C++ AVX Optimization

CPU SIMD Vectorization

David Spuler

Aussie AI Labs

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

https://www.aussieai.com

First published: July 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 three advanced C++ books (low latency, data structures, and safety), two generative AI LLM books, two CUDA C++ books, four non-fiction textbooks on C++ programming covering introductory and advanced C++ programming, efficiency/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.

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++, and Java, and JavaScript. He holds a Bachelor of Science with First Class Honors in Computer Science from James Cook University.

Preface

Why a Book on C++ AVX Instructions?

AVX is free parallel coding on the CPU! It's right there waiting for you to use it now, if only you know the right function names. AVX SIMD parallelism can also be used in conjunction with multithreading to get a double layer of parallel power.

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 or other Aussie AI software. Contact by email: `support@aussieai.com`.

Other Books by the Author

If you want fast code, here are a number of other books on efficient C++ coding:

- Efficient Modern C++ Data Structures: Container and Algorithm Optimizations
- C++ Low Latency: Multithreading and Hotpath Optimizations
- Safe C++: Fixing Memory Safety Issues

And some more with a particular focus on AI and fast LLM backends in C++:

- Generative AI Applications: Planning, Design, and Implementation
- Generative AI in C++: Coding Transformers and LLMs

And if you're a fan of going super-parallel with GPU chips:

- CUDA C++ Optimization: Programming Faster GPU Kernels
- CUDA C++ Debugging: Safer GPU Kernels

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

Table of Contents

Part I: AVX Optimizations

1. AVX Intrinsics

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-1 — 128-bit registers = 16 bytes
- AVX-2 — 256-bit registers = 32 bytes
- AVX-512 — 512-bit registers = 64 bytes
- AVX-10 — also 512-bit registers (with speedups)

In terms of numerical processing, you get this level of parallelism:

- AVX-1 — 4 x 32-bit `float` or `int` values
- AVX-2 — 8 x 32-bit values
- AVX-512 — 16 x 32-bit values

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 from arrays to AVX registers — `_mm_loadu_ps`, `_mm256_loadu_ps`
- Store registers 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
- ...and many more

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 the usual 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

Combining Multithreading and SIMD CPU Instructions

You can double up! C++ multithreading software can be interleaved with CPU SIMD instructions as an optimized optimization. It's totally allowed, and you can even put it on your resume.

The idea is basically this structure:

- Multithreading architecture — higher-level CPU parallelization.
- SIMD instructions — lower-level CPU vectorization.

You can even triple up your parallelism:

- Multithreading/multicore (CPU)
- SIMD instructions (CPU)
- GPU vectorization

Each different type of parallelization comes in at a different level. There's even a fourth level, because CUDA C++ GPU programming has its own SIMD instructions to run on the GPU, based on the `float4` family of types.

However, they're not AVX, and don't work on an x86 CPU, so we'll leave the GPU SIMD discussion to another day.

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
7. Microsoft, 2021, *__cpuid, __cpuidex*, https://learn.microsoft.com/en-us/cpp/intrinsics/cpuid-cpuidex?view=msvc-170 (Using CPUID to detect versions.)
8. Wikipedia, July 2025 (accessed), *Advanced Vector Extensions*, https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

2. Simple AVX 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 with 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])
{
    // Multiply 4x32-bit 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); // AVX SIMD Mult
    _mm_storeu_ps(vresult, dst);  // Convert to floats
}
```

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-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 AVX2 registers
    __m256 r1 = _mm256_loadu_ps(v1);   // Load floats
    __m256 r2 = _mm256_loadu_ps(v2);
    __m256 dst = _mm256_mul_ps(r1, r2);  // Multiply (SIMD)
    _mm256_storeu_ps(vresult, dst);  // Convert to 8 floats
}
```

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 back to float uses AVX-2 intrinsic "_mm256_storeu_ps".

AVX-512 SIMD Multiplication

Here is the basic 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 (SIMD)
    _mm512_storeu_ps(vresult, dst);  // Convert 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. Hence, it's important to check your platform before optimizing!

3. CPU Platform Detection

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!

Preprocessor Macro Tests

Firstly, you cannot generally use the preprocessor to decide what version of AVX you have (if any). This only works if:

1. There's only one platform, and

2. You're compiling on (or for) the same platform that will run the binary.

In other words, it's either you and your one box doing everything, or else you're carefully maintaining lots of different executable binaries for each platform.

Note that you can modify the default CPU platform target via compiler mode settings. During compilation, you can either take whatever platform you're on, or you can modify the setting with compiler flags for different compile-time platform effects:

- `-mavx` — GCC/Clang compiler
- `-march=native` — GCC/Clang compiler
- `/arch:AVX` — MSVC compiler
- `/arch:AVX2` — MSVC compiler

In those limited circumstances, you can use the builtin preprocessor macros:

- `__AVX__`
- `__AVX2__`
- `__AVX512F__`

There are also the SSE versions of these macros:

- `__MMX__`
- `__SSE__`
- `__SSE2__`
- `__SSE3__`
- `__SSE4_1__`
- `__SSE4_2__`

There are also some macros for specific types of CPU functionality or individual machine codes:

- `__FMA__` — fused multiply-add.
- `__BMI__` — bit manipulation instructions.
- `__POPCNT__` — popcount (set bits count instruction).

If you're also supporting non-AVX platforms, your AVX code probably should have a check like this somewhere :

```
#if defined(_M_ARM) || defined(_M_ARM64)
          || defined(_M_HYBRID_X86_ARM64)
          || defined(_M_ARM64EC) || __arm__ || __aarch64__
#error AVX not supported on ARM platform
#endif
```

Source: GGML AI inference backend open-source code (see Appendix for license details).

Runtime CPU Feature Checking

In general, for shipping a binary to customers, 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).

Preprocessing checks can help with the non-AVX platforms, but not so much on x86 CPUs. You cannot choose between AVX, AVX-2, and 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. Go ahead, prove me wrong!

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.)

CPUID Instruction

Given the preprocessor limitations, it is important to check your CPU platform has the AVX support that you need. 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, usually during initialization.

Fortunately, every CPU has a builtin machine-code instruction called "CPUID" that is very fast and provides this information. The main features of CPUID include:

1. It's a hardware opcode! (fast), and

2. The bit flags are very obscure, and therefore

3. Using it directly is a real pain.

The main way to do this is via one of several possible "`cpuid`" intrinsic functions at program startup. There are several versions of this non-standard C++ intrinsic:

- `cpuid` — the main CPU instruction.
- `__cpuid()` — basic CPU information (MSVC)
- `__cpuidex()` — extended information (MSVC)
- `__get_cpuid()` — GCC/Clang version in `<cpuid.h>`
- `__cpuid_count()` — also GCC/Clang, but more specific.

GCC also has a more user-friendly version without any bit flags needed:

- `__builtin_cpu_supports("NAME")` — look up CPU features by name (e.g., "SSE").
- `int _may_i_use_cpu_feature (unsigned __int64 a)` — an old version.

The GCC version is current and quite easy to use. The other one looks like a bad AI hallucination, but it's in some 2022 Intel documentation, so best of luck with that.

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!

References

1. Microsoft, 2021, *__cpuid, __cpuidex*, https://github.com/MicrosoftDocs/cpp-docs/blob/main/docs/intrinsics/cpuid-cpuidex.md
2. Microsoft, April 2025, *DirectXMath: DirectXMath is an all inline SIMD C++ linear algebra library for use in games and graphics apps*, https://github.com/microsoft/DirectXMath/blob/main/Extensions/DirectXMathAVX.h
3. Agner Fog, 2023, *version2: Vector Class Library, latest version*, https://github.com/vectorclass/version2/blob/master/instrset_detect.cpp, https://github.com/vectorclass/version2/blob/master/instrset.h
4. Wikipedia, July 2025 (accessed), *Advanced Vector Extensions*, https://en.wikipedia.org/wiki/Advanced_Vector_Extensions
5. Stack Overflow, 2013, *Intrinsics for CPUID like informations?*, https://stackoverflow.com/questions/17758409/intrinsics-for-cpuid-like-informations
6. Gnu, July 2025 (accessed), *x86 Built-in Functions*, https://gcc.gnu.org/onlinedocs/gcc/x86-Built-in-Functions.html
7. Intel, 2022, *Intel® C++ Compiler Classic Developer Guide and Reference*, https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/overview.html, PDF: https://cdrdv2.intel.com/v1/dl/getContent/767250?fileName=cpp-compiler_developer-guide-reference_2021.8-767249-767250.pdf
8. GGML, July 2025 (accessed),
9. llama.cpp: LLM inference in C/C++, https://github.com/ggml-org/llama.cpp/blob/master/ggml/src/ggml-cpu/arch/x86/cpu-feats.cpp (Testing the CPU platform.)

4. Common Bugs & Slugs

Common AVX Bugs

Nobody said that AVX was easy! There are certainly plenty of great speedups, but there are also some new ways to crash your code:

- AVX version not supported by CPU architecture (crash!).
- Bugs in tricky AVX loop bounds and incrementers (various mistakes possible).
- Alignment problems (usually 16-bit alignment is needed).
- CPU overheating (AVX instructions are heavy on the poor silicon).
- Pointer arithmetic errors (AVX types are bigger than normal).
- Wrongly mixing integers and floating-point numbers (they're the same size, after all).
- Bytewise comparison pitfalls (e.g., `memcmp`, `vpcmpeqb`, `vpmovmskb`, and `bzhi`; beware padding bytes, negative-zero, Inf/NaN floating-point values, and more).

In addition to AVX-specific bugs, there are all sorts of normal variable bugs! The AVX register variables can simply be uninitialized, or you can divide by zero, or any number of memory mistakes.

To catch some of these problems, you can still use the same debugging techniques on AVX variables. Runtime checkers will catch AVX-related memory errors, so make sure to use Valgrind or ASan.

Common AVX Slugs

If you do AVX correctly, your program goes much faster! But you can also accidentally slow it down, and here's some of the ways:

- Slow memory accesses — poor cache locality of your memory lookups will slow things, no matter what AVX instructions you use (e.g., prefer contiguous data storage like arrays or vectors).
- Alignment slugs — incorrect alignment is sometimes auto-corrected, but then it's slower, even when it doesn't crash, such as if you do unaligned stores.
- Overuse of alignment-safe AVX primitives — this is always slower, so avoid it where unnecessary.
- Downclocking of AVX instructions — use of AVX undermines any overclocking you might be doing!
- Setting AVX constants inside the loop — tune your inner loops even in AVX (a common mistake).
- Accidental redundant AVX code — e.g., wrong logic in loop indices.
- Gather instructions are often slower — poor memory access patterns.
- Auto-vectorization prevention — compilers sometimes don't speak AVX very well (check the assembly output).
- Pointers not declared as "restricted" — throw your poor compiler a bone.
- Too much prefetching — `_mm_prefetch()` can be a slowdown.
- Lookup tables can be a de-optimization — benchmark against raw computation.
- Caching can be a slug — for the same reasons, benchmark caching against recomputation.

Loop Invariant Code Hoisting

AVX statements can be misplaced like any other statements. Can you spot the slug in this code:

```
void aussie_vec_mult_scalar_AVX1_sluggy(
       float v[], int n, float c)
{
    for (int i = 0; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]); // Load floats
        __m128 rscalar = _mm_set1_ps(c); // Vector scalars
        __m128 dst = _mm_mul_ps(r1, rscalar); // Mult scalar
        _mm_store_ps(&v[i], dst);  // convert to floats
    }
}
```

The fixed code has a constant operation hoisted out of the loop. It doesn't change throughout the iterations:

```
void aussie_vector_mult_scalar_AVX1(float v[],int n,float c)
{
    const __m128 rscalar = _mm_set1_ps(c);  // Hoisted!!
    for (int i = 0; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]);
        __m128 dst = _mm_mul_ps(r1, rscalar); // Mul scalars
        _mm_store_ps(&v[i], dst);
    }
}
```

Accidental Redundant Computations

This is buggy and sluggy code. Can you see the bug? It's hidden by "code blindness" because of what C++ programmers are used to seeing.

```
void aussie_vector_multiply_scalar_AVX2(
        float v[], int n, float c)
{
    const __m256 rscalar = _mm256_set1_ps(c); // Vec scalars
    for (int i = 0; i < n; i++) {
        __m256 r1 = _mm256_loadu_ps(&v[i]);   // Load floats
        __m256 dst = _mm256_mul_ps(r1, rscalar); // Multiply
        _mm256_store_ps(&v[i], dst);  // convert to floats
    }
}
```

The bug is "`i++`" because it should really be "`i+=8`" to stride through the loop. This is the type of bug that can happen in any of the SIMD kernels.

Depending on the function, it can be a bug, or it can be an insidious slug, whereby the same computations are done over again, losing all benefit of the AVX vectorized instructions.

Too Much AVX

What do you think of this AVX routine to clear a vector in parallel? Here's the unoptimized code:

```
void aussie_vector_clear_AVX2(float v[], int n)
{
    const __m256 rzeros = _mm256_setzero_ps();
    for (int i = 0; i < n; i += 8) {
      __m256 r1 = _mm256_loadu_ps(&v[i]);   // Load floats
      _mm256_store_ps(&v[i], rzeros);  // store zeros
    }
}
```

Umm, yeah. Do you think I like AVX maybe a little too much? How about:

```
std::memset(v, 0, n *sizeof(float));
```

Don't worry. The compiler designers are certainly using something better than looping AVX calls in the standard library implementation.

List of AVX Optimization Tricks

A lot of these ideas are covered in other parts of the book. However, here's a convenient list of some of the major techniques:

- Unroll loops manually (reduce loop overhead and have fewer branches).
- Use "double unrolling" of loops (unroll once to AVX, then unroll those AVX instructions, too!)
- Parallel accumulators (with single or double unrolled loops).
- Avoid data dependencies for "out-of-order" execution (parallel accumulators; split integer versus floating-point arithmetic, etc.)
- Fused Multiply-Add (FMA) is fast (and a pleasure to use).
- Use "`alignas`" to maintain alignment.
- Use "broadcast" of constants (e.g., `_mm256_set1_ps()`).
- Manual prefetching (e.g., `_mm_prefetch`).
- Masked operations (useful branchless coding trick).
- Optimizer architecture flags such as "`-march`" for GCC/Clang.
- Compare `memcpy` vs. `vpcmpeqb`, `vpmovmskb`, `bzhi` (use with care!).
- Use permute and shuffle primitives to reorder data.
- Store data with "streaming stores" via `_mm256_stream_ps()`.
- Use vector sizes that are a multiple of loop unroll (or pad with zeros).

Special issues with some AVX instructions:

- `setr_ps` (gather) is slow
- `bzhi`
- `tzcnt`
- `_mm256_blendv_ps` can help branchless programming (but blends can also be slow)

General low-level coding optimization tricks that also apply to AVX programming:

- Cache locality
- Cache lines
- Avoid false sharing (multithreaded code)
- Prefetching
- Cache warming
- Branchless coding
- Reduce data sizes
- Pack data together
- Prefer contiguous data
- Prefer Structure-of-Arrays (SoA) over Array-of-Structures (AoS)
- Use transpose tricks in matrix multiplication (contiguous data)
- Avoid or reduce memory allocations (e.g., preallocation, memory pools)

Tools and commands to use:

- Check the compiler's assembly output (e.g., "`gcc -S`")
- Use optimizer settings such as "`-O`" and "`-march`" flags
- Check for memory errors with Valgrind (Memcheck) and ASan
- Profile low-level performance with "perf" or Intel VTune
- Linux kernel optimizations (e.g., "`noatime`" in `/etc/fstab`)

Useful third-party libraries to consider for their AVX SIMD methods:

- xsimd (header-only library)
- VCL (Vector Class by Agner Fog)
- Eigen (linear algebra)
- Highway (high-performance SIMD)
- SIMDe (portable SIMD operations)

But only look at the libraries if you don't want the fun of coding AVX yourself!

5. One-Dimensional Vectorization

What is Vectorization?

Vectorization is the name given to transforming a software loop from running sequentially on a one-dimensional 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 intrinsics cannot fully vectorize a larger vector of very 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.

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 aussie_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 aussie_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 aussie_vector_multiply_scalar_AVX1(
          float v[], int n, float c)
{
    const __m128 rscalar = _mm_set1_ps(c); // Vector scalars
    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);  // convert to floats
    }
}
```

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 aussie_vector_multiply_scalar_AVX2(
            float v[], int n, float c)
{
    const __m256 rscalar = _mm256_set1_ps(c); // Vec 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);  // convert to floats
    }
}
```

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); // vec scalars
    for (; n > 0; n -= 8, v += 8) {
        __m256 r1 = _mm256_loadu_ps(v); // Load 256-bits
        __m256 dst = _mm256_mul_ps(r1, rscalar); // Multiply
        _mm256_store_ps(v, dst);  // convert (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 + pointer arith: 340 ticks (0.34 s)
```

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 aussie_vector_add_scalar_AVX1(
          float v[], int n, float c)
{
    // Add scalar constant to all vector elements
    const __m128 rscalar = _mm_set1_ps(c); // vector scalars
    for (int i = 0; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]);   // Load 128-bits
        __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); // vec scalars
    for (int i = 0; i < n; i += 8) {
        __m256 r1 = _mm256_loadu_ps(&v[i]); // Load 256-bits
        __m256 dst = _mm256_add_ps(r1, rscalar); // Add scal
        _mm256_store_ps(&v[i], dst);  // convert (Aligned)
    }
}
```

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); // vector zeros
    for (int i = 0; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]);   // Load 128-bits
        __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); // vec zeros
    for (int i = 0; i < n; i += 8) {
        __m256 r1 = _mm256_loadu_ps(&v[i]); // Load 256-bits
        __m256 dst = _mm256_max_ps(r1, rzeros); // MAX(R1,0)
        _mm256_store_ps(&v[i], dst); // store back 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 aussie_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 128-bits
        __m128 dst = _mm_exp_ps(r1); // Exponentiate (expf)
        _mm_store_ps(&v[i], dst);  // convert (Aligned)
    }
}
```

AVX2 SIMD exponentiation of 8 values: The AVX2 intrinsic is "`_mm256_exp_ps`" to exponentiate 8 elements in parallel using the 256-bit registers. The code with loop unrolling every 8 values and AVX-2 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 256-bits
        __m256 dst = _mm256_exp_ps(r1); // Exponent (expf)
        _mm256_store_ps(&v[i], dst);  // convert (Aligned)
    }
}
```

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-exp operation 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)
```

6. Horizontal Reductions

What is a Reduction?

A reduction is an operation that "reduces" a vector down to a single number. It's called "horizontal" because it operates across the vector, requiring multiple elements from the same vector to be examined. An elementwise operation where each vector element is handled separately, without any other elements from the same vector, is called a "vertical" operation. Here's a rule of thumb:

- Vertical — two vector operands (e.g., add two vectors).
- Horizontal — one vector operand (e.g., sum elements of a single vector).

Some examples of common horizontal reductions on vectors include:

- Max
- Min
- Sum
- Average

These look easy enough. But, no!

Horizontal reductions are actually harder to code than vertical operations, even when there's two operands for the vertical algorithm. SIMD hardware instructions are much better suited to vertical elementwise operations across two different vectors.

The horizontal SIMD instructions are often slower!

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 aussie_vector_sum(float v[], int n)  // Summation
{
    float sum = 0.0;
    for (int i = 0; i < n; i++) {
        sum += v[i];
    }
    return sum;
}
```

Horizontal AVX Intrinsics

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.

Parallel Accumulators Trick

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). The overall algorithm becomes:

1. Initialize 4 (or 8) accumulators.

2. Scan the whole vector doing 4 parallel additions every iteration.

3. Sum the 4 accumulators (and done).

In this way, we can defer the horizontal addition ("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 aussie_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(); // Set accums zero
    for (int i = 0; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]); // Load 128-bits
        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)
{
    // Summation (horizontal) of a single vector
    if (n % 8 != 0) { // Safety check (no extra cases)
        aussie_assert(n % 8 == 0);
        return 0.0; // fail
    }

    __m256 sumdst = _mm256_setzero_ps(); // Set 8 accum zero
    for (int i = 0; i < n; i += 8) {
        __m256 r1 = _mm256_loadu_ps(&v[i]); // Ld 8x256-bit
        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(); // Set 8 accum zero
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:

```
__m256 sumdst = _mm256_loadu_ps(&v[0]); // first 8 values
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
    if (n % 4 != 0) {
        aussie_assert(n % 4 == 0);
        return 0.0; // fail
    }
    __m128 sumdst = _mm_loadu_ps(&v[0]);   // Initial values
    for (int i = 4 /*not 0*/; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]); // Load 128-bits
        sumdst = _mm_max_ps(r1, sumdst); // dst=MAX(dst,r1)
    }
    // 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
    if (n % 8 != 0) { // Safety check (no extra cases)
        aussie_assert(n % 8 == 0);
        return 0.0; // fail
    }
    __m256 sumdst = _mm256_loadu_ps(&v[0]); // Init 8 values
    for (int i = 8/*not 0*/; i < n; i += 8) {
        __m256 r1 = _mm256_loadu_ps(&v[i]); // Load 256-bits
        sumdst = _mm256_max_ps(r1, sumdst); // d=MAX(d,r1)
    }

    // 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
    if (n % 4 != 0) {
        aussie_assert(n % 4 == 0);
        return 0.0; // fail
    }
    __m128 sumdst = _mm_loadu_ps(&v[0]);   // Initial values
    for (int i = 4 /*not 0*/; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]); // Load 128-bits
        sumdst = _mm_min_ps(r1, sumdst); // d = MIN(d, r1)
    }
    // 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
    if (n % 8 != 0) { // Safety check (no extra cases)
        aussie_assert(n % 8 == 0);
        return 0.0; // fail
    }
    __m256 sumdst = _mm256_loadu_ps(&v[0]); // Init 8 values
    for (int i = 8/*not 0*/; i < n; i += 8) {
        __m256 r1 = _mm256_loadu_ps(&v[i]); // Load 256-bits
        sumdst = _mm256_min_ps(r1, sumdst); // d = MIN(d,r1)
    }

    // 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 would 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 a 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
    if (n % 4 != 0) { // Safety check (no extra cases)
        aussie_assert(n % 4 == 0);
        return 0.0; // fail
    }
    __m128 sumdst = _mm_setzero_ps();   // Zero accumulators
    for (int i = 0; i < n; i += 4) {
        __m128 r1 = _mm_loadu_ps(&v[i]);  // Load floats
        __m128 sqr = _mm_mul_ps(r1, r1);  // Square (V*V)
        sumdst = _mm_add_ps(sqr, sumdst); // SUM = 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
    if (n % 8 != 0) { // Safety check (no extra cases)
        aussie_assert(n % 8 == 0);
        return 0.0; // fail
    }

    __m256 sumdst = _mm256_setzero_ps(); // Zero accums
    for (int i = 0; i < n; i += 8) {
        __m256 r1 = _mm256_loadu_ps(&v[i]);  // Load floats
        __m256 sqr = _mm256_mul_ps(r1, r1);  // Square (V*V)
        sumdst = _mm256_add_ps(sqr, sumdst); // SUM=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 of pointer arithmetic and loop unrolling could be applied.

Hybrid Vertical-Horizontal Operations

There are also many more complicated reductions. There are also some one-dimensional vector operations that a "hybrid" and don't fit neatly into the categories of horizontal reductions or vertical elementwise operations.

Some examples include:

- Vector dot product
- AI normalization (BatchNorm)
- AI statistical normalization (Softmax)

For example, vector dot product is like a vertical elementwise multiplication followed by a horizontal summation reduction.

Normalization functions in AI, such as BatchNorm or Softmax, are another example, where the whole vector is processed to "normalize" every value, but doing so requires computations both horizontally to compute the scaling factor and vertically to scale every element.

Boolean Vector Operations

Weirdly, a lot of Boolean testing operations on vectors are also hybrid vertical-horizontal operations. Some examples include:

- Vector equality (two vectors)
- Vector is zero (one vector)
- Vector contains a value (one vector)
- Vector has a NaN or Inf (floating-point)

All of these seem like they're elementwise tests, where each vector element can be examined separately.

The two-vector comparison does pairwise comparison of each element separately, and the single-vector operations can examine one element at a time. That would seem to make them all meet the criteria for vertical elementwise operations, with very fast implementations via AVX SIMD instructions.

Alas, no, it's not that simple.

A more accurate way to think about them is that the first phase creates a Boolean vector (as a temporary), where each element of this interim vector has the test result for each element.

Hence, there's actually a second phase where the many Boolean results from analyzing each vector element need to be "reduced" to a single Boolean value as the final answer.

This last phase is like a horizontal reduction over a Boolean vector using a logical "or" or "and" operator.

7. Vector Dot Product

Vector Dot Product

An example of an operation with aspects of both vertical and horizontal arithmetic is: vector dot product. The two arithmetic requirements are:

- Multiplication — pairwise on all elements (vertical vectorization).
- Accumulation — summing up all those values (horizontal reduction).

Conceptually, this is the case, with a vertical multiplication of pairs, followed by a horizontal summation of these values. In practice, however, these two stages are merged in together for a faster method.

Sequential Code

Here is the basic non-vectorized dot product computation without any optimization attempts.

```
float aussie_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;
}
```

There are several ways that we could improve this code, such as:

- Loop unrolling
- Pointer arithmetic
- Instruction-level parallelism

However, first, let's look at making it work fast on the CPU using AVX SIMD instructions.

AVX 128-Bit Dot Product

The AVX instruction set has a vector dot product intrinsic that wraps an x86 dot product hardware instruction on the CPU. 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 prod
    float fret = _mm_cvtss_f32(dst);  // Extract float
    return fret;
}
```

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!

Loop Unrolled Version

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 aussie_vecdot_unroll4_basic(
      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 aussie_vecdot_unroll_AVX1(
    float v1[], float v2[], int n)
{
    // AVX-1 loop-unrolled (4 floats) 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; i += 4) {
        // AVX1: Vector dot product of 2 vectors
        //  ... process 4x32-bit floats in 128 bits
        __m128 r1 = _mm_loadu_ps(&v1[i]); // Load 128-bits
        __m128 r2 = _mm_loadu_ps(&v2[i]);
        __m128 dst = _mm_dp_ps(r1, r2, 0xf1); // Dot product
        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: Vecdot basic: 2805 ticks (2.81 seconds)
Time: Vecdot AVX1 (4 float 128-bit): 1142 ticks (1.14 s)
```

Fused-Multiply-Add (FMA)

The AVX-2 FMA intrinsic takes 3 vectors, each of size 256-bits, multiplies the 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.

Anyway, here's the code:

```
float aussie_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(); // Set accums zero
    for (int i = 0; i < n; i += 8) {
        // AVX2: process 8x32-bit floats in 256 bits
        __m256 r1 = _mm256_loadu_ps(&v1[i]); // Load 256-bit
        __m256 r2 = _mm256_loadu_ps(&v2[i]);
        // FMA of 3 vectors
        sumdst = _mm256_fmadd_ps(r1, r2, sumdst);
    }

    // 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 floats, 128-bits): 1169 ticks (1.17 s)
Vecdot AVX1 FMA (4 floats, 128-bits): 1314 ticks (1.31 s)
Vecdot AVX2 FMA (8 floats, 256-bits): 783 ticks (0.78 s)
```

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).

8. 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 — process data in a fast order for CPU caches (sequential).
- Parallelization —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 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 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 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). And for the sake of argument, let's pretend we don't know a better way to initialize 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 fully familiar with the unintentional introduction of 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 aussie_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 crosses over several `case` statements. Also, we see that 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 can't tweak the "`i++`" code 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 parts in 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, if 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 aussie_vecdot_rev(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 of 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 reduce the total number of 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 special 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 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 a 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 aussie_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 in 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 the basic loop sentinel version that sets up a dummy success in `v[n]`:

```
bool aussie_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.

And we also have to remember to check the last vector element that we temporarily overwrote wasn't also a real success.

```
bool aussie_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 when there are two nested loops, but the inner loop is difficult to parallelize because of a dependency on the outer loop variable. 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

9. Softmax

What is Softmax?

Softmax is a relatively simple component of the Transformer architecture for LLM backends, use in both training and inference. Softmax is one of the simpler AI kernels, since there are no tensors or matrix multiplications. It's a good candidate for SIMD optimization, but it's a hybrid of vertical and horizontal operations.

All it does is operate on a vector of numbers and change the numbers. It is a type of "normalization" (like BatchNorm or LayerNorm in the prior chapter) but Softmax is used for many different reasons in a Transformer.

The purpose of Softmax is to take a set of values in a vector of calculated values, and normalize them into probabilities in a new output vector. After Softmax, the output vector contains a new normalized set of values which all add up to 1, and they are intended to represent probabilities of the likelihood of each token/word associated with each vector element.

The Softmax algorithm is basically:

- Exponentiate each vector element — elementwise vertical operation.
- Add up the exponentials — horizontal reduction operation.
- Divide every vector element by this sum — vertical again.

In fewer words, scale by the sum of the exponentials.

So, why do we need all the exponentials? The idea is that the input vector contains "logits," which are logarithms of probabilities, so we are exponentiating each one to bring it out of log-domain into real-domain. Then with the division step, we are normalizing them all so that they are probabilities that total exactly 1.

Inputs, Outputs and Dimensions

To understand Softmax, let's examine its inputs and outputs in more detail. Overall, Softmax is a "vector-to-vector" algorithm. The input and output vectors have the same dimension, which is the model size.

Softmax is not an "element-wise" vector operation. The change to each element in the vector depends on all the elements in the vector, not only on the current element. For example, it adds up all the elements to use as a scaling factor (after exponentiating them).

The input to Softmax is a vector of floating-point numbers containing *logits*. These are coming out of the model's calculation layers, and are a rough representation of word probabilities.

However, the input vector is a bit messy. Firstly, they are in the "log-domain" rather than real probabilities. In other words, they are the logarithm of the probabilities. Secondly, the values in the vectors are not "normalized" so there are numbers outside the ranges `0...1`, including large numbers and negatives. Thirdly, the numbers also don't nicely add up to 1 like disjoint probabilities should.

The output of Softmax is a beautiful vector that's perfect in every way, with harps playing softly in the background. The log-domain has been fixed by exponentiation. All of the numbers are scaled into `0...1`, and they all add up to 1 in total like a good probability distribution of disjoint events. The output vector from Softmax fills every Statistician's heart with joy.

Softmax and Temperature

One important use of Softmax is in the decoding step. At the end of each decoder sequence, the Softmax function is used to normalize the logits before processing by a decoding algorithm to choose an output token with the highest probability. As part of this method, the Softmax function is usually changed to a "scaled Softmax" that uses an extra parameter called the "temperature."

What is the temperature? The purpose of the temperature parameter is as a hyper-parameter that influences the level of randomness or unpredictability in the output. A higher setting for temperature means that the decoder is more likely to output the lower-probability tokens (i.e., it has a fever and says silly stuff). If the temperature is low, the decoder is mostly going to output the highest probability token, meaning it is much less random (like a cold-hearted robot).

What is the value of the temperature? The temperature is a non-zero positive floating-point number that can be between 0 and 1, or can also be greater than 1. A temperature of zero cannot be used as it would cause divide-by-zero errors. When the temperature equals 1.0, it doesn't change the Softmax function at all (i.e., continues harmlessly without scaling). Since the Softmax function is scaled by the reciprocal of the temperature, the effect is to make randomness higher with a larger temperature setting (so it runs "hotter" and gets more "bubbly"). If the temperature is below 1.0, making it a fraction, the effect is to spread out the logits more, which has the effect of reducing randomness of the output. If the temperature is greater than 1.0, this contracts the logits towards each other, making the decoder more likely to choose each of them (although still with some randomness), thereby increasing output randomness.

Softmax C++ Optimizations

The Softmax function is an inefficient computation by default because it has two scans of the vector and each scan does an expensive operation (exponentiation and division). First, the whole vector is scanned to compute the sum of the exponential of each value. Then the whole vector is re-scanned to divide each vector element by this sum-of-exponentials.

Here is a naive implementation of Softmax in C++:

```
#include <math.h>  // Declare expf()
....
float aussie_vector_sum_of_exponentials(float v[], int n)
{
    float sum = 0.0;
    for (int i = 0; i < n; i++) {
        float e = expf(v[i]);
        aussie_assert(e >= 0.0);
        sum += e;
    }
    return sum;
}

void aussie_vector_softmax_basic(float v[], int n)
{
    float denom = aussie_vector_sum_of_exponentials(v, n);
    if (denom == 0.0) {
        aussie_assert(denom != 0.0);
        return;  // fail (should not occur)
    }
    for (int i = 0; i < n; i++) {
        v[i] = expf(v[i]) / denom;
    }
}
```

Reciprocal Multiplication. One simple speed improvement is to multiply by the reciprocal instead of using floating-point division:

```
float recip = 1.0f / denom;
for (int i = 0; i < n; i++) {
    v[i] = expf(v[i]) * recip; // Scale by recip mult
}
```

Common sub-expression elimination. If we look carefully, we can note that `expf` is called twice on `v[i]` for every element, and `expf` is quite an expensive mathematical function to be messing with. A faster method is to compute `expf(v[i])` once and then leave it in the vector to use again, thereby avoiding the second `expf` call.

