CUDA C++ Optimization

Coding Faster GPU Kernels

David Spuler

Aussie AI Labs

CUDA C++ Optimization

Coding Faster GPU Kernels

by David Spuler

Aussie AI

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

https://www.aussieai.com

First published: October 2024.

Preface

Why a Book on Optimizing CUDA C++?

NVIDIA's CUDA C++ environment is an incredible platform that allows the programmer to work at a much more productive level, far away from the low-level details of parallel programming on a GPU. But sometimes, you just can't avoid getting back down into the boiler room to make things even faster! This book examines a variety of techniques for optimizing CUDA C++ programs, from beginner to advanced, along with a catalog of common CUDA C++ slugs to avoid.

Who This Book is For

Anyone programming in CUDA C++ or trying to learn the language will benefit from better optimization skills! This book examines speedups in coding from beginner to advanced, starting with basic optimization techniques. In the later chapters, the book then covers a variety of advanced techniques for faster kernels, and a variety of common "slugs" that slow down CUDA programs.

How This Book is Organized

The best way to read this book is to open all 400+ pages and then read them all at the same time. That's how a GPU would do it, and what's good enough for silicon should work in carbon.

Alas, no.

Sadly, the book is organized sequentially, because we are mediocre lifeforms limited to reading one page at a time. Oddly, our brains can process a huge volume of input signals in parallel, but without much retention when we're trying to rationally learn something.

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

CUDA C++ Projects

Learn about our CUDA projects at https://www.aussieai.com/cuda/projects:

- Aussie CUDA C++ Debuglib — debug wrapper library for CUDA C++ primitives.
- Aussie CUDA C++ Emulator — educational tool for CPU execution of a limited CUDA subset.
- Aussie Lint — linter capability for CUDA C++.

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).

C++ Source Code

Most of the source code examples are excerpts from the Aussie AI C++ library, in many of the C++ source code examples. Details about source code availability can be found in the Aussie AI CUDA book area:

https://www.aussieai.com/cuda/optimization

Some code examples are abridged with various code statements removed for brevity or elucidation. For example, assertions, self-checking code, or function argument validation tests have sometimes been removed.

Most of the code is specific to CUDA C++, but should run across most platforms supported by the CUDA Toolkit. Some of the chapters also present generic C++ programming issues, which are not specific to CUDA, but nevertheless arise as problems in CUDA C++ programs as well.

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.

More Optimizing CUDA C++

A whole book on optimizing CUDA C++ isn't enough for you? You can find more on our website.

Updates and Bonus Materials: Additional book materials, bonus articles and chapters, updates and errata will be made available over time online at the Aussie AI website. Visit this URL: https://www.aussieai.com/cuda/optimization

Errata: Any bugs or slugs that we learn about in this work will be posted online on the Aussie AI website in the Errata section of Aussie AI research. Visit this URL to view these details: https://www.aussieai.com/cuda/errata

AI Research Literature Review: Ongoing updates to the AI research literature review are found in the Aussie AI Research pages, categorized by topic, starting at the entry page: https://www.aussieai.com/research/overview. The main CUDA research is available at: https://www.aussieai.com/research/cuda. If you have a correction to a citation or a new research paper to suggest for a category, please email `research@aussieai.com`.

Blog: Add a regular dose of *CUDA C++* to your feed. Review the Aussie AI blog at https://www.aussieai.com/blog/index, with a variety of articles on AI and CUDA programming.

Future Editions: Please get in touch with any contributions or corrections as future editions of the book are planned. I welcome suggestions for improvement or information on any errors you find in the book.

Disclaimers

Third-Party License Notices

Acknowledgements

This book would not have been possible without the help of others. Thank you to Michael Sharpe who lent his AI and C++ expertise to the project with industry guidance and technical reviews. Data scientist and architecture expert Cameron Gregory also provided much assistance with many contributions to various chapters on coding, architecture, and DevOps.

I would like to acknowledge the many GPU hardware engineers and other AI researchers and open source contributors who have made the AI revolution possible. In particular, the advanced coding skills shown in the many CUDA C++ projects and examples are acknowledged with both admiration and appreciation.

Please Leave a Review

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

Feedback and Contacts

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

About the Author

David Spuler is a 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 satirical fiction novella, *Animal Barn: A Cautionary Tail*, 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 ops book on BMC PATROL.

Other than writing, he's an avid AI researcher with a Ph.D. in Computer Science and decades of professional experience. Most recently, 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 also extends to areas of monitoring/observability, devops/MLOps, ITSM, low-resource LLM inference, Retrieval Augmented Generation (RAG) and AI-based agents.

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

Cameron Gregory is a technology entrepreneur including as co-founder of fintech bond trading startup BQuotes (acquired by Moody's corporation), 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++, Java, and JavaScript. He holds a Bachelor of Science with First Class Honors in Computer Science from James Cook University. His contributions to the book included detailed suggestions for scaling a high-traffic cloud architecture underpinning AI engines, and overall software development practices and tools.

Table of Contents

1. Parallel Programming

Parallel Execution

The reason that GPUs make code run fast is parallel execution of your code. You need to understand the basics of parallel algorithms for two reasons:

1. Understanding how the GPU is parallelizing computations, and

2. Organizing your high-level algorithms for greater parallelism.

The first part is mostly about one type of parallelism in the GPU. The second part ideally uses multiple types of parallelism to maximize overall speed of the total algorithm.

CUDA offers one particular type of parallelism, which is often known as Single Instruction Multiple Data (SIMD). What SIMD really means is that we do addition in parallel on 1,000 elements of a vector.

The "single instruction" is addition, and the "multiple data" is the many vector elements.

However, CUDA is not just SIMD, but it's more accurately called Single Instruction Multiple Threads (SIMT). This is a lot like SIMD, but it's somewhat more generalized, because it isn't just running an arithmetic data instruction. Rather, each computation runs in a separate "thread" in parallel, along with many other threads in lock-step over the data.

These threads have a fully-fledged runtime environment and they can all run in parallel under your control. As we'll see, CUDA threads can actually be like mini-programs, which is much more powerful than SIMD.

NVIDIA GPU Architectures

All of this is only possible from many years of GPU hardware optimizations. The state-of-the-art has continually advanced and recent GPUs are much more powerful than ones from only a few years ago. NVIDIA names its architectural generations for GPUs after famous individuals in Science, Mathematics, Physics, and Medicine:

- Blackwell (B100, B200 in 2024/2025) — Elizabeth Blackwell was a famous American Scientist and Physician, and was the first woman to gain an official medical degree in the USA in 1847.
- Hopper (H100 in 2022) — Grace Hopper, an American Mathematician, Computer Scientist and U.S. Navy Admiral, known for designing the COBOL programming language in 1959.
- Lovelace (L40 in 2022) — Ada Lovelace, an English Mathematician and early Computer Scientist in the 1800s, known for her work on the "analytical engine" of Charles Babbage, including identifying its non-computational applications.
- Ampere (A100 in 2020) — André-Marie Ampère, a famous French Physicist and Mathematician, known for electrical inventions such as the solenoid, and whose name is used as the "amps" unit for electrical current in modern day.
- Volta (V100 in 2017) — Alessandro Volta, a famous Scientist with a focus on electricity, whose name also denotes the unit of "volts" in modern usage.
- Turing (T4 in 2018) — Alan Turing, a famous British Mathematician, Computer Scientist and Cryptographer, known for discovering the use of computers for cryptography in wartime code-breaking.
- Pascal (P100 in 2016) — Blaise Pascal, a famous French Mathematician, Physicist, and Philosopher known for mathematical theory contributions such as Pascal's Triangle in binomial distributions.
- Maxwell (2014) — James Clerk Maxwell, a famous Scottish Mathematician and Physicist, known for the classical theory of electromagnetism.
- Kepler (2012) — Johannes Kepler, a famous German astronomer and mathematician in the 1500s/1600s, known for his laws of planetary motion.
- Fermi (2010) — Enrico Fermi, a famous American physicist and Nobel laureate, known for his work on nuclear physics in the 1900s.
- Tesla (2006) — Nikola Tesla, a famous Serbian-American engineer, physicist, and inventor, known for his pioneering work in electricity in the late 1800s and early 1900s.

CUDA C++

CUDA is the full software stack for the NVIDIA GPUs and CUDA C++ is the programming language used to program them. But CUDA isn't just a programming environment, but it's a whole ecosystem of tools, libraries, and platforms for parallel programming. A monumental amount of work has gone into offering an amazing suite of C++ libraries for almost anything you could think of.

Optimizing CUDA C++ has some aspects that are the same as standard C++ efficiency techniques, but then there's a whole gamut of extra techniques for fast parallelism and vectorization. The main advantage of CUDA C++ is that you can write these highly parallelized "kernels" to run on the GPU in a high-level language based on C++. This offers the benefits not only of speed, but also programmer productivity.

CUDA C++ uses a "dual model" of programming whereby you write the two programs inside the same C++ code. There are two main environments that you need to code:

- Host code — runs on the CPU.
- Device code — runs on the GPU.

Both types of code are written in fully high-level C++ statements. There's only a few differences that identify GPU code:

- `__global__` specifier on GPU kernel functions.
- A non-standard <<<...>>> "triple chevron" syntax for launching these GPU kernels.

If not for these differences, you'd hardly notice you weren't in standard C++. Hence, CUDA C++ leverages the experiences and capabilities of the C++ ecosystem, and then extends it to broad vectorization on the GPU.

In a typical application, the host code on the CPU is the "controller" of the overarching application algorithm, and the GPU is used as the grunt worker to smash through huge reams of data processing in its massively parallel hardware architecture.

It's hard for the poor CPU to keep up.

Porting CPU C++ to the GPU

Using CUDA C++ to unleash the power of the GPU onto your sequential C++ code is a great way to improve its performance.

The first point to note: *keep the CPU code!*

The CPU code is retained when porting to CUDA C++. The overall application is still controlled by the CPU code (called the "host"), and the tighter computationally expensive areas are sent to the GPU (called the "device").

Hence, the logical sequence of your application still follows the same sequences at a high level.

Also, the language differences between CUDA C++ and standard C++ are minimal. As a practical matter, the CUDA C++ compiler still uses the CPU's C++ compilers (e.g., GCC) for compiling the non-GPU code on the CPU platform, and only does special compilation of the GPU code (called "kernels").

Hence, it's not a rewrite (sigh). Instead, you have to troll through your CPU code to find parallelization opportunities. Some general pointers on where to start:

- Profile your CPU code to identify busy areas (as if you didn't already know!)
- Look for highly parallelizable algorithms (e.g., vectorizable, linear, 2D data tables, etc.)
- Unrolled loops are often a good indicator that you were trying to vectorize a busy loop. Roll them back up and build a CUDA kernel instead.
- Sharding of data tables often indicates an area that can be parallelized.

Some of the pitfalls of porting sequential code to CUDA C++:

- Multithreaded CPU code does not map one-to-one at a high level onto CUDA kernels. Don't try to port a huge high-level C++ block into a huge GPU kernel.
- Data transfer costs in partially ported CPU code can be a bottleneck, if the CPU and GPU have to keep sending computed data back-and-forth (in this case, finish the porting job!)

Overall, it should be an enjoyable and mind-expanding task to port most of your sequential C++ code to CUDA C++ on the GPU.

Optimizing CUDA C++

Optimizing your code is the main topic of this book. At a high level, some of the most important decisions in this regard include:

- Measure code performance with the CUDA profiler tools.
- Use the already-written CUDA library functions, as they're very fast.

To focus your research on the types of CUDA C++ optimizations to consider, the main points are:

- Parallelization is the main focus of code optimization for GPUs, affecting everything from the top-level task parallelism to the low-level GPU threads.
- Memory access and data transfer optimizations need to be considered in GPU applications, moreso than in standard C++ code, because there's simply a lot more data flying around.

There are multiple chapters in this book on each of those areas. You should be done soon if you speed-read over the weekend!

And it's not just the software. There are other layers of optimizations that are very important, perhaps moreso than the software:

- GPU hardware — more capable GPUs, and configuring them optimally.
- Networking technologies — high-volume switches and cabling is at a premium.

Happy optimizing!

2. Optimizing CUDA Programs

The best thing about optimizing programs written in CUDA C++ is that someone else has already done all of the work for you: the hardware engineers. Who knew that silicon could work so well? The GPU underneath the CUDA APIs is so blazingly fast that it really doesn't matter what miniscule software optimizations you try to apply. Your code will run fast no matter how hard you try to slow it down with debug wrappers.

So, go ahead, dig out your favorite bubble sorts and linear searches. Throw away that book by Donald Knuth, because you no longer need it. You'll never have to code another hash table in your life.

Oh, wait!

There's over 100,000 research papers getting written every year about optimizing AI software engines and their LLMs. Maybe there's still some CUDA C++ coding work to be done...

General Optimization Principles

There are many aspects to C++ code optimization on the CUDA platform. Some of the principles are general to all software programming, whereas some things are CUDA-specific.

Parallel Algorithms. What are you trying to do? Can it be parallelized for the GPU? That's the key. Honestly, if you can figure out how to run your algorithm in parallel, let CUDA do the grunt work on the GPU, and the rest is easy. Parallelization of software algorithms has spawned a thousand dissertations, but some general ideas include:

- Parallelize tasks that run sequentially (obviously)
- Overlap tasks
- Split data into sub-parts (i.e., parallelize partial computations).
- Break tasks into smaller sub-tasks that can be parallelized.

Other Algorithm Improvements. In addition to parallelization, there are algorithm-level improvements. Consider using optimizations such as:

- Incremental calculations
- Low-precision calculations
- Precomputations
- Approximate algorithms
- Specialized versions of general algorithms
- Lazy algorithms

Parallel Data Structures. Choose data structures with linearized data that is contiguous, as they tend to be parallelizable, such as:

- Arrays
- Vectors, matrices, and tensors
- Bit sets, bit vectors, Bloom filters

Inherently sequential data structures are often not great for GPUs, although there are numerous versions available already coded in CUDA libraries.

- Binary trees and tries
- Linked lists and queues/dequeues

Leverage CUDA Libraries. Don't code up things that have already been done by the folks at NVIDIA. There are many libraries for all sorts of applications, ranging from AI to games to science. These libraries offer extensive coverage of many of the common and obscure algorithms and data structures.

- cuBLAS — Basic Linear Algebra (e.g., vector and matrix operations).
- CUTLASS — advanced linear algebra operations.
- CCCL — CUDA C++ Core Libraries with: CUB, Thrust, and libcudacxx.
- CuFFT — Fast Fourier Transform
- cuDNN — Deep Neural Networks.
- CUDART — CUDA Runtime library.
- cuRAND — Random number generation.
- cuSPARSE — Sparse matrix multiplication.
- NVML — NVIDIA Management Library for low-level GPU access (similar to the `nvidia-smi` command).
- NVRTC — NVIDIA Runtime Compilation library for dynamic compilation of code (cool!).
- CUDA-X —libraries and micro-services for AI and other applications.
- RAPIDS — suite of GPU-accelerated AI libraries for CUDA.

Profilers. The capabilities of the NVIDIA performance profiling tools are amazing. There are GUI versions integrated into your IDE, or command-line versions to run. You can also add timing code into your own CUDA C++ programs, but you may not need to.

Compiler Auto-Optimizations. Modern compilers have amazing capabilities in optimizing code for you. Ensure that you have set all the optimizer flags, and also added "hints" to your code (e.g., restricted pointers, `const`, `constexpr`).

Compile-Time Optimizations. The run-time cost of anything that can be computed at compile-time is zero. Actually, literally, zero. You can compute a lot of things at compile-time using the various constant specifiers, specialized versions for particular sizes, and also via advanced `template` techniques.

Memory Costs. Many algorithms on a GPU are memory-bound, because the computations run faster than the memory access latency. The GPU has multiple levels of memory and caches (e.g., registers are the fastest, and global memory is the slowest), and there are various ways to optimize CUDA memory accesses. An important part of CUDA C++ optimizations is to think about data transfer costs and memory access costs.

Synchronization delays. Undoubtedly, some parts of your algorithm will need to wait for other parts to complete. These inherently sequential steps require you to use synchronization primitives. But don't overuse them, or use them when not necessary, as this slows down the entire algorithm.

Networking cost. Typical CUDA applications transfer data between the CPU host and the GPU device, which use an internal bus. However, if you have a multi-GPU application, or a large application that runs across multiple servers, then you'll also need to consider the network cost. There are various protocols to learn such as Direct Memory Access (DMA) and its remote version (RDMA).

Optimizing CUDA Kernels

Let's be honest. The only reason you're using CUDA is for speed. The whole goal of writing CUDA kernels is to tap into the awesome parallelization power of NVIDIA's GPU cores. Hence, the biggest way to "optimize" a CUDA C++ kernel is to figure out the very fastest way to massively parallelize the whole thing.

Algorithms, algorithms, algorithms!

This is a whole way of thinking, combining your knowledge of the problem being solved with how to get the most out of CUDA kernels. Thinks like figuring how to launch the most threads in parallel, avoid having dependencies between computations, and reducing the total amount of data transfers and data accesses.

A lot of the general CUDA optimization techniques apply broadly to many types of application, from generative AI to games to scientific processing.

Although we all know that CUDA C++ programs are very special, nevertheless, the usual general advice applies:

- Profile your code!
- Don't micro-optimize!
- Optimize your overall algorithms first.
- Focus on optimizing the busiest kernels.
- Pretend to be typing when your boss walks in.

The main high-level techniques for CUDA kernel speedups include:

- Parallelizing computations (surprise!)
- Reducing arithmetic workload (i.e., fewer FLOPs).
- Memory management optimizations (e.g., coalescing, locality).
- Data access and data transfer optimizations.
- High occupancy and load balancing of GPU streaming multiprocessors.
- Networking optimizations for multi-server data sharing.

At the algorithm level, this means strategies such as:

- Parallelize your algorithms at the top-level (task parallelism).
- Smaller data sizes (i.e., quantization of AI models).
- Fewer arithmetic operations (e.g., reducing multiplications by pruning LLMs).
- Memory-efficient algorithms (e.g., tiled matrix multiplication or memory efficient attention via Flash attention or Paged attention).

Compute Optimizations

Compute organization means organizing the computation so that it is done efficiently on your GPUs:

- High GPU occupancy rates (device utilization) via grid sizing and load balancing.
- Wave optimization (e.g., single-wave kernels, avoiding the "tail effect").
- Streams (task parallelism).
- CUDA Graphs (graph-structured compute path organization).
- Vectorization (do-it-yourself or compiler-optimized).
- Dynamic parallelism (further concurrency via kernels launching kernels).
- GPU isolation (limiting your GPU usage to benefit other workloads).

Compute reduction means optimizing the arithmetic workload on your GPUs:

- Tiling for improved data locality (e.g., matrix multiplication kernels).
- Kernel fusion (combining two kernels into one).
- Kernel fission (splitting one kernel into two).

Instruction processing optimizations include:

- Optimize Instruction-Level Parallelism (ILP)
- Increase instruction locality in code sequences (avoid GPU instruction cache misses).
- Thread coarsening (more instructions per thread, such as via loop unrolling or stride loops).
- Avoid using scatter and gather operations.
- Lazy loading of GPU modules in kernel initialization.

Synchronization optimizations include:

- Overlap computations where possible (asynchronously).
- Don't block in the host code unnecessarily.
- Run kernel threads in lock-step wherever possible.
- Use `__syncwarp()` sparingly.
- Same for `__syncthreads()` primitives.
- Avoid redundant barriers (unnecessary synchronization).

Some specific CUDA kernel optimizations include:

- Choose grid size, block size, and test them.
- Optimize the "occupancy" rates of blocks/threads.
- Use different configurations on different GPU architectures.
- Optimize horizontal reductions (many ways).
- Use warp horizontal reduction primitives.
- Warp coherence (avoiding warp divergence).
- Minimize wasted redundant threads (i.e., extra threads).
- Use multiples of 32 for data structure sizes.
- Use texture memory to optimize other types of spatial algorithms.

Other ways to run GPUs faster:

- Buy the next-generation of GPU (don't want for budget approval).
- Overclock your GPU! (as if it didn't run hot enough already).

Finally, don't forget that you're still programming in the C++ language! A large number of C++ optimizations still apply to CUDA C++, such as:

- Compile-time optimizations (e.g., `const`, `constexpr`, `inline`, restricted pointers, templates, etc.).
- Auto-vectorization via "hints" (e.g., restricted pointers, `#pragma`, loop unrolling hints).
- Use preprocessor directives (e.g., `#if/#ifdef`) to reduce live code.
- Operator strength reductions (e.g., bitshift for multiply/divide, bitwise-and replaces remainder).
- Loop optimizations (e.g., loop unrolling, loop fusion, loop fission, etc.).
- General usage of tight C++ coding practices.

That's already a long list, but we're only half way, because we haven't talked about memory yet.

Memory Optimizations

Computation is half the puzzle. The other half is the data, which means optimizing memory accesses and data transfers. Many CUDA kernels are memory-bound because the RAM chips can't keep up with the raw speed of the GPU.

Memory optimizations at a high level include:

- Faster memory usage (e.g., prefer shared memory over global memory).
- Cache optimizations (e.g., data locality, instruction locality).
- Memory access patterns (e.g., coalesced accesses).

Faster types of memory should be chosen:

- Maximize use of registers (i.e., use registers judiciously, or local memory where unavoidable).
- Reduce use of global memory (e.g., by using shared memory).
- Use warp shuffle operations rather than shared memory (or warp reduction primitives).
- Consider `malloc` heap memory versus `alloca` stack memory for kernel dynamic memory.

Fewer accesses to memory:

- Kernel fusion (combining two kernels into one).
- Memory-aware algorithms (e.g., Flash attention, paged attention).

Memory access optimizations include:

- Coalesced data access sequences in kernels.
- Grid-stride loop idiom for coalesced data access.
- Data locality optimizations (e.g., tiled algorithms).
- Use read-only cache memory for constant data (i.e., `__ldg`).
- Prefer Structure-of-Arrays (SoA) over Array-of-Structures (AoS) to arrange data contiguously.

Memory cache optimization is an important method:

- L2 cache-aware methods
- Reverse block access for cache optimization
- Instruction cache optimizations
- Memory prefetching

Avoid these memory access problems that will lead to increased latency:

- Avoid unaligned addresses in memory accesses.
- Avoid "bank conflict" contention on accesses to shared memory from multiple threads.
- Avoid exceeding Unified Memory limits (starts page swapping GPU memory).
- Use 128-byte aligned addresses for fully optimized coalesced data accesses.

Host-device data transfer optimizations include:

- Reduce host-to-device data transfers of input data (i.e., `cudaMemcpy`).
- Reduce device-to-host data transfers of results (also `cudaMemcpy`).
- Reduce global-to-shared memory transfers.
- Use page-locked "pinned" memory for faster host-to-device data uploads (e.g., `cudaMallocHost`).
- Unified Memory for shared CPU-GPU data (e.g., `cudaMallocManaged`)
- Overlap transfers with GPU computations (e.g., `cudaMemcpyAsync`).
- Heterogeneous Memory Management (HMM)
- Zero-copy memory for host and device joint access to fast memory.

Multi-GPU and network optimizations for data transfer:

- Multi-GPU peer-to-peer memory access.
- Memory-mapped I/O
- `nvlink` transfers
- Remote Direct Memory Access (RDMA)
- Lazy connection establishment for NCCL protocol.

Wow! That's a lot of optimization techniques. If I wrote a chapter on each of them, you'd be here till next Tuesday.

3. Vectorization

One-Dimensional Kernels

Vectorization is the basic optimization performed by GPUs, whereby each computation in a vector is parallelized into its own thread. In this way, all of the operations in a vector are performed in parallel.

The simplest type of CUDA C++ kernels are to operate on all the elements of a one-dimensional array or vector. There are two main types of one-dimensional kernels that you may need to implement:

- Vertical (element-wise) — super-fast!
- Horizontal (reductions) — slower.

The reason they are called “vertical” or “horizontal” is shrouded in mystery from time immemorial. After coming up empty in a search, I asked an AI about the origin of the terms, and it replied that the reason was based on whether the GPU was mounted horizontally or vertically inside the computer case. Umm, no. When I corrected it, then the AI suggested it was that vertical operations are “column-wise” whereas horizontal operations are “row-wise,” which is at least closer to an answer, but it’s really talking about matrices not vectors. When I asked who “coined” the terms, the AI claimed they “evolved organically” in the industry.

Vertical kernels. The simplest one-dimensional kernels are “vertical” or “element-wise” kernels, where each operation is only on one of the vector’s elements, and they are not combined. Examples include:

- Clear a vector to zero
- Set all vector elements to a scalar value
- Add a scalar to each element of a vector
- Scale a vector (multiply/divide each element by a scalar)
- Exponentiate each number in a vector (e.g., Softmax normalization in AI).
- Convert all negative values to zero (e.g., RELU activation function in AI).

I should get censured by the AI Worldwide Members Society for mentioning "divide." AI programmers never divide, but use reciprocal multiplication, thank you very much.

The above examples all involved only a single vector, but there are also common vertical operations on two vectors (or more):

- Copy a vector to a second vector
- Add a second vector to the first one (e.g., add a bias vector in AI)
- Add two vectors to create a third vector (common)
- Multiply element-wise two vectors (not very useful)
- SAXPY (Single-precision AX Plus Y)

Horizontal reductions. The other type is horizontal or "reduction" kernels, whereby the multiple elements of a vector are combined. The reason they are often called "reductions" is that they "reduce" a vector (many numbers) down to a scalar (single number).

Typical examples include:

- Sum of all vector elements
- Maximum of all vector elements
- Minimum of all vector elements
- Dot product (scalar product) of two vectors

Note that the dot product is multiplying the vector elements in pairs, and then adding that up, so it's like a combination of "vertical multiply" and "horizontal add" kernels. All AI engines are based on 2D matrix multiplications, which are based on 1D vector dot products, which is why you needed to know that.

Vertical Kernels

Vertical kernels can run so fast that we call them "embarrassingly parallel," because each parallel computation has no dependency on any others. The only trick is trying to figure out in what order we're going to process each vector element.

There are multiple ways to parallelize a one-dimensional element-wise operation on a vector in CUDA threads.

Let's examine some of them:

- Total parallelism — each thread processes one element.
- Segmented threads — each thread processes a contiguous segment of the vector.
- Grid-stride loops — each thread "strides" through the array, processing non-adjacent elements.
- Loop unrolled versions of all of these.

You might think at first blush that the best would be segmented processing, because that means adjacent addresses, which is good for "coalescing" of memory accesses. Well, unfortunately, that's when Data identifies a "sequential thought pattern" in the CUDA episode of *Star Trek*. The segmented version of the kernel actually has the *worst* performance with 100% non-coalesced accesses!

