CUDA C++ Debugging

Safer GPU Kernel Programming

by David Spuler

Aussie AI Labs

CUDA C++ Debugging: Safer GPU Kernel Programming

by David Spuler

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

https://www.aussieai.com

First published: October 31, 2024.

Preface

Why a Book on Debugging 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 weeds when something goes wrong. This book examines a variety of techniques for debugging CUDA C++ programs, from beginner to advanced, along with a catalog of common CUDA C++ errors to avoid, and preventive methods for writing more resilient AI applications.

Who This Book is For

Anyone programming in CUDA C++ or trying to learn the language will benefit from better debugging! This book examines bugs in coding from beginner to advanced, starting with basic debugging techniques for simple errors. In the later chapters, the book then covers a variety of advanced techniques for improving the quality and resilience of production-quality CUDA C++ 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.

Anyway, if I were you, I'd skip to the back of the book to the "Puzzles" appendix. It's the only part of the book that's fun, and you can still bill hours for reading it. Feel free to send ideas for more puzzles, or interesting bugs that you come across.

About Aussie AI

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

Our AI Research

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

- Fast on-device model inference algorithms 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 area: https://www.aussieai.com/cuda/overview.

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 chapters 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 Debugging CUDA C++

A whole book on debugging CUDA C++ isn't enough for you? You can find more on our CUDA website https://www.aussieai.com/cuda/overview.

CUDA C++ Projects. Learn more about our CUDA C++ projects at https://www.aussieai.com/cuda/projects:

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

Updates and Additions: Additional book materials, updates and errata will be made available over time online at the Aussie AI website. Visit this URL: https://www.aussieai.com/cuda/debug.

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 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

Although I hope the information is useful to you, neither the content nor code in this work is guaranteed for any particular purpose. Nothing herein is intended to be personal, medical, financial or legal advice. You should make your own enquiries to confirm the appropriateness to your situation of any information. Many code examples are simplistic and have been included for explanatory or educational benefit, and are therefore lacking in terms of correctness, quality, functionality, or reliability. For example, some of the examples are not good at handling the special floating-point values such as negative zero, `NaN`, or `Inf`.

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

Third-Party License Notices

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. 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 extends to monitoring/observability, devops/MLOps, ITSM, low-resource LLM inference, Retrieval Augmented Generation (RAG) and AI-based agents.

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

Cameron Gregory is a technology entrepreneur including as co-founder of fintech bond trading startup BQuotes (acquired by Moody's), previously 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. CUDA Introduction

What is CUDA?

CUDA is officially an acronym for Common Unified Device Architecture, which is just so excruciatingly boring. Instead, I prefer to think of CUDA as a barracuda, sleek and fast, shredding GPU chips with its gnashing teeth by sending them too much work.

CUDA is a platform to program NVIDIA GPUs, consisting of many tools and libraries. As a C++ programmer, I'm going to focus on CUDA C++ capabilities, but there is support for other programming languages, such as Fortran and Python.

CUDA is owned and maintained by NVIDIA, and used to write code for NVIDIA GPUs. The basic idea is to write C++ for both the CPU and the GPU, and the CUDA C++ compiler allows you to do both from the same source file. The CUDA platform is not open-sourced, but is priced free to use (although the chips are not!).

Why Use CUDA?

Oh, come on! You know the answer to this one: AI. NVIDIA GPUs are for AI, and CUDA is NVIDIA's answers to how you program a GPU. As for the C++ part, well, CUDA and C++ go together like AI and cat videos.

This book mostly assumes that you're using CUDA for generative AI, but there are many use cases. They're not as well-known as AI, but they're making a lot more money for companies than AI ever will.

Generally, any type of algorithm that needs to do a lot of number crunching can be parallelized within an inch of its life with CUDA.

Use cases for CUDA include:

- Generative AI training and inference
- Physics computations
- Drug discovery algorithms
- Cryptography (Bitcoin mining!)
- Linear algebra operations
- Optimization and search space computations

There are many more types of parallelizable algorithms. Feel free to add your own.

Main Features of CUDA

The CUDA environment is not just a C++ platform, but also an entire ecosystem. This includes:

- Documentation ranging from introductory to reference manuals.
- Articles and technical blogs on the NVIDIA website.
- Multiple language support (e.g., C++, Fortran)
- Example code of various ilks on NVIDIA's Github repo.
- Full implementations of AI backends.
- Forums and support platforms for questions.
- The annual GTC NVIDIA conference in California.

Generally, I have to say that everything I see on the NVIDIA website is underpinned by a high level of technical competence. NVIDIA is an impressive company.

Oh, yeah, I almost forgot. There is also some C++ stuff in CUDA:

- CUDA C++ compilers
- Debugging tools
- Memory checkers
- Synchronization checkers
- Performance profiler tools

Many of the tools have both graphical and command-line interfaces. Personally, I'm old-school and prefer the CLI versions, but many programmers prefer a nice GUI for increased productivity.

Advantages of CUDA C++

CUDA is popular and the market leading interface for programming NVIDIA GPUs. In fact, CUDA is regarded as a second level of value that helps NVIDIA stay on top in the GPU arms race. Here are some thoughts on why.

C++ Syntax. Writing a CUDA program is just like writing a C++ program, and everybody loves doing that. Personally, I've been doing that for 30 years, so we can just stop here at this point.

Fast! The programs that you write in CUDA run fast. It's probably not really about the software at this point, but us programmers like to think it is. Let's give CUDA the credit!

Dual coding model. You write both the CPU code and the GPU code in the same C++ program. This is very convenient, and keeps everything somewhat more orderly than otherwise.

Capable. You can do all of the basic C++ stuff, such as all the arithmetic and logical operators. There are also CUDA APIs for just about anything you could think of, such as memory management and thread synchronization.

Libraries. At a higher level than the CUDA APIs, there are also CUDA libraries for a lot of the common coding tasks, such as vector and matrix computations. These have been coded by professionals, and it's hard to write faster code than you'll find in these libraries, although many people keep trying.

Overall Limitations of CUDA

CUDA has been very successful and is often mentioned as a massive competitive "moat" for NVIDIA. Nevertheless, there are areas where CUDA can be problematic.

Steep Learning Curve. CUDA is not the easiest language to master. For starters, you need to know C++, and then there are a number of CUDA-specific syntax constructs and a lot of CUDA APIs and libraries to learn.

Even worse, the need to learn these new sets of C++ keywords and libraries is compounded by the utterly brain-bending nature of SIMD parallel programming. Hence, not all of the difficulty can be blamed on CUDA itself.

GPU specific. The use of CUDA C++ is limited to NVIDIA GPUs. Although some attempts have been made to layer CUDA over other non-NVIDIA GPUs, this does not work well, and is not supported by NVIDIA.

Proprietary License. The CUDA platform is not open-sourced by NVIDIA, although it is free. This compares with other similar GPU platforms, such as AMD's ROCm, which has an open-source license.

However, much of the NVIDIA code examples are open-sourced, under various licenses, so this limitation only applies to the core compiler and runtime platform.

Non-SIMD algorithms. Massive SIMD parallelism in GPUs only works to optimize certain kinds of algorithms. This is not a limitation of the CUDA platform itself, but moreso of GPUs in general.

Fortunately, there are plenty of demand for recurring vector calculations in optimizing applications such as AI inference and training, not to mention Bitcoin mining and video codec processing.

Other GPU Accelerator Platforms

CUDA is not the only way to program NVIDIA GPU chips, but it's the best. CUDA is widely regarded as a superior platform that helps sustain NVIDIA's dominance of GPU chips for AI applications, with a "software moat" that adds another layer to its capabilities.

However, there is plenty of ongoing activity from competitors and also in open source communities. Some of the other upcoming GPU software platforms are discussed below.

Triton (OpenAI). The Triton platform was initially created by OpenAI, and has been open-sourced as its own project. The goal of Triton is to make GPU programming simpler, so as to write GPU applications in a Python-like language. The idea is to hide a lot of the low-level issues, such as memory transfers, in a way that does not impact performance.

ROCm (AMD). The ROCm software platform is for AMD GPU programming. Unlike CUDA, the underlying code for ROCm has been open-sourced, and is available for review. This is a fully-capable platform and has a long history of development.

Intel OneAPI. The OneAPI platform was created by Intel, and initially focused on their GPU chips. It has since become an open standard and its own project, allowing OneAPI to be used to manage other vendors' GPU hardware.

Apple hardware. Apple makes its own M-series chips, based on the Arm architecture, for its PCs, tablets, and phones. To support developers of applications for these devices, Apple has developed its own software acceleration platforms, including CoreML, Apple Accelerate, Apple Metal, and the new "Apple Intelligence" platform. Apple's hardware chips are not as fully-capable as high-end GPUs, and the focus of these software platforms is more for execution on AI PCs (MacOS) and AI phones (iPhone/iOS).

Vulkan. The Vulkan API is a portable layer to operate across multiple types of GPUs. A lot of its historical functionality is related to gaming and similar GPU applications, but it has become focused more on AI lately. Vulkan is an open source project that is supported by various corporate entities in this space.

SYCL (pronounced "sickle"). The SYCL platform is also an open-source multi-GPU abstraction layer and standardization, backed by the Kronos Group. It allows the development of GPU-based applications at a higher-level, allowing deployment to different hardware stacks.

I'm not sure why you needed to know about all those platforms, because you really only need one: CUDA. And most of the CUDA backends were written in C once up on a time, but are now usually written in C++, so the best way to program GPUs is CUDA C++.

2. Debugging Hello World

Buggy First CUDA C++ Program

Amusingly, a typical "Hello World" program written in CUDA C++ will have a bug. How's that for a nice introduction to the parallel programming world?

Talk about a steep learning curve!

This chapter looks at a "hello world" program written in CUDA C++, that just tries to print out a message. Such a humble goal, and yet it fails, of course, and then the remainder of the chapter is trying to debug the code.

If you're a beginner at CUDA C++, you'll need to install the CUDA Toolkit on a computer with a GPU. If you don't have a GPU, you can use Google Colab without a GPU (mostly for free), as discussed later in the chapter. And if you're already an advanced CUDA programmer, well, you'll already have a GPU environment, but why are you reading this chapter?

Let's have a go at a basic program:

```
// Hello World, basic CPU version
#include <iostream>

int main()
{
    printf("Hello CUDA!\n");
}
```

Yes, that runs just fine and the output is:

```
Hello CUDA!
```

There's only one problem with this code: it's not running on the GPU. You can't call yourself a CUDA programmer if you run code on a CPU. All of the advanced CUDA programmers rip out the CPU from their computers, and run with just a GPU and a bunch of ping pong balls instead.

Real programming. The basic idea with GPU programming in CUDA C++ is:

- Both CPU "host" code and GPU "device" code in the same C++ file.
- The default is that C++ code is for the CPU host.
- We mark GPU device code with the "`__global__`" specifier (yes, it has four underscores).

The way your program runs on a GPU is:

- Execution starts in the CPU at the `main` function.
- The GPU function (called a "kernel") is just sitting there, twiddling thumbs, waiting.
- The CPU "launches" a GPU function.
- The GPU then runs that kernel function.

Hence, to modify our C++ code to run on the GPU, we need to:

- Define a function
- Declare it as "`__global__`"
- Launch it using a weird syntax.

Here's the very first attempt at a program that runs on a GPU:

```
// Hello World, buggy GPU version
#include <iostream>

__global__ void aussie_cuda_hello_world()
{
        printf("Aussie CUDA says Hello World!\n");
}

int main()
{
    aussie_cuda_hello_world <<< 1, 1 >>> ();
}
```

All this does is say "hello" without any other computations. The way that you know it's running on the GPU is the extra specifier "`__global__`" on the function definition. Not only is this function rather dull, it's also sluggy, because we really don't want the GPU to be calling `printf`, except for debug tracing.

Where's the CPU code? Any function that does not have a CUDA specifier like `__global__` or `__device__` is by default a CPU function. It's called "host" code in the CUDA vernacular, since CPU is the "host," and GPU is the "device." GPU code is called "device code" or a "kernel." For our hello world example, the CPU code is just the `main` function.

The strangest CUDA syntax is pretty obvious with the "triple chevron" tokens "<<<" and ">>>" surrounding some numbers. This code is a "launch" or "invocation" of the GPU "kernel." It's easier to visualize the function call without the fancy CUDA syntax, which is basically a zero-parameter standard C++ function call:

```
aussie_cuda_hello_world();
```

However, it has some extra parameters inserted between the function name and the function arguments:

```
<<< 1, 1 >>>
```

The meaning of the numbers is clearer if we do this:

```
<<< blocks, threads_per_block >>>
```

The modified kernel invocation would look like this:

```
int blocks = 1;
int threads_per_block = 1;
aussie_cuda_hello_world<<< blocks, threads_per_block >>>();
```

This syntax launches multiple copies of the CUDA C++ GPU device kernel function `aussie_cuda_hello_world`, each of which is called a "thread." How many? In this case, we are launching 1 block of 1 threads-per-block each, so there is a grand total of 1*1=1 function calls to our kernel, which is not exactly "multiple copies" of the kernel, as I vaguely promised above.

So, anyway, here's our full CUDA "hello world" program. Let's run it so we can bask in glory. Here's the output:

```
(sound of crickets)
```

What?!? There's no output! How can there be a bug when there's literally only two statements?

Fixing Hello World

Okay, here's the problem, in simple terms: the CPU didn't wait for the GPU's output. The whole program finished on the CPU before the GPU output anything.

The solution is simple: *make the CPU wait*. The simplest way to do this is to call `cudaDeviceSynchronize`:

```
aussie_cuda_hello_world<<< 1, 1 >>> ();
cudaDeviceSynchronize();
```

This forces the CPU to wait for all the GPU kernel threads to finish, which is called "synchronization." Hence, `cudaDeviceSynchronize` is a "blocking" or "synchronous" type of CUDA call.

One nice feature of `nvcc` compiler is that you don't need any `#include` of a the CUDA C++ runtime header file to call `cudaDeviceSynchronize`. This is because `nvcc` automatically includes "`cuda_runtime.h`" at the top of the CUDA C++ file.

More Details on GPU Output Buffering

This section is optional and quite advanced, but if you really want to know (and maybe you don't), here's a deeper look at why the output disappeared. In more detail, there's actually three problems:

1. CUDA kernel launches with "<<<...>>>" are asynchronous.
2. Kernel output is buffered, rather than immediately output.
3. Buffered GPU kernel output is discarded on CPU host program exit.

That's rather a mouthful. Let's try to break it down:

- The CPU didn't hang around for the GPU to do anything, because it doesn't wait for GPU kernels to finish.
- But the C++ code in `main` had no further statements, so the whole program immediately exited (the CPU part).
- The GPU still did its work, and called `printf` correctly inside the GPU code,
- The weird part is that `printf` inside the GPU is not actually printed out immediately by the GPU. Instead, it's stored ("buffered") for the CPU to print out later.

- Buffered GPU output doesn't get printed until the CPU runs again afterwards.
- But the CPU had already exited, so the CPU wasn't still there anymore to print out any of the GPU output.
- So, the GPU just gave up and threw it all away instead, and then the GPU quit too.

Any clearer? Maybe made it worse? I told you this section was optional for a reason.

Running Multiple Threads

Every call to the function in each block starts in a new thread at the same time, and runs in lock-step over the same set of statements. All of parallel calls to the kernel function have the same function parameters (i.e., none in this case).

This is also a rather dull kernel invocation, because it only runs 1 single instance of the GPU kernel. That's not parallel! It's only running 1 copy of the kernel at a time, which is something that a real CUDA programmer would never, ever do.

Let's run 5 threads, so we have 5 versions of the kernel running instead. Here's our updated code, including the bug fix call to `cudaDeviceSynchronize`:

```
// Hello World, 5 threads version
#include <iostream>

__global__ void aussie_cuda_hello_world()
{
    printf("Aussie CUDA says Hello World!\n");
}

int main()
{
    int blocks = 1;
    int threads_per_block = 5;
    aussie_cuda_hello_world<<<blocks,threads_per_block>>>();
    cudaDeviceSynchronize();
}
```

Here's the output:

```
Aussie CUDA says Hello World!
Aussie CUDA says Hello World!
Aussie CUDA says Hello World!
Aussie CUDA says Hello World!
Aussie CUDA says Hello World!
```

This runs 5 threads, because we have launched 1 block, and each block has 5 threads. Now let's modify it so that it tells you which threads are running. We can do this with a statement:

```
int tid = threadIdx.x;
```

Our new kernel is this:

```
__global__ void aussie_cuda_hello_world()
{
    int tid = threadIdx.x;
    printf("GPU thread %d says Hello World!\n", tid);
}
```

Here's the output:

```
GPU thread 0 says Hello World!
GPU thread 1 says Hello World!
GPU thread 2 says Hello World!
GPU thread 3 says Hello World!
GPU thread 4 says Hello World!
```

As you can see, the 5 threads are numbered 0..4 for their "thread index" value. Also, don't be misled by the fact that they appeared in sequential order from 0..4, because that's an idiosyncrasy of the `printf` handling in GPU kernel code. All five threads are actually running in parallel!

Running Multiple Blocks

We've only had a single "block" of threads so far. Let's try running two blocks by changing our CPU code:

```
int blocks = 2;  // Hooray!
int threads_per_block = 5;
aussie_cuda_hello_world<<< blocks, threads_per_block >>>();
cudaDeviceSynchronize();
```

And we can also make each thread figure out what block it's in using the `blockIdx` "block index" variable.

Here's our updated GPU kernel code:

```
__global__ void aussie_cuda_hello_world()
{
    int tid = threadIdx.x;
    int bid = blockIdx.x;
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    printf("GPU block %d thread %d says Hello World!\n",
          bid, tid);
}
```

The output is:

```
GPU block 1 thread 0 says Hello World!
GPU block 1 thread 1 says Hello World!
GPU block 1 thread 2 says Hello World!
GPU block 1 thread 3 says Hello World!
GPU block 1 thread 4 says Hello World!
GPU block 0 thread 0 says Hello World!
GPU block 0 thread 1 says Hello World!
GPU block 0 thread 2 says Hello World!
GPU block 0 thread 3 says Hello World!
GPU block 0 thread 4 says Hello World!
```

There were two blocks of five threads each, so 10 threads ran in total, and they all ran in parallel.

Here we can see that the two blocks had a "block index" of 0 and 1, and they printed in reverse order. Also note that the thread index was always 0..4 in both blocks (i.e., not 0..4 and 5..9).

Finally, let's show how to get two blocks of five threads to properly count to 10. The way to work out the "index" of a thread in the whole "grid" (multiple blocks), is to use this CUDA code, which is the most common statement you'll see every day in CUDA C++:

```
int id = blockIdx.x * blockDim.x + threadIdx.x;
```

Note that "`blockDim.x`" means "block dimension" and is a builtin variable that is the "threads-per-block" value, so it will equal 5 here.

Hence, this is the new GPU kernel C++ function:

```
__global__ void aussie_cuda_hello_world()
{
    int tid = threadIdx.x;
    int bid = blockIdx.x;
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    printf("GPU index %d block %d thread %d says Hello!\n",
        id, bid, tid);
}
```

Here's the output:

```
GPU index 5 block 1 thread 0 says Hello!
GPU index 6 block 1 thread 1 says Hello!
GPU index 7 block 1 thread 2 says Hello!
GPU index 8 block 1 thread 3 says Hello!
GPU index 9 block 1 thread 4 says Hello!
GPU index 0 block 0 thread 0 says Hello!
GPU index 1 block 0 thread 1 says Hello!
GPU index 2 block 0 thread 2 says Hello!
GPU index 3 block 0 thread 3 says Hello!
GPU index 4 block 0 thread 4 says Hello!
```

Okay, so it worked! Even though it looks messy, our 10 kernels counted from 0..9 in parallel.

Why Are Blocks Needed?

Why do you need blocks to run CUDA kernel functions on the GPU at all? I mean, they complicate the index calculations.

Why can't you just always use 1 block, and then specify as many threads as you want? Then every thread could just get its thread index number from the `threadIdx` builtin kernel variable. It would work properly without that weird calculation involving, `blockIdx` and `blockDim`.

Here's the idea for using single blocks:

```
int blocks = 1;
int threads = 16384;  // Too many!
aussie_cuda_hello_world<<< blocks, threads >>>();
```

Over 1024 threads is not allowed! Here's a more CUDA-style use of multiple blocks with thread sizes typically 256 or 512 threads-per-block.

```
int blocks = 64;
int threads = 256;  // 64x256=16384
aussie_cuda_hello_world<<< blocks, threads >>>();
```

Why do we do need to multiple blocks? Wouldn't all the threads run in parallel either way?

Short answer: no, the GPU does not actually run all threads in parallel. It depends.

Longer answer: The answers about blocks being needed are mainly relevant to more advanced CUDA C++ programming, but here are some:

- The GPU has hard limits on the block size (i.e., 1024).
- Shared memory with the "`__shared__`" specifier has block scope.
- Scheduling on the GPU is at a block level.

Okay, so the GPU has a fixed limit on threads-per-block. But even if we could use an unlimited number of threads in each block, we shouldn't do so.

But why?

Shared memory is an important optimization discussed later. This faster memory has block scope, so we need to control the block size so as to maximize this benefit from speedy shared memory.

Scheduling is also a low-level GPU issue whereby the Streaming Multiprocessors (SMs) only work on complete blocks. And there is an underlying scheduling issue.

The hardware scheduler tries to allocate as many blocks in parallel as it can, but sometimes it cannot fit all of them onto a single multiprocessor, because it's got too many other workloads.

This refers to production usage of a GPU, where it's overheating from doing other important computation work (e.g., serving Taylor Swift companion bots), not just the cold, lazy GPU under your desk that's only been playing FortNite.

The scheduler on the GPU does not necessarily run all the blocks in parallel, but has to guarantee that all the threads in a single block do. Hence, a big block will have to wait longer for enough space to be free on the GPU, whereas smaller blocks can get scheduled more easily.

3. CUDA for C++ Programmers

Basics of CUDA C++ Programming

CUDA C++ is similar to C++, but with many extensions and idiosyncrasies. Here are some of the salient differences.

Parallel programming C++ features. These capabilities are the main superpower of CUDA and the reason it exists. The idea is to take SIMD to the extreme, and send the same computations to the GPU in massive groups (e.g., 10,000+ operations in parallel). This requires not just a syntax change, with GPU function "`__global__`" declarations and triple-angle-brackets for "`<<<...>>>`" invocations, but an entirely new way of thinking about how to optimize the algorithms.

Filename suffixes. Most programs in CUDA C++ are written with "`.cu`" as the filename suffix for source code. CUDA header files often use "`.cuh`" but can also simply use "`.h`" or "`.hpp`" rather than a CUDA-specific filename. The intermediate PTX assembly files, created by `nvcc`, have a "`.ptx`" suffix, if you like that low-level kind of programming.

CUDA Development Tools. Some of the development tools include:

- `nvcc` — NVIDIA C++ compiler.
- `cuda-gdb` — CUDA's `gdb`-based debugger.
- `compute-sanitizer` — CUDA's memory-checker (like `valgrind`) and three other error detection tools: `racecheck`, `synccheck`, and `initcheck`.
- `ncu` — Nsight Computer CLI command-line performance profiler.
- `nvprof` — NVIDIA performance profiler (although deprecated).

There are various tools with graphical interface and extensive IDE integration. This offers many ways to be productive in coding CUDA C++.

Differences from Standard C++

By now you've probably noticed that CUDA C++ programming is a lot like C++ programming, but with some extra stuff. The main things are:

- Extra `#include` directives for CUDA header files.
- `__global__` specifier (equivalently, `__device__` means device-only).
- `<<< blocks, threads_per_block >>>` kernel launch syntax.

A lot of standard C++ code can be run on the GPU in the way that you'd expect, such as:

- C++ operators
- Expressions
- `if` statements
- Loops
- `switch` statements
- Types
- Local variables
- Assignments
- `printf` output

I don't know what you think, but I find this quite weird! When I was learning CUDA, I expected it to be launching special SIMD instructions and intrinsic function calls to do vector operations. But, no, it's more like just normal C++ programming, which makes it much easier to learn. They must have some very smart compiler design engineers at NVIDIA working on the CUDA Toolkit (and obviously a lot of brainy hardware engineers there, too).

Auto-included CUDA header files. An interesting and also rather pleasant improvement to the CUDA C++ programming environment is that CUDA doesn't need many `#include` directives. Your CUDA C++ "hello world" program doesn't need to actually include `<cuda_runtime.h>` or various others, because `nvcc` does it auto-magically for you, if it's processing a "`.cu`" file. You can call CUDA C++ APIs like `cudaMemcpy` and `cudaMalloc` without an explicit header file include.

This is not a hidden Easter egg that AI programmers whisper about, but an officially documented feature. The CUDA C++ programming guide explicitly says that `nvcc` "implicitly includes `cuda_runtime.h`" at the top of the source file.

Really, wouldn't it be nice if every C++ compiler did this? Why do we need this boilerplate at the top of every C++ program, when the compiler could almost always guess which header files we want when it sees the functions we've called?

Unfortunately, even `nvcc` doesn't guess a header file for everything non-CUDA. If you want to call `printf`, you still have to include `<iostream>` or `<stdio.h>`. And various CUDA add-on libraries still need to be explicitly included.

Kernel Limitations. Although the GPU code does look like ordinary C++, there are some important limitations on the device "kernel" functions that run on the GPU.

- No return type — the kernel above has type `void` for a reason.
- No pass-by-reference — don't use `&` parameters for kernels.
- `main` cannot be device code — the program always starts in the CPU host code.

There's quite a lot of other limitations for kernel code on the GPU, but we'll get to them later.

However, the host code is not restricted! There's much fewer C++ limitations on the host code, because it runs on the CPU. In fact, CUDA uses the underlying platform's C++ compiler, such as GCC, so there are a lot more things possible in the CPU code.

CUDA Dual Programming Model

To create a CUDA program, you need parts that run on the CPU, and parts that run on the GPU. You're probably fairly familiar with how to compile code to run on a CPU, and the CUDA C++ program is very similar.

But how do you write code for the GPU?

The answer couldn't be simpler: you just write C++ in the same file. CUDA has a "dual" programming mode in its C++ files (by which I mean its "`.cu`" files). The two parts are:

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

Host code versus device code. Your CUDA C++ code specifies both of these two distinct types of code. Host code runs on the CPU of the computer that "hosts" the GPU (or GPUs), and is intended to prepare data for the GPU, process the computed results, and other such high-level tasks. Device code is the optimized low-level code that actually runs on the GPU in a massively parallel SIMD manner, and it's typically called a "kernel."

How does the compiler know which is which? The short answer is:

- CPU host codes — the default meaning of ordinary C++ functions.
- GPU device code — extra `__global__` keyword (with double underscores)

The only other trick is when the CPU code launches a kernel on the GPU code. It's like a function call, but it's called a "launch" and it uses a special triple-chevron syntax. A simple example of a "CPU-to-GPU" execution launch of a GPU device function would look like this:

```
my_gpu_kernel<<<1,1>>>(parameters);
```

The two numbers inside the angled brackets are the number of blocks to launch, and the threads-per-block. Note that advanced calls can have four parameters.

In more detail, a normal function without extra keywords is for the CPU, whereas device code for the GPU has a new CUDA-specific keyword, the "`__global__`" specifier, on the C++ function declaration. One keyword is all that's needed to tell the compiler to run a function on the GPU rather than the CPU.

There are also other specifiers: "`__device__`" and "`__host__`." They can be useful to indicate functions that run on either the GPU device or the CPU host, and can be combined if both are true (e.g., a low-level utility function). However, note that "`__device__`" GPU functions cannot be launched or called from host code, so only "`__global__`" is used for kernel entry points. The meaning of "global" is effectively that control flow can cross over from the CPU host to the device GPU kernel function.