```
aussie_vector_expf(v, n);  // Element-wise expf...
float denom = aussie_vector_sum(v, n);  // Denominator
// ....
float recip = 1.0f / denom;
for (int i = 0; i < n; i++) {
    v[i] *= recip;  // NOTE: v[i] is already expf'd
}
```

This uses "`aussie_vector_expf`" to exponentiate each element of the vector. A naive implementation is:

```
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]);
    }
}
```

Fused Loop Softmax. The above code has two initial loops doing exponentiation and summation. There's an opportunity for "loop fusion" here by merging the two loops in "`aussie_vector_expf`" and "`aussie_vector_sum`" into one loop that exponentiates and sums as it goes. The call becomes:

```
float denom=aussie_vector_expf_and_sum(v,n); // Exp sum
```

This is how the fused version of "exponentiate-and-sum" looks in a simple C++ version without any optimizations:

```
float aussie_vector_expf_and_sum(float v[], int n)
{
    // Apply EXPF (exponential) to each element
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        v[i] = expf(v[i]);
        sum += v[i];
    }
    return sum;
}
```

More fusion? The Softmax algorithm still has two loops, but we have difficulty fusing the first part ("exponentiate and sum") with the second part ("scale by the sum"). The second code block awaits the sum to use as the scale factor (as a reciprocal). Hence, it doesn't work to "scale as we go" because then we'd have to go back and re-scale earlier vector elements anyway. I can't really see a good way that we can avoid this roadblock to fusion.

Vectorized Softmax

The Softmax code has two loops that run sequentially: summing the exponentials, and scaling by the sum's reciprocal. Both loops are candidates for vectorization. The only real problem is we can't fuse the two loops into one, because the second loop needs the result of the first loop as the scaling factor.

Second things first. The second loop is easy to vectorize because it's just multiplying a vector by a scalar. The second loop does not have any exponentiation, because the first loop has stored the exponentiated values in the vector, so there is only a scaling multiplication by the reciprocal.

```
for (int i = 0; i < n; i++) {
    v[i] *= recip;  // NOTE: v[i] is already expf'd
}
```

Vectorizing exponentials. The first loop has exponentiation and also summing of the results. That sounds like it's going to be expensive, but the "`exp`" and "`expf`" functions have had hardware support for years. The x86 processor architecture has opcodes to do various common math functions including exponentials, and these can be accessed via the AVX C++ intrinsic functions.

Vectorized Softmax with AVX

The AVX intrinsics use x86 SIMD instructions to operate on multiple `float` values or integers at once (e.g., 4 `float` values for AVX-1, 8 `float` values for AVX-2, 16 `float` values for AVX-512). Surprisingly, there are AVX SIMD exponential function intrinsics, to apply "`expf`" to multiple elements of a vector in parallel.

Example: Softmax with AVX exponential and summation. We can vectorize both these loops separately using AVX intrinsics. Vectorized versions of `expf` and summation were examined in the hardware acceleration chapter. The version for AVX1 becomes:

```
void aussie_vector_softmax_exponentiate_and_sum_AVX1(
          float v[], int n)
{
    aussie_assert(n % 4 == 0);
    aussie_vector_expf_AVX1(v, n); // AVX1-accel expf
    float denom = aussie_vector_sum_AVX1(v, n);  // AVX1 sum
    if (denom == 0.0) {
        aussie_assert(denom != 0.0);
        return;  // fail (should not occur)
    }
    float recip = 1.0f / denom;
    for (int i = 0; i < n; i++) {
        v[i] *= recip;  // NOTE: v[i] is already expf'd
    }
}
```

Actually, that's only vectorized two out of three loops. Here's the code with the third loop, multiply-by-scalar, also done with AVX, as was also shown in the vectorization chapter. This code is the AVX2 version:

```
void aussie_vector_softmax_fused_exp_sum_mult_AVX2(
       float v[], int n)
{
    // Softmax with EXP and SUM and MULT in AVX2
    aussie_assert(n % 8 == 0);
    // Element-wise expf...
    float denom = aussie_vector_fused_expf_sum_AVX2(v, n);
    if (denom == 0.0) {
        aussie_assert(denom != 0.0);
        return;  // fail (should not occur)
    }
    float recip = 1.0f / denom;
    aussie_vector_multiply_scalar_AVX2(v, n, recip);
}
```

Vectorized & Fused Loop Softmax

What about vectorization applied to these fused loops. Can we do better than using the two vectorized loops in sequence? Can we merge the exponentiation and summation into a single unrolled loop and vectorize that using AVX intrinsics? I'm just teasing you. Of course, we can!

Here is "kernel fusion" of the vector `expf` and vector summation into a fused-`expf`-summation kernel. I coded this for both AVX1 and AVX2, with both very similar in structure. Here is the code for AVX2:

```
float aussie_vector_fused_expf_sum_AVX2(float v[], int n)
{
    // Fused EXPF and SUMMATION of a single vector
    if (n % 8 != 0) { // Safety check (no extra cases)
        aussie_assert(n % 8 == 0);
        return 0.0; // fail
    }
    __m256 sumdst = _mm256_setzero_ps(); // Set accums zero
    for (int i = 0; i < n; i += 8) {
        __m256 r1 = _mm256_loadu_ps(&v[i]); // Load 256-bits
        __m256 expdst = _mm256_exp_ps(r1);  // Exponentiate
        sumdst = _mm256_add_ps(expdst, 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;
}
```

And here is the AVX2 code that uses that fused `expf`-summation routine as one loop, and has a multiply-by-scalar afterwards.

```
void aussie_vector_softmax_fused_exp_sum_mult_AVX2(
      float v[], int n)
{
    // Softmax with EXP and SUM and MULT in AVX2
    aussie_assert(n % 8 == 0);
    // Element-wise expf...
    float denom = aussie_vector_fused_expf_sum_AVX2(v, n);
    if (denom == 0.0) {
        aussie_assert(denom != 0.0);
        return;  // fail (should not occur)
    }
    float recip = 1.0f / denom;
    aussie_vector_multiply_scalar_AVX2(v, n, recip);
}
```

Softmax Benchmarking Results

Here's the result from my benchmarking 100,000 calls to the various Softmax versions for a vector with 1024 elements, for all these algorithms, including both sequential and AVX parallel versions.

```
Softmax benchmarks (N=1024, ITER=100000)
Softmax basic: 13186 ticks (13.19 seconds)
Softmax reciprocal: 12986 ticks (12.99 seconds)
Softmax expf-first: 6977 ticks (6.98 seconds)
Softmax expf-sum-fused: 6682 ticks (6.68 seconds)
Softmax expf with AVX1: 1095 ticks (1.09 seconds)
Softmax expf/sum AVX1: 910 ticks (0.91 seconds)
Softmax fused expf/sum AVX1: 1095 ticks (1.09 seconds)
Softmax fused expf/sum/mult AVX1: 831 ticks (0.83 seconds)
Softmax expf with AVX2: 538 ticks (0.54 seconds)
Softmax expf/sum AVX2: 306 ticks (0.31 seconds)
Softmax fused expf/sum AVX2: 252 ticks (0.25 seconds)
Softmax fused expf/sum/mult AVX2: 176 ticks (0.18 seconds)
```

Interestingly, fusing the `expf` and summation kernels was actually worse for AVX1, but it was faster for AVX2. Otherwise, our speedups were as we would expect, with the triple-AVX optimizations of `expf`, summation, and multiply-by-scalar (reciprocal) getting the best results by far. The triple-vectorized AVX2 version is 73 times faster than the naive C++ sequential version, using about 1.4% of its CPU time cost. And we haven't even tried AVX-512 optimization yet!

Softmax Overflow and Underflow

Note that a simplified version of Softmax has been used in the code examples in this chapter for simplicity of explanation. Not only is this computation still very slow (even if we precompute all those calls to `expf`), it's also prone to overflow and underflow. The real computation of Softmax needs to be further optimized algebraically and scaled to avoid these problems.

This scaled computation can then be optimized using many of the same methods as for the naive Softmax version, as above. Further optimizations may include the use of calls to hardware acceleration APIs, pre-computed lookup tables as approximations for the `expf` function, and converting the loops to pointer arithmetic.

Softmax Optimization Research

The Softmax function is a significant cost in Transformer inference because it is part of the attention mechanism, whereas it was less of a bottleneck in earlier neural network architectures. A vanilla Softmax implementation is very expensive because it involves computing the exponentials of all of the elements of the logits vector. Various attempts have been made to optimize and approximate Softmax calculations, including:

- Softmax code optimizations (sequential)
- Vectorized Softmax (parallelization)
- Softmax approximations
- Integer-only Softmax
- Pruned Softmax (removal)
- Fused Softmax (kernel fusion)
- Softmax replacements (use different functions)

Related Research Areas: Note that there are several other areas of theory that are relevant to Softmax optimizations and approximation. The denominator of the Softmax formula is a "sum of exponentials" and this type of calculation also appears in Logarithmic Number System (LNS) addition. Also, the sum of exponentials calculation, appears in "log-sum-exp networks," which are somewhat related to "tropical algebra." The area of "max-plus networks" may also be relevant to Softmax approximation research.

References

1. Kunal Banerjee, Vishak Prasad C, Rishi Raj Gupta, Karthik Vyas, Anushree H, Biswajit Mishra, Nov 2020, *Exploring alternatives to softmax function*, https://arxiv.org/abs/2011.11538
2. David Spuler, March 2024, Chapter 25. SoftMax, *Generative AI in C++: Coding Transformers and LLMs*, https://www.amazon.com/dp/B0CXJKCWX9
3. Tianhua Xia, Sai Qian Zhang, 22 Nov 2023, Softmax Acceleration with Adaptive Numeric Format for both Training and Inference, https://arxiv.org/abs/2311.13290 (Hardware-based Softmax accelerator.)
4. Yichuan Deng, Zhihang Li, Zhao Song, 26 Apr 2023 (v2), *Attention Scheme Inspired Softmax Regression*, https://arxiv.org/abs/2304.10411

5. Jiuxiang Gu, Chenyang Li, Yingyu Liang, Zhenmei Shi, Zhao Song, 6 May 2024, *Exploring the Frontiers of Softmax: Provable Optimization, Applications in Diffusion Model, and Beyond*, https://arxiv.org/abs/2405.03251
6. Jacob R. Stevens, Rangharajan Venkatesan, Steve Dai, Brucek Khailany, and Anand Raghunathan, 2021, *Softermax: Hardware/Software Co-Design of an Efficient Softmax for Transformers*, In DAC. https://arxiv.org/abs/2103.09301 (Attempts to optimize softmax with focus on the max operation and the use of base-2 exponentials.)
7. Ihor Vasyltsov, Wooseok Chang, 2021, *Efficient Softmax Approximation for Deep Neural Networks with Attention Mechanism*, arXiv preprint arXiv:2111.10770, 2021, https://arxiv.org/abs/2111.10770
8. Grave, Armand Joulin, Moustapha Cissé, David Grangier, Hervé Jégou, 2017, *Efficient softmax approximation for GPUs*, Proceedings of the 34th International Conference on Machine Learning, PMLR 70:1302-1310, 2017. http://proceedings.mlr.press/v70/grave17a.html
9. M Dukhan, A Ablavatski, 2020, *Two-pass softmax algorithm*, 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) https://ieeexplore.ieee.org/document/9150394, PDF: https://arxiv.org/pdf/2001.04438, Code: http://www.github.com/google/XNNPACK (A method to compute Softmax using two passes by caching some calculations.)
10. Jiachen Lu, Jinghan Yao, Junge Zhang, Xiatian Zhu, Hang Xu, Weiguo Gao, Chunjing Xu, Tao Xiang, Li Zhang, Apr 2022, *SOFT: Softmax-free Transformer with Linear Complexity*, https://arxiv.org/abs/2110.11945 Code: https://fudan-zvg.github.io/SOFT/
11. Soroush Abbasi Koohpayegani, Hamed Pirsiavash, June 2022, *SimA: Simple Softmax-free Attention for Vision Transformers*, https://arxiv.org/abs/2206.08898 Code: https://github.com/UCDvision/sima (Analyzes the cost of Softmax in Vision Transformers and proposes a method without Softmax, using l1 normalization instead.)

10. Advanced AVX Techniques

AVX Memory Alignment Issues

Some of the AVX examples gloss over the issue of managing "alignment" for 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 have ensured 16-byte alignment of all memory addresses via:

```
// Test with 16-byte alignment
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 AVX registers multiply 4x32-bit floats...
    __m128 r1 = _mm_loadu_ps(v1);   // Load floats 128-bits
    __m128 r2 = _mm_loadu_ps(v2);
    __m128 dst = _mm_mul_ps(r1, r2);   // Multiply
    _mm_store_ps(vresult, dst); // Aligned convert to floats
}
```

Ideally we'd like to ensure that the function is only called with aligned addresses at compile-time. The first attempt is to declare "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.

Permute and Shuffle

There are two classes of AVX instructions known as "permute" and "shuffle" operations. They're both very similar in that they reorder data in the AVX registers. There are various ways that this can be used to optimize the many different types of algorithms. Generally speaking, the permute options came later, and are better:

- Shuffle — AVX-1/SSE.
- Permute — AVX-2 and AVX-512.

Some example intrinsic functions:

- `_mm_shuffle_epi32` — shuffle (AVX-1)
- `vpermilps` — permute (AVX-2)

Was it a marketing name change? The permute and shuffle commands look very similar, except more bits in the later commands. I'm not 100% sure.

Blend Ternary Operations

The AVX "blend" operations are like a C++ ternary operator on steroids. Generally, they test a mask vector, and then choose from either of their two operands, depending on the value of a bit in the mask vector. You can see how it's a lot like doing:

```
z = bit ? x : y;
```

Except, you know, in parallel.

For example, there's the AVX blend functions, such as:

```
__m256 ret = _mm256_blendv_ps(x, y, mask);
```

One of the main ways to go fast with blend is to combine it with one of the many "cmp" comparison operations. This allows a vector comparison to create a mask, where each element is either 0 or 0xFF (all 1s). The main AVX comparison functions are:

```
_mm256_cmp_ps(x, y, cond)
_mm256_cmp_pd(x, y, cond)
```

The condition or "predicate" operand can be a builtin constant, such as:

- `_CMP_EQ_OQ` — equality
- `_CMP_LT_OQ` — less-than

There are many other operands with different sizes or operations. The operations include: EQ (equality), LE (less-equal), LT (less-than), NEQ (not-equal), NLT (not-less-than), NLE (not-less-equal), NGT (not-greater-than), NGE (not-greater-equal), ORD (ordered), UNORD (unordered). There's also the "nop" conditions of FALSE for always false, and TRUE for always true.

This idea of using comparisons with blend operations has a lot of similarities to the CPU non-SIMD equivalent of ternary operators, the CMOV assembly statement. The blend instructions are branchless logic, just like CMOV for a single operation.

Vectorization of Lookup Tables

The use of lookup-tables was once a powerful speed optimization, but I'm not sure they're being used much any more. Memory is slow, and CPUs are fast. Before you assume a LUT is better, you really should benchmark it against just plain old computation, or even re-computation!

Anyway, if you're using a LUT to trade space for speed, you 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 of 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 base on 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 with 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 for 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 for 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!

Part II: Low-Level Code Optimizations

11. Compile-Time Optimizations

C++ Compile-time Techniques

Compile-time processing is the optimal way to run a program. All the work is done by the compiler and none by your program. There are literally zero instructions executed on the CPU at runtime, whether it's doing training or inference. It will be blindingly fast for your users.

If only all code could be like that!

The reality is that programmers are still needed and that code still needs to run (sigh!). But to make it faster, there are lots of ways to have more computation done by the compiler, long before it ever goes near a user.

The C++ programming language has numerous features that help perform work at compile-time. These include ways to explicitly control what goes to the compiler, or to give more information to the compiler so that its optimizer can do good work on your behalf. Some of the various C++ language features to consider include:

- Conditional compilation — `#if`/`#ifdef` statements
- `inline` functions
- Templates — these expand at compile-time
- Symbolic constants — `const` or `#define`
- Function-like macros — `#define` with parameters
- Constant hints — `constexpr`, `if constexpr`, etc.
- Global and `static` variable initializations
- `static` data members — fixed data in C++ classes
- Type traits — compile-time type testing
- Restricted pointers — ignore aliasing risks

But when we're doing AI, there's another compile-time data structure to consider: the whole LLM model itself.

C++ Optimizers

Every C++ compiler has optimization built into the code generation phase. Typically, there are ways to specify that a higher degree of code optimization should be performed. Methods to control the settings include:

- Command-line arguments (e.g., "`-O1`" or "`/O1`")
- Configuration settings (e.g., Project Settings in the MSVS IDE)
- `#pragma` preprocessor directives

Take note of the meaning of the optimizer settings. For example, on MSVS the setting "`/O1`" optimizes for memory, not speed! Also, don't be like me and assume that the defaults are going to be what you want. Looking at the MSVS IDE optimizer settings in my AUSSIE project file, I found:

- "Optimization" was "disabled" by default.
- "Enable Intrinsic Functions" was "No" by default. Why not?
- "Favor Size or Speed" was "neither" by default. Come on, why is there no "both" option?
- "Inline Function Expansion" was "default" at least.

When to enable the optimizer? Should you run the optimizer at every build? At what level?

Note that your policy should *not* be to turn up the optimization to maximum level just before you ship your code to users, because your code can change in a very bad way. Don't assume that turning the optimizer mode up to super-crunch is always an easy win, as optimization can trigger latent glitches in your code by reorganizing memory or reordering instructions.

What does the optimizer do? In order to optimize code, it's important to know what sorts of optimizations your compiler is doing automatically. Compilers have been doing optimizations for literally 50 years, and the state-of-the-art is quite amazing, with an extensive body of research theory.

Some of the main automated compiler optimizations include:

- Constant folding/propagation
- Constant expression evaluation
- Common subexpression elimination
- Redundant assignment removal
- Strength reduction
- Algebraic optimizations
- Register allocation
- Loop optimizations (e.g., unrolling)
- Auto-vectorization

If you make simple changes to your code with some of the obvious things above, it's not going to give you a speedup. The compiler has already done it for you.

However, there's a limit to what compilers can do. They certainly can't make architectural changes, and there's also many mid-level algorithmic changes that cannot be automated.

Function calls inside expressions are a good example of code changes that might need to be manually optimized. When the compiler sees a function call used in arithmetic, it isn't always able to know what that function is going to do, and has to be conservative by avoiding possibly incorrect optimizations.

Floating-Point Optimizer Options

Some C++ compilers have optimizations that you can use to speed up your Floating-Point Unit (FPU). Some of the options for GCC include:

- "`-ffast-math`" option — This option is a broad enabler of multiple floating-point speedups, such as `-fno-math-errno` and `-ffinite-math-only`. It also disables negative zero.
- "`-fno-math-errno`" option — This allows the standard library math functions such as `sqrt` to run faster and also be more amenable to parallelization, simply by allowing them to never set the global "`errno`" variable. The use of `errno` was once a great way to track error codes, but it's also a blocker for thread-safety and parallelization. And let's be frank: you weren't ever checking `errno` anyway, so turn it off!
- "`-ffinite-math-only`" — This mode allows GCC math library functions to skip any checks for `Inf` or `NaN`, which can make them marginally faster.

Microsoft Visual Studio C++ also has its own set of FPU options:

- "Floating-Point Model" settings in a Project's Property Pages under "C++" for "Code Generation" has options "`/fp:precise`", "`/fp:strict`", or "`/fp:fast`"
- "Enable Floating-Point Exceptions" can be turned off if you like.

People Helping Parsers

The humble C++ compiler needs your attention. Hat in hand, the compiler is sitting there saying "I am but a poor, helpless lexer, without even a single neural network. Please help me." Hence, please consider donating your time to help a poor struggling compiler in your neighborhood.

There is a long history of the C++ compiler needing "hints" about optimization from the programmer. The early C++ language in the 1990s had a "`register`" specifier that hinted to the compiler that a variable was going to be highly used, and the compiler should optimize it by putting the variable in a CPU register. The "`register`" keyword has since been deprecated in C++17, which indicates that compiler register allocation algorithms no longer benefit from human help.

Some of the other longstanding C++ keywords that can be used for efficiency-related purposes include:

- `inline`
- `const`
- `static`

And with the evolving C++ standards, there's a whole new set of directives that are hints to the compiler about how to optimize:

- `constexpr`
- `constinit`
- `consteval`
- `reinterpret_cast`
- restricted pointers ("`restrict`")
- `[[likely]]` and `[[unlikely]]` path attributes

The `constexpr` and related directives help the compiler do "constant folding" and "constant propagation" to compute as much as possible at compile-time, thereby avoiding any runtime cost for lots of code.

In fact, the idea is extended to its logical asymptote, whereby you can declare an entire function as "`constexpr`" and then expect the poor compiler to interpret the whole mess at compile-time. Pity the overworked compiler designers.

The "`restrict`" 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 restricted pointer optimizations have been formalized in C++23, but non-standard versions have long existed. The possible benefit is that restricted pointer specifications might help the compiler do auto-vectorization of loops into parallel hardware-assisted code.

How much do these help? It's rather unclear, and the compiler is free to simply ignore these hints. Compilers already did a lot of constant propagation optimizations before the "`constexpr`" directives came along, so presumably compiler designers have upped their game even further now.

Inline Functions

Placing the keyword "`inline`" before any function declarations makes that function instantly disappear in a puff of smoke. Well, sort of. It gives your C++ compiler the hint to optimize the code by putting the function's body there instead of the function call. This is faster, but means there are many copies of the function's statements, so it increases code size.

Which functions should you inline? General wisdom is to do so for these special types of C++ functions:

- Short functions (esp. single-statement functions)
- Getters and setters in a class
- Frequently called functions at the bottom of the call hierarchy.

The `inline` specifier is just a hint. Your compiler is free to completely ignore you. In fact, this choice will probably disappear in a few years, as compilers become better than humans at choosing which functions to inline.

If you want to force the compiler to inline, use preprocessor macros. However, there's a whole minefield of problems in function-like macros. For example, you need to add parentheses around the whole expression and also around each parameter's appearance in the replacement text. Hence, `inline` functions are much safer than macros.

The value of `inline` functions is not only from avoiding function call overhead. The merging of the statements into the caller's code also allows many other optimizations to be applied there as follow-up transformations. Constants can be propagated further through the inlined statements, which is similar to `constexpr`, but the range of optimizations is much larger with `inline`.

GCC has some additional C++ language features related to inlining. There is the "`always_inline`" function attribute which says to always inline this function, and the "`flatten`" attribute which says to inline every call to other functions inside this function. There is also the "`gnu_inline`" attribute that prevents creation of a non-inlined function body.

inline function limitations

The `inline` specifier is wonderful when it works. A very important point to note about `inline` functions is that the `inline` specifier, by itself, is not enough to guarantee that inline code will be generated. The other requirement is that the compiler must know the function body code, where the function is called.

Hence, an `inline` keyword in a function prototype declaration is not enough. The executable statements inside the function's definition (i.e., the function body) must be available to the C++ compiler. Otherwise, how is the compiler to know what inline code to expand a function call into? I guess in theory the C++ compiler could maintain a huge database of all the functions in your source code, or scan through all the CPP files to find it, and that would be amazing, but we're not there yet. In practice, the compiler will only inline functions where it has seen the function body within the current C++ source file or an included header file. This requirement imposes two restrictions on the use of `inline` functions:

1. Member functions declared as `inline` should include the function body inside the same header file as the class declaration. This can be achieved by placing the function body of a member function inside the class declaration.

   For a more readable style when there are many `inline` member functions, the class declaration can declare the function prototypes, and then provide the `inline` function definitions immediately after it, in the same header file. This restriction ensures that whenever the class declaration is included as a header file, the member function body is available for inlining.

2. Non-member `inline` functions must be defined before they are used within a source file, preferably by placing the `inline` functions in a header file. Placing `inline` functions at the top of a source file allows the inlining of any function calls later in the same source file, but calls to the functions from a different source file cannot be inlined by the compiler unless the `inline` function definition is placed in a header file.

Non-inlined functions

Some functions declared as `inline` will not be expanded into inline code by the compiler, simply because they are too complicated for the compiler to handle. In this case, the `inline` specifier is ignored and the function is treated like any other function. The sophistication of the inline code generation depends on the compiler implementor.

Even if a compiler could theoretically inline a function, the compiler is sometimes still forced to generate a "real" function. There are various possible reasons for this:

1. The name of an `inline` function is used as a pointer-to-function constant.

2. A call to the `inline` function from within another source file.

3. `virtual` member functions.

When an `inline` function is called from a source file, where the function body has not been made available, the compiler generates a real function call (simply because it cannot inline the function). Hence, the real function must exist and be linked like any other function. Fortunately, the placement of `inline` functions in header files as discussed above will avoid this for any function the compiler decides to inline.

Inline Variables

Since C++17 you can define a *variable* as "`inline`". What does this do?

Basically, it's not really much of a speedup, but makes it easier to manage global constants, global variables, or `static` data members in C++ classes.

You can declare these variables as "`inline`" in a header file, with an initializer:

```
inline int g_x = 3;
```

Then you can with wild abandon include that header file all over the place without any problems whatsoever. The C++ linker is required to:

- Merge all of them into one variable at link-time.
- Guarantee that it's initialized as specified.
- Have the same address for that variable everywhere.

I find this addition to C++ somewhat humorous because it fixes up a huge mess that's existed since old K&R C code, and I've battled against it many times trying to get my program linked. I'm not going to irritate myself by repeating all the quirks, but it was always messy whether you had a global variable that was `extern` or non-`extern`, initialized or non-initialized, in a header file or a non-header file. So, if you ask me, the way that "`extern`" variable declarations "worked" was always broken, and now it's fixed in C++17. Hooray! (A bit late for me.)

Overall, allowing "`inline`" for variables is helpful to efficiency because you can be guaranteed about constants, `static` members, or global variables at compile-time. And it's always nice to get your program to link.

Constant Specifiers

The "`const`" keyword means that something is constant, and cannot be modified. It is helpful for efficiency, but its role is also to help detect programming errors, where code accidentally attempts to modify a constant variable or object. There are multiple places where "`const`" can be used.

- Symbolic constants
- `const` variables
- `const` objects
- `const` function parameters (i.e., "`const&`" idiom)
- `const` member functions (read-only)

But don't get me started on "`const` correctness." I've seen too many dawns fighting with compilers about `const`. Anyway, let's move on, and assume *we love `const`*.

Basic const symbols. Symbolic constants can be declared as a representation of a numeric value or other type data (instead of using `#define` symbols):

```
const float pi = 3.14;
```

Set-once variables with const. Variables can be made constant via "`const`", which is effectively the same as a symbolic constant, except that the initializer need not be a compile-time constant. It is a "set-only-once" variable. The C++ compiler ensures that `const` variables cannot be modified, once they are initialized.

```
const int scale_factor = get_config("scale");
const int primes[] = { 2, 3, 5, 7, 11, 13, 17 };
```

Function parameters and const. The `const` specifier can ensure that function parameters are not modified, especially for arrays passed by reference. `const` on a scalar parameter type such as `int` is not as useful, only ensuring that the code inside the function doesn't modify the parameter (which isn't really a problem anyway). However, the idiom of "`const&`" to specify a `const` reference as a function parameter allows constant pass-by-reference of object parameters, which is extremely important for C++ efficiency.

Instantiate-only objects with `const`. Class objects can be declared as `const` variables. When the variable is a `const` object, it can be instantiated via a constructor, but cannot be modified thereafter.

```
const Complex cfactor(3.14, 1.0);
```

Member functions declared `const`. Class member functions can be declared by adding the keyword "`const`" immediately after the function parameter list:

```
int MyVector::count() const;
```

The C++ compiler blocks a `const` member function from modifying data members, although it can still change "`static`" data members. For `const` object variables, the C++ compiler ensures that any calls to non-`const` member functions are disallowed.

Non-member functions. Note that a non-member function cannot be `const`. The actions of a `friend` function or other non-class function are controlled by using `const` on the parameters, rather than the whole function itself.

Beyond `const`. Newer C++ features have generalized and improved some of the uses of `const`. The "`constexpr`" specifier is much more powerful in terms of allowing compile-time optimizations, as are its derivatives "`constinit`" and "`consteval`." The newer use of "`inline`" on a variable (yes, a variable, not a function, supported since C++17), can be helpful for safely sharing constants across multiple files.

Constant Expressions Specifier

The `constexpr` keyword is an optimization hint for the compiler that's more powerful than "`const`." Whereas `const` only guarantees that something won't change, `constexpr` is a guarantee by the human that something can be evaluated at compile-time.

The compiler should use the `constexpr` hint to try to propagate constant values throughout the evaluation of expressions and function calls, producing an overall speedup. However, if the compiler doesn't have the capability to do the level of compile-time optimization required, or if the human has told the machine a bald-faced lie, there's no penalty and the code just runs like it never had a `constexpr` specifier.

There's not a whole lot of difference between `const` and `constexpr` if you use it only for named constants:

```
const float PI = 3.14f;
constexpr float PI = 3.14f;  // Same same
```

`constexpr` functions

The real power is when you use `constexpr` for functions.

```
const float SQRTPI = sqrtf(3.14f);   // Works?
constexpr float SQRTPI = sqrtf(3.14f); // Works?
```

Oh, dear! I just tested this code snippet, and the `const` version works, whereas the `constexpr` version fails to compile, which is the opposite of what I was expecting. According to an informed source that was trained on Internet scrapings, `sqrtf` is not going to be declared as a "`constexpr`" function until C++26. Alas, by then all C++ programmers will have been replaced by robots, so feel free to skip this section.

The apparently futuristic idea is that `sqrtf` should have a "`constexpr`" keyword in its declaration, because the function return value can be computed at compile-time if you pass it a constant argument. In other words, the compiler can evaluate "`sqrtf(3.14f)`" at compile-time. Hence, the whole function should be declared "`constexpr`" in the standard library header file. The `const` version is also probably not evaluating the `sqrtf` function at compile-time, but just calling it dynamically whenever the `const` variable is first initialized (this non-compile-time initialization is allowed for `const` variables, provided you don't later attempt to change its value).

Anyway, you can already declare your own function with the "`constexpr`" specifier.