Total Thread Parallelism

This method is launching one GPU thread per vector element. Such a plan would seem to be the ideal method, and it certainly does run fast for small-to-medium vector sizes.

```
__global__ void aussie_clearvec_basic(
        float* v, int n)
{
    // Compute offset
    int id = blockIdx.x* blockDim.x + threadIdx.x;

    if (id < n) { // Safety
        v[id] = 0.0;  // Clear one vector element..
    }
}
```

This is launched with a thread for every single vector element:

```
int nthreads = 256;
int blocks = (n + nthreads - 1) / nthreads;
aussie_clearvec_basic<<<blocks, nthreads >>>(dv, n);
```

We choose threads-per-block of 256 based on a rule-of-thumb. It must be divisible by 32, and cannot be more than 1024.

This is a super-fast kernel with "total parallelism" because every thread sets only one vector element. Hence, every vector element is processed in parallel.

Note that we could even dispense with the safety `if` statement if we could be sure than `n` was a multiple of 256. The kernel is tighter:

```
__global__ void aussie_clearvec_unsafe(
                       float* v, int n)
{
    int id = blockIdx.x* blockDim.x + threadIdx.x;
    v[id] = 0.0;  // Clear one vector element..
}
```

And we could do a safety check on the host side, which is even removed from production code:

```
assert(n % 256 == 0);
int nthreads = 256;
int blocks = (n + nthreads - 1) / nthreads;
aussie_clearvec_unsafe<<<blocks, nthreads>>>(dv, n);
```

And we can further micro-optimize the arithmetic for the restricted values of `nthreads`:

```
// Simpler if: n%nthreads == 0
int blocks = n / nthreads;
```

One of the limitations should be fairly obvious: what if the vector has more elements than the maximum number of threads we're allowed to run on a GPU?

CUDA is limited to 1024 threads per block, but can handle a rather large number of blocks. Even if we don't hit a hard limit, if we do too many, the blocks will get scheduled in "waves" of computation, and we've created another inefficiency because launching so many blocks into the GPU is not free.

The second problem is that each kernel thread does so little work that the kernel launch overhead, especially across multiple waves, can become a significant percentage of the cost. It would be better to launch fewer blocks so that there are fewer threads to launch.

The solution to our problems it that we use non-total parallelism so that each thread does some extra work by looping through the array.

Segmented Loop Kernel

In this version, each thread does multiple vector elements in a segment of the vector, such that they are in a contiguous sequence. Let's just do a simple one that does 4 assignments in each thread.

```
__global__ void aussie_clearvec_segment4(float* v, int n)
{
    int id = blockIdx.x* blockDim.x + threadIdx.x;
    const int seglen = 4;  // Segment length to process
    id *= seglen;
    if (id + seglen - 1 < n) { // Safety
        for (int off = 0; off < seglen; off++, id++) {
            v[id] = 0.0;
        }
    }
}
```

The trick is this line, which ensures that thread 0 starts at 0, thread 1 starts at 4, thread 2 starts at 8, and so on:

```
id *= seglen;
```

Hence, each thread is doing 4 times the work of those in the prior example. However, this means we also need to launch 4 times fewer total threads.

```
int nthreads = 256;
int blocks = (n + nthreads - 1) / nthreads;
blocks /= 4;   // Use 4 times fewer blocks
```

Optionally, we can add various safety self-tests:

```
assert(n % 256 == 0);
assert(blocks * 4 * nthreads == n);
```

Why aren't these good CUDA kernels? Well, there's two reasons:

1. The inner loop is inefficient, and

2. The memory access pattern is not coalesced.

We can fix the inefficiency of the inner loop using loop unrolling:

```
__global__ void aussie_clearvec_seg4_unroll(float* v, int n)
{
    int id = blockIdx.x* blockDim.x + threadIdx.x;
    const int seglen = 4;  // Segment length to process
    id *= seglen;
    if (id + seglen - 1 < n) { // Safety
        v[id] = 0.0; v[id + 1] = 0.0;
        v[id + 2] = 0.0; v[id + 3] = 0.0;
    }
}
```

But, it's still not efficient because it's not a "coalesced" memory access pattern.

Well, let's consider a simple example: to process 40 elements in 10 threads, we use one thread for 0..3, another thread for 4..7, another for 8..11, and so on. So, if you think about the parallel execution, you'll realize that the first iteration, the 10 parallel threads will process addresses:

```
0, 4, 8, 12, etc.
```

And the second iteration they will process addresses in parallel:

```
1, 5, 9, 13, etc.
```

Although each thread is processing contiguous addresses (sequentially), the parallel processing of a full warp of 32 threads is not accessing adjacent memory addresses. Clearly, this is not a coalesced memory access pattern. See Chapter 9 for more details on coalescing.

Grid-Stride Loop Kernel

These loops are the GPU celebrities, always snapping selfies at any AI convention, because they're both fast and flexible (like The Flash and Dead Pool combined?). Here's an example:

```
__global__ void aussie_clearvec_grid_stride(float* v, int n)
{
    int id = blockIdx.x* blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for ( ; id < n; id += stride) {
        v[id] = 0.0;  // Clear one vector element..
    }
}
```

Now, the weird thing about grid-stride loops is that it doesn't really matter how many blocks or threads we spawn — it still works! This will work:

```
    int nthreads = 256;
    int blocks = (n + nthreads - 1) / nthreads;
```

That is just like "total parallelism" with only one assignment per thread. This spawns one thread for every vector element.

And this will also work, with now four assignments per thread, and four times fewer total threads spawned:

```
    int work_per_thread = 4;
    int nthreads = 256;
    int blocks = (n + nthreads - 1) / nthreads;
    blocks /= work_per_thread;   // Use fewer blocks
```

And even this will work with completely sequential iterations in one thread only, which is inefficient but occasionally useful for debugging:

```
    int nthreads = 1;
    int blocks = 1;
```

Striding. The idea with grid-stride loops is that we have a whole warp of threads processing contiguous memory. Weirdly, this means that each thread is processing elements that are non-contiguous. This is called "striding" through the array, at intervals equal to the size of the grid, which is why we call them "grid-stride loops."

Note that the meaning of "stride" in these CUDA kernels is the number of threads across the grid of multiple blocks. Other applications in Computer Science use the term to mean the number of bytes between array elements, which isn't quite the same thing.

Analysis of Grid-Stride Loops

Grid-stride loops are extremely fast, but a little tricky to code up. However, once coded correctly, they are also easy to use in any grid dimensions, because each loop adjusts its iterations so that the threads combine to exactly cover the entire vector.

In fact, grid-stride loops will even work if you launch a kernel with one block and one thread, which is useful in debugging.

Grid-stride loops are widely used in CUDA C++ kernels. Clearly someone in Engineering discovered these new types of GPU kernels, because if the Marketing Department has been involved earlier, we'd be calling them "Jumping Superfast Amplifiers" or "Flash Astral Stride Turnips (FAST)" or something like that.

Actually, no, I got it! We should call them "Gazelles" because Grid-Stride Loop is GSL. I should work in Marketing, and I'm sure that I would just love it, every single day.

Anyway, the reason that grid-stride loops work so well is counter-intuitive. If we have 100 vector elements processed by 10 threads, then the first thread does 0, 10, 20, etc. And the second thread does 1, 11, 21, etc. However, when you look at the whole 10 threads in parallel, at the first iteration they are processing:

```
    0, 1, 2, 3, ... 9
```

And the second lock-step computation of all 10 threads processes:

```
    10, 11, 12, 13, ... 19
```

The computation progresses over vector elements 0..9 then 10..19, and so on. This is the very definition of coalesced access in parallel by each warp. Hence, as you'd expect from such a great name, Gazelles are fast!

References

1. Mark Harris, Apr 22, 2013, *CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops*, https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/
2. Mark Harris, Jan 25, 2017, *An Even Easier Introduction to CUDA*, https://developer.nvidia.com/blog/even-easier-introduction-cuda/
3. Justin Luitjens, Dec 04, 2013, *CUDA Pro Tip: Increase Performance with Vectorized Memory Access*, https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/

4. AI Kernel Optimization

GPU Demand for AI

The future is bright for GPU hardware, and also for CUDA C++ programmers. The demand should remain strong for GPUs to run AI backends because of these trends:

- Consumer adoption — increasing steadily, and yet people haven't really figured it out yet.
- Business productionization — there is a continual release of POC projects into real-world usage.
- Multi-step reasoning algorithms — as exemplified by the recent OpenAI o1 model, the way to make AI models smarter is through running multiple sub-queries (e.g., Chain-of-Thought, Reflection, etc.), thereby soaring AI backend processing requirements.
- On-device is inadequate — although some workload will migrate to on-device LLMs, they just won't be fast enough for multi-step algorithms.
- Application-layer is just starting — expect a whole generation of new AI-centric applications.

With these tail-winds and the power of NVIDIA's next-generation GPUs, I expect CUDA C++ will remain in demand for years to come.

AI Kernel Bottlenecks

AI models and inference engines have some particular characteristics that enable additional optimization techniques. Some of the AI-specific optimizations for CUDA kernels are discussed here.

The main bottlenecks in AI computations are similar across all engines. There are two main number crunching sequences in AI inference, which is:

- Attention algorithm
- Feed-forward networks (FFNs)

Note that “attention” is sometimes called “self-attention” or “attention-heads” or “Multi-Head Attention” (MHA). Furthermore, another older name for the FFNs is “Multi-Layer Perceptron” (MLP).

Both the attention kernel and the FFN kernels are doing matrix multiplications, but they’re not the same thing. FFN kernels are more classic in their matrix multiplication kernels, but need to do so twice with an activation function in between. Attention kernels also use matrix multiplication, but in a very specific higher-level algorithm using QKV matrices.

Attention Kernels

For attention optimizations, you probably have heard of one of the newer memory-efficient attention algorithms:

- Group-Query Attention (GQA)
- Multi-Query Attention (MQA)
- Flash attention (there’s version 1, 2, or 3, so far)
- Paged attention

There are some variations, such as combining Flash attention with Paged attention. There have also been some other derivative papers on other LLM components (i.e., not the attention heads), such as Flash Decoding and Flash Inference.

There was a flurry of work on memory-efficient attention kernels that lead to Paged attention (implemented in the open source VLLM engine) and Flash Attention (versions 1, 2, and 3). However, the pace of those theoretical papers has since slowed, although the CUDA C++ implementations of these algorithms are now appearing in industry engines.

The newer area of research focus is that more attention (haha) is going to the KV cache bottleneck in attention computations, especially its unlimited memory size for long context queries. There is currently an endless stream of papers on KV cache compression, which are effectively a re-application of all the model compression theories to the KV cache, such as KV cache quantization (4-bits), KV cache pruning, KV cache layer fusion, and KV cache matrix factorization, just to name a few.

Prefix and Substring KV Caching

The other alternative to compressing the KV cache size is to avoid needing to compute it entirely by caching the data. This optimization means that almost the entire prefill phase of computation is avoided, leading to very low latency. The benefit is so high that several industry players are offering cheaper per-token prices for "cached tokens" (e.g., OpenAI and DeepSeek). No doubt, there are CUDA C++ engineers rushing to add this functionality at many other AI platform companies.

Prefixes are surprisingly common in AI inference. For example, the conversational history with a chatbot is always a known prefix. However, it only works partially for RAG chunks, which must be first in line to be seen as a prefix.

The only problem with prefix caching? It needs a prefix.

This is problematic for RAG chunks and other documents, which aren't always at the start of the token stream. Hence, the obvious generalization is to make it work for non-prefix substrings of tokens. For example, prefix KV caching requires to have seen a particular prefix, whereas substring KV caching could work for any appearance of a known token sequence, in any order (e.g., RAG chunks in any order). There's already research on this "substring KV caching" or "fused KV caches" but only a handful of papers have been released so far. I predict more!

How do we merge multiple KV caches?

It seems like it should be a fancy GPU computation, and yet, no. Instead, we just put one after the other, in a sequence, and pretend that we've computed the cache for the full token sequence.

This is "fused KV caching" and it's obviously a fast method that only requires memory copies and literally zero extra computations. It seems like this surprisingly simple result needs a little more research, but it may be that the solution is that easy. There are only a couple of papers in this area so far: Yao et. al. (2024) and Gim et. al. (2023).

Matrix Multiplication Kernels

And for the FFNs, there are already some very-optimized MatMul/GEMM kernels to consider. However, there are many variants on matrix multiplication to consider, including quantized versions (e.g., 8-bit or 4-bit integer kernels) or sparse matrix multiplications. An AI engine uses both matrix-matrix multiplication and matrix-vector multiplication computations.

Note that most matrices in AI kernels are rectangular rather than square. In fact, they're usually three-dimensional tensors shaped like a brick, rather than two-dimensional matrices.

The right FFN matrix multiplication kernel is a very critical choice to make in terms of the overall speed of your CUDA-optimized AI inference engine. Might want to take some extra time to benchmark your CUDA C++ implementation carefully.

Compute-Bound versus Memory-Bound

The above analysis of the two main computation costs is somewhat misleading, because it's forgotten about memory costs. The main GPT architecture is called the "Transformer" and it has a lot of components, with different characteristics across multiple "layers" of components. Generally, it has been found that the initial phase of "prefill" before outputting the first token is compute-bound, whereas the subsequent "decoding algorithm" that spits out one token at a time (called "autoregressive decoding") is memory-bound. Note that compute-bound means "GPU-bound" and memory-bound really means "VRAM-bound."

However, it's even more nuanced than that:

- Prefill phase — compute-bound.
- Attention algorithm (during decoding phase) — memory-bound.
- Feed-forward networks (during decoding) — compute-bound.

The reason is complicated, but it's basically that FFN matrices are more static than the attention matrices during the decoding phase. The FFNs are sitting there multiplying the same matrices over and over (i.e., compute-bound). However, the attention algorithm gets new KV matrices from the "KV cache" at every iteration, so it has the extra memory cost of managing that changing data (i.e., memory-bound). Hence, the above "memory-efficient attention algorithms" are focused on reducing memory and data access costs.

The prefill phase has both FFNs and attention computations (and KV caches), but both subcomponents are compute-bound. The initial prefill phase is not as dynamic with its data. This phase involves processing the input query tokens, rather than generating new tokens, so all of the tokens are known ahead-of-time, and so they can be loaded in, and it crunches away (i.e., compute-bound for both FFNs and attention).

State-of-the-Art Kernels

Some examples of the kinds of CUDA C++ kernel algorithms that are state-of-the-art as of this writing:

- Hybrid local-global attention — alternating layers of sliding window attention and global attention.
- 4-bit kernels for weights, activations, and the KV cache — dense, sparse, GEMM/GEMV, fused activation functions, and other variants.
- Advanced KV caching — prefix KV caching, RAG cache, etc.

Some of the even more obscure research-level algorithms that I believe may break through to mainstream include:

- Fused/substring KV caching — generalizing prefix KV caches.
- Block-level mixed-precision integer quantization — different quantization parameters for vector segments.
- Block Floating-Point (BFP) — granular representation of floating-point exponents, allowing integer arithmetic.
- Multi-LoRA memory-efficient loading/swapping algorithms — avoiding the weight update to the large model.
- Heuristic generalized speculative decoding — eschewing the small drafter model for efficient heuristics.
- Parallel decoding/lookahead decoding/multi-token prediction — bypassing autoregression.
- Advanced prefill optimizations — going beyond chunked prefill and phase splitting (disaggregated prefill).

Might need some CUDA C++ written for all of those!

State-of-the-Art Industry Backends

It's somewhat difficult to determine the state-of-the-art in LLM serving backends, as used by the industry's top players. Much of this information is commercially sensitive and is no longer appearing in public research papers (maybe it's in patents!). Nevertheless, there are some public papers and articles about various issues. Let's look at a few of them.

Character.AI companionbots backend. As detailed in their blog post, Character.AI has a very high level of traffic to their models. Inference optimization techniques include:

- INT8 quantization of weights and activations
- KV cache quantization (also INT8)
- MatMul INT8 kernels
- INT8 training (QAT)
- Hybrid attention with interleaved layers of local attention and global attention (with global attention for only approximately 1 in every 6 layers)
- KV cache compression
- Multi-Query Attention (MQA)
- KV cache layer fusion
- Session KV caching (for chat sessions)
- Prefix KV caching
- Sticky sessions (avoids copying session caches)

They cite a 13.5X reduction in cost versus use of commercial model hosting (i.e., by doing it themselves), and also a 33X reduction compared to when they started optimizing.

Together AI data center networking. In various papers and announcements, Together AI has offered various details of its backend platform. Some example software optimizations available include:

- CUDA backend for NVIDIA GPUs
- Flash Attention
- Flash Decoding
- Medusa decoding (multi-token prediction)

Together AI also described their GPU cluster management, including the networking aspects and validation steps.

This involves techniques and components such as:

- H100 GPUs
- Infiniband networking
- NVLink and NVSwitch
- NCCL
- HPCX
- SLURM (workload management)
- Remote Direct Memory Access (RDMA) using GPUDirect
- Telegraf (open source monitoring)

Important steps in the process include:

- GPU validation
- Network validation
- Storage validation

Optimizing the AI Stack

CUDA C++ optimization is important, but it's only part of the optimization puzzle. The inference engine running the LLM queries is one component of the AI tech stack. Optimizing CUDA C++ will make the AI engine backend run faster, but there are many other techniques for optimizing the entire tech stack of an AI application. Some of the main ideas include:

- Hardware layer — buy a bigger GPU (and write CUDA for it!)
- OS layer — tweak your Linux kernel parameters and process priorities.
- Model compression — make the AI models smaller via quantization, pruning, or distillation (or just use a smaller model!).
- Deployment stack — optimize serving and scheduling of user requests.
- Streaming — pass back partial results incrementally.
- Caching —inference cache, prompt cache, KV cache, or other types.
- Networking and bandwidth optimizations —RDMA, `nvlink`, and others.

RAG Stack Optimization. The RAG architecture has all of the above, and also has a set of additional optimizations:

- Faster vector databases (e.g., indexing, in-memory databases, nearest neighbor optimizations).
- RAG-specific KV caching (i.e., prefix caching, and fused KV caching).
- Rerankers and packers to better organize multiple RAG chunks.

That's all getting to be far beyond a CUDA C++ conversation. Overall, let's note that the inference time of an LLM still remains one of the main bottlenecks in the AI stack, so the user's latency will be greatly affected by your CUDA C++ optimization skills.

References

- Character.AI, June 20, 2024, *Optimizing AI Inference at Character.AI*, https://research.character.ai/optimizing-inference/
- Together AI, Nov 13, 2023, *Announcing Together Inference Engine – the fastest inference available*, https://www.together.ai/blog/together-inference-engine-v1
- Ryan Lucchese, Niki Birkner, Yaron Hagai, Virginia Adams, August 13, 2024, *Together AI, A practitioner's guide to testing and running large GPU clusters for training generative AI models*, https://www.together.ai/blog/a-practitioners-guide-to-testing-and-running-large-gpu-clusters-for-training-generative-ai-models
- Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, Junchen Jiang, 3 Jun 2024 (v2), *CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion*, https://arxiv.org/abs/2405.16444, Code: https://github.com/YaoJiayi/CacheBlend.git (Generalizes prefix KV caching to KV cache fusion with selective recomputation of some KV cache data.)
- In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, Lin Zhong, Nov 2023, *Prompt Cache: Modular Attention Reuse for Low-Latency Inference*, https://arxiv.org/abs/2311.04934 (Unique and insightful advance of generalizing KV caching to multiple prompts by computing a cache for short "segments" of prompts, including methods to adjust the different KV cache values for text segments that appear in different positions of the overall prompt.)
- David Spuler, March, 2024, *Generative AI in C++: Coding Transformers and LLMs,* https://www.amazon.com/dp/B0CXJKCWX9
- David Spuler, September 2nd, 2024, *500+ LLM Inference Optimization Techniques*, Aussie AI Blog, https://www.aussieai.com/blog/llm-inference-optimization

5. Profiling Tools

Profiling CUDA C++ Execution

There are various ways to time the execution of your CUDA programs. There are two basic strategies:

- Profiler tools
- Timers in the C++ code

This is basically an "inside or outside" choice. Using C++ timers wrapped around your functions is discussed in the next chapter. This chapter focuses on profiler tools that take your executable and examine its runtime efficiency.

Some of the examples of profiler tools available for CUDA C++ include:

- NVIDIA Visual Profiler — performance profiling with a GUI interface.
- Nsight Systems — system profiling and tracing.
- Nsight Compute — performance profiling for CUDA kernels.
- Nsight Graphics — specialized profiling for graphics applications.
- Nsight Deep Learning Designer — profiler focused on AI/ML applications.

Command-line profiler tools include:

- `ncu` — NVIDIA Nsight Compute CLI.
- `nvprof` — command-line profiler (now deprecated)

There are also some advanced APIs and SDKs available if you want to get ambitious and do some very deep integrations into the CUDA profiling tools:

- CUDA Profiling Tools Interface (CUPTI) — profiling and tracing integration API.
- NVIDIA Tools Extension SDK (NVTX) — tool integration API.
- Nsight Perf SDK — performance profiling for graphics applications.
- Nsight Tools JupyterLab Extension — extension for profiling of Python applications.

Nsight Compute CLI: ncu

I'm a big fan of command-line tools on Linux, so this section demonstrates the various reports from the CLI. The most functional profiler from NVIDIA is the Nsight Compute CLI, which is simply "`ncu`" on the command-line. It also goes by the much more impressive name "`nv-nsight-cu-cli`" and whoever chose that name must be good at touch typing.

If you prefer the GUI version, this is launched by the "`nv-nsight-cu`" command. There are a variety of graphical reports available in that interface, but I'd rather stick to text, thank you very much, mainly because I was programming Unix back when there were 27 types of Unix, rather than 27 flavors of Linux.

Running the command-line interface to Nsight Compute is as simple as this:

```
ncu a.out
```

The result is a text-based report at the end of execution with a variety of sections, focused on kernel efficiency.

Here is one section:

```
Section: GPU Speed Of Light Throughput
----------------------- ------------- ------------
Metric Name               Metric Unit Metric Value
----------------------- ------------- ------------
DRAM Frequency          cycle/nsecond         4.91
SM Frequency            cycle/usecond       576.05
Elapsed Cycles                  cycle        4,962
Memory Throughput                   %         3.10
DRAM Throughput                     %         0.02
Duration                      usecond         8.61
L1/TEX Cache Throughput             %         5.46
L2 Cache Throughput                 %         2.36
SM Active Cycles                cycle     2,811.53
Compute (SM) Throughput             %         9.81
----------------------- ------------- ------------
```

Here is another report summary showing details of the grid size and other parameters of the kernel launch:

```
   Section: Launch Statistics
   -------------------------------- --------------- ------------
    Metric Name                              Metric Unit    Value
    -------------------------------- --------------- -----------
    Block Size                                                 32
    Function Cache Configuration                   CachePreferNone
    Grid Size                                               1,024
    Registers Per Thread             register/thread           16
    Shared Memory Configuration Size           Kbyte        32.77
    Driver Shared Memory Per Block        byte/block            0
    Dynamic Shared Memory Per Block       byte/block            0
    Static Shared Memory Per Block        byte/block            0
    Threads                                   thread       32,768
    Waves Per SM                                             1.60
    -------------------------------- --------------- -----------
```

And here is an example of the occupancy-related report:

```
    Section: Occupancy
    ------------------------------- ----------- ------------
    Metric Name                     Metric Unit Metric Value
    ------------------------------- ----------- ------------
    Block Limit SM                        block           16
    Block Limit Registers                 block          128
    Block Limit Shared Mem                block           16
    Block Limit Warps                     block           32
    Theoretical Active Warps per SM        warp           16
    Theoretical Occupancy                     %           50
    Achieved Occupancy                        %        35.00
    Achieved Active Warps Per SM           warp        11.20
    ------------------------------- ----------- ------------
```

There are a variety of command-line options for `ncu`, such as:

- `-h` or `-help` — helpful helping helpers.
- `-v` or `-version` — profiler version information.
- `-mode` — useful for attaching to running processes.
- `-hostname` — used for remote debugging.
- `-devices` — limit profiling to chosen GPU devices.

There are numerous other command-line options, but I've run out of room in the e-book. You might have to read the documentation.

NVIDIA Profiler: nvprof

The command-line `nvprof` profiler tool is now "deprecated" and won't be around forever, but it's still useful for now. There is an "`nvprof` Transition Guide" document available, which was like learning about submarine sonar workings in the *The Hunt for Red October*, except more useful.

The report from `nvprof` is much simpler than that from ncu, which has more functionality. The focus is much more on the percentage usage by function call.

Here's a report from `nvprof` (truncated), which looks a lot like the output I'm used to from `gprof`:

```
==3137== Profiling application: ./a.out
==3137== Profiling result:
            Type  Time(%)      Time     Calls       Avg       
Min       Max  Name
 GPU activities:   36.23%  13.728us         1  13.728us  
13.728us  13.728us  [CUDA memcpy HtoD]
                   32.18%  12.192us         1  12.192us  
12.192us  12.192us  [CUDA memcpy DtoH]
                   22.13%  8.3840us         1  8.3840us  
8.3840us  8.3840us  aussie_clear_vector_kernel_basic(float*, 
int)
                    9.46%  3.5840us         1  3.5840us  
3.5840us  3.5840us  [CUDA memset]
      API calls:   96.45%  93.891ms         3  31.297ms  
872ns  93.889ms  cudaDeviceGetLimit
                    2.63%  2.5584ms        21  121.83us  
107.43us  140.16us  cudaGetDeviceProperties
                    0.31%  299.66us         1  299.66us  
299.66us  299.66us  cudaLaunchKernel
                    0.15%  145.36us       114  1.2750us  
142ns  56.265us  cuDeviceGetAttribute
                    0.14%  139.41us         2  69.704us  
50.862us  88.547us  cudaMemcpy
                    0.14%  133.71us         1  133.71us  
133.71us  133.71us  cudaMalloc
                    0.11%  109.20us         1  109.20us  
109.20us  109.20us  cudaFree
                    0.04%  41.426us         1  41.426us
```

As you can see, this gives a simple breakdown of the time spent in the various routines and APIs. Interestingly, it looks like `cudaDeviceGetLimit` is quite expensive.

6. Compilers and Optimizers

Compiler Optimization Options

Like most robust C++ compilers, the `nvcc` CUDA C++ compiler has a "`-O`" optimization flag, but it goes further: it has separate options for device code optimization, too. The `nvcc` compiler supports several different levels for optimization flags:

- Host code optimization flags: `-O` or `--optimize` with a level.
- Device code optimization flags: `-dopt` or `--dopt kind`
- Link-time optimization flags: `--dlink-time-opt` (`-dlto`)

NVCC compiler control of code optimization settings and GPU modes can also be useful to optimize the generated code:

- Flush-to-Zero (FTZ) mode: faster floating point computations can be turned on with compiler flag "`-ftz=true`"
- Fast math mode: set with "`--use_fast_math`"
- Optimizer level: use "`--optimize`" flag
- Device code optimization level: set with the "`--dopt`" flag
- Division lower precision: "`-prec-div=false`"
- Square root lower precision: "`-prec-sqrt=false`

Disable Compiler Debug Flags. The "`-g`" (host debug) and "`-G`" (device debug) compiler options to `nvcc` should be disabled for performance. The issue is not so much that the executable contains extra symbol naming information, but that these options suppress a wide variety of auto-optimizations that would otherwise be performed by the compiler. This is less the case if the flags are passed to `gcc` for the host code, but especially true for kernel code.