CUDA's mixing of host code and device code together is sometimes called "heterogenous computing." A single CUDA C++ source file can contain code for both the host CPU and the GPU device. The host code is more like the usual type of non-GPU programming and uses mostly the standard C++ features. The device code typically runs across multiple GPU "threads" in parallel, uses a combination of basic C++ syntax with various SIMD builtin extensions, and you have to think in "vectorized logic" to write these device functions.

CUDA Programming Control Flow Model

When I was first learning CUDA, I thought it would be very focused on SIMD operations. What I mean is that adding two vectors would be done by uploading the data for both vectors, and then sending an opcode for "add" so that the GPU would do a parallel SIMD addition on the vectors. In other words, I thought it would be somewhat "declarative" or like assembly language or similar to AVX SIMD instructions on x86 chips.

Not at all!

The CUDA programming model is almost like a full CPU-based computing environment, multiplied by a thousand, on the GPU. It works like a thousand mini-programs running in multiple threads on the GPU. The top-level features include:

- Instructions (i.e., GPU-specific machine-code)
- Data
- Storage (for computed results)

You write an entire C++ function for each kernel computation, and then this function gets run in parallel across lots of "threads" on the GPU. In particular, each CUDA thread has its own versions of:

- Program counter (instruction pointer)
- Function call stack
- Variables (on the "stack" or in "registers" underneath)
- Local memory (e.g., for local variables)

You write your kernel in almost full C++ capabilities. For example, some of the basic stuff you can use includes:

- Sequences of statements
- If statements
- Loops
- Arithmetic expressions
- Parameter passing
- Variables

There are some limitations, however. For example, you can't use recursion, or overloaded `operator` functions, and template usage is somewhat restricted.

Turing completeness. If you like your obscure Computer Science theory (and who can honestly say they don't), you can see that this covers all of the three key control flow capabilities:

- Sequence
- Selection
- Iteration

And the fourth requiring of data storage is also covered by variables and various layers of memory. This means that GPU threads are "Turing complete" computation models. Hence, the GPU runs your CUDA kernel code almost like a thousand tiny fully-complete computers, all running the same code in parallel. Declarative that!

GPU Program Flow

A typical CUDA program has a conceptual sequence something like this:

- Initialization
- Copy data from CPU memory to the GPU
- Launch the GPU kernel (execute it)
- Copy the results back from GPU memory to the CPU
- Cleanup

Let's analyze these steps in more detail.

Initialization. The program initialization in the host code may have all the usual program initialization, but it also usually has one more step: allocating memory on the GPU. This is often done via the `cudaMalloc` function with the direction parameter set to `cudaMemcpyHostToDevice`. Addresses from `malloc` refer to host memory and are called "host pointers." Similarly, the return from `cudaMalloc` points into GPU device memory and is called a "device pointer."

Copy to GPU memory. Copying data between the host memory and the GPU memory uses the `cudaMemcpy` function. This runs in the host code, but affects the memory on the device. The "unified memory model" that is handled by CUDA means that an address for the GPU memory can be managed in the host code. The host code can allocate and free memory on the GPU.

Kernel launch. The host code that does the launching of the kernel uses the <<<...>>> triple-angle-bracket syntax. This starts the code running in the GPU. The kernel code itself is also defined in the CUDA C++ program, as a function with a "`__global__`" specifier.

Copying data from GPU memory. Copying the result data back from the GPU uses the `cudaMemcpy` function again, but with a twist. This function has an extra parameter that specifies whether to perform a host-to-GPU or GPU-to-host memory copy operation. Hence, the reverse copy just uses the other parameter, via `cudaMemcpyDeviceToHost` rather than `cudaMemcpyHostToDevice`.

Cleanup. The final program cleanup code is all of the standard program-ending logic. One final step may be to call `free` for host pointers, and `cudaFree` to release any device pointers with addresses of GPU memory objects, thereby avoiding memory leaks in either host or device memory.

CUDA Parallel Execution Model

CUDA has various layers of parallelism, some of which map to hardware components in NVIDIA GPUs, and some are more of a software abstraction. These are relevant to the GPU portion of the C++ code, i.e., the device code. This model specifies how many parallel invocations of the device code get launched for your kernel.

Threads. A thread is the lowest level of compute execution. CUDA threads are more of a software abstraction than a direct mapping to hardware.

Blocks. Multiple threads are organized into "blocks" of combined execution. Each block has a fixed number of threads.

Grids. The "grid" is the total span of all the blocks, which contain all the threads. Since threads-per-block is a fixed number (for each invocation, not for everyone), the structure of all the blocks is somewhat "rectangular" in shape.

Streaming Multiprocessors (SMs). The streaming multiprocessors, sometimes just called "multiprocessors" or "SMs," are a top-level execution unit on a GPU. There are not many of them, and execution of grids (i.e., multiple blocks of threads) is allocated onto parts of a SM, or sometimes across multiple SMs on the same CPU.

These are the four main conceptual structures: threads in blocks in a grid in a "multiprocessor" (i.e., SM). However, there are some other terms used.

Warps. A warp is a group of threads, usually 32 threads on NVIDIA chips. Blocks are actually organized into warps, each of 32 threads, so warps are a structure that sits awkwardly between threads and blocks in size.

Clusters. NVIDIA's H100 chip introduces a fourth major category: thread clusters. This allows some particular programming of threads that can span different blocks.

Why do we care about all this hardware stuff? In some sense, we don't care that much about these abstractions of the GPU hardware layers when programming CUDA, since our C++ only does a small amount of logic related to them. A lot of the issues of scheduling execution across different threads, blocks, and cores are hidden from us by the CUDA C++ compiler. However, there are some reasons to pay attention.

Thread computations. The main aspect of CUDA coding is to write the C++ function for each thread (i.e., each invocation of a "kernel" in a separate thread), and we only care about blocks, grids, and SMs because we want to be sure that enough threads are launched to perform all of our computations in parallel. Hence, every CUDA kernel launch involves some arithmetic computations about blocks, warps, and threads. We need enough for full parallelism!

Sharing data. Another reason arises when transferring data between different parts of our CUDA code. There are various different levels of memory and caches in a GPU. Some of these memory structures are limited to within a warp, within a block or within an SM. Hence, if we want our algorithm to share intermediate results across different threads running different parts of the kernel, and we want to use the fastest type of memory to achieve this, then we have to pay attention to which threads can access which data from which other threads, via shared memory and memory caches.

Features of CUDA C++ Programming

Non-blocking asynchronous kernel calling. When the host code calls a GPU kernel (e.g., a function declared as `__global__`), the invocation via the <<<...>>> syntax does not block and wait. It runs asynchronously, launching the GPU kernel, but continuing the execution of the host code immediately after the call. Hence, it will return before the results are available from the GPU kernel.

Maybe you want to wait until the results are available from the GPU? One way to make the host code block to await the completion of a kernel is the `cudaDeviceSynchronize` API, which blocks the host code on the CPU

until all prior threads have completed. This is a useful safety catch, but can also be a performance slug if you're needlessly waiting.

Unified memory model. CUDA allows programmers to use a "unified memory model" whereby the same block of memory is available to both host and device code. The same memory address space is abstracted so that both the CPU code and the GPU kernels can access the same memory. This simplifies some aspects in sharing data between the main program and the GPU acceleration kernels. The same memory can even be shared across multiple GPUs, but that's jumping ahead a little bit.

Device memory management. The GPU memory can be managed via the host code using builtin functions. The main builtin functions for managing GPU device memory are:

- `cudaMalloc` — allocated GPU memory (equivalent to `malloc`).
- `cudaFree` — de-allocate GPU memory (equivalent to `free`).
- `cudaMempcy` — copy bytes in device memory (i.e., GPU `memcpy`).
- `cudaMemset` — set all bytes to the same value (i.e., GPU `memset`).

There isn't a `cudaCalloc` function to zero the memory, but you can combine `cudaMalloc` with `cudaMemset` to create your own.

Memory transfer costs. An important point in using CUDA code for AI engines is that various Transformer inference algorithms are memory-bound, rather than compute-bound. Generally speaking, for inference tasks, the initial "prefill" phase (or "prompt processing") before the first token is emitted is compute-bound (i.e., a very busy GPU), whereas the subsequent decoding phase of token-by-token generation (i.e., "autoregressive decoding") is memory-bound.

Hence, the cost of transferring data between the different memory cache levels, or sending data up to the GPU, or pulling the results down from the GPU to the CPU, can be a bottleneck. Although the unified memory model is very convenient, it hides a lot of the data transfers between the CPU and GPU code, which must be optimized for faster AI kernels.

CUDA C++ Syntax

CUDA C++ is an extension of C++ syntax, and many features are the same. The CUDA extensions are many, mostly aimed at parallel programming support. However, CUDA lags in the adoption of some of the advanced standard features, so not everything is available.

Comments. Comments are supported via the "//" single-line and /*...*/ multi-line C++ comment styles. As with C++, the /*..*/ comments do not nest.

Host code versus device code. The syntax is slightly different for the (non-GPU) "host code" versus the GPU-executed "device code" in a CUDA program.

Device code for the GPU is specified via two syntax differences:

(a) "__global__" identifier, which declares a GPU-executable function, and

(b) "<<<...>>>" triple-angle-bracket syntax, which is akin to calling the GPU function (with parameters).

Starting execution on the GPU is conceptually more involved than a function call in a sequential C++ program, but the invocation of a kernel on the GPU is the effect. The "__global__" specifier allows the function to be called not just from the host, but also from the GPU itself (i.e., from the host or the device). There is also "__device__" for a GPU function only callable from the GPU, and "__host__" for a non-GPU host-executed function only callable from the host program.

Note that each of these function types runs on either the host or the device, but not both. However, you can declare a function as both "__device__" and "__host__" and there's a preprocessor macro "__CUDA_ARCH__" which can be used to define different blocks of code that execute on the host versus the GPU, or indeed for different types of GPU architectures.

The default meaning for a function without any specifier is the same as "__host__" where the function only runs on the CPU (not the GPU) and can only be called from the host code.

Hence, you don't usually need to use any of the specifiers except for the tight GPU kernel code.

inline functions. There are extra specifiers that control inlining optimizations of functions:

- __forceinline__
- __noinline__
- __inline_hint__

Builtin variables. There are various builtin variables or constants that are available to device programs.

- `threadIdx` — thread index in a block
- `blockIdx` — block index in a grid
- `gridDim` — grid dimensions (blocks-per-grid)
- `blockDim` — block dimensions (threads-per-block)
- `warpSize` — size of a warp (how many threads; usually 32)

Memory address specifiers. The CUDA memory model has an extended, shared address space. There are various C++ specifiers that can be applied to variables or addresses:

- `__device__`
- `__constant__`
- `__shared__`
- `__grid_constant__`
- `__managed__`

Pointer Specifiers. The CUDA language supports various extended specifiers for pointers:

- `__restrict__` for non-aliased restricted pointers to allow the auto-vectorizer to do more.
- `const` can be used at two levels in pointer declarations.

Thread synchronization functions. There are various ways to synchronize parallel execution of multiple threads. The builtin functions include:

- `cudaDeviceSynchronize()`
- `__synchthreads()`
- `__syncwarp()`

Device time functions. These functions are GPU equivalents of the standard C++ clock timing functions:

- `clock()`
- `clock64()`

Kernel Function Limitations

There are a number of limitations when writing GPU kernels in CUDA C++. Some limitations apply to the initial kernel launch and some apply more generally to any code running on the GPU. Note that the GPU-executed device portion of your C++ code is the functions with `__global__` or `__device__` specifiers.

Some of the limitations of kernel launches and device functions include:

- Stack memory size is limited to 32K (for function parameters and local variables).
- Pass-by-reference disallowed for kernel launches.
- Variable-argument functions disallowed in kernel launches.
- Copy constructor calls for kernel launches — bitwise-copy applies to object parameters.
- `static` local variables disallowed in any kernel functions (use shared memory instead).
- Global variables not available in the normal sense (use global memory, constant memory, or shared memory instead).

Some other capabilities are somewhat limited in device code:

- Function pointers
- Recursion (not recommended anyway!)
- `friend` functions
- `operator` functions
- `template` usage (some limits, but also powerful).

That's not even the full list, but there are far more C++ features that *are* supported, compared to these restrictions (e.g., basic operators, mathematical functions, control flow, etc.). Overall, these limitations are not central to coding up an algorithm in CUDA's version of SIMD parallelism on a GPU. Most of these are coding features that you can live without!

4. CUDA Emulation

CUDA CPU Emulation

Is it possible to run a CUDA program without a GPU? This is desirable for playing around to learn CUDA, or teaching a class of students about CUDA programming.

There was a CUDA emulator as part of the main toolkit, but it's since been removed. It's only supported as far back as the CUDA Toolkit 3.0 version, using a "`-deviceemu`" option.

Once upon a time there was also a PGI compiler that ran CUDA programs on a CPU. This company was acquired by NVIDIA in 2013, and the PGI compiler has since been merged into the NVIDIA HPC SDK and the PGI name and products were subsequently retired.

However, here's a solution in the cloud: Google Colab offers a free tier whereby you can run CUDA C++ code on a virtual machine. It's not really an "emulation" but more like a full GPU for free up in the cloud.

You can set up a Linux virtual environment with a real GPU attached and CUDA installed, and it's free for low-end T4 GPUs (as of this writing). You have to pay for some more advanced capabilities like A100 GPUs, but the low-end tier is fine for learning and experimenting with CUDA. I've described how to set that up further below.

CUDA C++ Emulation Library

At Aussie AI, we have implemented a CUDA wrapper library in basic C++ for emulation of a very small subset of CUDA on CPU. This is primarily useful as a learning and teaching tool, but does not support enough CUDA primitives for production usage. Find more details at https://www.aussieai.com/cuda/projects.

The idea is to run basic CUDA C++ code without a GPU, so that they can be tested in non-CUDA platforms like Microsoft Visual C++ on Windows and GCC on Linux.

The main advantages:

- No GPU needed!
- Does not need the CUDA Toolkit installed.
- Non-CUDA C++ compiler support.

This library is primarily for educational and basic testing purposes. You can run some simple kernels in a simple C++ environment, and learn some of the basics. The emulation will also detect some common failures in your basic CUDA kernels, as part of the emulation mode on CPU.

Main features. The emulator works by intercepting the CUDA primitives in basic C++, and then calling emulation versions of them.

The main capabilities include:

- Emulates several basic CUDA primitives (e.g., `cudaMalloc`)
- Runs in standard C++ on Microsoft Visual Studio on Windows and `gcc` on Linux.
- Launches CUDA kernels in emulation mode that runs the threads sequentially (simpler to debug).
- Detects various common CUDA primitive kernel errors (e.g., memory errors, double deallocation).
- Detects common kernel programming errors (e.g., array bounds violations in threads).

How it works. The basic architecture for the emulation library is:

- Source code interception in a basic C++ compiler (i.e., not NVCC).
- Preprocessor macro interception of CUDA primitives (e.g., `cudaFree`).
- Emulation of these basic CUDA primitives in simplified C++ coded versions.
- Preprocessor macro interception of C++ primitives (e.g., `malloc`, `free`).
- Link-time interception of C++ `new` and `delete` operators.
- Various error checks performed inside the emulated C++ and CUDA C++ functions.

Limitations. This emulation library is not a production-grade CUDA emulator by any means! Its value is more in the educational domain for learning CUDA basic concepts. Some of the main problems include:

- Limited subset of CUDA APIs are intercepted.
- Most CUDA library calls are not emulated.
- Syntax is not identical (e.g., the <<<...>>> kernel launch syntax must be modified).
- Synchronization across threads in CUDA kernels is not properly emulated.
- Shared memory usage in threads is not emulated.

This emulation library may be extended or modified. Feel free to use it to learn CUDA with my best wishes on your success.

Running CUDA in Google Colab

An alternative to using CUDA Toolkit on your own machine is to run it in the cloud on someone else's GPU. Google Colab is a free online environment for running and testing code in a virtual Linux box.

It's not really an "emulation" but it can feel like it. You can test CUDA C++ programs using `nvcc` compiler and real GPU hardware somewhere underneath the virtual layers.

And did I mention: for free!

The steps are basically:

1. Open a new notebook in Google Colab
2. Change the "runtime" to be a GPU (e.g., T4 GPU)
3. Upload a CUDA C++ file to Google Colab (e.g., "`test1.cu`")
4. Run the `nvcc` compiler.
5. Run `a.out` (the executable)
6. Save your notebook.

More details on each step are given below.

1. Open a new Google Colab virtual notebook. You need to follow these steps:

- You'll need to be signed in to your Google Gmail account, or create a Google account.
- Navigate your browser to Google Colab: https://colab.research.google.com/
- Click on `File > New Notebook`

2. Change the Notebook's Runtime to GPU. The steps in more detail:

- Click on Runtime > Change runtime type
- Choose a GPU, such as "T4 CPU" (free). Or you can pay more for A100 GPU environment. But you don't need more than the free one to test simple CUDA C++ code.
- Click "Save" to confirm your choice of GPU mode.
- Now you have a virtual Linux box which is setup for GPU, including with the CUDA Toolkit installed virtually.
- You don't need to do any steps to install CUDA or nvcc.

3. Upload a CUDA C++ file. The steps to upload your source code file:

- Store your CUDA C++ code on your PC in a single file (for simple examples), ready for upload.
- Ensure the file suffix is "`.cu`" or "`.cpp`" (e.g., `test1.cu`)
- Click on the "Folder" icon in Google Colab (an icon on the LHS vertical panel).
- This will expand out a view of your virtual files and folders.
- By default, you are probably in the "`/content`" directory on the virtual Linux filesystem.
- Click on the "Upload" icon (top LHS icon, with an up arrow on top of a file icon).
- Choose your "`test1.cu`" file from your local PC drive.
- Confirm your upload choice in the file browser (e.g., click "Open" on Windows).
- The newly uploaded file should, after a brief delay, appear in the files and folders view on Google Colab.

4. Run nvcc to compile your CUDA C++ file. Here are the steps:

- Create a new "+Code" cell in Google Colab.
- Edit the new cell to have a command like: `!nvcc test1.cu`
- Note that "!" is required, and means to run the command in a Cell. Also, use lower case letters.
- Note that "`nvcc`" in lower case letters is the command for the NVIDIA C++ Compiler (NVCC).
- Click on the "Play" (triangle) button or "run cell" to execute this new cell.
- This should run the `nvcc` CUDA C++ compiler to create your executable file into "`a.out`".
- Wait for the Cell to finish executing (i.e., wait for the button icon to stop spinning).
- After a brief delay, you should see a new file called "`a.out`" appearing in the Files/Folders view.

Failed compilation. If your CUDA C++ code has a compilation error, `nvcc` won't create an executable file, and you'll get some error messages instead appearing inside the cell's output area.

- If there's not a new `a.out` file in the Folder view, `nvcc` probably failed to compile, because of a syntax error in your CUDA C++ code. Review the warnings from `nvcc`.
- Edit your CUDA C++ source file to fix any errors.
- You can edit it in the virtual environment by double clicking on the filename. This opens a text editor in your Google Colab notebook, but note that you'll lose any changes if your notebook shuts down.
- Alternatively, you can re-edit the file on your PC and re-upload the edited file to Google Colab.
- Re-run the `nvcc` cell to compile the newly edited CUDA C++ file and create "`a.out`".

5. Run your a.out executable.

- Create another "+Code" cell in Google Colab.
- Use command: `!a.out`
- Note that "`!`" means run the command, and "`a.out`" in lower case letters is the name of the executable.
- Click on the "`Play`" (triangle) button to run the cell.
- The output from your CUDA C++ program should appear.
- Hooray!

6. Save your notebook (optional). Note that your uploads to Google Colab are not automatically saved. That's too much to expect for a free service. It will eventually time out, and your uploaded files will also disappear from your notebook folders if you close your browser. If you've edited these files inside Google Colab, you lose your changes.

One partial fix is to create backups of your notebook, either on your PC or in Google Drive. There is a "Download" option for your entire notebook. For Google Drive backups, when inside Google Colab, use the "`File > Save a copy in Drive`" menu. However, this doesn't seem to save and restore your uploaded files, but only the "notebook" part with all the cells.

A better fix to save all files and also avoid manually backup and restore of your entire notebook is to map Google Drive into your folder hierarchy. The idea is to "mount" your Google Drive files as a subdirectory inside your Colab notebook. Then you can save the files into that folder in Colab, and they'll then be stored in Google Drive. Example command to run:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

After this, if you upload or edit files in the "`gdrive`" folder, then they're in your Google Drive.

You can upgrade to a paid version to get the capability to store a notebook in your account. Alternatively, you can just repeat the steps each time you navigate to Google Colab, assuming that your CUDA C++ files are being edited on your local box, and not virtually in the notebook.

Troubleshooting Problems on Google Colab

I had a few problems with the CUDA source file getting uploaded to the wrong virtual directory in Google Colab, sometimes ending up in the parent directory (probably user error). This result was this sort of error from `nvcc`:

```
cc1plus: fatal error: test1.cu: No such file or directory
compilation terminated.
```

Maybe you've used the wrong filename, or maybe it's in a different subdirectory.

You can check where your "`test1.cu`" file is in the file hierarchy on the LHS by clicking on the Folder icon.

To see your current directory where `nvcc` is running in a Cell, create a new Code cell with "`!pwd`" command and run it ("`pwd`" is the Linux command for "print working directory").

You can also run "`!ls`" (without any quotes) to list the files in the current working directory in your virtual notebook.

If you somehow get `nvcc` running in "`/content`" but the "`.cu`" file in a higher directory, use this command in the cell to get `nvcc` to find the CUDA file in the parent directory:

```
!nvcc ../test1.cu
```

You might also get this type of error message:

```
nvcc fatal : Don't know what to do with 'test1.cu.txt'
```

This error is the wrong file suffix given to `nvcc` (i.e., "`.txt`" rather than "`.cu`" here), which is a reminder of the joyful experience of Windows protecting me from things.

It's hard to rename the file suffix in File Explorer from "`.txt`" to "`.cu`" and usually I have to resort to the DOS "`ren`" command in a command shell, but I digress.

No output appeared. Did any output appear?

If absolutely nothing appears from your CUDA "hello world" program (i.e., with `printf` in the GPU kernel), and there's no compile errors from `nvcc`, and no errors or runtime output from `a.out`, maybe you've made a common mistake of not calling `cudaDeviceSynchronize`, as discussed earlier in the chapter.

At the risk of repeating myself, CUDA kernel launches are asynchronous and `main` does not wait for your GPU code to finish, unless you force it to. Also, any `printf` inside a CUDA kernel on the GPU does not ever appear if the CPU code has already exited.

The code has run so fast that it all finished before any output got generated properly, so it shows nothing.

The solution is to add a call to `cudaDeviceSynchronize` after the kernel launch, or at the end of `main`, which forces the CPU to wait for the GPU kernel to finish.

5. Debugging Simple Kernels

Grid Dimensions

One of the most common parts of a kernel is to check its own position in the grid. The builtin variables to use are:

- `gridDim` — grid dimension (how many blocks in the grid)
- `blockDim` — block dimension (how many threads per block)
- `blockIdx` — block index (in a grid)
- `threadIdx` — thread index (in a block)

Each thread in a grid has a unique pair of values for the block index and thread index.

The typical CUDA idiom for the starting offset into a linear grid looks like this:

```
int starti = blockDim.x * blockIdx.x + threadIdx.x;
```

The above computation has a unique value in each thread, which ensures that each thread is working on a different index, such as when processing a vector or other linear array.

Note that the thread and block index values start at zero, rather than one, like C++ array offsets. The thread index ranges from zero to `blockDim.x-1`, and the block index ranges from zero to `gridDim.x-1`. These values are different within each thread (or block), and also stay constant within each thread until completion.

Dimension values are non-zero and fixed during execution of a thread. Neither the block dimension (threads-per-block) nor grid dimension (blocks-per-grid) can be zero.

All blocks in a grid have the same number of threads, so these dimension values should be the same for all of the threads executing a given kernel, and not change while executing the kernel.

For a fixed-size operation on an AI model, these builtin dimension functions could actually be replaced by a numeric constant. However, this micro-optimization is not usually worth doing. The optimizer in the `nvcc` compiler or the `ptxas` assembler is presumably handling this behind the scenes anyway.

The kernel in each thread may not need to know this, but the total number of threads in a grid can be calculated by multiplying `gridDim` (how many blocks in the grid) and `blockDim` (how many threads per block):

```
int total_threads = blockDim.x * gridDim.x;
```

Usually, the "`.x`" attribute is accessed from these builtin variables to get the value. However, these variables are all objects of type "`dim3`" and have three parameters: `x`, `y`, and `z`.

The most common type of grid is a "linear grid" which has an `x` value, but the `y` and `z` values are zero. However, advanced grids can be two-dimensional or three-dimensional, with non-zero values for `y` and/or `z`.

One-Dimensional Vector Kernels

Let's consider a vector addition kernel doing an in-place operation:

```
y = y + x
```

There are many ways to do so with different kernel functions:

- Single addition per kernel
- Single addition with a protective `if` statement
- Single block with a block-stride loop
- Looping over a strip
- Looping with a general grid stride

Generally, the main way that experienced CUDA programmers will handle this is a "grid stride" loop. But let's have a look at some of the other methods.

Single Operation Kernel

In this simple type of kernel, each thread does the addition on a single vector element. No loops. The main point is to launch enough threads for cover all the n elements in the vectors.

```
__global__
void aussie_vector_add(float *x, float *y, int n)
{
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    y[id] = x[id] + y[id];
}
...
int threads_per_block = 256;
int blocks = n / threads_per_block;
aussie_vector_add<<< blocks, threads_per_block >>>(x, y, n)
```

Bug! This code actually has an error if n is not an exact multiple of 256, due to integer division truncation. We launch one block too few, and the last few "extra" elements of the vectors won't get added.

One fix is to ensure we account for the extra cases to launch one more block, with the modified computation:

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

But this has another insidious and non-obvious bug, as discussed below.

Kernel Safety Checks

The problem with the basic single-operation kernel show above is that sometimes the value of `id` exceeds `n`. This occurs because if `n` is not an exact integer multiple of "`threads_per_block`" value. The above code launches one more block to handle the extra cases, but the problem is that it launches 256 threads for that block, even if there aren't 256 extra cases. These threads don't know any better, and try to run, with the index computation:

```
int id = blockDim.x * blockIdx.x + threadIdx.x;
```

In these problematic threads, this creates an index value for `id` that is larger than or equal to `n`. Then the accesses `y[i]` and `x[i]` are array bounds violations in global memory.

Kaboom!

One way to solve this is to always ensure the `n` is a multiple of a reasonable block size, such as 256. Note that this should be a multiple of the warp size, which is usually 32.

This is quite a viable plan in AI, because most of the Transformer operations can be chosen so that vectors, matrices and tensors have fixed dimensions that are multiples of 32.

That trade-off comes with some risks, and defensive coding says to do more to protect against crashes. Hence, another safer solution to the array bounds error is to add a protective `if` statement in every thread:

```
__global__ void aussie_vector_add(float *x, float *y, int n)
{
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    if (id < n) {  // Safety!
        y[id] = x[id] + y[id];
    }
}
...
int threads_per_block = 256;
int blocks = ( n + threads_per_block - 1) / threads_per_block;
aussie_vector_add<<< blocks, threads_per_block >>>(x, y, n)
```

All of the threads now must execute an extra "`if`" comparison. In the problematic threads, the `if` statement ensures that no work is done in the extra threads. This is a waste of GPU resources in launching unnecessary threads, but at least it doesn't crash.

One possible compromise solution is to change the `if` statement to an assertion, which you allow to run live during test runs:

```
__global__ void aussie_vector_add(float *x, float *y, int n)
{
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    assert(id < n);
    y[id] = x[id] + y[id];
}
```

This will actually prevent the crash because the builtin kernel `assert` primitive stops the thread. However, it's probably not fault-tolerant for the entire application, because the kernel launch will return a `cudaErrorAssert` runtime error.