```
constexpr int twice(int x)
{
    return x + x;
}
```

`constexpr` functions vs `inline` functions

A lot of the same value in terms of optimization can be had by making a function just `inline` rather than `constexpr`. Note that you can use both, but officially `constexpr` for functions implies `inline` on the function as well.

Is `constexpr` any better than just `inline`? If you pass a constant argument to a small `inline` function, then the expansion of the function body will trigger many constant propagation optimizations, effectively evaluating most of it at compile-time, which is almost the same as `constexpr`.

`constexpr` is supposed to be more formal in guaranteeing that the result of a function is a compile-time constant, and the compiler is honor-bound to do "compile-time function evaluation" to get the constant return value. Also, a `constexpr` function is more officially usable as a compile-time constant, so that you can use an expression with a `constexpr` function's return value in various places where C++ needs a constant (e.g., an array size declaration, some `template` situations, etc.).

An `inline` function is also supposed to be optimized at run-time for non-constant arguments, and `constexpr` functions are implicitly `inline` functions. The code generation requirements of dynamic inlining are often more advanced that constant expression evaluation.

Also, the limitations on how a `constexpr` function can be structured are a lot easier to code than the unrestricted nature of an `inline` function body. However, as a practical matter, the compile-time evaluation of expressions and the code generation for inlined expressions have a lot of overlap, so I expect C++ compilers will mostly try to do both on every type of function.

The `inline` keyword also serves a weird secondary purpose, by guaranteeing that there's only one copy of the function. This means we can include header files with the full definition of that `inline` function anywhere we like, without getting a compiler error at link-time about multiple definitions. But this isn't a performance optimization, and the linker feature of `inline` is almost the opposite of what we want in making a function `inline`, because we don't want a real function to be called at all.

`if constexpr` statements

There is an alternative usage of `constexpr` in terms of "`if`" statement conditions (since C++17):

```
if constexpr(cond)
```

This new syntax tags the condition as being amenable to computation at compile-time. Hence, the compiler should optimize the `if` statement to a constant value, and it can then determine at compile-time which branch should be executed. So, there is a double speedup from:

(a) the condition computation is removed at run-time, and

(b) code size reduction from unexecuted "dead code" being removed.

In fact, this determines at compile-time which code block will be *parsed*, so there are cases where you can avoid a compile-time error in templates by wrapping it inside an "`if constexpr`" check. This can be useful in compile-time situations such as `template` expansion, where you can prevent some expressions from being compiled, and also code bloat can be reduced.

`constinit` variables

The `constinit` specifier is a hybrid between `consteval` and `static` variables. The `constinit` specifier declares a variable that is `static`, with lifetime scope, that is initialized at compile-time.

A variable declared as `constinit` must be initialized, and cannot be modified (like "`const`"). However, the initializer needn't be a "constant expression" although it must be able to be calculated at compile-time.

Huh? That makes no sense. Sure, it does in the world of C++ standards. A "constant expression" with only constant arithmetic is a subset of the total overall set of expressions that can be calculated at compile-time.

The best example is a call to a function that has one path where it's constant, and another path where it's not. The definition of "`somefunc`" has two paths:

```
int somefunc()
{
    if (something) return 27;
    else return some_random_number();
}
```

The "`somefunc`" function cannot be declared "`const`" or "`constexpr`" because it isn't always a constant on all paths.

However, if we're using "`somefunc`" at program startup initialization, we can try:

```
constinit int s_myconst = somefunc();
```

Here, if we know that it will use the constant path for some reason, the initialization of "`s_myconst`" will go through the fixed path to get the compile-time constant value of 27, we can tell the compiler that by declaring the variable as `constinit`.

Anyway, now that you've been forced to learn all that, just forget about it. You'll rarely if ever be needing `constinit`.

`consteval` functions

Use `consteval` for functions that are always constant. A `consteval` function is strictly declared so that every invocation of the function *must* return a compile-time constant.

The `consteval` keyword is a subset of `constexpr` functions (and also implies `inline` on a function). Although a `constexpr` function is constant if its arguments are constant, it can also return a dynamic return value for non-constant arguments.

When would you use `consteval` versus `constexpr` functions? I mean, when you ask your boss to make you a cup of coffee, do you like to ask politely or do you issue commands? Supposedly `constexpr` is optional for the C++ compiler, whereas `consteval` is mandating compile-time evaluation.

Personally, I can't see much difference in general usage, since the compiler will probably optimize a `constexpr` function at compile-time if it's capable enough. Hence, for regular functions I don't see much benefit to `consteval` over `constexpr`. There are some complicated places in C++ where it helps to guarantee a compile-time constant, such as reflexive types and other tricks in compile-time `template` usage.

Templates

C++ templates can be used for compile-time optimizations, rather than merely as a programming convenience for algorithm generality and interface improvement. By specializing templated code for a particular type or constant parameter, the effect is that the resulting code is more specific, giving the compiler an opportunity for better optimizations.

For example, if we have vector and matrix classes, then rather than having our code dynamically check whether our precision is 32-bit `float`, or 8-bit integers, or some other low-level type, we can use templated versions of the vector and matrix classes. This generates different functions for each type of data. At the cost of some extra code space, we've given the compiler the chance to do a much better job of optimizing the code for the specific low-level data types.

Going beyond just using `template` code to write the same algorithm for different types, there are various ways to optimize code that is templated to do more at compile-time:

- Template class and function specializations
- Constant template parameters
- Compile-time conditional tests on types (e.g., `sizeof`, type traits, etc.)
- `if constexpr` syntax
- Variadic templates
- Template Metaprogramming (TMP) techniques
- SFINAE techniques

Constants can be used to instantiate `template` code in a way that helps the compiler to optimize by evaluating constant expressions. Template parameters don't need to be types, but can also be constant variables or numbers, such as the size of an array. Using a template in this way is as efficient as hard-coding the array size, which helps the compiler to know exactly what it can optimize, such as if the array size is used in any computations.

If you think you can do better than the compiler's optimizer, remember that you can also override the generic template code. For example, you can instantiate your own specific version of a template class for a particular type. Similarly, you can provide a generic function declaration that instantiates a templated function with your explicit version.

An alternative to specializing a version of a template class or function is to use compile-time tests inside the generic template code. For example, you can use conditional tests involving compile-time operations:

- `sizeof`
- `typeid`
- `std::is_same_v`
- `if constexpr` conditional test syntax

Next level templating

C++ templates are a very powerful programming mechanism. In fact, you can define entire projects as templates inside header files. To get the most out of template optimizations at compile-time, consider these methods:

- Type traits
- Variadic templates
- SFINAE
- Template Meta-Programming (TMP)

Type traits are a generic feature of C++ (since C++11) that you can use to interrogate the type of a variable. They are declared in the `<type_traits>` header file and there are numerous ways that you can test the type of a variable. The above example `std::is_same_v` is one example. As another example, there is `std::is_signed` and `std::is_unsigned` to test whether it's a signed or unsigned type. There's also `std::is_pointer` and `std::is_array` and various others.

Combining type traits with "`if constexpr`" gives a powerful way to ensure templated code gets evaluated at compile-time, and to specialize blocks of code for particular types.

Variadic templates are another way to level up your code and have been supported since C++11. These are variable-argument templates via the use of the ellipsis "`...`" operator in a `template` declaration. This allows templates to accept a variable number of parameters for instantiation.

SFINAE. Another optimization for advanced templating is to rely on SFINAE semantics. This refers to "Substitution Failure Is Not An Error" and means that `template` instantiation that fails should not itself trigger a compilation error that prevents execution. More specifically, if the compiler tries and fails to instantiate a template, but there's another way to run it, such as a different overloaded function available, then the code should execute via the non-templated method.

Relying on this capability in C++ not only avoids having compilation errors that block some advanced template usages, but can also be used to ensure compile-time calculations. However, although there are some good uses cases in making templates faster, SFINAE is an obscure programming technique that isn't widely used in everyday C++ programming.

Template Meta-Programming. Further optimization of templated code at compile-time is possible via the technique called "Template Meta-Programming" (TMP). Note that this refers to an unusual usage of templates in C++, where the idea goes beyond just using templates of code for different types (i.e., normal templating of classes). TMP is an advanced coding method that uses (misuses, perhaps) instantiation semantics of templates as a way of generating compile-time code, even for some conditional branches.

However, this is an obscure method that is rarely needed, because most of the effects can be achieved via preprocessor macros, function inlining, and using "`constexpr`" in modern C++.

References

1. 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 8 is on compile-time optimizations.)
2. Gnu.org (2023), *GCC Command Options*, GNU Compiler Collection, https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html
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. Sarah Butcher & Alex McMurray, 2 January 2025, *The C++ techniques you need for $600k hedge fund jobs*, https://www.efinancialcareers.com/news/low-latency-c

12. Zero Runtime Cost Operations

You want free CPU cycles? You got it! There are plenty of "freebies" in C++!

We've already talked about compile-time operations in C++, but here's a summary of some of the "hints" you can give to the compiler for a free gain, usually via helping the optimizer to do fancier optimizations:

- `inline`
- `template`
- `const`
- `constexpr` (also `consteval` and `constinit`)
- `noexcept`
- `static_assert`
- Restricted pointers (e.g., `__restrict`)
- `likely/unlikely` or `__builtin_expect` (expressions)
- `[[likely]]` and `[[unlikely]]` path attributes

I've missed a bunch of them, so you should re-read those chapters. Those are well-known optimizations via programmer hints.

Here are some other ones that are useful. If you see these keywords, these are free or compile-time operations:

- `auto` types (type deduction)
- `decltype`
- `final`
- `override`
- `explicit`
- `[[nodiscard]]` (function attribute)
- `= delete`

But there's always more.

Here are some advanced C++ language features that you might think cost real CPU juice, but are free for various language design reasons:

- Type traits — compile-time type operators (not RTTI).
- Concepts (C++20) — compile-time guarantees.
- Static reflection (C++26) — fixing RTTI inefficiencies.
- Profiles — safety with compile-time validation.
- Curious Recurring Template Pattern (CRTP) — useful for devirtualization.
- Structured bindings — grouped assignments are compile-time processed.

Type traits are a form of Compile-Time Type Information (CTTI) and work at compile-time. Some examples are operations like:

`std::is_trivial` or `std::is_same`

However, note that you have to be careful not to move across into the darker side of RTTI, which is `dynamic_cast` and `typeid`.

Free Type Cast Operations

There are various arithmetic operations that can look real, but actually disappear in a puff of compiler smoke. The first item on the list is type casts, which have many freebies:

- `reinterpret_cast`
- `static_cast`
- `const_cast`
- `std::move` (move semantics)
- `std::forward` (perfect forwarding)

Note that `std::move` is effectively a compile-time type cast, which turns an l-value into an r-value (I'm simplifying the idea here). However, there are also overloaded versions of `std::move` with two or more arguments that really do move bytes at runtime (effectively doing `memcpy`), so be aware of the distinction between free uses of `std::move` for move semantics versus real byte movers.

Arithmetic type casts between similarly represented numbers can often be optimized away. For example, these are usually free, or at least very fast:

- Downsizing integer type casts (e.g., `int` to `char`).
- Upsizing integer type casts (e.g., `char` to `int`)
- Floating-point type conversions (e.g., `float` to `double`)

Differently sized integer types seem like they would cost real instructions to convert between them. If a `char` is one byte and an `int` is four bytes, you'd think there's an operation that adds or removes three bytes. However, the compiler has many tricks up its sleeves here, such as:

- Copy propagation
- Register allocation
- Peephole optimizations

This is often true of the conversions between any of the many and varied integer types, from a 1-byte `char` to a 16-byte `long long`. In the cases where the compiler cannot find a way to do it freely, the operation is very inexpensive anyway.

But note that not all type casts are free. In particular, converting between integers and floating-point types is expensive, in both directions, because the way these two types of values are represented is very different. Be careful with explicit type casts, but also any expressions that mix integer and floating-point types may have implicit type casts.

Optimized Away

Here's a somewhat random list of stuff that should get optimized away by the compiler. We can be reasonably sure these are free:

- Constant expressions (via the general idea of "constant folding" and newer `constexpr` features)
- Small getter member functions (via inlining)
- Null-effect expressions (useful for compiling-out assertions)
- Unnecessary temporary variables (removed by copy propagation, peephole optimizations, and register allocation)
- Wrongly typed constants (e.g., using `1` or `1U` or `1.0` or `1.0f` should be implicitly type-converted at compile-time).
- Double negation (using "`!!(x)`" is a common trick).

- Algebraic simplifications (e.g., plus zero, subtract zero, times one, and many more).
- Explicit zero conditional tests (e.g., `if (x != 0)` or `if (ptr != nullptr)` equates to `if(x)` or `if(ptr)` at runtime).
- First data member in an object or structure (it's offset is zero, so there's a "plus zero" in the address calculation that is optimized away).
- Assertions and `#if DEBUG` (if compiled-out for production).

The compiler optimization of "dead code elimination" will make these control flow features free:

- `while(1)` — using `for(;;)` isn't faster!
- `if(true)` or `if(1)` or `if(0)` or whatever
- `do...while(0)` — a common macro trick.
- Short-circuited constants in `||` or `&&` operators
- Tested constants in the ?: ternary operator

You can always check the assembly code with "`gcc -S`" or the MSVS assembly debug window.

Standard Container Operations

A lot of the standard containers have many optimized specializations for builtin types. Hence, if you're using `std::vector<int>`, you can expect operations like `push_back` are inlined and very fast. All of the contiguous containers have a simple structure, and the non-contiguous linked containers would maintain incremental variables, making `begin()` and `end()` calls very fast. Similarly, most of the containers maintain an incrementer counter of objects inside, so all calls to `std::size` are as fast as a getter accessing an integer data member (inlined, of course).

There are some relatively simple standard C++ data types where operations can often be inlined or optimized away by the compiler:

- `std::pair`
- `std::tuple`
- `std::optional`
- `std::expected`
- `std::variant` (modern C++ unions)

Finally, note that some calls to containers can lead to memory allocations, which is a slowdown. And various containers when used on your own non-scalar objects can trigger many calls to constructors or assignment operators, which is slow regardless of whether it calls copy or move versions.

I mean, moving is better than copying, but an optimizer can only do so much.

The Opposite of Free

There are also features of C++ that look like they should be free, but are actually costly. Perhaps we should call them "costlies"?

Elegance and the beauty of short code sequences is not the same thing as fast. Here are some examples of beautiful things that can be slow:

- Calls to `virtual` functions
- RTTI (i.e., `dynamic_cast` and `typeid`)
- Lambdas, functors and other function objects
- `std::function`
- Comparators (except maybe standard ones like `std::less`)
- Fold expressions
- Exception handling

The issue with lambdas and function objects is not clear-cut. If you use a lambda with a simple capture and an immediate assignment to a functor variable, which is then called, the optimizer probably can handle this and inline the function call. However, if you declare your own complex lambda as a comparator that is sent to a function (e.g., to `std::sort`), all of the calls to that lambda are probably not inlined, leading to a performance bottleneck.

Also, if you use a builtin comparator like `std::greater` and pass it to `std::sort` or other library functions, it's likely that the operation has a pre-coded template specialization for that comparator, meaning it won't really be using it as a function call.

However, you might want to benchmark this or look at the standard library source to confirm there is such a specialization!

And here are some more slugs that are less obvious, because the code is concise and looks like it should be fast:

- Operator overloading (looks like a single instruction, but it's a function call, even if it's inlined).
- Initializer lists (can call lots of copy constructors).
- Pointer-to-function types (cannot be inlined).
- Implicit type conversions (especially via overloaded type cast operators).
- Temporary object creation (accidental)
- Type casts between `int` and `float` (explicit or implicit)
- Container `resize()` calls

Modern C++ is becoming such a complex language with conflicting goals of elegance and performance, so it's hard to know which things are freebies or costlies.

13. Bitwise Operations

AVX C++ Bitwise Operators

Here's a refresher on the standard C++ bitwise operators:

`x & y` — binary bitwise-AND

`x | y` — binary bitwise-OR

`x ^ y` — binary bitwise-XOR

`x << y` — binary left bitshift

`x >> y` — binary right bitshift

`~x` — unary bitwise-complement (bitwise-NOT)

There are equivalents of these operations in AVX coding, such as:

Bitwise-AND — `_mm256_and_s256()`

Bitwise-OR — `_mm256_or_s256()`

Bitwise-XOR — `_mm256_xor_s256()`

Bitwise-NOT — there's no equivalent AVX function!

Note that AVX-1, AVX-2 and AVX-512 have different names for these primitives. But the operations are the same, just on more bits. There are also different intrinsic function names for these AVX operations on various differently sized integers, signed versus unsigned integers, or floating-point operations. Instead of bitwise-NOT, AVX has "NAND" or often called "andnot". Intrinsics such as `_mm256_andnot_s256()` implement the bitwise double-operation:

```
a = b & ~ c;
```

Bitwise-NOT Emulation

As mentioned above, there's no AVX equivalent of the bitwise-not or one's complement ~ standard C++ operator. Instead, this is usually emulated with two instructions, based on the identity:

```
~x == x ^ 0xFFFFFFFF
```

A register with all 1s can be created from integer `-1`, such as:

```
// XOR with all 1s
a = _mm256_xor_s256(b, _mm256_set1_epi32(-1));
```

Alternatively, it can be done using a couple of AVX tricks with two instructions, because the "set" AVX instructions can be expensive. One way is to use the "`cmpeq`" (compare-equal) AVX instruction, because the AVX version of "true" is actually `0xFFFFFFFF` (i.e., `0xFF` in every byte). Hence we can set up a dummy "`0==0`" AVX comparison that's always true.

```
__m256i z = _mm256_setzero_si256();      // Zeros
__m256i ones = _mm256_cmpeq_epi8(z, z); // All 1s
a = _mm256_xor_s256(b, ones);            // XOR with 1s
```

Notes on Bitwise Coding

Binary literals. A reminder that C++ also supports binary literal constants with a "`0b`" prefix, similar to the hexadecimal "`0x`" prefix. For example, to represent the constant `10` (ten), your C++ code can use:

```
const int ten = 10;     // decimal
const int ten = 0xA;    // hexadecimal
const int ten = 012;    // octal
const int ten = 0b1010; // binary
```

Bitwise badness: A few pitfalls in C++ bitwise operators should be mentioned:

- Integer-only: the C++ bitwise operators do not work on floating-point data types.
- Quiet overflow: if you do anything to overflow an integer type, nobody's going to tell you. For example, shifting the sign bit too far left with "`1<<32`" instead of "`1<<31`" will simply lose it. You might get a compiler warning, though.

- Two is not better than one. The `&` operator is bitwise, but `&&` is logical. Similarly, `|` and `||`. It's the reverse for < and << or > and >>. Choose the wrong one and you might get a compiler warning, if the stars are aligned and the wind is blowing easterly.
- Operator precedence is tricky and not what you'd expect (it's arguably broken, but rather too late to fix), so use lots of parentheses in bitwise expressions, and don't ignore C++ compilation warnings.
- Bitwise operators are not always well-defined on negative values (e.g., bitwise right shift is officially "undefined behavior" on a negative), so it's best to use "`unsigned`" types as operands to bitwise operators. Note also that it's often useful to add the suffix letter "`u`" to integer constants (e.g., `10u`, `0xAu` or `0b1010u`), when dealing with bitwise operations. This makes the constant of type "`unsigned`" and avoids various bitwise operator problems with signed numbers.

Bitwise operation algebraic properties: The interaction with zero is an important difference between the main operations:

- Bitwise-AND with zero equals zero: `a & 0 == 0`
- Bitwise-OR with zero equals the other value: `a | 0 == a`

The following inequalities for bitwise operators on non-negative integers can also be useful to know:

- Bitwise-AND only clears bits and is <= each operand: `a & b <= a`
- Bitwise-OR only sets bits and is >= each operand: `a | b >= a`
- Bitwise-AND equals the larger value only for equal numbers.
- Bitwise-OR equals the larger value only for subset bit patterns.

Addition versus bitwise operations: The relationship between the bitwise operators and the integer "+" operator can be useful to understand:

- Bitwise-AND is <= the sum of its operands: `a & b <= a + b`
- Bitwise-AND equals addition only if both numbers are zero.
- Bitwise-OR is >= the sum of its operands: `a | b >= a + b`
- Bitwise-OR equals addition only for disjoint bit sets or zeros.

Note that these relationships are for positive integer values. Bitwise operators need positivity in their daily lives, whereas addition is fine with lots of negativity.

Bit Flag Basics

The main use of C++ bitwise operators is to use bit flags in integer variables, which is very efficient in both storage space and execution time. A vanilla "`int`" can store 32 bit flags, and a "`long`" can store 64 bits. The basic bit operations in C++ use these bitwise operators:

- Check a bit — bitwise-AND (`&`)
- Set a bit — bitwise-OR (`|`)
- Toggle a bit — bitwise-XOR (`^`)
- Clear a bit — bitwise-AND with complement (`&` with `~`)

Here are some example macros for examining the bits in a 32-bit integer, which should be of "`unsigned int`" type:

```
// Bit Flags in Integers
#define AUSSIE_ONE_BIT_SET(x, b)    \
  (( ((unsigned)(x)) & ((unsigned)(b))) != 0 )
#define AUSSIE_ANY_BITS_SET(x, b) \
  (( ((unsigned)(x)) & ((unsigned)(b))) != 0 )
#define AUSSIE_ALL_BITS_SET(x, b) \
  ((((unsigned)(x))&((unsigned)(b))) == ((unsigned)(b)))
#define AUSSIE_NO_BITS_SET(x, b)  \
  (( ((unsigned)(x)) & ((unsigned)(b))) == 0 )
```

The corresponding macros to set and clear these bit flags are:

```
#define AUSSIE_SET_BITS(x, b)     \
  (( ((unsigned)(x)) | ((unsigned)(b))))
#define AUSSIE_CLEAR_BITS(x, b)   \
  (( ((unsigned)(x)) & (~((unsigned)(b)))))
#define AUSSIE_TOGGLE_BITS(x, b) \
  (( ((unsigned)(x)) ^ ((unsigned)(b))))
```

Yikes! What a mess! But all those parentheses are necessary to avoid precedence issues with preprocessor macros.

Bit Sets

You can consider a 32-bit integer to be a "bit set" of 32 distinct bit flags, where all 1s represent a bit flag that is in the set. A bit set is an inherently parallel architecture, even in ordinary sequential C++ code. The basic idea is that a 32-bit unsigned int stores 32 bit flags.

Certain actions on the integer as a whole effectively process 32 bits in parallel. For example, it is fast to check if any bits are set by testing if the whole integer is zero.

In regards to bit sets stored in an integer, the basic set operations can be implemented very efficiently with C++ bitwise operators:

- Bitwise-AND (`&`) — intersection
- Bitwise-OR (`|`) — union
- Bitwise-complement (~) — set complement (negated set)
- Bitwise-and-complement ("`A&~B`") — set difference (set minus)

In addition, there are a number of fast operations that can be useful for bit sets:

- Integer zero — null set of bits.
- Integer negative-one — full set of all 1s.
- Bitwise "popcount" — set cardinality or number of elements.

Example code with these ideas for 32-bit sets implemented as unsigned integers:

```
u != 0          // Test if any bit is set
u3 = u2 & u1;   // Intersection of sets (Bitwise-AND)
u3 = u2 | u1;   // Union of sets (Bitwise-OR)
u3 = u2 ^ u1;   // Toggle bits in sets (Bitwise-XOR)
u3 = ~u1;       // Set complement or inverse
```

The total number of bits set out of 32 can be computed fast as a "popcount" operation using intrinsic functions, such as "`__popcnt`" in Microsoft Visual Studio and "`__builtin_popcount`" for GCC (there are also versions for 64-bit longs). In x86 architectures, popcount is a single CPU instruction (`POPCNT`) implemented in hardware, and is therefore very fast.

There's no SIMD popcount instruction in AVX-1 or AVX2. However, in AVX-512 there is a parallel operation with the `_mm512_popcnt_epi64()` intrinsic.

Note that these C++ macros assume type "`unsigned int`" in 32 bits, and therefore 32 distinct flags in a single integer variable. For more, the "`unsigned long`" type can be used (64-bit), and there is also the "`long long`" type (128-bit).

The above macros would need to be changed to use type casts to "`unsigned long`" rather than just "`unsigned`" for a 64-bit version. For even more bits, a data structure called a "bit vector" can be implemented as an array of unsigned integers, which generalizes the bit set idea.

Bitwise Intrinsic Functions

Intrinsic functions, or "builtin" functions, are special C++ functions that are specific to the compiler environment. For example, Microsoft Visual Studio and GCC have different builtins. Intrinsics are usually implemented in very efficient ways, often directly mapping to CPU instructions, so they can be very powerful optimizations.

Some of the useful builtin functions for integer bitwise arithmetic are listed below. Most of these functions are for "`int`" or "`unsigned int`" (32-bit), but have other versions for `long` 64-bit or `unsigned long` 128-bit types. There isn't usually a version for "`short`" 16-bit integers.

Count Leading Zeros (CLZ): Various functions count the leading zeros, or similarly, the offset of the first set bit. This is scanning the bits from left-to-right and finding the most significant bit. One application of the CLZ intrinsic is a fast way to compute a truncated log2 of an integer, or similarly, computing the highest power-of-two in a number.

- `VPLZCNTD` or `_mm512_lzcnt_epi32` (AVX-512): SIMD leading zeros for 32-bit lanes.
- `VPLZCNTQ` or `_mm512_lzcnt_epi64` (AVX-512): SIMD count leading zeros for 64-bit lanes.
- `_BitScanReverse` (Microsoft intrinsic `<intrin.h>`): Finds the most-significant bit in a 32-bit integer. There's also `_BitScanReverse64`.
- `clz`: Count leading zeros (various versions); also sometimes called "`nlz`" for "number leading zeros".
- `__lzcnt`: Leading zeros count in Microsoft Windows intrinsics, use `<intrin.h>` for Microsoft Visual Studio C++.
- `__builtin_clz` (count leading zeros): GCC function to count the number of leading prefix zeros in an unsigned integer.
- `_CountLeadingZeros`: Microsoft `<intrin.h>` ARM intrinsics.

For all you silicon addicts, here's the CPU instructions are underpin these intrinsics:

- `BSR`: Bit Scan Reverse x86 assembler instruction.
- `LZCNT`: x86 instruction for leading-zero count, similar to BSR.

Count Trailing Zeros (CTZ): Contrasting to the leading zero functions, these functions find the zeros on the right-hand-side of an integer. This is the least-significant bit.

- `_BitScanForward` (Microsoft intrinsic `<intrin.h>`): Finds the least-significant bit set. Long int version is `_BitScanForward64`.
- `__builtin_ctz` (count trailing zeros): GCC function counts zero bits on the right (least-significant bits).
- `ffs/ffsl`: Find first set (least-significant bit).
- `__builtin_ffs` (find first set): GCC function: find first set bit from the least significant bits (from the right bits).
- `_tzcnt_u32` or `_tzcnt_u64`: MSVC versions of trailing zeros.
- `VPTZCNTD` or `_mm512_tzcnt_epi32` (AVX-512): SIMD trailing zeros in 32-bit integers.
- `VPTZCNTQ` or `_mm512_tzcnt_epi64` (AVX-512): SIMD 64-bit version.

The related x86 CPU hardware instructions are:

- `BSF`: Bit Scan Forward x86 assembler instruction.
- `TZCNT`: x86 instruction for trailing-zero count, similar to `BSF`.

If you'd rather code it yourself, there's Brian Kernighan's bit trick for LSB: bitwise-and of `n` and `n-1` (i.e., in C++ `n&(n-1)` finds the lowest set bit). But using the intrinsics should be faster.

Popcount (Set Bits Count): The count of 1s in a number is known as the "popcount" (which is short for population count) and there are various intrinsics:

- `__builtin_popcount`: (GCC) count the 1s in an unsigned integer.
- `BitOperations.PopCount`: Microsoft intrinsic for bitwise popcount.
- `__popcnt`: AMD x86 popcount intrinsic using `POPCNT` x86 instruction (Microsoft platform)
- `_mm512_popcnt_epi64()`: AVX-512 SIMD intrinsic for popcount.
- `_mm_popcnt_u32`: Intel x86 popcount intrinsic using `POPCNT` x86 instruction (Microsoft platform); use `<intrin.h>` on MSVS C++.
- `__builtin_parity`: GCC function tracking bitwise binary parity (whether the number of 1s is odd or even).

The x86 CPU hardware instruction is `POPCNT`, which computes the popcount faster than a hummingbird's wings.

Example: Integer Popcount

The "popcount" is short for "population count" of a binary number, and is the number of binary 1s in an integer number. This has applications such as quickly counting the number of elements in a bit set or bit vector.

Bitwise arithmetic can be used to check for a '1' value in each bit of an integer. Usually an unsigned type is used (as below), but bit twiddling of signed integers is also possible. This is the slow version in C++ that simply loops through each bit, checking if it is set:

```
int aussie_popcount_basic(unsigned int x)
{
     // Count number of 1s
     const int bitcount = 8 * sizeof(x);
     int ct = 0;
     for (int i = 0; i < bitcount; i++) {
         if (AUSSIE_ONE_BIT_SET(x, 1u << i)) ct++;
     }
     return ct;
}
```

Kernighan Popcount Algorithm: A faster version is to use a bit trick found by Brian Kernighan, author of *The C Programming Language*. For all values of `n`, the previous number `n-1` has one less bit set. So, if you do bitwise-AND of `n` and `n-1`, it removes the rightmost bit that is 1 (i.e., least significant bit). Hence, you can use this to optimize popcount by only looping as many times as there are 1s in the number (rather than always doing 32 iterations). Here's the new C++ code:

```
int aussie_popcount_kernighan_algorithm(unsigned int x)
{
     // Count number of 1s with Kernighan bit trick
     int ct = 0;
     while (x != 0) {
         x = x & (x - 1);  // Remove rightmost 1 bit
         ct++;
     }
     return ct;
}
```

Intrinsic Popcount Functions: The Kernighan method is faster, but far from optimal. To do it super-fast, we have to look at existing builtin function primitives. For example, Microsoft intrinsics include "`__popcnt`" or "`_mm_popcnt_u32`" (in header file `<intrin.h>`).

The GCC library has a "__builtin_popcount" function, which count the number of 1s in an unsigned integer. On x86 CPUs, the underlying intrinsics should be using the x86 assembler instruction named POPCNT. Here is some example C++ code that works for Microsoft Visual Studio:

```
int aussie_popcount_intrinsics2(unsigned int x)
{
    return __popcnt(x);  // Microsoft intrinsics
}
```

Obviously, a faster version is to declare this one-line function as "inline" in a header file, or to convert to a C++ preprocessor macro, such as:

```
#define AUSSIE_POPCOUNT(x) (__popcnt((unsigned)(x)))
```

Example: Bitwise Log2 on Integers

Calculating the base-two logarithm of integers can be quite useful. There are various algorithms that use logarithms in AI.

Let's calculate the integer logarithm of an integer. This means we aren't doing the proper fractional logarithm of a number, but we are truncating it down to the nearest integer. For example, `log2(7)` will be truncated to `2`, rather than `2.807`. Note that we're assuming the input is unsigned numbers, since logarithms with negatives are undefined. Also, we have to decide how to handle zero, because `log2(0)` is undefined (or negative infinity if you prefer).

A simple way to implement a truncated integer `log2` function is to use floating-point functions and type casts back to int:

```
int aussie_log2_integer_slow(unsigned int u)
{
    // Slow float-to-int version
    return (int)log2f(u);
}
```

This works, but it's inefficient to use floating-point arithmetic on integers. Surely there's a faster way?

After some thoughts about binary bits, we notice that `log2` of an integer is just the index position of the highest bit in a number. The `log2` of `1` is `0`, because the '1' is in position 0. The `log2` of `2` (binary `10`) is `1` because the leftmost 1 is in position 1. The `log2` of `4` (binary `100`) is `2`, where the 1 is in index 2.

The number 7 is binary 111, so log2 is the position of the leftmost 1, which is position 2. So, log2(7) is the same as log2(4), but log2(8) is 3.

There are numerous builtin bitwise functions that can find the leftmost set bit. With sudden insight, we note that we can use "CLZ" (count leading zeros) to compute how many prefix zeros there are before the leftmost 1 bit (i.e., counts the zeros up to the most-significant bit from the left). We can then compute the bit index position from the right in a 32-bit integer as "32-CLZ". It's on the right track, and a bit of testing shows that the formula to use is "32-CLZ-1".

Here's some example code that uses this CLZ method to compute log2 of an integer. This works on Microsoft Visual Studio using the <intrin.h> header file to declare intrinsics.

