS = %d ", modulus);
                if (modulus == nstart)
                  printf(" PERFECT!!! (n=%d)", (int)nstart);
                printf("\n");
                break; // exit do loop. Do next MODULUS
            }
            done = true;
            // Finish when all coef are up to MAX_MULTIPLIER
            for (int i = 0; i < LEN; i++) {
                if (C[i] < MAX_MULTIPLIER) {
                    done = false;
                    break;
                }
            }
        } while (!done);
        if (done) {
            printf("FAILED with MODULUS %d\n", modulus);
        }
        modulus--; // Try the next modulus value
    } while (modulus >= nstart);
}
```

As shown in the source code above, the program is set to find all hash functions where the coefficient is either 0 or 1. These functions are a useful special case, as no multiplications are actually needed (all the characters with a 1 coefficient are simply added). When the program is run as shown on the ANSI C keywords as inputs, the best hash function it produces has modulus 134 (i.e., hash table size 134) and the following coefficients:

```
NO COLLISION:  1  0  1  1  1  1  1  0  0  0 , MODULUS = 134
```

This information can be coded up into a simple perfect hash function. Unfortunately, the memset and strncpy calls are necessary to ensure that characters beyond the end of the string are considered zero, as is assumed by the hash function generator.

```
int computer_hash(char* s)
{
    char s2[10];
    memset(s2, 0, 7);  // zero the first 7 letters
    strncpy(s2, s, 7); // copy up to 7 letters
    return ((int)s[0] + (int)s[2] + (int)s[3]
          + (int)s[4] + (int)s[5] + (int)s[6]) % 134;
}
```

This is not a *minimal* perfect hash function for these 32 keys. If the records to be stored with these keys are quite large, the space wastage of 134 hash table entries may be too large. A simple method of overcoming this is to add an array of 134 small integers (i.e., using the char type), where each entry in this array sets each C keyword to a unique value in the range 0..31. On the other hand, this may be a de-optimization as a sparse hash table can be more efficient than a minimal perfect hash function. If the table is large, it becomes likely that an unsuccessful search will map to a location containing a null pointer entry, and this avoids the need for the key comparison.

Perfect Final Thoughts

These computations we found here are not *minimal* perfect hash functions. If the stars align, you can sometimes find a mapping that works with the hash table size exactly equal to the number of keys. It might take a lot of CPU juice to find one, though. Good luck with that!

All of the hash functions in this section (both human and computer-generated) have multiple limitations, such as:

- ASCII-specific — not portable to the EBCDIC set or other character sets.
- Little endian — I haven't checked portability to big endian machines.

Finally, if you'd rather use a tool for perfect hashing than have as much fun as I just did, you can use the GNU gperf tool, which is a perfect hash function generator. GNU gperf will output the perfect hash function in C++ for you, and is highly customizable.

Extensions

1. Generalize the perfect hash functions to use parallel arithmetic in the hash function computation, such as AVX or ARM Neon SIMD instructions on a CPU or GPU kernel calculations.
2. Parallelize the search for a perfect hash function on either a CPU (e.g., AVX or ARM Neon functions) or on a GPU (e.g., in CUDA C++).
3. Implement multiple perfect hash functions on the same set of keys to get a Bloom filter data structure, where the string comparison can be omitted during lookup.
4. Try out the GNU gperf tool for one of the data sets.

20. Memory Pool Optimizations

What are Memory Pools?

Memory pools are a C++ optimization where you take control of the memory allocation used for a class of objects. The basic idea is to store all objects of the same type in a big array, next to each other, rather than being spread out over the heap wherever the `new` operator decides to put them.

Memory pools are a general optimization that can be used in C++ with the `new` operator, and also in C with `malloc`. Related data structures include:

- Bucket array
- Hive

A bucket array is like a memory pool, in that it's a big memory block, and you put your objects in there. However, a bucket array usually handles erasing an object by simply marking it as invalid using a Boolean flag. The memory for an erased object is not usually re-used when you insert a new object.

A hive is a generalization of a bucket array, whereby a hive can dynamically expand and contract the number of buckets. Notably, there's a `std::hive` class to use in C++26, which would make a good basis for an advanced type of memory pool. However, we're going to examine some of the simpler types of memory pools first.

Why Memory Pools?

Other than being a fun and gritty project in low-level C++ coding, the goal is speed, and this is achieved in various ways:

- Preallocation — no need to allocate memory on a low-latency hotpath.
- Fewer allocation calls — one big chunk rather than lots of small ones.
- Fewer deallocation calls — reusing memory addresses within the pool.
- No memory fragmentation —don't mix small and large allocations.
- Less memory overhead — hidden "control blocks" are not needed.
- Cache locality — all objects are stored contiguously.

In fact, you can even get the number of memory allocations for your class down to zero, if you really want to, by using a global memory pool object. Even the memory pool is not on the heap! But this only works for a fixed-size memory pool, and thus, only if you're really sure you won't need too many objects.

Memory fragmentation is also a slowdown that can be avoided or reduced with memory pools. The problems with fragmentation arise in two ways:

- Frequent allocations and de-allocations, and
- Different-sized memory blocks.

A memory pool is helpful in both respects. The memory pool avoids lots of allocations by using one big block, and avoids deallocations by re-using the locations inside the block. And because the memory block stores lots of blocks of the same size, we aren't mixing up different size allocations.

Disadvantages of Memory Pools

Firstly, this whole idea of memory pools is only about reducing allocated memory on the heap. This optimization is not relevant for objects stored on the stack (i.e., local variables), or static objects, such as global scope objects or `static` data members.

The other disadvantages of memory pools include:

- Fixed maximum number of objects (in the basic versions).
- Only works for single-sized objects (e.g., one class).
- Need one memory pool object for each type of object (via templating).
- Not useful for optimizing variable-sized objects (e.g., strings).
- Allocating too much memory in one massive chunk.

However, we can work around a lot of these disadvantages by using a templated class for our memory pool. The optimization of memory pools is a general algorithm that works for all types of objects.

Memory Control Block Overhead

Whenever you allocate memory on the heap, using the new operator or the old-style malloc function, it returns you the address of the block. But that's not actually the start of the *real* memory block.

There's actually an extra memory control block stored before that address. It contains meta-information about the memory block, which is used by the C++ standard library to keep track of things. For example, the size of the memory block is stored in that control block.

Whenever you deallocate a memory block by sending the address to `delete` or the `free` function, the standard library knows to look backwards a few bytes. Hence, it can find the size of the memory block, which helps it to deallocate the full block of memory. You don't need to worry about it, because the standard library takes care of it.

Hence, if you create a memory pool from one big chunk to contain 100 objects, rather than 100 separate calls to the `new` operator, there are 99 fewer memory control blocks. This is why memory pools reduce the memory overhead from your objects.

Fixed-Size Memory Pool Algorithms

For simplicity, we're going to limit our first memory pools to just one huge block of memory. This means that we can choose the overall capacity of the memory pool, but we can't increase it later by adding a second big block. This makes our memory pool more like a vector or array, rather than a dynamic bucket array or hive.

Even with these restrictions, there are still quite a few choices to make about designing our memory pool algorithm. Some of the alternatives include:

- Boolean flag — storing an "active" flag in each object.
- Index array — maintaining a list of indices of free blocks as a "free list" (instead of a per-object flag).
- Pointer array — tracking the free list via pointers.
- Permutation-based free list approach.

In the first case, we only have one array, and each block contains the "active" flag along with the stored user objects. In the other cases, we maintain two arrays, one of the user's objects, and another as the free list (with either indices, pointers, or permutations).

Boolean Flag Memory Pool

This is the simplest approach, but not the fastest. Let's examine it to get some of the basic ideas.

Some of the interesting features of this code include:

- Boolean flag — stored as a data member in every memory pool record.
- Pointer arithmetic — used in computing the offset when erasing an object.
- Incremental count — increment on allocation, decrement on release.
- Compile-time size —the simpler `std::array` not `std::vector`.

Here's the basic layout of the memory pool class.