Then, for performance reasons, you can remove the assertions from production builds by defining the special macro `NDEBUG` and ship that to customers (fingers crossed!). You should also add some assertions at the kernel launch to ensure that the total number of threads you need is an exact multiple of the block size:

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

Note that there's no way in CUDA to launch blocks of different sizes. We can't launch most of the blocks with a fixed thread size like 256, and then launch one extra block with a few extra threads. If we try to work around this by launching two kernels with the same function (i.e., the main blocks and one "extra" block), the block index (`blockIdx`) and block dimension (`blockDim`) will have values that are wrong in the threads for the second one. Hence, we need to either guarantee at the algorithmic level an exact multiple of the thread size (fast), or add safety checks inside the kernel (slow).

Two-Dimensional Matrix Kernels

Things get trickier in two-dimensional kernels, such as for matrix operations. Let's look at a matrix addition operation, which generalizes the vector addition idea.

Vector kernels need only use the "`.x`" attributes, but matrix kernels also need to look at the "`.y`" values as well. Here is the two-dimensional index calculation:

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
```

However, you can't do this with dynamic array sizes:

```
m3[x][j] = m1[x][y] + m2[x][y];
```

On the other hand, this array syntax works for fixed-size matrices, if the dimensions are known at compile-time. And if you can guarantee that, the above operation would very fast, so don't let me discourage you!

Instead, for dynamic arrays in two dimensions, you need to "linearize" the matrix offsets into a single flat one-dimensional array, which contains all the matrix elements, ordered one row at a time.

The calculation is:

```
int id = x + y * nx;
```

Our full kernel for matrix addition looks like this:

```
__global__ void matrix_add(
    float *m3,
    const float *m1,
    const float *m2,
    int nx, int ny)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int id = x + y * nx; // Linearize
    m3[id] = m1[id] + m2[id];
}
```

Again, we have a problem with unsafe array bounds violations if there are extra threads launched.

The safer version is:

```
__global__ void matrix_add_safe(
    float *m3,
    const float *m1,
    const float *m2,
    int nx, int ny)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   if (x < nx && y < ny) {
       int id = x + y * nx; // Linearize
       m3[id] = m1[id] + m2[id];
   }
}
```

This is safe but marginally inefficient if there are extra cases.

As before, there are other options such as using assertions, or being extra careful to ensure that our matrix dimensions are fully consistent with the grid dimensions, so that there aren't any extra threads.

Warps and Lanes

So far, we've been talking about blocks and threads, but let's add another wrinkle. There's a reason that the number of threads should be divisible by 32: warps.

What do you call a group of 32 threads?

No, it's not a block, but a "warp." Typical CUDA blocks run multiple warps, each containing 32 threads, which is why block sizes need to be a multiple of the warp size, which is 32.

Hence, our revised terminology is:

- Threads — single execution unit.
- Warp — 32 threads.
- Block — multiple warps (of 32 threads each).
- Grid — multiple blocks.

The GPU kernel code sometimes needs to know the details of which warp it's in, just like we've already seen it work out its thread offset. You can calculate which warp a kernel thread is in by examining the thread index in a block.

```
int warpid = threadIdx.x / 32;
```

If the block size is 64 threads, there are 2 warps, and the first 32 threads will get 0 here, and the latter 32 threads will get 1 as the warp index.

Lanes. Within a warp, the 32 threads are actually numbered 0..31. These are called the "lanes" for a warp, and you can calculate each thread's lane from the thread index using the integer remainder operator:

```
int laneid = threadIdx.x % 32;
```

This will compute the "lane" from 0..31. Each thread in a warp has a different lane, but multiple threads across a block can have the same lane.

If you want to be cleaner in your coding, use a named constant for the warp size:

```
const int WARP_THREADS = 32;
int warpid = threadIdx.x / WARP_THREADS;
int laneid = threadIdx.x % WARP_THREADS;
```

Or if you want dirtier coding that bets against the compiler design engineers, you can needlessly micro-optimize:

```
int warpid = threadIdx.x >> 5;
int laneid = threadIdx.x & 0x1F;
```

Lanes are not needed for simple kernels, but they are very important when doing more complex kernels. For example, using the warp shuffle operations to optimize a kernel to avoid shared memory will require careful tracking of the lane index within each of the threads.

6. Debugging Strategies

General Debugging Techniques

A lot of the work in debugging CUDA programs is nothing special: it's just C++ mistakes. Most of the errors in coding are ordinary, boring coding errors that every C++ programmer is prone to.

These can occur in the host code and the device code, although problems in the host code are more common.

On the other hand, if you get a bug in a CUDA C++ kernel, it's usually a nasty one. And there are a variety of ways to go wrong in handling pointers and addresses in the kernel, from basic beginner mistakes to traps that can catch the experienced CUDA practitioner.

The best way to catch a bug is to try to make it happen *early*.

We want the program to crash in the lab, not out in production. In this regard, some of the best practices are about auto-detecting the failures in your code, rather than waiting for them to actually cause a crash:

- Check every CUDA API return code (even the harmless functions that can "never" fail).
- Use macro wrappers to help handle errors.
- Add debug wrapper functions and enable them while testing.
- Run `compute-sanitizer` on your code regularly.
- Thrash the code in many ways in the nightly builds.

If you mess up, and a bug happens in the production backend of your AI training run, I suggest this: blame the data scientists. Surely, the problem was in the training data, not in my perfect CUDA C++ code. And if that doesn't work, well, obviously the GPU was overheating.

Very Difficult Bugs. Some bugs are like roaches and keep coming out of the woodwork. General strategies for solving a tricky bug include:

- Can you reproduce it? That's the key.
- Write a unit test that triggers it (if you can).
- Try to cut down the input to the smallest case that triggers the fault.
- Gather as much information about the context as possible (e.g., if it's a user-reported error).

Your debugging approach should include:

- Run `compute-sanitizer` to check for CUDA memory glitches.
- Run the other `compute-sanitizer` tools (it has four modes).
- Think about what code you just changed recently (or was just committed to the repo by someone else!).
- Memory-related failures often cause weird errors nowhere near the cause.
- Review the debug trace output carefully (i.e., may be that some other part of the code failed much earlier).
- Step through the code in `cuda-gdb` bout ten more times.
- Run a static analysis ("linter") tool on the code.
- Run an AI copilot debugger tool. I hear they're terrific.
- Refactor a large module into smaller functions that are more easily unit-tested (often you accidentally fix the bug!).

If you really get stuck, you could try talking to another human (gasp!). Show your code to someone else and they'll find the bug in three seconds.

Serializing Kernel Launches

Both beginner and advanced CUDA C++ programmers can make debugging easier via serialization of thread execution. Some basic strategies are:

- Serialize the kernel launches.
- Launch only one thread (i.e., `kernel<<<1,1>>>`)

Serialized kernel launches. The advantage of a serialized kernel launch is that only one kernel is running at a time.

This doesn't mean that the threads are running sequentially within that one kernel, but at least you don't have two things happening on the GPU at once. This is a great help in localizing the cause of any CUDA Runtime errors, which can come in asynchronously from any active kernel.

Serializing kernel launches is possible in several ways. One simple way is to manually add a call to `cudaDeviceSynchronize` immediately after every kernel launch. Since beginner programmers often already do this in their code, it's not going to add much benefit to a debugging session in the learning lab.

CUDA kernel launches are usually asynchronous, but you can make them synchronous or "blocking" using the settings to auto-serialize every kernel launch. You can set the environment variable "`CUDA_LAUNCH_BLOCKING`" to 1.

cuda-gdb serialized kernels. There are additional options when debugging your code in the `cuda-gdb` symbolic debugger. There are various flags you can set within an interactive debugging session. The one to serialize all kernel launches so they are "blocking" is to enable the "`launch_blocking`" setting:

```
set cuda launch_blocking on
```

Another useful `cuda-gdb` option is to set an auto-breakpoint on every kernel launch in your program with the "`break_on_launch`" setting.

The command is:

```
set cuda break_on_launch application
```

Single-thread kernels. Launching only a single thread with `kernel<<<1,1>>>` is also not really for beginners. If your kernel is really simple, such as a basic vector addition with one "+" operation on a single element, then your program simply won't work anymore. If you run only one thread, your kernel will only process one vector element.

The advice to launch a single thread is more relevant to advanced kernels that use a grid-stride loop. A single kernel like that will check the value of `blockDim.x`, which will now be 1, and will adjust the loop to iterate over every element of a vector. But, again, using a grid-stride loop is not for beginners.

Localizing the Error

One of the basic techniques in debugging for large and complex CUDA programs involves localizing the error. The problem arises because of these factors:

- CUDA kernel launches are asynchronous, so the host code keeps running.
- The CUDA Runtime API does not report GPU kernel errors immediately.
- Multiple kernels may be running in parallel.

The result of asynchronous kernel launching is a weird sequence, whereby a GPU error report can come back to the host code at any time. This assumes that the CPU kept going after launching the kernel launch, rather than blocking on a `cudaDeviceSynchronize` call.

Here's an example sequence of events:

- Host code launches a kernel (triple chevrons and all that).
- The CPU code keeps moving ahead (because the launch is non-blocking).
- The device code for that kernel starts running on the GPU.
- Stuff happens (on both CPU and GPU).
- An error occurs in the GPU kernel (for some reason).
- The GPU cannot interrupt the host code on the CPU.
- Instead, the GPU buffers the error code.
- The next call to the CUDA Runtime library on the host will return this error code.

Hence, there's some weird problems:

- The error code might cause a failure in some CUDA APIs that you think should never fail (e.g., `cudaSetDevice`, `cudaGetDeviceProperties`, or whatever).
- The error code might appear to be from setting up a new kernel (e.g., `cudaMemcpy` fails), but in fact, it's an error from the prior kernel launch.
- The error code might occur after a second kernel is launched, and you might think it's from the second kernel, when it's actually from the first kernel.

Random Number Seeds

Neural network code often uses random numbers to improve accuracy via a stochastic algorithm. For example, the top-*k* decoding uses randomness for creativity and to prevent the repetitive looping that can occur with greedy decoding. And you might use randomness to generate input tests when you're trying to thrash the model with random prompt strings.

But that's not good for debugging! We don't want randomness when we're trying to reproduce a bug!

Hence, we want it to be random for users, but not when we're debugging. Random numbers need a "seed" to get started, so we can just save and re-use the seed for a debugging session.

This idea can be easily applied in the code for old-style `rand/srand` functions or to all the newer `<random>` libraries like `std::mt19937` (stands for "Mersenne twister").

Seeding the random number generator in old-style C++ is done via the "`srand`" function. The longstanding way to initialize the random number generator, so it's truly random, is to use the current time:

```
srand(time(NULL));
```

Note that seeding with a guessable value is a security risk. Hence, it's safer to use some additional arithmetic on the `time` return value.

After seeding, the "`rand`" function can be used to get a truly unpredictable set of random numbers. The random number generator works well and is efficient.

A generalized plan is to have a debugging or regression testing mode where the seed is fixed.

```
if (g_aussie_debug_srand_seed != 0) {
    // Debugging mode
    srand(g_aussie_debug_srand_seed); // Non-random!
}
else {  // Normal run
    srand(time(NULL));
}
```

The test harness has to set the global debug variable:

```
g_aussie_debug_srand_seed
```

This is set for a regression test. For example, either it's manually hard-coded into a testing function, or it could be set via a command-line argument to your test harness executable, so the program can be scripted to run with a known seed.

This is better, but if we have a bug in production, we won't know the seed number. So, the better code also prints out the seed number (or logs it) in case you need to use it later to reproduce a bug that occurred live.

```
if (g_aussie_debug_srand_seed != 0) {
    srand(g_aussie_debug_srand_seed);   // Debug mode
}
else {  // Normal run
    long int iseed = (long)time(NULL);
    fprintf(stderr, "INFO: Random number seed: %ld 0x%lx\n",
         iseed,
         iseed
         );
    srand(iseed);
}
```

An extension would be to also print out the seed in error context information on assertion failures, self-test errors, or other internal errors.

There's one practical problem with this for reproducibility: what if the bug occurs after a thousand queries? If there's been innumerable calls to our random number generator, there's not really a way to reproduce the current situation.

One simple fix is to instantiate a new random number generator for every query, which really isn't very expensive.

Making the Correction

An important part of the debugging phase that is often neglected is actually making the correction. You've found the cause of the failure, but how do you fix it? It is imperative that you actually understand what caused the error before fixing it; don't be satisfied when a correction works and you don't know why.

Here are some thoughts on the best practices for the "fixing" part of debugging:

- Test it one last time.
- Add a unit test or regression test.
- Re-run the entire unit test or regression test suite.
- Update status logs, bug databases, change logs, etc.
- Update documentation (if applicable)

Another common pitfall is to make the correction and then not test whether it actually fixed the problem. Furthermore, making a correction will often uncover (or introduce!) another new bug. Hence, not only should you test for this bug, but it's a very good idea to use extensive regression tests after making an apparently successful correction.

Level Up Your Post-Debugging Routine. Assuming you can fix it, think about the next level of professionalism to avoid having a repetition of similar problems. Consider doing followups such as:

- Add a unit test or regression test to re-check that problematic input every build.
- Write it up and close the incident in the bug tracking database like a Goody Two-Shoes.
- Add safety input validation tests so that a similar failure is tolerated (and logged).
- Add a self-check in a C++ debug wrapper function to check for it next time at runtime.
- Is there a tool that would have found it? Or even a `grep` script? Can you run it automatically? Every build?

As with all applications, there's another level needed to get the code out the door into production. Some of the issues for fully production-ready CUDA C++ code include:

- Validate function parameters (don't trust the caller or the user).
- Check return codes of all CUDA primitives.
- Handle memory allocation failure (e.g., graceful shutdown).
- Kernels should correctly scale for large values (e.g., vector dimensions).
- Choose block/thread sizes for best occupancy
- Don't exceeding GPU device specifications.
- Add unique error message codes for supportability

Let's not forget that maybe a little testing is required.

High-quality coding requires all manner of joyous programmer tasks: write unit tests, warning-free compilation, static analysis checks, add assertions and debug tracing, run `cuda-memcheck`, write useful commit summaries (rather than "I forget"), don't cuss in the bug tracking record, update the doc, comment your code, and be good to your mother.

7. CUDA Debugging Tools

CUDA Tools Overview

Compiler and IDE tools for programming CUDA include:

- NVIDIA C++ Compiler (NVCC) — the `nvcc` command-line compiler.
- Nsight Eclipse Edition — integration with the Eclipse IDE.
- Nsight Visual Studio Edition (VSE) — CUDA's Microsoft Visual Studio integration.
- Nsight Visual Studio Code Edition (VSCE) — Visual Studio Code integration.

Debugging tools include:

- `cuda-gdb` — command-line debugging on Linux (very similar to `gdb`).
- Compute Sanitizer — command-line debugging tool with four sub-tools: `memcheck` for memory debugging, `racecheck` for race conditions, `synccheck` for synchronization checking, `initcheck` for initialization checks.
- `cuda-memcheck` — discontinued tool, replaced by `compute-sanitizer` and its `memcheck` default tool.

Optimization and performance profiling tools 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.
- `nvprof` — command-line profiler (now deprecated)

In-code debugging libraries and CUDA C++ code-related tools that you might need include:

- Debug wrapper library for CUDA C++
- Emulation test library
- Linter for CUDA C++

These three major ideas are active coding projects for us here at Aussie AI (see https://www.aussieai.com/cuda/projects).

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

- Compute Sanitizer API — create a new "tool" for `compute-sanitizer`.
- CUDA Debugger API — `cuda-gdb` API integration on Linux.
- NVIDIA Tools Extension SDK (NVTX) — tool integration API.
- CUDA Profiling Tools Interface (CUPTI) — profiling and tracing integration API.
- Nsight Aftermath SDK — postmortem crash debugging.
- Nsight Perf SDK — performance profiling for graphics applications.
- Nsight Tools JupyterLab Extension — extension for profiling of Python applications.

The remainder of this chapter focuses on the debugging tools and their capabilities.

Command-Line Debugging Tools

The main command-line debugging tools to use are:

- `cuda-gdb` (interactive debugger on Linux or Windows WSL)
- `compute-sanitizer` – including four sub-tools:

    a) `memcheck`,
    b) `initcheck`
    c) `racecheck`
    d) `synccheck`

Profiling tools you can use on the command-line include:

- `ncu` (Nsight Compute CLI)
- `nvprof` — useful, but it's deprecated, and will be riding off into the sunset.
- `gprof` — the standard Linux profiler is useful for host code.

Compute Sanitizer

Compute Sanitizer is an NVIDIA tool that has four sub-tools to detect different problems. In addition to multiple tool capabilities, it also supports multiple platforms: Linux and Windows.

The most frequent usage of the `compute-sanitizer` command-line tool is likely to be memory debugging. If you have an error in a CUDA C++ application on Linux, and you can reproduce it, then just re-run the application with the Compute Sanitizer tool:

```
compute-sanitizer a.out
```

This is equivalent to:

```
compute-sanitizer --tool memcheck a.out
```

You can also supply command-line arguments to your application after the executable name, or additional options to Compute Sanitizer before the executable name.

The default tool for `compute-sanitizer` is "`memcheck`" for memory fault detection, and you don't need a specific option to run it. This mode is very similar in usage to `valgrind` on Linux, which does similar memory checking functions on standard CPU C++ applications, but `compute-sanitizer` knows more about GPU memory problems. There also used to be a `cuda-memcheck` tool, which this has now superseded.

Compute Sanitizer does not require any re-compilation, and can run on your program just after it has crashed. However, its reports can be clearer to read if there is debug symbol information available in the executable, such as compilation with "`-g`" (host) or "`-G`" (device) debug information options to `nvcc`. Hence, it may be advisable to maintain these debug-enabled versions of executables in your build systems, as they are useful for both `cuda-gdb` and `compute-sanitizer`.

The error reported by the memory checking tools include:

- Memory access problems (including device-side)
- Errors with `malloc` and `free`
- Double `free`
- Memory leaks (especially with the "`--leakcheck=full`" option)

However, it is not limited to memory errors, and also finds:

- CUDA runtime errors
- Hardware exceptions

There are various additional options that you can turn on for additional error checking.

Abnormal program termination

One of the things about `compute-sanitizer` that can be tricky is that it doesn't fully detect the cause of actual crashes of your application. Instead, sometimes you only get a report like this:

```
========= Error: process didn't terminate successfully
========= Target application returned an error
========= ERROR SUMMARY: 0 errors
```

The last line is misleading, as there weren't zero errors, but the second-last line is more useful: "`Target application returned an error`". This often means that your program crashed in the host code.

cuda-gdb batch mode. You can detect this host program crash better in `cuda-gdb` as it will trap the signals, so just run an interactive debugging sessions. Alternatively, if you have a simple reproducible case, you can automate this with batch mode, where the command to run is like this:

```
cuda-gdb --batch --command=cuda-gdb-test.txt a.out
```

The batch input file is a set of `cuda-gdb` commands:

```
run
where
exit
```

Here's an example output (abridged):

```
Thread 1 "a.out" received signal SIGSEGV, Segmentation fault.
0x00007ffff7cdfa4e in ?? () from /lib/x86_64-gnu/libc.so.6
#0  0x00007ffff7cdfa4e in ??() from /lib/x86-gnu/libc.so.6
#1  0x000055555555fdb5 in aussie_cudaMalloc(void**, int)()
#2  0x0000555555562ea9 in aussie_run_clearvec_generic(int)()
#3  0x00005555555633aa in main ()
A debugging session is active.
Inferior 1 [process 5143] will be killed.
Quit anyway? (y or n) [answered Y; input not from terminal]
```

There are various other useful things that can be automated using batch `cuda-gdb` and various script commands. For example, you can use it as a trace mechanism that prints out the stack trace at every call to a certain function.

racecheck

The `racecheck` tool is a sub-tool of Compute Sanitizer for detecting "race conditions" or "data races." The command to run the tool is:

```
compute-sanitizer --tool racecheck a.out
```

This tool detects problems in thread accesses to shared memory, but won't help with any race conditions involving global memory accesses.

It works by detecting "hazards," which means conditions that indicate the potential for a race condition occurring. This is more effective than trying to detect actual race conditions, as they are transient and often non-reproducible.

The types of hazards found include:

- Write-write
- Write-read
- Read-write

Obviously, the accessed must be occurring to the same memory location for a hazard to exist. Also, note that a "read-read" sequence is never going to be a hazard for a race condition, but is just normal parallelism!

synccheck

The `synccheck` tool is a sub-tool of Compute Sanitizer that detects synchronization issues. It looks for "hazards" that indicate thread synchronization problems in GPU kernels. Here is the execution command:

```
compute-sanitizer --tool synccheck a.out
```

This tool focuses on synchronization issues in device code, such as with the APIs:

- `__syncthreads()`
- `__syncwarp()`

Some of the errors found by `synccheck` include:

- Invalid arguments
- Thread divergence (warp-level or block-level)

initcheck

The `initcheck` tool is a sub-tool of Compute Sanitizer for detecting initialization issues in device accesses to device global memory.

Execution is performed by:

```
compute-sanitizer --tool initcheck a.out
```

Note that this only examines global memory, so it won't find other types of uninitialized memory accesses in kernels. The default `memcheck` tool in `compute-sanitizer` can find other similar problems.

The main issues found by `initcheck` are:

- Accesses to uninitialized device global memory.
- Unused device global memory (never accessed).

Fixing an uninitialized memory access issue found by `initcheck` is usually to initialize your device memory properly, such as by:

- Device-side array initialization code
- `cudaMemset`
- `cudaMemcpy`

If `initcheck` reports on unused device global memory, this may indicate some sort of algorithm error, whereby some of the global memory is not being used.

cuda-gdb

I'm a big fan of `gdb` for debugging standard C++ on Linux, and `cuda-gdb` is even better. The platform support for `cuda-gdb` is primarily Linux, but also Windows WSL2, and also MacOS in a "host only" mode for remote debugging.

The `cuda-gdb` tool is a source-code modification of the open source `gdb` code to add NVIDIA GPU support, and you can actually find the `cuda-gdb` source code on Github at https://github.com/NVIDIA/cuda-gdb/.

Hence, `cuda-gdb` has most of the `gdb` features, and NVIDIA tries to keep up with new features. The basic commands from `gdb` are all supported:

- `r` or `run` — run the code (with optional arguments), or restart if already running.
- `c` or `continue` — continue running (after stopping at a breakpoint).
- `s` or `step` — stepping through statements (also just `Enter`).
- `where` — stack trace (also aliased to "`bt`" for backtrace).
- `list` — source code listing
- `p` or `print` — print a variable or expression.
- `up`
- `n` or `next`

Some of the CUDA-specific commands inside `cuda-gdb` include:

```
info cuda
```

To see the list, run the help command:

```
help info cuda
```

Examples of some of the many CUDA debugging sub-commands include:

```
info cuda devices
info cuda sms
info cuda warps
info cuda lanes
info cuda blocks
info cuda threads
```

Pre-Breakpointing Trick

One advanced tip for using `cuda-gdb` is to define a function called "`breakpoint`" in your C++ application. Here's an example:

```
void breakpoint()
{
    volatile int x = 0;
    x = 0;  // Set breakpoint here
}
```

It looks like a silly function, but it serves one useful purpose. The idea is that when you start an interactive debugging session in `cuda-gdb`, or automatically in your "`.cuda-gdbinit`" resource file, you can set a breakpoint there:

```
b breakpoint
```

Why do that? The reason is that you also add calls to your "`breakpoint`" function at relevant points in various places where failures can occur:

- CUDA error check macros
- Assertion macros
- Debug wrapper function failure detection
- Unit test failures

Hence, if any of those bad things happen while you're running interactively in the debugger, you're immediately stopped at exactly that point. If you're not running in the debugger, this is a very fast function (though admittedly, it can't be `inline`!), so it doesn't slow things down much.

You can even consider leaving this debugger breakpoint in production code, since the `breakpoint` function is only called in rare situations where a serious failure has already occurred, in which case execution speed is not a priority.

This technique is particularly useful because don't have to go back and figure out how to reproduce the failure, which can be difficult to do for some types of intermittent failures from race conditions or other synchronization problems. Instead, it's already been pre-breakpointed for you, with the cursor blinking at you, politely asking you to debug it right now, or maybe after lunch.

Postmortem Debugging

Postmortem debugging involves trying to debug a program crash, such as a "core dump" on Linux. In this situation, you should have a "core" file that you can load into `cuda-gdb`.

The command to use is:

```
cuda-gdb a.out core
```

Unfortunately, not all errors in a CUDA application will trigger a core dump, so you might have nothing to debug if it doesn't. One way to ensure that you get a `core` file is to set the environment variable:

```
CUDA_ENABLE_COREDUMP_ON_EXCEPTION
```

This will cause the CUDA Runtime to produce a `core` file on various additional failures.

Sometimes in large environments, it's hard to know where a `core` file came from. An advanced feature of this environment variable is that you can format the filename to be more useful than just "`core`." You can specify the format to include things like the time/date and the name of the executable.

Programmatic C++ core dumps. If you're wanting to have your CUDA C++ take control of its own core dumps (e.g., exceptions, assertion failures, etc.), there are various points:

- You can always `fork`-and-`abort` on Linux.
- Maybe `putenv("CUDA_ENABLE_COREDUMP_ON_EXCEPTION")` migh t work?
- Surely you can write some code to crash!

On the other hand, maybe you're only thinking about core dumps because you want to save debug context information. Doing this might obviate the need for a core dump:

- Use `std::backtrace` or another backtrace library.
- Print error context information (e.g., user's query)
- Print platform details

Customer core dumps. One of the supportability issues with postmortem debugging is that you want your customers to be able to submit a `core` file that they have triggered on your CUDA-based application. These are usually large files, so there are logistical issues to overcome with uploads.

Another issue is that in order to run `cuda-gdb` on a `core` file, the developer needs to have exactly the right executable that created the core dump. Hence, your build and release management needs to maintain available copies of all executable files in versions shipped to customers or in beta testing (or to internal customers for in-house applications). And there needs to be a command-line option whereby the phone support staff can instruct customers to report the exact version and build number of the executable they are using. It's easy to lose track!

Valgrind for CUDA

Can you use the Linux Valgrind tool to detect memory errors in CUDA C++ programs? More specifically, this refers to the Memcheck tool that is part of Valgrind.

In short, you can run Valgrind on your host code, but you're probably better off using `compute-sanitizer` for CUDA C++ programs. Valgrind does run both the host and the device code, and can be used to find errors.

The CUDA Toolkit used to include a tool called "`cuda-memcheck`" but it has since been deprecated and removed in favor of `compute-sanitizer`. Interestingly, there's an old research paper [Baumann and Gracia, 2013] on using Valgrind with CUDA, called the "CudaGrind" tool. There's even a Github repo for this tool, although it hasn't been edited in 9 years, so I'm not sure it's still valid.

In any case, the basic Linux version of Valgrind Memcheck is still very well supported. The method to use Valgrind for Linux on a CUDA application is simply to run the executable:

```
valgrind a.out
```

If Valgrind is not installed in your Linux environment, you'll need to do something like this:

```
apt install valgrind
```

The start of the Valgrind output is like this:

```
==1143== Memcheck, a memory error detector
==1143== Copyright (C) 2002-2017, and GNU GPL'd, by Julian
Seward et al.
==1143== Using Valgrind-3.18.1 and LibVEX; rerun with -h for
copyright info
==1143== Command: ./a.out
```

As it executes your program, the output from your program will be interleaved with error reports from Valgrind. Hopefully, there won't be any!

The end of the Valgrind execution gives you a nice summary of memory leaks and errors.