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 (C++20)
- `[[fallthrough]]` (C++17)
- `[[expects]]`, `[[ensures]]`, `[[assert]]` (C++20)
- `__assume` (CUDA 11.2)
- `__builtin_assume` (CUDA 11.2)
- `__builtin_assume_aligned` (CUDA 11.2)
- `__builtin_unreachable` (CUDA 11.3)

Note that all of these capabilities are available in both device kernels and host code. Various additional capabilities are available in host-only code, using the underlying compiler, such as additional GCC builtin primitives:

- `__builtin_prefetch` (control data fetching)
- `__builtin___clear_cache`
- `__builtin_alloca_with_align_and_max`
- `__attribute__((fallthrough))`
- `__attribute__(assume(expression))`
- `__attribute__(cold/hot/unused)`
- `__builtin_expect`
- `__builtin_expect_with_probability`

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 are actually of more interest than constexpr for AI development. These have been formalized in C++23, but non-standard versions have long existed. The possible benefit for C++ AI engines 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.

Optimizer-Triggered Glitches

Here's a pro tip: don't turn the optimizer on the night before you ship!

And there's a corollary, too: don't disable the "-g" and "-G" debug flags for the compiler just before you ship. That's because these flags suppress various nvcc optimizations, so if you suddenly remove them, it's the same as enabling a much higher level of optimization.

You need to run the optimizer regularly in your build, since it might shake out a few bugs. This is far less embarrassing if it's only in front of the team. In fact, it can be a desirable self-testing method to run your kernel against its unit tests and regressions with all sorts of different optimization levels enabled.

There are a large number of obscure coding errors that can be triggered by higher levels of optimization, whereas the code may run fine without optimization, or in the interactive debugger. Examples include various memory errors, which has to be top of mind, because the optimizer may be changing the way in which variables are arranged in memory, or how memory management is optimized.

Examples include the normal culprits:

- Use of uninitialized memory (e.g., from `new` or `malloc`)
- Use of already-deallocated memory (e.g., after `delete` or `free`)
- Dangling pointers
- Buffer overruns
- Doubly-deallocated memory
- Deallocating non-allocated memory
- Returning the address of a local stack variable or buffer.

Less commonly, there are also various arithmetic oddities that are not technically certain in the C++ language, and may be changed subtly by optimization. Some examples to consider if you cannot find a memory problem include:

- Integer division or modulus by negative integers.
- Right-bitshift on a signed negative integer.
- Order of evaluation undefined behavior (e.g., a side-effect like ++ on one operand of a binary arithmetic operator, whereby the order matters, or the modified variable is used twice in the one expression).

So, if the code suddenly breaks when you turn up the optimization level, it's more likely to be a latent bug in your C++ code, rather than an optimizer bug. Nobody to blame but yourself!

Compiler Auto-Optimization

Your compiler is trying to do its best. If it sees you code "3+5" then it's going to change that to "8" as it's spinning along. That's called "constant folding" and is one of the many techniques that the "optimizer" part of your compiler uses.

If you write "`sqrt(3.14)`" then the compiler could pre-compute that, too. This is going beyond constant folding to precomputing all constant expressions at compile-time, based on `constexpr` and related specifiers. I'm not sure that all C++ compilers do this yet, but the state of the art is continually improving.

What optimization techniques does your compiler use automatically? Here is an overview of some of the various methods whereby compilers emit better low-level instructions during their code generation phase. There is an extensive body of research on how compilers can auto-optimize your code.

Some of the major auto-optimization techniques include:

- Constant folding
- Constant propagation
- Operator strength reduction
- Algebraic identities in expressions
- Compile-time expression evaluation
- Compile-time function execution
- Common subexpression elimination
- Dead code elimination (unreachable code)
- Loop optimizations (unrolling, hoisting)

Constant Folding

Constant folding is the name of the optimization where the compiler automatically merges constant expressions at compile-time during code generation. For example, assume your code has:

```
    x = 3 + 4;
```

The compiler should "fold" the two constants (3 and 4) by doing the "+" operation at compile-time, and then automatically generate code without the "+" operation. Effectively, it should execute as if you'd written:

```
    x = 7;
```

So, how good are the compilers? The answer is that they're getting pretty amazing, but it's not always clear. Here's some cases about constant folding:

- `sizeof` is a constant. Any use of the `sizeof` operator in expressions should be treated like an integer. If your code says "`8*sizeof(int)`", then it's hopefully folded to give "32" at compile-time.
- Named constants. If you declare "`const int n=3`", then hopefully all subsequent uses of "`n`" will be folded as if they said "3".
- Named macro constants. This is trivially handled.
- Type casts: Hopefully your compiler should propagate types correctly while doing constant folding.

What about some harder cases? Consider this code with an intrinsic math function:

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

Can the compiler fold this at compile-time? Surely it should do "2.0f*3.14159f" correctly. But what about the sqrtf calculation? It's theoretically possible, but I'm far from certain. I'd be inclined to declare it as a global constant, or a local "static" constant (host code), so as to ensure it only gets pre-calculated at most once:

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

Now, since it's declared as "static" (and "const"), hopefully the executable code would only compute it once at program startup, even if it wasn't fully folded. Another faster alternative that ensures it's not calculated even once is to work out the formula yourself, and put in the result as a hard-coded floating-point constant.

```
   // sqrtf(2.0f * 3.14159f);
   static const float scalefactor = 2.506627216f;
```

Constant Propagation

Constant propagation is the compiler optimization whereby constants are "propagated" through expressions and control flow. This is a longstanding area of research and the technique is done by many compilers. For the basic idea of constant propagation, consider this code:

```
    x = 3;
    y = x;
```

The aim is to "propagate" the constant "3" through expressions for faster code:

```
    x = 3;
    y = 3;  // Propagated
```

Constant Expression Evaluation

The idea of "constant expression evaluation" is to generalize the older techniques of constant folding and constant propagation to their logical extreme. Whenever a value can be computed at compile-time, the optimizer should attempt to do so. For example, consider the code:

```
    f = sqrtf(3.14f);
```

In theory, this can be computed at compile-time, because it has a fixed value, and the return value of the sqrtf function only depends on its input.

If the compiler is advanced enough, it can "interpret" or "evaluate" the value with the `sqrtf` function from the constant argument (3.14f), and replace this function call with the numeric result.

The "`constexpr`" C++ directive is one manner in which programmers attempt to guide the compiler as to what expressions or functions it can compute at compile-time. If you look in the system header files, you'll certainly see that `sqrtf` function is declared as having property "`constexpr`". Whether the compiler actually evaluates this at compile-time is another matter, but the capabilities of optimizers are moving very rapidly.

The `constexpr` property can be applied to your own declared functions. In theory, the compiler could then evaluate at compile-time any calls to your function that are passing in a constant value. This becomes constant propagation on steroids.

Algebraic Identities

The compiler knows about all sorts of algebra! 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 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)
```

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) {
        // ...
    }
```

7. Timing CUDA C++ Programs

CUDA C++ Timers

You can embed timing code into your CUDA C++ applications. Usually, you do this in the host code, rather than the device code, because you're trying to time the overall performance of your parallelization optimizations, rather than just one GPU thread.

There are various different C++ code libraries for tracking and recording time information. Here are some of them:

- `clock` standard C++ function (in <time.h)
- CUDA event timers
- `time` standard C++ function (in <time.h)

The clock function

The C++ `clock` function records "clock ticks" at a microsecond level of granularity. The basic method of use is to wrap your code between two calls to the `clock` function:

```
#include <time.h>  // clock()
   ...
clock_t before = clock();  // clock ticks
// Launch the kernels here...
cudaDeviceSynchronize();  // Wait for kernel completion
clock_t after = clock();  // clock ticks

double secs = (after - before) / (double) CLOCKS_PER_SEC;
double usecs = secs * 1000000.0;   // Microseconds
```

CUDA Event Timers

CUDA C++ has a builtin set of "event timers" that track and record timing events at a millisecond granularity. The basic method of use is also a "before-and-after" wrapper around the code.

```
 // Set up the timer objects
cudaEvent_t event_before;
cudaEventCreate(&event_before);
cudaEvent_t event_after;
cudaEventCreate(&event_after);

// Wrap the kernel
cudaEventRecord(event_before);
// Launch the kernels here...
cudaDeviceSynchronize();  // Wait for kernel completion
cudaEventRecord(event_after);

cudaEventSynchronize(event_after); // Wait for "after" event
float event_ms;  // milliseconds
cudaEventElapsedTime(&event_ms, event_before, event_after);
float event_secs = event_ms / 1000.0f;
```

Note that no `#include` is needed for CUDA timers, because these are part of the CUDA runtime API. The `nvcc` compiler auto-includes this header file.

How Is This CUDA Kernel Even Fast?

Looking at a basic CUDA kernel, it's hard to believe it could be fast. Here's an example that just clears a vector in an element-by-element parallel kernel:

```
__global__ void aussie_clearvec_basic(float* v, int n)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < n)
        v[id] = 0.0;  // Clear vector element
}
```

The grand unified theory of CUDA states that this kernel code is fast, because it gets split into warps of 32 threads, which are then spread over blocks of multiple warps. The idea is that we get lots of threads all running in lock-step parallel, each clearing one vector element, so the whole vector gets cleared in one big parallel step.

But there's a lot of statements! Let's count them out:

- Kernel function argument setup: `v` and `n`
- Builtin CUDA object setup: `blockIdx`, `threadIdx`, `blockDim`.
- Structure field access: `.x` (thrice)
- Multiply: `blockIdx.x * blockDim.x`
- Add: `+ threadIdx.x`
- Assignment: `id =`
- Less-than: `id < n`
- Conditional branching: `if`
- Array index: `v[id]`
- Assignment (floating point): `= 0.0`

I'm up to 10 distinct operations now, but every time I look at this code, I see another one. How in the name of Edsger Dijstra is that even close to fast?

And what about the multiplication operation? Every AI developer runs screaming at the sight of an asterisk, and there's a whole stream of research papers on "zero multiplication" AI models. And yet, here we are wasting an entire multiplication operation just to compute an index.

What gives?

Questions. This analysis of all the operations triggers a bunch of questions about optimizing this CUDA C++ code sequence:

- Would the kernel be 10% faster if I chose `N` as a value without extra cases, so I can remove the safety "`if(id < n)`" statement?
- Would it be faster if I make `N` a `const` or a `#define` rather than passing it in?
- Or should I use template specialization so that `N` is constant?
- If I removed the "`id`" temporary variable, would it be faster?
- Should I replace `blockDim.x` with a constant? Or some kind of pre-computed global variable?
- Is that multiplication operation expensive? i.e., `blockIdx.x * blockDim.x`

Short answer: no, to all.

The longer answer is that global memory access cost dwarfs all other statements. For example, in a V100 GPU, it costs 400-800 GPU clock cycles to access global memory. The cost to write a value to the global array, `v[id]`, is hundreds of GPU clock cycles, but the other operations are one cycle, or at most single-digit cycles.

Furthermore, the compiler is auto-optimizing these things. The `id` temporary variable is trivial for the compiler to auto-remove, so it's a register operation.

Even the multiplication is probably not a real multiplication. The expression "`blockIdx.x * blockDim.x + threadIdx.x`" is such a standard idiom in CUDA C++ that the compiler will surely auto-optimize it to some much faster builtin operations. We can look at the PTX assembly to see what's really going on.

And one final point: yes, there is a CUDA runtime function called `cudaMemset`, which would clear my vector in a much faster way.

Timing The Kernels

I decided to try timing some simple kernels while rearranging some of these internal operations. The findings were basically — spoiler alert! — that the main cost was the global memory accesses, but that there was a lot of overhead cost in launching lots of blocks. However, it wasn't clear in some ways, so let's time four kernels:

- Basic kernel (all operations)
- No assignment (all operations except the global memory assignment)
- Do nothing (no operations)
- Two assignments (two global memory assignments)

Here's the timing code:

```
    #include <time.h>  // clock()
       ...
    clock_t before = clock();  // clock ticks
    for (int i = 0; i < repeats; i++) { // Repeats for timing
        kernelfnptr<<< blocks, threads_per_block>>>(dev_v, n);
        cudaDeviceSynchronize();  // Wait for whole kernel...
    }
    clock_t after = clock();  // clock ticks
    double secs = (after - before) / (double) CLOCKS_PER_SEC;
    double usecs = secs * 1000000 / (double)repeats;
    fprintf(stderr, "TIMING: %s: %d iterations of N=%d
blocks=%d/%d took %3.2f seconds, us/kernel=%3.2f\n",
           name_of_kernel, (int)repeats, (int)n, blocks,
           threads_per_block, secs, usecs);
```

Note that the timing code has to call `cudaDeviceSynchronize`, because kernel launches are asynchronous. Without this explicit synchronization, we'd just be timing kernel launch time, rather than kernel execution time. Note that this does not measure the time cost of any other overheads before and after kernel launch, such as memory allocation, memory copies, and memory de-allocations.

Here's the basic kernel code:

```
__global__ void aussie_clearvec_basic(float* v, int n)
{
    // Kernel runs on GPU -- safely!
    int id = blockIdx.x* blockDim.x + threadIdx.x;
    if (id < n) { // Safety
        v[id] = 0.0;  // Clear one element..
    }
}
```

Here's the change to the kernel without any assignment:

```
    // Commented out // v[id] = 0.0;
```

And here's the "two assignment" kernel with a few tricks to try to stop the compiler optimizing too much:

```
__global__
void aussie_clearvec_two_assign(float* v, int n)
{
    volatile int id = blockIdx.x* blockDim.x
                               + threadIdx.x;
    if (id < n) { // Safety
        v[id] = 0.0;  // Clear one element..
        v[id + ((unsigned)n>>30)] = 0.0;  // Two
    }
}
```

You can probably guess the "do nothing" kernel code:

```
__global__
void aussie_clearvec_do_nothing(float* v, int n)
{
}
```

Timing Results

I used these settings:

```
   Repeats = 100,000
   N=2,097,152
   blocks=2048
   threads-per-block=1024
```

The results varied significantly with block size and vector size. Generally, I had to make block size at least 512 threads and use `N` of at least a million vector elements before significant divergence in timing arose. With block size of 32 threads and small `N`, there was almost no difference in any of the four kernels.

Here are the final results for the above settings:

```
TIMING: Basic: 100000 iterations of N=2097152
blocks=2048/1024 took 4.30 seconds, us/kernel=42.96
TIMING: No Assign: 100000 iterations of N=2097152
blocks=2048/1024 took 1.62 seconds, us/kernel=16.23
TIMING: Do nothing: 100000 iterations of N=2097152
blocks=2048/1024 took 1.62 seconds, us/kernel=16.19
TIMING: Two assign: 100000 iterations of N=2097152
blocks=2048/1024 took 4.24 seconds, us/kernel=42.43
```

If we look at these results, here are some conclusions:

- The assignment to global memory was 62% of the cost, with 38% overhead cost.
- Two assignments to global memory were the same (!) as one assignment (was one optimized away?).
- The non-assignment code in the kernel, such as index calculations, had almost zero cost (because the "no assignment" and "do nothing" versions cost almost exactly the same).
- The 38% overhead must be from something other than kernel instructions or global memory assignments (i.e., overhead from launching blocks and threads).

PTX Assembly Analysis

To confirm that the optimizer was not removing too many of the statements, I had a look at the PTX assembly. The command to "keep" these intermediate files using `nvcc` on Linux was:

```
    nvcc -keep -keep-dir=. aussie-time-vector-kernels1.cu
```

The files that were created by this are here, using the command "`ls -la`" in Google Colab (truncated/edited):

```
root     976 Sep 28 10:38 a_dlink.fatbin
root    3256 Sep 28 10:38 a_dlink.fatbin.c
root    2936 Sep 28 10:38 a_dlink.o
root      32 Sep 28 10:38 a_dlink.reg.c
root     896 Sep 28 10:38 a_dlink.sm_52.cubin
root  976576 Sep 28 10:38 a.out*
root 1249860 Sep 28 10:38 aussie-time-vector-kernels1.cpp1.ii
root 1146101 Sep 28 10:38 aussie-time-vector-kernels1.cpp4.ii
root    7767 Sep 28 10:33 aussie-time-vector-kernels1.cu
root     386 Sep 28 10:38 aussie-time-vector-kernels1.cudafe1.c
root 1066055 Sep 28 10:38 aussie-time-vector-kernels1.cudafe1.cpp
root   17165 Sep 28 10:38 aussie-time-vector-kernels1.cudafe1.gpu
root    4080 Sep 28 10:38 aussie-time-vector-kernels1.cudafe1.stub.c
root    7168 Sep 28 10:38 aussie-time-vector-kernels1.fatbin
root   19862 Sep 28 10:38 aussie-time-vector-kernels1.fatbin.c
root      52 Sep 28 10:38 aussie-time-vector-kernels1.module_id
root   23312 Sep 28 10:38 aussie-time-vector-kernels1.o
root    2693 Sep 28 10:38 aussie-time-vector-kernels1.ptx
root    6248 Sep 28 10:38 aussie-time-vector-kernels1.sm_52.cubin
root    4096 Sep 25 18:24 .config/
root    4096 Sep 25 18:24 sample_data/
```

To understand what is going on here, you need to know the `nvcc` compiler does two separate things (and does a good job of hiding this, unless you use the "`-keep`" option):

- Device code is compiled to PTX assembly ("`.ptx`" file) while is them assembled to binary formats.
- Host code is source-to-source compiled to standard C++ (e.g., the "`.ii`" and "`.c`" files) which is then sent to the native C++ compiler (e.g., `g++` on Linux)

All of the PTX versions of the kernels are in the one ".ptx" file. Here's the PTX for the "do nothing" kernel:

```
    // .globl        _Z37aussie_clear_vector_kernel_do_nothingPfi
    .visible .entry _Z37aussie_clear_vector_kernel_do_nothingPfi(
        .param .u64 _Z37aussie_clear_vector_kernel_do_nothingPfi_param_0,
        .param .u32 _Z37aussie_clear_vector_kernel_do_nothingPfi_param_1
    )
    {
        ret;
    }
```

Here's the longer PTX for the "basic" kernel, without any optimizations (abridged):

```
    // .globl        _Z32aussie_clear_vector_kernel_basicPfi
    // … (removed)
    {
        .reg .pred      %p<2>;
        .reg .b32       %r<7>;
        .reg .b64       %rd<5>;

        ld.param.u64    %rd1,
[_Z32aussie_clear_vector_kernel_basicPfi_param_0];
        ld.param.u32    %r2,
[_Z32aussie_clear_vector_kernel_basicPfi_param_1];
        mov.u32         %r3, %ctaid.x;
        mov.u32         %r4, %ntid.x;
        mov.u32         %r5, %tid.x;
        mad.lo.s32      %r1, %r3, %r4, %r5;
        setp.ge.s32     %p1, %r1, %r2;
        @%p1 bra        $L__BB0_2;

        cvta.to.global.u64          %rd2, %rd1;
        mul.wide.s32    %rd3, %r1, 4;
        add.s64         %rd4, %rd2, %rd3;
        mov.u32         %r6, 0;
        st.global.u32   [%rd4], %r6;
    $L__BB0_2:
        ret;
    }
```

Note that the main global memory assignment is the `st.global` "store global" instruction.

Let’s annotate some of this PTX gobbledegook to clarify the executable instructions in these assembly statements:

```
        // Load parameters: RD1=void*v, R2 = int n
        ld.param.u64        %rd1, 
[_Z32aussie_clear_vector_kernel_basicPfi_param_0];
        ld.param.u32        %r2, 
[_Z32aussie_clear_vector_kernel_basicPfi_param_1];

        // C++: int id = blockIdx.x* blockDim.x + threadIdx.x;
        mov.u32         %r3, %ctaid.x;  // R3 = blockIdx.x
        mov.u32         %r4, %ntid.x;   // R4 = blockDim.x
        mov.u32         %r5, %tid.x;    // R5 = threadIdx.x
        mad.lo.s32      %r1, %r3, %r4, %r5; // R1=(R3*R4)+R5 (mult-add)

        // C++: if (id < n) -- The PTX does: !(id >= n)
        setp.ge.s32     %p1, %r1, %r2; // greater-equal: R1 (id)>=R2(n)
        @%p1 bra         $L__BB0_2;    // Branch if true

        // C++: v[id] = 0.0; (global assignment)
        cvta.to.global.u64        %rd2, %rd1; // RD2=RD1 (v)
        mul.wide.s32        %rd3, %r1, 4; // RD3 = R1 * 4 (id * 4 bytes)
        add.s64         %rd4, %rd2, %rd3; // RD4 &v[id]=RD2(v)+RD3 (id*4)

        mov.u32         %r6, 0;      // R6 = 0
        st.global.u32   [%rd4], %r6; // *RD4 = R6 (Store: v[id*4]=0)
$L__BB0_2:    // Branch label for if
        ret;  // Return
```

Here is the “no assignment” version in PTX:

```
    // .globl  _Z40aussie_clear_vector_kernel_no_assignmentPfi
    // … etc
    {
        .reg .b32    %r<7>;
        .reg .b64    %rd<3>;

        mov.u32      %r1, %ctaid.x;
        mov.u32      %r2, %ntid.x;
        mov.u32      %r3, %tid.x;
        mad.lo.s32   %r4, %r1, %r2, %r3;
        mov.u32      %r6, %r4;
        mov.u32      %r5, %r6;
        ret;
    }
```

Note that it does indeed have no assignment, since the `st.global` “store global” instruction is gone. However, the `if` statement is also gone, with no greater-equal test, no branch instruction and no branch label, since the compiler noticed that it was empty. However, not everything has been optimized away, and it’s still doing more arithmetic instructions in registers than the “do nothing” kernel.

Finally, here is the PTX of the "two assignment" version:

```
    // .globl _Z37aussie_clear_vector_kernel_two_assignPfi
    // … etc.
    {
        .reg .pred      %p<2>;
        .reg .b32       %r<13>;
        .reg .b64       %rd<9>;

        ld.param.u64    %rd2,
[_Z37aussie_clear_vector_kernel_two_assignPfi_param_0];
        ld.param.u32    %r1,
[_Z37aussie_clear_vector_kernel_two_assignPfi_param_1];
        mov.u32         %r2, %ntid.x;
        mov.u32         %r3, %ctaid.x;
        mov.u32         %r4, %tid.x;
        mad.lo.s32      %r5, %r3, %r2, %r4;
        mov.u32         %r12, %r5;
        mov.u32         %r6, %r12;
        setp.ge.s32     %p1, %r6, %r1;
        @%p1 bra        $L__BB1_2;

        cvta.to.global.u64     %rd4, %rd2;
        mov.u32         %r7, %r12;
        mul.wide.s32    %rd5, %r7, 4;
        add.s64         %rd6, %rd4, %rd5;
        mov.u32         %r8, 0;       // R8 =0
        st.global.u32   [%rd6], %r8;  // Store global v[0]

        // C++: v[id + ((unsigned)n>>30)] = 0.0;
        shr.u32         %r9, %r1, 30;   // Shift-right: R9=n>>30
        mov.u32         %r10, %r12;     // R10 = R12 (id)
        add.s32         %r11, %r10, %r9;// R11 = R10 (id) + R9
        mul.wide.u32    %rd7, %r11, 4;  // RD7 = R11 * 4 bytes
        add.s64         %rd8, %rd4, %rd7; // RD8 = v + (id*4)
        st.global.u32   [%rd8], %r8;    // Store global v[RD8]=R8
    $L__BB1_2:
        ret;
    }
```

Note that there are two "`st.global`" instructions, so it is doing the second global memory assignment. Hence, we can conclude that, no, the second assignment was not optimized away at the instruction level, so it must have been sorted at some other hardware level. For example, a low-level hardware pipeline of global memory assignments for the threads noticed that the second one was redundant, or did them both in parallel.

8. Memory Optimizations

Memory Optimization Techniques

Memory processing is often an important part of optimizing CUDA C++ kernels. The costs of memory arise from:

- Memory access costs
- Memory transfer costs

Here are some of the main techniques:

- Different types of memory
- Reduce overall memory usage
- Coalescing and striding memory access patterns
- Caching and data locality
- Overlapping data transfers and computation
- Avoid memory access contention
- Optimize allocated memory

Let's examine them in more detail below.

CUDA Memory Hierarchy

The GPU has several layers of memory storage with different speeds and sizes. Hence, the simple idea for CUDA optimization: *Use the fastest type of memory!*

The hierarchy of CUDA memory types, from fastest to slowest, includes:

- Scalar constants and numerics (no memory)
- Registers
- L1 and L2 caches
- Local memory (including the stack)
- Shared memory (block scope)
- Constant memory (global, but read-only)
- Global memory

Unified Memory. Note that CUDA's capability for Unified Memory is not a distinct type of physical memory. This refers to the software capability whereby CUDA allows both the host and device to access the memory address space, but have it managed behind-the-scenes by the CUDA Runtime and the GPU hardware. This is beneficial to both performance and programmer productivity by allowing automatic optimization of certain types of memory transfers and data flow in an application.

Warp memory? Note also that there's not really any warp-specific type of memory. The famous "warp shuffle" CUDA intrinsics do not actually have a separate memory mechanism in the normal sense but are actually a register-to-register transfer of data between threads in the warp. This is further based on various criteria such as which threads are participating within the warp and what arithmetic operation is to be performed by the shuffle. These warp-scope operations are a way to avoid using shared memory via warp-wide data exchange, but they don't use a distinct warp-only type of memory.

Numeric Constants. Constant scalar values such as number constants and named symbolic constants in CUDA C++ are very efficient. These are examples:

```
    #define MYCONST 32
    const int myconst = 32;
```

These constants do not consume memory as they are not really stored anywhere. You can see them in the PTX assembly listings (using the "`-keep`" compiler option), so they are "stored" in the instruction code as operands for the GPU hardware machine-code instructions. However, larger constants such as string literals are stored in constant memory.

This means that if you can make anything a constant, you should absolutely do so. Rather surprisingly, the best example of using constant numbers is actually AI and the LLMs. All of the LLM's data and their inference engines have a fixed size, known at compile-time, for all of their matrices and vectors. These values can be used as constants! Furthermore, putting constant values into the code allows the auto-optimizer in the compiler to do a great many additional optimizations (e.g., constant folding, constant propagation, constant expression evaluation). Hence, CUDA C++ programmers for AI engine backends should use constants for these values wherever possible in the engine code, rather than trying to generalize the code for all sizes.

Alas, most variables cannot be made into constants, but must be stored in some type of memory. Memory optimizations apply to any of these various levels of memory. Let's examine each of the types of memory in turn.