```
template<typename T, int N>
class MemoryPool {
    struct Node {
        T data;
        bool active;
    };
private:
    std::array<Node, N> arr_;
    int nextfree_;
    int ct_;
    // ...
};
```

The constructor has to set all the "`active`" flags (although using `memset` would be faster than a loop):

```
MemoryPool() : arr_(), nextfree_(0), ct_(0) {
  for (int i = 0; i < N; i++) arr_[i].active = false;
}
```

The code maintains the index of the "next free" object. Initially, it's increasing as the first blocks get used, but later it's necessary to scan linearly.

```
int find_next_free(int offset) {
    if (offset == -1) offset = 0;
    int i = offset;
    do {
        if (!arr_[i].active) return i; // Found
        i = (i + 1) % N;
    } while (i != offset);
    return -1;  // It's full!
}
```

Here's the code for the allocation of a memory pool block:

```
T* alloc() {
     if (full()) return nullptr; // fail!
     assert(nextfree_ != -1);
     int oldindex = nextfree_;
     arr_[oldindex].active = true; // Not free
     nextfree_ = find_next_free(nextfree_);
     ct_++; // Incremental count
     return reinterpret_cast<T*>(&arr_[oldindex]);
}
```

And here's the code whereby a block is released by the caller. Note that the index computation requires pointers converted to the correct type. This code has some safety checks that are quite expensive, and might later be removed for production usage.

```
void erase(T* addr) {
    assert(ct_ >= 0);
    Node* nptr = reinterpret_cast<Node*>(addr);
    if (nptr >= reinterpret_cast<Node*>(&arr_[0])
      && nptr <= reinterpret_cast<Node*>(&arr_[N - 1])
       ) {
        // Valid pointer...
        int offset = nptr - &arr_[0];  // Ptr arith
        assert(nptr->active);
        nptr->active = false;  // Free now
        ct_--;  // Incremental count
        if (nextfree_ == -1) { // Was full?
            nextfree_ = offset;
        }
    }
    else { // Invalid pointer...
        assert(false);
    }
}
```

Constructor inefficiency. This implementation has a high-level slug if the memory pool is instantiated for use with a non-trivial class type. The definition of `std::array` will cause the constructors for every single object to run needlessly on the empty storage bytes, when the memory pool is first created or defined.

The solution here is simply to use bytes instead of the class type for the storage declaration:

```
struct Node {
    unsigned char data [sizeof(T)]; // Raw object
    bool active;
};
```

But we also need to be careful of memory alignment in this situation. The template could be instantiated on any type, some of which will need aligned addresses. Character addresses won't get automatically aligned, so we have to use `alignas` specifier. However, it's hard to fix in this implementation, because I cannot use `alignas(alignof(T))`. The extra "`active`" flag in the structure is messing everything up. But that's only one disadvantage of this method.

Disadvantages of Boolean Flag Method

The first point to remember is that this memory pool is a significant optimization. It achieves all the advantages of a memory pool as outlined above: preallocation, fewer allocations and deallocations, less memory fragmentation, and so on. Hence, it's a good start, and a worthy improvement to our classes.

We could stop now, and go home with a smile on our face.

However, it's not optimal. There are even better ways to code up a memory pool. The suboptimal features of this version of a memory pool include:

- Mixing hot and cold data
- Alignment issues for some types
- Extra padding bytes needed
- Slow insertions

One problem with the above approach is that it mixes "hot" and "cold" data. Your objects are probably hot areas of processing that are doing whatever you need. The Boolean flags are only used by the memory pool when inserting and deleting objects, and are thus cold data for the main processing algorithms. It would be better for cache locality if the cold data was separated from our hot objects.

Memory size is also not optimal. By adding a single Boolean variable to each object, it's not just 1 byte extra, because the compiler probably has to add a number of padding bytes to meet the alignment requirements (depending on what's inside your objects).

This will increase the memory size, and worsen cache locality when processing multiple objects.

However, the main problem with the Boolean flag approach is that it's slow. In fact, it has worst case *O(n)* performance for an insertion, because it might have to scan the entire array to find a free block. This worst case won't happen initially, but the performance can degrade as the memory pool fills up, and we do lots of insertions and deletions.

We can do better!

Boolean Flag Array Method

One way that we can address some of these issues is by separating all of the Boolean "`active`" flags into a different array. Rather than storing a flag in each object, we just store the user's object in the main block, and have a second block that contains the Boolean flags.

The advantages are that it fixes the hot-cold data problem, addresses alignment concerns, and the compiler won't need to add extra padding to the array of user objects. The array of Boolean flags should be one byte per object, but stored in a different array.

Firstly, we move the "active" flag out of the structures:

```
struct Node {
    unsigned char data[sizeof(T)];  // Raw object
};
```

And put it into a separate array:

```
bool activearr_[N];
```

The handful of places that used the "`active`" flag need to be changed to the "`activearr_`" array member.

We can also fix the alignment issues using the `alignas` and `alignof` specifiers:

```
alignas(alignof(T)) std::array<Node, N> arr_;
```

Bit packing. This active flag array method can be further improved by using bit packing. We only need one bit flag per object, rather than one byte each. Hence, we can pack them all into an array of 64-bit `unsigned long`, and can check for a free block using one integer comparison, testing 64 memory blocks at a time.

In practice, this version is pretty fast. Even so, it is technically still an *O(n)* worst case algorithm for insertion or deletion with large numbers of objects. And there are a few ways to fix that.

Index Array Memory Pool

The faster solution is to maintain an array of integer indices for the free locations. The advantages of this index array approach over the earlier "active" flag method include:

- Insertion and deletion always have *O(1)* complexity.
- Separates hot data from cold data.
- No extra padding bytes needed.

Here's the basic definition of the class:

```
template<typename T, int N>
class IndexMemoryPool {
    struct Node {
        unsigned char data[sizeof(T)]; // Raw object
    };
private:
    alignas(alignof(T)) std::array<Node, N> arr_;
    int freelist_[N];  // array free (stack-like)
    int ct_;
    int ctfree_;
// ...
};
```

Some of the basic primitives are simple:

```
bool empty() { return ct_ == 0; }
bool full() { return ct_ == N; }
int capacity() { return N; }
int count() { return ct_; }
int count_free() { return ctfree_; }
```

The index array is a "free list" that tells us where to find a free memory block. After a lot of insertions and deletions, if functions a lot like a stack of free locations. At the start, it's a fixed-size stack that's full with the index of every element available.

```
IndexMemoryPool() : arr_(), ct_(0), ctfree_(N) {
    for (int i = 0; i < N; i++) {
        freelist_[i] = i;  // Store all indexes
    }
}
```

When we allocate a new block, that's a "pop" of the stack, because we're removing from the free list:

```
int pop_free_index()
{
    assert(ctfree_ > 0);
    int index = freelist_[ctfree_ - 1];
    assert(index != -1);
    freelist_[ctfree_ - 1] = -1; // Clear it
    ctfree_--;
    return index;
}
```

The allocation of a block is mostly a call to this "pop" of the free list:

```
T* alloc() {
    if (full()) return nullptr; // fail!
    int index = pop_free_index();
    assert(index != -1);
    ct_++; // Incremental count
    return reinterpret_cast<T*>(&arr_[index]);
}
```

And the reverse is true when the caller releases a memory block. This is a push of a newly free index onto the stack.