```
==1143== HEAP SUMMARY:
==1143==     in use at exit: 12,710,766 bytes in 10,810 blocks
==1143== total heap usage: 15,851 allocs, 5,041 frees, 47,396,077 bytes
==1143==
==1143== LEAK SUMMARY:
==1143==    definitely lost: 0 bytes in 0 blocks
==1143==    indirectly lost: 0 bytes in 0 blocks
==1143==      possibly lost: 30,965 bytes in 199 blocks
==1143==    still reachable: 12,679,801 bytes in 10,611 blocks
==1143==         suppressed: 0 bytes in 0 blocks
==1143== Rerun with --leak-check=full to see details of leaked memory
==1143==
==1143== For lists of detected and suppressed errors, rerun with: -s
==1143== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Running CUDA programs in Valgrind is obviously slower because of the instrumentation, but this is also true of similar tools like `compute-sanitizer`. There is also a problem with "`ioctl`" where you get errors like these from Valgrind (abridged):

```
==1143== Warning: unhandled ioctl 0x30000001 no size/direction hints.
==1143==  This could cause spurious value errors to appear.
==1143==  See README_MISSING_SYSCALL_OR_IOCTL for guidance
```

These warnings are probably indicative of Valgrind having problems understanding the CUDA primitives related to GPU kernel code. Valgrind is fine for checking your host code for C++ memory usage errors, but lacking for device code checking. Hence, `compute-sanitizer` is preferred overall.

Warning-Free Build

Don't ignore compiler warnings! A very good goal for C++ software quality is to get to a warning-free compile. You should think of compiler warnings as doing "static analysis" of your code. To maximize this idea, turn on more warning options, since the warnings are rarely wrong in modern compilers, although some are about harmless things.

Harmless doesn't mean unimportant. And anyway, the so-called "harmless" warnings aren't actually harmless, because if there's too many of them in the compilation output, then the bad bugs won't get seen. Hence, make the effort to fix the minor issues in C++ code that's causing warnings. For example, fix the "unused variable" warnings or "mixing `float` and `double`" type warnings, even though they're rarely a real bug. And yet, sometimes they are! This is why it's powerful to have a warning-free compile.

Tracking compilation warnings. One way to take warning-free compilation to the next level is to actually store and analyze the compiler output. It's like log file analysis in DevOps, only it's not for systems management, but for debugging. On Linux, I typically use this idea:

```
make build |& tee makebuild.txt
```

Here's an actual example from a `Makefile` in an Aussie AI project on Linux:

```
build:
    -@make build2 |& tee makebuild.txt
    -@echo 'See output in makebuild.txt'
```

The `Makefile` uses prefix "`-`" and "`@`" flags, which means that it doesn't echo the command to output, and doesn't stop if one of the steps triggers an error.

When the build has finished, then we have a text file "`makebuild.txt`" which can be viewed for warning messages. To go further, I usually use `grep` to remove some of the common informational messages, to leave only warning messages. Typically, my Linux command looks like:

```
make warnings
```

Here's an example of the "warnings" target in a Makefile for one of my Aussie AI projects:

```
warnings:
    -@cat makebuild.txt | grep -v '^r -' \
    | grep -v '^g++ ' | grep -v '^Compiling' \
    | grep -v '^Making' | grep -v '^ar ' \
    | grep -v '^make\[' | grep -v '^ranlib' \
    | grep -v '^INFO:' | grep -v 'Regressions failed: 0' \
    | grep -v 'Assertions failed: 0' | grep -v SUCCESS \
    |more
```

Note that this uses grep to remove the informational messages from g++, ar, ranlib, and make. And it also removes the unit testing success messages if all tests pass (but not if they fail!). The idea is to show only the bad stuff because log outputs with too many lines get boring far too quickly and then nobody's watching.

One annoying thing about using grep with make is that you get these kind of error messages:

```
make: [annoying] Error 1 (ignored)
```

Here's a way to fix them in a Makefile on Linux:

```
    -@grep tmpnam *.cu *.cpp || true
```

The "true" command is a shell command that never fails. Note that this line uses the double-pipe "||" shell logical-or operator, so it only runs "true" if grep fails. But don't accidentally use a single "|" pipe operator, which would actually be a silent bug!

This idea makes the line calling grep return a non-zero status, and then make is silent.

Finally, your warning-free tracking method should ideally be part of your "nightly builds" that do more extensive analysis than the basic CI/CD acceptance testing.

You should email those warnings to the whole team, at about 2am ideally, because C++ programmers don't deserve any sleep.

Linters for CUDA C++

Linters, or "static analyzers," are tools that examine your source code for errors or stylistic concerns. General advice in regard to using linters for CUDA C++ programming is:

- Use compiler warnings as free linting.
- Use a separate linter build sequence.
- Have two linter paths (one for bugs, one for style).
- Use multiple compilers and linters for extra coverage.
- Automate linting into the nightly build.

Note that we have an active project for a CUDA C++ linter. Find more information about Aussie Lint at https://www.aussieai.com/cuda/projects.

Using gcc as a linter. If you want more warnings, and who doesn't, you can enable more warnings in `gcc` on Linux. You can either do this in your main build by enabling more compiler warnings, or use a separate build path (e.g., choose an inspiring name like: "`make lint`") so that the main build is not inundated with new warnings. The way to do this is via the "`--compiler-options`" command-line option to `nvcc`, which specifies pass-through options for the underlying C++ compiler.

By default, this compiler is `gcc` on Linux and `cl.exe` on Windows.

Since `nvcc` uses source-to-source compilation for the host code, these options will be running on most of your CUDA C++ host code, except for the parts that `gcc` wouldn't understand (e.g., the <<<...>>> kernel launch syntax will be modified before being passed through).

An example command with extra linting power would be:

```
nvcc --compiler-options="-Wall" aussie-test-crashes.cu
```

Some useful `gcc` warning flags include:

- `-Wall` — "all" warnings (well, actually, some).
- `-Wextra` — the "extra" warnings not enabled by "`-Wall`".
- `-Wpedantic` — yet more of the fun ones.

Hence, a longer command is:

```
nvcc --compiler-options="-Wall -Wpedantic -Wextra"
```

You know, I really cannot say that I am a fan of endlessly scrolling warnings from the "pedantic" mode. Maybe, turn that one off, and pick-and-choose from the list of flags in the "pedantic" list. For example, I have used "`-Wpointer-arith`" in projects.

Linting device code

Device code is directly compiled by `nvcc`, rather than via source-to-source compilation, so the device code won't get linted this way. We could try to bypass `nvcc` and use `gcc` directly, such as this:

```
gcc -I/usr/local/cuda/include -Wall myfile.cu
```

But there's at least two problems:

(a) the file suffix "`.cu`" needs to be changed to "`.cpp`" or similar, and

(b) code sequences like "<<<" and "`__global__`" won't be understood by `gcc`.

Hence, you need to have a separate linting build sequence that renames files. You may also need source code changes to wrap kernel launch syntax with `#if` statements, or alternatively, use some fancy `sed` replacement tricks.

Code that is both host and device code will presumably go through both paths, and thus will be linted. This inspires another idea of a linting strategy to get device code covered, too, which is:

- Use a separate linting path.
- Use `nvcc` as the compiler, but with the `gcc` compiler warnings on.
- Mark all the device C++ code as also "`__host__`" code.

But we don't want to change this in our main code base that is processed by `nvcc`. Hence, this requires tricks like a macro that's only enabled in linting mode, or a `sed` trick to add `__host__` wherever there's a `__device__` specifier. This starts getting into murky territory, and I'm going to say it's probably not worth the effort, but maybe it has some value.

Fixing Linter Warnings

Here's some advice about fixing the code to address linter concerns:

- Aim for a warning-free compilation of bug-level messages.
- Don't overdo code changes to fix any stylistic complaints.

Fix the bugs found by warnings (obviously), but as far as the stylistic type warnings are concerned, be picky. I say, aim for code quality and resilience, not code aesthetic perfection.

Warning-free linting. As with the main build, if you're not fixing the less severe linter warnings, turn them off, or have two separate build sequences for the main anti-bug linting versus stylistic linting. You want any newly found serious problems to be visible, not lost in a stream of a hundred other spurious warnings. Hence, high quality code requires achieving a warning-free linting status for the main warnings.

On the other hand, you don't want programmers doing too much "busy work" fixing minor coding style warnings with little practical impact on code reliability. Hence, you might find that your policy of "warning-free linting" needs to suppress some of the pickier warnings. And that'll be a fun meeting to have.

References

1. GNU, Sep 2024 (accessed), *3.8 Options to Request or Suppress Warnings (GCC warning options)*, https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
2. Thomas M. Baumann, Jose Gracia, 3 Oct 2013, *Cudagrind: A Valgrind Extension for CUDA*, https://arxiv.org/abs/1310.0901, https://github.com/dpc-grindland/Cudagrind (Valgrind Memcheck for CUDA C++, but over 10 years old)

8. Error Checking

CUDA Error Checking

Everyone's always known that it's good programming style to check all error return codes. It's extra work, but everyone does it anyway, because it's so important. I've been coding in C++ for years, and I've never seen a `printf` or `fopen` without an `if` statement immediately after it.

CUDA puts all your good intentions to a much stronger test, because literally every CUDA API function can fail, at random times, with any possible error code. One of the issues is that a CUDA kernel can fail asynchronously, and then report its error at the next available opportunity, with timing unrelated to the host code on the CPU.

The solution is that after every CUDA function call, you add an `if` statement. Here's an example for manually checking `cudaMemcpy` calls:

```
errval = cudaMemcpy(device_v, v, sz,
                    cudaMemcpyHostToDevice);
if (errval != cudaSuccess) {
      // CUDA error...
      AUSSIE_ERROR(errval, "cudaMemcpy host-to-device");
}
```

But then your fingers eventually get tired from too many keystrokes, and you start making copy-paste errors, too. Maybe you should try an AI copilot?

Alternatively, let's define a macro for that.

The main styles I've seen for CUDA error checking macros are either:

- Wrap every CUDA function call in a macro, or
- Test after CUDA function calls.

There are pros and cons of each approach, but they both suffer from a major limitation: manual code changes.

I have suggestions of two ways to automate it:

- Recursive preprocessor macro intercepts.
- Macro intercepts to debug wrapper functions.

These are all explained in detail further below.

CUDA Error Check Macros

The use of a CUDA error check macro works as a replacement for manual error checking. Here's an example usage of a macro:

```
AUSSIE_CUDA_ERRORCHECK(cudaDeviceSynchronize());
```

Note that you can also wrap the assignment of the error code to a variable for further analysis.

```
cudaError_t errval = cudaSuccess;
// ...
AUSSIE_CUDA_ERRORCHECK( errval = cudaDeviceSynchronize() );
```

Here's one version of what the macro can look like:

```
#define AUSSIE_CUDA_ERRORCHECK(codeexpression) \
     do {  \
       cudaError_t err = codeexpression ;  \
       if (err != cudaSuccess) { \
          fprintf(stderr, \
            "CUDA ERROR: %d (%s) in %s at %s:%d\n", \
                 (int)err, cudaGetErrorString(err), \
                     __func__, __FILE__, __LINE__); \
       } \
     } while(0)
```

This macro definition uses the `do..while(0)` common C++ idiom to make the macro fully like a statement. This style avoids some problems with semicolons that would arise if you just use curly braces, so don't add a semicolon at the end of this macro, or all that work is in vain. This also avoids a serious "dangling-`else`" bug if you only used the `if` statement alone.

But don't forget the extra inside pair of parentheses in the calls:

```
AUSSIE_CUDA_ERRORCHECK(cudaDeviceSynchronize ); // Wrong!
AUSSIE_CUDA_ERRORCHECK(errval = cudaDeviceSynchronize);
```

Here's another more elegant method of doing the macros in combination with an `inline` function in a header file:

```
#define AUSSIE_CUDA_ERRORCHECK2(codeexpression) \
    aussie_cuda_check_function((codeexpression), \
            __func__, __FILE__, __LINE__)

inline void aussie_cuda_check_function(cudaError_t err,
    const char *func, const char *fname, int lnum)
{
   if (err != cudaSuccess) {
        fprintf(stderr,
            "CUDA ERROR: %d (%s) in %s at %s:%d\n",
            (int)err, cudaGetErrorString(err),
            func /*__func__*/,
            fname /*__FILE__*/,
            lnum /*__LINE__*/ );
   }
}
```

Checking After CUDA Calls

The alternative method is to check after the CUDA API calls. The macro includes a call to `cudaGetLastError` or `cudaPeekAtLastError`. The use of the macro looks like:

```
AUSSIE_CUDA_CHECKAFTER();  // calls cudaGetLastError
```

Here is one way to define it:

```
#define AUSSIE_CUDA_CHECKAFTER() \
    do {  \
       cudaError_t err = cudaGetLastError();  \
       if (err != cudaSuccess) { \
            fprintf(stderr, \
              "CUDA ERROR: %d (%s) in %s at %s:%d\n", \
              (int)err, cudaGetErrorString(err), \
              __func__, __FILE__, __LINE__); \
       } \
     } while(0)
```

An equivalent method that is perhaps clearer is to do one of these methods instead after a kernel launch:

```
AUSSIE_CUDA_ERRORCHECK2( cudaPeekAtLastError() );
AUSSIE_CUDA_ERRORCHECK2( cudaGetLastError() );
```

Kernel launch special case. I've got some better ideas for the error check macros, but there's one situation where you definitely must use the "check after" style: kernel launches.

The <<<...>>> kernel invocation syntax does not return a status code, so there's nothing to check. Also, there's also an obscure situation whereby synchronous kernel launch errors (e.g., threads-per-block size more than 1024, or too much shared memory requested) can get missed, unless they are immediately checked for using either `cudaGetLastError` or `cudaPeekAtLastError`.

Hence, the best solution for kernel launch error detection, at least in non-production mode, is something like this:

```
mykernel <<< blocks, threads >>> (v, n);
AUSSIE_CUDA_ERRORCHECK2( cudaPeekAtLastError() );
```

Alternatively, maybe one day in the distant future this will work:

```
err = mykernel <<<blocks,threads>>>(v, n); // FAILS!
AUSSIE_CUDA_ERRORCHECK2(err);
```

Recursive Macro Error Checks

C++ allows macros to be recursive in the sense that they can use their own name. It's not actually "recursive" and is actually limited to a once-only expansion, rather than an infinitely recursive expansion. This feature is a longstanding feature of C and C++ languages since they were created, so you can rely upon it. For example, these would be harmless:

```
#define cudaMemset(a,b,c)    cudaMemset(a,b,c)
#define cudaMemcpy(a,b,c,d) cudaMemcpy(a,b,c,d)
```

The idea is to automatically add the error check macros:

```
#define cudaMemset(a,b,c) \
    AUSSIE_CUDA_ERRORCHECK(cudaMemset(a,b,c))
#define cudaMemcpy(a,b,c,d) \
    AUSSIE_CUDA_ERRORCHECK(cudaMemcpy(a,b,c,d))
```

But that doesn't quite work, when used with this type of call:

```
errval = cudaMemcpy(....);
```

The `do...while(0)` trick expands out to give a compilation syntax error:

```
errval = do { ... // etc.
```

Similarly, the version with a combined macro and `inline` function also gets a different type of compilation error:

```
errval = aussie_cuda_check_function(....)
```

The problem is that the return type of the `inline` function is `void`. Hence, we'd need to go back and fix any code that uses the return value of `cudaMemcpy` or `cudaMemset`, which would be a good job for a coding copilot, if only I didn't have so many trust issues.

Instead, we can just fix the return type to be `cudaError_t` and use a pass-through of the error code:

```
#define AUSSIE_CUDA_ERRORCHECK3(codeexpression) \
  aussie_cuda_check_function2((codeexpression), \
         __func__, __FILE__, __LINE__)

inline cudaError_t aussie_cuda_check_function2(
    cudaError_t err,
    const char *func, const char *fname, int lnum)
{
   if (err != cudaSuccess) {
        fprintf(stderr,
            "CUDA ERROR: %d (%s) in %s at %s:%d\n",
            (int)err, cudaGetErrorString(err),
            func /*__func__*/,
            fname /*__FILE__*/,
            lnum /*__LINE__*/ );
   }
   return err;  // pass through!
}
```

And we really should add a ridiculous number of round brackets around the macro parameters, and also use `#undef` for total macro safety:

```
#undef cudaMemset  // safety
#undef cudaMemcpy
#define cudaMemset(a,b,c) \
 (AUSSIE_CUDA_ERRORCHECK3(cudaMemset((a),(b),(c))))
#define cudaMemcpy(a,b,c,d) \
 (AUSSIE_CUDA_ERRORCHECK3(cudaMemcpy((a),(b),(c),(d))))
```

Voila!

Now we have a set of macros that automatically adds CUDA return code error checking around all calls to `cudaMemcpy` and `cudaMemset`. And it should work irrespective of whether their returned values are used or not in the calls.

To use them properly, we just need to `#include` a header file near the top of every CUDA C++ source file. But it has to be after any CUDA toolkit header files like "`cuda_runtime.h`" because those system header files have prototype declarations of functions like `cudaMemcpy` that our tricky macros will break.

Now we only have to add similar recursive macros for all 1,657 of the CUDA Runtime API functions. No, relax, I'm just kidding. The number is 50 according to this command:

```
grep cudaError_t /usr/local/cuda/include/cuda_runtime.h| wc -l
```

Yeah, I could probably have used "`grep -c`" but I just don't want to, and you can't make me.

Macro Intercepted Debug Wrapper Functions

Is there any way you can level up? We've already auto-added the error checking macros around all the CUDA Runtime API calls. Can we do better?

Of course, we can!

One extension is to build debug wrapper function versions for the main API calls. These functions can then perform more extensive error self-checking than is performed within the CUDA Runtime.

```
#undef cudaMemcpy
cudaError_t aussie_cudaMemcpy_wrapper(void *destp,
  const void *srcp, size_t sz, enum cudaMemcpyKind mode)
{
    cudaError_t err =
    AUSSIE_CUDA_ERRORCHECK3(cudaMemcpy(
                        destp,srcp,sz,mode));
    return err;
}

#define cudaMemcpy(a,b,c,d) \
   aussie_cudaMemcpy_wrapper(a,b,c,d)  // Intercept!
```

Note that the `#undef` is really important here, and must be before the wrapper function body. If we're not careful, our wrapper function can wrap itself, and become infinitely recursive.

The above example doesn't do any extra error checking, other than what we've already put into the CUDA error checking macro (i.e., `AUSSIE_CUDA_ERRORCHECK3`), which checks for the `cudaSuccess` return code. However, we could add extra self-checking code for common errors that arise from `cudaMemcpy` copy-pasting:

- Destination or source pointers are null
- Destination or source pointers are the wrong address scopes
- Destination pointer equals source pointer

The CUDA Runtime already finds a lot of those errors, and `compute-sanitizer` would find even more. However, we could go further with our analysis. For example, some more extensive error checks possible could be:

- Pointer allocated by `cudaMalloc` but never copied by `cudaMemcpy`.
- Pointer allocated by `cudaMalloc` only partially copied by `cudaMemcpy`.
- `cudaMemcpy` size argument is zero or negative (after conversion to `size_t`).
- `cudaMemset` arguments appear to be in reverse order.

The possible error checks from this type of API interception are discussed further in the full section on debug wrapper functions.

Limitations of Macro Interception

Two of these methods rely on preprocessor macro interception to auto-wrap the calls with debug checks. Unfortunately, macro interception isn't a perfect solution, and some of the problems that macros may have include:

- No way to auto-intercept the <<<...>>> kernel launch syntax.
- Problematic for device code (e.g., CUDA Dynamic Parallelism, `fprintf` not available).
- Interception of `new` and `delete` operators is only possible at link-time, and even this trick won't work for device code.
- Namespace-scoped calls fail: e.g., `cuda::cudaMemcpy(...)` or `std::malloc(...)`
- Use of CUDA API names as function pointers won't work.
- Non-standard calling syntax: e.g., parentheses around the function name.

Much better than macro interception would be a way to link to a debug version of the CUDA Runtime library. If only it were open source code! Many more complex error checks are possible than are performed, and this would significantly improve the timeframe to detect many types of coding errors.

Alternatively, tools such as `compute-sanitizer` could link with a debug runtime library version that contained a more extensive set of checks. Maybe this is feasible to do via the callback methods in the Compute Sanitizer API, which is an official part of the CUDA Toolkit.

Reporting and Handling CUDA Errors

What should an error checking macro do on failures? Some of the many options include:

- Print an error message
- Print the error code number and its name with `cudaGetErrorString`
- Give source code context information
- Exit the program (or not?)

That's not the full list, and some more advanced ideas for production-grade error handling include:

- Throw an exception and hope someone's listening.
- Full stack trace (e.g., `std::backtrace` in C++23).
- Report a full error context for supportability in the wild.
- Log information to a file, not just to `stderr`.
- Try to recover if it's a non-sticky error.
- Gracefully crash if it's a sticky error.
- Try to localize if it's a current failure or from a prior kernel launch.
- Call a debug breakpoint function to help with interactive debugging.
- Abort the program to generate a "`core`" file.

A key aspect of reporting the error context is the CUDA C++ statements that triggered the issue.

The basics of error context are these macros:

- `__func__`
- `__FILE__`
- `__LINE__`

I don't know why one is lower case and two are upper case, but it's called international standardization. That's an example of what makes C++ programming so fun.

However, I have to say that I think these source code context macros are on their way out. Once reporting the full stack trace in C++23 with `std::backtrace` is widespread, why would we need those macros? Also gone would be various preprocessor macro tricks that only exist in order to report the source code context. Instead, use an `inline` function and `std::backtrace`.

More advanced error context that can help with supportability includes things like:

- Date and time of error.
- User query that triggered the failure.
- Random number seed (for reproducibility of AI errors).
- Full stack trace (if available)

I feel like there should be an LLM for this. Maybe I'll go look on Hugging Face.

Limitations of CUDA Error Checking

Some problem areas include:

- No return code for kernel launches.
- Sticky errors.
- Some odd idiosyncrasies in the CUDA Runtime API (are they bugs?)
- Not all types of bugs are raised as runtime errors.
- Limited possibilities in device code

But I have to say that the really major limitation is this:

Remembering to add it every time!

I've given a few suggestions for auto-fixing that issue above, but they're far from perfect. Maybe the CUDA Runtime API needs a callback mechanism, or some other method whereby programmers can ensure that they never miss an error return.

Error checking in device code. It used to be that you only needed this error checking in the host code, because none of the runtime APIs worked in device code. And then someone at NVIDIA decided to change that with nested kernel launches, and then someone else in marketing decided to give it a trendy name and a three-letter acronym: CUDA Dynamic Parallelism (CDP).

The downside is that the C++ features are much more limited in device code. For example, you can't easily use global variables or write to files. You can't even use `fprintf`, for heaven's sake, so you can only print to `stdout`.

Which is great news, because it means you get to have the fun of defining an error checking macro all over again!

9. Sticky Errors

What are Sticky Errors?

There is a subset of the error codes that are called "sticky," which is another way of saying "really bad." They're called "sticky" because they get stuck in the CUDA error code, and cannot be cleared (not even by `cudaGetLastError`). Even worse, they stop any other CUDA API from working, so you can't launch another kernel, and functions like `cudaMemcpy` just fail immediately.

The list of sticky errors on the GPU is short, but includes the worst kinds of error:

- Invalid address — `cudaErrorIllegalAddress` 700.
- Illegal instruction — `cudaErrorIllegalInstruction` 715.
- Kernel time-out — `cudaErrorLaunchFailure` 719, but later.
- Misaligned access — `cudaErrorMisalignedAddress` 716.
- Null dereference — `cudaErrorLaunchFailure` (later).

Note that some of these sticky errors will return `cudaErrorLaunchFailure`, but not as synchronous errors at the time of the kernel launch. They are triggered later asynchronously by kernel code execution and can thus occur much later, at any point between kernel launch and the first synchronization after the failure. The name of the error code is somewhat misleading!

If you're wondering where are "segmentation fault" or "core dumped" on this list, that's "CPU thinking" and you really need to get your head into GPU world. The problematic coding errors that trigger a segfault in C++, when run on a GPU, will cause some of the above sticky errors with a `cudaErrorLaunchFailure` error code, or in some cases, won't trigger a CUDA runtime error at all in device code (except they can be found by `compute-sanitizer`).

There's no easy recovery from sticky errors, as the GPU won't accept further work from the CPU. The rest of your host code will keep running on the CPU, but anything GPU-related will fail with an error code from CUDA sent to the host.

The good news is that you can intercept all these error codes with your glorious "CUDA error check" macro, but the bad news is there's no way to fix it. They're called "sticky" errors because you're stuck!

Detecting Sticky Errors

How do you detect a sticky error? Unfortunately, you cannot just check the numeric code value for a few specific `enum` values, because some codes can be both sticky and non-sticky. For example, `cudaErrorLaunchFailure` could be a synchronous launch failure from too many threads-per-block (non-sticky) or an asynchronous error from a null pointer dereference in a kernel (sticky).

Since `cudaGetLastError` returns the current error, but also clears the error flag for the next call, you might think this would work:

```
bool aussie_is_sticky_error_FAILS(bool warn)  // Buggy!
{
  cudaError_t err = cudaGetLastError(); // Clear prior error
  err = cudaGetLastError();  // Twice
  if (err != cudaSuccess) { // Sticky error?
    if (warn) fprintf(stderr, "CUDA STICKY ERROR: %d %s\n",
                  (int)err, cudaGetErrorName(err));
    return true;
  }
  return false;  // not sticky
}
```

Actually, this won't report a sticky error. The function `cudaGetLastError` will return `cudaSuccess` the second time, even if you're in a sticky error state. It seems that sticky errors are not sticky for this CUDA Runtime API function.

Unfortunately, the only reliable way to detect a sticky error is to issue a dummy call to the CUDA runtime, like using `cudaMemcpy` or `cudaMalloc`, such as this:

```
bool aussie_is_sticky_error(bool warn)
{
  // Do a dummy cudaMalloc to see if it's a sticky error...
  void *vdummy = NULL;
  cudaError_t err = cudaGetLastError(); // Clear prior
  err = cudaMalloc(&vdummy, 1);  // Sticky error?
  if (err != cudaSuccess) {
     if (warn) fprintf(stderr, "CUDA STICKY ERROR: %d %s\n",
           (int)err, cudaGetErrorName(err));
     return true;  // Yes, sticky..
  }
  return false;  // not sticky
}
```

Probably it would be better to use a dummy `cudaMemcpy` call than the above, which needlessly fragments device allocated memory.

Actually, here's a cleaner way to detect sticky errors suggested on the NVIDIA Forums: call `cudaDeviceSynchronize` twice, although this has the inefficiency that it causes synchronization, which would invalidate any gain if you're using data transfer overlapping optimizations.

```
bool aussie_is_sticky_error_SYNCHRONIZED(bool warn)
{
    // Call cudaDeviceSynchronize twice to test stickiness
    cudaError_t err = cudaDeviceSynchronize(); // First call
    if (err != cudaSuccess) {
        err = cudaGetLastError();  // Clear it
        err = cudaDeviceSynchronize();  // Second call...
        if (err != cudaSuccess) {  // Sticky error?
            if (warn)
               fprintf(stderr, "CUDA STICKY ERROR: %d %s\n",
                     (int)err, cudaGetErrorName(err));
            return true;  // Yes, sticky..
        }
    }
    return false;  // not sticky
}
```

What Causes Sticky Errors?