```
int aussie_log2_integer_clz_intrinsic(unsigned int u)
{
    // LOG2 using CLZ
    int clz = __lzcnt(u);  // Count leading zeros
    const int bits = 8 * sizeof(u);
    return bits - clz - 1;
}
```

And here's the macro version for those who don't trust compilers to inline properly:

```
#define AUSSIE_LOG2_LZCNT(u) \
 ((8 * sizeof(unsigned)) - (__lzcnt((unsigned)(u))) - 1)
```

And this is actually not optimal. We really should help the C++ optimizer by reordering this to move the "-1" subtraction operation next to the other constant, noting that "sizeof" is a compile-time constant expression in C++. Putting them together would make sure that the compiler correctly merges these operations using constant folding. On x86 implementations, the CLZ builtin functions are presumably using the x86 LZCNT or BSR assembler instructions, which are both similar and fast.

Bug alert! Note that you can't use "ffs" (find first set bit) for this log2 method, because it gives you the offset of the least-significant set bit (i.e., the rightmost bit rather than the leftmost bit). The other x86 instructions of TZCNT (Trailing Zeros Count) and BSF (Bit Scan Forward) are also incorrect.

Example: Highest Integer Power-of-Two

Another simple trick related to the `log2` calculation is to truncate a number to its largest power-of-2. This is equivalent to the value of its leftmost bit in binary representation.

For example, `8` (binary `1000`) stays as `8`, because it's `2^3`, but `7` (binary `111`) reduces down to `4` (binary `100`), which is `2^2`. As with the truncated integer `log2` calculation, this method focuses on computing the leftmost 1 bit, which is known as the Most-Significant Bit (MSB).

Whereas the `log2` calculation found the index position of that MSB, this power-of-two calculation requires the *value* of the MSB. In other words, we need to find the bit that is the MSB, and then keep only that bit. A simple way to do this is to compute the `log2` of the integer efficiently, and then left-shift a `1` by that many places (using `unsigned` type). The basic idea is:

```
int bitoffset = log2_integer_fast(i);
int highestpowerof2 = 1u << bitoffset;
```

Note that this doesn't handle cases like zero, so it still needs a bit of extra code polishing work.

Integer Overflow and Underflow

Integer arithmetic overflow and underflow have traditionally been ignored in C++ programs, mostly by assuming that operations won't exceed the range of 32-bit integers. Most platforms don't fail on integer overflow, and quietly continue, without even triggering a signal like `SIGFPE` (floating-point error).

The absence of runtime warnings can potentially leave insidious bugs in your code, and is also an undefended attack vector for security. Also, perhaps ignoring overflow isn't the best strategy.

Integers have a fixed range of numbers that they can represent. For example, a signed 16-bit integer represents the relatively small range of `-32,768` to `+32,767`, and an unsigned 16-bit number can be from `0` to `65,535`. A 32-bit signed integer has a much bigger range from about negative 2 billion (`-2,147,483,648`) to about positive 2 billion (`+2,147,483,647`). For an unsigned 32-bit integer, there's no negatives, and the range is from zero up to about 4 billion (`+4,294,967,295`).

Feel free to memorize those numbers, as you'll be needing them at least once a decade. The ranges for 64-bit integers are massive numbers around `2^64`, which is approximately decimal `10^19`.

If integer arithmetic on a data type falls outside the range supported by that integer type, then an overflow or underflow occurs. There are symbolic constants for the minimum and maximum numbers for many types declared in the `<limits.h>` standard header file.

- `int` — `INT_MAX` and `INT_MIN`
- `unsigned int` — `UINT_MAX` and `UINT_MIN`

The effect of integer overflow or underflow is platform-specific, but on most platforms, it is usually: *nothing!* It's a silent insidious bug in many cases. For a signed integer, overflow quietly wraps around from positive to negative, and underflow does the reverse.

Here's an example of overflow of an `int` type:

```
int x = INT_MAX;
assert(x >= 0);
++x;  // Overflow!
assert(x < 0);
```

And this is underflow of `int`:

```
int x = INT_MIN;
assert(x < 0);
--x;  // Underflow!
assert(x > 0);
```

Floating-point types can represent much larger magnitude numbers than integers. Hence, another way for an integer to overflow is in a conversion from floating-point numbers.

```
float f = (float)INT_MAX * (float)INT_MAX;  // Fine!
int x = (float)f;  // Overflow!
```

For an unsigned integer, the results are a little different, since negatives are not possible. Instead, overflow wraps around from a large number to zero, and underflow (going below zero) wraps around to the largest unsigned number.

Preventing Integer Arithmetic Overflow. There's not really a good way to detect arithmetic overflow or underflow before it happens. Post-testing is easier.

For example, GCC and Clang have some intrinsics, such as "`__builtin_add_overflow`" for addition, which use post-testing of the x86 CPU overflow or carry flags for detecting integer overflow, and return a Boolean flag which you can use. The GCC documentation say it uses "conditional jump on overflow after addition" and "conditional jump on carry" for unsigned overflow. Here's an example:

```
if (__builtin_add_overflow(x, y, &z)) {
    // Overflow!
}
```

The mainstream prevention strategy is simply to choose a big integer type (at least 32-bit) and then hope that no outliers occur in your input data. Most programmers let the overflow occur and then check. Or rather, just between you and me, most programmers simply don't even check at all!

Technically, integer overflow is "undefined behavior" on C++, and it's certainly non-portable, so you really should check. But most platforms handle it the same way, by quietly wrapping the integers around in two's complement form.

Increment overflow. For incrementing integers, you can do a pre-test like:

```
if (INT_MAX == x) {
    // Overflow!
}
else {
    x++;  // Safe increment
}
```

Addition overflow. And here's a version to pre-test addition of two positive integers for overflow:

```
if (x > INT_MAX - y ) {  // x + y > INT_MAX
    // Overflow!
}
else {
    x += y;  // Add safely
}
```

Multiplication overflow. The test for multiplication overflow is even worse because it uses division:

```
if (x > INT_MAX / y ) {  // x * y > INT_MAX
    // Overflow!
}
else {
    x *= y;  // Multiply safely
}
```

Head in the sand approach. Unfortunately, pre-testing for overflow is massively inefficient, as shown above. Do you really want to do this for every addition or increment? Even post-testing for overflow isn't much better. Overall, there's good reason why most C++ programmers just skip it, and hope for the best.

Overflow management. The alternative to ignoring the problem is to consider various different risk mitigation strategies for integer overflow:

- Larger data types (e.g., `long`) for a larger range.
- Use floating-point types instead.
- Use `unsigned` type for non-negative variables (e.g., sizes, counts).
- Use `size_t` for the `unsigned` variable type (it's standardized).
- Enable compiler runtime checks (when debugging/testing)
- Range checking input numbers (e.g., model weights).
- Post-testing the sign of arithmetic results.
- GCC and Clang intrinsic functions with overflow testing.
- The `<stdckdint.h>` header file in C23 (that's the C standard, not C++23).
- Safe integer class wrappers.

Runtime overflow detection. Some C++ compilers provide limited support for runtime error checking of arithmetic. The x86 CPU has builtin overflow detection, with a quietly-set overflow flag and a carry flag, which some C++ compiler-writers have made use of.

GCC has an "`-ftrapv`" option which elevates overflow errors (presumably by using post-checking). GCC has defined a number of C++ intrinsic functions which you can use to perform overflow-safe integer arithmetic, such as:

- `__builtin_add_overflow` — addition
- `__builtin_mul_overflow` — multiplication

Microsoft Visual Studio C++ provides the "`/RTC`" option, which stands for "Run-Time Checks", or there's "Basic Runtime Checks" in the MSVS IDE Project Settings. However, these MSVS features don't check much for arithmetic overflow, with a focus on stack frame checking and uninitialized variables. The closest is "`/RTCc`" to detect data type truncations at runtime.

There's also a runtime debugging tool that focuses on integer overflow and other oddities. It's named "Undefined Behavior Sanitizer" or UBSAN for short. It works like Valgrind, by adding runtime instrumentation code.

Safe integer classes. Currently there's no standard safe integer types in C++, but adding them was unsuccessfully proposed in 2016. If you like a busy CPU, and what programmer doesn't, you can replace all `int` variables with "safe integer" class objects, with many examples of such classes available on the Internet. They're probably not as bad as I've implied, since C++ inlining should make the critical path quite short.

Missing Bitwise Operators: NAND, NOR, XNOR

Note that there's no simple operator for NOR, NAND or XNOR in standard C++. However, as discussed earlier, there is an AVX SIMD version for NAND with the "andnot" instructions.

You might need these extra operators, since neural networks uses these uncommon bitwise operations more than normal C++ coding. For example, XNOR is needed as the vector dot product operator for binarized bit vectors, such as in binary quantization and also XNOR neural networks.

These missing operators can be easily simulated using two C++ bitwise operations, with a binary bitwise operation and the "~" bitwise two's complement unary operator afterwards.

```
NAND(x,y) = ~(x & y)
NOR(x,y)  = ~(x | y)
XNOR(x,y) = ~(x ^ y)
```

So, you can just code this as fast C++ macros, right?

```
#define NAND(x,y) ~(x & y)  // Bug alert!
#define NOR(x,y)  ~(x | y)
#define XNOR(x,y) ~(x ^ y)
```

No, this is broken in about half a dozen ways.

To write macros correctly, you need to ensure there's parentheses around the whole expression, and also around each parameter name, to avoid getting bitten by C++ macro expansion operator precedence problems. And these macros also don't work correctly if you pass in a non-unsigned integer.

Here's some example C++ macros that work for 32-bits:

```
#define AUSSIE_BITWISE_NAND(x,y) \
  (~(((unsigned)(x)) & ((unsigned)(y))))
#define AUSSIE_BITWISE_NOR(x,y)  \
  (~(((unsigned)(x)) | ((unsigned)(y))))
#define AUSSIE_BITWISE_XNOR(x,y) \
  (~(((unsigned)(x)) ^ ((unsigned)(y))))
```

You could also declare these macros as "`inline`" functions if you prefer. Note that these macros have a lot of parentheses to avoid various insidious precedence errors, and they also are limited to 32-bit operations. For 64-bit, you'd need to create alternative "`unsigned long`" versions.

These NAND/NOR/XNOR macros are convenient, but not very efficient since they perform two arithmetic operations. Single-operation versions are available in assembler if you really need them, accessible via C++ builtin intrinsic functions such as:

- `_kxnor` — x86 intrinsic for XNOR bitwise operation.
- `KXNORW/KXNORB/KXNORQ/KXNORD` — x86 assembler bitwise XNOR operations.
- `VPTESTNMB/VPTESTNMW/VPTESTNMD/VPTESTNMQ` — x86 assembler bitwise NAND operations.

Note for the sake of completeness that there are more weird bitwise operators that do different things on a pair of bits. There are four input combinations and therefore 16 possible binary operator functions. There are three C++ bitwise operators (AND/OR/XOR), plus the three extra ones coded above (NAND/NOR/XNOR), two trivial always-zero and always-one operations, two copy-operand functions, and six other ones that are equivalent to variations with negated operands (e.g., "`x&~y`" is one).

I'm not sure why you needed to know that.

Bitwise AI Applications

Bitwise operations are a well-known coding trick that has been applied to neural network optimization. Bitwise-shifts can be equivalent to multiplication and division, but faster. Other bitwise operators can also be used in various ways in inference algorithms. Some common uses of bitwise operators in AI include:

- **Arithmetic computation speedups:** Bit tricks are used in optimizations of multiplication operations with bitshifts, and also faster approximate arithmetic methods.
- **Sign bit manipulation:** Various optimizations are possible by direct bitwise operations on the sign bit of integers or floating-point numbers. For example, the RELU activation function tests for negatives, which are changed to zero, but positive values are unchanged. This can be implemented efficiently as a sign bit test.
- **floating-point bit operations**: The bits of the numeric representations in IEEE 754 floating-point numbers, or the Google `bfloat16` type, include a sign bit, an exponent, and a mantissa. Normal bitwise arithmetic operators cannot be applied to floating-point numbers, because the C++ bitwise and bitshift operators only work on integer types. However, floating-point numbers are really just integers underneath, so there are various tricky ways that bitwise operators can be used on the underlying IEEE standard bit representations that are used by floating-point numbers. This is discussed in the next chapter on floating-point optimizations.
- **Look-up Tables**: Algorithms that use table lookups for speed improvement typically involve bitwise shifts in computing the table offset.
- **Data structures:** Some data structures used in optimization of neural networks that involve bits include hashing and Bloom filters.

Bits of AI Research: Some of the advanced areas where bitwise optimizations have been used in neural network research include:

- **Power-of-two quantization (bitshift quantization):** quantize weights to the nearest power-of-two, bitwise shifts can replace multiplication.
- **Bitserial Operations**: Bitserial operations are bitwise operations on all the bits of an integer or bit vector. For example, the "popcount" operation counts how many 1s are set in the bits of an unsigned integer. The bitserial operations can be useful in neural network inference for computing the vector dot products in binary quantization or 2-bit quantization.
- **Advanced number system division:** See dyadic numbers and dyadic quantization for an obscure number system involving power-of-two division, which can be implemented as bitwise right-shifting.

- **Low-bit integer quantization:** When quantized to only a few bits, inference can use bitwise arithmetic and bitserial operations to replace multiply-accumulate. The main examples are binary quantization and ternary quantization, both of which avoid multiplications in favor of bitwise operations (or addition) and sign bit handling.
- **Shift-add networks:** Multiply-and-add (or "multiply-accumulate") can be replaced with bitshift-and-add.
- **Bit arithmetic neural networks**. These are neural networks where the neurons operate as bitwise operations. See Weightless Neural Networks.
- **XNOR Networks:** XNOR neural networks are similar to binarized networks. Their internal operations rely on the bitwise XNOR operation. The idea is that XNOR is actually an implementation of the multiplication operation on binary values. There's no builtin C++ operator for binary XNOR. However, there is always hardware XNOR support, such as a 64-bit XNOR instruction in the x86 CPU instruction set.

References on Bitwise Operations

If I've whetted your appetite for bit fiddling magic, there's plenty more:

1. Sean Eron Anderson (2005), *Bit Twiddling Hacks*, Stanford University, https://graphics.stanford.edu/~seander/bithacks.html
2. Ian Brayoni (2020), https://github.com/ianbrayoni/bithacks (Python code inspired by Sean Eron Anderson's Bit Twiddling Hacks.)
3. Henry S Warren (2012), *Hacker's Delight, 2nd Edition*, Addison-Wesley Professional, https://www.amazon.com/Hackers-Delight-2nd-Henry-Warren/dp/0321842685 Code: https://github.com/hcs0/Hackers-Delight
4. Antonio Gulli (2014), *A Collection of Bit Programming Interview Questions solved in C++ Kindle Edition*, https://www.amazon.com.au/Collection-Programming-Interview-Questions-solved-ebook/dp/B00KIIDPUG/
5. Jörg Arndt (2010), *Matters Computational: Ideas, Algorithms, Source Code*, https://dl.acm.org/doi/10.5555/1941953, https://www.jjj.de/fxt/fxtpage.html#fxtbook, Code: https://www.jjj.de/bitwizardry/bitwizardrypage.html
6. Sigrid/Jasper Neuman (2023), Programming pages, http://programming.sirrida.de/
7. Harold (2023), *Bits, Math and Performance*, Sep 2023, http://bitmath.blogspot.com/
8. Stephan Brumme (2023), *The bit twiddler*, https://bits.stephan-brumme.com/
9. Gurmeet Manku (2008), *Fast Bit Counting*, 5 Aug 2008, https://gurmeet.net/puzzles/fast-bit-counting-routines/

14. Floating-Point Computations

What are Floating-Point Numbers?

Floating-point numbers are typically stored in 32 bits for single-precision C++ "`float`" types, and it's actually a 32-bit integer behind the scenes. The main floating-point types that you already know from C++ programming are:

- Single-precision floating-point — 32-bit `float` (FP32)
- Double-precision floating-point — 64-bit `double` (FP64)

The smaller 16-bit floating-point numbers that are never used in everyday C++ coding, but are important for AI, include:

- Half-precision IEEE type — 16-bit "`short float`" (FP16)
- Half-precision Bfloat16 type — 16-bit "Brain float" (BF16)

If only there was really a "`short float`" type in C++. The BF16 type is the non-IEEE 16-bit float version from Google Brain. Note that there is new standardized support for these 16-bit types in C++23.

Which type of floating-point number should you use? That's when things get tricky, because there are many wrinkles in the choice between 32-bit and 16-bit floating-point. It's not always clear which floating-point size is the best to use. FP32 is the most common size used in basic Transformer inference, but FP16 is a good choice for quantization of models, because they are compressed to half the size and retain good accuracy. And BF16 has been very effective in terms of GPU-accelerated algorithms.

Some hardware accelerators support different formats and sizes for their parallel operations. And there are various software problems with portably coding 16-bit floating-point data types in C++, along with variable hardware support for 16-bit operations across platforms.

Less importantly, there are also some other floating-point sizes, both bigger and smaller:

- Quarter-precision type — 8-bit floating-point (FP8)
- Quadruple-precision type — 128-bit "quad" floating-point (FP128)

FP8 is mainly seen in research papers, and hasn't really caught on for quantization (8-bit integers are typically used instead). The bigger sizes FP64 and FP128 aren't really needed to make your model work accurately, so their significant extra cost in speed and size isn't worthwhile for only a small perplexity gain in most use cases.

Bit Representations of Floating-Point Numbers

Standardized bit patterns are used to represent floating-point numbers in a kind of scientific notation. There are three types of bits:

- Sign bit
- Exponent bits
- Mantissa bits

Firstly, there's one bit for the sign, indicating whether the whole number is positive or negative. Then the remaining bits are split up between the "exponent" (i.e., the "power"), and the "mantissa" (also called the "digits" or the "significand" or the "fraction"). In a standard 32-bit "`float`" type used in AI, there is:

- 1 sign bit
- 8 exponent bits
- 23 mantissa bits

How does that even make a number? Well, it's like scientific notation, if you are familiar with that. The exponent is the power and the mantissa is the digits.

Let's pretend computers use decimal digits. If it were in base 10 storage, the decimal number `1234` would be stored as:

- "`0`" for the sign bit — because non-negative.
- "`3`" in the exponent — the power is `10^3=1000`.
- "`1234`" as the mantissa — the digits make the fraction "`1.234`".

This would represent `+1.234x10^3` (which hopefully equals `1234`). That's how it would work for a decimal version.

But, as you know, silicon beasts are not decimal. A floating-point number is actually stored in binary, in a kind of base-two "binary scientific notation" numbering scheme. So, conceptually, `1234` would be stored as a power-of-two exponent that represents the largest power-of-two, which would be `1024`, because `2^10=1024`, so the exponent has to store power "`10`" (ten), which is `1010` in binary. And the `1234` would be converted to whatever the heck `1234/1024` is when you represent that in binary 0's and 1's, and remove the decimal point (which is implicitly "floating," you see?).

It's more complicated than this, of course. That's what standards are for! The exponent bits are actually stored with an "offset" number (also called a "bias"), which differs by the size of the exponent bits. And there also some special bit patterns for particular numbers, such as zero or "`NaN`" (not-a-number).

Clear as mud? Don't you wish someone could go back in time and invent a base-10 computer?

Standardized Bit Representations

There's nothing magical about the choices of how many exponent versus mantissa bits. In the early days, there were many variations, but then they were mostly standardized by the IEEE 754 standard.

32-bit Floating-Point Numbers: The most common type of floating-point is 32-bits, such as the C++ "`float`" type. Other than the sign bit, there are usually 31 bits to split between the two other types, and the standard method is:

- Standard FP32 (IEEE754). Usually a "`float`" in C++, or "single precision" number. Standard 32-bit floating-point is represented in binary as: 1 sign bit, 8 exponent bits, and 23 mantissa bits (plus an implied prefix '1' mantissa bit that isn't actually stored, so it's really 24 bits of mantissa values). The exponent is stored with offset 127.

16-bit floating-point Numbers: With the "half" float types, there are 16 bits. There are a few common representations of floating-point numbers in different numbers of bits.

The main ones are:

- Half-precision (FP16). This is the standard 16-bit floating-point number, also sometimes called "`float16`". Annoyingly, there no standard "`short float`" or other widely used predefined type in C++, although the C++23 standard adds one, so this may be changing soon. The most common IEEE754-standardized version of FP16 type uses 1 sign bit, 5 exponent bits, and 10 stored mantissa bits (plus implicit mantissa bit makes 11 bits). The exponent is stored with offset 15.
- `Bfloat16` (brain float 16 or BF16): This is a different 16-bit floating-point numeric format, originally proposed by the Google Brain division, specifically for use in AI applications. `Bfloat16` has 1 sign bit, 8 exponent bits and offset 127 (like FP32), and 8 mantissa bits (7 stored, 1 implicit). It is like FP32 but with the two lowermost bytes just thrown away, so conversion between `bfloat16` and FP32 is simpler than converting from FP32 to FP16.

8-bit Floating-Point (FP8). The use of FP8 mainly appears in quantization research papers, but its usage is increasing within industry. There is usually 1 sign bit, 4 exponent bits, and 3 mantissa bits (which makes 4 bits with an implied extra mantissa bit). The other type of FP8 is 1 sign bit, 5 exponent bits, and 2 stored mantissa bits (3 bits total). Interestingly, the NVIDIA H100 GPU supports both of these FP8 formats.

FP16 Problems in C++

I already mentioned how there's not a standard half-precision type in C++, although that is fixable in the future, once compilers have implemented the C++23 standard. Here are some of the attempts at a 16-bit type:

- `__fp16` — only supported by ARM architecture.
- `_Float16` — not portably supported.
- `short float` — doesn't seem to exist (I'm just wishful-thinking!).
- `std::float16_t` — defined in the C++23 standard.
- `std::bfloat16_t` — defined in the C++23 standard.

So, as of writing, if you want to code a 16-bit float in a portable way with C++, there's an ugly hack: `short int`.

A less fixable obstacle is that converting between FP32 and FP16 is not easy because their exponent bit sizes are different. So, it's fiddly to code, and not very efficient.

The alternative idea is to use "`bfloat16`" (BF16), which is the upper-most two bytes of FP32. Converting is just a bitshift 16 places or playing with bytes, so it's faster than FP16.

However, BF16 isn't high precision. With 8 mantissa bits (7 stored, 1 implicit), that's only about 3 decimal digits, because `8/3.3=3`, and `3.3` is `log2(10)`, in case you were wondering. But it's not much worse than FP16, which is only about 4 decimal digits using 11 binary mantissa bits.

Representing Zero

The sign bit, exponent, and mantissa can represent a lot of numbers, but not zero. We cannot just set all the mantissa bits to zero, because that's not zero, which is rather strange.

There's an implicit extra "1" bit so all the mantissa bits clear isn't `0.0000`, it's `1.0000`. It always starts with a "`1`" and there's literally no way to represent `0.0000`.

Also, the exponent can represent `-127` to `+128`, but setting the exponent to `0` also isn't zero, because `2^0` is `1`. And `2^-127` is very small and does get us very close to zero, but it's also not zero. With sudden horrifying insight, we realize:

There's no way to represent zero!

The solution is that the IEEE 754 standard designers decided to treat all bits zero as being really zero. All bits zero in the exponent is `0`, but then subtracting the `127` offset, means that it is `-127` (the smallest number). So, if we clear all the exponent and mantissa bits to zeros, the number should be `1.0x2^-127`, but we can all pretend it's actually zero. Then we can do some pretend coding, ahem, I mean *microcoding*, so that all our Floating-Point Units (FPUs) pretend it's zero, too.

Negative zero. Weirdly, there are two zeros: normal zero and negative zero. The IEEE 754 standard allows two different bit patterns to mean zero, depending on the sign bit. If we clear all the exponent and mantissa to zero, then the sign bit zero means zero, but the sign bit set to "1" means "negative zero".

I'm not really sure what negative zero even means! But sometimes when you work with floats, a `0.000` number will get printed with a "-" in front of it. Maybe it's negative zero, or maybe it's a tiny negative number with hidden digits at the 15th decimal place.

Fortunately, most of the arithmetic operations treat negative zero the same as zero. The C++ compiler handles it automatically. Adding negative zero does nothing, and multiplying by negative zero is also zero. But one of the gotcha's if you're being tricky with the bits of a 32-bit floating-point number, by pretending it's a 32-bit integer: testing for zero isn't one integer comparison, it's two!

Representing Special Numbers

We've already discussed how zero is handled specially, and has a wonderful dichotomy. The full list of special floating-point numbers is:

- Zero
- Negative zero
- `+Inf` (positive infinity)
- `-Inf` (negative infinity)
- `NaN` (Not a Number)
- Denormalized numbers (subnormal numbers)

Whereas zero is represented by the exponent being all 0s, the special numbers `Inf` and `NaN` are represented by the exponent with all 1s. So, this means that the huge number `2^+128` is not actually represented, but reserved for these special values. And honestly, that's fine, because if `2^+128` isn't infinity, then I don't know what it is.

Infinity: `Inf` is represented by all 1s in the exponent, but all 0s in the mantissa. And if the sign bit is 1, then it's `-Inf` (negative infinity).

Not-a-Number: `NaN` also has all 1s for the exponent, but any other pattern of the mantissa bits means `NaN`. This means that there are many versions of `NaN`, for all variations of the mantissa bits, except when all mantissa bits are 0 (which means `Inf`). Also, if the sign bit is set, then the same patterns are also `NaN` (a kind of "negative `NaN`", but that distinction is rarely used).

Denormalized numbers: Apparently, the designers of the floating-point standards think there's a "huge" difference between `2^-127` and zero. So, they decided to "smooth" it out a little by using some special numbers called "denormalized numbers" (also called "subnormal numbers").

The standard does this by getting rid of the "implicit" mantissa bit. For one special exponent value, all 0s, the standard changes the meaning to consider the implicit hidden mantissa bit to be a leading 0, rather than a leading 1.

Hence, the mantissa can represent fractions less than `1.0`, such as `0.1101` rather than only `1.1101` (in binary). The special exponent with all 0s therefore never represents `-127`, but represents the special value zero (or negative zero) if all the mantissa bits are 0s, or a tiny denormalized number if any of the mantissa bits are set. And even though the exponent with all 0s should represent `-127`, we pretend that it is `-126`, one less, for the denormalized numbers, for "smoothness" reasons that I leave as an exercise to the reader, mainly because I don't understand it. Note that denormalized numbers can also be tiny negatives if the sign bit is set.

Denormalized numbers are all very, very tiny, being less than `2^-126`, so this feature of floating-point standardization is more useful for high-precision scientific calculations at NASA or SpaceX, rather than for most applications. In fact, here's the news about denormalized numbers in most coding:

We don't use denormalized numbers.

In fact, we hate them, because they make our FPU run slow. So, really, the slowness of our floating-point code is the fault of the FPU hardware engineers, as we've long suspected. Fortunately, there's a way to turn denormalized numbers off and run faster, which is discussed below.

To summarize and/or to further confuse things, the exponent has two special cases: all 0s and all 1s. If the exponent bits are all 0s, the number is either zero (or negative zero) or a denormalized number (a tiny positive or negative). If the exponent bits are all 1s, then the number is `Inf` or `NaN` (or negative `Inf/NaN`).

Testing for Special Values: The C++ standard has a number of fast routines to test a floating-point number. Some of the useful ones in `<cmath>` include:

- `std::isinf()`
- `std::isnan()`
- `std::isnormal()`
- `std::isfinite()`

For more general analysis of floats, `std::fpclassify()` in `<cmath>` returns a code that matches special enum values:

```
FP_INFINITE, FP_NAN, FP_NORMAL, FP_SUBNORMAL, FP_ZERO
```

Unfortunately, it's hard to distinguish positive and negative infinity, or to detect negative zero using these functions.

You'll need to add a call to the "`std::signbit`" function (since C++11 for `float` arguments or C++23 for `double`), which returns `true` if a floating-point number has the sign bit on. There also a "`std::copysign`" function to copy the sign from one `float` to another, which can be used for sign bit manipulations. Alternatively, define your own bitwise macro tricks for these inspections.

Underflow and Overflow

Underflow is when a tiny floating-point number becomes so small that we can only represent it as zero. This can be a very tiny positive or negative number. Note that a negative number with a huge magnitude (near negative infinity) isn't underflow; that's actually negative overflow. Underflow refers to tiny fractions.

Generally, underflow isn't a problem for most code, because a number that low isn't going to affect the results. Similarly, I don't think we need to worry much about subnormal/denormalized tiny numbers either. If a probability is `2^-127` (or `2^-126` for denormalized), well, it might as well be zero anyway.

If we're using `Bfloat16` for 16-bit processing, it still has 8 bit exponents, so the lowest value is almost the same number (about `2^-127`). If we've quantized the network to FP16 (also 16-bit but with a 5-bit exponent), then the lowest probability we can represent is `2^-31`, which is also a tiny probability.

Generally speaking, applications don't tend to worry about underflow in floating-point. If a floating-point calculation underflows, it should just go harmlessly to zero. More concerning would be integer underflow, which is a different issue of large negatives wrapping around to positives. Floating-point underflow is better behaved.

Overflow is when a number gets so large that it cannot be represented in floating-point. Note that there are two types of overflow: positive overflow and negative overflow.

The exponent is the problem for overflow. When the number is larger than the highest exponent power, then it's either a very large positive or a very large-magnitude negative number.

For an 8-bit exponent, that means `2^+127` (because `+128` is reserved for the special `Inf/NaN` numbers). For a 5-bit exponent in FP16, this means `2^+31`, which is, coincidentally, also a good salary to request at your performance review.

Overflow can be a problem, but usually only in the low-bit processing code where arithmetic computations can sometimes go too high. When overflow occurs, it could become a special floating-point number (NaN or Inf), or an integer number might toggle over to negative (e.g., if integer-only-arithmetic quantized).

FTZ and DAZ CPU Modes

In many CPUs, the need to handle overflow, underflow and denormalized values is a cause of inefficiency. The CPU can do floating-point computations faster if it can ignore those situations. This would be in violation of the IEEE 754 standard, but sometimes you have to sacrifice greatness for speed.

There are two commonly used modifications to CPUs that speed up floating-point arithmetic, by ignoring underflow and tiny numbers:

> **Flush-To-Zero (FTZ)**. This mode means that when the results are "subnormal" they are "flushed" to zero instead of calculating the correct "denormalized" result. Since these denormalized numbers are tiny, this isn't a concern in most code.
>
> **Denormalized-Are-Zero (DAZ)**. This is similar to FTZ, but allows treating inputs that are some type of denormalized floating-point as a zero input.

Both these modes, FTZ and DAZ, are only relevant to very tiny numbers, well below the resolution that most applications need to worry about, so you can totally enable them, provided we can figure out how to do so. CPUs with support for the FTZ and DAZ modes include x86 CPUs and ARM Cortex cores, and likely other processors. Google TPU doesn't support FTZ/DAZ because it operates on `bfloat16` floating-point numbers.

Enabling FTZ and DAZ. Finding details on how to enable FTZ and DAZ is quite hard! For command-line options, it seems to be "`-ftz`" on Linux/Mac or "`/Qftz`" on Windows. To control these modes dynamically in C++ code, you need to modify the MXCSR x86-64 CPU control register at runtime to set (or clear) the bits corresponding to FTZ and DAZ. Some of the primitives available to do so via GCC intrinsics include:

- `__builtin_ia32_ldmxcsr`
- `__builtin_ia32_stmxcsr`
- `_mm_getcsr`
- `_mm_setcsr`

In MSVS, there are preprocessor macros for FTZ in <xmmintrin.h> and for DAZ in <pmmintrin.h> header files. These control the FTZ and DAZ bits in the MXCSR, which is a CPU register with flags to control the CPU and the FPU. The C++ snippet to enable these modes looks like:

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

void aussie_float_enable_FTZ_DAZ(bool ftz, bool daz)
{
  if (ftz) {    // FTZ mode
    _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);
  }
  else {
    _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_OFF);
  }

  if (daz) {    // DAZ mode
    _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
  }
  else {
    _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_OFF);
  }
}
```

These intrinsics for FTZ and DAZ are dynamic C++ calls. You can also disable these modes in C++, or switch back-and-forth between them dynamically. The MXCSR values are per-thread, so these modes must be set at the start of every new thread.

Negative Zero

Floating-point representations have two zeros: positive zero (the usual "`0.0f`" one) and negative zero ("`-0.0f`"). Note that there's no negative zero in integers, but only in floating-point types, because integers use two's complement in C++.

Usually, you don't have to worry about negative zero float values, because all of the floating-point operations treat zero and negative zero as equal. Negative zero is not less than positive zero, but is equal instead. For example, the "==" and "`!=`" operators should correctly handle both zeros as the same, and testing "`f==0.0f`" will succeed for zero and negative zero.

Normal C++ operations on `float` types will automatically handle negative zero for you, such as "<" will treat the two zeros are equal, not less-than. This happens at the cost of some inefficiency.