```
void push_free_index(int index)
{
    assert(ctfree_ < N);
    freelist_[ctfree_] = index;
    ctfree_++;
}
```

And here's the version to release the memory:

```
void erase(T* addr) {
    assert(ct_ >= 0);
    Node* nptr = reinterpret_cast<Node*>(addr);
    if (nptr >= reinterpret_cast<Node*>(&arr_[0])
      && nptr <= reinterpret_cast<Node*>(&arr_[N - 1])
        ) {
        // Valid pointer...
        int offset = nptr - &arr_[0];
        push_free_index(offset);
        ct_--;  // Incremental count
    }
    else { // Invalid pointer...
        assert(false);
    }
}
```

In summary, both push and pop of the free list stack are very efficient with O(1) complexity. Everything in this index array version has constant-time efficiency.

Memory Pools Versus Containers

Why do you need a memory pool? Why not just use the standard C++ containers for your objects? Isn't a memory pool about the same as `std::vector`?

Yes and no.

Yes, a memory pool for your objects is very similar to managing them all in a standard vector. After all, the memory pool code can use a `std::vector` object inside it as the big pool. So, yes, you can manage your objects in a standard vector if you:

- Use a single `reserve` or `resize` call to allow the vector memory in one call.
- Keep track of objects going in and out of the vector.

In other words, it's almost the same thing as writing a memory pool, except it's mixed in the middle of your application's main logic.

Hence, no, it's not quite the same thing. There are two types of containers:

- Contiguous storage containers — it's very similar.
- Maps, sets, hash tables — memory management performance gains.

We'll examine vectors and arrays in a minute, but first let's look at the other containers. There are two aspects to use normal memory allocation and storing your objects in these advanced containers:

- Allocating memory for your objects — you've improved nothing (it's one allocation call per object).
- Extra container allocations — the container also needs memory allocation and a memory pool doesn't help with that.

But for the containers based on contiguous memory, the issue is less clear cut. The standard containers based on contiguous storage include:

- `std::vector`
- `std::array`
- `std::inplace_vector` (C++26)

When you compare a memory pool to using a standard vector of your objects, there is less gain to performance. However, creating a memory pool as a standalone class has several practical advantages:

- Separate memory management optimizations from business logic.
- Ensures only a single (huge) memory allocation occurs (or only a few if it's dynamic).
- Callers of the interface or API don't need to know about the memory management aspects.

Creating a memory pool as a separate idiom is good for encapsulating the performance optimization aspects of memory management. It encourages modularity by isolating high-level business logic from low-level resource management.

Advanced Memory Pools

Higher-level improvements to the memory pool interface are also possible. Most of the discussion here has been about a memory pool for one type of class, with a focus on reducing the number of distinct blocks requested on the heap. More advanced memory allocators are well-known, and they offer a variety of generalized performance optimizations:

- Thread safety (e.g., a single mutex or a lock-free version).
- Multiple object types supported in the memory pool.
- Dynamic size of objects allowed by allocating multiple large "pools" or memory chunks.
- Downsizing the memory pool if fewer objects are required.
- Intercepting the class-specific `new` and `delete` operators.
- Placement `new` operator — does not really allocate memory!
- Custom allocators — memory pools via allocator functor objects.

Even more general than memory pools is the concept of "custom allocators." The idea with custom allocators is not just to enhance the memory handling of a few classes, but to take over the whole memory allocation shemozzle from the standard library.

Extensions

1. Build your own simple memory pool templated class.
2. Add a memory pool to your object class by overloading a set of class-specific `new` and `delete` operators, sending these requests to the memory pool instead.
3. Code up multiple types of memory pools and measure their performance.
4. Generalize your memory pool class to dynamically manage multiple big chunks of memory, rather than just one.
5. Implement an advanced dynamic memory pool using the new advanced container `std::hive` (C++26) as the underlying data structure, rather than a vector or array.

21. Fast Ring Buffers

What is a Ring Buffer?

A ring buffer is an array-like data structure where the data moves around in a "ring" so that the end wraps around to the beginning. It's also known as a "circular buffer" and is often what is meant when people talk about a "fixed-size queue."

A ring buffer is stored in a single array or vector of contiguous data, but is not accessed in the same idiom. The data is processed in a FIFO (First-In-First-Out) idiom, where items are added to the "tail" of the queue, and removed from the "head" for processing.

Hence, a ring buffer is a good data structure for implementing a fixed-size queue or dequeue (double-ended queue).

Some of the main design decisions when implementing a ring buffer involve error handling:

- Overflow — inserting into a full buffer
- Underflow — removing from an empty buffer

Should the ring buffer throw an exception, or just return a Boolean failure status to the caller?

Simple Ring Buffer

A basic ring buffer data structure has three main elements:

- Array or vector of objects (fixed-size)
- Head index (integer)
- Tail index (integer)

Here's some code using `std::array` for a ring buffer:

```
template<typename T, int sz>
class RingBuffer {
private:
    std::array<T, sz> arr;  // Fixed-size array
    int head;
    int tail;
    // ....
};
```

New objects are inserted at the tail, and retrieved for processing from the head. In a typical implementation, the progression goes from left to write, using a "+1" idea for the next location. Technically, the ring buffer data could be handled in reverse order, but the forward progression around the ring is simpler and allows marginally more efficient arithmetic because there are no negatives to handle.

Thus, the basic primitives needed by a ring buffer:

- Insert at the tail
- Remove at the head

Here's the basic insertion method:

```
bool push(const T& x) {
    int newtail = (tail + 1) % sz;
    if (newtail == head) {
        // Overflow (full)
        return false;
    }
    tail = newtail;
    arr[tail] = x;
    return true;  // success
}
```

And here's the "top" method for an interface that allows "top" to access, and "pop" to remove:

```
T top() {
    if (is_empty()) {
        // Underflow
        return T(0);
    }
    return arr[head];
}
```

The "pop" method actually removes the item from the ring buffer:

```
void pop() { // Just remove (no return)
    if (is_empty()) {
        // Throw exception? (optional)
        return;
    }
    else {
        head = (head + 1) % sz;
    }
}
```

And there are also various simple primitives:

- Capacity — the fixed-size of buffer.
- Empty — zero elements
- Full — fixed-size array is full.

The code is reasonably simple:

```
int capacity() const { return sz; }
bool is_empty() const { return head == tail; }
bool is_full() const { return (tail+1) % sz == head;}
```

Pros and Cons of Ring Buffers

The main advantage of a ring buffer is that it has contiguous data. This means that our fixed-size queue should be faster to access than one stored as a linked list using `std::queue`.

The main disadvantage of a ring buffer is that it has a fixed size, unlike `std::queue`, which grows dynamically.

This ring buffer size doesn't necessarily need to be known at compile-time, but does need to be set when you initialize the ring buffer. There are also more advanced types of ring buffers which use multiple arrays, which can be dynamically grown in size.

The other disadvantages are that the ring buffer is very specific to a FIFO access pattern.

It's not a fast data structure for these operations:

- Searching for a value
- Sorting data
- Inserting at a random location (rather than the tail)
- Deleting from a random location (rather than the head)

Insertions and deletions are slow because they require a "shuffle" of all objects. Note that there's an interesting wrinkle: we could make insertion and deletions fast if we don't mind violating the FIFO ordering and moving objects around (invalidating any pointers or iterators referencing them). The idea is that the ring buffer becomes like an unsorted array (with wraparound):

- Fast random insertion — move the current element at the insertion location to a free location at the end of the ring buffer, then insert.
- Fast random deletion — move the last element to the location we are deleting from.

It's not all bad news. The data in a ring buffer is mostly stored contiguously, so there are some operations that still have good cache locality properties:

- Scanning or visiting all data elements
- Random access of data by integer index

A linear scan of all the elements can be quite fast, provided you don't mind that it's unsorted (or rather, it's sorted by order-of-insertion). The data elements are always in one or two contiguous data blocks, which is better than dispersed data structures like linked lists or binary trees. However, it's not quite as fast as an array or vector of objects, which is always one contiguous block.

Accessing one of the objects via an integer ordinal is still quite fast (i.e., `0...n-1`). Mainly, it's just some integer arithmetic with head and tail to find its array offset in the ring buffer.

Incremental Count Optimization

Computing the count of how many elements are currently inside the ring buffer is somewhat tricky: In the above computations, we can compute the "count" of how many elements are in the buffer using arithmetic on head and tail indices.