Which kernel device errors cause sticky errors? I wrote some dummy kernels to trigger crashing code.

```
__global__ void null_deref_local(float *f, int n)
{
    int *ptr = NULL;
    *ptr = 1;
}

__global__ void array_underflow_write(float *f, int n)
{
    f[-1] = 0.0;
}

__global__ void array_overflow_write(float *f, int n)
{
    f[n + 1] = 0.0;
}

__global__ void array_underflow_read(float *f, int n)
{
    volatile int x = f[-1];
    x = x;
}
```

```
__global__ void array_overflow_read(float *f, int n)
{
    volatile int x = f[n + 1];
    x = x;
}

__global__ void cudamalloc_uninit_read(float *f, int n)
{
    volatile int x = f[3];
    x = x;
}

__global__ void do_nothing_kernel(float *f, int n)
{
    volatile int x = 0;
    x = x;
}
```

The idea with "`volatile`" is to prevent the CUDA compiler from optimizing my bad code away. Maybe I'm giving it too much credit, because it didn't even remove my blatantly obvious null dereference.

Additionally, I used a test harness to launch the kernels:

```
 fnptr<<<1,1>>>(dest_v, n);   // Launch kernel

// Check for CUDA synchronous launch errors
err = cudaPeekAtLastError();  // Any error?
if (err != cudaSuccess) {
    // ... etc... (did not occur)
}
err = cudaDeviceSynchronize();  // Wait for completion
if (err == cudaSuccess) {
    fprintf(stderr, "%s: no error\n", name);
}
else {
    if (aussie_is_sticky_error(false)) {
      fprintf(stderr, "%s: CUDA sticky error: %d %s\n",
           name,
           (int)err,
           cudaGetErrorName(err));
    }
    else {
      fprintf(stderr, "%s: CUDA non-sticky error: %d %s\n",
           name,
           (int)err,
           cudaGetErrorName(err));
    }
}
```

Here's the results, abridged from multiple invocations (because of the sticky one!):

```
Null dereference: CUDA sticky error: 719 cudaErrorLaunchFailure
Array_underflow write: no error
Array underflow read: no error
cudaMalloc uninitialized read: no error
Array_overflow write: no error
Array_overflow read: no error
```

Just between you and me, I did initially have a "`cudaMalloc` uninitialized write" and it didn't fail. But then I removed it.

Compute Sanitizer did a lot better at finding problems than the basic CUDA runtime. For example, here's the output from the "array underflow write" kernel:

```
========= Invalid __global__ write of size 4 bytes
=========     at 0x20 in array_underflow_write(float *, int)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x7b5cf32001fc is out of bounds
=========     and is 4 bytes before the nearest allocation at
0x7b5cf3200200 of size 400 bytes
=========     Saved host backtrace up to driver entry point at
kernel launch time
```

Overall, these results constitute a non-peer reviewed, statistical sample of one, based on one GPU on one particular platform at one time of year in one country. Totally guaranteed results!

Sticky Error Recovery

The main way to "recover" from this situation is to shut down the whole application, such as by calling `exit` or `abort`. You can print a nice, helpful message to your users for good supportability, but that's about all you can do. Your host code looks like this:

```
if (aussie_is_sticky_error(false)) {
  fprintf(stderr, "I apologize for my very existence!\n");
  abort();
}
```

If you aspire to being a perfect programmer, you can close all your open files to flush the buffers, and free all your allocated memory, but the operating system will do that anyway.

In the category of facts you didn't want to know, this situation is analogous to the old-school Unix crashes in CPU code that cause core dumps, such as segmentation faults and illegal instructions. You can try to register a signal handler function to intercept the `SIGSEGV` or `SIGILL` signals, and then try to return from your signal handler, because you want to keep going. Unfortunately, this fails because the CPU will just re-raise the same signal, so it spins, and you can't recover. Just like GPU sticky errors, the best you can do is register a signal handler that prints a grovelling message and then aborts.

Multi-Process Fix for Sticky Errors

This is in the category of fixes that don't really fix it: There is a way to try a more extensive recovery of an application that fails with a CUDA sticky error. The solution is: abort your process, and try again. This idea works if:

- Parent process is the main controlling application on the CPU.
- Parent process launches a sub-process (e.g., `fork` and `exec`).
- Child sub-process launches the CUDA kernel.

If the child process detects a sticky error in a CUDA error code, then the child process can shut itself down, telling the parent that it failed, before it shuts itself down. Then the parent can detect the child's failure status, and try again by launching a new child process to re-do this entire kernel.

The advantage of this method is that a single failed kernel doesn't kill your entire application, which can be resilient to a transient failure on the GPU. The downside is that it has to re-do the entire kernel, with no partial results available.

The extra work you have to do for this includes:

- Parent: Fiddly file descriptor work in `fork-exec` sequences (like it's 1990 all over again).
- Child: Detect sticky errors in your CUDA error check macros (making them even more spectacular).
- Child: Report success or failure status back to the parent process from the child process.
- Parent: Check the child sub-process return status and re-try (but not infinitely).

It's a certain kind of fun.

10. GPU Kernel Debugging

Kernel Debugging Techniques

The kernels running on the GPU are the most important C++ code you'll ever write, but also the most difficult to debug. When you're focused on getting the most speed out of the silicon, it's far too easy to introduce an insidious bug.

We've already examined some of the main techniques that look at the kernel from the "outside" in the host code:

- Error checking CUDA Runtime API calls in the host code.
- Managing sticky errors to the extent possible.

Tools are also something that should be top of the list. Some of the NVIDIA debugging tools are amazing in terms of resolving device issues:

- `cuda-gdb` for interactive debugging.
- `compute-sanitizer` and all four of its sub-tools.

There are also two important builtin methods that work in the kernel C++ code:

- `printf` (but not `fprintf`)
- `assert`

Technically, these are part of CUDA Runtime and defined without an explicitly included header file, but sometimes you may need to include `<stdio.h>`/ `<iostream>` or `<assert.h>`.

Do not underestimate what you can achieve with just these two methods! The main strategies are:

- Add tracing with `printf` and use `cudaDeviceSynchronize` to ensure you see the output messages (beware the dreaded kernel buffer overflow!).
- Add lots of `assert` calls peppered throughout (afterwards, you can remove them or leave them, at your discretion).

Some additional techniques can be helpful in finding the cause of a failure:

- Serialize kernel launches (e.g., set variable `CUDA_LAUNCH_BLOCKING`, or use the command "`set cuda launch_blocking on`" inside `cuda-gdb`, or via added calls to `cudaDeviceSynchronize` after launches).
- Launch a single thread or a single warp, if your kernel uses grid-stride loops.
- Add `cudaSynchronizeDevice` to serialize, and also to flush the kernel output buffers (`printf` and `assert`).

Triggering Bugs Earlier

Many kernel bugs can be found using the techniques already mentioned. The above methods are very powerful, but they can be limited in some less common situations:

- Intermittent bugs — hard to reproduce bugs.
- Silent bugs — how would you even know?

You can't really find a bug with `cuda-gdb` or the `compute-sanitizer` memory checker if you can't reproduce the failure. On the other hand, an intermittent failure might be a race condition or other synchronization error, so you probably should run `racecheck` and `synccheck`.

Silent bugs are even worse, because you don't know they exist. I mean, they're not really a problem, because nobody's logged a ticket to fix it, but you just know it'll happen in production at the biggest customer site in the middle of Shark Week.

How do you shake out more bugs? Here are some thoughts:

- Set the `CUDA_ENABLE_COREDUMP_ON_EXCEPTION` variable (because the code won't core dump on some GPU errors, but can quietly continue).
- Add more assertions on arithmetic operations in device code (e.g., more tests for floating-point `NaN` or negative zero).
- Auto-wrap CUDA API calls in host code with error checking for *all* calls.
- Fast self-checks for simple kernel launch mistakes via asserting simple checks (e.g., `nthreads%32==0` and `nthreads<=1024`).
- Arithmetic overflow or underflow is a very silent bug for both integers and floating-point (e.g., check `unsigned` integers aren't so high they'd be negative if converted to `int`).
- Index safety tests in kernels hide bugs (use `printf` messages or assertions instead, assuming you're managing sizes to avoid extra wasted threads).
- Add a unit test kernel to test `__device__` utility functions brick-by-brick.

There are also some changes to the host code that can help detect several common types of kernel bugs:

- Add self-testing code with more complex sanity checks for kernel launches.
- Consider debug wrapper functions with extra self-testing.
- Add more function parameter validation

With all of these things, any extra runtime testing code requires a shipping policy choice: remove it for production, leave it in for production, only leave it in for beta customers, leave in only the fast checks, and so on.

If you're still struggling with an unsolvable bug, here are a few "hail Mary" passes into the endzone:

- Add a call to `cudaGetLastError` immediately after kernel launches before having any call to `cudaDeviceSynchronize` or other implicit synchronizations (otherwise, synchronous kernel launch failures, such as more than 1024 threads or too much shared memory, may be silent; admittedly, you can see the kernel's not running in a debugger).
- You can run `valgrind` on CUDA C++ executables, though it's not any better than `compute-sanitizer`; there may be a few rare things.
- Review the latest code changes; it's often just a basic mistake hidden by "code blindness" (e.g., check your "`.x`" and "`.y`" attributes).
- Mixing up the indices of square matrices is a silent, nasty bug in your algorithm that's hard to detect with most debugging approaches.
- Add more calls to synchronization primitives like `__syncthreads` (this may help prove it's a synchronization error, but won't help you find it).
- Add a `cudaMemset` call after every `cudaMalloc` (and variants) to see if initializing the memory fixes it (admittedly, tools should find this anyway).
- Similarly, try `memset` after `malloc` or `new`, or change to `calloc` (note that there's no `cudaCalloc`!).

And some other practical housekeeping tips can sometimes help with detecting new bugs as soon as they enter the source code and planning ahead for future failures:

- Examine compiler warnings and have a "warning-free build" policy.
- Have a separate "`make lint`" build path with more warnings enabled.
- Keep track of random number generator seeds for reproducibility.
- Add some portability sanity checks, such as confirming the size of data types: `static_assert(sizeof(float)==4);`

I guarantee that last one will save your bacon one day!

De-Slugging Kernels

Your code has just slowed down and you don't know why? Well, first thing is to run one of the various CUDA profiling tools.

Some ideas for slugs in your code include:

- You left self-testing code in the source when you were trying to fix a bug!
- Logic around `cudaSetDevice` is broken, and the code is now reduced to only running on one GPU.
- Launching too few blocks, so each thread is doing a lot of work.
- The "`-O`" or "`--dopt`" optimization flag was removed or changed.
- Too much synchronization with `cudaDeviceSynchronize`.
- Environment variable `CUDA_LAUNCH_BLOCKING` is enabled.
- New control flow paths caused serious branch divergence in threads.
- Your grid-stride loop has "`i++`" instead of "`i+=stride`" and every thread is computing every element (endless redundant computation).
- The build process lost the "`-DNDEBUG`" flag and assertions are live again.
- You're running on a non-NVIDIA GPU for some strange reason.

Some general areas of sluggish execution include:

- Non-coalesced memory access patterns are slower.
- Thread divergence (warp or branch divergence).
- Implied synchronization in various CUDA Runtime APIs (on host).
- Non-aligned memory accesses are slower (aim for 128-byte alignment).
- Shared memory contention ("bank conflicts").
- Nested kernels can balloon runtime cost.
- `cudaMemcpy` with non-pinned host memory (causes paging).
- Register spills (and "register pressure").
- Instruction locality issues (instruction cache misses).

That's enough for here, but CUDA C++ optimization can be the next book.

11. Basic CUDA C++ Bugs

Common Bugs in CUDA C++

Some of the main classes of bugs include:

- General C/C++ types of bugs (many!)
- Synchronization errors
- Memory errors and other resource problems

An empirical study of CUDA bugs found that "general" errors in C++ are the most common cause of GPU kernel coding errors, moreso than the CUDA-specific capabilities (Wu et.al, 2019). C++ is a difficult language to get right!

Dual programming mode errors in the control of device versus host code include:

- Kernel not defined as `__global__` or `__device__` (basic mistake)
- Device kernel code calling a host-only function.

Memory access errors. Memory-related errors can include:

- Mixing `malloc/free` with `cudaMalloc/cudaFree` addresses
- Uninitialized device global memory
- Address alignment errors
- Array bounds access errors in device memory — often arising from incorrect computations involving thread index and dimensions.

Synchronization errors. Parallel programming can have a variety of errors in the synchronization of multiple threads. Some examples include:

- Host code not waiting for asynchronous device kernel to complete
- Race conditions
- Barrier-related issues
- Early device call reset

Novice Kernel Launch Mistakes

There are many common CUDA idioms and they exist for a reason. Break the pattern, and you might trigger a bug or a slug. There are problems including.

- Not enough threads
- Not enough blocks
- Not enough threads per block
- Too many blocks (each with threads)
- Vector underflows (silently)
- Vector buffer overflows (gasp!)

All these combinations can make your head spin! Let's examine some of these.

Run multiple threads. This is the most beginner issue when you're creating your first CUDA program. Using this style is not really a bug, but more of a slug:

```
mykernel<<<1,1>>>(dv,n);  // SEQUENTIAL!
```

It's not parallel programming if you're only running one thread!

Kernel launch syntax. Here's another simple bug in a kernel launch:

```
mykernel<<1,32>>(dv,n);
```

We've fixed the number of threads to be 32, so that's no longer the problem. To help you out, here's the compiler error from `nvcc`:

```
error: expression must have integral or unscoped enum type
      mykernel<<1,1>>(dv, n);
      ^
```

Well, that's not much help. It's hard to see the bug. The error is "<<" should be "<<<" (three less-than characters). Unfortunately, the compilation error for that is not very clear, because C++ thinks the "<<" is the bitwise left shift operator that works on integers, which is why it wants an "integral" or "`enum`" value.

Wrong Block and Thread Computations

The first part of launching a GPU kernel is to work out how many blocks and threads we need. This is called calculating the "grid dimensions."

There are many ways to go wrong:

- Threads-per-block should be a multiple of 32
- Too few blocks
- Too many blocks

Threads-per-Block Multiple of 32

The number of threads per block (aka the "block size") should be a multiple of the warp size, which is 32 threads. Hence, it can be as low as 32, but commonly recommended block sizes in real-world kernels are often 256 or 512. The maximum permitted by CUDA is 1024 threads per block. This is not good:

```
int n = 54;
mykernel<<<2, 27>>>(dv, n);  // BAD
```

This might work, but it's inefficient, and offends the sensibilities of any experienced CUDA programmer, for reasons discussed below. But first, note that this is worse:

```
mykernel<<<54, 1>>>(dv, n);  // BAD
```

If the threads per block is not 32, or a multiple of 32, there will be odd threads in a warp that aren't properly utilized (or might be doing the wrong thing). The reason is that CUDA allows threads in "warps" that contain exactly 32 threads. With the threads per block of 27, there were 5 extra threads, and for 1 thread per block, there were 31 wasted threads. So, instead, you want something more like this:

```
int n = 64;
mykernel<<<2,32>>>(dv,n);  // BETTER
```

Or you can do this:

```
mykernel<<<1,64>>>(dv,n);  // BETTER
```

CUDA can only schedule 32 threads (a warp) at a time, and if you schedule fewer, other threads in the warp still run (a bug or a slug), or are unavailable to run (a slug).

Too Few Blocks

You don't always know the incoming size `N` of your vector data structure (well, actually, you often do in AI engines, because they have static dimensions, but anyway). Let's try to generalize our computation of how many blocks with each having a fixed number of threads. There's a few basic points:

- Each block has the same number of threads (i.e., the threads-per-block)
- You can't run half a block (all threads run, even if you don't need them).

In our block calculations, we have a Goldilocks problem: it's easy to get too many or too few, when we need it to be exactly right. Let's generalize our computations:

```
int n = 33;
...
int threads_per_block = 32;   // or 64 or 256...
int blocks = n / threads_per_block;  // BUGGY!
mykernel<<<blocks,threads_per_block>>>(dv,n);
```

Does this work? No, because the "/" operator is integer division of "`33/32`", which truncates to 1, and we can't get fractions of a block. The result is:

```
threads_per_block == 32
blocks == 1
```

We're only running 32 threads, but our vector has 33 elements. Hence, if each kernel is processing one vector element, we've missed one of the extra elements. This is a code logic bug. It won't crash CUDA, and nobody's going to whisper in your ear that the end of your vector didn't get processed.

Now, maybe the kernel code has a loop to handle the extra vector elements, which would fix the code logic bug, but it's still a slug. But I'm jumping ahead by mentioning a loop, when your kernel probably doesn't even have an `if` statement yet. It would be better just to use better block size computations.

Too Many Blocks

The previous section had too few blocks, so let's add an extra block:

```
int threads_per_block = 32;
int blocks = (n + threads_per_block)/threads_per_block; // BUG!
```

This is actually the same as:

```
int threads_per_block = 32;
int blocks = (n / threads_per_block) + 1;  // BUGGY!
```

This seems to work better because "`(33+32)/32`" or "`(33/32)+1`" both evaluate to 2 blocks, which is what we want. However, we want the computation to work for all values of `n`, and it's actually wrong if `n==32` because it needlessly uses 2 blocks when 1 block of 32 threads is enough. We have a whole extra block doing nothing, or maybe crashing stuff. You'd think the GPU would be thrilled by an extra block, but not so much. The trick is to use a common CUDA idiom:

```
// CORRECT!
int blocks = (n + threads_per_block - 1) / threads_per_block;
```

The subtraction of 1 in the numerator fixes it using the power of mathematics. For `n==32`, we get 1 and for `n==33` there are correctly 2 blocks. It also works correctly for `n==63` or `n==64`, although I feel like I should add a unit test.

Odd Vector Sizes. All of this fuss about the number 32 being special because it's the warp size might suggest something: all your data should be in size 32 arrays (or matrices or tensors). It's hard to fix it if `N` is a prime number or other oddity.

That's why a lot of the AI engines have parameters and "hidden dimensions" and "context window sizes" that are a multiple of 32 (e.g., 4096 or 32k or whatever). You won't see this in ChatGPT's source code:

```
float mytensor[27][1531][17];   // Weird
```

Better would be:

```
float mytensor[4096][256][32768]; // The power of 32!
```

Now you know: it's all CUDA's fault. Which is more true than we like to admit, since all of this AI frenzy might not have happened without CUDA's blazing speed.

Wrong Kernel Index Calculations

Okay, so now that our computations of blocks and threads are sorted, let's now actually show a very simple kernel. All good kernels start with the special "`__global__`" keyword, which means that it runs on the GPU. Here's a vector clearing to zero kernel, which is a great piece of work if we pretend we don't know about functions like `memset` or `cudaMemset`.

Anyway, here's the kernel code for the GPU to run lots of times in parallel:

```
__global__ void aussie_clear_buggy1(float* v, int n)
{
    int id = threadIdx.x;  // BUG!
    v[id] = 0.0;  // Clear one vector element..
}
```

And the host code that runs on the CPU, only once, looks like this:

```
int n = 33;
...
int threads_per_block = 32;
int blocks = ( n + threads_per_block - 1)/threads_per_block;
aussie_clear_buggy1<<<blocks, threads_per_block>>>(dv, n);
```

I've left out a lot of the details on the CPU side before and after the fancy <<<...>>> syntax (it's called "triple chevron" syntax). The host code also has to allocate memory (twice) and set up the vectors and copy them from the CPU to the GPU, and back again. But for now, let's just look at the blocks and threads.

Is there a bug? Umm, well, there is the word "BUG" in the comments, and also in the function name, so maybe I gave it away a little bit. But why is it a bug? First, let's look at how many blocks and threads-per-block we've got:

```
threads_per_block == 32
blocks == 2
```

We've now got 32*2=64 total threads, each setting one element in a vector of size 33. Aha! Obviously, it's a buffer overflow inside the kernel when we access `v[id]`!

Nice try, but no cigar. Actually, the problem is this statement:

```
int id = threadIdx.x;  // BUG!
```

What is `threadIdx.x` when it's at home (on a GPU)? Well, the variable "`threadIdx`" stands for "thread index" and the "`.x`" part means the offset in the x-dimension (i.e., 1D vectors). There's also a "`.y`" for 2D (matrices) and "`.z`" for 3D (tensors), but we're just using a vector, so the "`.x`" part is correct.

The problem is the "thread index" means the offset of a thread within the current block of threads, not across all the total threads in multiple blocks. We have set blocks-per-thread as 32, so each block launches 32 threads. Hence, the offset of any thread in its current block has the range 0..31 only.

By launching 2 blocks of size 32 threads each, we get this sequence. The first block launches 32 threads, each having a "`threadIdx.x`" value of 0..31, so they correctly set all the first 32 vector elements to zero (in parallel). Our second block launches another 32 threads at the same time, also in parallel to our first block, but even though they're in the second block, their "`threadIdx.x`" value only relates to their own block, so they also have offset values of 0..31. Hence, we get another 32 threads running the kernel function, setting exactly the same values in the vector.

Overall, the outcome is a logic bug in the vector clearing function. The vector elements `v[0]..v[31]` all get cleared correctly (twice!), but the 33rd element `v[32]` is not touched. This CUDA kernel will only work correctly if `n<=threads_per_block`.

Array Bounds Violations

Like the six-million dollar man, we can rebuild the kernel. We need our first block to have values 0..31 and our second block to have values 32..63. But how do we know which block we're in?

The answer is another CUDA builtin variable called "`blockIdx`" which stands for "block index" and has value 0 for the first block, 1 for the next, and so on. It also has fields `x`, `y`, and `z`, but we only care about the "`.x`" values for a vector. By the way, the CUDA type is named "`dim3`" for these three-dimensional builtin object variables, but you don't need to declare them yourself.

Using the block index can compute indexes higher than threads-per-block:

```
int id = blockIdx.x * 32 + threadIdx.x; // BETTER
```

This is workable, but inelegant, and we could define a global constant:

```
const int threads_per_block = 32;     // PRETTIER
int id = blockIdx.x * threads_per_block + threadIdx.x;
```

This looks like good C++ code, but the CUDA idiom is actually to use another builtin variable called "`blockDim`" which stands for "block dimension" and in this case that means threads-per-block (in the `x` direction). Hence, the fully correct CUDA code, which is somewhat uglier, looks like:

```
// CORRECT
int id = blockIdx.x * blockDim.x + threadIdx.x;
```

The full GPU kernel function looks like this:

```
__global__
void aussie_clear buggy1(float* v, int n)
{
    // CORRECT
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    v[id] = 0.0;  // Clear one vector element.. OKAY?
}
```

Now you can run your corrected kernel, and it will clear all the elements of your vector, rather than only the first 32. But if you're running it in production, might want to get your mop and bucket ready to clean up the mess, because this kernel is going to segfault all over the place.

Remember that array bounds violation you were worried about before? Now it's happening for real.

The `blockIdx.x` value is 0..1, `blockDim.x` is 32 here (because that was our threads per block), and `threadIdx.x` has range 0..31 here. If you crank through all the options (sequentially since you don't have a GPU in your head), you'll find out the `id` array index variable has range 0..63, which all run in parallel.

Which isn't great for computing `v[id]` on a vector of size 33. You get 31 array bounds violations, in 31 different threads, all at the same time.

The correction is another common CUDA kernel idiom: the safety `if` statement. Yes, you can use control flow statements like `if` statements and loops inside the kernel.

The non-crashing version of the kernel looks like this:

```
__global__ void aussie_clear buggy1(float* v, int n)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < n)
        v[id] = 0.0;  // Clear vector element.. Safely!
}
```

For some reason, most of the CUDA examples write it as above, without curly braces, because adding braces would slow it down or something. I'd personally prefer this minor tweak to the C++ styling:

```
if (id < n) {
    v[id] = 0.0;
}
```

Finally, it works! This safety test stops any of the 31 threads that would have overflowed the vector from writing to bad memory, and they simply do nothing. The other 33 threads correctly clear all the elements of the vector, 32 of them in the first block, and 1 in the second block.

This code won't crash, but it's somewhat sluggy for two reasons:

- Redundant threads — the extra 31 threads run but do nothing.
- Warp divergence — some threads take different paths at the `if` statement.

The redundant threads don't actually slow anything down, since they run harmlessly at the same time in parallel with the useful threads. However, they waste electricity, and prevent the GPU from having better utilization of its silicon wafers to do other computations.

Warp divergence or "branch divergence" refers to the fact that GPUs like all their threads to follow exactly the same instruction path in parallel. Adding an `if` statement or a loop is problematic for performance, because some threads go left or right, and this slows down the whole process.

Mixing Host and Device Pointers

CUDA has two main classes of pointers to allocated memory: host pointers and device pointers. Host pointers are your regular pointers from `malloc` or `new` on the CPU, whereas device pointers are allocated on the GPU by functions such as `cudaMalloc`. The two cannot mix!

Here's a common bug:

```
float *vdest;
err = cudaMalloc((void**)&vdest, n * sizeof(float));
if ( err != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed: %d\n",
                    (int)n*(int)sizeof(float));
    return;
}
vdest[0] = 0.0;  // Oops!
```

What's wrong with that? Isn't it good style to initialize newly allocated memory, especially since `cudaMalloc` does not initialize the memory block for you?

Unfortunately, this is a pointer mix-up. The `cudaMalloc` function allocated a block of memory on the device, and I've just tried to initialize it on the CPU. Accessing the device vector in host code will literally crash the program.

There are two main ways to correct this:

> (a) use `cudaMemset` to initialize the device pointer, called from your host code on the CPU, or
>
> (b) allocate a host memory block with the normal C++ `malloc` function, initialize that newly allocated host vector (like the assignment used above), and then copy the host vector up to the GPU device using `cudaMemcpy`.

Both of these methods are host code that you run on the CPU. The second one is rather a mouthful, but it's the normal sequence for advanced kernel functions.

References

1. Mingyuan Wu, Husheng Zhou, Lingming Zhang, Cong Liu, Yuqun Zhang, 29 May 2019 (v3), *Characterizing and Detecting CUDA Program Bugs*, https://arxiv.org/abs/1905.01833.

12. Advanced CUDA Bugs

Advanced Bugs in CUDA C++

Mastery of CUDA means no bugs, right? Alas, no, it just means a better class of bugs. Here are some of the things that might go wrong:

- Scaling beyond the GPU's maximum thread count.
- Exceeding shared memory size with over-large blocks.
- Race conditions and other synchronization errors
- Using up too many registers and spilling into local memory.
- Kernel calls overflowing the GPU device's stack (esp. if using `alloca`).
- Shared memory access synchronization errors across threads.
- Cross-compilation on a different architecture to execution.

Some specific errors in CUDA runtime API usage:

- `cudaSetDevice` return error code needs to be checked, even at startup on a single GPU machine (it can fail sometimes).
- Calling host-only functions from device code.
- Kernel `printf` output buffer overload due to buffer not flushed.
- Warp shuffle when the target thread is inactive is undefined behavior (the returned value is undefined).
- Using `__assume`, `__builtin_assume`, or `__builtin_unreachable`, when it's actually false (oops!).
- Unsupported obscure `printf` formats (it's not exactly the same as the standard library).
- Kernel `printf` buffered output missing at program termination (needs synchronizing call, such as `cudaDeviceReset` or `cudaDeviceSynchronize`).
- Thread group blocks must have all threads participating.
- Trying to extend CUDA namespaces is undefined (e.g., "`cuda::`" or "`nv::`").
- Not checking for CUDA error return codes after every CUDA runtime call, and after every kernel launch with `cudaGetLastError`, which can miss errors or at least give a misleading appearance of where the error is occurring.

CUDA-specific arithmetic problems include:

- Kernel-called math functions are silent on errors (e.g., do not set `errno` or emit floating-point exceptions).
- Integer division by zero does not cause an exception in device code (and integer remainder operator).
- Integer overflow does not cause an exception (which is also normal for standard C++!)
- Floating point to integer conversion overflow is `INT_MAX` (or equivalent constant for other types) in device code.