Detecting Negative Zero. Testing for negative zero is not easy. Unfortunately, you cannot use the `std::fpclassify` function because it returns `FP_ZERO` for both positive and negative zero. Here are some fast macros for 32-bit floats that look at the bits by pretending it's an `unsigned` 32-bit integer:

```
#define AUSSIE_FLOAT_TO_UINT(f)  (*(unsigned int*)&f)
#define AUSSIE_FLOAT_IS_POSITIVE_ZERO(f) \
    (((AUSSIE_FLOAT_TO_UINT(f) )) == 0)  // All 0s
#define AUSSIE_FLOAT_IS_NEGATIVE_ZERO(f)  \
    (((AUSSIE_FLOAT_TO_UINT(f) )) == (1u<<31)) // Sign bit
```

Note that these macros only work for `float` variables, not constants, because the address-of "`&`" operator gets a compilation error for floating-point constants (e.g., `0.0f` or `-0.0f`). Also, these only work for 32-bit `float` types, and comparable macros are needed for 64-bit `double` or 128-bit `long double` types.

Pitfall: Bitwise tricks on negative zero. There are some pitfalls with negative zero if you are trying to subvert the normal floating-point number representations and do bitwise operations on them (as I just did above!).

For example, if you're doing bitwise tests on a `float`, you may still need to test for two values of zero, such as using one or both of the above zero testing macros.

For magnitude comparisons of `float` types via their underlying bits, there's also a problem. Whereas positive zero is all-bits-zero and will equal integer zero or unsigned integer zero, negative zero has the uppermost bit set (the sign bit), so it will be a negative integer or a very large unsigned number. Hence, negative zero will sort as less than positive zero if using signed integer tests, or will sort as massively greater than many numbers if using unsigned integers for testing.

The problem with negative zero also means that doing any bitwise comparisons will fail. You cannot just compare the underlying integers for equality against each other, nor can you use byte-wise testing.

For example, using `memcmp` for equality testing a `float` vector will occasionally fail for `float` values where positive zero compares against negative zero, leading to insidious bugs.

Optimization by Suppressing Negative Zero. Since negative zero introduces an inefficiency into basic `float` operations (e.g., `==` or `!=` with `0.0`), can we block it for a speedup? Are there any settings that fix the CPU or the compiler to ignore negative zero?

The FTZ and DAZ modes are mainly for subnormal numbers, not negative zero. I'm not aware of any hardware CPU modes specifically for disallowing skipping negative zeros, and I wonder whether they would actually be a de-optimization anyway, by forcing the FPU to explicitly check for negative zeros. Apparently, FTZ might help avoid negative zero in computations, but I'm not sure it's 100% of cases. There is a GCC flag "`-ffast-math`" which disables the production of negative zero in software.

Negative Zero. Can we speed up the floating-point computations of our code by blocking all floating-point negative zeros? Then the FPU or GPU can assume there's only one type of zero, and run faster. We could either run in a negative-zero-disabled mode, or use our own bitwise test for floating point zero as all-bits-zero (i.e., using the unsigned integer trick).

What about zero values at runtime? Can we guarantee that it never contains a negative zero, and thereby speed up analysis?

Getting to the Bits in C++

The basic 32-bit floating-point number in C++ is a `float` with a size of 4 bytes. How can you manipulate the bits in a floating-point value, using the 32-bit `float` type? You cannot use any of the C++ bitwise operators on floating-point numbers, as they only work for integers.

The trick is to convert it to an unsigned integer (32-bit) with the same bits, and then use the integer bitwise operations. The obvious way to convert a `float` to `unsigned` is casting:

```
float f = 3.14f;
unsigned int u = (unsigned)f;  // Fail!
```

Nope. That doesn't get to the bits, because it does a proper conversion between floating-point numbers and integers, which is usually what you want when you aren't thinking about bits (i.e., all normal people).

To get to the bits in C++, we have to trick the compiler into thinking that it's already got an unsigned integer with pointer type casts:

```
unsigned int u = *(unsigned int*)(&f);  // Tricky!
```

That's a bit old-school for type casting. Here's using `reinterpret_cast`:

```
unsigned int u = *reinterpret_cast<unsigned int*>(&f);
```

Once we have the bits, then we can twiddle the bits of our unsigned integer to our heart's content. When we're finished, we can do the same trick in reverse to re-create a floating-point number:

```
f = *(float *)(&u);   // Floating again...
f = *reinterpret_cast<float*> (&u);  // Trendy version
```

And here's a timely reminder that it's important to use an "`unsigned`" type in C++ for the bit faking code, because the ">>" right-shift operator has undefined behavior on negatives.

Other Methods: Type casts aren't the only way in C++. There's also a trick involving "`union`" structures, and you can also directly copy the bits to a differently typed variable using "`memcpy`" or "`bcopy`".

It seems to me that this type cast trick should be the fastest way, because a good compiler should convert the address-of, `reinterpret_cast` and indirection sequence into a simple variable copy, especially with the "`reinterpret_cast`" hint. However, I haven't actually benchmarked the speed of the different methods.

Pitfalls and Portability

Bitwise manipulation of float data is not the most portable code in the world. Let's examine some of the possible pitfalls in using these techniques.

Bitwise zero testing: If you've gone to the trouble to access the bits of a floating-point number, you might as well use them. Obviously, testing for "`0.0`" is a common requirement, so let's make it faster:

```
#define FLOAT_IS_ZERO(f) \
 ((*reinterpret_cast<unsigned int*>(&f)) == 0u) // Bug!
```

Oops! We forgot about negative zero. There are two zeros in floating-point, depending on the sign bit, and it's hard to test it efficiently with bitwise operations (e.g., mask the sign bit or shift left first).

Strict anti-aliasing rule. An important point about all this is that most of it is platform-dependent, and officially "undefined behavior". Some of it is standardized by IEEE 754, but many variations are possible.

Another issue is that there's a "*strict anti-aliasing rule*" that specifies that many of these tricks are officially non-standard methods. Accessing a floating-point number as if it's an unsigned number is a technical violation of this rule. The "`reinterpret_cast`" method is probably less likely to run afoul of this problem, but it's still not guaranteed.

Anyway, the `union` trick and the use of `memcpy` don't really strike me as being particularly more portable, although `memcpy` might be less likely to be optimized wrongly by a compiler making wrong assumptions. Some additional risk mitigations are warranted, such as adding a lot of unit tests of even the most basic arithmetic operations. However, you're still not officially covered against an over-zealous optimizer that might rely on there being no aliases allowed.

Byte sizes. Another much simpler portability issue is checking the byte sizes of data types, which can vary across platforms. Most of this bit-fiddling stuff relies on particular 16-bit and 32-bit layouts. It doesn't hurt to add some self-tests to your code so you don't get bitten on a different platform, or even by a different set of compiler options:

```
aussie_assert(sizeof(int) == 4);
aussie_assert(sizeof(short int) == 2);
aussie_assert(sizeof(float) == 4);
aussie_assert(sizeof(unsigned int) == 4);
```

Also note that for this to work well, both types must be the same size. So, this would be a useful code portability check if it worked:

```
#if sizeof(float) != sizeof(unsigned int)  // Fails!
#error Big blue bug
#endif
```

This macro preprocessor trick doesn't work because `sizeof` isn't allowed in a preprocessor expression, because the preprocessing phase precedes the syntax analysis. A better version uses a "`static_assert`" statement, which does compile-time checking in a more powerful way.

```
static_assert(sizeof(float) == sizeof(unsigned), "Bug!");
```

Floating-Point Builtin Functions

The alternative to directly accessing the bits as an unsigned integer is to use the existing C++ functions. There are various existing functions for bitwise manipulation of floating-point numbers, in two categories: standard C++ library functions and compiler-specific intrinsics.

C++ has standard functions for the manipulation of floating-point numbers, and their bitwise representations.

- `std::signbit` — Portably test the sign bit of a floating-point number.
- `std::copysign` — Portably copies the sign bit from one `float`, merging it with the value of another (i.e., another's exponent and mantissa).

There are also various compiler-specific "intrinsics" or "builtins" to manipulate floating-point numbers. For Microsoft Visual Studio C++, these are in `<intrin.h>` and there are also versions for GCC and other compilers.

- `frexp` — Get the mantissa and exponent.
- `ldexp` — Bitshifting by an integer shift-count.
- `scalbn` — Also integer bitshift on a `float`.
- `logb` — Extracts the exponent.
- `ilogb` — Extracts the exponent to integer.
- `modf` — Splits into whole and fractional parts.
- `fma` — Fused multiply add on `float` (Microsoft intrinsic)
- `remainder` — Get fractional part of floating-point (Microsoft intrinsic)
- `_fcvt` — Low-level convert `float` to string (Microsoft intrinsic)

For many of the listed functions, there are additional versions for different floating-point data types, such as `float`, `double` and `long double`. For example, "`frexp`" will split a `double` type into its significand (fractional part) and exponent integer, but there's also "`frexpf`" for 32-bit `float` types, and "`frexpl`" for long double types.

Floating-Point Bit Tricks for AI

Once you've got the bits into an unsigned integer, what can you do?

Assuming you're willing to throw the standards documents to the curb, you can do quite a lot. The bits can be directly manipulated in non-obvious ways to speed up some types of floating-point arithmetic with bitwise arithmetic on the bits.

Examples of floating-point bit manipulations used to optimize neural networks include:

- Sign bit flipping: this can be used for fast non-multiplication binarized networks with floating-point computations.
- Exponent bit manipulations: bitshifting `float` values in logarithmic quantization can be implemented as integer addition on the exponent bits of a float.
- Add-as-integer networks: This method simply adds the underlying bit representations together as integers, to create a type of multiplication-free neural network. Weirdly, this simple trick implements an approximate multiplication algorithm known as Mitchell's algorithm.
- Fast `log2` computation on `float` types using the exponent bits directly.

The first step is to extract the bit patterns. Let's assume it's a standard 32-bit float type with 1 sign bit, 8 exponent bits, and 23 stored mantissa bits. You can get the different bits:

```
int signbit = (u >> 31);
int exponent = ( (u >> 23) & 255 );  // Fail!
int mantissa = ( u & ((1 << 23) - 1 ));
```

Nice try, but that's only 2 out of 3. The exponent is wrong here! The bits are correct, but it's not the right number. We have to subtract the "offset" (or "bias") of the exponent, which is 127 for an 8-bit exponent. This is correct:

```
int exponent = ( (u >> 23) & 255 ) - 127;  // Correct!
```

Note that the sign bit and mantissa can be stored as `unsigned` (i.e., positive or zero), but the exponent must be a signed integer, even though it is extracted from the bits of an unsigned int. For a fraction like decimal `0.25` (i.e., a quarter), this is equal to `2^-2`, so the exponent is `-2`. In an 8-bit exponent, the range of the exponent is `-128` to `+127`. Note that the sign bit in a `float` specifies the overall sign of the whole number, and is not the sign of the exponent.

Here are some macro versions of the above bit extractions:

```
#define AUSSIE_FLOAT_SIGN(f)      \
  ((*(unsigned *)&(f)) >> 31u)    // Leftmost bit
#define AUSSIE_FLOAT_EXPONENT(f) \
  ((int)(((((*(unsigned*)&(f)))>> 23u) & 255) - 127))
#define AUSSIE_FLOAT_MANTISSA(f) \
  ((*(unsigned*)&(f)) & 0x007fffffu) // Right 23 bits
```

Note that these macros don't work for constants, but give a compilation error such as "l-value required". This is because of the "&" address-of operator trick being used needs a variable, not a constant. I don't see an easy way around it for bitwise trickery.

If you dislike bits for some strange reason, here's a simple way to define the sign bit macro using the "<" operator, which also works on constants:

```
#define AUSSIE_FLOAT_SIGN(f)  ( (f) < 0.0f)  // Sign test
```

Example: Add-as-int Approximate Multiply

The add-as-integer method suggested by Mogami (2020) simply adds the integer bit representation of two floating-point variables, as if they are integers. It's quite surprising that this has any useful meaning, but it's actually a type of approximate multiplication called Mitchell's algorithm. Here's what the C++ code looks like on 32-bit `float` types:

```
float aussie_add_as_int_mogami(float f1, float f2)
{
    // Add as integer Mogami(2020)
    int c = *(int*)&(f1)+*(int*)&(f2)-0x3f800000;
    return *(float*)&c;
}
```

The magic number `0x3f800000` is (obviously) equal to "`127<<23`" and its purpose is to fix up the offset of the exponent. Otherwise, there are two offsets with 127 combined. (Is there a faster way? It's annoying to waste a whole addition operation on what's just an adjustment.)

Note that this algorithm is one exceptional case where we don't want to use `unsigned` integer types when tweaking bit representations. This trick needs the temporary variable of type "`int`" and the pointers to be "`int*`" so that it can correctly handle the sign bits of the two floating-point numbers.

This add-as-integer algorithm is not restricted to 32-bit `float` data. It should also work for 16-bit floating-point numbers in both `float16` and `bfloat16` formats, provided the magic number is changed to a different bitshift count and another offset of 15 (not 127) for 5-bit exponents.

Example: Float Bitshift via Integer Addition

This is another surprising bitwise trick on floating-point numbers. You cannot perform the standard bitshift operators on `float` types in C++, so you cannot easily speed up floating-point multiplication via bitshifts in the same way as for integers.

Bitshifts are a fast way of doing an integer multiplication by a power-of-two (e.g., "`x<<1`" is the same as "`x*2`"). Note that it also doesn't work to convert the `float` to its `unsigned int` bit version and shift it using integer bitshift operators.

On some platforms, there are some builtin special functions such as `ldexp` and `scalbn` for doing bitshifting on `float` data. The `ldexp` function accepts an integer power, and then bitshifts a floating-point number by this many places. The `ldexp` function is for `double` types, `ldexpf` is for `float`, and `ldexpl` is for `long double` types. The `scalbn` set of functions appears to be almost identical to `ldexp` functions. There is also a reverse function "`frexp`" which extracts the significant (fraction) and the power-of-two for a floating-point argument.

Although we can't bitshift floating-pointer values, there is an intriguing alternative optimization using integer arithmetic directly: *addition*. The suggestion in the DenseShift paper (Li et al., 2023) is to simply add the shift count to the exponent bits using integer addition.

Here's some example C++ code that works for 32-bit floating-point numbers:

```
float aussie_float_bitshift_add_int(float f1, int bits)
{
    // Bitshift float by adding int to exponent bits
    // FP32 = 1 sign bit, 8 exponent, 23 mantissa
    unsigned int u = *(unsigned int*)&f1; // Get the bits
    if (u == 0) return f1;  // special case, don't change
    u += (bits << 23);  // Add shift count to exponent
    return *(float*)&u; // Convert back to float
}
```

How does it work? Well, it makes a certain kind of sense. The exponent in a floating-point representation is a power-of-two, and we are bitshifting, which is increasing the number by a power-of-two. Hence, we can increase the power-of-two by adding 1 to the exponent, and it also works the same for adding numbers more than 1.

Note that this code also works for bitshift of a negative count (e.g., bitshift of -1 subtracts from the exponent and thereby halves the number) or zero (unchanged). However, this exponent-addition trick can overflow if the resulting number overflows or underflows the exponent range (e.g., `-128` to `+127`).

This method has thereby improved the performance of floating-point multiplication by changing it to integer addition. The idea works provided we are multiplying by a power-of-two, which is done in logarithmic quantization. However, it's a little tricky in that special formats like zero (and `NaN`) are problematic for this algorithm. I had to add the test "`u==0`" which slows things down (maybe there's a better way?). Also, this approach can theoretically overflow the exponent bits, messing up the sign bit, but that's only if the `float` is very big or very tiny. Checking for all these wrinkles will slow down the code.

Example: Log2 of Floating-Point is the Exponent

The `log2` function for float types is a non-linear function that is quite expensive to compute. We already computed `log2` of an integer with low-level bit fiddling methods based on a count-leading-zeros algorithm in the bitwise operations chapter. There's also a different bitwise trick for `log2` of floating-point numbers. This method computes the truncated integer version of the `log2` algorithm (e.g., for use in logarithmic power-of-two quantization). There's a very easy way:

The base-2 logarithm is the exponent!

It's sitting right there, already calculated, hidden in plain sight amongst the 32 bits of your friendly `float` variables. Here's some C++ code to extract it:

```
int ilog2_exponent(float f)  // Log2 for 32-bit float
{
    unsigned int u = *(unsigned int*)&f;
    int iexp = ((u >> 23) & 255);  // 8-bit exponent
    iexp -= 127;  // Remove the "offset"
    return iexp;
}
```

Alternatively, for greater portability and probably extra speed, too, there are some standardized builtin C++ functions available across various platforms (including Linux and Microsoft) that can extract the exponent: `frexp`, `ldexp`, `ilogb`, and `scalbn`, are some that come to mind.

References on Floating-Point

1. Eric Sakk (2018), *Understanding Floating-Point Numbers*, Concepts in Computer Systems (Volume 2), 7 June 2018, https://www.amazon.com/dp/1983093025/
2. Sean Eron Anderson (2005), *Bit Twiddling Hacks*, Stanford University, https://graphics.stanford.edu/~seander/bithacks.html
3. T. Mogami (2020), *Deep neural network training without multiplications*, In Beyond BackPropagation WS at 34th Conference on Neural Information Processing Systems, 2020, https://arxiv.org/abs/2012.03458 (Uses integer addition of the bits of an IEEE 754 floating-point representation to perform approximate floating-point multiplication.)
4. Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lerevre, Guillaume Melquiond, Nathalie Revol, Damien Stehle, Serge Tones (2018), *Handbook of Floating-Point Arithmetic*, Birkhauser, 2018, https://link.springer.com/book/10.1007/978-3-319-76526-6, Contents: https://cds.cern.ch/record/1315760/files/9780817647049_TOC.pdf
5. Wonyeol Lee, Rahul Sharma, Alex Aiken (2016), *Verifying Bit-Manipulations of Floating-Point*, Stanford University, USA, https://theory.stanford.edu/~aiken/publications/papers/pldi16b.pdf
6. Xinlin Li, Bang Liu, Rui Heng Yang, Vanessa Courville, Chao Xing, Vahid Partovi Nia (2023), *DenseShift: Towards Accurate and Efficient Low-Bit Power-of-Two Quantization*, Oct 2023, https://arxiv.org/abs/2208.09708 (Uses integer addition on the sign and exponent bits of IEEE 754 floating-point to perform bitshifts on floats to perform power-of-two number quantization on 32-bit floats.)
7. Mostafa Elhoushi, Zihao Chen, Farhan Shafiq, Ye Henry Tian, Joey Yiwei Li (2021), *DeepShift: Towards Multiplication-Less Neural Networks*, July 2021, https://arxiv.org/abs/1905.13298 (Bitwise shifting and sign bit manipulation.)

15. Arithmetic Optimizations

Types of Arithmetic Optimizations

There are two basic ways that arithmetic computations can be sped up whilst retaining the same results:

- Single operator improvements
- Expression-level optimizations (multiple operators)

As an example of single operator optimizations, consider replacing the multiplication operator. Alternative forms of arithmetic include bitwise shifting or addition. The ways to do fewer multiplications tend to involve higher-level algorithmic changes to the model, such as pruning or quantization.

Some of the methods of speeding up arithmetic come from the theory of compiler optimization (e.g., strength reduction, sub-expression elimination). Hence, the compiler will often automatically perform these types of optimizations (when the optimizer is invoked). To some extent, this makes these transformations redundant.

Even so, good programming practice is to avoid situations where these optimizations are needed on a large scale. The compiler does not look at the program as a whole and can miss some "obvious" optimizations.

Operator Strength Reduction

Individual operations in C++ can be optimized in several ways. The general term is "strength reduction" because a stronger operator with high computation complexity is "reduced" to an equivalent operator that is simpler and faster.

Strength reduction is a technique used in automatic optimization by compilers, but can also be used by programmers to improve algorithms.

The main "strong" operations that we're trying to avoid are:

- Floating-point arithmetic (even addition)
- Multiplication
- Division
- Remainder (`%` operator)
- Math functions (e.g., `sqrtf` or `expf`)

Strength reduction has particular relevance to AI engines because the main bottleneck is floating-point multiplication. Many of the research papers on speedups are about replacing the floating-point multiplication operation with something simpler, like addition or integer arithmetic.

Some of the general approaches in regard to strength reduction include:

- Bitwise operations (e.g., bitshifts can replace multiplication)
- Multiplication is slower than addition.
- Avoid division and modulo/remainder operators (they're the worst!)
- Use integer arithmetic rather than floating-point (where possible)
- Use `float` single-precision arithmetic, not double-precision.
- Approximate arithmetic (e.g., for math functions)

Bitshift for multiplication: The canonical example that everybody knows is that shift operators can replace multiplications by a power of two. But it's only for integers, not for floating-point numbers. Here's a dummy example of integer multiplication;

```
y = x * 4;
```

This can be more efficiently coded as a left bitshift:

```
y = x << 2;
```

Bug alert! If you're making this code change, you're likely to introduce some bugs. The "<<" and "*" operators have different precedence levels, so make sure you add more parentheses. Also, consider whether you need to use "`unsigned`" type when switching to a bitwise operator.

Right shift for division: bitshifting works for division, too (but only for unsigned):

```
y = x / 4;
y = x >> 2u;  // faster
```

Bitwise remainder calculations: The arithmetic modulus operator (remainder) can also be optimized for power-of-two operands (but only on integers):

```
y = x % 512;     // Remainder (mod)
y = x & 511u;    // Bitwise-AND
```

And here's another one with integer relative comparisons versus bitwise-and, although this one might not necessarily be faster:

```
if (x >= 512)
if (x & ~511u)  // Bitwise-AND of the complement (unsigned)
```

Avoiding multiplication: There are some simple cases even with the most basic operators that have multiple options:

```
y = x * 2;
y = x + x;    // Addition
y = x << 1;   // Shift
```

Automatic Strength Reduction: In theory, C++ compilers could know what will be faster on its platform, and perform all these optimizations automatically when compiling the program. The optimizers probably do some of them, but they cannot do them all.

Intrinsic Functions: Other more advanced types of strength reduction involve avoiding costly primitives, such as mathematical functions. For example, there are bitwise arithmetic tricks to quickly compute the integer `log2` function.

GPU Strength Reduction: One final note is that when doing AI coding work, we aren't as concerned about which C++ operator works the best. The more important concern is which operation is most efficient in the GPU or other non-GPU hardware acceleration (e.g., AVX-512 on CPU).

Finally, note that these optimizations are local optimizations, and the same ideas apply globally to the entire AI engine architecture. There's been a lot of research trying to change *all* of the arithmetic in model inference from multiplication to bitshifting, such as using addition or bitshifts.

Avoid % Remainder Operations

One common use of the remainder operator is the use of modulo arithmetic, such as the wraparound array implementation of a queue abstract data type, where the value of a variable is cyclically counted from `0` up to `N-1`, and then back to 0. The most common idiom for coding this is:

```
x = (x + 1) % N;
```

However, the `%` operator is expensive, and in this case it is not really needed. The following code sequence performs the same task more efficiently:

```
if (x == N - 1)
    x = 0;
else
    x++;
```

This can also be written more concisely, but not necessarily more efficiently, as an expression with the "`?:`" ternary operator:

```
(x == N - 1) ? (x = 0) : (x++);
```

Another example of a clever avoidance of `%` is when the operand is similar to the usual byte or word size. For example, consider this remainder:

```
x % 256
```

This can be more efficiently coded with bitwise-and using:

```
x & 255
```

But this can be even more efficiently coded as a type cast:

```
(unsigned char) x
```

The conversion to this "`unsigned char`" type will be efficiently implemented by grabbing a byte out of a word. Unfortunately, this method is not portable to all obscure systems, as it relies on an "overflow" being handled harmlessly, and on "`unsigned char`" always containing 8 bits.

Reciprocal Multiplication

Division is a slow operation, whether in a CPU or a GPU. Multiplication is often significantly faster than division, and in some cases a division can be replaced by a multiplication using the reciprocal. A case in point is floating-point division by a constant. For example, consider the division:

```
f = g / 100.0;
```

This can be replaced by the multiplication:

```
f = g * 0.01;  // Reciprocal
```

If the divisor is a symbolic constant, it is possible to replace the symbolic constant with a hard-coded constant (or another symbolic constant). However, it is more convenient to replace the constant with an explicit reciprocal calculation. For example, consider the code:

```
f = g / DIVISOR;
```

This can be rewritten as:

```
f = g * (1.0 / DIVISOR);
```

The compiler should calculate the reciprocal using "constant folding" at compile-time. Note that the brackets around the division expression are probably not strictly necessary because optimizers know about associativity, but are certainly helpful to make life easier for the optimizer (these poor critters need a break every so often).

If the divisor is a complex expression, the compiler might not automate the use with a reciprocal. Here's the slow version of division by a scale factor:

```
v[i] /= sqrtf(3.14159f);
```

Here's the faster way using the reciprocal of the constant:

```
v[i] *= 1.0f / sqrtf(3.14159f);
```

We should pre-calculate this constant as a `static` variable:

```
static const float scalefactor = 1.0f / sqrtf(3.14159f);
v[i] *= scalefactor;
```

Integer Arithmetic

Real arithmetic is slow compared to integer arithmetic. Hence, it is favorable to replace real arithmetic by equivalent integer arithmetic. Real arithmetic can be replaced by integer arithmetic when only limited precision is required (e.g., 1-3 decimal places).

To do this, work in integer units that are 10, 100 or 1000 times larger (for 1, 2 and 3 decimal places) so that the decimal places appear as the lower digits of the integers.

To convert the integer into its true integer and fractional parts is quite simple. To get at the fractional part, calculate the number modulo 10, 100 or 1000 (using the `%` operator). To get the true integer part, divide by 10 or 100 or 1000 — remember that integer division truncates the fractional part.

A good example is: when working in dollars and cents, do all calculations in terms of cents (an integer). Then when printing it out, convert to dollars and cents using:

```
cents = value % 100;
dollars = value / 100;
```

However, note that this is now using two of the worst integer operators: remainder and division. The hierarchy of cost for integer operations is similar to floating-point: integer addition and subtraction are faster than multiplication, but division is still the worst, even for integers.

There appears little to be done to replace integer division with multiplication. Multiplying by the reciprocal will change an integer operation to a floating-point operation and will probably increase execution time. A power-of-two integer division could be done via the ">>" right bitshift operator, provided that it cannot be negative and uses an `unsigned` type.

Expression Transformations

Arithmetic improvements on an expression with multiple operations include:

- Constant folding (compile-time precomputation of constant expressions)
- Common subexpression elimination (only computing things once in expressions)
- Algebraic identities in computations
- Type consistency (avoid conversions)

Common Subexpression Elimination

Common subexpression elimination (CSE) is avoiding the recomputation of the same expression twice. There are many cases where the same computation appears multiple times in a single expression, or across the control flow of a program. Compiler optimizers attempt to automatically detect such cases and reuse the first computation.

In a complicated expression, there are often repeated sub-expressions. These are inefficient as they require the computer to calculate the same value twice or more. To save time, calculate the sub-expression first and store it in a temporary variable. Then replace the sub-expression with the temporary variable. For example:

```
x = (i * i) + (i * i);
```

With a temporary variable, this becomes:

```
temp = i * i;
x = temp + temp;
```

Note that this attempt to be concise is incorrect:

```
x = (temp = i * i) + temp; // Bug
```

This may fail because of its reliance on the order of evaluation of the + operator. It is not actually guaranteed in C++ that the + operator is evaluated left-to-right.

Common sub-expressions do not occur only in single expressions. It often happens that a program computes the same thing in subsequent statements. For example, consider the code sequence:

```
if (x > y && x > 10) {
    // ...
}
if (x > y && y > 10) {
    // ...
}
```

The Boolean condition "x>y" need be calculated only once:

```
temp = (x > y);
if (temp && x>10) {
    // ...
}
if (temp && y>10) {
    // ...
}
```

Algebraic Identities

The calculations in some complicated expressions can be reduced by transforming the expression into another equivalent form. The aim when using algebraic identities is to group the operations differently, to reduce the total number of arithmetic operations. Care must be taken to ensure that the new expression has equivalent meaning. For example, the short-circuiting of the logical operators can cause differences. Some useful algebraic identities are:

```
2 * x == x + x == x << 1
a * x + a * y == a * (x + y)
-x + -y == -(x + y)
```

There are also Boolean algebraic identities that can be used to perform fewer logical operations:

```
(a && b) || (a && c) == a && (b || c)
(a || b) && (a || c) == a || (b && c)
!a && !b == !(a || b)
!a || !b == !(a && b)
```

Float Type Conversions

Hidden unnecessary C++ type conversions are a common source of extra inefficiency. The main type in a Transformer is usually "`float`" (32-bit), rather than "`double`" (64-bit). Avoid unnecessary type conversion code in two ways:

- Don't mix float and double
- Don't mix float and int

The use of `float` and `int` tends to be something professional C++ programmers are aware of, after having been burned a few times, and doesn't occur that often by accident.

However, inadvertently mixing `float` and `double` is difficult to avoid, and sneaks into your code all the time. For example, here's some C++ code that looks perfectly correct:

```
float scalefactor = sqrt(2.0) * 3.14159;
```

You know this isn't AI code because it doesn't have 27 decimal places for pi, which we've memorized by rote. AI engines don't really need anywhere near that much precision, but it looks good for the boss.

The above code is also a small slug, because it may be unnecessarily using "`double`" size arithmetic, although the compiler might fix it with constant folding (but emit a warning anyway). Here's the corrected code:

```
float scalefactor = sqrtf(2.0f) * 3.14159f;
```

Note that this example shows there are two places where an "f" suffix is needed to signify that `float` arithmetic is required:

- Numeric constants (i.e., "`2.0f`" specifying a 32-bit `float`, rather than "`2.0`", which is a 64-bit `double` constant).
- Standard C++ functions (i.e., "`sqrtf`" returns `float` rather than "`sqrt`" returning `double`).

Without the "`f`", the default is `double` type constants and `double` arithmetic functions. A lot of C++ compilers will warn about these type conversions losing precision, so if you aim for warning-free compilation as a quality goal, you'll also fix most of these wasteful hidden type conversions.

16. Branch Prediction

What is Branch Prediction?

Branch prediction is an optimization in the CPU whereby efficiency is improved by considering upcoming branches. The CPU in its execution tries to smartly predict which of the two paths of a branch is more likely to be taken.

Some CPUs also do "speculative execution" of the future instructions, to get ahead, which must be discarded if the "wrong" branch is actually executed by the code.

For the programmer, these branch prediction capabilities give the opportunity to further optimize your code to capitalize on the CPU's abilities. Optimization techniques for the C++ programmer include:

- Eliminating branches in the hotpath so that the code runs straight and narrow (i.e., fast!).
- Hinting to the compiler about the most likely execution path branches (e.g., `[[likely]]` and `[[unlikely]]` specifiers).
- Keep unavoidable branches in the same neighborhood (e.g., short loop bodies).

Branch prediction has a problem in HFT: the hot path is rarely executed (i.e., actually submitting a trade). All of the branch prediction logic would try to run the cold path, as it would always be predicted. But what we want is for the branch prediction logic to always choose the hot path, even though it would mostly fail to be correct.

Thus, all of HFT is at odds with a whole swathe of computing theory about branch prediction. HFT needs a "set opposite world mode" flag, but I'm yet to find one in the GCC documentation.

Types of Branches

First things: analyze your hotpath code for branching. The main types of branches in C++ code include:

- `if` statements and `if-else` statements.
- Loop conditions and loop bodies.
- Loop control statements: `break`, `continue`.
- Function calls and `return` statements.
- `switch` statements (multi-way branching).

Some of the less obvious types of branches are:

- Ternary operator (`?:`)
- Short-circuiting in the `&&` and `||` operators

There are also hidden branches in C++ code features such as:

- Virtual function calls
- Function pointers (and function names)

Branch Compiler Hints

There are several ways for the programmer to give "hints" to the compiler and its optimizer about which pathways are more likely. As always, the compiler is free to ignore hints, so you have to check in the assembly output what effect your changes have.