Registers

The compiler does a great job of putting all of the simple variables in a CUDA C++ kernel into registers. The way to declare a variable in a register is simply:

```
int x;
```

Hence, all the typical computations of the index, and the builtin variables like `blockIdx` and `threadIdx`, are all stored in GPU registers. You can check this by examining the PTX assembly code using the "`-keep`" compiler option.

Registers are the fastest type of memory on the GPU, and are specific to a thread. However, there are only a limited number of registers available, and a "register spill" is where variables exceed registers, and must be stored in local memory on the stack.

You can review the register limits by calling `cudaGetDeviceProperties` using the `regsPerBlock` property, such as:

```
// Registers for whole GPU
int device_number = 0;
cudaDeviceProp prop;
CUDACHK( cudaGetDeviceProperties(&prop, device_number));
int kreg = (int)prop.regsPerBlock / 1024;
printf("Max Registers (whole GPU): %d registers (%dK)\n",
       (int)prop.regsPerBlock, kreg);
```

Technically, this is the "maximum" registers a block can use, but if you used this many, there'd be no other registers for other workloads. The registers are shared by all blocks running on a multiprocessor. Hence, you need to manage the registers across all your kernels, and any other workloads running on the same GPU.

Avoiding Register Spills

Methods to reduce the number of registers, and reduce the likelihood of register spills into local memory, include:

- Minimize temporary variable usage (e.g., in index or stride calculations)
- Minimize kernel function parameters
- Limit the scope and lifetime of any local variables
- Be careful with loop unrolling
- Use compile-time optimizations (e.g., `const`, `constexpr`, etc.)

If you want to know about registers, there's a "-v" option to the ptxas PTX assembler that can tell you about register access patterns (and also shared memory). The option you need to add to the nvcc compiler is:

```
    --ptxas-options=-v
```

There's some irony here: the original versions of C and C++ languages had a "register" keyword, that allowed specification of which local variables should be stored in CPU registers. This was long before GPUs even existed. But it's no longer in the official C++ language, having been deprecated in C++11 and fully removed in C++14, because the compiler became better than humans at deciding what CPU registers to use. And yet, here we are talking about GPU register allocation. CUDA programmers want the register keyword back!

Local Memory and the Stack

Each thread on the GPU has a limited amount of local memory, which maintains the thread's execution stack and local variables. This is not as efficient as registers, but is more efficient that shared memory or global memory. You can simply declare local variables in CUDA C++ device functions, which are automatically allocated to either registers or local memory.

The size of the local memory for the "stack" is a limit property of the GPU that you can check (and set). Here's an example of printing it out:

```
    // Get stack size
    CUDACHK( cudaDeviceGetLimit(&val, cudaLimitStackSize));
    assert(err == cudaSuccess);
    double kb = val / ((double)1024);
    printf("Device: stack size = %d bytes (%3.2f KB)\n",
         (int)val, kb);
```

You can also set the stack size using cudaDeviceSetLimit.

Another optimization method is to use the alloca function for dynamically allocated stack memory. This memory should be faster than shared memory (or global memory), but there's only a limited amount, and it only has thread-scope, being on the stack of each thread. If you want to share stack memory between threads (rather than using shared memory), you'll have to use warp shuffle or warp reduce primitives.

Shared Memory

Shared memory is declared using the "`__shared__`" double-underscore specifier. This makes it convenient to declare and access, but not so simple to use efficiently.

There are two types of shared memory: static and dynamic. You can declare a static block of shared memory, with size known at compile-time, by using `__shared__` in a thread-level local variable declaration, such as:

```
__global__ void mykernel()
{
    __shared__ float myshared[128];  // Static
    // ...
}
```

You can declare "dynamic" amounts of shared memory as a parameter to a kernel launch, which specifies the bytes. The GPU allocates the amount of shared memory (if it's not too large), and links it with a global variable inside the kernel code. Hence, an example:

```
int sharedsz = 128 * sizeof(float);  // Bytes
mykernel<<<blocks, threads, sharedsz>>>(); // Dynamic
```

Note that the above kernel launch syntax does not use `__shared__`. Instead, you need to declare a device-level global variable inside your kernel code with both `__shared__` and `extern` (and without a fixed size), such as:

```
extern __shared__ float myshared[];   // Dynamic

__global__ void mykernel()
{
    // ...
}
```

Obviously, for CUDA to connect the two parts, just like in *Highlander*, there can be only one `extern` shared memory variable.

Efficiency of shared memory. Using shared memory is faster than using global memory, and it has the scope of all the threads in a block. Hence, its scope limit to a single block is more restricted than global memory, which can be accessed by all blocks. Shared memory is faster than using global memory, and can be very efficient. However, warp-level shuffle or reduction primitives can often be used for even faster kernels.

Shared memory size. There is only a limited amount of shared memory available to each block. And I mean small, like I'm back in 1982 programming a *Commodore 64*, because it can be as little as 48K. Yeah, I know, that looks like a typo, where I should have written 48G. Imagine trying to sell an iPhone with only 48K.

On some of the GPUs, the shared memory and L1 cache have to split the available memory. If there's only 64K per multiprocessor, this has to be partitioned between L1 cache and shared memory (i.e., 16K L1 cache and 48K shared memory, or 16K shared memory and 48K L1 cache, or 32K-32K if you want to be trendy). You need to treat shared memory like it's gold. And speaking of gold, I actually had a *TRS-80 Co Co* that wasn't rose gold, but I digress.

You can manage this split between shared memory and L1 cache as a runtime policy. In CUDA host code, use the APIs `cudaDeviceSetCacheConfig` (all kernels) or `cudaFuncSetCacheConfig` (one kernel). I'm not going to give you a code snippet, because you can just look it up on Stack Overflow anyway. Anyway, here are some of the GPU shared memory sizes:

- B100/B200 — 128K
- A100 — 164K
- V100 — 64K

Yes, it seems that the recent B100/B200 GPUs (Blackwell) have *less* shared memory than an A100 (Ampere). I'm sure there's a good reason for that, something to do with optimizing the transistors on the blah blah blah. I'm not a hardware engineer.

Anyway, that shared memory is *per multiprocessor*, so it's not quite global across the entire GPU for everyone. You can find the shared memory size of a GPU in C++ via `cudaGetDeviceProperties` with the `sharedMemPerBlock` property. Here's an example:

```
// Shared memory per block
int device_number = 0;
cudaDeviceProp prop;
CUDACHK( cudaGetDeviceProperties(&prop, device_number) );
double sharedkb = (double)prop.sharedMemPerBlock/(1024.0f);
double sharedmeg = (double)prop.sharedMemPerBlock
                       / (1024.0f * 1024.0f);
double sharedgig = sharedmeg / 1024.0f;  // In your dreams!
printf("Shared memory: %ld bytes (%3.2f KB, %3.2f MB)\n",
    (long int)prop.sharedMemPerBlock, sharedkb, sharedmeg);
```

However, you cannot increase it! There's no `cudaSetDeviceProperties` API in CUDA, because these are a GPU's read-only hardware properties.

Bank Conflicts (Shared Memory Contention)

Shared memory can suffer performance degradation if there is contention in accesses to shared memory from multiple threads. These are called "bank conflicts" and should be avoided for better performance. Take care with the memory access patterns when using shared memory to avoid this performance pitfall.

Bank conflicts occur when multiple threads attempt to access the same address "bank" of shared memory at the same time. This means that each thread must access a separate bank, or else the accesses become serialized.

I feel like I'm in opposite world! Didn't we learn that "coalesced memory access patterns" need adjacent memory addresses in global memory. But now it's the exact reverse for shared memory? Adjacent addresses are the worst?

Nope! Wrong!

Actually, successive memory addresses map to *different banks*. That's not how I would think of "banks" and maybe they should have called it "stripes" or something involving animals (zebras?).

But that's actually good news, because it means that our optimizations whereby each thread in a warp accesses an adjacent memory address will still work. That means: grid-stride loops work for shared memory! Well, actually the "stride" won't be the grid size, but now who's being picky. So, we'll name then "zebra loops" instead or "gazelles."

Note that it won't work too well if the threads are trying to access adjacent memory addresses that are more than 32-bits (the default), such as a `double`, which is 64-bits. In this case, a thread that accesses a 64-bit address will inherently trigger a bank conflict by accessing two 32-bit locations across two banks (I mean, zebras).

Fortunately, there are the useful CUDA C++ runtime APIs to dynamically change it: `cudaDeviceSetSharedMemConfig` with the two symbolic constants for policies `cudaSharedMemBankSizeFourByte` or `cudaSharedMemBankSizeEightByte`. I'll let you figure out which is which.

Another way to avoid bank conflicts is to change from the use of shared memory to warp-level primitives. These include "warp shuffle" and "warp reduce" primitives, which are very fast, but are limited in scope to the warp (32 threads), whereas shared memory has block scope level.

Constant Memory for Read-Only Data

Constant memory is declared using the "`__constant__`" special double-underscore specifier. Another optimization for memory usage is to use the faster constant memory for read-only data. However, there is a much more limited size of constant memory compared to the read-write global memory.

Constant data can be declared at global scope in device code by adding `__constant__` to a global variable declaration:

```
    __constant__ float constarr[128];
```

In the above example, the variable is an array. Although this is the most common usage, it does not need to be of array type. In the device code, you can just access this variable by its name. Note that `__constant__` relates to the memory address and its read-only nature in the device. This specifier does not imply `const` or `constexpr` or other similar specifiers.

However, in host code you cannot use the name of the variable, because it's a device address. Instead, you use the CUDA APIs to gets its address in Unified Memory.

From the host side, constant memory can be accessed via the CUDA runtime APIs, such as `cudaMemcpyToSymbol` and `cudaMemcpyFromSymbol`. It is also possible to use `cudaGetSymbolAddress` to get the address in Unified Memory and then use `cudaMemcpy` with this address.

Interestingly, both methods can be used to modify the variable in the device memory. Umm, this means that the host can modify constant memory on the device. When is a constant not a constant?

The size of constant memory is small, as low as 64KB, even on recent GPUs. You can print out the size of the available constant memory on the GPU using the `totalConstMem` named property from `cudaGetDeviceProperties`:

```
    // Constant memory size
    int device_number = 0;
    cudaDeviceProp prop;
    CUDACHK( cudaGetDeviceProperties(&prop, device_number));
    double constkb = (double)prop.totalConstMem / (1024.0f);
    double constmeg = (double)prop.totalConstMem
                                / (1024.0f * 1024.0f);
    double constgig = constmeg / 1024.0f;
    printf("Constant memory: %ld bytes (%3.2f KB)\n",
            (long int)prop.totalConstMem, constkb);
```

Global Memory

Global memory is memory on the GPU device that can be accessed by any kernels, with an address in Unified Memory to access this variable. The usual method is `cudaMalloc` on the host to create allocated global memory on the device.

```
    float *device_ptr = NULL;
    int sz = n * sizeof(float);  // bytes
    CUDACHK( cudaMalloc((void**)&device_ptr, sz);
```

After the above code sequence, a global block of memory has been allocated on the device, and its generalized address has been passed back to the host code. The host code cannot directly examine this memory:

```
    float val = device_ptr[0];   // Fails on host
```

However, the host code can use this address in other CUDA API calls that require the device pointer, such as `cudaMemcpy` or `cudaMemset`.

The device code cannot immediately use the allocated memory in "device_ptr" for the simple reason that it doesn't have the address. It's only on the host side. For this reason, it's typical for a host-side kernel launch to pass this address to the device as a function parameter to the kernel function. An example of this type:

```
    mykernel <<< blocks, nthreads >>> (device_ptr, n);
```

Optimizing global memory. Global memory is the largest and also the slowest, so consider any algorithmic changes that can reduce the amount of memory stored globally. Use of global memory can be replaced with shared memory, local memory, constant memory, or registers where possible. In order words, use anything else!

Other optimizations such as the smaller data sizes in quantization will inherently reduce global memory usage. Note that "global" memory is not available to everyone else across the GPU, but only your application (we hope so anyway). You can print out the total amount of global memory for your GPU via the `cudaGetDeviceProperties` API with the `totalGlobalMem` property:

```
  cudaDeviceProp prop;
  CUDACHK( cudaGetDeviceProperties(&prop, device_number));
  double totalmeg =(double)prop.totalGlobalMem/(1024.0*1024.0);
  double totalgig = totalmeg / 1024.0f;
  printf("Global memory: %ld bytes (%3.2f GB)\n",
              (long int)prop.totalGlobalMem, totalgig);
```

Modern NVIDIA GPUs have many dozens of gigabytes of global memory. Here's a list of some of them:

- B100/B200 (Blackwell) — 192GB
- H100 (Hopper) — 80GB
- A100 (Ampere) — 80GB
- V100 (Volta) — 16GB or 32GB
- P100 (Pascal) — 16GB

Caching and Data Locality

Data Locality. This is a method of optimizing data cache accesses to speed up memory accesses. A good example of this speedup is tiled matrix multiplication, which does small local computations on 2-D "tiles" within a matrix.

Memory Cache Optimizations. There are multiple levels of hardware caching for memory in a GPU. Various optimizations aim to maximize cache hits, so that memory accesses are as fast as possible. Optimizations with data locality are based upon speeding up the cache by increasing the cache hit percentage.

L2 Cache Optimizations. The L2 cache is fast and one optimization is to aim to maximize its use. One possibility is to modify the kernel so as to ensure that the total amount of data being used fits in the L2 cache in its entirety.

The size of the L2 cache can be reported programmatically in CUDA C++ programs via `cudaGetDeviceProperties` and the `l2CacheSize` property:

```
int device_number = 0;
cudaDeviceProp prop;
CUDACHK( cudaGetDeviceProperties(&prop, device_number));
double l2kb = (double)prop.l2CacheSize / (1024.0f);
double l2meg = (double)prop.l2CacheSize / (1024.0f * 1024.0f);
double l2gig = l2meg / 1024.0f;
printf("L2 cache memory: %ld bytes (%3.2f KB, %3.2f MB)\n",
            (long int)prop.l2CacheSize, l2kb, l2meg);
```

Memory Prefetching. The GPU has various builtin hardware capabilities for "prefetching" data from memory. This involves making a reasonable guess as to what address will next be required by the kernel. One simple heuristic is an adjacent address to the most recently accessed memory. Thus, optimizations such as data locality and coalesced memory access patterns may benefit from this hardware optimization behind-the-scenes.

CUDA C++ has the `cudaMemPrefetchAsync` runtime API, which can be used for prefetching at a relatively high level in Unified Memory. For example, this can be used to prefetch data to be ready for a kernel. However, this is not a way to control low-level memory caches.

The GCC compiler has an intrinsic function `__builtin_prefetch` that can be used by the programmer to give hints to the data prefetching algorithm. However, this isn't currently supported in CUDA C++ device kernels, where it gives a compilation error. Hence, it is used in host code, passed through to GCC on Linux.

A complicated method of using simple C++ assignments to local variables can be used to prefetch data into registers, for fast computation in threads. The details of this approach are discussed in Wijngaart and Oh (2022), cited in the reference articles. The general method is based on using assignments to four `double` local variables of the values from four global memory locations, analogous to:

```
    double d0 = v[0];
    double d1 = v[1];
    double d2 = v[2];
    double d3 = v[3];
```

However, their method is more complicated than this, but more efficient. The kernel then uses the local variables for computations (i.e., they'll be stored in registers). As an alternative approach, the article also considers the use of shared memory for prefetching optimizations, but found problems with bank conflicts.

Tiled Memory Access Algorithms. Tiling is operating on a small subset of the data space, such as a two-dimensional "tile" (or a three-dimensional "block" in tensors). The gain from "tiling" an algorithm is from increased data locality and reduced storage of temporary data, which makes much better cache utilization. The classic example is tiled matrix multiplication algorithms, but many kernels can benefit from this approach.

Memory-Aware AI Attention Algorithms. The self-attention module in AI engines is known to be memory-bound. Hence, there are several cutting edge algorithms to reduce memory accesses during attention computations, including:

- Multi-Query Attention (MQA)
- Group Query Attention (GQA)
- Flash Attention (already with versions 1, 2, and 3)
- Paged Attention

There is also the combination: Paged Flash Attention.

Reverse Block Processing. There are some types of algorithms whereby better cache utilization occurs by having the blocks process the data in reverse order. However, these are the exception, not the rule, and reverse accesses can sometimes worsen performance by undermining data prefetch caches.

Memory Size Reduction

If you've got a lot of data, it's going to consume a lot of memory, especially on the GPU. There are various ways to reduce the amount of memory needed to store the information.

Smaller Data Sizes (Quantization). Using smaller data sizes is cheaper to compute and also uses less memory. This is best known in "quantization" of LLMs from 32-bit floating-point to smaller sizes such as 8-bit integers or even 4-bit integers. If the data is smaller, both the access and transfer costs are lower (and also computation costs).

Are you using CUDA C++ for an LLM backend with quantization? Generally speaking, the following notes apply to quantization levels in AI applications, when reducing FP32 (32-bit floating-point) to smaller data types in the LLM:

- FP16 quantization (16-bit floating point) — this is standard, with 50% memory gain and minimal decline in accuracy.
- INT16 quantization (16-bit integers) — also very effective and accurate, plus allowing integer arithmetic.
- INT8 quantization (8-bit integers) — very commonly used for a four-fold memory reduction with a slight accuracy loss.
- INT4 quantization (4-bit nibbles) — surprisingly, this is widely used for an eight-fold size reduction and acceptable accuracy loss in many applications.
- Binary quantization (1-bit) — very fast because it can be implemented with addition replacing multiplication, but is generally regarded as having too large accuracy degradation.
- Ternary quantization (1.5-bit) — like binary, this is very fast via addition, but inaccurate.
- INT32 quantization (32-bit integer) — not widely used, as offers no memory gain, and integer multiplication is not much faster than floating-point multiplication on modern hardware.

These are the main ones in industry practice, but research has numerous other options, such as FP8 and FP4 (low-bit floating-point), and also other integer sizes such as 2-bit (INT2), or weird bit sizes like, 3-bit, 5-bit, etc. There does seem to be some promise to techniques involving FP8, since some GPUs now support FP8.

Also deserving more attention is INT2 quantization, which can be implemented as additions (one or two), and is more accurate than binary or ternary quantization. Finally, here's a weird wrinkle: sometimes floating-point addition is *slower* than floating-point multiplication, because of IEEE 754 oddities.

Reduce Object Sizes. If you're using any `struct` or `class` objects in your kernels, there are some ways to play with their in-memory size. The first point: print it out! Use the `sizeof` operator to know which objects are large, and also whether your changes make any difference.

The various techniques to get smaller structures and objects in C++ include:

- Biggest data members first (rule-of-thumb is largest to smallest).
- Try different orders of data members (i.e., consider "packing" and alignment issues).
- Bitfields (use type `unsigned` so you don't need a sign bit).
- Small types: `bool` and `enum` versus `int`, `short` or `unsigned char`.
- Unions (if you must)

Note that these changes can affect speed. Bitfields have runtime cost for packing and unpacking. The first object in a class is often fastest to access, because its offset is zero (optimized away). Hence, moving the largest data member first may interfere with the "most frequently used data member first" speed optimization. Also, fiddling with ordering can affect data locality, either for better or for worse.

Advanced Memory Optimization Techniques

Warp Shuffle and Warp Reduction. Algorithms can avoid using shared memory by using the much faster warp data-sharing intrinsics. There are various types, including warp shuffle primitives and horizontal reductions with warp reduction intrinsics.

Memory alignment. There are several cases where aligned memory accesses are more efficient than non-aligned addresses. This is now less important in advanced GPUs than in older architectures, but is still a consideration. Generally, accessing memory via a non-aligned address does not cause a crash or any error, but can slow down the request. This was once a serious efficiency problem, but is less important when optimizing on the more recent GPU architectures.

Texture and Surface Memory Optimizations. Although originally designed for graphics processing, the capabilities of texture memory can be used to optimize any algorithm. The advantages of texture memory include that it is cached and does not suffer from poor performance for non-coalesced access patterns. If your algorithm requires more unpredictable access to data, this may be a good option rather than non-coalesced access to other types of memory.

Kernel Fusion. This optimization merges two separate kernels into one kernel. This does not usually reduce computation, because the arithmetic from both kernels must still be performed. However, kernel fusion reduces memory accesses by avoiding the storing and re-loading of any temporary data that is created by the first kernel and used by the second kernel. Hence, the total cost of memory access is significantly reduced.

References

1. Yuan Lin and Vinod Grover, Jan 15, 2018, *Using CUDA Warp-Level Primitives,* NVIDIA Technical Blog, https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/
2. Rob Van der Wijngaart and Fred Oh, Mar 23, 2022, *Boosting Application Performance with GPU Memory Prefetching*, NVIDIA Technical Blog, https://developer.nvidia.com/blog/boosting-application-performance-with-gpu-memory-prefetching/
3. Mark Harris, Jan 28, 2013, *Using Shared Memory in CUDA C/C++*, NVIDIA Technical Blog, https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

9. Coalescing and Striding

Memory Access Patterns

In addition to choosing the fastest type of memory, it also matters in what order the kernel accesses data. These are called "memory access patterns" and include issues such as data locality and coalescing.

Contiguous Memory Blocks. Many CUDA kernels perform better when operating on contiguous blocks of memory, usually via accessing adjacent memory addresses within a warp. Generally, attempt to keep any data that you have in a way that it stays together, rather than needing to be accessed randomly. Some of the problem areas in this regard include:

- Multidimensional arrays (i.e., matrices and tensors).
- Hierarchical data structures (e.g., trees and tries).
- Permutation arrays (e.g., sparse representations using permutations).
- Lookup tables (e.g., precomputations of activation functions).

Linearize Multi-Dimensional Data. Don't store matrices or tensors in a complex multi-indirection layout with pointers. Instead, CUDA prefers having all of the data in one contiguous linearized array, where you have to manually compute the index offset when doing 2D or 3D accesses. This helps GPU parallelization by achieving both data locality and coalesced memory accesses.

Structure-of-Arrays (SoA). To achieve contiguous storage of data in arrays, it is recommended to use the Structure-of-Arrays (SoA) storage structure, rather than the Array-of-Structures (AoS) layout. This helps optimizations that linearize complex data and use contiguous access for data locality and coalescing.

Coalesced Memory Access Patterns. This technique involves having all threads in a warp or block accessing memory locations that are adjacent to each other. Coalesced memory access patterns are optimized in the GPU hardware, and allow the warp to access an entire contiguous block of memory in parallel. This is faster than accessing memory with intervals between the locations. Storing multi-dimensional data in a linearized format, and preferring Structure-of-Arrays (SoA) over Arrays-of-Structures (AoS) are two ways to gain coalesced accesses. Non-coalesced memory usage is also called a "scattered" memory access pattern.

Grid-Stride Loops. The performance advantage of grid-stride loops is that they achieve coalesced memory access patterns, which are efficient across a full warp. Another advantage is that they are flexible to any number of blocks and threads, so they allow optimization of occupancy rates and wave management without needing to alter the internal methods of the kernel.

This type of loop is also safe from array bounds violations and simple to debug, because it also works in a fully sequential mode with a single thread, in which case it has a stride of one. Although best known for use in linear kernels (e.g., vector operations), there are generalizations of grid-stride loops to two-dimensional and three-dimensional kernels, which is similar to tiling.

Avoid Scatter and Gather. These memory access patterns are the polar opposite to coalescing, so it's little surprise that they would run slow. Try to avoid using scatter/gather methods in your algorithms wherever possible.

Coalesced Data Access

An important way to optimize CUDA kernel data access costs is to use a "coalesced" pattern of accessing chunks of data in global memory. Coalesced data accesses are parallel accesses to contiguous memory addresses. In other words, this means to access contiguous memory with a group of threads, all at the same time.

The basic idea is that:

- Each thread accesses an adjacent memory address
- None of the threads access the same address (no contention)

For a warp of 32 threads, it would access 32 adjacent memory addresses in a contiguous block, with each thread accessing a separate address. These coalesced accesses to global memory are faster in a GPU than random accesses to global memory. Kernels run faster when the memory being accessed is together.

Note that is similar to the CPU speedup of accessing continuous memory blocks in sequence, because of memory cache prefetch optimizations. However, that refers to sequential CPU logic, which we have totally forgotten now that we've got a better parallel processor (called a GPU).

What does coalesced memory accesses look like a kernel? Nothing exciting, let me assure you. In fact, they're very basic and unexciting:

```
__global__ void aussie_clearvec_coalesced(float* v, int n)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < n) {
            v[id] = 0.0;  // Clear one vector element..
    }
}
```

Why is this coalesced? Because every thread accesses a different element of the vector v, in global memory. For simplicity, let's assume that n==128 and we're running 2 blocks of 64 threads each (i.e., 2 warps of 32 threads in each block). When it runs, all 128 threads run in parallel, all in lock-step, and they all process v[id] at exactly the same time. The first warp has 32 threads that access indices 0..31, the second warp is also in the first block, and it accesses 32..63, and the next two warps in the second block access array elements 64..95 and 96..127.

These accesses are all coalesced. Each warp of 32 threads accesses all 32 of the contiguous memory elements, all in parallel.

But that's just an ordinary CUDA kernel, I hear you say. That's correct, and the basic thread model of kernels tends to use coalesced memory accesses, unless we try hard not to.

Non-Coalesced Memory Accesses

Let's look at a different kernel, where each thread is clearing 4 elements of the vector, rather than just one. This is not usually a good plan, unless our vectors are so large that we can't run enough threads to do each element in a separate thread. Hence, here's the code:

```
__global__
void aussie_clearvec_non_coalesced(float* v, int n)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    id *= 4;
    if (id < n) {
        v[id] = 0.0;  // Clear 4 vector elements..
        v[id + 1] = 0.0;
        v[id + 2] = 0.0;
        v[id + 3] = 0.0;
    }
}
```

Now, if you've got your old CPU sequential C++ programming hat on, this code looks better. It looks like a loop unrolling optimization that does 4 assignments in a row, and should run a lot faster.

Not at all! Everything is upside-down in parallel world (which is why Australians like it), because this kernel is actually:

- Slower, in parallel, and
- Non-coalesced memory accesses

Because each kernel thread is clearing 4 elements, we actually need to run 4 times fewer threads so it's much less parallel. And each kernel is doing 4 accesses to global memory, one after another, in an apparently sequential algorithm. This is much less parallelization, except to the extent that the NVCC compiler auto-parallelizes anything. We really should expect this kernel to run 4 times slower.

Furthermore, it's not a coalesced memory access pattern. Let's just examine the first warp of 32 threads. These will set `id` variable to 0...31, and then it's quadrupled to have values of 0, 4, ..., 124 (31*4). This seems superficially like it might be a coalesced memory pattern, since the 4 assignments in each thread will iterate over a contiguous memory block of 128 vector elements, and ensure that elements `v[0]`..`v[127]` are all cleared properly.

But it's not a coalesced access order. The first assignment `v[id]` in these 32 threads will set `v[0]`, `v[4]`, `v[8]`, etc. This is not a coalesced pattern, since none of these addresses are adjacent.