Memory access errors include:

- Mixing `malloc/free` and `cudaMalloc/cudaFree`.
- Modifying `__constant__` address data.
- Accessing `__device__` addresses in host code.
- Local memory access outside that thread.
- Shared memory address ("`__shared__`") accessed in host code (probably a crash).
- Shared memory access outside the block that defined it on the device (i.e., thread block scope).
- Tensor fragment mismatches.
- Kernel invocation parameter sends a local address (it should be from `cudaMalloc`, `new` or `__device__` addresses)
- Streams or events created by host code accessed in device code.
- Event accessed outside the kernel block that created it (e.g., another block, in host code or a child grid).
- Streams created in device code accessed outside that block.
- Virtual alias synchronization problems with `cuMemMap`.
- `cudaFreeAsync` synchronization issues with its allocation (`cudaMallocAsync`) or other usage of the address.
- Synchronization issues with `cudaMalloc` (non-async version) and `cudaFreeAsync`.
- Accessing `__constant__` addresses in host code (likely segfault or other crash).
- Address from `cudaGetSymbolAddress` can only be used in host code.
- Mismatched `virtual` function call in host versus device code (i.e., object created in one, `virtual` function called in the other).

Portability and compatibility issues include:

- Generally, use of unsupported compute capability features is undefined.
- Arithmetic operations of the GPU may differ from x86 in areas undefined by the IEEE 754 standard.
- Unified Memory usage on a device lacking full support is undefined (compatibility issue).

And don't forget the slugs:

- Low occupancy rates on SMs.
- Poor load balancing across cores and SMs.
- Memory transfer costs
- Non-coalesced memory access patterns.
- Redundant barriers (unnecessary synchronization).
- Shared memory bank conflicts.
- Kernel output with `printf` (useful for tracing, not production).
- Register spills
- Poor cache management

There's also some common plain old bugs in AI algorithms that are fairly common:

- Tensor shape errors
- Mixing the offsets in square matrices (e.g., image data)

And since CUDA C++, is really just C++ plus a layer on top, there's still a boatload of low-level C++ coding mistakes. But don't fret too much, because soon you'll just be writing the comments and your AI copilot will write all the C++ statements.

Python Brain Mode

What's wrong with this CUDA C++ kernel code for GPU vector addition?

```
__global__ void aussie_add_vector_kernel_buggy_python(
        float* v1, float *v2, float *destv3, int n)
{
    // BUGGY: Add vectors, but C++ ain't Python!!
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < n)
        float ftmp1 = v1[id];  // Put into registers
        float ftmp2 = v2[id];
        destv3[id] = ftmp1 + ftmp2;
}
```

C++ is not Python! This is what happens when a programmer is forced to learn Python, and then has to context switch to a real programming language like CUDA C++.

Whereas indentation is used by Python for semantics, the C++ compiler does not use indentation for anything other than tokenization, and almost completely discards all whitespace.

The above `if` statement without braces around its body is actually trying to do this control flow in C++:

```
if (id < n) {
    float ftmp1 = v1[id];
}
float ftmp2 = v2[id];
destv3[id] = ftmp1 + ftmp2;
```

Hence, the safety test is not actually safe. In fact, it won't even compile, because `ftmp1` in declared inside the `if` statement branch, and has limited scope, so it can't be used in the addition operator at the end.

The corrected code is simply to add curly braces:

```
if (id < n) {
    float ftmp1 = v1[id];
    float ftmp2 = v2[id];
    destv3[id] = ftmp1 + ftmp2;
}
```

The other way to fix it is to remove the temporaries so that it's only one statement:

```
if (id < n)
    destv3[id] = v1[id] + v2[id];
```

Incidentally, I'm not convinced that using temporary variables in this way to force GPU register usage is really an optimization. The `nvcc` compiler probably puts them into registers anyway.

Confusing Host and Device Pointers

When there's a lot of different pointers to vectors and matrices floating around, it's easy to get confused. What's the bug in this code to clear a vector? Note that this is host code and various error checking has been removed for clarity.

```
// Set up the host vector
float* v = (float*)malloc(n * sizeof(float));  // Dynamic

// Set up the self-test data...
aussie_set_vector(v, n, 3.0);  // Set non-zero (test)

// Set up the device vector...
float* device_v = NULL;
int sz = n * sizeof(float);
cudaMalloc((void**)&device_v, sz);

// Copy to device vector
cudaMemcpy(device_v, v, sz, cudaMemcpyHostToDevice);

// Kernel launch
int threads_per_block = 32;
int blocks = (n + threads_per_block - 1)/threads_per_block;
aussie_clear_basic <<< blocks,threads_per_block >>> (v, n);

// Copy GPU data back to the CPU host vector....
cudaMemcpy(v, device_v, sz, cudaMemcpyDeviceToHost);

// Cleanup allocated memory
cudaFree(device_v);   // Free the device vector
free(v);   // Free the host vector
```

It's hard to see, but the kernel launch won't do what it's asked, but will fail with a CUDA error code. Hopefully this is being checked by the host code, but not in the code fragment above, though! If a kernel fails in a GPU forest, and there's no error check to hear it fail?

Anyway, this doesn't crash, but if you call `cudaGetLastError` anywhere, you'll see that this gets CUDA error 700, which is `cudaErrorIllegalAddress`. The error first appears after the kernel launch, so it's happening in the device code.

If you're stumped, one way to find out the cause of error 700 would be to run it with the `compute-sanitizer` tool, which finds memory access errors. It's part of the CUDA Toolkit and is free to use.

On Linux or a Google Colab virtual version of Linux, the command would be:

```
command-sanitizer a.out
```

The error report would be something like this:

```
========= Invalid __global__ write of size 4 bytes
=========  at 0x40 in aussie_run_clear_basic (float *, int)
=========  by thread (0,0,0) in block (0,0,0)
=========  Address 0x583fefd3e950 is out of bounds
=========  and is 37,289,833,797,296 bytes before the nearest
allocation at 0x7a2a27200000 of size 131,072 bytes
```

I particularly enjoy the fact that the faulty address is 37 quadzillion bytes away from where it should be. And that is actually quite a useful hint as to the cause. To narrow our focus, here's the kernel launch with the mistake:

```
aussie_clear_basic<<<blocks,threads_per_block>>>(v, n);
```

Your first thought is that something's wrong in the blocks and threads calculations. But actually, the culprit is simply `v`, which is a host vector allocated by `malloc` on the CPU, not a kernel vector allocated by `cudaMalloc`. The GPU cannot access memory on the host, unless it's been specially marked as "global" or similar. So, we've passed the kernel a host vector that device code isn't allowed to touch! The fix is to use the correct vector `device_v`, which is a device vector:

```
aussie_clear_basic<<< blocks,threads_per_block >>>
  (device_v /*FIX!*/ , n);
```

Copy-Paste Bugs for cudaMemcpy

It seems a little ironic that humans make copy-paste bugs when using the `cudaMemcpy` API. However, it's almost always used twice, before and after a kernel launch, and there are many ways to go wrong.

Firstly, here are the two main ways to write them correctly:

```
// Before (CPU-to-GPU)
cudaMemcpy(device_v, v, sz, cudaMemcpyHostToDevice);

// ... launch the kernel here

// After (GPU-to-CPU)
cudaMemcpy(v, device_v, sz, cudaMemcpyDeviceToHost);
```

Below are the many ways to call it incorrectly. In fact, I coded up some tests of `cudaMemcpy` to see which ones return a CUDA error status, and which are silent errors. Here are the ways to go wrong:

```
// device-to-host to a device pointer
cudaMemcpy(device_v, v, sz, cudaMemcpyDeviceToHost);
// host-to-device to a host pointer
cudaMemcpy(v, v, sz, cudaMemcpyHostToDevice);
// host-to-device from a device pointer
cudaMemcpy(device_v, device_v, sz, cudaMemcpyHostToDevice);
// device-to-host to a device pointer
cudaMemcpy(device_v, device_v, sz, cudaMemcpyDeviceToHost);
// device-to-host from NULL pointer
cudaMemcpy(v, NULL /*dv*/, sz, cudaMemcpyDeviceToHost);
// device-to-host to NULL pointer
cudaMemcpy(NULL/*v*/, device_v, sz, cudaMemcpyDeviceToHost);
// device-to-host too few bytes
cudaMemcpy(v, device_v, n /*sz*/, cudaMemcpyDeviceToHost);
// device-to-host zero bytes
cudaMemcpy(v, device_v, 0 /*sz*/, cudaMemcpyDeviceToHost);
// host-to-device too few bytes
cudaMemcpy(device_v, v, n /*sz*/, cudaMemcpyHostToDevice);
// host-to-device zero bytes
cudaMemcpy(device_v, v, 0 /*sz*/, cudaMemcpyHostToDevice);
// reverse sz and mode params
cudaMemcpy(device_v, v, (int)cudaMemcpyHostToDevice /*sz*/,
     (cudaMemcpyKind)sz /*cudaMemcpyHostToDevice*/);
// host-to-host from device pointer
cudaMemcpy(v, device_v, sz, cudaMemcpyHostToHost);
// host-to-host to device pointer
cudaMemcpy(device_v, v, sz, cudaMemcpyHostToHost);
// host-to-host same host pointer
cudaMemcpy(v, v, sz, cudaMemcpyHostToHost);
// device-to-device from host pointer
cudaMemcpy(device_v, v, sz, cudaMemcpyDeviceToDevice);
// device-to-device to host pointer
cudaMemcpy(v, device_v, sz, cudaMemcpyDeviceToDevice);
// device-to-device same device pointer
cudaMemcpy(device_v, device_v, sz,cudaMemcpyDeviceToDevice);
// device-to-device too many bytes
cudaMemcpy(device_v,device_v,sz*2,cudaMemcpyDeviceToDevice);
// host-to-host too many bytes");
cudaMemcpy(v, v, sz*2, cudaMemcpyHostToHost);
// host-to-device too many bytes
cudaMemcpy(device_v, v, sz*2, cudaMemcpyHostToDevice);
// device-to-host too many bytes
cudaMemcpy(v, device_v, sz*2, cudaMemcpyDeviceToHost);
```

And here is the output of my test program (abridged over multiple runs):

```
CUDA ERROR: 1 (invalid argument) - device-to-host to a device pointer
CUDA ERROR: 1 (invalid argument) - host-to-device to a host pointer
NO error detected - host-to-device from a device pointer
NO error detected - device-to-host to a device pointer
CUDA ERROR: 1 (invalid argument) - device-to-host from NULL pointer
CUDA ERROR: 1 (invalid argument) - device-to-host to NULL pointer
NO error detected - device-to-host too few bytes
NO error detected - device-to-host zero bytes
NO error detected - host-to-device too few bytes
NO error detected - host-to-device zero bytes
CUDA ERROR: 21 (invalid copy dir memcpy) - reverse sz and mode params
NO error detected - host-to-host from device pointer
NO error detected - host-to-host to device pointer
NO error detected - host-to-host same host pointer
CUDA ERROR: 1 (invalid argument) - device-to-device from host pointer
CUDA ERROR: 1 (invalid argument) - device-to-device to host pointer
NO error detected - device-to-device same device pointer
CUDA ERROR: 1 (invalid argument) - device-to-device too many bytes
NO error detected - host-to-host too many bytes
CUDA ERROR: 1 (invalid argument) - host-to-device too many bytes
CUDA ERROR: 1 (invalid argument) - device-to-host too many bytes
```

Some of the silent errors may be detected by `compute-sanitizer` when it runs, but I ran this test, and it didn't seem to find any more of the "too many bytes" overflows, except for the ones that emitted a CUDA error code.

Note that the `compute-sanitizer` tool also helpfully reports any of the CUDA error return statuses that get triggered by the CUDA runtime API, along with any other memory address failures detected.

Silent Kernel Launch Failures

This is another weird oddity about the CUDA Runtime API. If the kernel launch fails in a synchronous way, such as from too many threads-per-block or other grid dimension error, the error code gets lost.

Rather surprisingly, this lost error code occurs in a very common CUDA idiom:

- Kernel launch with <<<...>>> syntax.
- `cudaDeviceSynchronize` immediately thereafter.

Here's an example of code that will show this silent kernel launch failure:

```
int BADTHREADS = 32 + 1024; // More than 1024 is illegal
int blocks = (n + BADTHREADS - 1) / BADTHREADS;

// Launch failing kernel
aussie_clear_vector <<<blocks,BADTHREADS>>> (device_v, n);

errval = cudaDeviceSynchronize();
if (errval != cudaSuccess) {
    // CUDA error...
    AUSSIE_CUDA_ERROR(errval, "cudaDeviceSynchronize fail");
}
```

In this case, `cudaDeviceSynchronize` will incorrectly return `cudaSuccess`. And since kernel launches cannot return an error code, the failure code is lost. The kernel will never run any threads, and also never again report any error code. It's a silent kernel launch failure, and our poor clueless CPU probably thinks that the kernel is running.

It's not just `cudaDeviceSynchronize` that fails to detect the error code, but also other calls with implicit synchronization. For example, if we use `cudaMemcpy` just after the kernel launch, it too will return `cudaSuccess` after such a failed kernel launch.

The solution and apparently the only way to detect this kernel launch error is to call `cudaGetLastError` or `cudaPeekAtLastError` before any call to `cudaDeviceSynchronize`.

In any case, an example of the code that works is:

```
// Launch failing kernel
aussie_clear_vector_basic<<<blocks,BADTHREADS>>>(dv, n);
errval = cudaGetLastError();   // Correct!
if (errval != cudaSuccess) {
    // CUDA error...
    AUSSIE_CUDA_ERROR(errval, "cudaGetLastError fail");
}
errval = cudaDeviceSynchronize();
// etc.
```

Hence, we probably should update our preferred CUDA idiom to:

- Kernel launch with <<<...>>> syntax.
- Call `cudaGetLastError` or `cudaPeekAtLastError` afterwards.
- Optionally call `cudaDeviceSynchronize`.

And one final oddity: if we call `cudaPeekAtLastError` just after a failed kernel launch, the error code is returned correctly, but is somehow missed by subsequent calls such as `cudaMemcpy`. It's like it gets cleared, even though it was only supposed to be a "peek"!

Device Thread Limits

You have to feel sorry for the poor little hapless GPU chips. For a while they get to run mission-critical AI queries, like suggesting recipes using ingredients that start with 'P' and other important stuff. But then they overheat a little, and get sent to the remote camps to do Bitcoin mining.

There are several ways that you can exceed a GPU's limits:

- More than 1024 threads
- Too many blocks

The block size limit to 1024 is a hard limit, and you really should add an assertion before every kernel launch to assure that. The problem with launching too many blocks, and thus too many threads, on a GPU is more insidious.

Launching Far Too Many Blocks. Just when you thought it was safe to go back into the water, here's the news: this apparently safe and very simple kernel is actually broken:

```
__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.. Safely!
}
```

This is not production-grade CUDA C++ code. The kernel fails if `N` is too large, because it tries to always create exactly one thread per item. Hence, a very large value of `N` means a large number of blocks.

This can blow the little GPU's mind. If you try to work on a vector with a billion elements, that's a billion divided by 32 warps. Even the amazing NVIDIA GPUs have their limits. For example, a V100 tops out at 16,384 threads, which is a lot less than a billion.

Actually, I underestimated CUDA! When I ran this code in a test just now, it worked just fine! CUDA just schedules lots and lots of blocks, and takes care of it. The program takes a few seconds to run, so I guess it's not doing all those blocks in parallel, but there's no crashing or CUDA error codes. Even the unit tests passed. Amazing.

It does eventually fail, but I had to use `N=1<<30`, which is a huge value. And the error is not even in the block scheduling, it's that we get to the level of integer overflow, and this messes with the CUDA memory addressing scheme. Here is what `compute-sanitizer` finds:

```
========= Invalid __global__ write of size 4 bytes
=========    at 0x80 in aussie_clearvec_basic(float *, int)
=========      by thread (0,0,0) in block (0,0,0)
=========      Address 0x0 is out of bounds
=========      and is 140,236,623,642,624 bytes before the
nearest allocation at 0x7f8b62230000 of size 65,536 bytes
```

It actually reports errors about addresses `0x0`, `0x4`, `0x8`, and so on. This looks like some kind of integer overflow in the memory address logic. I guess it finally stretched CUDA's memory model to breaking point!

No, that's wrong! But then, no, after further investigation I've discovered that it's my bug, not CUDA's. I mean, address `0x0` is the null pointer, not some random address, so I should have twigged earlier.

I added a lot of assertions and error return checks to the caller code, and here's what I found:

- The device vector passed as a kernel parameter was null.
- The `cudaMalloc` vector was null.

Here's the culprit:

```
int sz = n * sizeof(float);
```

It turns out that if `n==1<<30` and `size(float)==4`, then `1<<32` is zero, by integer overflow. Hence, the size parameter passed to `cudaMalloc` was zero, and it was failing. My code was effectively doing:

```
int sz = 0;
cudaMalloc((void**)&device_v, sz);  // sz==0
```

There were no runtime warnings or CUDA error codes. If I'd declared the variable `n` as `const`, I would have got a compiler warning about integer constants overflowing, but at runtime there are no checks for integer overflow.

Anyway, this mistake has some important points:

- Humble pie for me!
- `cudaMalloc` does not return a CUDA error code if its size is zero.
- But `cudaMalloc` does return a null address, and I wasn't checking for that (i.e., "allocation failure").
- `cudaMalloc` will act oddly if the size overflows to a negative integer, because its parameter type is `size_t`, which is typically an `unsigned` type, so a negative integer value will overflow to become a very large positive unsigned integer (silently).
- Device pointer operations don't crash from a null dereference on the GPU, nor for an invalid address of `v[id]` that is based on a null pointer, such as `0x4` from `v[1]`, `0x8` from `v[2]`, etc.
- I should have safety-checked or asserted that "`device_v != NULL`" before passing it as a kernel launch parameter.

Solution: Use fewer blocks. The solution is obviously to modify the kernels so that they process more than one vector element, so that we can use fewer blocks overall (and don't need to overflow any integers!). If you're only a little too high, you can maybe manually modify the kernel so that each thread sets 2 vector elements by doing two assignments in sequence.

For a more general case, you actually need to add a loop into the kernel function, and do the calculations to work out how many elements each kernel needs to process, which is basically `N` divided by the maximum number of allowed threads.

There's nothing wrong with a kernel like this, and it just looks like an ordinary C++ loop, but we're getting a bit too advanced now. We'll defer the discussion of how to do loops in kernels, because there are scary things like "grid-stride loops" and "coalesced data accesses" to think about.

13. Self-Testing Code

What is Self-Testing Code?

Instead of doing work yourself, get a machine to do it. Who would have ever thought of that?

Getting your code to test itself means you can go get a cup of coffee and still be billing hours. The basic techniques include:

- Unit tests
- Regression tests
- Error checking
- Assertions
- Self-testing code blocks
- Debug wrapper functions

The simplest of these is unit tests, which aim to build quality brick-by-brick from the bottom of the code hierarchy. The largest techniques are to run full regression tests suites, or to add huge self-testing code blocks.

Self-Testing Code Block

Sometimes an assertion, unit test, or debug tracing printout is too small to check everything. Then you have to write a bigger chunk of self-testing code.

The traditional way to do this in code is to wrap it in a preprocessor macro:

```
#if DEBUG
   ... // block of test code
#endif
```

Another reason to use a different type of self-testing code than assertions is that you've probably decided to leave the simpler assertions in production code.

A simple test like this is probably fine for production:

```
assert(ptr != NULL);  // Fast
```

But a bigger amount of arithmetic may be something that's not for production:

```
assert(aussie_vector_sum(v, n) == 0.0);  // Slow
```

So, you probably want to have macros and preprocessor settings for both production and debug-only assertions and self-testing code blocks. The simple way looks like this:

```
#if DEBUG
assert(aussie_vector_sum(v, n) == 0.0);
#endif
```

Or you could have your own debug-only version of assertions that are skipped for production mode:

```
assert_debug(aussie_vector_sum(v, n) == 0.0);
```

The definition of "`assert_debug`" then looks like this in the header file:

```
#if DEBUG
#define assert_debug(cond) assert(cond) // Debug mode
#else
#define assert_debug(cond) // nothing in production
#endif
```

This makes the "`assert_debug`" macro a normal assertion in debug mode, but the whole coded expression disappears to nothing in production build mode.

The above example assumes a separate set of build flags defining the preprocessor macros for a production build.

Self-Test Code Block Macro

An alternative formulation of a macro for installing self-testing code using a block-style, rather than a function-like macro, is as follows:

```
SELFTEST {
    // block of debug or self-test statements
}
```

The definition of the `SELFTEST` macro looks like:

```
#if DEBUG
#define SELFTEST // nothing (enables!)
#else
#define SELFTEST if(1) {} else // disabled
#endif
```

This method relies on the C++ optimizer to fix the non-debug version, by noticing that "`if(1)`" invalidates the `else` clause, so as to remove the block of unreachable self-testing code that's not ever executed.

Note also that `SELFTEST` is not function-like, so we don't have the "forgotten semicolon" risk when removing `SELFTEST` as "nothing". In fact, the nothing version is actually when `SELFTEST` code is *enabled*, which is the opposite situation of that earlier problem.

Furthermore, note that we cannot use the "`do-while(0)`" trick in this different syntax formulation.

Self-Test Block Macro with Debug Flags

The compile-time on/off decision about self-testing code is not the most flexible method. The block version of `SELFTEST` can also have levels or debug flag areas.

One natural extension is to implement a "flags" idiom for areas, to allow configuration of what areas of self-testing code are executed for a particular run (e.g., a decoding algorithm flag, a normalization flag, a MatMul flag, etc.). One Boolean flag is set for each debugging area, which controls whether or not the self-testing code in that module is enabled or not.

A macro definition of `SELFTEST(flagarea)` can be hooked into the run-time configuration library for debugging output. In this way, it has both a compile-out setting (`DEBUG==0`) and dynamic runtime "areas" for self-testing code.

Here's the definition of the self-testing code areas:

```
enum selftest_areas {
    SELFTEST_NORMALIZATION,
    SELFTEST_MATMUL,
    SELFTEST_SOFTMAX,
    // ... more
};
```

A use of the `SELFTEST` method with areas looks like:

```
SELFTEST(SELFTEST_NORMALIZATION) {
    // ... self-test code
}
```

The `SELFTEST` macro definition with area flags looks like:

```
extern bool g_aussie_debug_enabled;  // Global override
extern bool DEBUG_FLAGS[100];     // Area flags

#if DEBUG
    #define SELFTEST(flagarea) \
        if(g_aussie_debug_enabled == 0 || \
              DEBUG_FLAGS[flagarea] == 0) \
        { /* do nothing */ } \
        else
#else
#define SELFTEST if(1) {} else // disabled completely
#endif
```

This uses a "debug flags" array idea as for the debugging output commands, rather than a single "level" of debugging.

Naturally, a better implementation would allow separation of the areas for debug trace output and self-testing code, with two different sets of levels/flags, but this is left as an extension for the reader.

Debug Stacktrace

There are various situations where it can be useful to have a programmatic method for reporting the "stack trace" or "backtrace" of the function call stack in C++.

Some examples where it's useful include:

- Your assertion macro can report the full stack trace on failure.
- Self-testing code similarly can report the location.
- Debug wrapper functions too.
- Writing your own memory allocation tracker library.

C++ has a standard call stack trace capability with its standardization in C++23. This is available via the "`std::stacktrace`" facility, such as printing the current stack via:

```
std::cout << "Stacktrace: "
  << std::stacktrace::current()
  << std::endl;
```

The C++23 `stacktrace` library is already supported by GCC and early support in MSVS is available via a compiler flag "`/std:c++latest`". There are also two different longstanding implementations of stack trace capabilities: glibc `backtrace` and Boost `StackTrace`.

Note that the C++23 standardized version is actually based on the original Boost version of stack trace functionality.

Unified Address Self-Testing

Pointers are doubly complicated in CUDA C++, because they can be host or device pointers, not to mention shared or constant memory. Adding some self-testing code can be beneficial to quality.

You can use the `cudaPointerGetAttributes` function to query information about any address.

Here's an example utility function:

```
bool aussie_is_device_pointer(void *ptr)
{
    cudaPointerAttributes attrib;
    cudaError_t err = cudaPointerGetAttributes(&attrib, ptr);
    if ( err != cudaSuccess) {
      printf("ERROR: %s: cudaPointerGetAttributes fail: %p\n",
            __func__, ptr);
        return false;
    }
    if (attrib.type/*memoryType*/==cudaMemoryTypeDevice) {
        return true;  // Device pointer
    }
    else if (attrib.type/*memoryType*/==cudaMemoryTypeHost) {
        return false;   // Host pointer
    }
    printf("ERROR: %s: pointer neither device nor host: %p\n",
        __func__, ptr);
    return false;
}
```

A few points about this idea:

- This runs in host code.
- Host pointer address detection is not as simple (discussed below).
- The documentation says the structure field name is "`memoryType`" but I had to use "`type`" instead (after scouring the header file).

Host pointer issues. Defining an `aussie_is_host_pointer` would seem to be just reversing two return values, but that doesn't work as well as you might think.

The type of `cudaMemoryTypeHost` only applies to host pointers in Unified Addressing, so this method fails for ordinary `malloc` or `new` pointers on the host. These basic addresses will get another setting value for "`type`" with value 0, whereas `cudaMemoryTypeHost` is 1, and `cudaMemoryTypeDevice` is 2.

Various extensions of this idea are possible. For example, you can also get the device details if it is a device pointer, and other details for host pointers. Unfortunately, I'm not aware of any way to get more detailed information, such as whether it's a `cudaMalloc` block, and its allocated byte size. But I can dream.

Kernel Launch Self-Testing

Calculations of grid dimensions such as block counts and block sizes can be error-prone. One idea is to add some self-testing code to auto-validate the calculations. This may be particularly beneficial for novice CUDA acolytes, but less so for experienced programmers.

Here's an example of the types of self-tests that are possible for a one-dimensional vector kernel:

```
#define AUSSIE_ERROR(mesg) \
    printf("ERROR: %s: %s\n", __func__, (mesg))

bool aussie_check_kernel_dimensions_1D(
       int blocks,
       int threads,
       int n
       )
{
    if (n == 0) {
        AUSSIE_ERROR("N is zero");
        return false;  // fail
    }
    if (n < 0) {
        AUSSIE_ERROR("N is negative");
        return false;  // fail
    }
    if (blocks == 0) {
        AUSSIE_ERROR("Zero block count");
        return false;  // fail
    }
    if (blocks < 0) {
        AUSSIE_ERROR("Negative block count");
        return false;  // fail
    }
    if (threads == 0) {
        AUSSIE_ERROR("Zero thread count");
        return false;  // fail
    }
    if (threads < 0) {
        AUSSIE_ERROR("Negative thread count");
        return false;  // fail
    }

    if (blocks == 1 && threads == 1) {
        AUSSIE_ERROR("WARN: Sequential execution!");
        // It's allowed (for beginners), drop down...
    }
    if (threads > 1024) {
        AUSSIE_ERROR("Thread count more than 1024 max");
        return false;  // fail
    }
```

```
        if (threads %32 != 0) {
            AUSSIE_ERROR("WARN: Threads not multiple of 32”);
            // Allow for novice ... drop down to keep going
        }

        // Note: Some total thread count checks assume
        //   ... 1 operation per thread (i.e., no loops)
        //   ... so this is really mainly for educational use
        //   ... in checking of simple kernels.

        int num = blocks * threads;
        if (num == n) {
            // Perfection...
            return true;
        }
        if (num < n) {
            // NOTE: Error in simple kernel,
            // ... but could be valid grid-stride loop usage...
            AUSSIE_ERROR("WARN: Thread total is lower than n (not enough
threads or grid-stride loop)");
            if (n % threads != 0) {
                AUSSIE_ERROR("WARN: Grid-stride loop kernel would have
wasted iterations");
            }
            return true;  // allow it
        }
        if (num > n) {
            AUSSIE_ERROR("WARN: Thread total > n (wasted threads)");
            return true;  // allow it
        }
        return true;  // No serious errors found...
    }
```

Note that this self-checking idea can be extended to a lot of CUDA Runtime C++ calls. This is discussed in detail in the debug wrapper function chapter.