```
int count() const {
    return (tail >= head)
       ? tail - head
       : sz - (head - tail);
}
```

An alternative that can be faster, if the `count()` method is called often, is to maintain an incremental count, and store it in the ring buffer. The idea is pretty simple:

- Insertions — `count++` (except if full)
- Deletions — `count--` (except if empty)
- Count — just return the `count` variable.

Hence, the computations during insertion and deletion are only a single integer increment or decrement, and the `count()` function becomes a simple getter of an integer data member. In addition, the availability of a "count" variable actually allows some optimizations to some of the other methods:

- `empty()` — test `count==0`
- `full()` — test `count==capacity`

These are much faster than the earlier versions using head and tail index arithmetic. Hence, these efficiency gains may override the extra costs from incrementally computing the count during object insertions and removals.

Avoiding Three Integers

If we use an incremental count optimization for the number of items in the ring buffer, we end up with three integer values:

- Head
- Tail
- Count

It turns out that we don't need all three, because they are inter-related numbers.

We can calculate the "tail" variable from the "head" and the "count" value.

```
tail = (head + count) %sz;
```

There are actually some other numbers that are also related, which we could also use. For example, the total number of insertions and deletions of objects is related to the head and tail values, and the count is simply the difference between them.

Alternative Variable Pairs. It turns out that a ring buffer can be defined by any two variables from a set of several related calculations. Some of the possible pairs include:

- Head and tail
- Head and count
- Tail and count

Note that there are two main implementations of the initialization of head and tail values. These yield implementations that differ by one in all calculations, so you have to consistently choose between them:

- `head = tail = 0`
- `head = 1, tail = 0`

The meanings of head and tail differ slightly in these two variants. Hence, the inter-relationship with the count is also different by one. Care must be taken to avoid off-by-one errors!

Combining Two Variables. The optimization ideas above reduced our three variables (head, tail, and count) down to two variables. Any pair of them will do, since they are inter-related.

But what about reducing it to one variable? Having only one integer variable in our ring buffer might be desirable because:

- Efficient single arithmetic operations.
- One integer value as an atomic for lock-free versions.

Can it be done?

The key point to note is that we really do need two distinct values. However, we can put them together into a single integer with encoding and packing ideas.

For example, we could store the head as 16 bits and the count as 16 bits, and put both in a 32-bit unsigned integer. Note that this limits the capacity of the ring buffer to 2^16 which is 65,536. We could also pack them into a 64-bit `unsigned long` where we needed more capacity.

Modulo Arithmetic Optimizations

The `%` operator for modulo arithmetic (or remainders) is one of the slowest operations in C++. The typical code we want to optimize in a ring buffer or fixed-size queue uses this idiom:

```
head = (head + 1) % N;
```

Modulo arithmetic is based on division, which is also slow, even on integers. Hence, our ring buffer can be improved by getting rid of the percent!

How? There are several options:

- Bitwise arithmetic
- Type casts
- Ternary operator
- Branchless coding
- Unsigned arithmetic

Bitwise-and trick. Firstly, if we choose the buffer size `N`, to be a power-of-two, then we can use bitwise arithmetic. A remainder of a power-of-two is the bitwise-and of the number one less.

These are equivalent:

```
head = (head + 1) % 16;    // Modulo
head = (head + 1) & 15;    // Bitwise-and
```

Validating power-of-two. One thing you might want is a safety net to ensure nobody uses the ring buffer for a size that's not a power-of-two. We want this:

```
static_assert(is_power_of_two(N)); // How?
```

We can use the Kernighan bit trick:

```
static_assert( (N & (N-1)) == 0);  // Kernighan
```

How does this work?

It's just magic, and let's forget about it. No, actually, the Kernighan trick is that "`N&(N-1)`" clears the value of the rightmost bit of a number. Hence, if the number without the rightmost bit equals zero, then there's only one bit set in the number. And the set of numbers with only one bit set: powers of two.

Note that lots of parentheses are necessary around the bitwise operator to avoid an operator precedence glitch. Also note that the Kernigan trick fails with a false positive if `N` is zero or negative, so we should add some more safety checks at compile-time:

```
static_assert(N > 0);
```

Type casts. The use of bitwise-and is limited to powers of two, which is annoying, but there's an even more specific way to do this for some of them: type casts. If we can choose the size as 256 (8-bits) or 65,536 (16=bits), we can do this:

```
head = (unsigned char)(head + 1);   // 8-bits
head = (unsigned short)(head + 1);  // 16-bits
```

Note that type casts are often effectively free after C++ does its optimization thing. The register allocation algorithm can just choose to use a value in a different way, and propagate that forward to other arithmetic. Thus, a type cast operation may result in zero runtime instructions.

Ternary operator. But why are we using arithmetic in general, when there's actually only one case where we want to reset the value. Another way is to use the ternary operator instead of arithmetic. The calculation becomes:

```
head = (head + 1 == N) ? 0 : head + 1;
```

We can also implement this logic in two instructions, which is worth a try:

```
head++;
if (head == N) head = 0;
```

Or if you like short-circuiting operators, you can do this:

```
(++head) == N && (head = 0);
```

The compiler probably treats that the same, but you never know, and you might want to check the assembly output (e.g., using "`gcc -S`").

Branchless coding tricks. Another trick is to notice that we just want to zero the value in one specific case. Hence, we can use the branchless coding trick of using logical operators as 0 or 1 integers. The goal of branchless coding is to remove all control flow branches, so that the CPU's branch prediction logic can run fast. Note that the ternary operator is actually like an `if` statement, and it has two branches. The branchless version with only fixed arithmetic is:

```
head = (head + 1) * (head + 1 != N);  // Branchless
```

The way this works is to multiple the value by 0 or 1, depending on the logical test. Again, we can also try this as two statements:

```
head++;
head *= (head != N);  // Branchless
```

Note that I doubt the branchless versions are very efficient, because they've added a multiplication operation. The ternary operator version is likely better, and isn't that bad despite its branches, if you look at the assembly. Most compilers will convert it to a single CMOV (conditional move) CPU instruction, which makes it effectively branchless, too.

Unsigned arithmetic. One final trick is to note that we have modulo arithmetic for free in the CPU: unsigned integer arithmetic. Overflow of unsigned integers is not an exception in C++ and when you think about it, implements the exact semantics of modulo arithmetic. Hence, here's the idea:

```
unsigned char head;
...
head++;
```

It works! And there's not a single percent operator anywhere! All this time and we had cheap modulo arithmetic hiding in plain sight.

We really need to time this, because it isn't 100% guaranteed to be faster. A lot of the uses of `head` will involve converting it from `unsigned char` to an integer offset, such as array indexing in the vector of objects that makes up the ring buffer.

A variation of this idea would be to store the head and tail as integers or unsigned integers, so that they can be used as the fastest type of normal integer, but still use unsigned arithmetic overflow tricks for modulo arithmetic. This is the idea for an N=256 size ring buffer:

```
int head;
....
((unsigned char*)&head)++;
```

This relies on the platform being "little endian" with the lowest-order byte stored on the left, which is true in most modern CPUs (but not if you're sending integers over the network in "network byte order"). And, yes, you got me, I really should use `reinterpret_cast` here rather than the old C-style type cast.

Obviously, these tricks of using `head` and `tail` as unsigned integers only work for a limited set of sizes:

- N=256 — `unsigned char` (8-bits)
- N=65,536 — `unsigned short` (16-bits)
- N=4.7 billion — `unsigned int` (32-bits)

We can even do decrement and negative calculations this way, since underflow is also not an exception, whereas the `%` operator and negatives don't talk to each other at parties.

Move Semantics

If our ring buffer contains complex objects, there are many more considerations for making it efficient. One of the biggest inefficiencies in a ring buffer class is inserting and deleting any non-trivial objects. If we do it wrong, we're calling copy assignment operators and copy constructors to make new objects in the array, and running the destructor when we release an object.

Move semantics to the rescue!

The first point to note is that it doesn't matter for simple data types in our ring buffer. Any scalar values like integers or floating-point numbers don't have any copy constructors or destructors to worry about. In fact, this is also true of simple structures and classes, so long as they are "plain-old data" or POD data types.

But anything more complicated than this will have costly calls to copy constructors and copy assignment operators.

To optimize this, we need to talk about:

- Move constructor and move assignment operator
- R-value references
- Copy elision
- Return Value Optimization (RVO)

But these are covered in the separate chapter on move semantics, where there is an explanation of the theory of move semantics.

In practice, the problems arise in both our "push" and "top" versions. The "pop" routine causes a copy assignment operator invocation:

```
bool push(const T& x) {
    // ....
    arr[tail] = x;  // Copy assignment
    return true;  // success
}
```

And the "top" member has the problem of returning an object type, which will use a copy constructor call at the `return` statement.

```
T top() {
    // ...
    return arr[head]; // Copy constructor
}
```

The automatic compiler optimization of "copy elision" might help improve the performance of the "top" method. Returning an object is exactly the situation it's meant for. However, we can use move semantics explicitly to ensure it's improved:

```
bool pop_top_move(T& outobj) {
    if (is_empty()) { return false;    }
    ct_incremental--;
    int oldhead = head;
    head = (head + 1) % sz;
    outobj = std::move(arr[oldhead]); // Move asst
    return true;  // success
}
```

Note that `std::move()` is a compile-time type-cast here, without any runtime cost. And it's required to convert to an R-value reference, as otherwise the assignment statement would still call a copy assignment operator.

Constructor Problems

One of the performance problems with our ring buffer implementation is that `std::array` calls the constructor for every object whenever a new ring buffer object is defined or created. This occurs with this use of std::array for our ring buffer:

```
std::array<T, sz> arr;  // Fixed-size array
```

How to avoid these constructor calls? After all, our ring buffer is supposedly empty with zero objects initially. Some of the solutions that don't work and will still call constructors:

- Raw arrays
- Pointer to std::array

Using a raw array like this will still call all the constructors when our ring buffer is created:

```
T arr[sz];
```

Similarly, we could use an allocated copy of `std::array`, since it's really an object not an array. It works like this:

```
std::array<typename T,sz> * arrptr;
....
arrptr = new std::array<T,sz>;  // in constructor
```

This allocates our big array in the constructor rather than as a non-allocated data member. This adds an extra inefficiency from the extra allocated block, and doesn't work anyway. The `new` operator will still run all the individual object constructors.

What about using `std::vector` instead?

Standard Vector Problems

Using `std::vector` can be better than `std::array`, because it delays both its memory allocation and its construction of objects,

```
std::vector arr<T>;
```

Unfortunately, I'm not a big fan of this approach, because it has other difficulties:

- Extra memory allocation call (inefficient).
- Bounds checking failures in debug libraries.

The first point is that `resize()` has the same problem with too many constructor calls. Doing this in the constructor will still call all the constructors:

```
arr.resize(sz);  // Constructors!
```

So, maybe we can call the `reserve()` function instead of `resize()`. That won't call constructors:

```
std::vector arr<T>;
// ....
arr.reserve(sz);  // No constructors!
```

This has hopefully allocated the memory for all the objects, without running their constructors. But this can run into various problems when we try to use the vector elements. The problem is on this type of statement in our `push` method:

```
arr[tail] = x;
```

And the same problem still occurs with our code that gets items out of the ring buffer. Note that the issue is not move semantics, because this has the same issue:

```
outobj = std::move(arr[oldhead]); // Move assignment
```

The issue is bounds checking on the `[]` operator for `std::vector`. In theory, the reserve() function has allocated valid memory for enough objects. However, the `size()` function is still zero, so the runtime bounds checking will trigger on any debug run of the code.

Yes, maybe some platforms this will work, with no bounds checking. But you can run into portability problems. For example, it makes the code fail with spurious runtime errors on any type of "hardened" standard C++ library.

Explicit Destructor Calls

Another problem with our ring buffer implementation when instantiated with class types is destructor calls. Instead of too many constructor calls, we have too few destructor calls. The problems include:

- Destructor calls missed after move assignments (e.g., popping).
- Destructor calls on destroying the whole ring buffer.

One solution: don't bother. If the object that's used in a ring buffer doesn't have important destructor actions after a move (and it shouldn't), or if destroying the whole ring buffer is in the shutdown sequence of the application, then you can maybe just forget about this problem.

Another solution is to explicitly call the destructor ourselves. You can call the destructor of a class like any other member function using the `~T()` syntax. For example, in the `pop` function, we can do:

```
arr[head].~T();  // Explicit destructor
```

Basic types don't need destructor calls, so we ideally want to distinguish trivial types from fancy class objects. We can also use type traits to do this, which are wonderfully efficient compile-time operators during instantiation of the template.

```
if (!std::is_trivially_destructible<T>::value) {
    arr[head].~T();  // Explicit destructor
}
```

The alternative is to note that trivial types have no-op destructors, and the compiler would remove them anyway. Hence, the above type trait test may be unnecessary, but it's a fast compile-time test anyway, so either way is fine.

Note that we are assuming here that the class being used has a destructor that works properly after an object has been moved away. In other words, it doesn't do something silly like assuming a pointer in the object is non-null. The move assignment operator also needs to properly clear all the non-trivial data members, such as pointers, to zero or null values, so that the destructor doesn't access bad memory after a move.

Class Interface Bypass

There are a couple ways to bypass the class interfaces, and thereby avoid the inefficiencies of construction and destruction. This makes the caller of our ring buffer manage when the objects are created and destroyed. The main ways are:

- Blocking non-trivial types
- Raw character buffer arrays
- Pointers to objects

Trivial types only. We can make our ring buffer, or other home-grown containers, faster simply by disallowing their use with complex objects. We can efficiently trigger compiler warnings with the type trails, so that users of the template know to only use scalars or other POD types. Here's some examples using the various different settings:

```
static_assert(std::is_pod<T>::value); // Plain-Old Data
static_assert(std::is_trivial<T>::value); // Trivial type
```

Raw character-array memory buffers. The idea is to use a character array as a raw buffer, rather than `std::array` or `std::vector`, for our container class (e.g., our ring buffer). To bypass class constructions by using raw memory buffers, we have choices like:

```
char arr[sizeof(T) * sz];  // Static data member
char *arr = new char[sizeof(T)*sz]; // Dynamic alloc
```

This raw byte idea is workable, but every use of the array has to involve index calculations and type casts to object-type pointers. It's fiddly and annoying, but it's faster, because it avoids constructor calls, and doesn't need all the extra messing around to avoid `std::vector` bounds checking. There are also concerns with:

- Uninitialized bytes in the buffer
- Alignment of addresses

We really should also initialize the bytes in our array buffer to all nulls in the constructor using `memset` on the whole array. To do this, we also need to make sure that all the classes using the ring buffer have properties like:

- All-bytes-null is a stable but invalid initial status of the object.
- Destructor doesn't fail on an all-bytes-null object.

We also need to manually take care of alignment of the addresses, since the compiler thinks we only have characters, which don't have alignment issues. There's the `alignas` standard specifier and various non-standard implementations for older language versions.

If we're really careful, maybe the initialization is not needed and we can leave out the `memset` call in the constructor. There's some new "uninitialized memory" primitives coming in C++26 that may also help to do so. You can maybe avoid needing the null byte initialization, but I'm betting against you when I run `valgrind` on your code.

Pointers. As much as I admire the design of move semantics, there is a simpler way to avoid the overhead of objects moving in and out of our ring buffer. Old-school coding still works: store pointers to the objects in the ring buffer instead of full objects. The upside is avoidance of object copying and moving overhead.

The downside of pointers is the extra level of indirection, and double hit to memory with poor cache locality because of that. And pointers have a few pitfalls with a bad reputation as being unsafe, but I'm sure you've heard that before.

Extensions

1. Implement a reverse ring buffer that uses decremented indices for head and tail, rather than addition, so that it grows from right-to-left instead of left-to-write.
2. Implement a dequeue in a ring buffer by adding "insert-at-head" and "remove-from-tail" operations for the ring buffer (rather than the normal insert-at-tail and remove-from-head idiom). The trick is we'll need to subtract one from indices and go in reverse.
3. Implement a ring buffer with initialization of "head=1" and "tail=0" (rather than "head=tail=0"). All calculations will differ by one, such as the "empty" calculations is not "head==tail" anymore.
4. Implement a ring buffer using two full-size integers that count the number of insertions and deletions. Note: the relationship between head and tail versus insertions and deletions is not that difficult!

22. AI Data Structures

AI Engine Overview

The main data structures used in AI engines are vectors, matrices, and tensors. Examples of how these are used:

- Vectors (1-D arrays): The input sequence of words is converted to a vector of tokens. Each token is processed to create an embedding vector.
- Matrices (2-D arrays): The weights and activations are stored in matrices. Applying a set of weights to an embedding vector (which is a vector with probabilities) is a matrix multiplication of the weight matrix over the vector, creating a new vector that has updated probabilities (with amazing intelligence added).
- Tensors (3-D arrays): Every "slice" of a 3-D tensor is a 2-D matrix. Because there are so many 2-dimensional matrix multiplications happening in AI engines, it can be efficient to generalize this procedure into 3-dimensional tensors. It is very mind-bending to try to understand what's happening, but at its core, it's basically just doing a lot of 2-D matrix multiplications, where the 3-D structure of tensors allows for some fancy parallelizations.

Okay, we're done. There's more on vectors and matrices in Chapters 15 and 18, and that's all you need to know about data structures for AI. You can stop reading this chapter.

I'm only half kidding, because AI inference and training does a whole lot of vector and matrix operations (using tensors), and not a whole lot of anything else. In fact, I'm struggling to think of where in an AI engine there's even one hash table. Ah, yes, there's probably a small hash table in the tokenizer that maps 50,000 words to token numbers, but there doesn't need to be, because you could implement your tokenizer as an automaton, and that's more of an algorithm than a data structure.

So, I'm going to say it out loud:

You don't need classic data structures in AI.

I think it's fair to say that a plain vanilla Transformer needs a lot of fancy coding of algorithms, but doesn't need all those obscure data structures you learned in Computer Science 101. You need maybe one hash table in the tokenizer, but then its vectors, vectors, vectors (e.g., embeddings, dot product, probabilities) and matrices, matrices, matrices (e.g., FFNs, attention heads, GEMM/MatMul kernels), some weird statistical math functions (e.g., activation functions, normalization, Softmax), and some AI-specific algorithms (e.g., decoding algorithms, parallelization, vectorization, tiling).

I'm not seeing any binary trees.

Where the data structures come out to play is when you try to *optimize* any of that tensor stuff to go faster. Then the roster of data structures looks like:

- Lookup tables. Precomputed arrays are used to optimize activation functions, Softmax, and other mathematical methods. If you work in AI research for long enough, you'll call it a LUT, and it's your go-to data structure for speedups (and not in the Edsger Dijkstra sense).
- Permutation arrays. Used to sort data without losing track of the indices (e.g., for mappings between word tokens and their probabilities) and also important for sparse matrices.
- Bit vectors. Can be a fast way to do masks, or to mark some items as pruned.
- Locality-sensitive hashing (LSH). This is "vector hashing." Can be useful for optimizing weights and tracking previously seen inputs.
- KV Caching. This is a widely used optimization that needs a specific hand-coded data structure.
- Inference caching. This overall cache of user input strings can potentially be done using many data structures. Probably not a binary tree, though.
- Bloom filters. These are a probabilistic combination of hashing and bit vectors. I've only seen these in research papers, although they look fast to me, and deserve more consideration.

The reason that classic data structures are missing from AI engines seems simple: parallelization. It's much easier to do parallel arithmetic on the contiguous memory blocks that underly vectors, matrices, and tensors. Similarly, lookup tables, permutation arrays, bit vectors, and vector hashing also have good vectorization characteristics.

Bit Vectors

Bit vectors are conceptually an array of N bits with 0 or 1 values. The term "bit set" is almost synonymous, but has a slightly different meaning. A bit vector maps a number at the index position to its binary bit value, whereas a bit set specifies whether a number is in a set of numbers. Both interpretations are valid, depending mostly on the application, and the underlying implementation of the data structure is almost identical.

In AI applications, a bit vector may represent a set of weights with 0 or 1 values, such as with binary quantization or XNOR neural networks. The operation with vector dot product on two bit vectors can be performed arithmetically using bitwise arithmetic.

Sparsity optimizations are another application of bit vectors. Pruning can often create "sparse" weight matrices, with lots of zeros and very few non-zero weights. A bit vector can then efficiently represent whether a weight in a vector has a non-zero value, which is then used to avoid doing any computations on zero values. An alternative to bit vectors for sparsity is to use permutation arrays of indices, as discussed further below.

Another application of bit vectors occurs in Bloom filter data structures, which are a probabilistic hybrid of hash tables and bit vectors. In this usage, a bit set represents whether an input number is found in the set of already-mapped numbers.

In practice, bit vectors or bit sets are often implemented as arrays of unsigned integers, with the bits packed into each integer. If the underlying unsigned type is 32-bits or 64-bits, then many bitwise operations on bit vectors can be performed 32 or 64 bits at a time, achieving significant parallelism without using any form of hardware acceleration beyond basic CPU instructions. Use of AVX SIMD instructions can then further vectorize many operations without a GPU. But it absolutely flies if you use a GPU with bit vectors or bit sets, because that's two levels of parallelization.

There are several pre-built C++ bit set classes that can be considered:

- `std::bitset<N>` (in `<bitset>`)
- `std::vector<bool>`
- `boost::dynamic_bitset<>`

If the maximum size of the bit vector is known at compile-time, which is often the case with AI models, then `std::bitset` is a good choice.

If not, then `std::vector<bool>` or `boost::dynamic_bitset<>` are good choices for dynamic-sized bit vectors. Alternatively, you can build your own bit vectors, if there is a particular need to hand-code them or if you just want some fun.

Permutation Arrays

Most of the vectors in AI engines are not just random lists of numbers. Rather, they are (conceptually) an array of the probabilities of output words, where the position in the vector indicates which word. So, if we have our `logits` array, then `logits[0]` is the probability of "`the`" whereas `logits[1]` is the probability for "`cat`", and so on, up to about 50,000, which is a common vocabulary size for LLMs.