The second assignment is also non-coalesced, which also reduces performance. Each of the threads will run it in lock-step, but the array assignment `v[id+1]` will set `v[1]`, `v[5]`, `v[9]`, etc. Similarly, the third and fourth assignments are not coalesced.

Strided Memory Accesses

The way to get coalesced memory accesses, if you can't have each thread accessing only one address, is to have each thread "stride" through the array. In this way, each thread can do multiple memory access operations, but each step of the sequence has all the 32 threads in a warp accessing adjacent memory addresses, so as to achieve coalescing.

How does a stride work? We want to have 32 threads, each doing 4 assignments, but in such a way that each of the 4 assignments in spreading a contiguous range over 32 vector elements.

Obviously, we want the first assignment `v[id]` to set `v[0]`...`v[31]` for the 32 threads in the warp. The second assignment should set `v[32]`...`v[63]`. The third should cover 64..95 and the fourth has 96..127. If we look at just the first thread in the warp, this should set:

- First assignment — `v[0]`
- Second assignment — `v[32]`
- Third assignment — `v[64]`
- Fourth assignment — `v[96]`

You can see the pattern: we want to do so every 32 vector elements. This value of 32 is called the *stride* of the array. Or, rather, the number of bytes is more properly called the stride on an array, but you get the idea.

Here's our first attempt at a strided kernel with coalesced memory access patterns:

```
__global__
void aussie_clear_vector_stride(float* v, int n)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < n) {
        int stride = 32;
        v[id] = 0.0;  // Clear 4 elements.. STRIDED!
        v[id + stride] = 0.0;  // BUG!!
        v[id + 2*stride] = 0.0;
        v[id + 3*stride] = 0.0;
    }
}
```

Well, this is indeed coalescing its memory accesses, but not always in the right place — it's just a little buggy. The test "`id<n`" does not prevent array bounds overflow for the subsequent assignments such as "`v[id+stride]`". Hence, this code craters for any values of `n` with extra threads. It really needs an "`if`" safety test before all the four assignments, rather than just the first.

The way to fix it is actually to use a "stride loop" idiom, where array bounds are checked each iteration. Note that this code isn't yet one of the fabled "grid-stride loops" that lives in the GPU unicorn forest.

This is just a basic stride loop, but not over the grid size.

```
__global__
void aussie_clear_vector_stride_loop(float* v, int n)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = 32;
    for (int i = 0; i < 4; i++, id += stride) {
        if (id < n) {
            v[id] = 0.0; // Clear 4 .. STRIDED!
        }
    }
}
```

The loop version is basically like the non-loop stride version, but re-rolled into a short loop. It does the same set of array accesses in the same order, but the safety occurs because "`id<n`" is performed each iteration.

This is a coalesced access order, as discussed above, because each assignment runs in a warp in lock-step. The contiguous memory addresses of `v` are processed like this by the 32 threads in the first warp: 0..31, 32..63, 64..95, and 96..127.

Contiguous Memory Blocks

CUDA C++ typically requires that matrices and tensors are stored in linearized contiguous memory, whether in CPU RAM or GPU VRAM. A critical part of optimizing backend engines is to manage the storage of data in a contiguous memory block, so that they have a sequential address space for high data locality and coalesced access patterns.

Processing chunks of data in parallel is the main optimization used in GPU acceleration. All of the vectors, matrices, and tensors need their underlying data in a block. For example, this applies to both pre-computed static weights and dynamic interim activation results in an AI engine's computations.

Processing data that is in adjacent addresses is much faster than jumping all over the place for both host and device code. Vectors should obviously be stored in a simple contiguous array of memory. Less obviously, similar comments apply to the memory storage of matrices and tensors.

The use of contiguous memory is an important optimization for both sequential and parallel algorithms.

The reasons that memory blocks are more efficient include:

- Data locality (cache hits)
- Data block GPU uploads (model weights from memory-to-cache)
- Coalesced access patterns (GPU data access)
- 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 RAM (VRAM) is done in blocks via `cudaMemcpy`. 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 memory 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 on a CPU. GPUs are different from CPUs, and striding through an array can be a perfect optimization for coalescing in GPU kernel threads.

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. Memory block operations in the standard C++ libraries are implemented using fast assembly language behind the scenes.

The main functions in the C++ library that operate on binary bytes in a memory block are:

- memset, cudaMemset — set bytes to a value, or clear bytes to zero.
- memcpy, cudaMemcpy — copy bytes in a block (non-overlapping).
- memmove — copy bytes, allowing overlapping blocks (no CUDA equivalent).
- memcmp — compare two blocks of bytes (no CUDA equivalent).
- memchr — search for a byte in a block (no CUDA equivalent).

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 fixed 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 / _memicmp — compare bytes ignoring letter case.
- bcopy — copy a block of bytes.
- bzero — clear a block of bytes to zero.
- bcmp — compare bytes in two blocks.
- _byteswap_uint64 — swap the bytes of an integer (Microsoft).
- __builtin_bswap16 — swap the bytes in an integer (GCC), with 32-bit and 64-bit versions.

Memory Initialization Optimizations

There are several schools of thought with regard to initialization costs:

- Don't initialize (speedy) but run compute-sanitizer often!
- Initialize everything to zero so it's safe for everyone (slower).
- Initialize to non-zero bytes to shake out more "use of uninitialized memory" type bugs.
- Initialize debug versions with non-zero bytes, but compile-out for production (faster for customers).
- Initialize to non-zero bytes for debug versions, but initialize to zero for production (safer for customers).

Making that choice is above my pay grade, but you're free to choose whatever you like. It's not even the full list, because the "shake out more bugs" idea has a few variants:

- Regularly use `compute-sanitizer` in the nightly builds.
- Use a debug library that quickly marks memory with magic values and detects bad accesses based on the magic numbers.
- Use a full debug library that tracks all allocated addresses in a hash table.

Initialize memory blocks with cudaMemset and memset

The `cudaMemset` function is for initializing device memory (or host memory in Unified Memory), such as a typical call:

```
    cudaMemset(dptr, 0, n * sizeof(float));   // 
Initialize
```

The `memset` function is a lower-level function in standard C++ that sets all of a memory block to a byte value. It can be used in both kernel and device code, and is widely used as a fast way to initialize a block of memory to all zeros. Typical usage is therefore:

```
    memset(&x, 0, sizeof(x));   // Initialize
```

Almost all usages of `cudaMemset` or `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 method for mutation testing to catch uses of uninitialized memory. Here's an example:

```
    cudaMemset(dptr, 0x55, sz);  // Magic value for debug
```

memset sizeof problem. Here's a common glitch in using the `memset` call inside functions with array parameter:

```
    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, which is an insidious bug.

Personally, I also recommend a `memset` debug wrapper function to catch this kind of problem at runtime. Alternatively, maybe use a tricky preprocessor macro can detect it at compile-time with a `static_assert` somehow. These techniques are discussed in my book *CUDA C++ Debugging*.

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 (regardless of endianness), floating-point zero (positive zero is all bits zero), and the null pointer.

Fast memory block copying with cudaMemcpy and memcpy

The fast way to copy an entire memory block is with `cudaMemcpy` or `memcpy`. The `cudaMemcpy` API is widely used in CUDA C++ programs to copy data between host and device.

The `memcpy` function tends to be used at a lower level for data manipulation. This applies to copying vectors and to matrices and tensors that are linearized into a contiguous array. 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 block in one statement:

```
  memcpy(destarr, srcarr, sizeof(srcarr)); // Copy bytes
```

Note that our use of `sizeof` operator here is risky, if the variable is an array parameter of a function, because this will return the smaller value representing the size of a pointer.

Note that `memcpy` is a bitwise copy of the array intended for simple data types. For example, it won't run C++ 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, but `memcpy` will copy everything, even null bytes and hidden padding bytes, and will always copy a fixed number of bytes.

memcpy overlapping blocks error: `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. Note that `cudaMemcpy` does not seem to have this problem (there's no `cudaMemmove` function), but overlapping parameter ranges might be a coding bug anyway. And don't worry, `cudaMemcpy` doesn't get left out, because it has enough of its own problems, like ensuring the pointers are the right types, and the copy mode is the correct direction.

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 this code will run correctly. The fix is simply to use memmove, which always works correctly for overlaps:

```
memmove(s, s+1, strlen(s+1)+1);  // Correct
```

The memmove function is a safer version of memcpy, which also works correctly when 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.

memcmp byte comparisons

The memcmp function does a byte-wise comparison of a memory block. There's no equivalent (i.e., no cudaMemcmp function), but an alternative is the vector equality test Thrust::equal.

The memcmp return value is like strcmp, returning 0 for equality, and a negative or positive value otherwise. However, note that memcmp is not like strcmp, and will not stop when it finds a zero byte.

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 bug. 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.

Unfortunately, there are several obscure pitfalls with this approach. There are multiple ways in C++ that the data in two memory blocks might be "identical" in a programming sense, but the bytes different:

- Padding bytes (alignment)
- Two types of floating-point zero
- Multiple types of floating-point `NaN`
- Bitfield data members (unset padding bits)

You can only use `memcmp` for memory block comparisons (e.g., tensor equality) if you're sure that these situations cannot occur, such as a contiguous array of primitive data types. Depending on context, you may or may not want floating-point zero and `NaN` values to map as equal. Take care with this!

10. Data Transfer Optimizations

Bottlenecks in Data Transfer

Transfer cost of data can be a bottleneck, which includes:

- Host-device data transfers (i.e., CPU-to-GPU or GPU-to-CPU).
- Multi-GPU data transfers.
- Server data transfers

General optimizations that reduce overall data requirements are effective as reducing overall transfer volume (e.g., smaller data types, algorithm-level changes). There are also several optimization techniques that are specific to data transfers, including:

- Pinned host memory that cannot be page-faulted by the CPU.
- Overlapping data transfers and CPU computations.
- Overlapping data transfers and GPU kernel computations.
- Splitting data transfers into chunks that can be overlapped.
- Prefetching to trigger a transfer sooner (e.g., `cudaMemPrefetchAsync`).

Host-Device Transfer Costs

If you are using a simple kernel, the typical sequence is:

- `malloc` — allocate CPU memory.
- Initialize CPU copy of input data.
- `cudaMalloc` — initialize device vector on GPU.
- `cudaMemcpy` — copy input data up from CPU-to-GPU.
- Kernel launch — snazzy grid-stride loops.
- Synchronize — e.g., `cudaDeviceSynchronize` to wait.
- `cudaMemcpy` — copy results back from GPU-to-CPU.
- `cudaFree` — cleanup kernel memory.
- `free` — cleanup CPU memory.

In this sequence, you can see where the data transfers are: `cudaMemcpy`. This does the data transfers in both directions.

When I did some timings on simple non-overlapped non-pinned data transfers and a very basic kernel (element-wise vector clear), I saw this pattern of results across multiple different versions of my kernel (basic, segmented, grid-stride loop):

- `cudaMalloc` remote allocation — 5%
- `cudaMemcpy` input data transfer up — 40%
- Kernel launch and synchronization — 10%
- `cudaMemcpy` results data transfer back down — 40%
- `cudaFree` remote de-allocation — 5%
- Everything else — negligible.

Oh, and one point: I excluded the CPU costs of initializing the vector with data, and running the self-tests on the CPU to unit test that it worked. I had to do this with the CPU costs of a couple of `for` loops came in at around 70% of execution time, because CPUs are slow! Hence, the above analysis is only data transfer costs and GPU computations.

Let's just stop and bask in that glory for a moment. We made the right choice in learning CUDA because it's so very fast.

On the other hand, it means you bought the wrong book, because you're optimizing only 30% of the problem, and you need a book on non-CUDA C++ optimization on the CPU.

Early Runtime Initialization. Also, I noticed that the very first call to `cudaMalloc` in the program execution was very expensive, circa 100ms, but the second and subsequent `cudaMalloc` calls were no longer this costly. Presumably, this is the setup time for Unified Memory addressing, with a handshake between the CPU and GPU happening across the PCIe bus behind-the-scenes.

If you don't want your first user query to have a 100ms extra delay then issue a dummy call to warm up the GPU and trigger the CUDA Runtime API initialization. Actually, I removed it from the main computation path simply by adding a call to `cudaDeviceSynchronize` earlier in my startup code.

Data Transfer Costs. Analyzing the above data without that extra initialization cost, we can surmise a few notable factors:

- Direction not important — the data transfers were approximately the same cost in either direction.
- Hidden CPU-GPU synchronization costs in Unified Memory address management (i.e., `cudaMalloc`, `cudaFree`).
- Data transfer costs were 80%!
- Allocation/cleanup of remote memory raised that to 90%!

Admittedly, this was a very simple kernel, but it nevertheless underscores the fact that host-device data transfer costs are significant. My test code is definitely not compute-bound. What can we do about that?

Pinned Host Memory

Memory in the CPU will normally be subject to paging, so primitives like `cudaMemcpy` have to run slower by taking a copy. We can avoid this expense by using "pinned" memory (also called "page-locked"), where operating system paging is blocked, and the memory transfer can run faster. The calls to manage pinned allocated memory are:

- `cudaMallocHost` or `cudaHostAlloc`
- `cudaFreeHost`

Note that this is *host* memory, so you make these changes to the handling of CPU data:

- `malloc` — replace with `cudaMallocHost` or `cudaHostAlloc`.
- `free` — replace with `cudaFreeHost`.

You don't change `cudaMalloc` or `cudaFree`! In fact, you leave the both in there, because that's the other step of setting up the allocated memory on the device side. Pinned memory is on the host side.

My basic test code looked like this:

```
    if (pinned) {
        // Pinned host array
        CUDACHK(cudaMallocHost((void**)&v, n * sizeof(float)));
    }
    else {
        // Non-pinned host memory
        v = (float*)malloc(n * sizeof(float));
    }
```

And the matching de-allocation code at the end:

```
    if (pinned) {
        CUDACHK(cudaFreeHost(v));  // Free pinned host vector
    }
    else {  // not pinned...
        free(v);   // Free non-pinned host vector
    }
```

When I made these changes and re-profiled the CPU time for this version, it was *worse*! The overall time cost on the CPU went up. The general pattern of the time cost changed to:

- `cudaMallocHost` pinned allocation — 40-50% (very high!)
- `cudaMalloc` remote allocation — 3% (lower)
- `cudaMemcpy` input data transfer up — 10-15% (down from 40%)
- Kernel launch and synchronization — 10%
- `cudaMemcpy` data transfer back— 10-15% (down from 40%)
- `cudaFree` remote de-allocation — 3% (lower)
- `cudaFreeHost` pinned de-allocation — 15-25% (very high!)

Whereas the calls to `malloc` and `free` were "negligible" in the non-pinned version, the calls to `cudaMallocHost` and `cudaFreeHost` were very expensive. On the other hand, the amount of CPU time used by the `cudaMemcpy` calls, both up and down, was reduced to about half its original cost.

Is this a bad result?

What this shows is that initialization of pinned host memory is expensive, but that the data transfer time can be greatly reduced to about half the time. Hence, there's not an efficiency gain, and in fact a loss, if you have to allocate and de-allocate the pinned host memory for every user query. However, if you can pre-initialize pinned memory blocks on the host and then re-use them over many user queries, the latency for users will be reduced, because the data transfer costs are down by 50%.

Overlapping Data Transfers and CPU Computation

When the host code launches a kernel, it does so asynchronously, and the CPU can go do other work while the GPU processes the kernel, thereby overlapping or parallelizing CPU and GPU computation. A similar idea can be applied to overlapping data transfers and CPU computation.

To do this, we need an asynchronous way for the host to start data transfers. The default behavior of `cudaMemcpy` is synchronous on the host, and will block waiting for the data transfer to complete. Hence, you need to change to `cudaMemcpyAsync`, which is the non-blocking version. Since the host no longer waits around for the data to be copied up to or down from the GPU, the host code can do other computations in parallel.

Usually, the host will need to know when the GPU work has finished on the data that it uploaded. Or it may need to know when results data from the GPU has been downloaded. Thus, the host code needs a way to determine when:

(a) an asynchronous data transfer has completed, and/or

(b) the kernel processing on that data has occurred.

This type of synchronization is usually done via CUDA streams.

Streams

What are CUDA Streams? The way that streams work is that you can queue up a series of work jobs on a stream. All jobs on a stream have to be run sequentially, but the work can be parallelized across different streams. In this way, CUDA C++ offers an easy way for the application code to specify which work can be parallelized and what must remain serialized.

CUDA Runtime calls that don't specify a stream are on the "default stream." This is a simple but inflexible way of running some work, but does not offer as fine-grained parallelization as with the use of streams.

CUDA handles most of the work for streams, and the programmer just has to load up the work. For example, the CUDA Runtime will handle the parallel scheduling and sequencing requirements of these jobs.

The basic idea for efficient parallelization using streams from the host code viewpoint is:

1. Load up the data transfers, kernels, and other work jobs onto a stream.

2. Do other stuff in parallel.

3. Synchronize when the stream has finished its work.

For example, if the host code needs to do a vector dot product computation:

1. Queue up two asynchronous host-to-device data transfers for the two input vectors onto the stream (i.e., `cudaMemcpyAsync`).

2. Queue up the vector dot product GPU kernel launch (asynchronously on the same stream).

3. Queue up the device-to-host data transfer of the results vector (same stream).

4. Do other stuff in parallel on the CPU.

5. Synchronize with the stream when the results data is available.

This is highly parallel for the CPU, but less so for the GPU. Although the GPU can do work from other kernels in parallel, most of this sequence is sequential. The GPU has to execute the data transfers, kernel launch and return data transfer in serial order, as they are queued on the same stream.

Asynchronous CUDA Operations

Efficient use of streams requires asynchronous actions. Although kernel launches are asynchronous, most of the simplest CUDA runtime functions are synchronous by default, such as `cudaMalloc`, `cudaMemset`, and `cudaMemcpy`. The code will block waiting for them to finish, even if the work is being done on the other hardware (e.g., the CPU host code blocks waiting for the GPU to do a memory set or memory allocation).

To do parallelization and overlapping optimizations with streams, you need to use the asynchronous CUDA methods:

- Kernel launch `mykernel<<<...>>>` syntax (asynchronous by default)
- `cudaMemcpyAsync` (and also `cudaMemcpy2DAsync` and `cudaMemcpy3DAsync`)
- `cudaMemsetAsync`
- `cudaMemPrefetchAsync`
- `cudaMallocAsync` (CUDA 11.2)
- `cudaFreeAsync` (CUDA 11.2)

The most basic GPU operation is already asynchronous: launching kernels. You have to do the work to synchronize the host with a kernel launch, such as by calling `cudaDeviceSynchronize`, which is simple but inefficient. For the other work, you need to do two changes:

1. Add a stream argument to the kernel launch syntax, and

2. Call the asynchronous versions of the CUDA APIs, and provide a stream to track them.

Overlapping Kernels and Data Transfers

It also possible to overlap GPU kernel execution with asynchronous data transfers. Note that the GPU is capable of doing data transfers without launching any threads, and the threads can be doing other things (i.e., other workloads can run). These host-device data transfers are processed in a different part of the GPU hardware, which runs in parallel with the execution of kernel threads. This is true for both uploads (CPU-to-GPU) and downloads (GPU-to-CPU), and has been supported in GPUs since approximately Compute Capability 1.1 (i.e., for many years).

However, a naive attempt to overlap an upload of input data (i.e., CPU-to-GPU data transfer) and the kernel that processes that data, is not going to work. The kernel has to wait! It cannot start processing until it has the data.

This model only works if there are other kernels doing work on the GPU. The other kernels that don't require the input data can run in parallel with the data transfer. However, our kernel is blocked waiting for the data upload to complete.

Overlapping Partial Transfers and Computations. The idea of overlapping data transfers and computations doesn't need to occur within different kernels to get a parallelization gain. At first thought, you can't overlap the data transfer for a kernel that requires the data, because the kernel has to wait for the full data transfer to complete. Only when it has the data, can it can begin computation. Hence, it seems that this kernel can't do any work in parallel with the data transfer.

This is annoying, because surely the kernel could start adding two vectors, even if it didn't have the whole vectors. Indeed, that's the idea, but CUDA cannot do it automatically for you. If the kernel is an algorithm that can be parallelized into chunks (e.g., any element-wise vector or matrix operation, just for starters), the transfer and processing of the separate chunks can be overlapped by splitting them onto multiple streams.

Data transfers and kernel processing of two different chunks can be overlapped. Obviously, you have to serialize the processing of each chunk, so that the upload of one chunk has to finish before the kernel operates on that same chunk. However, the kernel can start processing on the first chunk (after it finishes uploading) while the second chunk is uploading in parallel. This overlaps the first kernel computation with the second data transfer, and then the second is processed while the third is uploaded, and this can continue over many pairs of different chunks. The only parts that are not overlapped are the first and last chunks.

The implementation of this idea requires multiple streams initialized, one for each chunk. And each of these streams needs to have work queued:

1. Transfer the chunk (e.g., vector segment) to the GPU.

2. Kernel launch to work on that chunk.

3. Transfer the resulting chunk back to the CPU.

Note that you can't set up a single thread to handle each chunk (why would you want to?). You have to queue up a full kernel launch, with one or more blocks, and warps of threads, on the right stream for that chunk.

Another even more advanced way to parallelize work on chunks of data would be to use a "persistent kernel" that handles each chunk. A persistent kernel never exits, but has its threads endlessly running, tirelessly waiting for work, and then receiving their work jobs on a scheduler queue. It's quite an involved architecture, and is discussed in a later chapter.

Additional Host-Device Optimizations

Tuning the interaction between the host and device code is a never-ending programming task. Here are some additional techniques:

1. Managed Memory. The CUDA APIs for “managed memory” are convenient, but not necessarily more efficient. On the other hand, you might use a poor algorithm, whereas the CUDA Runtime will manage the transfer of data behind-the-scenes.

2. Reduce Data Transfer Volume. The cost of data transfer is often a significant cost, whether it is host-device transfers, or between GPUs or servers. Algorithmic changes can sometimes be used to reduce the size of these transfers, or you can use smaller data types (i.e., quantization).

3. Batch Multiple Small Transfers. If you have a lot of small data transfers, the overhead can be significant. Batching them together into a single, larger transfer won’t reduce volume, but it cuts down on the overhead.

4. Don’t Transfer Interim Results. The last thing you want is data going back-and-forth between CPU and GPU. If your CPU application has multiple steps in the overall top-level algorithm, don’t interleave CPU and GPU computations. Instead, you need to migrate all of the processing to the GPU, rather than send back partial results that are re-processed by the CPU. If you’re porting legacy CPU code to the GPU, don’t stop halfway!

5. Larger Kernel Launch Parameters. CUDA C++ allows passing data from host-to-device as kernel launch parameters, which are stored in constant memory. Although these are more commonly used to pass single values, such as pointers, sizes and dimensions, this is also a fast way to pass large data chunks to the GPU efficiently. These parameters were previously limited to 4K in total size, but this limit is now 32K per launch for the latest GPUs.

6. Direct Memory Access (DMA) and GPUDirect. This optimizes GPU to storage accesses locally, such as accessing an NVMe or flash memory device. This bypasses the CPU by allowing direct memory transfers from local storage to the GPU. It works by copying memory blocks asynchronously over the PCIe bus.

7. PCIe Bus Bandwidth. This is the bus for data transfer between CPU and GPU. Its speed is an important consideration for host-device data transfers (e.g., via `cudaMemcpy`), which are any important part of CUDA kernel performance.

8. Zero-Copy Memory. This is pinned memory on the host, as used for asynchronous memory copying, but this method does not require copying (i.e., no call to `cudaMemcpy`). However, the data on the host is accessed by the GPU over the PCIe bus, which is not especially efficient and can have significant latency.

Networking Optimizations

Larger GPU applications may need to transfer data between multiple GPUs or across the network to other servers. A large data center running many CUDA backends will also have a lot of work to do in terms of the various networking protocols and software stacks. There are various optimizations in these cases, and here's a summary of some of them.

A data center running H100 GPUs will have different types of networking:

1. Front-end networking — Ethernet from external accesses.
2. Back-end networking — optimizing inter-GPU transfers.
3. Out-of-band networking — for internal monitoring and management.

The front-end networking is typically an Ethernet connection from the internet into the hosts. This is how customers and external users connect into the data center for reaching servers and for data storage needs.

The back-end technologies are much more intense and high-bandwidth, because they manage bursts of inter-GPU communications for reductions and gather operations. AI training applications have a particularly bursty pattern of concurrent data sending at high volume when updating the parameters. Technologies to use include InfiniBand, Spectrum-X, or RoCEv2 Ethernet. This may require optimizations to the NVIDIA Collective Communications Library (NCCL), such as to make Ethernet run fast enough. Different connectivity topologies may be considered viz network switches and the GPU servers. Connectivity hardware options include various network switches and the choice between optical or electrical cabling.

Monitoring and management of both software and hardware devices is important as failures and errors are common in hot GPUs and in other network devices and servers. Monitoring tools for data centers include Grafana and Prometheus. Insidious failures in GPUs from overheating that cause incorrect results in computations can be diagnosed by running self-diagnostics, such as NVIDIA's "`dcgmi`" diagnostics (at level 4).

As a CUDA C++ programmer you are largely abstracted away from all this frantic nonsense in the networking layers of a high-end datacenter, but it's a very specialist skill in high demand at the moment. Some additional information on particular networking technologies is below, and more details are also available in the references.

1. Remote Direct Memory Access (RDMA). This is a network protocol whereby servers can access the memory in other servers, without having to interrupt the remote CPU (or GPU). Using RDMA can allow fast network data transfers between servers without slowing down their computations.

2. Lazy Connection Establishment in NCCL. This is an optimization to the NVIDIA Collective Communications Library (NCCL) protocol, often pronounced as "nickel," for inter-GPU communication. Lazy connection establishment delays the establishment of connections by the GPU until they are required, thereby reducing the initialization time for NCCL.

The feature is controlled by the `NCCL_RUNTIME_CONNECT` environment variable, and can be disabled by setting this to zero. Note that this is not the same optimization as "lazy loading," which refers to GPU loading of machine code instructions.

3. Multi-GPU Peer-to-Peer Memory Access. This is sometimes called "P2P" in CUDA and is relevant to motherboards with multiple GPUs running on them. It is an optimization method that involves one GPU accessing the memory of another GPU directly, without any involvement of the CPU.

4. nvlink Data Transfers. This method is for multi-GPU communication within a server. It offers a faster communication protocol that bypasses the PCIe bus for data transfer, so as to allow GPUs to communicate more efficiently with each other.

5. Memory-Mapped I/O. This is an optimization where I/O peripherals are directly connected to memory, rather than needing the CPU's involvement to control data transfers. There are a variety of peripherals that could be attached to your CUDA algorithm, starting with a Tardis or a Holodeck.

References

1. Mark Harris, Dec 04, 2012, *How to Optimize Data Transfers in CUDA C/C++*, NVIDIA Technical Blog, https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/
2. Mark Harris, Dec 13, 2012, *How to Overlap Data Transfers in CUDA C/C++*, NVIDIA Technical Blog, https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/
3. Mark Harris, Jan 22, 2015, *GPU Pro Tip: CUDA 7 Streams Simplify Concurrency*, NVIDIA Technical Blog, https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/
4. Vivek Kini and Jake Hemstad, Jul 27, 2021, *Using the NVIDIA CUDA Stream-Ordered Memory Allocator, Part 1*, NVIDIA Technical Blog, https://developer.nvidia.com/blog/using-cuda-stream-ordered-memory-allocator-part-1/
5. Ram Cherukuri, Dec 16, 2020, *Enhancing Memory Allocation with New NVIDIA CUDA 11.2 Features*, NVIDIA Technical Blog, https://developer.nvidia.com/blog/enhancing-memory-allocation-with-new-cuda-11-2-features/
6. Dylan Patel and Daniel Nishball, Oct 03, 2024, *AI Neocloud Playbook and Anatomy*, https://www.semianalysis.com/p/ai-neocloud-playbook-and-anatomy
7. Together AI, Nov 13, 2023, *Announcing Together Inference Engine – the fastest inference available*, https://www.together.ai/blog/together-inference-engine-v1

11. Heap Memory Allocation

Heap Memory Optimizations

In an ideal world, we wouldn't use memory allocation inside GPU kernels. In the real world, there are optimization techniques for that. These techniques can also be used for host code, because the issues with dynamic memory allocation are similar for CPU and GPU code.

Larger Device Heap Size. You can increase the heap memory size for GPU kernels via the `cudaDeviceSetLimit` function with the property symbolic name `cudaLimitMallocHeapSize`. But maybe sit on your hands before you do this, because this is a reminder that your kernel should use less heap memory.

You can see the heap size of your GPU by getting and setting the "limit" property for the heap. Here's how to print it:

```
// Get heap size
CUDACHK( cudaDeviceGetLimit(&val, cudaLimitMallocHeapSize));
double meg = val / ((double)1024*1024);
printf("Device: heap size = %d bytes (%3.2f MB)\n",
       (int)val, meg);
```

Kernel Memory Allocation Optimizations. Lots of `malloc` and `free` calls in a kernel can really get it down. This is true for all C++ code, but using memory allocation on CUDA kernels is also a sluggish practice, and there's only a limited allocated memory heap size for all of the threads on the GPU, which you can query via API `cudaDeviceGetLimit` and property `cudaLimitMallocHeapSize`. The general types of optimizations include:

- Change to non-heap memory (various ways).
- Fewer allocation calls in general.
- Less data overall with smaller data types.
- Reduce fragmentation with consistent sizing.
- Allocate later and free earlier.

Instead of allocating memory, try to use iterations, or store the data in other memory.

For small amounts of data, the performance of the `alloca` function, which allocates stack memory, should be compared with the normal heap allocations with `malloc` or `new`.

If it's slow, just do it less! Don't needlessly allocate heap memory for convenience. For example, instead of an allocated buffer, have a dynamic buffer class which includes a fixed size of data, and only allocates heap memory if the total data exceeds this default size. Rather than catering to the general case with allocated memory, specialize your algorithm for smaller sizes to avoid needing memory allocations.

Fragmentation of the heap is another issue. Lots of allocations of small blocks, or interleaving allocations of differently sized blocks, can cause "fragmentation" of your memory heap. The allocator does its best, but sometimes it needs your help. Find ways to not only allocate fewer blocks overall, but keep the different sizes to a minimum.

An advanced way to avoid allocations and deallocations on the device is to use a "persistent kernel" architecture. This is a never-exiting thread, in which case you can pre-allocated the memory blocks you need, rather than repeated allocations and de-allocations.

Late Allocation and Early Free

A typical simple CUDA C++ kernel, such as a test version of vector addition, has this sequence:

1. Allocate three local host vectors (`malloc`).
2. Allocate three device vectors (`cudaMalloc`).
3. Store the input data into the two host vectors (host code).
4. Copy the two input vectors (`cudaMemcpy` host-to-device).
5. Launch the kernel for vector addition (<<<...>>> syntax).
6. Synchronize host-device (e.g., `cudaDeviceSynchronize`).
7. Download the results vector (`cudaMemcpy` device-to-host).
8. Process the local results vector (host code).
9. De-allocate three device vectors (`cudaFree`).
10. De-allocate three local vectors (`free`).

But that's not actually optimal and needlessly retains heap memory on both host and device over time periods where it doesn't need that memory. The results vector should be allocated later, and the two input vectors should be freed earlier.

Here's a much better sequence:

1. Allocate two local host input vectors (`malloc`).
2. Store the input data into the two host vectors (host code).
3. Allocate two (not three) device vectors (`cudaMalloc`).
4. Copy the two input vectors (`cudaMemcpy` host-to-device).
5. De-allocate two local input vectors (`free`) — earlier!
6. Allocate the third results device vector (`cudaMalloc`) — later!
7. Launch the kernel for vector addition.
8. Synchronize host-device.
9. De-allocate the two input device vectors (`cudaFree`) — earlier!
10. Allocate the third results local vector (`malloc`) — later!
11. Download the results vector (`cudaMemcpy` device-to-host).
12. De-allocate the third results device vector (`cudaFree`) — earlier!
13. Process the local results vector (host code).
14. De-allocate third local results vector (`free`).

It's more steps, but it's more efficient in its use of dynamic memory on both the host and the device. There are variations on this and my suggestion is possibly not the best overall sequence. Arguably, you should prioritize GPU time over host time, so maybe you shouldn't spend the time to de-allocate the two vectors with `free` on the host until later. So, our policy to maximize the critical section where the GPU is working should perhaps be to de-optimize the host memory allocation:

- Call host `malloc` before any GPU interactions, and
- Delay host `free` calls until all GPU work is finished.

Furthermore, this is a highly sequential algorithm that needs much more major changes to be fast on the GPU. Instead of only using the default CUDA stream, the faster method is breaking up the vectors into smaller segments across multiple streams to allow much greater parallelization.

As another idea, you might consider using the asynchronous allocation primitives such as `cudaMallocAsync` to overlap the device memory allocation and the host code that loads the input data locally.

You might also think that you can do the free asynchronously, with a further speedup, but the behavior of `cudaFreeAsync` is a little disappointing.

Asynchronous Memory Allocation

CUDA C++ has two asynchronous functions for memory allocation since CUDA 11.2: `cudaMallocAsync` and `cudaFreeAsync`. There are a few opportunities for optimization with these primitives, but there are also various pitfalls in the change from synchronous (blocking) calls to `cudaMalloc` and `cudaFree`, which have implicit host-device synchronization, to launching their asynchronous equivalents on a stream.

Another major risk of the unsynchronized versions is using the address from `cudaMallocAsync` before it is initialized. Changing to `cudaFreeAsync` is unlikely to trigger a new "use-after-free" or "double-free" fault, because that bug would have already occurred in the `cudaFree` blocking version.

Using asynchronous allocation with `cudaMallocAsync` can be part of overlapping memory management and kernel execution. This is discussed for data transfer optimizations in Chapter 10.

The use of `cudaFreeAsync` is not as useful as you might hope. For starters, you have to set up a stream, including creating it, and then destroying it later (so you don't have a "stream leak").

But it's also not that efficient. Arguably, you should be able to just throw out an asynchronous de-allocation request, and then forget about it completely, since nobody's ever going to be waiting for that memory.

However, `cudaFreeAsync` does not actually free the memory until a synchronization point on the stream, so you need to incur this stream synchronization delay at some point, and the heap memory is still used up until you do.

Heterogeneous Memory Management

HMM is a CUDA C++ feature announced in 2023 that simplifies host and kernel memory allocation. This is a software-based layer that merges the two main sources of allocated memory:

- System memory allocation (e.g., `malloc`)
- CUDA-managed memory allocation (e.g., `cudaMalloc`)

Effectively a generalization of Unified Memory, HMM allows memory allocated on either the host or GPU to be allocated by the other party. This works for memory allocated by any primitive, including standard C++ memory allocation (i.e., `malloc`, `calloc`, `new`), and CUDA memory allocation (e.g., `cudaMalloc` or `cudaMallocManaged`). It also works for memory-mapped blocks via the `mmap` system call. Thus, the programmer is relieved of the burden of needing to explicitly manage the various sources of memory allocation.

This new capability simplifies various coordination between the CPU and GPU. Some examples of areas that where direct access can be beneficial:

- Device access to large CPU memory blocks on (without transferring them!)
- Configuration flags and message-passing between host and device.
- Synchronization between host and device (e.g., atomics).
- Memory-mapped I/O (directly accessible from the GPU).

Custom Memory Allocators

If you don't like how much memory overhead there is from allocated memory, you can define your own. But here's a warning: These methods are not for the feeble or uncommitted, and can get quite involved. What's more, they can even be a de-optimization, because it's hard to beat operating system capabilities that have had many years of optimization work.

Memory Pooling. This is a memory management technique for allocated memory on the heap (e.g., `malloc`) that aims to reduce the costs from memory fragmentation. You can take control of the memory allocation algorithm by using your own "pools" of allocated memory blocks. This works particularly well if you have a large number of allocated memory blocks that are exactly the same size. A common example is where a particular C++ `class` object is allocated a lot, in which case you can override its memory allocation by defining your own `class`-specific `new` and `delete` operators.

Dynamic Memory Allocators. You can also try to define your own generic memory allocator routines to replace the default versions provided by CUDA. But honestly, you'd have to get up very early in the morning to do better than what's already been done. But if you must do it, this is one way:

- Macro interception of `malloc`, `calloc` and `free`.
- Link-time interception of the global `new` and `delete` operators.

If you think you're up for it, feel free to take on this challenge!

Allocating Stack Memory: alloca

The alloca function can be used to allocated stack memory rather than heap memory. It works in both host and device code, but this optimization may be less applicable to host code, simply because it has a huge heap space.

This main reason to try stack allocation is that the stack is faster memory than shared memory or global memory. Although the alloca function can be used in device code to dynamically allocate memory blocks on the thread's stack, there are various advantages and disadvantages that should be considered carefully.

The main advantages of alloca include:

- Stack memory can be faster to access.
- alloca function is itself fast.
- Automatic de-allocation on function exit when the stack unwinds.
- Helps avoid heap memory fragmentation.

The pitfalls include:

- Alignment problems (there's also aligned_alloc).
- Limited stack size (i.e., alloca may fail in device code).
- Addresses are invalid after function exit (i.e., the scope is reduced and lifetime is shorter than with heap allocations).
- Uninitialized memory by default (but nor do malloc or cudaMalloc).
- No way to de-allocated it programmatically, before the function exits.

There is a limited amount of local memory, and you can view or increase this using the cudaDeviceGetLimit and cudaDeviceSetLimit API functions with the named property cudaLimitStackSize property.

Here's how to print the value:

```
// Get stack size
CUDACHK( cudaDeviceGetLimit(&val, cudaLimitStackSize));
double kb = val / ((double)1024);
printf("Device: stack size = %d bytes (%3.2f KB)\n",
       (int)val, kb);
```

Note that it's much more important to test for memory allocation failure when using `alloca`, because the GPU stack is so small. But, of course, you're always using good programming style and checking the return values of every CUDA primitive, right?

Memory Leaks

A common problem in managing allocated memory is leaking memory, from blocks that are not de-allocated by `free` or `cudaFree`. Finding memory leaks is often considered a debugging task, but it also helps with performance, so it's both "debugging" and "deslugging."

Nevertheless, many of the debugging tools are helpful in detecting leaked blocks:

- `compute-sanitizer`
- `valgrind` (limited)

Compute Sanitizer is more focused on memory errors than leaks with its default settings. The full memory leak detection available in `compute-sanitizer` can be enabled via the "leak checking" command-line argument:

```
compute-sanitizer --leak-check=full myexecutable
```

You can even use the Linux `valgrind` tool to chase down memory leaks, if you prefer. I'm not sure how fully it works for device code allocations, but it certainly works on host code.

References

1. Vivek Kini and Jake Hemstad, Jul 27, 2021, *Using the NVIDIA CUDA Stream-Ordered Memory Allocator, Part 1*, NVIDIA Technical Blog, https://developer.nvidia.com/blog/using-cuda-stream-ordered-memory-allocator-part-1/
2. Ram Cherukuri, Dec 16, 2020, *Enhancing Memory Allocation with New NVIDIA CUDA 11.2 Features*, NVIDIA Technical Blog, https://developer.nvidia.com/blog/enhancing-memory-allocation-with-new-cuda-11-2-features/

3. Vivek Kini and Jake Hemstad, Jul 27, 2021, *Using the NVIDIA CUDA Stream-Ordered Memory Allocator, Part 2*, NVIDIA Technical Blog, https://developer.nvidia.com/blog/using-cuda-stream-ordered-memory-allocator-part-2/
4. John Hubbard, Gonzalo Brito, Chirayu Garg, Nikolay Sakharnykh and Fred Oh, Aug 22, 2023, *Simplifying GPU Application Development with Heterogeneous Memory Management*, NVIDIA Technical Blog, https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/
5. Mark Harris, Jun 19, 2017, *Unified Memory for CUDA Beginners*, https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

12. Compute Optimizations

Parallelization

Really? CUDA can do parallelization on a GPU? Who would have thunk it.

Obviously, the whole point of a GPU is to do parallelization in its kernel threads. The main ways to do parallelization on a GPU include:

- Run kernel threads in parallel (i.e., vectorization or "data parallelism").
- Run multiple kernels in parallel (i.e., higher-level "task parallelism" in the algorithm).
- Multi-GPU architectures if you're flush.

The general idea of taking an array or vector of data and operating in parallel on each of its elements is called "vectorization." This is the bread-and-butter of CUDA kernels! The compiler can also auto-optimize in some cases to do additional lower-level vectorization by unrolling loops within kernels.

Task parallelism means looking for high-level "tasks" within your application that can be parallelized (i.e., don't depend on each other's results) This can be managed in CUDA applications using "streams" in many of the CUDA Runtime primitives. By defining different activities in different streams, CUDA has the information to know what tasks depend on each other, and can automatically schedule for the best possible parallelism.

And we can stop there, because the rest of the improvements seem less important. It's quite plausible to have the CPU running as the overall controller of the GPU kernels, but using sequential logic itself. Time to go and get a cup of coffee.

If you're still reading, there are some other ways to squeeze out a bit more juice.

Some of the additional ways to add parallelism to your CUDA applications include:

- CPU and GPU workload parallelism.
- Overlapping data transfers and GPU computation.
- Overlapped data transfers for the CPU, too.

CPU and GPU Workload Parallelization. In any CUDA architecture, you have both a CPU and a GPU, and you can put them both to work in parallel. I mean, the GPU will probably do 99% of the grunt work, but let's at least keep the CPU happy with some token jobs.

Kernel launches are asynchronous on the host, so the CPU can keep going with its code while the GPU starts cranking away on those threads. To achieve this type of simple CPU-GPU compute parallelism, you need to do this:

1. Launch the GPU kernel in a stream.

2. Don't synchronize!

3. Do other work on the CPU.

4. Synchronize with the stream when it's ready.

The host code needs to avoid synchronization, because that would cause the CPU to block. Obviously, that means to avoid calling `cudaDeviceSynchronize`, which is the worst because it blocks the CPU to wait for *everything*. However, there are also many CUDA functions that have an "implicit synchronization" with the GPU, such as:

- `cudaGetLastError`
- `cudaMemcpy`
- `cudaMemset`
- `cudaMalloc`
- Various others.

Instead of synchronizing either explicitly or implicitly, the CPU needs to do its other work, while watching the stream *asynchronously* for completion of the GPU kernel.

Synchronization Slugs

The other side of the coin for parallelization is synchronization. You can't do too little synchronization, because that's a bug in your parallel algorithm. But if you do too much synchronization, then it's a slug.

If the CPU or GPU is sitting around waiting for somebody else to do something, that's not fast. Over-synchronization can represent a bottleneck to many algorithms. High-level inter-dependence should be reviewed to see if any algorithm-level changes can be made. Low-level used of synchronization primitives should be reviewed to ensure that none are redundant.

Some of the beginner mistakes in synchronization include:

- Calling `cudaDeviceSynchronize` at every second statement.
- Hidden implicit synchronization in `cudaGetLastError` and others.
- Blocking nature and "implicit synchronization" of simple CUDA runtime functions (e.g., `cudaMemcpy`, `cudaMalloc`).
- Too many calls to `__syncthreads()` and `__syncwarp()`.
- Accidentally leaving "serialized kernel launch" settings enabled (e.g., `CUDA_LAUNCH_BLOCKING`).
- Overuse of atomics (like in *Dune*) can cause delays due to synchronization.
- Redundant barriers are unnecessary synchronization, whereby no race condition would arise without the barrier.

The host should avoid synchronization with device kernels as much as possible, so as to allow the host to launch multiple kernels in parallel, or to perform other computations on the CPU while the GPU is also processing. Obviously, the ability to do this depends on the overall algorithm. Explicit synchronization occurs with CUDA Runtime calls such as `cudaDeviceSynchronize`, but there is also "implicit synchronization" in functions, such as `cudaMemcpy` and `cudaMemset`.

More effective types of synchronization include:

- Use `cudaPeekAtLastError` to avoid implicit synchronization.
- Use `cudaStreamSynchronize` with streams on the host, along with asynchronous primitives, for more granular control of synchronization.
- Synchronize blocks and warps when you need to, but not too often, and hopefully just right.
- Avoid redundant atomics, but use them when you need to.

It's hard to get the balance right, but at least you can watch Netflix while you try.

Thread Bottlenecks

Imagine if you went to all the trouble of analyzing your application to find task parallelism, and then implemented kernels to launch threads to vectorize all of that, and then it still ran slow?

How awkward!

Don't let your threads get bogged down for really simple reasons. Various things, both internal and external, can prevent your kernel threads from achieving greatness. Always beware the various bottlenecks that may arise in thread execution, such as:

1. Excessively Large Threads. The extreme of "thread coarsening" is large kernels running bloated threads, with a large body of code, or many sub-functions being called, which results in very low thread parallelism. Such code might need some refactoring, or even an algorithm re-design.

2. Function Call Overhead versus Inlining Functions. When the GPU kernel calls another function, this creates function call overhead and uses its stack memory. Using an `inline` function is allowed for device code, and benefits from compile-time optimizations and reduce stack memory usage. You can also consider whether to use preprocessor macros versus `inline` functions.

3. Recursion. Well, if you're using recursion in production code, especially in a kernel, you deserve to go slow. I have no sympathy for you! The only valid place for recursion is in a University assignment about binary tree traversals. Recursion and GPUs don't mix.

4. CUDA Runtime Errors. An aspect of both performance and debugging is to check every CUDA Runtime API function for an error return. A failing kernel is not just a bug, but also a slow kernel. Most CUDA programmers use macros to wrap every call to these CUDA runtime C++ functions. You can also regularly use `cudaGetLastError` or `cudaPeekAtLastError`, but be aware of implicit synchronization.

5. Assertions Failing. Assertion failures from the `assert` function on devices are a special case of CUDA Runtime errors. These will allow the working threads to keep going, but have some threads fail, which is a slow-down. These failures also trigger `cudaErrorAssert` errors. There's a great book called *CUDA C++ Debugging* with a whole chapter written by me on assertions.

6. Debug Trace Statements. The kernel has the builtin `printf` function, but it slows down performance, especially for large amounts of output, so only use it when debugging. Come on, you don't really need all that trace output! We have debuggers for that. Also, another chapter in the *CUDA C++ Debugging* book.

7. Busy Wait Functions or Timers. In the category of "you did this to yourself": don't implement timers or "sleep" functions on the GPU using a busy wait. It's better to run timers on the host, and the kernel should not be marking time by spinning in a loop.

8. Insidious CUDA Runtime Slugs. The CUDA runtime checks a lot of its pointer arguments, but there are a few cases where sending the wrong address pointer with Unified Addressing causes a slug, rather than a CUDA error code. The whole idea is to relieve you of the burden of managing pointers, but sometimes if you send a host pointer instead of a device pointer, that means it'll be transferring data without you realizing. This may occur on host code or device code. For example, try sending a host pointer from `cudaMallocHost` as an argument where `cudaMemcpy` expects a device pointer (in a few cases only).

Additional Thread-Level Optimizations

1. Thread Coarsening. This optimization involves having "coarse" threads that perform more work. A classic example is unrolling loops inside a kernel thread. The advantage of having coarser threads is reduced kernel launch overhead for a set amount of computation. However, they can also worsen performance by reducing thread parallelism, so there is always a balance, and benchmarking is desirable to optimize.

2. Loop Unrolling. The idea with loop unrolling is to perform multiple computations without testing the loop condition. Fixed-size loops with the size known at compile-time can even be fully unrolled. Note that although loop unrolling is often very powerful in sequential programming, it can actually reduce thread parallelism and introduce non-coalesced memory access patterns when used in CUDA kernels.

The cost of a simple index variable comparison (e.g., an "`i < n`" test), where both `i` and `n` will be in a register, is comparatively low against global memory access costs (often over 100 in order of magnitude). Hence, the benefit of loop unrolling in CUDA is moreso the avoidance of kernel launch overhead, rather than loop comparison test overhead.

3. Instruction Cache Optimizations (Instruction Locality). Every CPU maintains its own instruction pointer and has an instruction cache that aims to speed up loading of instructions. Using instructions that are "close" to each other can be significantly faster, whereas branching widely over a large set of instructions destroys instruction locality.

4. Avoid Redundant Extra Threads. Threads are allocated in groups of 32, and cannot be launched in a smaller number. Hence, if the algorithm needs some "extra" computations, there can be threads running or inactive, which wastes resources. Entire warps of threads can also run needlessly in some poorly designed algorithms. Take care to examine your "safety" conditions in threads, in case you accidentally have many threads doing nothing.

5. Detect Redundant Thread Computations. This is where multiple threads are doing the same computations on the same data. This is more likely to be in a novice CUDA kernel, but experts occasionally make errors that can cause additional threads to perform the same work. This error can be hard to detect, because it is not failing, but multiple threads are doing the correct calculations. Sometimes, it can be whole redundant warps!

GPU Configurations

Some additional ways to maximize your use of a GPU are listed below:

1. GPU-Specific Optimizations. More advanced GPUs allow additional types of optimizations. Make sure you take advantage of the full hardware capabilities. These can be examined programmatically in the CUDA C++ code with API calls such as `cudaGetDeviceProperties`. Useful information includes the Compute Capability level and details of grid dimension constraints.

2. Lazy Loading of GPU Module Instructions. This is a CUDA optimization to kernel launch capabilities that delays the loading of machine code instructions by the GPU until it is needed. It is enabled by default since CUDA 12.2, and can be controlled by setting the environment variable `CUDA_MODULE_LOADING` to either `LAZY` (default) or `EAGER`.

3. GPU Overclocking. This is something that can be considered, but it's also a risk. GPU overheating is a real problem in production environments, which is why NVIDIA has the command-line `dcgmi` GPU diagnostic tool (it's part of the NVIDIA's Data Center GPU Manager, not the CUDA Toolkit, so it's not installed by default).

The risk of GPU failure is especially high when the GPUs have been running for a while, or have been previously used in Bitcoin mining or other heavy loads. Note that GPU failure is often insidious, calculating incorrect matrix multiplications, rather than triggering a runtime error!

You can get the current GPU clock speed in deviceQuery (a CUDA sample program) or via the nvidia-smi command, or programmatically in CUDA C++ using the cudaGetDeviceProperties function and the clockRate property:

```
int device_number = 0;
cudaDeviceProp prop;
CUDACHK(cudaGetDeviceProperties(&prop, device_number));
double ghz = prop.clockRate / (1024.0f * 1024.0f);
printf("GPU Clock Rate: %d KHz (%3.2f GHz)\n",
       (int)prop.clockRate, ghz);
```

Also available are other properties, such as the memory clock speed and the width of the PCIe bus (in bits) between GPU and RAM.

Note that you cannot set any of these GPU properties via the CUDA Runtime API (i.e, there's no "cudaSetDeviceProperties"). You might be able to programmatically change the GPU clock speed using the NVIDIA Management Library (NVML) "nvmlDeviceSetApplicationsClocks" API, or in some circumstances by launching an "nvidia-smi" command as a sub-process.

4. Process Prioritization. An application running as a process on Linux can be given an increased priority when launched, using Linux operating system commands (i.e., the nice command). This can give the application priority access to the CPU, above many other less important processes. Alternatively, the nice command can also be used to lower the priority of less important processes.

If you want to run at highest priority:

```
nice -20 a.out
```

The default priority for the nice command is 10. Note that there's also the renice command to adjust the priority of an already-running process. This suggestion assumes that you have the security privileges to do so.

Programmatically, in C++ host code, you could use these APIs:

- `nice` function in `<unistd.h>` — Linux
- `setpriority` or `getpriority` — Linux
- `SetPriorityClass` in `<processthreadsapi.h>` — Windows API

Note that the `nice` C++ function adds to priority, rather than setting the value.

5. Device Property Testing. Examining the GPU device properties seems like it should be fast, but there are two ways with slightly different execution speeds. Generally, `cudaGetDeviceProperties` is slower because it has to get *all* of the properties, some of which require an expensive PCIe bus read. Depending on which attribute you seek, `cudaDeviceGetAttribute` can be much faster.

The slower attributes include:

- `cudaDevAttrClockRate`
- `cudaDevAttrKernelExecTimeout`
- `cudaDevAttrMemoryClockRate`
- `cudaDevAttrSingleToDoublePrecisionPerfRatio`

For an example of how to get every type of device property, look up the free `deviceQuery` sample in NVIDIA's Github area.

6. GPU Isolation. This is an optimization where your application only uses a single GPU on a multi-GPU system. Without paying attention to "isolating" its execution, an application will consumer resources across multiple GPUs, which wastes overall resources. This method does not optimize the execution of the application that is using isolation, but benefits the other workloads on the other GPUs. It's kind of like quarantine for silicon beings.

13. Warp Divergence

Thread Divergence

Thread divergence, also called "warp divergence" or "branch divergence," occurs when some subset of threads in a warp takes a different path at a control flow branch, such as an *if* statement or loop conditional test. The reason is rather strange: CUDA runs both branches sequentially, rather than in parallel.

Apparently, the underlying algorithm on the GPU runs like this at a conditional branch:

- Evaluate the condition on all 32 threads (in parallel)
- For all threads where the condition was true, run the first "if" path, with the other threads disabled.
- Reverse the logic, and run the code in the "else" branch, with the first subset of threads disabled.

The reasons why a GPU would work this way are somewhat involved, but basically it's a huge vector processor that's really more like SIMD than it appears. It runs the "if" and "else" sections as a non-stop sequence of SIMD operations, but uses the trick to suppress it on a subset of the 32 threads.

The good news is that this is all totally hidden from the programmer by the CUDA stack. C++ programmers can just write code for *if* statements, loop conditions, or even *switch* statements, without worrying about enabling or disabling threads in a warp. Kudos to the compiler design engineers at NVIDIA for making this work so well.

The bad news is that branch divergence causes a significant slow-down in the execution of a kernel, because it forces serial execution.

Ugh! Hence, one kernel optimization is to try to avoid divergent branches as much as possible.

Reducing Divergence

Methods to reduce thread divergence (warp divergence) include:

- Reduce use of C++ control flow statements in kernels (e.g., `if` statements, loops, `switch` statements).
- Reduce any complex side-effects in short-circuiting control flow of C++ operators (i.e., `&&`, `||` and `?:`)
- Use CUDA math functions or other CUDA SIMD intrinsic function calls.
- Hoist `if` statements out of kernels to higher-level logic.
- Use predication for `if` statements in kernels.
- Use compile-time preprocessor directives to avoid live `if` statements.
- Use conditions at the warp level (in each warp, all 32 threads follow either the `if` or the `else` pathway).
- Empty `else` code pathways (the extra serialized code does nothing).

Remember the basics. General advice on trying to minimize branch prediction by reducing *if* statements can be taken too far. The general code optimization advice follows:

- Help the compiler auto-optimize the branches (it uses "predication").
- Examine the PTX assembler to see if there's any branches happening.
- Don't micro-optimize branches in non-critical kernels that don't matter.
- A lot of other optimizations matter more than branch divergence.

Device code only. One point to make, which is probably stating the obvious, but I'm going to say it anyway: host code is fine! Branch divergence is only a problem for kernel code running on the GPU device. The basic C++ code on the host is serialized on the CPU anyway, so use as many *if* statements and loops as you like in the host code.

The compiler fixes many cases. The CUDA C++ compiler will optimize a lot of cases to avoid branch divergence. And many micro-optimizations either won't help, or will result in the same code. You can even make things worse for the compiler if you use complex semantics, because it might stop the auto-optimizer from fixing things for you. You need to examine the assembler code to determine whether your super-cool method to avoid an `if` statement is actually changing anything at all.

Another point about the compiler handling things for you: not all loops are bad! Simple loops do not cause thread divergence if every thread executes all iterations.

A simple example is a constant loop:

```
for (int i = 0; i < 8; i++) {   // Harmless
    // ...
}
```

Control flow Branch divergence can occur with any of the control flow statements where there are two or more pathways:

- `if` and `if-else` statements
- `for` loops
- `while` loops
- `do-while` loops
- `switch` statements

Note that some other C++ control flow statements that alter the flow path, but do not really introduce divergence, because they are non-conditional branches that don't offer two pathways at that point:

- `return` statement
- `continue` statement
- `break` statement

Also, as discussed further below, there are several CUDA C++ operators that have control-flow characteristics, such as the ternary operator.

Predication

The method of predication is to make the `if` and `else` clauses more explicit. The compiler will often do this optimization for you via auto-predication, because it mirrors how the GPU actually executes branches. However, you can do it manually.

The normal non-predication way of writing a branch is like this code:

```
if (condition) {
    // Code
}
else {
    // Other code path
}
```

Manual predication means writing this:

```
if (condition) {
    // Code
}
if (!condition) {
    // Other code path
}
```

There's no `else` statements in predication, but don't go crazy, because the compiler usually handles it anyway. You would also need to manually convert the ternary operator (`?:`) into `if` statements to do this manually. And operator short-circuiting is also an occasional concern.

C++ Operators and Functions

Some C++ operators have a hidden control flow aspect where two blocks of code may differ. The main ones are:

- `&&` logical and operator — short-circuited if first operand is false.
- `||` logical or operator — short-circuited if first operand is true.
- `?:` ternary operator — ternary operator has an `if-else` meaning.

The Boolean logical operators do short-circuiting in C++ and also in CUDA C++. The `&&` has an "and then" flow, and the `||` operator has an "or else" flow. But this won't really matter in regard to branch divergence in simple expressions involving basic variables (e.g., `i<10&&i!=0`), because the compiler won't really evaluate conditions using branching.

The only issues arise if you're doing tricky side effects in the second operand expression of these operators, such as assignments or increments/decrements, or function calls.

On the other hand, a ternary operator (`?:`) is definitely affecting control flow, and could lead to branch divergence. The ternary operator is not really an optimization compared to using `if-else` statements.

It's the same!

Use CUDA math functions. A lot of the traditional C++ tricks aren't that great for CUDA because of the branch divergence problem.

Here's some examples of things *not* to do in kernel code:

```
#define MAX(x,y) ( ((x) > (y) ) ? (x) : (y) )  // BAD
#define MIN(x,y) ( ((x) < (y) ) ? (x) : (y) )  // BAD
```

Also, don't do the equivalent in small *inline* functions, because the problem for branch divergence is the presence of an *if* statement, not the function call overhead:

```
inline float abs(float x)
{
    if (x < 0.0)      // BAD
        return -x;
    else
        return x;
}
```

Instead, you should use the many builtin CUDA functions inside kernels, such as these:

- `max`
- `min`
- `abs`

It seems reasonable to assume that they've been implemented in a way that doesn't suffer from branch divergence, and indeed, many of them map to single PTX instructions, with no branches or function call overhead. There are many more such functions in the CUDA runtime libraries.

Example: RELU activation function. Implementing the RELU activation function on CUDA is a good example. Don't use a manual method:

```
inline float RELU(float x)
{
    if (x <= 0.0) return 0.0;
    return x;
}
```

Instead, you can just use the builtin CUDA "`max`" function, which has various overloaded declarations for different types.

You can even use a preprocessor macro for this simple change:

```
#define RELU(x)  (max((x),0.0))
```

Research on branch divergence optimizations. There is a research area that focuses on techniques to reduce branch divergence, and how to automate them in the compiler, including papers over a decade old. Some of the basic methods are:

- Branch fusion
- Tail merging
- Control-flow melding
- Thread data remapping

I'm not going to go into details of these methods, but they sure are fun!

14. Grid Optimizations

Grid Size Optimizations

Choosing the grid size is an important aspect of optimization for CUDA kernels. The goals of this analysis include:

- Occupancy rates (GPU thread utilization)
- Load balancing of the workload over the GPU (and over multiple waves)
- Wave optimizations (execution of multiple blocks)

Risks include allocating too many threads for the GPU, or inefficient wave patterns, such as the "tail effect" of a small final wave. Choose block sizes carefully. Experimentation and benchmarking may be required to find the best size.

Grid size basics. How many blocks and how many threads-per-block should you run? The main basic constraints are that the warp size is 32 threads and the block size (i.e., threads-per-block) is at most 1024 threads. One rule of thumb is to start with block sizes of 256 or 512 threads, and it must be a multiple of the warp size (i.e., 32).

Note that there are no NVIDIA GPUs for which these constraints are higher (at least, as of this writing). However, different GPUs can have more or less blocks and an overall size of a wave (multiple blocks).

You can print out the basic grid constraints of your GPU to check they haven't changed by using `cudaGetDeviceProperties`:

```
int device_number = 0;
cudaDeviceProp prop;
CUDACHK( cudaGetDeviceProperties(&prop, device_number));
// Warp size (should be 32)
printf("Warp Size: %d threads\n", (int)prop.warpSize);
// Maximum Threads per block (should be 1024)
printf("Max Threads Per Block: %d threads\n",
       (int)prop.maxThreadsPerBlock);
```

You should get this output:

```
    Warp Size: 32 threads
    Max Threads Per Block: 1024 threads
```

The block size should be a multiple of 32 threads, which is the warp size. If you don't, the program will still run, but the GPU only schedules threads on a per-warp basis, so all the other "extra" threads will be idle, and unavailable to you or any other GPU kernel.

A simple way to ensure this inefficiency doesn't accidentally occur:

```
    assert(threads_per_block % 32 == 0);
```

You can use the optimization "`&  0x1F`" if you don't trust compiler design engineers (but don't forget the extra parentheses!).

Block size limit is 1024. The block size is also limited to a maximum of 1024 threads, and common sizes in CUDA programs are 256 or 512, but it can be exactly 1024 if you prefer.

However, if you choose a block size of more than 1024 threads, the kernel launch will immediately fail with an error. Weirdly, this type of synchronous kernel launch error is actually an insidious error, because it seems to get lost if you don't check for it right away.

```
    mykernel <<< blocks, 1025 >>> (); // BUG!
    CUDACHK( cudaMemcpy(...) );  // Won't fail!
```

Note that kernels cannot return a value and must be of type `void`. So, you can't do:

```
    err = mykernel <<<blocks, 1025>>>(); // Compile error!
```

And similarly, this idea fails:

```
    // Compile error!
    CUDACHK( mykernel <<< blocks, 1025 >>>() );
```

The way to actually catch this synchronous kernel launch error is with an immediate call to `cudaPeekAtLastError` or `cudaGetLastError`:

```
mykernel <<< blocks, 1025 >>> ();
CUDACHK(cudaPeekAtLastError(...)); // Fails (correctly!)
CUDACHK( cudaMemcpy(...) );
```

Since we may not want the performance cost of the implicit synchronization this causes, a better solution for catching these synchronous kernel launch failures is either:

(a) Don't do that!, or

(b) Put `cudaPeekAtLastError` or `cudaGetLastError` inside a self-testing "`#if DEBUG`" sequence that disables the synchronizing function calls for non-debug builds, or

(c) Use an `assert(threads_per_block<=1024)` before the kernel launch, which can be compiled-out by setting `NDEBUG`.

Balancing workload. One way that can help to balance the workload in the GPU is to choose a number of blocks that is an exact multiple of the number of streaming multiprocessors on the chip.

You can get this number as a device property via `cudaDeviceGetAttribute` and the property `cudaDevAttrMultiProcessorCount`:

```
// Streaming Multpirocessor count
int smcount;
CUDACHK( cudaDeviceGetAttribute(&smcount,
         cudaDevAttrMultiProcessorCount, 0));
printf("Numbers of SMs: %d\n", smcount);
```

This number can then be used in the block count calculation.

Note that this method does not at all guarantee workload balancing, but does give the GPU more chance to do so.

Wave Optimizations

In CUDA, a "wave" is a set of thread blocks running in parallel. Hence, it is a high-level concept of a large amount of parallel execution. Optimizing the number of waves is an important aspect of choosing the number of blocks and block size for your kernel.

Generally, if you can parallelize your algorithm well enough, you would only want one wave, so that all of your workload finishes immediately. But it's common that that would exceed the number of threads possible on a GPU, so it has to be split into multiple waves.

Tail Effect (Small Final Wave). One aspect of wave optimization is to avoid the "tail effect" at the end of your algorithm. This refers to having a smaller wave at the end, because the early waves didn't quite finish off all of the workload. Hence, you get a final wave with a very low occupancy level.

The tail effect occurs when computation is split over multiple waves. If the algorithm maximizes occupancy for the start of the algorithm, there is often a much smaller last wave that does any left-over computations. This final phase has a low occupancy, and the overall GPU utilization can often be improved by a more balanced allocation of computations to slightly smaller waves, or to a "single-wave kernel" where possible.

One tip that can be useful for wave optimization is to consider SM balancing in your grid and block size values. Choosing a grid size where the number of blocks is a multiple of the number of multiprocessors can encourage a load-balanced execution by the GPU, although it does not guarantee it. This would mitigate the tail effect if the heuristic is effective. Note that you can programmatically find the number of SMs:

```
// SMs on the GPU
int device_number = 0;
cudaDeviceProp prop;
CUDACHK( cudaGetDeviceProperties(&prop, device_number));
printf("SMs on GPU: %d SMs\n",
       (int)prop.multiProcessorCount);
```

Single-Wave Kernels The idea of a single-wave kernel is simply to design the algorithm in such a way that each kernel can run as a single wave on the GPU, achieving maximum occupancy. Achieving this may involve an algorithm re-design and careful computation of grid dimensions.

Occupancy Optimization

High occupancy rates are an important goal in selecting the grid size. Occupancy is measured as the number of active threads as a percentage of the total theoretically possible threads in a GPU. There is a useful "occupancy calculator" and also an "occupancy API" available from CUDA to help.

Occupancy API. There are some CUDA API functions that can be useful in calculating the best grid and block size dimensions for achieving occupancy. These are defined in "`cuda_occupancy.h`" and note that this is auto-included via "`cuda_runtime.h`" by `nvcc` for "`.cu`" files.

Some of these API calls are:

- `cudaOccupancyMaxPotentialBlockSize`
- `cudaOccupancyMaxActiveBlocksPerMultiprocessor`
- `cudaOccupancyMaxPotentialBlockSizeVariableSMem`

Additional properties of the GPU and the SMs can be queried via the `cudaGetDeviceProperties` function. Note that these are read-only properties of the hardware and there's no "set" API.

There are dozens of them, but some are:

- `cudaDevAttrMultiProcessorCount` — SMs per GPU.
- `totalGlobalMem` - total global memory.
- `sharedMemPerBlock` — shared memory per block.
- `warpSize` — 32 always!
- `regsPerBlock` — maximum registers per block.
- `maxThreadsPerBlock` — maximum threads-per-block (1024).
- `maxThreadsPerMultiProcessor` — maximum threads per SM.

The function `cudaDeviceGetLimit` can be used to examine three other configuration settings, which are read-write settable GPU limits:

- `cudaLimitPrintfFifoSize` — printf FIFO buffer size
- `cudaLimitStackSize` — Stack size
- `cudaLimitMallocHeapSize` — Heap size

General occupancy techniques

If you're not achieving a high occupancy, there a number of optimizations to consider:

- Too few warps. If you're not allocating enough threads in warps to utilize the SM, then your occupancy will simply be low. Consider increasing the parallelization of your algorithm so that more warps are used. But don't go too high either.
- Warps per kernel: choose your block count and block size so that the total number of warps in your kernel is exactly the maximum number of warps allowed for a multiprocessor.
- Blocks per SM: ensure the number of blocks in a kernel is adjusted according to the SM maximum allowed. If may be desirable to adjust the block size to contain more warps, and reduce the number of blocks.
- Register bottlenecks. There are only a limited number of registers on each SM. Hence, using too many registers for your kernels may restrict occupancy. Consider reducing the allowed number of registers via compiler options, although be aware that this is reducing memory efficient, and comes at a runtime cost.
- Shared memory bottlenecks. There is also a restricted amount of shared memory available on each SM. This can be an area that reduces your occupancy rates. Review the amount of shared memory being used by your kernel.
- Unbalanced threads. If your kernel has a lot of threads at the start, but reduces to fewer active threads, it is said to be unbalanced. This can occur in many types of kernels, notably reductions, and may lead to poor load balancing over the SMs.
- Unbalanced blocks. Similarly, if some of the blocks remain active for longer than others, this is unbalanced. Review the block-level activity across your kernel for effects on occupancy in the multiprocessors.

15. Compile-Time Optimizations

CUDA 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 or GPU 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.
- Loop unrolling — `#pragma unroll`.
- 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 — avoid aliasing-related slowdowns.

Conditional Compilation

The CUDA C++ preprocessor can be used to simply remove code from the program. You can just go crazy and put `#if` directives around any statements you like, and your program will definitely run a lot faster, but this idea is more typically used to remove:

- Debug code
- Self-testing code
- Assertions
- Tracing code

The way to remove such code is to define the preprocessor macros (e.g., using the "`-D`" option for `nvcc` in a `Makefile`), or more precisely to *not* define macros like "`DEBUG`" or "`SELFTEST`" or whatever you've used. For the builtin `assert` macro, you can define the "`NDEBUG`" (no debug) macro flag to compile them out.

It's a policy decision on whether to leave self-testing code in for production or not. There's a supportability benefit to leaving it in, but a performance cost to doing so. If you want a lot of meetings, ask the Marketing Department what to do about that.

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 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 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, should 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 object variables 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 any 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 capabilities 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 in 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 lots of 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 like a hybrid between `consteval` and the old-school `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 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 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 the more familiar `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.

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 whole function body with the "`__restrict__`" keyword. This is placed after the closing right parenthesis of the function parameters (i.e., similar to a `const` member function declaration). The declaration looks like:

```
    void MyClass::myfunc(int x) __restrict__;
```

Templates for Kernels

You can define templated versions of CUDA kernels, much as you would for standard C++ kernels. The advantages of templated kernels include:

- Extra compile-time auto-optimizations from the use of constant parameters.
- Multiple specialized versions of your kernels can be defined.

In other words, templating your kernels allows either you or your compiler to do a better job optimizing.

To define a templated CUDA kernel, just use the "`template`" keyword, much like standard C++ templates. The basic syntax is:

```
    template < /*...template params ...*/ >
    __global__ mykernel( /*... kernel params ...*/ )
    {
        // kernel code...
    }
```

In other words, it looks exactly like a standard C++ templated function, except for an extra "`__global__`" or "`__device__`" specifier. And you can also invoke a templated kernel using the same ideas:

```
  mykernel< /* template params */ >
        <<< blocks,threads >>> ( /* kernel params */ );
```

For example, a templated kernel launch would look like:

```
    vector_clear <<< blocks,threads >>> (dv, n );
```

It looks a little messy with "<...>" for `template` parameters and "<<<...>>>" sequences for the kernel launch, but it works.

Template Compile-Time Optimizations

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
- SFINAE techniques
- Template Metaprogramming (TMP) 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 Metaprogramming

Template metaprogramming, often abbreviated TMP, is weird. Several years ago, some bright spark figured out that the C++ `template` syntax was Turing-complete, because it has:

1. Sequence,

2. Selection, and

3. Recursion

Sequence and selection are trivial with multiple statements and `if` statements. The third requirement is usually "iteration" with loops, but the use of well-defined recursion is actually equivalent. There's also the fourth requirement of "data storage" performed by the compiler as it parses the code.

Voila! We have a fully-fledged programming language.

Although there aren't any loops in C++ template declarations (yet?), you can have one template declaration declare another, for a different value. Hence, there are "recursive" `template` definitions (e.g., "`factorial<n-1>`"), and you can separately declare a "base case" for a particular constant value (e.g., "`factorial<1>`").

This amazing insight means that you can use a fancy `template` definition to perform full computations at compile-time. For example, there are TMP "programs" that compute the Fibonacci numbers or factorials.

Here's an example:

```
template
struct Factorial
{
    // Recursive case N! = N * (N-1)!
    enum { value = N * Factorial::value };
};

// Base case 1! = 1
template <>
struct Factorial<1> {
    enum { value = 1 };
};

void print_factorial()
{
    printf("Factorial 5 = %d\n", Factorial<5>::value);
}
```

This code "tricks" the compiler to use the standard C++ template machinery to evaluation `5*4*3*2*1=120` at compile-time.

And I have to tip my hat to whoever figured that out! According to Wikibooks it was discovered accidentally by Erwin Unruh.

Unfortunately, or perhaps fortunately, if you let your excitement subside, there are two points to consider:

- TMP only works if the input value is a constant, and
- Standard C++ now has "`constexpr`" which does almost the same thing in a non-ugly syntax.

So, I'm not sure how much time you really should spend trying to optimize your CUDA kernels with TMP, but trying to will definitely grow your brain size!

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. Wikipedia, Oct 2024 (accessed), *Template metaprogramming*, https://en.wikipedia.org/wiki/Template_metaprogramming
4. Wikibooks, Oct 2024 (accessed), *History of TMP*, https://en.wikibooks.org/wiki/C%2B%2B_Programming/Templates/Template_Meta-Programming#History_of_TMP
5. Arthy Sundaram, Jaydeep Marathe, Hari Sandanagobalane, Gautam Chakrabarti, Mukesh Kapoor, Steve Wells and Mike Murphy, Feb 12, 2021, *Boosting Productivity and Performance with the NVIDIA CUDA 11.2 C++ Compiler*, https://developer.nvidia.com/blog/boosting-productivity-and-performance-with-the-nvidia-cuda-11-2-c-compiler/

16. Arithmetic Optimizations

Arithmetic Overload

You wouldn't be reading a CUDA book unless you had a lot of arithmetic to do! Crunching through large volumes of data is where a GPU shines. The methods of optimization to fix an arithmetic bottleneck are basically in two big categories:

- Do some other type of arithmetic instead.
- Do fewer computations.

Alternative forms of faster arithmetic include changing to integers or doing bitwise shifting or addition. The ways to do fewer arithmetic computations tend to involve higher-level algorithmic changes to your application or reducing the amount of data.

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)

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.

General Arithmetic Optimization Methods

Smaller Data Sizes (Quantization). Reducing the size of data from 32-bit floating-point to 8-bit integers (or 4-bit) is effective at reducing both arithmetic computation costs and memory access costs. The computation kernels for 4-bit tensors are much more efficient than for 32-bit calculations.

Low-Precision Arithmetic. Choose `float` over `double`. Also prefer the lower-precision mathematical library functions, such as `expf` rather than `exp`.

Integer arithmetic. Calculations involving integers are faster than in floating-point. This is another advantage of quantizing tensors from 32-bit floating-point to smaller integers. Even 32-bit integers may have a lower compute, but don't have any memory benefit from a smaller data type.

Bit Parallelism (Bit-Level Kernels). Integer operations can be considered as 32 or 64 bit operations in parallel. This idea is used in data structures such as bit vectors and Bloom filters, or AI kernels such as low-bit quantization. Using these in CUDA adds a second layer of concurrency beyond thread parallelism, thereby even further parallelizing the algorithm.

Instruction Latency. Some hardware instructions are simply faster than others. Different generations of GPUs have instructions with improved speed, and additional machine-code instructions. Knowing the exact speed of individual instructions can be used to optimize by inserting inline PTX assembly with the `asm` statement.

String Data Processing. The C and C++ style of null-terminated string data is not well suited to parallel processing, being somewhat linear in nature. In general, prefer string representations with a fixed size, or at least a "length" variable, rather than null-terminated storage where the length is not known. On the other hand, there are several CUDA C++ libraries for text processing on a GPU, such as RAPIDS, which are very fast.

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`)

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. Note that there's a whole class of AI engines that rely on this integer multiply-vs-bitshift optimization called "logarithmic quantization" or "power-of-two quantization." 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: The use of bitshifting works for division, too (but only for unsigned):

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

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
```

Avoid % Remainder Operations

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

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

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
```

Another 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 the bitwise version of remainder:

```
    x % 256
    x & 255  // Bitwise version
    (unsigned char) x   // Type cast version
```

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), or even more efficiently with a 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 helpful to make life easier for the optimizer (and these poor critters need a break every now and then).

If the divisor is a complex expression, the compiler might not automate the use of 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);
```

And we really should pre-calculate this constant using constant folding and a `const` variable:

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

Implicit 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, having been burned a few times, and doesn't occur 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 real 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 suffix "`f`" on either usage, 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.

17. Floating-Point Bit Tricks

Bits in Floating-Point Numbers

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 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 the modern C++ way with the newer `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 bit coding, because ">>" right-shift 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

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 for 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:

```
assert(sizeof(int) == 4);
assert(sizeof(short int) == 2);
assert(sizeof(float) == 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`.

Floating-Point Bit Tricks

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 integer bitwise arithmetic on the underlying 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` integer. 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
```

Add-as-Integer 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 above 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 of 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 an offset of 15 (not 127) for 5-bit exponents.

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 builtin special functions such as `ldexp` and `scalbn` for bitshif on `float` data. The `ldexp` function accepts an integer power, and then shifts a floating-point value. The `ldexp` function is for `double`, `ldexpf` for `float`, and `ldexpl` for `long double`.

The scalbn set of functions appears to be almost identical to ldexp functions. There is also a reverse function "frexp" which extracts the significand (fraction) and power-of-two from a floating-point.

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 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.

Log2 of Floating-Point is the Exponent

The `log2` function for `float` types is a non-linear function that is quite expensive to compute. There's also a bitwise trick for `log2` of floating-point numbers. 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) to extract exponents: `frexp`, `ldexp`, `ilogb`, and `scalbn`.

References

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.)

18. Advanced Techniques

Advanced CUDA Kernel Declarations

The simplest types of CUDA kernels look like basic C++ functions with an extra "`__global__`" specifier. However, you can also use many of the more advanced types of functions declarations to offer a wider variety of kernel functions.

Examples include:

- Class member functions — defined in the style: `MyClass::mykernel`
- Function overloading — offer multiple versions of the kernel.
- Templated kernels — using the "`template`" C++ keyword.
- Function pointers — uses random `*` and `()` sequences and yet somehow works.

Nevertheless, be aware that the kernel code is not running in a standard C++ environment, and the NVCC compiler does not support every type of capability for kernels. For example, here are some of the limitations:

- No return type — you can't define a non-`void` kernel that returns a value.
- Reference parameters not supported.
- Default arguments for parameters not supported.

Function Pointer Kernels. Another trick you can do with kernel functions: *use function pointers.* Although they're a little mind-bending at first, function pointers are supported for kernel launches (and other things) in CUDA C++, and have been a part of standard C++ for decades.

This idea is very useful for wrapping your kernel launches in boilerplate code, such as the calculation of blocks and threads for the grid dimensions, and managing the up-and-down host-device data transfers.

Here's the idea of a general kernel launch wrapper to do different vector operations:

```
void kernel_launch(
       void (*myvectorfnptr)(float *v, int n),
       float *v, int n)
{
    // Set up ... e.g., cudaMalloc, cudaMemcpy, etc.

    myvectorfnptr <<< blocks, nthreads >>> (f, n);

    // Synchronize ... e.g., cudaGetLastError
    // Get results... e.g., cudaMemcpy
    // Cleanup ... e.g., cudaFree
}
```

And here's how you'd call it:

```
__device__ void vector_sum(float *v, int n)
{
    // Sum reduction of vector ...
}

// ....
kernel_launch(vector_sum, dv, n);
```

In the call to the "`kernel_launch`" function, we have passed the function name "`vector_sum`" as a function pointer. Note that we didn't use any "`()`" after the name. The name of a function when used by itself with parentheses is a function pointer in CUDA C++, and it points to the code for that function. We can call a function pointer just by adding a `(...)` sequence after it. Function pointers can have parameter lists, or can have zero parameters.

Persistent Kernels

Nobody's telling your kernels they have to exit. Non-exiting threads can remain resident in the GPU, and receive more work in a type of job scheduling architecture. This can be very efficient for some types of kernel where there is a continual stream of work (e.g., batching or chunked prefill in AI engines).

The advantages of persistent threads include:

- Avoids kernel launch overhead
- Low latency (fast response time to new queries)

The disadvantages of a persistent kernel architecture include:

- Monopolized GPU
- Code implementation complexity
- Busy wait
- GPU never idle (consumes electricity)
- Kernel timeouts

The architecture of a persistent kernel has the threads simply waiting for work to be queued. Typically, this would come from the host code, which acts as the overarching controller of the work queue.

Why do you need the GPU threads to be persistent? Why not have the host code wait for work and then launch the kernel threads? The main reason is to avoid the delay from kernel launches, thereby increasing response times.

Similarly, a persistent kernel could have a small kernel managing the work queue, and then launch a larger kernel via dynamic parallelism. But that also undermines the advantage of having the large kernel pre-launched, so as to have a very low latency.

Note that there have been some problems, whereby a persistent long-running kernel starves a non-persistent kernel. These are related to the "lazy module loading" optimization, controlled by the `CUDA_MODULE_LOADING` environment variable (i.e., change to "`EAGER`" from default value "`LAZY`").