14. Assertions

Why Use Assertions?

Of all the self-testing code techniques, my favorite one is definitely assertions. They're just so easy to add! The use of assertions in CUDA C++ programs can be a very valuable part of improving the quality of your work over the long term. They ensure that you find bugs early in the life cycle of code, and they don't have much impact on performance (if used correctly). I find them especially useful in getting rid of obvious glitches when I'm writing new code, but then I usually leave them in there.

The standard C++ library has had an "`assert`" macro since back when it was called C. In CUDA C++, things are a little more complex, because there are two aspects of using assertions:

- Device code assertions — use the `assert` builtin function or `printf`.
- Host code assertions — lots of options!

The use of assertions in kernel C++ code is very limited, but host code runs on standard C++ compilers on the CPU, so you can use the many available techniques for host platforms.

The simplest idea is just to use the builtin `assert` macro, which works in both device and kernel code. The `assert` macro is a convenient method of performing simple tests. The basic usage is illustrated to validate the inputs of a simple vector kernel:

```
#include <assert.h>
...
__device__ vector_sum(float v[], int n)
{
    assert(v != NULL);  // Easy!
    // ... etc
}
```

Compile-Time Assertions: static_assert

Runtime assertions have been a staple of C++ code reliability since the beginning of time. However, there's often been a disagreement over whether or not to leave the assertions in production code, because they inherently slow things down.

The modern answer to this conundrum is the C++ "`static_assert`" directive. This is like a runtime assertion, but it is fully evaluated at compile-time, so it's super-fast. Failure of the assertion triggers a compile-time error, preventing execution, and the code completely disappears at run-time.

Unfortunately, there really aren't that many things you can assert at compile-time. Most computations are dynamic and stored in variables at runtime. However, the `static_assert` statement can be useful for things like blocking inappropriate use of template instantiation code, or for portability checking such as:

```
static_assert(sizeof(float) == 4, "float is not 32 bits");
```

This statement is an elegant and language-standardized method to prevent compilation on a platform where a "`float`" data type is 64-bits, alerting you to a portability problem.

Device code assertions

There are two basic methods to implement assertions on device code:

- CUDA `assert` primitive
- Custom macro with `printf`

Note that you can also use `static_assert` in both host and device code, assuming you have a compile-time condition (e.g., `const` or `constexpr` result).

CUDA assert method for devices. Since compute capability 2.0, there has been an "`assert`" primitive in the CUDA runtime library that works on devices. It's in the C++ Programmer Guide, with declaration:

```
void assert(int expression);
```

If the assertion is successful with a non-zero expression value, nothing happens. If the assertion fails with a zero value, there are a few effects:

- Thread termination for all threads where it fails.
- Sets the `cudaErrorAssert` return error code (value 710).
- Error message printed to standard error (after kernel completion).
- Next CUDA call with synchronization with fail with error `cudaErrorAssert`

Here's an example of an assertion failure message:

```
aussie-clear-vector-test-assertions.cu:40:
void aussie_clear_vector_kernel_basic(float *, int):
block: [0,0,0], thread: [5,0,0]
Assertion `id < 5` failed.
```

An important point is that the assertion failure message does not appear immediately. Rather, the assertion message appears when the kernel has finished, at the next synchronization with the host code. This behavior is the same as the built-in `printf` function, when executed in kernel code. It gets buffered until the CPU is ready to print it out. However, note that `printf` output goes to `stdout`, whereas assertion failures print to `stderr`.

I'm not sure if assertion messages are printed if the host code on the CPU has already exited (i.e., didn't wait to synchronize). The behavior for buffered device `printf` messages is they are discarded in this situation, so maybe assertions use the same mechanism.

Removing device assertions from production code. This method is very similar to the original standard C assertions, which were declared in <assert.h>. As with the old-school C assertions, you can remove CUDA assertions from device code by defining the "`NDEBUG`" preprocessor macro at compile-time. Hence, the production build needs re-compilation of all CUDA C++ source files, not just re-linking.

Should you leave assertions in production code? There's a school of thought that it's worth the expense of extra assertion checking to get the supportability benefits of having your users finding your bugs for you. However, CUDA kernels are probably not the right place for this idea, since efficiency is critical in these code sections, but you might want to consider this policy for host code.

Custom printf assertions for device code. Since `printf` statements are allowed in device code, you can also declare your own custom assertion macro. For example, you might want an "assert warning" macro that doesn't abort the thread, or perhaps have a more graceful shutdown of the kernel in some way.

On the other hand, a custom device assertion is not really the preferred method in general, because CUDA `assert` failures are more properly handled, and meshed into the CUDA return code handling. If a CUDA device assertion fails, the error code `cudaErrorAssert` (710) is returned by the next synchronization primitive on the host.

Note that you can't even use `fprintf` in device code, but only `printf`, so it's hard to print to `stderr`. Here's an error message for device code:

```
  aussie-clear-vector-test-assertions.cu(41): error:
  calling a __host__ function("fprintf") from a __global__
function
  ("aussie_clear_vector_kernel_basic") is not allowed
```

If you want to define your own custom assertion macro for device code, make sure it has these features:

- Prints a message (obviously)
- Compiles to nothing if `NDEBUG` is declared.
- Aborts the thread (optionally), such as by `assert(0)` or `asm("trap;")`.

Assertions for both host and device code. If you want consistent assertion handling in both types of CUDA C++ code, there are a few options:

- `assert` primitive
- Custom assertion macro with `printf`

The builtin CUDA assert macro is a little idiosyncratic across device versus host code. For example, `assert` works fine in device code without any header include, but gets a compilation error in host code, and <assert.h> is needed in host code.

Note that it's somewhat difficult to define your own custom assertion macro in a way that it works on both device and host code. For example, I don't know of an easy way to get your custom assertion failure message to appear on `stderr`, since you're limited to using `printf` on devices. You could launch your kernels in a subprocess and use `freopen` to redirect the file pointers, but that seems a bit extreme to me.

Note that trying to define two different versions of the same assertion macro on device versus host code is very difficult to do with the same macro name. There are at least two obstacles:

(a) you can't use the `__CUDA_ARCH__` macro to separate them, because this macro is actually undefined in host code, and

(b) `nvcc` is always in host compilation mode in header files.

You can, of course, declare two different assertion macros with different names for device and host C++ code.

```
#define aussie_assert_HOST(cond)   // etc...
#define aussie_assert_DEVICE(cond) // etc...
```

If you want to ensure they get used in the right code, just trigger compiler errors by putting `fprintf` in the host version, and `__CUDA_ARCH__` in the device version. Actually, no, that idea of using `__CUDA_ARCH__` to prevent misuse didn't quite work, but you can instead use assertion macros that includes calls to wrong cross-mode functions, by declaring two "assertion failure" functions to call, which are declared as either `__device__` or `__host__`.

Custom Assertion Macros

An important point about the default "`assert`" macro on both host and device code is that its failure handling may not be what you want. The default device code assertion failure will trigger a `cudaErrorAssert` CUDA Runtime error when the condition fails. And the default C++ `assert` macro on the host CPU code will literally crash your program by calling the standard "`abort`" function, which triggers a fatal exception on Windows or a core dump on Linux.

That is fine for debugging, but it isn't usually what you want for production code. Hence, most professional C++ programmers declare their own custom assertion macros instead.

For example, here's my own "`aussie_assert`" macro in my own header file:

```
#define aussie_assert(cond) \
    ( (cond) || \
     aussie_assert_fail(#cond, __FILE__, __LINE__))
```

This tricky macro uses the short-circuiting of the "||" operator, which has a meaning like "or-else". So, think of it this way: the condition is true, *or else* we call the failure function. The effect is similar to an if-else statement, but an expression is cleaner in a macro.

The __FILE__ and __LINE__ preprocessor macros expand to the current filename and line number. The filename is a string constant, whereas the line number is an integer constant. The expression "#cond" is the "stringize" operator, which only works in preprocessor macros, and creates a string constant out of its argument.

Note that you can add "__func__" to also report the current function name if you wish. There's also an older non-standard __FUNCTION__ version of the macro. Note that the need for all these macros goes away once there is widespread C++ support for std::stacktrace, as standardized in C++23, in which case a failing assertion could simply report its own call stack in an error message.

When Assertions Fail. This aussie_assert macro relies on a function that is called only when an assertion has failed. And the function has to have a dummy return type of "bool" so that it can be used as an operand of the || operator, whereas a "void" return type would give a compilation error. Hence, the declaration is:

```
// Assertion failed
bool aussie_assert_fail(char* str, char* fname, int ln);
```

And here's the definition of the function:

```
bool aussie_assert_fail(
    char* str, char* fname, int ln)
{
    // Assertion failure has occurred...
    g_aussie_assert_failure_count++;
    printf("AUSSIE ASSERTION FAILURE: %s, %s:%d\n",
         str, fname, ln);
    return false;  // Always fails
}
```

This assertion failure function must always return "false" so that the assertion macro can be used in an if-statement condition.

Assertion Failure Extra Message

The typical assertion macro will report a stringized version of the condition argument (i.e., `#cond` is the special stringize operator), plus the source code filename, line number, and function name. This can be a little cryptic, so a more human-friendly extra message is often added. The longstanding hack to do this has been:

```
aussie_assert(fp!=NULL && "File open failed"); // Works
```

The trick is that a string constant has a non-null address, so `&&` on a string constant is like doing "*and true*" (and is hopefully optimized out). This gives the extra message in the assertion failure because the string constant is stringized into the condition (although you'll also see the "`&&`" and the double quotes, too).

Note that an attempt to be tricky with comma operator fails:

```
aussie_assert(fp!=NULL, "File open failed"); // Bug
```

There are two problems. Firstly, it doesn't compile because it's not the comma operator, but two arguments to the `aussie_assert` macro. Even if this worked, or if we wrapped it in double-parentheses, there's a runtime problem: this assertion condition will never fail. The result of the comma operator is the string literal address, which is never false.

Optional Assertion Failure Extra Message: The above hacks motivate us to see if we could allow an optional second parameter to assertions. We need two usages, similar to how "`static_assert`" currently works in C++:

```
aussie_assert(fp != NULL);
aussie_assert(fp != NULL, "File open failed");
```

Obviously, we can do this if "`aussie_assert`" was a function, using basic C++ function default arguments or function overloading.

If you have faith in your C++ compiler, just declare the functions "`inline`" and go get lunch. But if we don't want to call a function just to check a condition, we can also use C++ variadic macros.

Variadic Macro Assertions

C++ allows `#define` preprocessor macros to have variable arguments using the "`...`" and "`__VA_ARG__`" special tokens. Our `aussie_assert` macro changes to:

```
#define aussie_assert(cond, ...) \
   ( (cond) || \
      aussie_assert_fail(#cond, \
       __FILE__, __LINE__, __VA__ARG__) )
```

And we change our "`aussie_assert_fail`" to have an extra optional "message" parameter.

```
bool aussie_assert_fail(
   char* str, char* fname, int ln, char *mesg=0);
```

This all works fine if the `aussie_assert` macro has 2 arguments (condition and extra message) but we get a bizarre compilation error if we omit the extra message (i.e., just a basic assertion with a condition). The problem is that `__VA_ARG__` expands to nothing (because there's no optional extra message argument), and the replacement text then has an extra "`,`" just hanging there at the end of the argument list, causing a syntax error.

Fortunately, the deities who define C++ standards noticed this problem and added a solution in C++17. There's a dare-I-say "hackish" way to fix it with the `__VA__OPT__` special token. This is a special token whose only purpose is to disappear along with its arguments if there's zero arguments to `__VA_ARG__` (i.e., it takes the ball and goes home if there's no-one else to play with). Hence, we can hide the comma from the syntax parser by putting it inside `__VA_OPT__` parentheses. The final version becomes:

```
#define aussie_assert(cond, ...) \
   ( (cond) || \
      aussie_assert_fail(#cond, \
          __FILE__, __LINE__ \
          __VA_OPT__(,) __VA__ARG__) )
```

Note that the comma after `__LINE__` is now inside of a `__VA_OPT__` special macro. Actually, that's not the final, final version. We really should add "`__func__`" in there, too, to report the function name. Heck, why not add `__DATE__` and `__TIME__` while we're at it? Why isn't there a standard `__DEVELOPER__` macro that adds my name?

Assertless Production Code

Not everyone likes assertions, and coincidentally some people wear sweaters with reindeer on them. If you want to compile out all of the assertions from the production code, you can use this:

```
#define aussie_assert(cond)  // nothing
```

But this is not perfect, and has an insidious bug that occurs rarely (if you forget the semicolon). A more professional version is to use "`0`" and this works by itself, but even better is a "`0`" that has been typecast to type "`void`" so it cannot be accidentally used in any expression:

```
#define aussie_assert(cond) ( (void)0 )
```

The method to remove calls to the `aussie_assert` variadic macro version uses the "`...`" token:

```
#define aussie_assert(cond, ...) ( (void)0 )
```

Personally, I don't recommend doing this at all, as I think that assertions should be left in the production code for improved supportability. I mean, come on, recycle and reuse, remember? Far too many perfectly good assertions get sent to landfill every year.

Assertion Return Value Usage

Some programmers like to use an assertion style that tests the return code in their `assert` macro:

```
if (assert(ptr != NULL)) {  // Risky
    // Normal code
    ptr->count++;
}
else {
    // Assertion failed
}
```

This assertion style can be used if you like it, but I don't particularly recommend it, because it has a few risks.

The risks include:

1. The hidden assert failure function must return "`false`" so that "`if`" test fails when the assertion fails.

2. Embedding assertions deeply into the main code expressions increases the temptation to use side effects like "++" in the condition, which can quietly disappear if you ever remove the assertions from a production build:

```
if (assert(++i >= 0)) { // Risky
    // ...
}
```

3. The usual assertion removal method of "`((void)0)`" will fail with compilation errors in an `if` statement. Also using a dummy replacement value of "0" is incorrect, and even "1" is not a great option, since the "`if(assert(ptr!=NULL))`" test becomes the unsafe "`if(1)`". A safer removal method is a macro:

```
#define assert(cond) (cond)
```

Or you can use an `inline` function:

```
inline void assert(bool cond) { } // Empty
```

This avoids crashes, but may still leave debug code running (i.e., a slug, not a bug). It relies on the optimizer to remove any assertions that are not inside an "`if`" condition, which just leave a null-effect condition sitting there. Note also that this removal method with "`(cond)`" is also safer because keeping the condition also retains any side-effects in that condition (i.e., the optimizer won't remove those!).

Generalized Assertions

Once you've used assertions for a while, they begin to annoy you a little bit. They can fail a lot, especially during initial module development and unit testing of new code. And that's the first time they get irritating, because the assertion failure reports don't actually give you enough information to help debug the problem. However, you can set a breakpoint on the assertion failure code when running in `cuda-gdb`, so that's usually good enough.

The second time that assertions are annoying is when you ship the product. That's when you see assertion failures in the logs as an annoying reminder of your own imperfections. Again, there's often not enough information to reproduce the bug.

So, for your own sanity, and for improved supportability, consider extending your own assertion library into a kind of simplified unit-testing library. The extensions you should consider:

- Add `std::stacktrace` capabilities if you can, or use Boost Stacktrace or GCC `backtrace` as a backup. Printing the whole stack trace on an assertion failure is a win.
- Add extra assertion messages as a second argument.
- Add `__func__` to show the function name, if you haven't already.

And you can also generalize assertions to cover some other common code failings.

- Unreachable code assertion
- "Null pointer" assertion
- Integer value assertions
- Floating-point value assertions
- Range value assertions

Creating specialized assertion macros for these special cases also means the error messages become more specific.

Unreachable code assertion

This is an assertion failure that triggers when code that should be unreachable actually got executed somehow. The simple way that programmers have done this in the past is:

```
aussie_assert(0);  // unreachable
```

And you can finesse that a little with just a better name:

```
#define aussie_assert_not_reached() \
        ( aussie_assert(false) )
...
aussie_assert_not_reached(); // unreachable
```

Here's a nicer version with a better error message:

```
#define aussie_assert_not_reached() \
    ( aussie_assert_fail(  \
           "Unreachable code was reached", \
            __FILE__, __LINE__) )
```

Once-only execution assertion

Want to ensure that code is never executed twice? A function that should only ever be called once? Here's an idea for an assertion that triggers on the second execution of a code pathway, by using its own hidden "`static`" call counter local variable (only works in host code):

```
#define aussie_assert_once()  do { \
    static int s_count = 0; \
        ++s_count; \
        if (s_count > 1) { \
        aussie_assert_fail("Code executed twice", \
                __FILE__, __LINE__); \
        } \
   } while(0)
```

Restricting any block of code to once-only execution is as simple as adding a statement like this:

```
aussie_assert_once();    // Not twice!
```

This can be added at the start of a function, or inside any `if` statement or `else` clause, or at the top of a loop body (although why is it coded as a loop if you only want it executed once?). Note that this macro won't detect the case where the code is never executed. Also note that you could customize this macro to return an error code, or throw a different type of exception, or other exception handling method when it detects double-executed code.

Function Call Counting

The idea of once-only code assertions can be generalized to a count. For example, if you want to ensure a function isn't called too many times, use this code:

```
aussie_assert_N_times(1000);
```

Here's the macro, similar to `aussie_assert_once`, but with a parameter:

```
#define aussie_assert_N_times(ntimes)  do { \
        static int s_count = 0; \
        ++s_count; \
        if (s_count > (ntimes)) { \
          aussie_assert_fail( \
          "Code executed more than " \
               #ntimes " times", \
                __FILE__, __LINE__); \
            } \
   } while(0)
```

This checks for too many invocations of the code block. Checking for "too few" is a little trickier, and would need a `static` smart counter object with a destructor. Again, this only works in host code, as we don't have `static` local parameters in device code, and we'd need some other approach.

Detecting Spinning Loops

Note that the above call-counting macro doesn't work for checking that a loop isn't spinning. It might seem that we can use the above macro at the top of the loop body to avoid a loop iterating more than 1,000 times. But it doesn't work, because it will count multiple times that the loop is entered, not just a single time. If we want to track a loop call count, the counter should not be a "`static`" variable, and it's more difficult to do in a macro. The simplest method is to hand-code the test:

```
int loopcount = 0;
while (...) {
    if (++loopcount > 1000) {  // Spinning?
        // Warn...
    }
}
```

The upside of using a simple approach: this should work on both device and host code.

Generalized Variable-Value Assertions

Various generalized assertion macros can not only check values of variables, but also print out the value when the assertion fails. The basic method doesn't print out the variable's value:

```
aussie_assert(n == 10);
```

A better way is:

```
aussie_assertieq(n, 10);  // n == 10
```

The assertion macro looks like:

```
#define aussie_assertieq(x,y) \
     (( (x) == (y)) || \
      aussie_assert_fail_int(#x "==" #y, \
          (x), "==", (y), \
         __FILE__, __LINE__))
```

The assertion failure function has extra parameters for the variables and operator string:

```
bool aussie_assert_fail_int(char* str, int x,
     char *opstr, int y, char* fname, int ln)
{
    // Assert failure has occurred...
    g_aussie_assert_failure_count++;
    fprintf(stderr, "ASSERT FAIL: %s, %d %s %d, %s:%d\n",
        str, x, opstr, y, fname, ln);
    return false;  // Always fails
}
```

If you don't mind lots of assertion macros with similar names, then you can define named versions for each operator, such as:

- aussie_assertneq — !=
- aussie_assertgtr — >
- aussie_assertgeq — >=
- aussie_assertlss — <
- aussie_assertleq — <=

If you don't mind ugly syntax, we can generalize this with an operator parameter:

```
aussie_assertiop(n, ==, 10);
```

The macro with an "op" parameter is:

```
#define aussie_assertiop(x, op, y) \
     (( (x) op (y)) || \
      aussie_assert_fail_int(#x #op #y, \
          (x), #op, (y), \
         __FILE__, __LINE__))
```

And finally, you have to duplicate all of this to change from `int` to `float` type variables. For example, the simplistic way is to define new macros named "`aussie_assertfeq`" and "`aussie_assertfop`", and then also a failure function named "`aussie_assert_fail_float`".

There's probably a fancy way to avoid this using function overloading with different types, or C++ templates and compile-time type traits, but only if you're smarter than me.

Assertions for Function Parameter Validation

Assertions and toleration of exceptions have some tricky overlaps. Consider the modified version of vector summation with my own "`aussie_assert`" macro instead:

```
__device__ float vector_sum(float v[], int n)
{
    aussie_assert(v != NULL);
    // etc..
}
```

What happens when this assertion fails in a custom assertion macro? In both host and device code, the execution will progress after the assertion, in which case any use of v will be a null pointer dereference.

This code is not very resilient!

Hence, the above code works fine only if your custom "`aussie_assert`" assertion macro throws an exception on the host. This doesn't work on the host, but your custom macro could call the builtin `assert` primitive on the device. This requires that you have a robust exception handling mechanism in place above it, for the caught exception on the host, or the `cudaErrorAssert` code from the device, which is a significant amount of work.

The alternative is to both assert and handle the error in the same place, which makes for a complex block of code:

```
aussie_assert(v != NULL);
if (v == NULL) {
     return 0.0;  // Tolerate
}
```

Slightly more micro-efficient is to only test once:

```
if (v == NULL) {
    aussie_assert(v != NULL); // Always triggers
    return 0.0;  // Tolerate
}
```

This is a lot of code that can get repeated all over the place. Various copy-paste coding errors are inevitable.

Assert Parameter and Return

An improved solution is an assertion macro that captures the logic "check parameter and return zero" in one place. Such a macro first tests a function parameter and if it fails, the macro will not only emit an assertion failure message, but will also tolerate the error by returning a specified default value from the function.

Here's a generic version for any condition:

```
#define aussie_assert_and_return(cond,retval) \
    if (cond) {} else { \
        aussie_assert_fail(#cond " == NULL", \
                __FILE__, __LINE__); \
        return (retval); \
    }
```

The usage of this function is:

```
float aussie_vector_something(float v[], int n)
{
        aussie_assert_and_return(v != NULL, 0.0f);
    ...
}
```

The above version works for any condition. Here's another version specifically for testing an incoming function parameter for a `NULL` value:

```
#define aussie_assert_param_tolerate_null(var,retval) \
    if ((var) != NULL) {} else { \
        aussie_assert_fail(#var " == NULL", \
                __FILE__, __LINE__); \
        return (retval); \
    }
```

The usage of this function is:

```
aussie_assert_param_tolerate_null(v, 0.0f);
```

If you want to be picky, a slightly better version wraps the "`if-else`" logic inside a "`do-while(0)`" trick. This is a well-known trick to make a macro act more function-like in all statement structures.

```
#define aussie_assert_param_tolerate_null2(var,retval) \
      do { if ((var) != NULL) {} else { \
          aussie_assert_fail(#var " == NULL", \
                      __FILE__, __LINE__); \
          return (retval); \
      }} while(0)
```

The idea of this macro is to avoid lots of parameter-checking boilerplate that will be laborious and error-prone.

But it's also an odd syntax style to hide a `return` statement inside a function-like preprocessor macro, so this is not a method that will suit everyone.

Next-Level Assertion Extensions

Here are some final thoughts on how to further improve your assertions:

- Change any often-triggered assertions into proper error messages with fault tolerance. Users don't like seeing assertion messages. They're kind of like gibberish to ordinary mortals.
- Add extra context information in the assertion message (i.e., add an extra information string). This is much easier to read than a stringized expression, filename with line number, or multi-line stack trace.
- Add unique codes to assertion messages for increased supportability. Although, maybe not, because any assertion that's triggering often enough to need a code, probably shouldn't remain an assertion!
- `inline` assertion function? Why use macros? Maybe these assertions should instead be an `inline` function in modern C++? And it could report context using `std::backtrace`. All I can say is that old habits die hard, and I still don't trust the optimizer to actually optimize much.

The downside of assertions is mainly that they make you lazy as a programmer because they're so easy to add. But sometimes no matter how good they seem, you have to throw an assertion into the fires of Mordor.

The pitfalls include:

- Don't use assertions instead of user input validation.
- Don't use assertions to check program configurations.
- Don't use assertions as unit tests (it works, but bypasses the test harness statistics).
- Don't use assertions to check if a file opened.

You need to step up and code the checks of input and configurations as part of proper exception handling. For example, it has to check the values, and then emit a useful error code if they've failed, and ideally it's got a unique error code as part of the message, so that users can give a code to support if they need. You really don't want users to see the dirty laundry of an assertion message with its source file, function name, and line number.

15. Debug Wrapper Functions

Why Debug Wrapper Functions?

The idea of debug wrapper functions is to fill a small gap in the self-checking available in the CUDA ecosystem. There are two types of self-testing that happen when you run CUDA C++ programs:

- Error checks in the CUDA Runtime API (i.e., when an API doesn't return `cudaSuccess` in host code).
- `compute-sanitizer` detection of numerous run-time errors.

Both of these methods are highly capable and will catch a lot of bugs. To optimize your use of these capabilities in debugging, you should:

- Test all error return codes (e.g., a fancy macro method), and
- Run `compute-sanitizer` on lots of unit tests and regression tests in your CI/CD approval process, or, when that gets too slow, at least in the nightly builds.

But this is not perfection! And there's two main reasons that some bugs will be missed:

- CUDA Runtime doesn't detect all the bugs (because the main aim is *fast*!).
- You have to remember to run `compute-sanitizer` on your code.

Okay, so I'm joking about "remembering" to run the debug tests, because you've probably got them running automatically in your build. But there's some real cases where the application won't ever be run in debug mode:

- Many internal failures trigger no visible symptoms for users (silent failures).
- Customers cannot run `compute-sanitizer` on their premises (unless you ask nicely).
- Your website "customers" also cannot run it on the website backends.
- Some applications are too costly to re-run just to debug an obscure error (I'm looking at you, AI training).

Hence, in the first case, there's bugs missed in total silence, never to be fixed. And in the latter cases, there's a complex level of indirection between the failure occurring and the CUDA C++ programmer trying to reproduce it in the test lab. It's much easier if your application self-diagnoses the error!

Fast Debug Wrapper Code

But it's too slow, I hear you say. Running the code with `compute-sanitizer` is much slower than without. We can't ship an executable where the kernels have so much debug instrumentation that they're running that much slower.

You're not wrong, and it's the age-old quandary about whether to ship testing code. Fortunately, there are a few solutions:

- Use fast self-testing tricks like magic numbers in memory.
- Have a command-line flag or config option that turns debug tests on and off at runtime.
- Have "fast" and "debug" versions of your executable (e.g., ship both to beta customers).

At the very least, you could have a lot of your internal CUDA C++ code development and QA testing done on the debug wrapper version that self-detects and reports internal errors.

As the first point states, there are "layers" of debugging wrappers (also ogres, like Shrek). You can define very fast or very slow types of self-checking code into debug wrapper code. These self-tests can be as simple as parameter null tests or as complex as detecting memory stomp overwrites with your own custom code. In approximate order of time cost, here are some ideas:

- Parameter basic validation (e.g., null pointer tests).
- Address type validation (e.g., via `cudaPointerGetAttributes`).
- Magic values added to the initial bytes of uninitialized and freed memory blocks.
- Magic values stored in every byte of these blocks.
- Tracking 1 or 2 (or 3) of the most recently allocated/freed addresses.
- Hash tables to track addresses of every allocated or freed memory block.

I've actually done all of the above for a debug library in standard C++, which I'm now working on updating for CUDA C++. Make sure you check the Aussie AI website to see when it gets released.