Problems arise if we want to sort our probabilities in the logit array, and we need this for our decoding top-k algorithm. We can't just sort the vector of probability numbers, because we'll lose track of which probability maps to which token number.

Permutation arrays to the rescue! A permutation array is an array that is the same size as some other array, but maps to the *indices* of the other array. A permutation array for our vocabulary has 50,000 integers, each of which is the index into other arrays.

The downside of permutation arrays is that they introduce inefficiency in both space and time. Space usage is increased by having two vectors. The time cost to access a vector element increases, too. Rather than just looking up the probability for the nth word in the logits array (i.e., "`prob=logits[n]`"), we have a two-step procedure:

1. Look up the index in the nth element of the permutation array (i.e., "`i=permut[n]`"),

2. Use that index to look up the probabilities in the main logits array (i.e., `prob=logits[i]`").

So, it's bigger and slower. *Some rescue.*

However, permutations can be valuable if it allows us to do much less arithmetic overall, which is the case with "sparse" arrays where most elements are zero. This is why permutation arrays are used for LLM sparsity optimizations, but not in normal practice.

Sorting with a Permutation Array: The way to sort another array, indirectly via a permutation array, is shown in detail for the top-k decoding algorithm in Chapter 18. The basic idea is:

1. Set up the identity permutation.

2. Sort using an indirect procedure: (a) compare elements in the main array indirectly accessed via the permutation array, (b) swap the indices in the permutation array (not changing the main array).

So, the original array doesn't actually get sorted with only the permutation array changing. If we want to print out the main array in a sorted list, we have to do so via the permutation array. The original main array is still unsorted if we access it directly.

Sparsity with Permutation Arrays. Sparsity is an optimization where most of the weights have been "pruned" to zero, and only a small percentage remain non-zero. This saves a lot of storage space for the model, and can also run much faster. The basic vector dot product kernel only needs to calculate with non-zero weights, so we want a way to avoid processing all of the many zero weights. Again, permutation arrays are the solution!

Sparse vectors (or matrices or tensors) can be stored as parallel arrays of:

- Non-zero weights only
- Permuted integer index of that non-zero weight in the original vector

These two arrays are much shorter than the original vectors if there is high sparsity. If sparsity is 90%, then 10% of numbers are non-zero, and the permutation approach uses two arrays, so it is 20% of the original size. The cost of doing a sparse dot product has reduced from the full length of the original vectors, down to the average sparsity factor (i.e., how many non-zero values). In other words, the number of multiplication computations goes down to 10% FLOPs, although there's the extra permutation calculation, so it's might seem like it's 20%, but we can often hardware-accelerate the permutation array step in CPU or GPU architectures. Hence, sparse vector dot products are fast. Calculation of the vector dot product for AI inference need only multiply using the much smaller number of non-zero weights.

Can we vectorize permuted arrays for hardware acceleration? Short answer: yes.

Permutations can be vectorized with hardware acceleration in both CPU and GPU versions. The C++ AVX "gather" (load) and "scatter" (store) intrinsics work for x86 CPUs. Different GPU primitives are available for permuted arrays.

Sparsity doesn't really work without permutations. A raw full-size vector containing lots of zeros doesn't vectorize well, because it still sends all of those zeros for processing. A permuted index of sparse values works much better because it only considers non-zero values.

Vector Hashing

Vector hashing is needed in various parts of an AI engine as a speedup. There are various AI research papers on using hashing for various computations involving vectors and tensors of higher dimensions. Implementations of such algorithms are available in open source and commercial "vector database" products that you can use. Some of the applications for LLMs include inference caching, embeddings, and RAG architectures.

But how do you hash a full-length vector? Or a matrix? It's a complicated theoretical area. One of the main techniques is Locality-Sensitive Hashing (LSH), which is hashing to find vectors that are "close" in *n*-dimensional space.

One of the interesting research areas for vector hashing is total precomputation of vector dot products. Think about precomputation of vector dot products in AI inference. If you could hash the two vectors, then you could replace the main bottleneck in AI inference with two hash lookups. Is there a way to efficiently convert a vector dot product operation on two vectors into a hash lookup, thereby avoiding all those multiplications? What about speedup of matrix multiplication by hashing?

Remember that you can pre-compute anything about the weights before inference, because they don't change during inference. Hence, one of the vectors could potentially be pre-hashed offline. Maybe you could even use some type of "perfect hashing" for those vector hashes, if you've got a big enough compute budget. But you can't pre-hash both of the vectors or pre-compute the dot product, because the other vectors are dynamically calculated along the way, dependent on user inputs. This is being examined by advanced researchers, and is still a work in progress.

Perfect Hashing

Perfect hashing aims to achieve collision-free O(1) hashing at runtime, by investing a lot of offline compute budget to find an optimal hash function for a set of static data. There are many possible hash functions, and some are better than others. Perfect hashing tries to find an optimal hash function within the search space for possible methods. Mostly, it's by trial-and-error. Searching for a perfect hash function typically uses a brute-force and computationally expensive method for simply trying multiple hash functions and testing them for collisions.

Perfect hashing only works in the situation where all of the possible keys are known in advance (i.e., static data). Interestingly, this is exactly the situation with AI model vocabularies!

Hence, the idea of perfect hashing can be used to improve the performance of a hash table in the tokenizer. The general concept is that different hash tables are tested with various different meta-parameters (e.g., the hash table size, and multipliers in the hashing function). So, you can test various different hash functions against the 50,000 known tokens in the vocabulary, until you find a "perfect" one where there are no clashes. Amusingly, this longstanding algorithmic method sounds exactly like doing Neural Architecture Search (NAS) to find the best AI model hyper-parameters.

Bloom Filters

Bloom filters are a probabilistic data structure based on a combination of hashing and bit vectors. Multiple hash functions are computed for each key, and this is used to set bitflags, as described in more detail below. Bloom filters are mentioned in various research papers on AI, but are not yet used much in industrial AI applications. Perhaps they should be, as they seem very efficient.

Like hashing, Bloom filters have been used as a data structure to speed up neural network inference. However, much of the research literature about Bloom filters is about a different topic: Weightless Neural Networks (WNNs). WNNs have a different type of neuron based on binary bits, rather than matrix multiplications. These bitflag neurons can be approximated using Bloom filters. As such, that part of the research is less relevant to optimization of Transformer inference, and has not been examined in detail below.

How do Bloom Filters work? Given a key, multiple hash functions are calculated for that key, and a binary flag is set in a bitflag table for each of those hash offsets. In this way, an input key maps to a pattern of multiple bits.

The Bloom filter lookup for a key value works as follows: To test whether a key is found, the multiple hash functions are computed, and then the bitflag table is analyzed to see if all those bits are set. If any of the bits are missing, the key is *not* in the Bloom filter. If all of the bits are found, the key is *probably* in the Bloom filter, but it may also be that other keys have coincidentally set all those bits (a "false positive"), so it is not 100% guaranteed to be present.

If a probabilistic speedup is good enough, then a Bloom filter is all you need. For a 100% accurate table lookup, adding a second different type of backup data structure needs to be queried to confirm. Hence, the Bloom filter is a fast test to see if a key is not in a set, but a slow test if the key is found. This makes it an example of a "common case first" optimization, where fast computations may skip more involved computations.

The computational complexity of Bloom filters is constant, but not as fast as hashing. A hash filter uses only a single hash function, so it has O(1) lookup. However, a Bloom filter uses multiple functions, k, so it has O(k) lookup complexity.

Appendix 1. Source Code

Tester Object Instrumentation Class

This code is for "object instrumentation" that can be useful for performance analysis, and also for debugging and unit testing.

Here's a test usage to see what constructors and move operations are performed by `push_back` in the `std::vector class`:

```
Tester::reset_counters();
std::vector<Tester> vectest4;
for (int i = 1; i <= 100; i++)
    vectest4.push_back(i);
Tester::print_report();
```

Here's the full code:

```
class Tester {
private:  // Static data members
    static bool traceall_;
    static int count_default_constructor;
    static int count_copy_constructor;
    static int count_move_constructor;
    static int count_copy_assignment;
    static int count_move_assignment;
    static int count_destructor;
    static int count_int_constructor;

private:  // Object data members
    int ival_;
    bool trace_;

public:
    Tester() {
        ival_ = 0;
        count_default_constructor++;
        trace_ = false;
        if (traceall_) {
            cout << "Tester: default constructor: "
```

```
                << ival_ << endl;
        }
    }

    Tester(int val) {
        count_int_constructor++;
        ival_ = val;
        trace_ = false;
        if (traceall_) {
            cout << "Tester: int constructor: "
                 << ival_ << endl;
        }
    }

    Tester(const Tester &other) // Copy constructor
    {
        ival_ = other.ival_;
        trace_ = other.trace_;
        count_copy_constructor++;
        if (trace_ || traceall_) {
            cout << "Tester: copy constructor: "
                 << ival_ << endl;
        }
    }

    Tester(Tester&& other) noexcept  // Move cons
    {
        ival_ = other.ival_;
        trace_ = other.trace_;
        other.ival_ = -1;  // Invalidate moved data
        count_move_constructor++;
        if (trace_ || traceall_) {
            cout << "Tester: move constructor: "
                 << ival_ << endl;
        }
    }

    Tester&operator=(const Tester& other) // Copy
    {
        count_copy_assignment++;
        if (this != &other) {  // Avoid aliasing
            ival_ = other.ival_;
            if (trace_ || traceall_) {
                cout << "Tester: copy assignment: "
                     << ival_ << endl;
            }
        }
        else {
```

```
        if (trace_ || traceall_) {
            cout << "Tester: copy asst aliasing: "
                 << ival_ << endl;
        }
    }
    return *this;
}

Tester& operator=(Tester&& other) noexcept // Move
{
    count_move_assignment++;
    if (this != &other) {  // Avoid aliasing
        ival_ = other.ival_;
        if (trace_ || traceall_) {
            cout << "Tester: move assignment: "
                 << ival_ << endl;
        }
    }
    else {
        if (trace_ || traceall_) {
            cout << "Tester: move asst aliasing: "
                 << ival_ << endl;
        }
    }
    other.ival_ = -1;  // Invalidate moved data
    return *this;
}

~Tester()
{
    count_destructor++;
    if (trace_ || traceall_) {
        cout << "Tester: destructor: "
             << ival_ << endl;
    }
    ival_ = -1;  // Safety
}

// Equality operators
bool operator==(const Tester& other) {
     return ival_ == other.ival_;
}


// Setters for object members
void trace(bool bval) { trace_ = bval; }

// Setters for static data members
```

```
    static void traceall(bool bval) {
                traceall_ = bval; }
    static void reset_counters() {
        count_default_constructor = 0;
        count_copy_constructor = 0;
        count_move_constructor = 0;
        count_copy_assignment = 0;
        count_move_assignment = 0;
        count_destructor = 0;
        count_int_constructor = 0;
    }
    static void print_report() {
        cout << "Tester Count Report" << endl;
        cout << "- Default constructor: "
             << count_default_constructor << endl;
        cout << "- Int constructor: "
             << count_int_constructor << endl;
        cout << "- Copy constructor: "
             << count_copy_constructor << endl;
        cout << "- Move constructor: "
             << count_move_constructor << endl;
        cout << "- Copy assignment: "
             << count_copy_assignment << endl;
        cout << "- Move assignment: "
             << count_move_assignment << endl;
        cout << "- Destructor: "
             << count_destructor << endl;
    }

    static void selftest() {
        // Constructors should equal destructors
        // ... but move constrs don't increase count
        int errors = 0;
        int total_constructors =
               count_default_constructor
             + count_int_constructor
             + count_copy_constructor;
        if (total_constructors != count_destructor) {
            if (total_constructors>count_destructor) {
                cout << "Tester selftest: cons ("
                     << total_constructors
                     << ") more than destructors ("
                     << count_destructor
                     << ")" << endl;
                errors++;
            }
            else {
                cout << "Tester selftest: dest ("
```

```
                        << count_destructor
                        << ") more than constructors ("
                        << total_constructors << ")"
                        << endl;
                    errors++;
                }
            }

            if (errors == 0) {
                cout << "Tester selftest: no errors found"
                     << endl;
            }
        }
};

// Define Tester static data members
bool Tester::traceall_ = false;

int Tester::count_default_constructor = 0;
int Tester::count_copy_constructor = 0;
int Tester::count_move_constructor = 0;
int Tester::count_copy_assignment = 0;
int Tester::count_move_assignment = 0;
int Tester::count_destructor = 0;
int Tester::count_int_constructor = 0;
```

Intercepted new and delete

This source code is the global scope intercept functions for the memory allocation `new` and `delete` operators. The library tracks basic statistics about calls and bytes allocated.

```
// Global counters
unsigned long int s_new_count = 0;
unsigned long int s_newarr_count = 0;
unsigned long int s_delete_count = 0;
unsigned long int s_deletearr_count = 0;
unsigned long int s_new_bytes = 0;
unsigned long int s_newarr_bytes = 0;
```

```
void memory_reset_counters()
{
    s_new_count = 0;
    s_newarr_count = 0;
    s_delete_count = 0;
    s_deletearr_count = 0;
    s_new_bytes = 0;
    s_newarr_bytes = 0;
}

void memory_report()
{
  cout << "MEMORY CALLS REPORT" << endl;
  cout << "- new calls: " << s_new_count << endl;
  cout << "- new[] calls: " << s_newarr_count << endl;
  cout << "- delete calls: " << s_delete_count<< endl;
  cout << "- del[] calls: "<< s_deletearr_count<<endl;
  cout << "MEMORY SIZE REPORT" << endl;
  cout << "- new bytes: " << s_new_bytes << endl;
  cout << "- new[] bytes: " << s_newarr_bytes << endl;
}

void* operator new(size_t n)
{
    s_new_count++;
    s_new_bytes += n;
    return malloc(n);
}

void* operator new[](size_t n)
{
    s_newarr_count++;
    s_newarr_bytes += n;
    return malloc(n);
}

void operator delete(void* v)
{
    s_delete_count++;
    free(v);
}

void operator delete[](void* v)
{
    s_deletearr_count++;
    free(v);
}
```