Some of the ways to give hints include:

- `[[likely]]` and `[[unlikely]]` path attributes (C++20).
- `likely()` and `unlikely()` condition markers (C++20)
- `noexcept` attribute (C++11)
- `[[noreturn]]` attribute (C++11)
- `[[assume(expression)]]` attribute (C++23)

GCC also has various extensions available to give hints:

- `__builtin_expect(expression, value)` (GCC extension)
- `hot` (GCC function attribute)

It's common in pre-C++20 Linux code to define your own macro versions for use with the GCC compiler:

```
#define likely(expr)   __builtin_expect((expr), 1)
#define unlikely(expr) __builtin_expect((expr), 0)
```

Branch Profiling

Branch profiling is the recording of pathway stats to analyze the most likely branches. This can also be re-used in the compiler's optimization mode, so that the optimizer can perform branch-aware optimizations. Hence, there is a two-step process whereby better branch prediction can be incorporated into your C++ executable code.

GCC has capabilities to store and use branch prediction statistics in its optimization phase. The arguments to use are:

- `-fprofile-arcs` (GCC command-line argument)
- `-fprofile-generate` (GCC command-line argument)
- `-fprofile-use` (GCC command-line argument)

Following this process will allow GCC to generate more optimal code under assumptions based on branch frequency in its seen executions. Obviously, this is an automatic method, but needs multiple steps in the build:

- Compile without branch hints
- Run the tests
- Output the branch prediction data
- Re-compile the code with branch optimizations enabled

Note that for HFT, the fully hot path (i.e., trade execution) is actually a rare branch, so this historical branch data won't be that useful. One solution is to run GCC in a test mode in which the hotpath is always dummy-executed! Other early parts of the hotpath in HFT can still benefit in both situations, such as the trading decision logic, which is always executed on incoming market data. Obviously, non-HFT applications can always benefit, as the most likely paths are also the most heavily-executed.

Branch Heuristics

In the absence of other branch prediction data, the CPU and compiler tools fall back on some heuristics. Some of the common ones include:

- The `if` code block is more likely to be executed than the `else` code block.
- Loops tend to be executed multiple times.
- Backwards branches are assumed to be loop iterations (and are preferred due to the prior assumption).

Hence, we can make heuristic recommendations for how to organize your code:

- Put common case code in the `if` block.
- Have error handling in the `else` block.
- Don't use once-only loop executions.

Branch Elimination

The simplest way to avoid branch prediction issues is to have fewer branches. There are various ways to achieve this, ranging from minor code tricks to re-writing your entire algorithm to have fewer conditional tests.

Which branches to eliminate? The worst kinds of branches that need elimination include:

- Long if-else-if sequences
- Nested if-else statements

What data is being tested by a branch condition is also critical, and some of the problematic branches are based on unpredictable conditions:

- Branches depending on user inputs
- Branches depending on random numbers
- Branches depending on system clocks

The best types of conditional tests include:

- Compile-time known tests
- Predictable conditions

The techniques available to eliminate your least favorite branches include:

- Reorganize the overall algorithm to have fewer branches.
- Defer or combine error checking for multiple errors so that there's only one error handling branch.
- Function call optimizations such as inlining and call hierarchy flattening.
- Loop conditional test reductions such as loop unrolling and iteration bounds known at compile-time.
- Branchless programming techniques and tricks to change conditional paths to arithmetic computations.

Branchless Programming Tricks

Branchless programming is a variety of coding tricks to get rid of control flow branches. The main approach is to remove conditional tests, such as `if` statements, by using a variety of arithmetic computations instead. Code that has no branches in a long block can run very fast on a CPU because of instruction prefetching.

Advantages of branchless programming:

- Avoids branch prediction issues (CPU speedup).
- Avoids warp divergence in CUDA C++ (GPU speedup).
- Job security

Possible general software engineering disadvantages of these branchless arithmetic bit tricks:

- Code complexity — isn't it a good thing?
- Unreadable code — as if we care.
- Maintainability — is someone else's problem.

Even worse, the speed benefit might be a mirage. The issues include:

- De-optimizations from too many arithmetic operators — benchmark your tricks!
- Don't underestimate the optimizer's capability on simple code (even if it's "branchy").
- Code tricks can confuse the optimizer (undermining any benefit).
- Memory access costs may dominate over branchless code.

One of the risks with branchless code is that it runs too fast, and gets blocked by memory access delays. Hence, you may need to combine branchless code sequences with software-based memory prefetch primitives, such as with GCC builtins:

- `__builtin_prefetch()`
- `_mm_prefetch()`

Branchless Coding Techniques

Now, let's look at some of the fun tricks in branchless C++ sequences. The types of methods for branchless coding of basic sequential CPU code include:

- Bit masks
- Bit arithmetic (bitshifts, bitwise AND/OR/XOR)
- Mapping Boolean flags to 0 or 1
- Mapping logical operator results to 0 or 1
- Multiplications by 0 or 1 using Booleans
- Lookup tables (maybe)
- Conditional move (CMOV) assembly statements
- Ternary operator (`?:`)

How can we do that in AVX? Note that all SIMD operations are branchless and efficient. All of the basic arithmetic should be vectorized into SIMD operations where possible. Branches can also be removed in parallel using some types of AVX instructions. The AVX equivalents of these branchless coding ideas include:

- Boolean flags — map to 0 or 0xFFFFFF (-1) in AVX.
- Bitmask tricks — permute/shuffle AVX intrinsics.
- CMOV/ternary operator — combined "cmp" and "blend" AVX primitives.

Some of the more traditional C++ optimizations techniques can also reduce branching as an extra benefit:

- Loop code hoisting of conditional tests.
- Compile-time settings and configurations.

Ternary Operator and CMOV

Using the C++ ternary operator is one way to help the compiler write branchless code. Consider the basic `if` statement:

```
if (x > y) {
    max = x;
}
else {
    max = y;
}
```

This can be more concisely written with a ternary operator:

```
max = (x > y) ? x : y;
```

The ternary operator can be implemented in the compiler backend using a CMOV (conditional move) register assignment statement. This is a branchless instruction that implements the conditional assignment very efficiently.

In theory, both pieces of code are equivalent, and the compiler really should generate identical code. In practice, the use of the ternary operator makes it easier on those poor compiler engineers, because it's 100% guaranteed that an assignment is required, whereas the `if` statement requires a significant amount of extra compile-time static analysis to deduce that both assignments are setting the same variable. The C++ compiler is more likely to emit a branchless CMOV assembly statement with a ternary operator.

Boolean Flags are 0 and 1

Another way to reduce branches is to use Boolean flags in arithmetic, using them as having the values of integer 0 and 1. Here's a simple example:

```
bool inc_flag;
int x = 0;

if (inc_flag) {
    x++;
}
```

This can be implemented in a branchless manner:

```
x += (int)inc_flag
```

Note that the type cast to `int` is not really needed, but helps with readability, and ensures you don't get compiler or static analyzer warnings.

Whether that is faster is something that needs testing because it forces an addition operator into one of the pathways that previously had none, but at least its branchless so it helps with branch prediction.

That was a simple example, but many other ideas are possible. Instead of this:

```
if (clear_flag) x = 0;
```

You can try this branchless version:

```
x *= (int)!clear_flag;
```

It's not clear that this is faster, since multiplication is an expensive operation, but a good compiler can actually notice that it's a fake multiplication over two possible values (0 and 1), and the optimizer can then use a CMOV instruction. Who's to know without checking the assembly code or running a benchmark.

Logical Operators are 0 and 1

In the same vein, the Boolean values of the `&&` and `||` operators can be treated as 0 and 1 in integer arithmetic expressions. Here's an example of the maximum computation:

```
max = (x > y) * x + (y >= x) * y;
```

Note that the optimizer can notice that a multiplication over a Boolean operand can be replaced with a CMOV, and there are two here.

Again, the ternary operator's single CMOV instruction is probably faster than this possible de-optimization, because this version has either two multiplications or two CMOV instructions.

Bitwise XOR Tricks

There's the well-known XOR trick to swap two integers without using a temporary:

```
x = x ^ y;
y = y ^ x;
x = x ^ y;
```

Don't worry; nobody understands how this works. But it uses three assignments, no temporary variable, and no branches.

Self XOR to Zero

There's also a well-known assembly language trick of zeroing a register using XOR with itself. The idea is that instead of an "`x=0`" statement, do this:

```
x ^= x;  // Self XOR
```

The result is zero, and we don't even need to initialize the variable! However, we don't usually do this in C++, but the equivalent is common in assembly listings and compiler backend implementations.

Sign Bit Extension Masks

If you're doing any arithmetic with negative values, you can use bitwise tricks by creating two masks depending on the sign bit. The idea is that the bitmask is:

- All 0's if the number is positive (or zero).
- All 1's if the number is negative.

In other words, the bitmask is 32 bits all set to the same bit value as the sign bit. The bitmask value is either 0 or 0xFFFFFFFF, which is also that artist previously known as `-1`. One way is a ternary operator:

```
unsigned int mask = (x >= 0) ? 0 : 0xFFFFFFFFu;
```

We can also generate this bitmask using right bitshift operators and sign extension:

```
unsigned int mask = x >> 31;
```

Yes, I really should portably compute the bitshift count using the pre-defined `CHAR_BIT` macro and `sizeof(int)` as nicely done in [Farrier, 2025].

Subtraction Bit Mask

Another way to get the same result is by noting the joke about `-1` being the same value. Hence, this trick with subtraction on 2's complement signed integers works:

```
unsigned int mask = (unsigned) ( (int)(x < 0) - 1 );
```

The comparison generates an integer 0 or 1, and then we subtract 1 to get either 0xFFFFFFFF or 0. Hence, we needed to reverse the comparison test to "<" instead. All of the type casts are "free" without runtime costs, and are probably not necessary because implicit conversions would work, anyway.

Example: RELU Activation Function

Let's have a go at making the RELU function branchless. RELU is an "activation function" in LLM backends, and it's quite simple:

```
if (x < 0) {
    RELU = 0;
}
else {
    RELU = x;
}
```

In other words, change negatives to zero, but leave positives unchanged. Here's the ternary version (faster):

```
RELU = (x < 0) ? 0 : x;
```

The mask-by-subtraction version combines with bitwise-and to get:

```
unsigned int mask = (x < 0) - 1;
RELU &= mask;
```

Another idea for a branchless version of a bitwise RELU is:

```
unsigned int umask = (x >> 31); // All 0's or 1's
RELU = (x | umask);
```

Actually, that's buggy, with masking the wrong way around. Here's the correction:

```
unsigned int umask = ((-x) >> 31); // All 0's or 1's
RELU = (x | umask);
```

Beware this might be a de-optimization, because the ternary version might be a single CMOV instructions, whereas this version has three operators: negative, right bitshift, and bitwise-AND.

Sign Bitshift Portability

There's a major portability problem with this code, because right bitshift on a negative signed integer is actually undefined behavior in C++. The compiler is free to shift in zero bits or to sign bit extend on the leftmost bit position, in its sole discretion. Hence, you need to check your platform to see what the >> operator does, and whether this rightshift bitmask idea will work.

Note that we cannot fix this by doing the right bitshift on an `unsigned` type, which is guaranteed to shift in a zero bit (well-defined in standard C++, but not what we want). Note also that this is only undefined for right bitshift, not for left bitshift, which is well-defined and always shifts zero bits in on the right side (again, not what we want).

Of course, you can create the sign-based bitmask more portably by avoiding the right bitshift operator, but this loses the branchless benefits:

```
unsigned int mask = (x >= 0) ? 0 : 0xFFFFFFFF;
```

That's safe and slow, and what's the point of that?

Lookup Tables

Precomputation of lookup tables is a fast way to get a double benefit of fast computation and branchless code. A good example in the standard C++ library are the functions for character types. Here's a slow branching version:

```
#define islower(c)   (((c) >= 'a') && ((c) <= 'z') )
```

This has lots of computation and also branches in the short-circuiting of `&&`.

A faster version uses a precomputed lookup table with 256 bytes.

```
#define islower(c)  _islower_table[(unsigned char)(c)]
```

This is faster and branchless, at the cost of 256 bytes of global memory, and has already been done for you in the standard libraries by those uber-brainy compiler engineers.

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. Sarah Butcher & Alex McMurray, 2 January 2025, *The C++ techniques you need for $600k hedge fund jobs*, https://www.efinancialcareers.com/news/low-latency-c
3. Paul Alexander Bilokon, Maximilian Lucuta, Erez Shermer, 27 Aug 2023, *Semi-static Conditions in Low-latency C++ for High Frequency Trading: Better than Branch Prediction Hints*, https://arxiv.org/abs/2308.14185, Code: https://github.com/maxlucuta/semi-static-conditions (Advanced branch prediction analysis, a way to do branches by self-modifying code at assembly level.)
4. John Farrier, March 2025, *Branch Prediction: The Definitive Guide for High-Performance C++*, https://johnfarrier.com/branch-prediction-the-definitive-guide-for-high-performance-c/
5. Srdjan Delić, Apr 10, 2023, *Branchless programming — Why your CPU will thank you*, https://sdremthix.medium.com/branchless-programming-why-your-cpu-will-thank-you-5f405d97b0c8
6. Jared Gorski, 11 August, 2020, *Branchless programming*, https://jaredgorski.org/notes/branchless-programming/
7. Algorithmica, March 2025 (accessed), *Branchless Programming*, https://en.algorithmica.org/hpc/pipelining/branchless/
8. Michael Kerrisk, Oct 5, 2012, *How much do __builtin_expect(), likely(), and unlikely() improve performance?* http://blog.man7.org/2012/10/how-much-do-builtinexpect-likely-and.html
9. Agner Fog, 28 May, 2024 (last update), *The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, https://www.agner.org/optimize/microarchitecture.pdf
10. GCC, March 2025 (accessed), *Common Function Attributes*, https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html
11. Algorithmica, July 2025 (accessed), *Binary Search*, https://en.algorithmica.org/hpc/data-structures/binary-search/ (Shows a branchless binary search algorithm with prefetching.)
12. Paul-Virak Khuong, Pat Morin, 15 Mar 2017 (v2), *Array Layouts for Comparison-Based Searching*, https://arxiv.org/abs/1509.05053 (Branchless and cached versions of binary search on sorted arrays.)
13. Agner Fog, 22 June 2024 (last updated), *Optimizing subroutines in assembly language: An optimization guide for x86 platforms*, https://www.agner.org/optimize/optimizing_assembly.pdf

17. Instruction-Level Parallelism

What is Instruction-Level Parallelism?

Instruction-Level Parallelism (ILP) is a CPU optimization at the lowest levels in machine instruction processing. If you thought parallel programming was about multithreading, SIMD vectorization and GPU kernels, there's a whole another level deep down in the CPU.

Modern CPUs are amazingly advanced, and they have been architected to use various types of extra parallelism. Some of the types of instruction-level parallelism in a modern CPU include:

- Parallel execution units
- Pipelined execution of micro-ops
- Out-of-order execution of instructions
- Prefetching of instructions
- Branch prediction Memory data prefetching

Importantly, the CPU has total parallelism in its instruction execution units. In fact, a CPU can typically run four or more machine instructions in parallel in the same clock cycle, but using multiple execution units on different parts of the chip.

Instruction Reordering Optimizations

Instruction reordering is a type of Instruction-Level Parallelism (ILP), and is an optimization performed inside the CPU where it actually runs the machine code instructions out-of-order. The way this works in simple terms is:

- Delay any opcodes that don't have the data they need (e.g., from memory).
- Run any instructions that are ready as soon as possible.

There's a whole smash of fun to be had researching how this all works in the CPU. There are schedulers and "stations" and various queues and caches. Kudos to all those hardware engineers.

Another special type of fun is for compiler engineers. GCC does a lot of fancy optimizations in the code generation backend in terms of taking advantage of instruction orders.

But what about C++? Is there anything you can do in C++ to optimize your code? Or with inline assembly instructions?

Safety first. Most of the discussion of out-of-order execution and C++ occurs in relation to safety. Problems can arise across multiple threads if the reads and writes from our C++ statements are running out-of-order. I mean, how can it be good to just run my C++ code in any random order that the CPU chooses?

The issue of preventing out-of-order errors involves "memory order." These are especially useful for correctly implementing lock-free algorithms with atomics, but they also act as memory barriers that can prevent any undesirable types of out-of-order execution.

Speed second. But the goal is to go faster! Rather than stopping the CPU from reordering instructions by using memory barriers, let's maximize it! There are at least two major ideas:

- Minimize memory-waiting delays
- Exploit out-of-order instructions

The first point is to minimize the slowdowns whereby instructions get delayed. The main one is memory accesses, which has well-known solutions such as: cache hit maximization, cache lines, tiled memory accessing, contiguous memory blocks, reducing data sizes, etc.

Other than cache locality, there's not a lot of discussion anywhere in books or on the internet about exploiting out-of-order instruction execution to make code run faster. But there's some discussion of this in Agner Fog's astounding CPU resources; see (Fog, 2024). The key point is:

> *Free extra parallelism!*

The average CPU has hidden parallelism in terms of its various computation pathways.

For example, the CPU can run these two computations in parallel:

- Integer arithmetic — Arithmetic-Logic Unit (ALU)
- Floating-point arithmetic — Floating-Point Unit (FPU)

That's not the full list. Modern CPUs now have more than one ALU, so they can perform two or more integer additions or comparisons in parallel. Some CPUs can also run different types of integer arithmetic, such as addition and multiplication, on separate pathways. Similarly, some of the SIMD operations run separately from the non-SIMD instructions.

Out-of-Order Execution Optimizations

So, you can see the opportunity here, right? Not only can the CPU run the same operations in parallel via SIMD instructions, but it can run two (or more!) different types of computations in parallel.

Unfortunately, the opportunities for huge improvements to your C++ are somewhat limited. For example, if you have a computation with both integer and floating-point computations, can you parallelize them? Yes, but only in limited circumstances, where:

- The two computations don't depend on the results of the other.
- Not requiring memory accesses for the computations.
- Computation operands are values already in CPU registers.

If there's a dependency, they can't run in parallel. And if they both require memory requests, that's the bottleneck regardless of whether the instructions can run in parallel. The data needs to be already loaded from memory into CPU registers to run fast.

That's quite a list of limitations, but it's not insurmountable. The optimization methods include:

- Prefetching the memory (e.g., `__builtin_prefch()` with GCC).
- Removing "dependency chains" from the sequence of arithmetic data instructions.

One common way to remove data dependencies is to use multiple separate variables for intermediate results.

Multiple Accumulator Optimizations

A simple example of using parallel arithmetic computations in a CPU is using multiple accumulator variables for vector dot product. Here's an unrolled version of the dot product:

```
float vector_dot_product_unroll2_ILP(
    const float v1[], const float v2[], int n)
{
    float sum = 0.0f;
    for (int i = 0; i < n; i += 2) {
        sum += v1[i] * v2[i];
        sum += v1[i+1] * v2[i+1];
    }
    return sum;
}
```

The problem is there's a data dependency between the two additions. The two multiplications can run in parallel, if the CPU can do so, but the second "`sum+=`" operation must await the completion of the first one. The solution that increases the opportunity for CPU instruction-level parallelism is:

Multiple separate accumulators!

Hence, the code becomes:

```
float vector_dot_product_unroll2(
    const float v1[], const float v2[], int n)
{
    float sum = 0.0f, sum2 = 0.0f; // Two accumulators!
    for (int i = 0; i < n; i += 2) {
        sum += v1[i] * v2[i];
        sum2 += v1[i+1] * v2[i+1];
    }
    return sum + sum2; // Add the accumulators
}
```

This new version now allows the compiler to use out-of-order execution or other instruction-level parallelism optimizations, because the two "+=" operations are now independent inside the loop body.

This function also needs other optimizations applied to it, which are orthogonal to this idea of breaking data dependency chains, such as marking the pointers are "restricted" and using AVX SIMD vectorized instructions.

Double Loop Unrolling

The idea with double loop unrolling is to do two levels of unrolled vectorization:

- AVX SIMD instructions — vectorize 4 or 8 numbers.
- Unrolled again — multiple SIMD AVX instructions per loop body.

Here is the first level of loop unrolling by a factor of 8 using an AVX-2 method with 256-bits SIMD instructions:

```
float aussie_vecdot_FMA_unroll_AVX2(
        float v1[], float v2[], int n)
{   // AVX2 vecdot using FMA (Fused Multiply-Add) primitives
    aussie_assert(n % 8 == 0);
    __m256 sumdst = _mm256_setzero_ps(); // Set accums zero
    for (int i = 0; i < n; i += 8) {
        // AVX2: process 8x32-bit floats in 256 bits
        __m256 r1 = _mm256_loadu_ps(&v1[i]); // Load 256-bit
        __m256 r2 = _mm256_loadu_ps(&v2[i]);
        sumdst = _mm256_fmadd_ps(r1,r2,sumdst); // FMA 3 vec
    }

    // 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;
}
```

But then we can unroll the loop into 2 (or more) of these SIMD instructions to process 16 numbers at a time:

```
float aussie_vecdot_FMA_double_unroll_AVX2(
             float v1[], float v2[], int n)
{   // Double-unrolled AVX2 vecdot using FMA
    __m256 sumdst = _mm256_setzero_ps(); // Set accums zero
    __m256 sumdst2 = _mm256_setzero_ps(); // Parallel accum!
    for (int i = 0; i < n; i += 16) {
        // AVX2: process 8x32-bit floats in 256 bits
        __m256 r1 = _mm256_loadu_ps(&v1[i]); // Load 256-bit
        __m256 r2 = _mm256_loadu_ps(&v2[i]);
        sumdst = _mm256_fmadd_ps(r1,r2,sumdst); // FMA 3 vec
        // Double unrolled!
        __m256 r3 = _mm256_loadu_ps(&v1[i+8]);
        __m256 r4 = _mm256_loadu_ps(&v2[i+8]);
        // 2nd FMA of 3 vectors
        sumdst2 = _mm256_fmadd_ps(r3, r4, sumdst2);
    }
```

```
    // Add the final 8 accumulators manually
    float* farr = (float*)&sumdst;
    float* farr2 = (float*)&sumdst2;
    float sum = farr[0] + farr[1] + farr[2] + farr[3]
              + farr[4] + farr[5] + farr[6] + farr[7]
              + farr2[0] + farr2[1] + farr2[2] + farr2[3]
              + farr2[4] + farr2[5] + farr2[6] + farr2[7]
              ;
    return sum;
}
```

I mean, why bother?

It's a combination of three optimizations: (a) loop unrolling, (b) SIMD vectorization, and (c) out-of-order execution in the CPU using instruction-level parallelism. By having two sets of parallel accumulators in one loop body, `sumdst` and `sumdst2`, we have removed the data dependency between the two sets of SIMD operations. By having two sets of AVX SIMD FMA operations available in parallel, the CPU is free to do both AVX operations in parallel, or at least to pipeline them.

Yes, the above code is not yet polished. The last step of 15 additions can be optimized with horizontal-add primitives. The loop condition could perhaps be optimized with pointer arithmetic. The array parameters should be marked as restricted and `const`. And we could, should we want to, unroll the loop to more than two sets of parallel SIMD accumulators.

After we do all this, ... hopefully, it's faster!

References

1. Agner Fog, 22 June 2024 (last updated), *Optimizing subroutines in assembly language: An optimization guide for x86 platforms*, https://www.agner.org/optimize/optimizing_assembly.pdf
2. Agner Fog, 28 May, 2024 (last update), *The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, https://www.agner.org/optimize/microarchitecture.pdf
3. 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/
4. 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/
5. Katecpp, Oct 5, 2015, *Improve performance with cache prefetching*, http://katecpp.github.io/cache-prefetching/

18. Core Pinning

What is Core Pinning?

Core pinning is a multithreading optimization where a thread is "pinned" to only one of the cores to give it higher priority. This means that important thread that runs the hotpath can have guaranteed CPU availability, rather than waiting for the default thread scheduling algorithms. Hence, core pinning can be a solution to avoid lock contention worries or excessive context switch in the main hotpath thread.

Core pinning is also called "thread affinity" and has multiple other names (e.g., "processor affinity" or "CPU affinity" or "CPU pinning"), but if you hear the words "pinning" or "affinity" in relation to threads, this is it.

Pinning has other meanings in related architectures. There's a higher-level type for pinning whereby whole processes or applications are pinned to a CPU core by the operating system, rather than just a single thread, which isn't quite the same thing. Note also that CUDA C++ has another type of "pinned memory" for GPUs, but that's a memory upload optimization rather than a compute improvement.

The other side of core pinning is that you obviously don't pin the less important threads. All the lower-priority threads have fewer cores available, and are downgraded.

Pros and Cons

The use of core pinning is a very powerful type of hotpath optimization. The main pathways are super-optimized because of these factors:

- No context switches
- Fewer cache misses (no invalidated caches)
- Highest priority execution
- Guaranteed core availability (no delay)

The downsides are fairly obvious:

- That core isn't available for other work.
- Load balancing only available on the other cores.

And also, you can't do it too many times, because the CPU only has a fixed number of cores.

Counting Cores

The code to set up core pinning is really a two-part procedure with these steps:

1. Determine how many CPU cores are available.

2. Pin a thread to one of them.

There are various non-standard ways to interrogate the system for its CPU settings. The standard method is to call `hardware_concurrency()` in the standard thread library, which tells you how many physical cores are in the CPU.

```
int number_of_cores()
{
    return std::thread::hardware_concurrency();
}
```

This has been a standard method since C++11, so it should be available to you. Alternatively, non-standard methods include:

- `sysconf()` — POSIX version in `<unistd.h>` for Linux.
- `GetSystemInfo()` — Win32 API in `<windows.h>`.
- `__cpuid()` — low-level intrinsic function in `<cpuid.h>` that wraps the `CPUID` machine instruction on x86 CPUs (Intel/AMD).

All of these functions offer a whole wealth of other hardware information about the CPU, rather than just the number of cores.

Setting Up Core Pinning

There's no standard way to set up the core pinning optimization using the C++11 `std::thread` library, nor does anything appear forthcoming in C++26 for this area. However, there are longstanding platform-specific functions to do this.

Sometimes, you don't need to code up core pinning in C++, but can use OS settings or commands. On Windows, you can set up a process-level CPU pinning for an application via the GUI. On Linux, there is a "`taskset`" command that allows running a program with core pinning.

Both Windows and Linux have non-standard C++ system calls that can set up core pinning for either a process or a thread. Linux uses the "pthreads" library to do core pinning, and Windows has some Win32 features.

The sequence at a high-level looks like:

1. Get a native thread id

2. Call the platform-specific core pinning API.

To implement core pinning in C++ on Linux you need to bypass `std::thread` to get to the underlying POSIX thread id, which has type `pthread_t` as defined in `<pthread.h>`. This is required because the core pinning calls are POSIX functions on Linux.

There are at least two ways to do this:

- `pthread_self()` — POSIX call to return the id of the current thread.
- `std::thread::native_handle()` — returns the "native" thread ID of a standard C++ thread object, which is a POSIX thread id on Linux.

Once you have a valid thread id, then you can set up core pinning for that thread. The programmatic C++ APIs on Linux are:

- Pinning processes — `sched_setaffinity()`
- Pinning threads — `pthread_setaffinity_np()`

On Windows, these are the C++ APIs:

- Pinning processes — `SetProcessAffinityMask()`
- Pinning threads — `SetThreadAffinityMask()`

Now let's look at a full example on Linux.

Linux Core Pinning

Here's a native pthreads sequence to pin the current thread to a core:

```
#include <pthread.h>
#include <unistd.h>
#include <sched.h>

bool pin_me(int corenum)
{
    pthread_t tid = pthread_self(); // Get current thread id
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);         // Clear all core bit flags
    CPU_SET(corenum, &cpuset);  // Set one core bit flag
    // Pin the thread!
    int ret = pthread_setaffinity_np(tid,
                  sizeof(cpuset), &cpuset);
    return ret == 0;  // Zero return is success
}
```

Note that failures can occur when attempting to pin a thread to a core. The process needs adequate permissions to do so, and the core number needs to be valid for the given system.

This code uses "`cpu_set_t`" from `<sched.h>`, which is a bitmask (or other data structure) that represents a mask of one or more cores. There are various bit manipulation macros also defined in `<sched.h>` for use with this bitmask type:

- `CPU_ZERO()` — clears all the bits.
- `CPU_SET()` — sets one bit.
- `CPU_CLR()` — unsets one bit.
- `CPU_ISSET()` — tests one bit.
- `CPU_COUNT()` — counts how many bits are bit.

There are also some arithmetic operations on the CPU bit sets in <sched.h>:

- CPU_EQUAL() — test if two bitsets are equal.
- CPU_AND() — bitwise-and on all bits.
- CPU_OR() — bitwise-or on all bits.
- CPU_XOR() — bitwise-xor on all bits.

The CPU bitmask type cpu_set_t is not a C++ object, but a raw C-like structure, which means it can be copied or moved by bitwise copy using memcpy.

Note that pthread_setaffinity_np() can be passed a CPU set with more than one bit set, in which case the thread will be migrated to one of those cores. You can also examine the thread affinity bitmasks via"

```
pthread_getaffinity_np()
```

Isolating Linux Cores

To fully implement core pinning of a thread to a particular core on Linux, some further actions may be needed. Changes are required to Linux kernel settings to do things like:

- Isolating the core
- Disabling interrupts

Some of the Linux kernel parameters you may need to adjust include:

- nohz or nohz_full
- isolcpus
- irqaffinity
- rcu_nocbs

There is some industry wisdom to avoid core zero on Linux systems, because that's the CPU core that the kernel always tries to run system tasks on, as described in Bernhardt (2023). There's also a discussion of some odd issues with core 1 on Linux in Dawson (2023).

References

1. Machinet, March 13, 2024, *How to optimize C++ code for use in high-frequency trading algorithms?* https://www.machinet.net/tutorial-eng/optimize-cpp-code-high-frequency-trading-algorithms
2. Dung Le, Aug 13, 2020, *Optimizations for C++ multi-threaded programming*, https://medium.com/distributed-knowledge/optimizations-for-c-multi-threaded-programs-33284dee5e9c
3. Larry Jones, 27 Feb 2025, *Mastering Concurrency and Multithreading in C++: Unlock the Secrets of Expert-Level Skills*, https://www.amazon.com.au/Mastering-Concurrency-Multithreading-Secrets-Expert-Level-ebook/dp/B0DYSB519C/
4. Eli Bendersky, January 17, 2016, *C++11 threads, affinity and hyperthreading*, https://eli.thegreenplace.net/2016/c11-threads-affinity-and-hyperthreading/
5. Bytefreaks, 23 November 2016, *C/C++: Set Affinity to process thread – Example Code 3*, https://bytefreaks.net/programming-2/c/cc-set-affinity-to-process-thread-example-code
6. Mark Dawson, Jr., February 12, 2023, *My Fear of Commitment to the 1st CPU Core*, https://www.jabperf.com/my-fear-of-commitment-to-the-1st-cpu-core/ (avoiding core 1 for CPU affinity).
7. Manuel Bernhardt, 16 Nov, 2023, *On pinning and isolating CPU cores*, https://manuel.bernhardt.io/posts/2023-11-16-core-pinning/ (examines costs of arithmetic operations versus cache mispredictions and context switches).
8. Davood Ghatreh Samani, Chavit Denninnart, Josef Bacik, Mohsen Amini Salehi, 3 Jun 2020, *The Art of CPU-Pinning: Evaluating and Improving the Performance of Virtualization and Containerization Platforms*, https://arxiv.org/abs/2006.02055
9. Kernel.org, May 2025 (accessed), *The kernel's command-line parameters*, https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html

19. Cache Locality

What is Cache Locality?

Cache locality is the idea of staying "local" in our accesses to memory locations to maximize the benefits of some hardware caches in the CPU. There are two general categories of cache locality:

- Instruction cache locality — machine code instruction execution.
- Memory cache locality — data access from memory locations.

There's a lot going on in the CPU in terms of caching accesses and also prefetching possible future accesses. Cache locality is the idea of ensuring that our C++ code maximizes the value of those hardware cache optimizations.

Caching occurs primarily at a lower-level than multithreading, which means that each thread's execution can benefit from these optimizations. Most of the methods to improve cache locality are related to the general code structure, rather than specific ways to do thread synchronization or other multi-threading requirements. The general ideas include:

- Tight code blocks and loops — instruction cache locality.
- Localized and predictable memory access sequences — data cache locality.

You can do both together if you like, since they have orthogonal speedups. Easier said than done!