Alternatively, a workaround is to dry-run the non-persistent kernel prior to launching the persistent kernel, thereby forcing it to be loaded once in the GPU.

Generally, it would make sense to use the persistent kernel architecture only when there is an expectation of a continual feed of work jobs. In that situation, the downsides are limited, because the GPU is always busy.

Kernel Timeouts

Actually, your GPU can definitely tell your kernels to exit, and it doesn't ask nicely. There are "kernel timeouts" where the GPU will kill a kernel after a set duration in time. Running a persistent kernel requires taking control of these timeouts.

You can determine programmatically in CUDA C++ whether kernel timeouts are enabled on the GPU by using the `cudaGetDeviceProperties` runtime API and the `kernelExecTimeoutEnabled` property.

```
// Kernel timeout property (enabled/disabled)
int device_number = 0;
cudaDeviceProp prop;
CUDACHK( cudaGetDeviceProperties(&prop, device_number) );
bool kernel_timeout = (prop.kernelExecTimeoutEnabled != 0);
printf("Kernel timeout: %s\n",
    kernel_timeout ? (char*)"enabled" : (char*)"disabled");
```

If you're running a persistent kernel, it would make sense to check that the timer has been disabled, and emit a severe error if not.

The length of a CUDA timeout is typically 5 to 10 seconds. When a timeout occurs for a kernel, a runtime error is raised: `cudaErrorLaunchTimeout` (702), also known as `CUDA_ERROR_LAUNCH_TIMEOUT` in those heady days when we used underscores in our iconoclastic years.

You can't directly change the kernel timeout settings from within CUDA C++. There are operating system configuration settings that can be modified, including Windows registry settings, disabling the Watchdog timer, adjusting Linux kernel parameters, or altering the configuration properties of the GPU driver.

Lookup Tables

Lookup tables are such a well-known optimization that they're just called LUTs. The idea is to precalculate a lot of data results, thereby avoiding that processing cost at runtime.

AI engines provide some good examples. Let's say you have a vector of intermediate results ("activations") and you want to apply the *sigmoid* function to them:

```
sigmoid(x) := 1 / (1 + exp(-x))
```

In basic C++, the code is:

```
float sigmoid(float x)
{
    return 1.0f / ( 1.0f + expf( - x) );
}
```

So, we know how to write a super-fast GPU kernel to scan all the vector elements in them, since this is an element-wise algorithm (i.e., it's actually "embarrassingly parallel" in the vernacular).

The problem is that each of our kernel threads is going to have to compute the above function, which includes an exponential and some other operations. Isn't there a faster way?

Well, the first point is this: what are you worried about? There's a builtin CUDA function for exponentials, and it's basically a single machine code instruction these days, because hardware engineers have long since coded these basic mathematical functions in microcode. Then it's a negation, addition, and division, so it's not really that much arithmetic cost, and we can do them all in parallel with GPU threads.

Anyway, back to my point about LUTs. If we wanted to super-optimize it, we could precompute this computation for all 32-bit input `float` values, and then have a table of `float` values that are the results. The input is the bits from the `float` converted to a 32-bit unsigned integer (as in Chapter 17), which is used as the array index. The code looks something like this:

```
    float sigmoid_precomp(float x)
    {
        unsigned int offset = *(unsigned int*)&x;
        return g_sigmoid_LUT[offset];
    }
```

As you can see, this is faster! How big is the precomputed "`g_sigmoid_LUT`" table? Well, it's 2^32 (approximately 4.7 billion) times four bytes for the `float` results values, This is therefore about 19 Gigabytes. It's a little more than will fit into the GPU's 48KB constant cache, but we shall persevere anyway.

We could use a smaller cache. For example, the rightmost bits in a `float` are the least-significant digits of a floating point number. If we ignore 8 of them, we lose about 2 or 3 digits, which are not important. Hence, the code becomes:

```
    float sigmoid_precomp(float x)
    {
        unsigned int offset = *(unsigned int*)&x;
        offset >>= 8;
        return g_sigmoid_LUT[offset];
    }
```

Note that it's a good thing we're using an `unsigned` integer type, because right shift is undefined on signed integers, if they're negative. It might shift-in zeros on the left, or it might sign-extend.

How big is our LUT? Well, now it's 2^24 times 4, which is about 16.7 million times 4, so our LUT is about 67 Megabytes. It's still too big for our caches or constant memory, but it will fit in GPU global memory. In order to get it into global memory, we need to either:

(a) upload it from the CPU each time the kernel is launched, or

(b) declare a global `__device__` array for the LUT data, or

(c) use a persistent kernel with that memory uploaded once.

I feel like our GPU needs a ROM chip here, but I don't think it has one. The easiest solution is to declare a big global array with the `__device__` specifier, which allows CUDA runtime to manage that memory, with paging in and out of GPU memory, and the table has lifetime of the application.

There's another problem, though. If each of our threads is computing the `sigmoid` of one vector element, the values in the vectors won't be the same. In fact, they'll be effectively random, so each thread will be accessing a random index into the LUT. These are certainly not adjacent addresses, and so it's not a coalesced access pattern into the global memory.

After all this, we come to a sudden realization about LUT algorithms, and their characteristics:

- Large data requirement for the precomputed LUT, and
- Non-coalesced almost random pattern of accessing the LUT.

And then there's the dreaded realization of an alternate reality:

It might be better on the CPU.

So, them's your choices:

- Use parallel kernel threads without a LUT,
- Upload 67 MB every time to GPU global memory.
- Set aside 67 MB global memory and run a persistent kernel.
- Run it on the CPU instead.

You pays your money and you takes your chances.

Source Code Precomputation

No matter what method you choose to implement your LUT, there's the problem that you need to fill the LUT by computing the `sigmoid` function about 16.7 million times for a 24-bit LUT (67MB size), or 4.7 billion times for a 32-bit LUT (18.8GB size).

When are you going to run them? You could do it at application startup time, although that will be a bit like an old Windows box starting up. Maybe you could code the whirring disk drive sound-effects for fun? Alternatively, you could do the calculations offline and write them to a binary file, and then you can load the binary file at startup time.

There's a better solution: let the CUDA C++ compiler sort it out. Instead of writing our results to a binary file, here's the trick:

Create a C++ source code file.

It's just a single variable name, with an array initializer about a mile long. Then you can compile that C++ source code file, just like any other code. Our program looks like this:

```
void generate_sigmoid_LUT(FILE *fp, bool gpu)
{
    unsigned int maxi = (1<<24);
    if (gpu) fprintf(fp, "__device__ ");
    fprintf(fp, "float g_sigmoid_LUT[]={\n");
    for (unsigned int i = 0; i < maxi; i++) {
        float f = *(float *)&i;
        fprintf(fp, "%10.10f", sigmoid(f));
        if (i + 1 < maxi) fprintf(fp, ",");
        if (i % 5 == 4) fprintf(fp, "\n");
    }
    fprintf(fp, "};\n");
}
```

It's not the tightest code I've ever written, but this is to run offline anyway.

The output is huge and looks like this:

```
  __device__ float g_sigmoid_LUT[]={
      // ... lots of data
      0.6224593520,0.6224593520,0.6224594116,0.6224594116,0.6224594116,
      0.6224594116,0.6224594116,0.6224594712,0.6224594712,0.6224595308,
      0.6224595308,0.6224595308,0.6224595308,0.6224595308,0.6224595308,
      // ... more data
  };
```

There are practical problems with the file size, however. If our LUT has 16.7 million numbers, and we use 10 digits in ASCII for the numeric constants in source code, our output file will be over 167 MB, which is quite a large C++ source file for the `nvcc` compiler to handle.

When I tested it, my generated CUDA C++ file was 221MB, so I tried to compile it. Unfortunately, `nvcc` is probably still spinning and thinking, as you read this. Just kidding, but it did take a few minutes. Kudos to all the compiler engineers at NVIDIA!

Appendix: CUDA C++ Slugs

Slug Hunting Advice

This appendix is about speeding up your C++ programs through general improvements to sequential or parallel coding. A more detailed analysis of these various techniques with code examples is found in the bonus materials online for this book: https://www.aussieai.com/cuda/optimization.

But, before we begin with anything that's actually useful, I have to introduce the obligatory wrist-slapping politically-correct deslugging advice for programmers. Hence, here are some general nuggets of advice when attempting to speed up your program:

- Profile twice, code once. Performance profiling tools exist for a reason. Find your busiest kernels.
- Focus on parallelization of sequential code to GPU kernels.
- Don't micro-optimize. Unless you're into that kind of thing. But really, try to sit on your hands.
- Do macro-optimize. Think about your data structures and algorithms.
- Optimizing introduces new bugs. 100% guaranteed! Don't optimize the night before your release. Re-run your test suite.
- Don't optimize exception handling. Tweaking rarely-executed code is a poor use of your geniousness.
- Use open source third-party libraries that have already been optimized by others.

Or just ignore that advice and go crazy. It's just too much fun optimizing when the alternative is dreary debugging.

Pro tip: it's even more fun writing a book on optimizing!

Where to hunt slugs? Some of the common large-scale issues with coding inefficiency in typical C++ programs include:

- Sequential algorithms
- Function call hierarchies
- Nested loops
- Overuse of memory allocation
- Constructor and destructor inefficiencies
- Inefficient sequential algorithms (e.g., linear search of arrays)
- Unnecessary overhead or wrappers
- Recursion. After you've coded up your university assignments (Tower of Hanoi, anyone?), please forget recursion exists.

C++ Speedup Techniques: Some of the general ways to speed up C++ programs at the design structure or algorithmic level include:

- Parallelize via CUDA kernels.
- Further parallelize via multi-kernel, multi-threading, multi-process, multi-core, multi-GPU, multi-something.
- Faster data structures (parallelizable linear ones preferably).
- Faster algorithms (e.g., restructure algorithms for vectorization).
- Vectorization (parallelize your important linear loops).
- Precompute expensive functions into a lookup table at compile-time (e.g., activation functions).
- Cache any complex calculations to trade extra space for time savings (e.g., KV caching).
- Change floating-point to integer operations (quantization, anyone?)
- Replace recursion with iteration. Subtract ten bonus points if you need to do this.

Some of the high-level C++ coding optimizations include:

- Flatten function call hierarchies (stop wrapping everything so much, and inline the small functions at the bottom).
- Optimize loops, especially nested loops (e.g., move loop-invariant code out, loop unrolling, vectorization, etc.)
- Templates are effectively a compile-time optimization that improves speed at the cost of code space.
- Reduce memory allocation (use less memory overall or replace memory allocation with temporary stack buffers).
- Operator strength reduction (e.g., replace "*" with "+", a pipe dream of all AI engineers).

- Declare variables as close as possible to where they are used. This avoids instantiating objects that aren't needed on some paths.
- Use pointer arithmetic, especially for loops over arrays.
- Bitwise operations are fast, but the basic C++ integer operations are also fast too, nowadays. Benchmark, don't assume.
- Use the short-circuiting standard property of the `&&` and `||` operators, and also the ternary `?:` operator, to avoid expensive function calls.

And finally, some things you might forget (and some that are forgettable):

- Benchmark any important changes (e.g., operator strength reductions).
- Turn up your C++ optimizer. There are higher settings you could try.
- Add compile-time optimization hints (e.g., `constexpr` and `restrict`).
- Overclock your GPU (like a gamer).
- Sell your car to buy a better GPU.
- Put every function in a header file and make them all `inline`.
- Reorder your `case` labels. Surely it helps.
- Change all uses of `i++` to `++i` in everyone else's code.

Slugs in CUDA C++

The whole point of CUDA is to replace sequential sluggish code with streamlined SIMD kernels. But there are a few ways to go wrong and actually introduce slugs into device code.

Computational slugs:

- Launching too many kernel threads (wasted resources)
- Kernels duplicating the same work (wasted resources)
- Forgetting to enable the NVCC optimization flags on both host and device code.
- Low occupancy rates on the GPU
- Tail effect of a small final wave
- Block size not a multiple of 32
- Segmented loop kernels (prefer grid-stride loops)
- Nested kernels (CUDA dynamic parallelism) can get out of control (e.g., too many threads, indirect recursion).

Thread execution bottlenecks:

- Avoid recursion (generally a poor idea overall, except in school assignments, but here it also uses up GPU local memory on the stack).
- Loop unrolling with care (can increase register pressure and causes non-coalesced memory accesses)
- Output in kernel code (i.e., `printf` calls run slow and have weird buffering delays)
- Complex class objects as kernel parameters (pass-by-reference is not supported and pass-by-value will run the constructor).
- Device debug mode — Don't compile production code with `nvcc -G` or `--device-debug`, which turns off all the auto-optimizations on kernel code (can also use both by adding the option: `--dopt`).
- `nvcc` debug and profiler modes — take care whether to use various debug and profiler flags in production: "`-g`", "`-pg`", "`-profile`", "`-debug`" or "`-lineinfo`"/"`--generate-line-info`" in production.
- Debug tracing output or failing assertions slowing down execution.

Instruction execution slugs:

- Warp divergence (from `if` statements or loops in kernels)
- Low instruction-level parallelism
- Poor instruction locality and instruction cache misses
- Overly large thread code size (bloated kernels)
- Non-restricted pointer declarations

Synchronization slugs:

- Too many calls to `cudaDeviceSynchronize`
- Overuse of `__syncwarp()` or `__syncthreads()`
- Redundant barriers (synchronization barriers that don't actually stop any race condition)
- Implied hidden synchronizations in many CUDA runtime API calls (e.g., `cudaGetLastError`, `cudaMemcpy`, etc.)
- Overuse of unnecessary atomics

Sequential execution slugs (ugh!):

- Serialized kernel execution of multiple different kernels
- Accidentally leaving the "kernel blocking" environment variable enabled.

Memory usage slugs:

- Using too much global memory (or repeatedly accessing the same data)
- Memory leaks (host or device memory)
- Non-coalesced data access patterns
- Excessive data transfer costs
- Using pageable memory for data transfers (use pinned memory)
- Poor data locality in global memory usage
- Array-of-Structures (AoS) layout has non-contiguous data
- Using shared memory for inter-thread data movement when warp shuffle operations would be faster
- Contention in memory accesses (shared memory or global memory)
- Register spills and "register pressure"
- Cache inefficiency
- Shared memory "bank conflicts" (access contention in shared memory accessed by multiple threads).
- Non-linearized data layout for matrices (2D) or tensors (3D)

C++ Class Slugs

The C++ class features are designed to add encapsulation and modularity, while retaining speed, but there's still plenty of ways that slugs can crawl into your classes. C++ class optimizations include:

- Ensure small member functions are `inline`, especially those that do "get" and "set".
- Add `inline` to other `friend` or non-class functions (esp. if small or commonly used).
- Pass objects to functions using "`const&`" (pass-by-reference), rather than pass-by-value.
- Watch out for temporary objects. These can occur in simple assignments or function call expressions or in weird ways like accidentally making your overloaded assignment operator have the wrong type.
- Avoid `virtual` functions (they have a runtime cost and block compile-time optimizations like inlining and `constexpr`).
- Specialize inherited member functions in derived classes.

- Declare objects as close to first use as possible.
- Initialize objects at declaration, rather than calling a constructor and then later an initialization function.
- Overloaded binary operators can be a slug returning an object. Instead of `c=a+b`, use a three-parameter function with references, such as `add3(a,b,c)`.
- Use reference variables instead of copying objects into temporary variables.
- Take care templating class objects (e.g., when using the `std::vector` class for a vector of your class objects). Lots of hidden calls to constructors and destructors may arise in resizing.
- Don't return objects as the return type of the assignment operators.
- Declare assignment operators as return type `void`.
- Use the initializer list in the constructor for initializing data members.
- Use `friend` functions for faster accesses to internal object data.
- Block accidental calls to the copy constructor or class assignment operator (i.e., if you aren't defining them, make a dummy version that is "`private`" with a "`void`" function body).
- Avoid returning objects if you can. Return a reference if it's safe to do so.
- Take care with "wrapper" classes like "smart pointers", "smart integers" or "smart buffers". Usually, they're safer but slower. How smart is that?
- Singleton classes can be made more efficient than general classes.
- Postfix overloaded assignment operators are inherently inefficient (make them return `void`, or disallow them as a `private` declaration with `void` body).
- Destructor execution with cleanup code can be a slug on program exit (consider avoiding memory deallocations in destructors by setting a global "`i_am_shutting_down`" flag).
- Reduce object byte size with "largest to smallest" member ordering.
- Put the most-used data member first in the object (it has a zero offset, which is compiled-out).

Function Slugs

Functions are an important building block of your code. Some ways to get the slugs out of general non-member functions include:

- Declare small functions `inline`.
- Scour open source libraries for versions of your function already written by experts (e.g., the many CUDA source code libraries).
- Avoid recursion (use loops instead).
- If you can't fully remove recursion, use (a) tail recursion elimination, (b) collapse recursive calls by unwinding or pre-computing more of the lower-level base cases, or (c) use a stack data structure.
- Call builtin or intrinsic functions.
- Call library functions rather than writing your own.
- Avoid function pointers (they block compile-time optimizations like `inline` and `constexpr`).
- Optimize the most frequently called function (profile it!).
- Use parameters as if they're local variables (reduces register usage).
- Use `constexpr` functions.
- Pass objects by reference.
- Use `const` references for objects to allow auto-optimizations by the compiler.
- Declare pointer parameters using `__restrict__` for additional compiler optimizations.
- Avoid function pointers.
- Optimizing functions with default arguments by creating a separate specialized function without the extra parameter (i.e., remove the default argument one).

Medium-Sized Slugs

There are a lot more examples of possible inefficiencies in C++ coding. Some of the types of errors that are "medium-sized" slugs include:

- Avoid non-`static` automatic array initializations with constant data.
- Don't put function calls in a loop test (i.e., expensive loop conditional tests).
- Member initializations in the constructor body (they should be in the initializer lists).
- The new `for` loop auto-iterators are elegant, but not necessarily the fastest way.
- Program startup hidden initializations (global or `static` object constructors).
- Small non-`inline` functions called frequently.
- Busy wait loops are a worst case.
- Disk I/O with file operations is often slow (e.g., load the whole file into memory) .
- Don't accidentally call your global initialization routine twice (use a flag to check it's already run).
- Unnecessary code inside loops should be "hoisted" out.
- C++ classes wrapping simple data types (e.g., overuse of "smart pointers" or "smart integer" classes).
- Overuse of standard string concatenation operations.
- Recursion is almost always a slug.

More Slug Repellent

There's plenty of other optimizations in the earlier chapters. Well, actually all of the book! Nevertheless, here's a list of some more C++ code optimization techniques for you to consider. Some of the bigger ideas:

- Use "move constructors" instead of copy constructors where appropriate (since C++11).
- Use `static` data members where appropriate, so they are initialized once only.
- Use `std::sort` rather than `qsort`.
- Don't put `try`..`catch` inside an inner loop that's a bottleneck.
- Use `std::bitset` for bit sets or bit vectors.
- Use the "iterators" design pattern rather than returning a full scan of a data structure all at once (saves memory and allows early exit).

- Consider basic C++ arrays instead of `std::vector` if it has a fixed size (known at compile-time) or its maximum size is small enough.
- Consider C++20 coroutines where appropriate for the architecture.
- Structure of arrays (SoA) data layout is more vectorizable than Array of Structures (AoS).

And some of the smaller optimizations:

- Commonly used object or struct fields should be first. On some platforms, smaller offsets from the start of an object are accessed faster. Also, the very first field has offset zero, which is optimized away, so put the most used field first.
- Avoid long `else-if` sequences. You are effectively doing linear search on the problem space in a long block of `if-else-if` statements. The best alternative is to use a `switch` statement, if the conditions are constants. For non-constant conditions or string comparisons, consider tabularizing the options and/or using heuristics to bifurcate the search space (e.g., start with a `switch` on the first letter of a string).
- Use compact numeric ranges for `switch`. If the case numbers are close together, the compiler will probably use a lookup-table in assembler. If the cases are sparse, it can be forced to do an `if-else-if` equivalent in machine code.
- Correct choice of loop. If the condition is true at the first iteration, use `do-while` loops.
- Instead of range checking a signed integer with "`i>=0 && i < MAX`" use a typecast with "`(unsigned)i<MAX`" because negatives become large unsigned positives, and a cast from `int` to `unsigned int` isn't a real instruction at run-time.
- Enable the FTZ ("flush-to-zero") and/or DAZ ("denormals-are-zero") floating-point modes on your CPU, even though they violate the IEEE 754 standard. You probably don't care about tiny floating-point numbers in your weight or probability calculations.
- Enable GCC's floating-point arithmetic speedup options: `-ffast-math`, `-fno-math-errno`, `-fno-trapping-math`, and `-ffinite-math-only`.
- `bsearch` is slow. Choose a better method.
- Use `static_assert` rather than `assert` (e.g., to check data type sizes).
- Copy arrays by wrapping them in a dummy `struct` and using C++ `struct` bitwise assignment. It might be faster than `memcpy`.
- Use `memcpy` rather than `memmove` if you're sure the arguments won't overlap.

- Move local non-`static` objects outside of a critical loop. Reuse the same object rather than re-running constructors and destructors every loop iteration. Add a "`reset`" member function if needed.
- Use scaling factors that are a power-of-two, so that multiplication or division can be a bitshift.
- Specialize a function with a `void` and non-`void` version if you find yourself ignoring the return value sometimes. This avoids all of the calculations to determine the return value inside the `void` function, because the function itself cannot tell whether or not the caller will use its return value.
- Prefer pre-increment (`++i`) to post-increment (`i++`) for non-scalar values. And it's better to use pre-increment even for "`int`" types, even though it's the same, just to get into the habit.
- Use the GCC `__builtin_unreachable()` statement and the "`noreturn`" function attribute to help the GCC optimizer identify dead code paths, allowing unreachable code removal (not that we care that much) and also better optimization of path-specific optimizations on other live paths (e.g., compile-time constant propagation).
- Test the first character of two strings directly with character tests before calling `strcmp`.
- Replace calls to "`round`", "`floor`" or "`ceil`" functions with a type cast to `int` (as an approximation).
- Consider using the simpler `putchar`, `putc`, `fputc`, `puts`, or `fputs` functions rather than `printf` or `fprintf`.
- Write your own versions of `abs` and `fabs`/`fabsf` (but benchmark it).
- Avoid the floating-point `pow` function for computing integer powers.
- Instead of `strlen("literal")`, declare an initialized `char[]` array and use `sizeof(arr)-1`.
- Merge a large number of function parameters into an object. Don't pass 10 Boolean flags as differently named function parameters. Create an object or structure and make them fields instead.
- Avoid calling `strlen` in a "`for`" loop test. Compute `strlen` before the loop, or test for the null byte.
- Merge multiple Boolean function parameters into a bit set. packed into an `int` or `long`. The gain from passing fewer values as function arguments will be offset by the cost of packing and unpacking bits, but still should be better.
- Use `int` type mostly, not `char` or `short`. Maybe prefer `int` to `size_t`, too.
- Specialize functions being called with a constant for an argument using a template function with an integer field. This will increase code size, but the constant will be propagated more at compile-time, and you also don't have the cost of passing it as an argument.

- Add “`noexcept`” specifiers to functions wherever it applies, because this allows the compiler to know not to worry about adding any extra exception handling code.
- If you’re “searching” an array or set of constant integers, known at compile-time, consider “proceduralization” by putting the numbers as cases in a `switch`. (Trust the compiler engineers.)
- Consider writing your own faster `atoi/itoa` functions, as the standard libraries need to handle lots of rare cases, making them slower. (I’m not sure I agree and you might want to benchmark.)
- Don’t overuse “`alignas`” to specify address alignments if you don’t need them, as the enforcement of alignment requirements can impose runtime cost.
- `sprintf` is a slow and unsafe function. `snprintf` is safer but still slow. Find another way.
- Post-increment can be faster in pointer arithmetic, so prefer using the normal idiom “`*ptr++`” rather than “`*++ptr`” to scan a vector.

References

1. Agner Fog, 2023, *Optimizing software in C++: An optimization guide for Windows, Linux, and Mac platforms*, PDF: https://www.agner.org/optimize/optimizing_cpp.pdf
2. Kurt Guntheroth, 2016, *Optimized C++: Proven Techniques for Heightened Performance*, O’Reilly Media, https://www.amazon.com/dp/1491922060
3. Dov Bulka and David Mayhew, 1999, *Efficient C++: Performance Programming Techniques*, https://www.amazon.com//dp/0201379503
4. Fedor G. Pikus, 2021, *The Art of Writing Efficient Programs: An advanced programmer’s guide to efficient hardware utilization and compiler optimizations using C++ examples*, Packt Publishing, https://www.amazon.com/dp/1800208111
5. ISO/IEC, Feb 15, 2006, *Technical Report on C++ Performance*, ISO/IEC TR 18015:2006(E), https://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf (Design of the C++ language from an efficiency perspective, including discussion of virtual functions and other language features.)
6. Nicolai M. Josuttis, 2012, *The C++ Standard Library: A Tutorial and Reference*, Second Edition, Supplementary Chapter, https://www.amazon.com/Standard-Library-Tutorial-Reference-2nd/dp/0321623215, PDF (extra chapter): http://www.cppstdlib.com/cppstdlib_supplementary.pdf (C++ optimizations such as bit sets and user-defined memory allocators.)

7. Bjarne Stroustrup, 2013, *The Essence of C++ with examples in C++84, C++98, C++11, and C++14*, PDF Slides: http://www.staroceans.org/e-book/essenceOfC++.pdf
8. Wikibooks, 2023, *Optimizing C++/Writing efficient code/Performance improving features*,
Wikibooks, https://en.wikibooks.org/wiki/Optimizing_C%2B%2B/Writing_efficient_code/Performance_improving_features
9. Dave Abrahams et. al., 2003, *Technical Report on C++ Performance*, http://web.archive.org/web/20040608203404/http://www.research.att.com/~bs/performanceTR.pdf
10. Jakob Engblom, 2001, *Getting the Least Out of Your C Compiler*, https://www.engbloms.se/publications/engblom-esc-sf-2001.pdf
11. Jon Louis Bentley, 1982, *Writing Efficient Programs*, Prentice Hall.
12. Thomas Plum and Jim Brodie, 1985, *Efficient C*, Plum Hall Inc.
13. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, 1986, *Compilers—Principles, Techniques and Tools*, Addison-Wesley.
14. Donald E. Knuth, 1973, *The Art of Computer Programming (Vol. 3): Sorting and Searching*, Addison-Wesley.
15. James O. Coplien, 1992, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley.
16. Jonathan S. Shapiro, 1991, *A C++ Toolkit*, Prentice Hall.
17. Bjarne Stroustrup, 1991, *The C++ Programming Language (2nd edition)*, Addison-Wesley.
18. David Spuler, March 1994, *Generative AI in C++: Coding Transformers and LLMs*, https://www.amazon.com/Generative-AI-Coding-Transformers-LLMs/dp/B0D14LHGZ6/