There are various tools you can use to examine the rates of cache hits and cache misses in the instruction or data caches. Some of the main ones include:

- perf (Linux)
- cachegrind (valgrind)
- Intel VTune
- gperftools
- uprof (AMD)
- likwid-perfctr

Depending on how you look at it, these speedups make cache locality either more or less important in multithreaded applications versus sequential code. It's more important in multithreading because we have lots of threads in different places doing different things, all of which need to have good cache locality.

Or maybe it's less important, because the CPU has to throw away all of those per-thread hardware caches at every context switch, so why bother with cache locality? I'll leave it to you to judge that.

Instruction Cache Locality

The instruction cache stores recently executed machine code instructions in a CPU hardware cache. There's also a separate mechanism of "instruction prefetching" to try to load the next instruction that will be executed. As part of this prefetching method, there's also "branch prediction" in the CPU, which attempts to predict which of two branch directions will get chosen.

To get the best out of these instruction speedups, our C++ code should generally use:

- Short and tight loops
- Fewer branches

Keeping loops short will mean that the CPU stays within the same block of code, maximizing the chances that it already has an instruction in its cache. Interestingly, this means that some common code optimizations can be bad for instruction cache locality:

- Inlining of functions
- Loop unrolling

Both of these can cut both ways, since they both reduce branches, but also lengthen code blocks. Whenever you're tempted to maximize your use of such optimizations, think about the plight of the poor instruction cache as it tries to keep up.

Branches are another separate issue from short code blocks. In fact, long code sequences of compute instructions are fine for branch prediction. To maximize the CPU's branch prediction capability, we should either have few branches, or at least have very predictable branches. At the limit, we could use branchless programming, which is a set of tricks to get rid of branches. See Chapter 4 for more on branch prediction and branchless coding methods.

Data Cache Locality

There are numerous improvements that you can make to improve cache locality for the memory access caches. And there are rather a lot of different caches for CPU memory accesses:

- L1 and L2 caches (per-thread)
- L3 cache (shared)
- TLB cache (virtual address accesses)
- NUMA multi-core caching

There are some general recommendations for the entire application, that aim to reduce memory cache misses:

- Use less memory!
- Fewer memory allocations
- Smaller data sizes

But particular algorithms can also be modified to keep nearby memory in the caches. Data structures can affect the level of cache locality, with improvements such as:

- Separate cold data from hot data — improve cache locality for hot data.
- Structure of Arrays (SoA) vs Array of Structures (AoS) — which one is best depends on the context.
- Contiguous data structures — array/vector, not linked lists or binary trees.
- Compact data structures — smaller memory sizes are easier to maintain in the cache.

The code execution of various algorithms can alter the sequence of memory accesses, and thereby maximize cache locality. Some well-known improvements include:

- Loop segmenting — process short sub-sequences of a longer array.
- Tiling algorithms — process 2D "tiles" in a matrix or multidimensional data structure (also called "blocking").

The goal of these algorithm modifications is to iterate over a small sub-section in the data, keeping cache locality during that "hot" computation, and then move on to the next part. This works particularly well with matrix multiplication, because it involves multiple computations with every element of the matrix.

There are also some dynamic approaches whereby you can manually ensure that data is already in the cache when you need it:

- Memory prefetching
- Cache warming

See Chapter 20 for more about prefetching and cache warming.

Memory Hierarchy

To fully understand the caches, we need to know of all the different types for memory used in a C++ program. Handling memory properly is one of the most important parts of C++ optimization, because memory access is much slower than the CPU. Memory is the bottleneck, and you need to know where the compiler puts everything.

Learn to love the linker-loader!

When your program starts running, the "loader" puts all sorts of things in different places. The basic moving parts that happen *before* execution starts are:

- Instructions — the code's machine instructions.
- Global read-write memory — initialized or zero-initialized global variables.
- Read-only data — string literal data.

To get deeper into the memory segments used by the linker-loader, these are the main ones:

- text — stores the machine code instructions (read-only, executable)
- bss — all zero'd global data such as global arrays without non-zero initializers (read-write)
- data — Initialized non-zero global variable data (read-write)
- rodata — read-only data such as string literals or constant globals (read-only)

Yes, the "text" segment has a confusing name, and it's sometimes called the "code" segment. According to Wikipedia, BSS stands for "Block Started by Symbol," but you didn't need to know that.

All of the above segments are statically resolved, for the most part, by the linker. However, once the program gets going, there are more dynamic allocations for memory within its virtual address space. The main types of dynamic memory are:

- Stack memory — the function call stack with parameters and local variables (also `alloca`).
- Heap memory — dynamically allocated by the `new` operator or `malloc` function.
- Thread-local storage — via the "`thread_local`" keyword (C++11).

See Chapter 8 for more about reducing stack and heap memory, and now let's discuss thread-local storage.

Thread-Local Storage

Thread-Local Storage (TLS) is memory that is exclusive to a particular thread. The other threads do not have access to it. In C++, this is defined via the "`thread_local`" keyword, available since C++11. The usage is simple:

```
thread_local int tls_variable;
```

There are also some earlier and non-standard versions:

- `_Thread_local` — older version of specifier.
- `__thread` — GCC non-standard modifier with similar semantics.
- `__declspec(thread)` — on Microsoft C++.

The key features of `thread_local` variables are:

- Accessible in one thread only.
- Persistent memory storage.
- Variables, objects or arrays only (cannot have a `thread_local` function).

Per-thread access. If you declare a variable as "`thread_local`" then the C++ compiler has to ensure the semantics. Accesses to that variable in C++ must go to the version of that variable for the current thread. Typically, this means that the variable has multiple copies, with different addresses for each thread.

How is it implemented? It's not necessarily using any particular hardware support behind the scenes, and it's not necessarily using any magic per-thread caching.

The C++ compiler can allocate different addresses per thread to the same data, and then ensure that accesses within each thread get the correct version. After all, the C++ compiler knows that a particular variable is "`thread_local`" because it's a type specification.

Persistent memory semantics. The thread_local specifier is very similar to the static keyword in terms of its memory persistence. Its effect is similar to:

- Global variables (with external scope linkage)
- `static` file-scope variables
- `static` local variables (in a function)
- `static` data members (in a C++ class)

A `thread_local` variable is created when a thread starts and destroyed when the thread finishes. This has some implications:

- At most one copy is created at program startup.
- Dynamically created (along with the thread itself).
- Does not persist across thread shutdown and restarts.

Note that persistence and scope are different things. Persistence is whether the data is maintained across multiple accesses, whereas scope is simply whether its name can be referenced within code statements.

For example, if you use a `thread_local` variable as a local variable in a function, its value will persist across invocations to that function, and always have the same address. However, it's scope is limited to within the function, where its name is accessible. This is the same as a `static` local variable, but with the extra semantics that only one thread can see this version. If multiple threads call the function, they'll get different versions of the `thread_local` variable inside the function.

Thread-local variables occupy a special niche in the programmer's bag of tricks. You don't need to wrap accesses with any locking or other synchronizations, which is nice. They are like atomics, in that they cannot be messed up by another thread, but unlike atomics because they are not shared across threads.

The main usage is to have some shared code, but also have a special non-shared variable, especially where you want the variable to persist, such as having per-thread counters, flags, intermediate calculations, and so on.

References

1. Wikipedia, May 2025 (accessed), *.bss*, https://en.wikipedia.org/wiki/.bss
2. Milan Stevanovic, 2014, *Advanced C and C++ Compiling*, Apress, https://www.amazon.com.au/dp/B01HXFLQH0/
3. John R. Levine, 1999, *Linkers and Loaders*, Morgan Kaufmann, https://www.amazon.com/dp/1558604960
4. CPP Reference, May 2025 (accessed), *Storage class specifiers*, https://en.cppreference.com/w/c/language/storage_class_specifiers.html
5. Microsoft, 2021 *Thread Local Storage (TLS)* https://learn.microsoft.com/en-us/cpp/parallel/thread-local-storage-tls
6. Larry Jones, 27 Feb 2025, *Mastering Concurrency and Multithreading in C++: Unlock the Secrets of Expert-Level Skills*, https://www.amazon.com.au/Mastering-Concurrency-Multithreading-Secrets-Expert-Level-ebook/dp/B0DYSB519C/

20. Cache Warming

What is Cache Warming?

Cache warming is a specific type of prefetching optimization 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 (e.g., a trade execution in HFT), then memory accesses are very fast.

There are multiple ways to trigger prefetching of data 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 multithreading, 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_order_book[0];
```

The C++ compiler may notice that "`temp`" is not used anywhere else, so it can throw away that entire assignment statement. The solution is to use the `volatile` specifier:

```
volatile int temp = my_order_book[0];
```

The compiler is forced to load the data into memory even when it seems to be unused by the remainder of the code, because assigning 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!

Dry-Run Executions

A simple approach to cache warming is to still execute all the steps, even if you're not going to do anything. For example, in HFT, you could call the "execute trade" function even if the decision is to *not* trade any stocks.

The method is simply to pass a Boolean flag indicating a "dry run" or "test run" or "warm-up run" or whatever term you like. A simple conceptual example:

```
if (!dry_run) {
    orderobj.setup(ticker, price);
    execute_trade(orderobj);
}
```

A better way to get more cache warming is to populate all the objects as if you were going to actually do a trade. At the very last step, the dry-run flag is tested, and no trade gets submitted.

```
orderobj.setup(ticker, price);
if (!dry_run) {
    execute_trade(orderobj);
}
```

But we really want to warm up the entire path, even the trade execution logic. Hence, we go deeper by passing the flag inside:

```
orderobj.setup(ticker, price);
execute_trade(orderobj, dry_run);
```

And our trade execution code looks like:

```
void execute_trade(Order &order, bool dry_run)
{
    if (!dry_run) {
        g_order_count++;  // Count total
        // Other accounting stuff..
        // Submit the order...
    }
}
```

That isn't really much better, is it? We didn't warm anything extra, but just pushed the test inside the function.

Double Data Trouble

We really need to actually prefetch some data! One way is to double up all our data. The basic data for order count tracking is like this:

```
int g_order_count = 0;
```

One common trick is to use an array of two values with two meanings:

- Live data
- Dry-run data (unused)

Hence, our order count becomes:

```
int g_order_count[2] = { 0, 0 };
```

Then we can try this:

```
if (!dry_run) {
    g_order_count[0]++;  // Live run
}
else {
    g_order_count[1]++;  // Dummy
}
```

The point of the dummy is that we access the `[1]` array element in order to warm up the `[0]` element (without changing it). This works because of "false sharing" with "cache lines," which is often a slowdown problem, but here they offer an advantage. We can warm the cache by touching adjacent array elements, without disturbing the main data. (Note that here we don't use the `alignas` trick to avoid false sharing, because we actually want it to occur!)

In the spirit of branchless programming, we can make this code tighter by mapping the Boolean flag to 0 and 1 integer values:

```
g_order_count[(int)dry_run]++;
```

Note that we have actually added extra computation to our hot path! Instead of a global variable increment, it's now an array index lookup plus the increment.

We need to measure our optimizations to ensure that the gain from memory cache warming is greater than the extra cost of these array indexing operations. (We've also added a large amount of extra computation to our cold path, including whole extra function invocations, but we care less about that.)

Our conceptual trade execution routine starts to look like:

```
void execute_trade(Order &order, bool dry_run)
{
    g_order_count[(int)dry_run]++;  // Count total
    // Other accounting stuff.. same tricks
    if (!dry_run) {
        // Submit the order...
    }
}
```

The idea is that our "dry run" mode has run over as much of the code as possible, only stopping short of actually submitting the order. By maintaining two full copies of all data, with dry-run and live values, we can prefetch all of those arrays into memory caches.

Problems with Cache Warming

The above cache warming double-array trick has used false sharing of cache lines for good, not evil. And yet it has a problem: false sharing.

Our use of false sharing was harmless (and helpful) because we assumed only a single thread was in use. There's no cache invalidation slowdown when it's only one thread. The cache warming idea for the L1 and L2 caches requires a single thread, although the L3 cache can be warmed for multiple threads. Hence, this cache warming idea has limitations:

- Single thread required for all order submissions (if you want L1/L2 cache warming).
- Thread pools and other multi-thread design patterns are therefore problematic.

We cannot really have a thread pool model where each consumer thread could potentially submit a trade. The above cache warming logic only works for one thread. If we try to use multiple threads, our cache warming logic is actually a cache freezing de-optimization, because we've got the "false sharing" problem for real.

Even worse, consider what happens if we try to use a thread pool model with the following modifications:

(a) multiple consumers, where each thread tries to decide whether to trade,

(b) single trade submission thread.

In other words, multiple decider threads, where each decider then hands off to the single trading thread (which is kept warmed).

But then we've made another conceptual error. The hot path should really include the decision logic, as the overall latency is from receiving incoming data to submitting a trade. However, we haven't kept the cache warm for these multiple "decider" threads, particularly so for all the data they use in deciding whether to trade, so the decision modules won't run fast.

Possible solutions include:

- Single thread for all decision and order submission (with L1/L2 warming), or
- Keep multiple threads warm (tricky!), or
- Modify the cache warming code tricks to use reads only, not writes (avoiding the cache invalidation problem), or
- Only warm up the L3 cache (for multiple threads).

But these solutions have additional problems:

- Single order thread idea lacks a failover or backup plan.
- Single order thread cannot issue two trades without blocking.
- Warming multiple threads means each thread needs its own copy of the data.

None of these solutions are great, so that's why they pay you the big bucks.

Further Optimizing Cache Warming

Another further iteration of advanced cache warming would be to actually submit a dummy order, such as if the exchange connectivity allowed the sending of test-only transactions. Doing this would allow us to keep warm any of the data structures that are actually inside the client API of the exchange connection.

The advantage of the use of dry-run cache warming is that all the various data structures used to prepare a trade are kept warm in the memory caches (L1/L2/L3). The downside is extra processing that occurs whenever you're not trading. 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 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 any one of the bytes 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.

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. Sarah Butcher & Alex McMurray, 2 January 2025, *The C++ techniques you need for $600k hedge fund jobs*, https://www.efinancialcareers.com/news/low-latency-c
3. Edelweiss Global Markets Oct 14, 2024, *Cache-Warming*, https://edelweissgm.github.io/hft/2024/10/14/CacheWarming.html
4. Ibrahim Essam, Jul 19, 2024, *Cache warming and memory access*, https://ibrahimessam.com/posts/cache/
5. Nimrod Sapir, 2019, *High-Frequency Trading and Ultra Low*, Latency Development Techniques, https://corecppil.github.io/CoreCpp2019/Presentations/Nimrod_High_Frequency_Trading.pdf, Code: https://github.com/DanielDubi/StaticFlatMap
6. 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/
7. 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/
8. Katecpp, Oct 5, 2015, *Improve performance with cache prefetching*, http://katecpp.github.io/cache-prefetching/

21. 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 big chunks of 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 in memory. Less obviously, similar comments apply to the memory storage for 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 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 there's negative zero, with the sign bit set, but all other bits zero. Hence, the bitwise testing of both float numbers fails if there's ever 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`. I think this `NaN` issue is less serious than negative zero, because if your computations are generating `NaN`, then they're probably already failing, and an incorrect `memcmp` equality test won't matter as much.

Bitfield problems. If our structure has any bitfield data members, this `memcmp` idea fails too. Bitfields are a standard C++ feature that is defined with a suffix colon and a number of bits like:

```
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, really sure that 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 inherited C++ base classes. You might have to intercept all `new` and `new[]` operators with your own link-time function 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 for 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); // Fail
```

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.

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 a 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.

22. False Sharing

False Sharing and Cache Line Sizes

False sharing is a slug in C++ multithreaded code preventing two threads from running as fast as they should. The idea of "false sharing" is that two threads can interfere with each other's memory caching. The sharing is "false" because it can occur with data that's not actually being intentionally shared between the threads, but is impeded simply because the memory addresses are too close together.

Why does it occur? The CPU's L1 and L2 caches don't just cache in single bytes, 16-bit words, or even 32-bit integers. Instead, they have caching in "chunks" in the hardware level, which are called "cache lines" (also "cache sectors" or "cache blocks" or "cache line sizes" or "bananas in pyjamas" if you prefer).

How big? Some examples of common sizes of these cache lines include:

- Intel CPUs — 64 bytes.
- Apple M2 — 128 bytes.
- Some AMD and other CPUs — 256 bytes.

Note that you can get this number for the L1 cache line size in bytes programmatically in C++17 via the functions declared in the `<new>` header:

```
hardware_destructive_interference_size()

hardware_constructive_interference_size()
```

What this means is that, on an Intel CPU, the caches are updated 64 bytes at a time, because one "cache line" is read or written as the minimum size. This is good because:

- Cache loads are 64 bytes in parallel (in hardware).
- Cache writes (updates) store 64 bytes in parallel.

But this is bad because:

- Invalidating one cache byte also invalidates all 64 cache line bytes.

This is where we have a slowdown from false sharing. If one thread sets any value in a 64-byte cache line, then all of the other 63 bytes are also invalidated in the cache. If a second thread needs to use any of those other 63 bytes, then it needs a cache line refresh. Slowness ensues.

Example of False Sharing

A common example would be two integers, each 4 bytes in size, but close together so that they sit inside the same 64-byte cache line. The most common problems arise with atomics or mutexes close together, but they can affect any global variable.

Hence, first a simple example without any atomics, mutexes, or other thread synchronization. Let's just look at two threads that are updating their own global variable, with no overlap between the threads. In theory, these two threads should not affect each other at all. In reality, there are CPU cache lines.

Here are our two global counter variables:

```
int g_counter1 = 0;
int g_counter2 = 0;
```

In practice, false sharing is more likely to occur with two atomics declared close together. However, in this example we're just testing with two completely unrelated threads, with absolutely zero synchronization happening between them. They really shouldn't impact each other, if not for false sharing.

Here is the sequential code, which sets two global variables:

```
void runtest1_no_threads(int n)
{
   for (int i = 0; i < n; i++) {
      g_counter1++;
   }
   for (int i = 0; i < n; i++) {
      g_counter2++;
   }
}
```

Here are the two threads that aim to set those two global variables in parallel. Note that each thread only accesses one variable, without any "sharing" going on.

```
void thread1(int n)
{
   for (int i = 0; i < n; i++) {
      g_counter1++;
   }
}

void thread2(int n)
{
   for (int i = 0; i < n; i++) {
      g_counter2++;
   }
}
```

And here's the basic thread launching code:

```
void runtest1_threads(int n)
{
   std::thread t1(thread1, n);
   std::thread t2(thread2, n);
   t1.join();
   t2.join();
}
```

Finally, here is the timing code using <chrono>:

```
g_counter1 = g_counter2 = 0;
auto before = std::chrono::high_resolution_clock::now();
runtest1_no_threads(n);
auto now = std::chrono::high_resolution_clock::now();
auto diff = std::chrono::duration_cast
     <std::chrono::microseconds>(now - before).count();
std::cout << "Time (no threads): " << diff
          << " microseconds" << std::endl;
```

Here are the speed results from executing the sequential and threaded code for 100 million iterations using g++ on Linux.

```
Time (no threads): 256079 microseconds
Time (2 threads): 209341 microseconds
```

Note that the threaded code does not actually run twice as fast as the sequential code, despite having two threads that should run in parallel. In fact, it only improves on the sequential code by about 19%, rather than 50%. Why?

It's the magic of false sharing, whereby one thread writing to its variable slows down the other unrelated variable that's only being used by the other thread. The two threads are constantly writing to their own variable, which messes with the cached value of the other global variable used in the other thread. It's kind of like entanglement in quantum physics, if you like that kind of thing.

Detecting False Sharing

According to the documentation, Valgrind's DRD tool should be able to detect false sharing (and numerous other thread errors). However, I ran the command:

```
valgrind --tool=drd ./test1
```

I did not get any warnings:

```
ERROR SUMMARY: 0 errors from 0 contexts
```

On closer reading of the DRD documentation, DRD seems to only detect a false sharing situation if the two threads are running on different cores, which may have been the reason.

Solutions for False Sharing

There are a few coding solutions to prevent false sharing. The basic idea is ensuring that the addresses of unrelated thread-shared global addresses are not too close. Options include:

- Putting global variables in random spots throughout your C++ code.
- Using `alignas` to enforce address spacing on alignment boundaries.

The first one is kind of a joke, although it would probably work in most cases. However, it's not technically guaranteed where the linker will put unrelated global variables in the address space.

A more elegant solution is to put variables, especially atomics, on address alignment boundaries. The idea is to ensure that each important global variable is alone in its 64-byte block.

The global variables in our declarations become:

```
alignas(64) int g_counter1 = 0;
alignas(64) int g_counter2 = 0;
```

By declaring them both as `alignas(64)`, it guarantees two things:

- The variables start on a 64-byte alignment boundary (we don't care about this here), and
- They are the only variable in that 64 bytes (this fixes false sharing).

The downside is that each 4-byte integer is stored in 64 bytes, so there's 60 bytes with unused padding added to global memory usage. But it's better to pad memory than to waste CPU cycles! (On the other hand, the CPU cache lines are also loading and storing 60 unused bytes, so we've somewhat undermined the efficiency advantages of the L1/L2 cache lines for this 64-byte block.)

Anyway, who cares, it works! Here are the faster speed measurements just from adding `alignas` statements:

```
Time (no threads): 260277 microseconds
Time (2 threads): 133947 microseconds
```

Wow! It's almost exactly half the time! The performance gain is about 49%, which is much better than 19% (due to false sharing slowdowns), and is close to the 50% gain we were aiming for with two threads. Maybe there's something to this multithreading stuff, after all.

Some Final Tweaks

As a finesse, you can assure that the addresses are far enough apart by simply checking in code. One possible method to make sure that some junior code jockey hasn't deleted your `alignas` statements:

```
assert( (char*)&var2 - (char*)&var1 >= 64);
```

Unfortunately, you can't do it faster at compile-time, since addresses of global variables are not "constant" enough for the compiler:

```
static_assert((char*)&var2-(char*)&var1>=64); // Fail
```

Note that some CPUs have cache line sizes up to 256 bytes. Hence, you might need `alignas(128)` or `alignas(256)` on those platforms.

Note also there are various other non-standard ways to achieve alignment, most of them having existed on platforms prior to the `alignas` specifier in the C++ standardization. For example, GCC has a whole set of old builtins. Feel free to use those old things and charge extra because you're writing antique C++ code.

Another point is that false sharing slowdowns can arise for non-global variables, such as dynamic allocated memory or stack addresses. It's not very likely for two threads to see contention over stack addresses inside their respective call frames, but it can occur with allocated memory blocks that are shared. There are various ways to get aligned addresses inside dynamic memory allocation, including aligned memory allocation primitives, so the same ideas can solve the problem.

Nevertheless, atomics declared as global variables are probably the most likely area where false sharing can occur. This suggests a general rule: all global atomics should be declared as `alignas`. I'm not sure I agree, and it does sound a bit drastic. This does avoid the performance slug of false sharing, but it will also waste significant memory with padding bytes.

References

1. Dung Le, Aug 13, 2020, *Optimizations for C++ multi-threaded programming*, https://medium.com/distributed-knowledge/optimizations-for-c-multi-threaded-programs-33284dee5e9c
2. Paul J. Lucas Jul 13, 2023, *Advanced Thread Safety in C++*, https://dev.to/pauljlucas/advanced-thread-safety-in-c-3ap5
3. Larry Jones, 27 Feb 2025, *Mastering Concurrency and Multithreading in C++: Unlock the Secrets of Expert-Level Skills*, https://www.amazon.com.au/Mastering-Concurrency-Multithreading-Secrets-Expert-Level-ebook/dp/B0DYSB519C/
4. Valgrind, March 2025 (accessed), *DRD: a thread error detector*, https://valgrind.org/docs/manual/drd-manual.html#drd-manual.limitations

23. Memory Pools

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 programming with `malloc`.

Some of the 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 — we don't mix small and large memory allocations.
- Less memory overhead — hidden heap memory "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.

Memory pools are not the only option for optimization memory allocation. In fact, the use of an open-source drop-in replacement for the standard C++ memory allocators is another significant option:

- jemalloc — the original FreeBSD allocator, now a Facebook favorite.
- tcmalloc — from Google, with an Apache 2.0 license.

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 `free`, 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 pool size —uses `std::array` rather than `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 storage
    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 large 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 storage
};
```

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 storage
    };
private:
    alignas(alignof(T)) std::array<Node, N> arr_;
    int freelist_[N];  // array of free indexes (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, "pop" the stack, to remove 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 to release a memory block. This is a push onto the stack.

```
void push_free_index(int index)
{
    assert(ctfree_ < N);
    freelist_[ctfree_] = index;
    ctfree_++;
}
```

And here's the version for 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...
    }
}
```

In summary, note that the push and pop of the free list stack is very efficient with O(1) complexity. Everything in this 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 and convenience features:

- Thread safety (e.g., a single mutex or a lock-free version).
- Intercepting the class-specific `new` and `delete` operators.
- Passing arguments to object constructors via parameter packs and `std::forward()`
- Placement `new` operator — does not really allocate memory!
- Custom allocators — memory pools via allocator functor objects.

Additional memory management features that could be added to a memory pool include:

- Dynamic expansion with multiple chunks rather than a fixed-size pool.
- 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.

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 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 standard C++ `std::hive` (C++26) as the underlying data structure, rather than a vector or array.

References

1. Sourav Ghosh, July 2023, *Building Low Latency Applications with C++*, Packt Publishing, https://www.amazon.com/dp/1837639353
2. Larry Jones, 27 Feb 2025, *Mastering Concurrency and Multithreading in C++: Unlock the Secrets of Expert-Level Skills*, https://www.amazon.com.au/Mastering-Concurrency-Multithreading-Secrets-Expert-Level-ebook/dp/B0DYSB519C/
3. Devansh, Feb 27, 2024, *A quick introduction to Memory Pools: Optimizing Memory Management in Software Engineering*, https://machine-learning-made-simple.medium.com/a-quick-introduction-to-memory-pools-cc3198d004db
4. happyer, Apr 23, 2024, *Memory Pool Techniques in C++*, https://medium.com/@threehappyer/memory-pool-techniques-in-c-79e01f6d2b19
5. Bernardo Palos, 2025, *Memory Pools in C++_ What They Are and How to Implement Them*, https://palospublishing.com/memory-pools-in-c_-what-they-are-and-how-to-implement-them/
6. Stack Overflow, 2019, *C++11 memory pool design pattern?* https://stackoverflow.com/questions/16378306/c11-memory-pool-design-pattern
7. Boost, 2011, *Boost.Pool*, https://www.boost.org/doc/libs/1_53_0/libs/pool/doc/html/index.html
8. Roger Ferrer Ibáñez, Nov 19, 2017, *A very simple memory pool in C++11*, https://thinkingeek.com/2017/11/19/simple-memory-pool/
9. Contributors, May 2025 (accessed), *jemalloc memory allocator*, https://jemalloc.net/, https://github.com/jemalloc/jemalloc (originally from FreeBSD, then updated by the Mozilla Foundation and Facebook, Inc.)
10. Google, May 2025 (accessed), *TCMalloc*, https://github.com/google/tcmalloc (Apache 2.0 License)

Appendix A: Long List of Low Latency Techniques

This is a compilation of coding efficiency and low latency C++ programming techniques from various books and articles:

- C++ Low Latency, David Spuler, March 2025.
- CUDA C++ Optimization, David Spuler, June 2024.
- Generative AI in C++, David Spuler, March 2024.
- 500+ LLM Inference Optimization Techniques (blog article)

Here's the long list:

Low Latency C++ General Software Approaches:

1. Cache warming
2. Core pinning ("affinity")
3. False sharing (avoiding)
4. Branch prediction optimizations
5. Hotpath optimizations
6. Slowpath removal
7. Kernel bypass
8. Lock contention (reducing)
9. Lock-free programming (with atomics and memory ordering issues)
10. Thread pools
11. SIMD CPU instructions
12. Inline assembly language ("asm" statements)
13. Intrinsic functions (often closely mapping to machine code instructions)
14. In-memory logging
15. Cache locality (for L1/L2/L3 memory caches and instruction caches)
16. Specialized data structures
17. Thread-Local Storage (TLS) ("`thread_local`" type in C++11)
18. Shared memory (e.g., `shmctl`, `shmget`, `shm_open`, `ftruncate`)
19. Memory mapped files/devices (e.g., `mmap`, `munmap`)
20. Asynchronous programming (`std::async`)

Concurrency-Friendly Data Structures:

21. Read-only data structures
22. Reader-friendly data structures (e.g., many readers, one writer)
23. Copy-on-write data structures (for readers)
24. Versioned data structures (for readers)
25. Partition data across threads (vertically: columns)
26. Shard data across threads (horizontally: rows)
27. Read-Copy-Update(RCU)—mostly the same as copy-on-write.
28. NUMA-aware data structures—reduce cross-node communications
29. Transactional memory (synchronization efficiency, reduces contention) — use atomic/isolated transactions (an emerging technology)

Hotpath Optimizations:

30. Optimize all steps in the hotpath (e.g., data ingestion, decision, trade execution, logging, risk management)
31. Profile the hotpath specifically (e.g., a test mode that always runs the hotpath)
32. Examine assembly code of the hotpath
33. Avoid memory allocation calls on hotpath (e.g., memory pools, preallocation)
34. Avoid free/deallocation of memory on hotpath
35. Use preallocated memory on hotpath
36. Review data de-serialization and serialization costs
37. Use in-memory databases for any significant amounts of incoming data
38. Keep the client network connection warm (method depends on the API)
39. Re-use objects to avoid constructor/destructor calls on hotpath

General Tuning Advice:

40. Avoid micro-optimization
41. Avoid optimizing error handling code (it's a slowpath)
42. Loop optimizations (see below)
43. Avoid nested loops
44. Tune inner loop for nested loops
45. Avoid excessive function wrapper overhead

Performance Profiling Tools:

46. `gprof`
47. `perf`
48. `prof` (older)
49. `pixie` (older)

Lock Contention Reduction:

50. Late lock acquisition
51. Early lock release
52. Short critical section of code
53. Generally reduce total numbers of locks used
54. Locking fine-grain vs coarse-grain
55. Use fine-grain locks for contested resources
56. Use a hybrid fine-grain/coarse-grain lock strategy
57. Release locks before significant computation
58. Copy data to temporary variables to unlock before computation
59. Release locks before blocking for I/O
60. Release locks before blocking for system calls
61. Release locks before blocking for networking
62. Tolerate lockless output overlaps
63. `std::shared_mutex` and `std::shared_lock` — multiple reads, one writer.
64. Double lock check method (check first without a lock)
65. Use message-passing via `std::promise` and `std::future` rather than shared memory.
66. Thread-specific queues and "work stealing" design pattern
67. Use a lock-free queue data structure
68. `thread_local` keyword (C++11)
69. `std::lock_guard` (C++11)
70. `std::lock_guard` early release by scope control
71. `std::unique_lock` (C++11) (more granular control than std::lock_guard)
72. `std::scoped_lock` (C++17)
73. Locking with timeouts (try locks)
74. Avoid spinlock busy waiting
75. Exponential backoff to avoid spinlock costs

See also "lock-free programming"
See also "concurrency-friendly data structures"

Thread/lock overhead reduction (generally):

76. Reduce thread launch overhead
77. Reduce thread destruction overhead
78. Reduce lock acquisition/release overhead
79. Reduce lock contention overhead
80. `std::make_shared()` or `std::allocate_shared()` do only one allocation (combined shared pointer and control block), whereas `shared_ptr<type>` does two allocations (shared pointer and the control block are separate).
81. Weak pointers (`std::weak_ptr`) can delay the deallocation of a `shared_ptr` and its object even after the main reference count is zero.

System code optimizations (general ideas):

82. Avoid system calls to reduce context switches (in Linux)
83. Use C++ "intrinsics" functions (highly optimized assembly-level code)

Linux socket programming:

84. Non-blocking sockets versus using `select()` with a timeout—allows thread to do "other" useful work rather than just wait.
85. `poll()` or `epoll()` system call rather than waiting

Context Switching Reduction:

86. Thread counts (not too many threads)
87. Thread specialization
88. Thread specialization (producer-consumer thread model)
89. Use custom thread pools with only preallocated memory block pools.
90. spinlocks avoid context switches (especially good if spins for only a short time)
91. Avoid context switch cost by having a thread do "other" work, rather than just blocking.

Cache Locality Optimizations:

92. Tiling/blocking algorithms
93. Tiling/blocking matrix multiplication (MatMul/GEMM)
94. Smaller data type sizes for increased locality
95. Choose a CPU with a larger L1 "cache line size" (64-256 bytes common)
96. `std::hardware_destructive_interference_size` (C++17)
97. `std::initializer_list` (C++11) can be used as a lightweight container with contiguous elements
    See also "cache warming (prefetch)" optimizations
    See also "false sharing (avoid)" optimizations

Instruction Cache Locality Optimizations:

98. Prefer shorter blocks of code in the hotpath
99. Consider not inlining function calls (for instruction cache locality)
    See also "branch prediction optimizations"

Branch Prediction Optimizations (General):

100. Branch elimination
101. Branch compiler hints
102. Branch prediction heuristics
103. Branch profiling (two-phase)
104. Branchless programming
105. Tools—measure branch prediction data (e.g., `perf`)

Branch Reductions Techniques:

106. Algorithm-level changes to reduce branches
107. Keep loop bodies short (shorter branches)
108. Reduce far branching (e.g., function calls)
109. Reduce overall use of function calls (see function call optimizations)
110. Reduce use of `if` statements
111. Reduce use of loops
112. Reduce use of `break` statements (in loops, not `switch`!)
113. Reduce use of `continue` statements
114. Reduce use of `switch` statements
115. Reduce short-circuiting in `&&`/`||` operators
116. Reduce short-circuiting of `?:` ternary operator
117. Avoid `virtual` function calls (hidden dynamic branches)
118. Avoid pointer-to-functions (hidden dynamic branches; blocks inlining)
119. Avoid function objects/functors (hidden dynamic branches)
120. Avoid lambda functions passed as arguments (depends on how well the optimizer can handle them)
121. Reduce long `if-else-if` sequences
122. Reduce nested `if-else` sequences
123. Avoid branches depending on anything unpredictable
124. Avoid branches depending on user inputs
125. Avoid branches depending on random numbers
126. Avoid branches depending on system clocks
127. Sort array data for efficient branch prediction, if scanning through the array comparing the data (e.g., before testing for error range)
    See also "compile-time optimizations" (remove branches at compile-time)
    See also "loop optimizations" (reduce loop iterations, e.g., loop unrolling)

Branch Prediction Heuristics:

128. Common case code in `if` block
129. Uncommon case code in `else` block
130. Error handling code in `else` block (uncommon code)
131. Avoid zero-iteration loops (never entered)
132. Avoid single-iteration loops (never loop back)

Branch Prediction Compiler Hints:

133. `[[likely]]` and `[[unlikely]]` path attributes (C++20)
134. `likely()` and `unlikely()` expressions (C++20)
135. `__builtin_expect` (GCC)
136. Define `LIKELY` and `UNLIKELY` macros with `__builtin_expect` (pre-C++20)
137. `[[noreturn]]` (C++11)
138. `[[assume(expression)]]` attribute (C++23)
139. `hot` (GCC function attribute)
140. GCC `__builtin_unreachable`
141. `std::unreachable`—helps branch prediction (C++23)
142. `[[fallthrough]]` — more for safety than speed (C++17)
143. `-fdelayed-branch` compiler flag
144. `-fguess-branch-probability` compiler flag
145. `-fif-conversion` and `-fif-conversion2` compiler flags
146. Use "`likely`" and "`unlikely`" in custom assertion macros
147. Use "`likely`" and "`unlikely`" in error handling code macros

Branch Profiling:

148. `-fprofile-arcs` (GCC option)
149. `-fprofile-generate` (GCC command-line argument)
150. `-fprofile-use` (GCC command-line argument)
151. Branch profiling with 100% hotpath (test modes)

Branchless Programming Techniques:

152. Ternary operator preferred over `if` statements (if `CMOV` instruction)
153. Boolean variables as 0 or 1 in arithmetic
154. Logical operators (`&&`/`||`) as 0 or 1 in arithmetic
155. Bitwise operators (`&`/`|`) replace logical operators (`&&`/`||`)
156. Sign bit extension bit masks
157. Lookup tables for branchless programming
158. `XOR` trick to swap two integer variables without a temporary variable

Slowpath Removal:

159. Optimize error checking pathways
160. Remove error checking tests
161. Defer error checking tests to later
162. Combine error checking tests together (and do it later)
163. Avoid adding error checks deeper in the call hierarchy
164. Never-failing functions (cannot return an error)
165. Don't use memory allocation (avoids memory allocation failure)

Cache Warming Methods:

166. Prefetch memory primitives
167. `__builtin_prefetch` (GCC)
168. `_mm_prefetch` (GCC)
169. `volatile` on temporary variables
170. Dry-run execution mode
171. Branchless dry-run execution with `arr[2]` declarations
172. Use read-only cache warming pathways (avoids cache invalidation for other threads)
173. Use deep cache warming all the way down into the NIC
174. Optimize cache warming code by reducing data reads (relies on cache line sizes)
175. Reduce cache warming code to the maximum size of the memory cache (avoids redundant cache warming when cache is already full).

False Sharing (Avoiding):

176. Using `alignas(64)` or 128 or 256 to avoid false sharing (C++11)
177. Use `alignas` on all shared memory or atomics (C++11)
178. Tools to automatically detect false sharing (DRD fails?)

Parallelism (General Categories):

179. Multithreading
180. Multiprocess
181. Vectorization
182. Pipelining
183. Parallel execution modes (C++17)
184. Coroutines (C++20)

Advanced C++ Concurrency Data Structures:

185. Read-only ("immutable") data structures
186. Lock-free algorithms and data structures
187. Linear search can be efficient for small sizes because of cache prefetching (e.g., rather than binary search; also doesn't need sorting maintained)

SIMD Instructions:

188. AVX (x86 CPUs)
189. ARM Neon
190. `std::simd` (experimental/C++26)
191. `<immintrin.h>`

Linux O/S Optimizations:

192. Process priority upgrades ("`nice`" command or system call)
193. Disable unimportant processes
194. Overclocking CPU
195. Overclocking GPU
196. Disable Security Enhanced (SE) Linux
197. Disable accounting mode in Linux (should be off anyway)

Linux Kernel Optimizations:

198. Scheduling algorithm kernel modifications
199. Tweak TCP/UDP network buffer settings (Linux kernel)
200. Turn off file "last access date" storage ("`noatime`" in `/etc/fstab`)

System Hardware Optimizations (Categories):

201. Processor hardware (CPU)
202. Network optimizations
203. Disk optimizations
204. RAM Memory optimizations

Processor Hardware Major Categories of Optimizations:

205. CPU
206. GPU
207. NPU
208. FPGA
209. ASIC

Networking Hardware Optimizations (Categories):

210. NIC
211. Switches
212. Load balancer devices
213. Size of the packet buffer of a switch (optimizing for)

Networking Transmission/Protocol Optimizations (Categories):

214. Physical proximity
215. Co-Lo
216. TCP vs UDP (faster than TCP but unreliable)
217. Optical networking (optical fiber cables)

218. Microwave network transmission
219. Packet fragment manipulations (e.g., out-of-order)
220. Reduce packet fragment collation overhead
221. Reduce packet consistency checking (error safety overhead)

Networking Software Optimizations:

222. TcpDirect/Onload
223. SolarFlare/OpenOnload (kernel bypass)
224. Exablaze (NIC with kernel bypass support)
225. DMA
226. PCIe bus
227. Compress data sizes for your network transmissions
228. Sticky sessions (avoids needing to send user caches between servers)
229. Shared storage vs other server-to-server networking (e.g., NAS/SAN)
230. Use custom wrappers for TCP and UDP network processing

GPU & Distributed Networking Optimizations:

231. RDMA
232. nvlink
233. Infiniband
234. RoCE
235. GPUDirect
236. PXN

Deployment Optimizations (Website backends):

237. DNS optimizations
238. Round-Robin DNS (RRDNS)
239. SSL time optimizations
240. etags (website server speedup)
241. Multiple identical servers architecture
242. Use subdomains for static files
243. CDN for static files
244. Compression modes enabled
245. Static files compressed
246. Minify static files (CSS, JavaScript)
247. Merge multiple small files together
248. Use smaller image files (low precision)
249. Merge multiple small icon images into one image file
250. Cache duration settings
251. Database optimizations (various, e.g., MySQL/MariaDB/MongoDB)
252. Database indexes
253. Application server optimizations (e.g., Tomcat)

Apache/Nginx Subprocess Optimizations:

254. Use FCGI not classic CGI integrations
255. Flush `stdout` of subprocesses (sends partial output earlier to Apache or Nginx)
256. Close `stdout` of subprocesses before shutdown sequence (finishes earlier to Apache or Nginx)
257. Early tests for violations and invalidity (fails quickly)

Algorithm Enhancements:

258. Precomputation (lookup tables)
259. Precomputation to data file
260. Precomputation of source code
261. Incremental algorithms
262. Data structure augmentation
263. Parallelization
264. Vectorization
265. Caching
266. Lazy evaluation
267. Common case first
268. Simple case first
269. Approximate tests first
270. Bounding box approximate tests
271. Bounding sphere approximate tests
272. Avoiding `sqrt` by using arithmetic on squares
273. Integer arithmetic on squares: avoid floating-point by using arithmetic on squares
274. Use variance not standard-deviation (arithmetic on squares)
275. Approximations
276. Compute budget algorithms
277. Probabilistic/stochastic algorithms
278. Skipping algorithms
279. Heuristic algorithms
280. Greedy algorithms

Memory Reduction Strategies:

281. Take care with memory reduction as some methods can reduce speed (trade-offs)
282. Reduce allocated memory
283. Smaller data sizes
284. Pack data into smaller integer sizes
285. Pack data into bits
286. Pack data using bit-fields
287. Pack data into unions

288. Use `std::bitvector`
289. Use `std::vector<bool>` (it is a special bit-packed template instantiation)
290. Structure packing (also for class data members): reorder different-sized data members for better packing and fewer padding bytes
291. Structure packing: biggest data types first (heuristic)
292. Structure packing: MSVS `/d1reportSingleClassLayout` compiler option to report on it
293. `#pragma pack` reduces padding to reduce size, but may worsen structure access costs
294. Stack data reductions
295. Avoid deallocation of heap memory when in shutting-down mode

Heap Allocated Memory Reduction Strategies:

296. Fewer allocated memory blocks
297. Avoid frequent small allocations
298. Preallocation of dynamic memory
299. Memory fragmentation avoidance
300. Memory leak avoidance
301. Merge memory allocations together
302. Memory pools (fixed-size allocations, often a type of preallocation)
303. Memory pool with O(1) deletion and O(1) insertion via permutation array
304. Merge fixed-size allocated objects into a large array
305. Custom memory allocators (generalized)
306. Class-specific memory allocator
307. Custom global memory allocator
308. Late allocation (allocate memory as late as possible)
309. Early `free` memory (deallocate as early as possible)
310. Early `delete` memory (deallocate early)
311. Avoid `realloc` (slow, memory fragmentation)
312. Smart dynamic buffers (hybrid of allocated and non-allocated memory)
313. `std::aligned_alloc` - memory alignment improvement (C++17)
314. `std::aligned_union` (C++11)

Static Memory Size Reductions:

315. Avoid large global arrays and buffers
316. Avoid large static arrays and buffers
317. Avoid large static C++ data members
318. String literal memory reductions

Stack Memory Size Reductions:

319. Avoid large local arrays and buffers
320. Avoid large function non-reference parameter arrays and buffers
321. Use pass-by-reference on large function parameters
322. Use integer parameters as local variables
323. Consider stack versus memory allocation
324. Flattening/reducing function call hierarchy
325. Inline small functions (compiler can disappear them)
326. Use `#define` macros for small functions (versus inlining)
    See also: function call hierarchy flattening
    See also: recursion avoidance

Code Size Reduction Strategies:

327. Code size reductions
328. DLLs versus static libraries
329. Remove executable debug information
330. Avoid the compiler "`-g`" debug option
331. Avoid the compiler "`-p`" profiler option
332. Unix `strip` command
333. Avoid large `inline` functions (instruction cache locality)
334. Don't overuse "always inline" or "force inline"
335. Template overuse
336. Google "`bloaty`" tool

Standard Library Optimizations (STL Optimizations):

337. String processing efficiency (e.g., "+" for `std::string` can be slow)
338. `std::vector` of non-trivial class objects calls constructor/destructors
339. Control array size for `std::vector` using "`reserve()`"
340. Use `std::sort` rather than `qsort`
341. `bsearch` is not your friend
342. Consider hard-coded arrays versus `std::array` versus `std::vector`
343. Compare the first letters of strings before calling `strcmp`
344. Consider type casts to `int` versus `round()`, `ceil()`, `floor()`
345. Avoid `printf/fprintf` format string processing
    with `putchar/putc/fputc` or `puts/fputs`
346. Hand-code versions of `abs` and `fabs/fabsf` that don't
    handle `Inf/NaN` numbers (but benchmark it).
347. Change `strlen("literal")` to `char arr[]="literal"` and
    use `sizeof(arr)-1`
348. Don't use `strlen(s)` in a `for` loop condition
349. Consider your own `atoi/itoa` versions that don't handle all the
    obscure cases.

350. Avoid `sprintf` and `snprintf` (both are slow)
351. `sync_with_stdio(false)`
352. `std::stringstream` is slow (hand-code text field processing instead)

Data Structures:

353. Hashing (basic)
354. Perfect hashing
355. Bit vectors
356. Bit sets
357. Bloom filters (bit vectors + hashing)
358. Binary tree
359. Sorted arrays
360. Unsorted arrays
361. Stacks
362. Queues
363. Dequeues
364. Vector hashing
365. Permutation arrays
366. Locality-sensitive hashing (LSH)
367. Bit signatures (vector algorithm)
368. K-means clustering (vector algorithm)
369. Hyper-cube (vector algorithm)
370. Approximate nearest neighbor (ANN) (vector algorithm)

Variable Optimizations:

371. Prefer `int` types to `char` or `short` (usually)
372. Prefer `int` types to `unsigned int` (usually)
373. Prefer `int` types to `size_t` (usually `unsigned long`; `uint32_t`)
374. Avoid unnecessary initializations
375. Re-use objects to avoid initializations/destruction
376. Avoid temporary variables
377. Use reference variables instead of full temporary variables
378. Avoid creating temporary objects
379. Put commonly used data fields first in struct/class
380. Declare variables as close as possible to usage
381. `if` initializer syntax (C++17)
382. `switch` initializer syntax (C++17)
383. Avoid bit-fields (smaller but slower to access or set)
384. Use memory alignment primitives to avoid slow-downs
385. Put the most-used data member first (it has a zero offset)
386. Order data members most used to least (smaller offsets are faster)
387. Array initializer lists as local variables (re-initialized each call)
388. Structure of arrays (SoA) data layout is more vectorizable than Array of Structures (AoS).

Arithmetic Optimizations:

389. Operator strength reduction
390. Reciprocal multiplication
391. Integer arithmetic
392. Use `float` not `double`

Expression Optimizations:

393. Expression transformations
394. `const`
395. `mutable` keyword — bypasses `const` (C++98) (speedy but unsafe)
396. Common subexpression elimination (CSE)
397. Constant folding
398. Template fold expressions (C++17) are concise but often lots of computation
399. Expression templates—avoids explicit temporary variables, compiler optimizes it better.
400. Constant propagation
401. Redundant assignment removal
402. Strength reduction
403. Algebraic identities
404. Implicit type conversions (avoiding; type consistency)
405. `explicit` keyword (prevent implicit type conversions) (C++98)
406. Brace initialization syntax `{}` (avoids implicit narrowing conversions)
407. `auto` variable declarations avoid accidental temporaries and implicit type conversions.
408. Don't mix `float`/`double` types (including their constants)
409. Don't mix integer types
410. Prefer signed integers over unsigned types
411. Short-circuiting of sub-expressions (using `&&`/`||`/`?:`)
412. Register allocation optimizations
413. `mprotect` page system call — used as optimization to make memory writeable
414. `<algorithm>` simple algorithms: `min`, `max`, etc.
415. Range check faster with casts via "`(unsigned)i < MAX`" not "`i >=0 && i < MAX`"

Memory Block Operations:

416. Prefer contiguous memory (locality, efficient block operations, etc.)
417. Different class types can allow block copying: POD (Plain Old Data), trivial types, standard layout types (e.g., check in a template using `std::is_trivial<T>`)
418. Copy arrays by wrapping them in a dummy `struct`
419. Copy arrays with `memcpy`
420. Compare arrays with `memcmp` (very dangerous: padding bytes, negative zero, NaNs)
421. Use `memcpy` not `memmove` if arguments won't overlap.
422. Linearize multi-dimensional arrays (contiguous memory blocks)

Operator Strength Reduction Optimizations:

423. Replace `*` with bitshifts
424. Replace `*` with addition
425. Replace `x*2` with `x+x`
426. Replace `%` with bitwise-and (`&`)
427. Replace `%` with increment and test
428. Replace `%` with type casts (if byte sizes)

Bitwise Optimizations:

429. Intrinsic bitwise functions
430. `CLZ` (count leading zeros) bitwise intrinsics
431. `CTZ` (count trailing zeros) bitwise intrinsics
432. Popcount bitwise intrinsics (set bit count)
433. Kernighan bit trick (find highest bit set)
434. Fast NOR/NAND/XNOR via assembly instructions
435. Fast LOG2 of integers
436. Fast largest power-of-two of integers

Floating-Point Optimizations:

437. Convert float to 32-bit integers (float bit manipulations)
438. FTZ (Flush to Zero) mode
439. DAZ (Denormals Are Zero) mode
440. LOG2 of floating-point is the exponent
441. Zero/negative zero bitwise tests
442. Disallow negative zero (to use faster zero comparisons)
443. NaN (Not-a-Number) bitwise tests
444. Inf/-Inf bitwise tests
445. Avoid denormalized numbers
446. Disable denormalized numbers (subnormals) (compiler/library modes)

447. Avoid underflow in floating-point (ignore it)
448. Avoid overflow in floating-point (ignore it)
449. `memcmp` float vector equality (disallow special values for fast `float` vector equality comparison)
450. Fast detection of special values in `float` vectors (bitwise operations)
451. Floating-point intrinsic functions (various)
452. Exponent addition: bitshift floating-point by add of the exponent bits
453. Sign bit flipping/extraction/setting (bitwise tricks)

Compiler Settings for Floating-Point:

454. GCC `-ffast-math` option — faster math mode.
455. GCC `-fno-math-errno` — faster math by not setting `errno`.
456. GCC `-ffinite-math-only`
457. GCC `fno-trapping-math`
458. MSVS `/fp:precise`, `/fp:strict`, `/fp:fast`
459. Disable floating-point exceptions

Loop Optimizations:

460. Exit loops early (e.g., `break` or `return` statements)
461. Finish loop body early (i.e., `continue` statement)
462. Correct choice of loop
463. Loop unrolling
464. `#pragma unroll`
465. Loop fusion
466. Loop perforation (probabilistic)
467. Loop tiling/blocking
468. Loop fission
469. Loop reversal (don't use!)
470. Loop code motion ("hoisting")
471. Loop distribution
472. Loop iterator strength reduction
473. Loop coalescing
474. Loop collapsing
475. Loop peeling
476. Loop splitting
477. Loop interchange
478. Loop sentinel
479. Loop strip mining (loop sectioning)
480. Loop spreading
481. Loop normalization
482. Loop skewing
483. Loop interleaving

If Statement Optimizations:

484. Replace `if-else-if` sequences with `switch`.
485. Replace `if-else-if` sequences with lookup table loop.

Switch Statement Optimizations:

486. Use compact numeric ranges in `switch` (compiler can use a LUT)

Compile-Time Optimizations:

487. `inline` functions
488. `always_inline` specifier
489. GCC `flatten_inline` specifier
490. `gnu_inline` GCC specifier
491. Keep `inline` functions short (helps compiler to inline)
492. Keep `inline` functions in header files (source available to all its calls)
493. Avoid making a `virtual` function "`inline`"—compiles but usually is a slug.
494. `sizeof`
495. Use `sizeof` with `static_assert` (e.g., portability checks)
496. Virtual functions cannot be inlined (although it compiles)
497. Pointer-to-function usages of functions cannot be inlined
498. Function objects (functors) cannot always be inlined
499. Lambda functions cannot always be inlined
500. `inline` variables (C++17) (helps with linking)
501. `static_assert` (compile-time assertions)
502. `const` is good
503. `constexpr` (C++11) is great
504. `constexpr` functions allow `if`, `switch`, loops, etc. (C++14)
505. `constexpr` lambda functions (C++17)
506. `constexpr` and placement `new` (C++26)
507. References to `constexpr` variables (C++26)
508. `if constexpr` statements
509. `constinit`
510. `consteval`
511. `if consteval` (C++23)
512. Type traits <type_traits> (C++11)
513. `typeid` is slow (RTTI)
514. `std::is_same_v` (type trait test)
515. Template specialization (for specific types)
516. Template specialization (for constant integers)
517. Variadic templates (C++11)
518. Template Meta Programming (TMP) still works, but prefer `constexpr`

519. Auto-vectorization (by compiler)
520. Auto-unrolling of loops (by compiler)
521. SFINAE tricks (mostly an issue for compiler engineers)

Pointer Aliasing:

522. Reorganize functions with awareness of pointer aliasing issues
523. Restricted pointers (to avoid pointer aliasing slowdowns)
524. `-fstrict-aliasing` compiler option (alternative to using "`restrict`")

Pointer Arithmetic:

525. Loop pointer arithmetic
526. End pointer address tricks (Loop pointer arithmetic)
527. Use references not pointers (avoids null testing)
528. Prefer postfix operations with the `*ptr++` idiom (not prefix `++ptr`)
529. Pointer comparison tricks
530. Pointer difference tricks
531. Avoid safe pointer class wrappers (prefer raw pointers for speed)

Pointer Optimizations (Other):

532. `reinterpret_cast` (helps the optimizer and is effectively a free compile-time hint)
533. Avoid `dynamic_cast` (to downcast from a base to a derived class, which can be helpful for specializing member calls, but dynamic casts can be expensive at runtime because of RTTI)

Function Optimizations:

534. Return early from functions
535. Flatten function call hierarchies
536. Callbacks are an extra layer of function call
537. Lambda functions are convenient but are an extra function call layer (though often inlined)
538. Function objects (functors) are an extra function call
539. Avoid recursion (completely; we're not in High School anymore)
540. Replace simple recursion with a loop
541. Replace complex recursion with a stack
542. Tail recursion elimination
543. Recursion higher base level
544. Collapse recursion levels
545. Specialize functions with default arguments (use two versions)
546. Specialize functions with `void` and non-`void` versions (if return value often ignored)
547. Avoid function pointers (cannot be `inline` or `constexpr`)

548. Merge multiple Boolean function parameters into a "config" object with Boolean data fields.
549. `noexcept` attributes allow compiler to avoid adding extra code (C++11)
550. `std::initializer_list` can be used to return multiple values (benchmark against other methods)

C++ Class Optimizations:

551. `friend` functions (bypass interfaces)
552. `friend` classes (bypass interfaces)
553. Return references rather than objects
554. Avoid temporary class objects in expressions
555. Add extra member functions to avoid temporary object creation
556. Pass objects by reference to functions (i.e., "`&`" or "`const&`")
557. Disable copy constructors with "`private`" or "`= delete`"
558. Disable assignment operators with "`private`" and "`= delete`"
559. Declare assignment operators with `void` return type (except when defaulting)
560. Re-use objects to avoid constructor and destructor calls
561. Avoid calling the destructor when in shutting down mode
562. Uninitialized memory algorithms, e.g., `std::uninitialized_fill` (C++17)
563. CRTP (Curiously Recurring Template Pattern): derived class derives from base class which is itself a template involving a pointer to the derived class (optimizes polymorphism to be compile-time, avoiding virtual function calls; also this allows more inlining of these calls.)
564. Move constructors
565. Move assignment operators
566. `std::move` (C++11, C++14) is usually a compile-time cast.
567. Return object reference types (not complicated objects)
568. Avoid virtual function calls with explicit calls to the specific function
569. Specialize inherited member functions (for the more restrictive type)
570. Avoid overloading the postfix increment/decrement operators
571. Block the overloaded postfix increment/decrement operators (`void` body or `=delete`)
572. Consider skipping destructor cleanup if program is shutting down
573. Avoid accidental double initialization of data members in constructors
574. Avoid redundant initialization of same members in both constructor and "setup" methods
575. Specialize member functions with default arguments (use two versions instead)
576. Default constructors/destructors with "`=default`" may be more efficient than hand-coded versions.

577. Trick for singleton pattern in multithreading — threads initializing function-local static variable, other threads block, once-only initialization guaranteed by C++ compiler.

Advanced C++ Compiler Optimizations:

578. Copy elision (compiler auto-optimization with avoidance of a copy constructor in certain cases)
579. Guaranteed copy elision (C++17)
580. Named return value elision (a type of copy elision)
581. Temporary return value elision (a type of copy elision)
582. Copy elision in exception handling (special case for copy elision)
583. Allocation elision (new operator) (C++14)
584. Use xvalue or "expiring value" optimizations (various)
585. Trick: to disallow creating an object on stack, make its destructor private.
586. Trick: to disallow creating an object on the heap, make its `new` and `new[]` operators private.

Byte Block Operations in C++ Classes: (Use with extreme care!)

587. `memset/bzero` to zero in a constructor — fast but dangerous, overwrites internal "vtable" data in object if class has any `virtual` functions, does not call constructors of its data members or base class members; also cannot use an initializer list as this overwrites with zero after any objects were set by the initializer list.
588. `memcpy` to bitwise copy in a copy constructor or assignment operator — fast but dangerous, improperly copies internal vtable data in object if class has any virtual functions, does not deeply copy any of its members or base class members nor call their constructors.
589. `memcpy` to bitwise copy in a move copy constructor or move assignment operator — fast but dangerous; improperly copies "vtable".
590. `memcmp` to bitwise compare for equality/inequality tests — fast but fails in many situations due to pitfalls: padding bytes, bit-field members, negative versus positive zero floating-point values, NaN floating-point values.
591. Virtual inheritance — usually for pure virtual base classes; avoids double objects if the same base class is inherited in two different ways.

Timing C++ Methods:

592. `std::chrono` C++ class (highly granular)
593. `clock()` C/C++ function
594. `time` command (Linux shell)
595. `time()` function (granularity is only in seconds)
596. `gettimeofday()`

Benchmarking C++ Methods:

597. Loop unrolling for accurate benchmarking
598. Use `volatile` specifier for accurate benchmarking
599. Loop overhead measurement for accurate benchmarking
600. Google Benchmark: Apache 2 license;
code: https://github.com/google/benchmark

Compiler Settings:

601. Optimizer settings
602. Optimizing for space/memory size (compiler flags)

General Build & Software Development Practices for Efficiency:

603. Maintain separate builds for slow testables versus production executables
604. Compile-out assertions
605. Compile-out self-testing code
606. Compile-out debug code or tracing code
607. Ensure test code not accidentally left in production (test a global flag based on these macros at startup)

CUDA C++ GPU Optimizations:

608. Coalesced memory accesses
609. Thread specialization (GPU)
610. GPU thread pools
611. Producer-consumer thread pools
612. GPU kernel optimizations
613. Striding (GPU kernels)
614. Overlapping GPU uploads and compute
615. Overlapping with recomputation/rematerialization
616. Offloading to CPU
617. Pinned memory blocks
618. Warp divergence (warp coherence)
619. Grid optimizations
620. Grid size optimizations

Core Utility Classes (Efficiency Helpers): (to build for overall efficiency practices)

621. Bitwise macro library (bitflag management)
622. Floating-point fast bitwise operations macro library
623. Benchmarking/timing library
624. Smart buffer library (reduce allocations by combining allocated/non-allocated memory management)
625. TCP/UDP wrapper library
626. Specialized data structures for small amounts of data (faster than STL)

627. Sorted array and binary search (small array size)
628. Lock-free queues
629. Perfect hashing library
630. Bit vector data structures (possibly based on STL)
631. Bit set data structures (possibly based on STL)
632. Bloom filter library
633. Vector hashing library
634. Caching utilities library
635. Source code precomputation library
636. Basic data and statistics on vectors (e.g., averages, std dev/variance, etc.)
637. Incremental vector algorithms (averages, min, max, etc.)
638. Branchless coding primitives library
639. Graph library for locking analysis
640. Data compression library
641. Approximate tests library
642. Math library (versus STL)
643. Memory pools library (fixed-size custom memory allocators)
644. Custom memory allocator library
645. Placement `new` operator versions
646. Placement `delete` operator (write your own)
647. Multi-dimensional array library (linearize your vectors/matrices/tables/tensors)

AI Kernel Optimizations (using LLM Inference Optimizations for non-AI low latency applications): (subset of methods to consider)
Reference: 500+ LLM Inference Optimization Techniques (blog article)

648. Kernel fusion
649. Kernel fission
650. Kernel tiling/blocking
651. Quantization (integer-based approximation of floating-point)
652. Low-bit quantization
653. Binary quantization (1-bit)
654. Integer-only arithmetic
655. Floating-point quantization (FP16/FP8/FP4)
656. Mixed precision quantization
657. Logarithmic quantization
658. Dyadic quantization
659. Low rank matrices
660. MatMul/GEMM optimizations (many)
661. MatMul data locality optimizations
662. Sparse MatMul
663. Approximate matrix multiplication

664. Contiguous memory block matrix multiplication
665. Cached transpose MatMul
666. Fused transpose MatMul
667. Tiled/blocked MatMul
668. Sparsification (Pruning/Sparsity)
669. Token pruning (input compression)
670. Token skipping
671. Token merging
672. Data compression algorithms
673. Early exiting (of layers)
674. Caching optimizations
675. Vector computation caching
676. Zero skipping
677. Negative skipping
678. Padding optimizations
679. Zero padding removal
680. Zero-multiplication arithmetic
681. Adder/addition (zero-multiply)
682. Bitshifts (zero-multiply)
683. Bitshift-add (zero-multiply)
684. Double bitshift-add (zero-multiply)
685. Add-as-integer (zero-multiply)
686. Logarithmic arithmetic (zero-multiply)
687. Hadamard element-wise matrix multiplication
688. End-to-end integer arithmetic
689. Table lookup matrix multiplication
690. Weight clustering (grouped quantization)
691. Vector quantization
692. Parameter sharing
693. Activation function optimizations (non-linear functions)
694. Precomputation of Activation functions
695. Approximation of Activation functions
696. Integer-only approximation of Activation functions
697. Fused activation functions
698. Normalization optimizations (non-linear vector data functions)
699. Fused normalization optimizations
700. FFN optimizations (double MatMul)
701. FFN approximations
702. FFN integer-only
703. Decoding algorithm optimizations
704. Speculative decoding
705. Multi-token decoding
706. Ensemble decoding
707. Consensus/majority-vote decoding

708. Easy-hard queries
709. Batching computations
710. Advanced number systems
711. Posit numbers
712. Dyadic numbers
713. Hybrid number systems
714. Fixed point numbers (integers not floating-point)
715. Block floating-point (BFP) hybrids
716. Logarithmic number system (LNS)
717. Disaggregation (prefill/decoding)
718. Computation re-use
719. Conditional computation
720. Approximate caching
721. Addition arithmetic optimizations
722. Approximate addition
723. Bitwise arithmetic optimizations
724. Fast multiplication arithmetic
725. Approximate multiplication
726. Logarithmic approximate multiplication
727. Approximate division
728. Bitserial arithmetic

Appendix B: License Details

GGML License

Source URL: https://github.com/ggerganov/ggml/blob/master/src/ggml.c
Project: GGML, https://github.com/ggerganov/ggml
Authors: Georgi Gerganov and ggml authors.
Copyright: Copyright (c) 2023-2024 The ggml authors.
Date: July 26th, 2024 (accessed)
License Type: MIT License
License URL: https://github.com/ggerganov/ggml?tab=MIT-1-ov-file#readme

Llama.cpp License

Source URL: https://github.com/ggerganov/llama.cpp
Project: Llama.cpp framework, https://github.com/ggerganov/llama.cpp
Author: Georgi Gerganov
Copyright: Copyright (c) The ggml authors
Date: August 2024 (accessed)
License Type: MIT License
License URL: https://github.com/ggerganov/llama.cpp/blob/master/LICENSE