CUDA C++ Runtime Wrapper Functions

You can use macros to intercept various CUDA Runtime API calls. For example, here's a simple interception of `cudaMalloc`:

```
// intercept cudaMalloc
#undef cudaMalloc
#define cudaMalloc aussie_cudaMalloc
cudaError_t aussie_cudaMalloc(void** addr_of_v, int sz);
```

Once intercepted, the wrapper code can perform simple validation tests of the various parameters. Here's a simple wrapper for the `cudaMalloc` function in a debug library for CUDA C++ that I'm working on:

```
cudaError_t aussie_cudaMalloc_simple(void** addr_of_v, int sz)
{       // Debug wrapper version: cudaMalloc()
    AUSSIE_DEBUGLIB_TRACE("cudaMalloc called");
    AUSSIE_DEBUG_PRINTF("%s: == cudaMalloc: addr=%p, sz=%d\n",
          __func__, addr_of_v, sz);
    g_aussie_cuda_malloc_count++;
    if (!addr_of_v) {  // null pointer...
        AUSSIE_CHECK(addr_of_v != NULL, "AUS006",
                          "cudaMalloc null address");
        return cudaErrorInvalidValue ;
    }
    AUSSIE_CHECK(sz!=0, "AUS007", "cudaMalloc size zero");
    AUSSIE_CHECK(sz>=0, "AUS008", "cudaMalloc size negative");
    // Call the real cudaMalloc
    void *new_v = NULL;
    cudaError_t err = cudaMalloc(&new_v, sz);
    if (err != cudaSuccess) {
          AUSSIE_ERROR("AUS200", "ERROR: cudaMalloc error");
    }
    *addr_of_v = new_v;   // Store it for return to caller
    AUSSIE_CHECK(new_v != NULL, "AUS009", "cudaMalloc fail");
    return err;
}
```

This actually has four levels of tests:

- Validation of called parameter values.
- Detection of CUDA runtime errors (pass-through).
- Detection of memory allocation failure.
- Builtin debug tracing macros that can be enabled.

A more advanced version could check pointer addresses are valid and are not been previously freed, and a variety of other memory errors. Coming soon!

Standard C++ Debug Wrapper Functions

In addition to wrapping the CUDA Runtime API calls, it can be helpful during debugging to wrap some standard C++ library function calls with your own versions, so as to add additional parameter validation and self-checking code. Some of the functions which you might consider wrapping include:

- `malloc`
- `calloc`
- `memset`
- `memcpy`
- `memcmp`

If you're doing string operations in your code, you might consider wrapping these:

- `strdup`
- `strcmp`
- `strcpy`
- `sprintf`

Note that you can wrap the C++ "`new`" and "`delete`" operators at the linker level by defining your own versions, but not as macro intercepts. You can also intercept the "`new[]`" and "`delete[]`" array allocation versions at link-time.

There are different approaches to consider when wrapping system calls, which we examine using `memset` as an example:

- Leave "`memset`" calls in your code (auto-intercepts)
- Use "`memset_wrapper`" in your code instead (manual intercepts)

Macro auto-intercepts: You might want to leave your code unchanged using `memset`. To leave "`memset`" in your code, but have it automatically call "`memset_wrapper`" you can use a macro intercept in a header file.

```
#undef memset // ensure no prior definition
#define memset memset_wrapper  // Intercept
```

Note that you can also use preprocessor macros to add context information to the debug wrapper functions.

For example, you could add extra parameters to "memset_wrapper" such as:

```
#define memset(x,y,z) \
  memset_wrapper((x),(y),(z),__FILE__,__LINE__,__func__)
```

Note that in the above version, the macro parameters must be parenthesized even between commas, because there's a C++ comma operator that could occur in a passed-in expression.

Also note that these context macros (e.g., __FILE__) aren't necessary if you have a C++ stack trace library, such as std::stacktrace, on your platform.

Variadic preprocessor macros: Note also that there is varargs support in C++ #define macros. If you want to track variable-argument functions like sprintf, printf, or fprintf, or other C++ overloaded functions, you can use "..." and "__VA_ARGS__" in preprocessor macros.

Here's an example:

```
#define sprintf(fmt,...) \
   sprintf_wrapper((fmt), \
      __FILE__,__LINE__,__func__, __VA_ARGS__ )
```

Manual Wrapping: Alternatively, you might want to individually change the calls to memset to call memset_wrapper without hiding it behind a macro. If you'd rather have to control whether or not the wrapper is called, then you can use both in the program, wrapped or non-wrapped.

Or if you want them all changed, but want the intercept to be less hidden (e.g., later during code maintenance), then you might consider adding a helpful reminder instead:

```
#undef memset
#define memset dont_use_memset_please
```

This trick will give you a compilation error at every call to memset that hasn't been changed to memset_wrapper.

Example: memset Wrapper Self-Checks

Here's an example of what you can do in a wrapper function called "`memset_wrapper`" from one of the Aussie AI projects:

```
void *memset_wrapper(void *dest, int val, int sz) // Wrap
{
    if (dest == NULL) {
        aussie_assert2(dest != NULL, "memset null dest");
        return NULL;
    }
    if (sz < 0) {
        // Why we have "int sz" not "size_t sz" above
        aussie_assert2(sz >= 0, "memset size negative");
        return dest;  // fail
    }
    if (sz == 0) {
        aussie_assert2(sz != 0,
             "memset zero size (reorder params?)");
        return dest;
    }
    if (sz <= sizeof(void*)) {
        // Suspiciously small size
        aussie_assert2(sz > sizeof(void*),
            "memset with sizeof array parameter?");
        // Allow it, keep going
    }
    if (val >= 256) {
        aussie_assert2(val < 256, "memset value not char");
        return dest; // fail
    }
    void* sret = ::memset(dest, val, sz); // Call real one!
    return sret;
}
```

It's a judgement call whether or not to leave the debug wrappers in place, in the vein of *speed versus safety*. Do you prefer sprinting to make your flight, or arriving two hours early? Here's one way to remove the wrapper functions completely with the preprocessor if you've been manually changing them to the wrapper names:

```
#if DEBUG
    // Debug mode, leave wrappers..
#else // Production (remove them all)
    #define memset_wrapper memset
    //... others
#endif
```

Compile-time self-testing macro wrappers

Here's an idea for combining the runtime debug wrapper function idea with some additional compile-time tests using `static_assert`.

```
#define memset_wrapper(addr,ch,n) ( \
    static_assert(n != 0), \
    static_assert(ch == 0), \
    memset_wrapper((addr),(ch),(n),\
       __FILE__,__LINE__,__func__))
```

The idea is interesting, but it doesn't really work, because not all calls to the `memset` wrapper will have constant arguments for the character or the number of bytes, so the `static_assert` commands will fail in that case. You could use standard assertions, but this adds runtime cost.

Note that it's a self-referential macro, but that C++ guarantees it only gets expanded once (i.e., there's no infinite recursion of preprocessor macros).

Generalized Self-Testing Debug Wrappers

The technique of debug wrappers can be extended to offer a variety of self-testing and debug capabilities. The types of messages that can be emitted by debug wrappers include:

- Input parameter validation failures (e.g., non-null)
- Failure returns (e.g., allocation failures)
- Common error usages
- Informational tracing messages
- Statistical tracking (e.g., call counts)

Personally, I've built some quite extensive debug wrapping layers over the years. It always surprises me that this can be beneficial, because it would be easier if it were done fully by the standard libraries of compiler vendors.

The level of debugging checks has been increasing significantly (e.g., in GCC), but I still find value in adding my own wrappers.

There are several major areas where you can really self-check for a lot of problems with runtime debug wrappers:

- File operations
- Memory allocation
- String operations

These are left as an exercise for the reader!

Link-Time Interception: new and delete

Macro interception works for CUDA APIs like `cudaMalloc`, and for standard C++ functions like `malloc` and `free`, but you can't macro-intercept the `new` and `delete` operators, because they don't use function-like syntax. Fortunately, you can use link-time interception of these operators instead, simply by defining your own versions.

Note that defining class-level versions of the `new` and `delete` operators is a well-known optimization, but this isn't what we're doing here. Instead, this link-time interception requires defining four operators at global scope:

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

Note that you cannot use the real `new` and `delete` inside these link-time wrappers. They would get intercepted again, and you'd have infinite stack recursion. However, you can call `malloc` and `free` instead, assuming they aren't also macro-intercepted. Here's the simplest versions:

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

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

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

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

This method of link-time interception is an officially sanctioned standard C++ language feature since the 1990s. Be careful, though, that the return types and parameter types are precise, using `size_t` and `void*`, as you cannot use `int` or `char*`.

Also, declaring these functions as `inline` gets a compilation warning, and is presumably ignored by the `nvcc` compiler, as this requires link-time interception.

This code runs fine with `nvcc` compilation, but the above example isn't much of a debugging wrapper, more like just a "wrapper," because it does no error checking.

However, when I started adding more self-tests, I triggered warnings about "`calling a __host__ function from a __host__ __device__ function.`" It seems that `nvcc` compiles these functions as both host and device code, which makes sense.

Unfortunately, when I tried to work around this by declaring the operators as host-only versions using the `__host__` specifier, it triggered compilation errors. Declaring two versions, one with `__host__` and the other with `__device__`, also didn't work (nor did `__global__`).

Maybe there's a workaround possible by putting these operator definitions into standard C++ code that's only processed by `gcc`, not by `nvcc`, and then link it in.

In the absence of a solution for now, this means that we're limited to using the subset of C++ that can run on the device inside these link-time interceptions. Hence, there are significant problems trying to generalize this into a useful debugging wrapper library, because any use of host-specific aspects such as global variables triggers compilation errors.

Here's an example of some ideas of some basic possible checks with `printf` outputting:

```
#define AUSSIE_ERROR(mesg, ...) \
    ( printf((mesg) __VA_OPT__(,) __VA_ARGS__ ) )

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

Note that you can't use `__FILE__` or `__LINE__` as these are link-time intercepts, not macros. Maybe you could use `std::backtrace` instead, but I have my doubts.

References

Note that Aussie AI has an active project for a CUDA C++ debug wrapper library with support for intercepting a wide range of CUDA C++ functions. Find more information at https://www.aussieai.com/cuda/projects.

16. Debug Tracing

Debug Tracing Messages

Ah, yes, worship the mighty `printf`!

A common debugging method is adding debug trace output statements to a program to print out important information at various points in the program. Judicious use of these `printf` statements can be highly effective in localizing the cause of an error, but this method can also lead to huge volumes of not particularly useful information.

One desirable feature of this method is that the output statements can be selectively enabled at either compile-time or run-time.

Debug tracing messages are informational messages that you only enable during debugging. These are useful to software developers to track where the program is executing, and what data it is processing.

The simplest version of this idea looks like:

```
#if DEBUG
printf("DEBUG: I am here!\n");
#endif
```

A better solution is to code some BYO debug tracing macros. Here's a macro version:

```
#define aussie_debug(str) \
   ( printf("DEBUG: %s\n", (str)) )
...
aussie_debug<("I am here!");
```

Device output limits. The output from `printf` on the GPU is limited to a buffer size. Firstly, this means that GPU trace output may not appear immediately. Secondly, it means that some output can get lost.

If the CPU does not synchronize often enough, or the GPU emits far too much output, then tracing messages will overflow the circular buffer and overwrite the first output. Hence, the earlier trace messages will be lost forever. Consider carefully the volume of output needed when debug tracing, and also add more frequent host calls to synchronize with the device.

You can also check or change the size of the circular buffer by calling `cudaDeviceGetLimit` or `cudaDeviceSetLimit` using the property `cudaLimitPrintfFifoSize`. The GPU and CPU clear the output buffer in some behind-the-scenes magic whenever they synchronize. Remember that the output is only stored in a buffer on the GPU, and is actually coming to the screen from the CPU!

Output to stderr. Note that we could use `fprintf` to `stderr` if we were sure it wasn't needed to run on the device, which only supports `printf`. And here's the C++ stream version, which also won't work in device code:

```
#define aussie_debug(str) \
   ( std::cerr << str << std::endl )
...
aussie_debug("DEBUG: I am here!");
```

In order to only show these when debug mode is enabled in the code, our header file looks like this:

```
#if DEBUG
    #define aussie_debug(str) \
       ( std::cerr << str << std::endl )
#else
    #define aussie_debug(str) // nothing
#endif
```

Missing Semicolon Bug: Professional programmers prefer to use "`0`" rather than emptiness to remove the debug code when removing it from the production version. It is also good to typecast it to "`void`" type so it cannot accidentally be used as the number "`0`" in expressions. Hence, we get this improved version for removing a debug macro:

```
#define aussie_debug(str) ((void)0)  // better!
```

It's not just a stylistic preference. The reason is that the "nothing" version can introduce an insidious bug if you forget a semicolon after the debug trace call in an `if` statement:

```
if (something) aussie_debug("Hello world") // Missing!
x++;
```

If the "nothing" macro expansion is used, then the missing semicolon leads to this code:

```
if (something) // nothing
x++;
```

Can you see why it's a bug? Instead, if the expansion is "`((void)0)`" then this missing semicolon typo will get a compilation error.

Variable-Argument Debug Macros

A neater solution is to use varargs preprocessor macros with the special tokens "`...`" and "`__VA_ARGS__`", which are standard in C and C++ (since 1999):

```
#define aussie_debug(fmt,...) \
     printf((fmt), __VA_ARGS__ )
...
aussie_debug("DEBUG: I am here!\n");
```

That's not especially helpful, so we can add more context:

```
// Version with file/line/function context
#define aussie_debug(fmt,...)  \
    ( printf("DEBUG [%s:%d:%s]: ", \
        __FILE__, __LINE__, __func__ ), \
    printf((fmt), __VA_ARGS__ ))
...
aussie_debug("I am here!\n");
```

This will report the source code filename, line number, and function name. Note the use of the comma operator between the two `printf` statements (whereas a semicolon would be a macro bug).

Also required are parentheses around the whole thing, and around each use of the "`fmt`" parameter.

Here's a final example that also detects if you forgot a newline in your format string (how kind!):

```
// Version with newline optional
#define aussie_debug(fmt,...)  \
    (printf("DEBUG [%s:%d:%s]: ", \
      __FILE__, __LINE__, __func__ ), \
    printf((fmt), __VA_ARGS__ ), \
    (strchr((fmt), '\n') != NULL \
            || printf("\n")))
...
aussie_debug("I am here!");  // Newline optional
```

Dynamic Debug Tracing Flag

Instead of using "`#if DEBUG`", it can be desirable to have the debug tracing dynamically controlled at runtime. This allows you to turn it on and off without a rebuild, such as via a command-line argument or inside a `cuda-gdb` session. And you can decide whether or not you want to ship it to production with the tracing available to be used. Your phone support staff would like to have an action to offer customers rather than "turn it off and on."

This idea of dynamic control of tracing can be controlled by a single Boolean flag:

```
extern bool g_aussie_debug_enabled;
```

We can add some macros to control it:

```
#define aussie_debug_off() \
  ( g_aussie_debug_enabled = false )
#define aussie_debug_on() \
  ( g_aussie_debug_enabled = true )
```

And then the basic debug tracing macros simply need to check it:

```
#define aussie_dbg(fmt,...) \
    ( g_aussie_debug_enabled && \
    printf((fmt), __VA_ARGS__ ))
```

So, this adds some runtime cost of testing a global flag every time this line of code is executed.

Here's the version with file, line, and function context:

```
#define aussie_dbg(fmt,...)  \
    ( g_aussie_debug_enabled && \
    ( printf("DEBUG [%s:%d:%s]: ", \
            __FILE__, __LINE__, __func__ ), \
    printf((fmt), __VA_ARGS__ )))
```

And here's the courtesy newline-optional version:

```
#define aussie_dbg(fmt,...)  \
    ( g_aussie_debug_enabled && \
    (printf("DEBUG [%s:%d:%s]: ", \
      __FILE__, __LINE__, __func__ ), \
    printf((fmt), __VA_ARGS__ ), \
    (strchr((fmt), '\n') != NULL \
        || printf("\n"))))
```

Device Code Dynamic Debugging

That all sounds great, except when you realize that device code can't just create a global flag. Accesses to the "`g_aussie_debug_enabled`" global variable inside a kernel are a compile error.

To use this idea in device code, you would have to do this:

```
__device__ bool g_aussie_debug_enabled;
```

Whenever this variable is accessed by device code in a debug macro, it triggers a global memory access, which is a very expensive access. An alternative would be to use `__constant__` to have the value in the "constant cache," which should be faster, but it's still slower than kernel local variables.

Furthermore, the mechanics of enabling or disabling this debug flag's value on the device based on a command-line argument in the host code are quite difficult. The host code can't just set the global variable on the device.

The performance cost of either `__device__` or `__constant__` may not be worth the value from the extra tracing flexibility. Another way that's fast, but requires code changes, is to pass a debug flag around as a parameter to kernel functions.

Alternatively, the simpler debug trace methods with `#if` can be used. Similarly, you could use these basic C++ constant styles:

```
#define g_aussie_debug_enabled true
const bool g_aussie_debug_enabled = true;
```

However, these methods now require a re-compile to change, so we haven't achieved the "dynamic debug tracing" that we wanted!

Multi-Statement Debug Trace Macro

An alternative method of using debugging statements is to use a special macro that allows any arbitrary statements. For example, debugging output statements can be written as:

```
DBG( printf("DEBUG: Entered print_list\n"); )
```

Or using C++ `iostream` output style:

```
DBG( std::cerr << "DEBUG: Entered print_list\n"; )
```

This allows use of multiple statements of debugging, with self-testing code coded as:

```
DBG( count++; )
DBG( if (count != count_elements(table)) { )
DBG(   aussie_internal_error("ERROR: Count wrong"); )
DBG( } )
```

But it's actually easier to add multiple lines of code or a whole block in many cases. An alternative use of `DBG` with multiple statements is valid, provided that the enclosed statements do not include any comma tokens (unless they are nested inside matching brackets). The presence of a comma would separate the tokens into two or more macro arguments for the preprocessor, and the `DBG` macro above requires only one parameter:

```
DBG(
  count++;
  if (count != count_elements(table)) { // self-test
      aussie_internal_error("ERROR: Count wrong");
    }
)
```

The multi-statement DBG macro is declared in a header file as:

```
#if DEBUG
#define DBG(token_list) token_list  // Risky
#else
#define DBG(token_list) // nothing
#endif
```

The above version of DBG is actually non-optimal for the macro error reasons already examined. A safer idea is to add surrounding braces and the "do-while(0)" trick to the DBG macro:

```
#if DEBUG
#define DBG(token_list) do { \
       token_list } while(0)   // Safer
#else
#define DBG(token_list)   ((void)0)
#endif
```

Note that this now requires a semicolon after every expansion of the DBG macro, whereas the earlier definition did not:

```
DBG( std::cerr << "Value of i is " << i << "\n"; );
```

Whenever debugging is enabled, the statements inside the DBG argument are activated, but when debugging is disabled they disappear completely. Thus, this method offers a very simple method of removing debugging code from the production version of a program, if you like that kind of thing.

This DBG macro may be considered poor style since it does not mimic any usual syntax. However, it is a neat and general method of introducing debugging statements, and is not limited to output statements.

Yet another alternative style is to declare the DBG macro so that it follows this statement block structure:

```
DBG {
    // debug statements
}
```

Refer to the implementation of a block "SELFTEST" macro in the prior chapter for details on how to do this.

Multiple Levels of Debug Tracing

Once you've used these debug methods for a while, you start to see that you get too much output. For a while, you're just commenting and uncommenting calls to the debug routines. A more sustainable solution in a large project is to add numeric levels of tracing, where a higher number gets more verbose.

To make this work well, we declare both a Boolean overall flag and a numeric level:

```
extern bool g_aussie_debug_enabled;
extern int g_aussie_debug_level;
```

As for running this in device code, the same provisos about global memory access on the GPU apply, except doubly so. This method is probably more likely to be considered for host code, and general application code running on the CPU.

Here's the macros to enable and disable the basic level:

```
#define aussie_debug_off()  ( \
    g_aussie_debug_enabled = false, \
    g_aussie_debug_level = 0)

#define aussie_debug_on()  ( \
    g_aussie_debug_enabled = true, \
    g_aussie_debug_level = 1 )
```

And here's the new macro that sets a numeric level of debug tracing (higher number means more verbose):

```
#define aussie_debug_set_level(lvl)  ( \
    g_aussie_debug_enabled = (((lvl) != 0)), \
    g_aussie_debug_level = (lvl) )
```

Here's what a basic debug macro looks like:

```
#define aussie_dbglevel(lvl,fmt,...)  ( \
    g_aussie_debug_enabled && \
    (lvl) <= g_aussie_debug_level && \
    printf((fmt), __VA_ARGS__ ))
...
aussie_dbglevel(1, "Hello world");
aussie_dbglevel(2, "More details");
```

Now we see the reason for having two global variables. In non-debug mode, the only cost is a single Boolean flag test, rather than a more costly integer "<" operation.

And for convenience we can add multiple macro name versions for different levels:

```
#define aussie_dbglevel1(fmt) \
    (aussie_debuglevel(1, (fmt)))
#define aussie_dbglevel2(fmt) \
    (aussie_debuglevel(2, (fmt)))
...
aussie_dbglevel1("Hello world");
aussie_dbglevel2("More details");
```

Device debug levels. As with the simpler debug flag earlier, controlling an integer setting for a dynamic debug level is difficult in device code. Options include:

```
__device__ int g_aussie_debug_level = 3;
__constant__ int g_aussie_debug_level = 3;
const int g_aussie_debug_level = 3;
#define g_aussie_debug_level 3
```

All of the above options are either relatively inefficient, or require a re-compile.

A workable solution is passing a debug level as a parameter to all kernel launches from the host, and between device function calls within the device code. The host code can get the debug level (e.g., from a command-line argument), and pass its debug setting to the device via kernel launches. This is a relatively efficient way to achieve a dynamic level of debug tracing. In this way, both you and your customers could run your application with different tracing levels, but without needing a different binary for each debug level.

Very volatile. Note that if you are altering debug tracing levels inside a symbolic debugger (e.g., `cuda-gdb`) or IDE debugger, you might want to consider declaring the global level variables with the "`volatile`" qualifier. This applies in this situation because their values can be changed (by you!) in a dynamic way that the optimizer cannot predict. On the other hand, you can skip this, as this issue won't affect production usage, and only rarely impacts your interactive debugging usage.

BYO debug printf: All of the above examples are quite fast in execution, but heavy in space usage. They will be adding a fair amount of executable code for each "`aussie_debug`" statement. I'm not sure that I really should care that much about the code size, but anyway, we could fix it easily by declaring our own variable-argument debug `printf`-like function.

Advanced Debug Tracing

The above ideas are far from being the end of the options for debug tracing. The finesses to using debug tracing messages include:

- Environment variable to enable debug messages.
- Command-line argument to enable them (and set the level).
- Configuration settings (e.g., changeable inside the GUI, or in a config file).
- Add unit tests running in trace mode (because sometimes debug tracing crashes!).
- Extend to multiple sets or named classes of debug messages, not just numeric levels, so you can trace different aspects of execution dynamically.

Supportability Tip: Think about customers and debug tracing messages: are there times when you want users to enable them? Usually, the answer is yes. Whenever a user has submitted an error report, you'd like the user to submit a run of the program with tracing enabled to help with reproducibility. Hence, consider what you want to tell customers about enabling tracing (if anything). Similarly, debug tracing messages could be useful to phone support staff in various ways to diagnose or resolve customer problems. Consider how a phone support person might help a customer to enable these messages.

17. CUDA Portability

Portability of CUDA C++ Applications

The portability model of CUDA programs to multiple architectures is quite complicated. Hence, let's start with the most basic point about CUDA:

Only NVIDIA GPUs are supported.

Beyond that, things get more complicated. There are two specific issues for the portability of your code:

1. Host code portability (CPU), and

2. Device code portability (GPU).

If you're trying to run an AI application in the data center, then it's probably running the host code on Linux and the device code on a H100 GPU.

But if you're using CUDA to write an application for gaming or video editing on a desktop PC, then the host code is running on an x86 CPU, and the GPU is a graphics card like a GeForce RTX 4090 or whatever is the latest chip as you read this.

Forget portability in AI! This simplest case is where you don't have to worry about any of this. And this is often the case for an AI workload, where you have control over all of the Linux machines with their eight-pack of H100's. You only need to compile for this one platform.

Hence, stop reading this section, because you don't care about portability: just compile it for your one platform and go to lunch.

Summary of Commands and API Calls

There's a lot of details in the discussion below, but let's do a quick summary of the things that you might need.

Here are some of the Linux commands you might use:

- `nvcc --version`
- `nvidia-smi`
- `whereis cuda`
- `which nvcc`

Here are some of the many `nvcc` compiler flags:

- `-g` or `--debug` — CUDA compiler flag for compilation in debug mode, with extra debug information put into the executable (i.e., similar to "`-g`" flag for GCC).
- `-G` or `--device-debug` — CUDA compiler option for "device debug" mode, when compiling CUDA C++ code that runs on the GPU.
- `-lineinfo` or `--generate-line-info` — NVCC generates extra information for profiling.
- `-pg` or `--profile` — generates profiler information for use with `gprof`.

Here are the CUDA C++ preprocessor macros defined during `nvcc` compilation, which mostly have a double underscore as both prefix and suffix:

- `__NVCC__` — predefined preprocessor macro when compiling in `nvcc`.
- `__CUDACC__` — another preprocessor macro when compiling CUDA C++.
- `CUDART_VERSION` — CUDA Runtime version as a number (preprocessor macro).
- `__CUDA_ARCH__` — GPU architecture preprocessor macro as a constant number (but be warned that this works in device code only and is undefined in host code).
- `__CUDACC_DEBUG__` — preprocessor macro set when compiling in debug mode.

Here are the CUDA Runtime C++ API calls:

- `cudaRuntimeGetVersion` — CUDA Runtime version (C++ function call).
- `cuDeviceGetAttribute` — get attributes of the current GPU device.
- `cudaGetDeviceCount` — how many GPUs on this box?
- `cudaGetDeviceProperties` — get properties of the current GPU.
- `cudaSetDevice` — set the current GPU device, so you can query its properties.
- `cudaDriverGetVersion` — CUDA driver version details.

I won't be insulted if you stop reading now and hit Stack Overflow instead.

Detailed CUDA Portability

CUDA compilation model. Multiple platforms are more complicated to support. The compilation model in CUDA has support for several types of files:

- Executable files (e.g., Linux executables)
- Binary files ("`.cubin`")
- PTX assembly files
- Non-CUDA C++ source files
- CUDA C++ files ("`.cu`")

I've mixed some host and device code issues together here, but I don't feel bad because that's what CUDA does inside its C++ programs. Anyway, let's split it out.

Host code portability. The host code is like a normal non-CUDA C++ program. You need it to compile into a native binary, just as you would any other C++ program on Linux or Windows. The output from compiling host code is a native executable file (not a "`.cubin`" file).

The `nvcc` compiler can do this, but it's not really doing everything itself. Behind the scenes, it actually calls another non-CUDA C++ compiler, such as `g++` on Linux.

For the host code, `nvcc` generates an intermediate C++ format, with all the CUDA syntax removed (e.g., `__global__` and the "<<<...>>>" triple chevron syntax). Hence, `nvcc` acts like a cross compiler that outputs C++ as its target language.

Beyond this, the portability issues for getting the host code running on Linux versus Windows versus MacOS are the same types of concerns as for a non-CUDA C++ program. There are literally whole books on C++ portability, so we'll be here for a while if I get started.

Device code portability. Where CUDA really shines is its support for multiple GPU chips. I mean, only NVIDIA ones, but it's still great. You can use `nvcc` to output two low-level formats:

- CUDA binary files ("`.cubin`")
- PTX assembly language files

The binary files are specific to each GPU, and are machine code for the GPU chip. Hence, you cannot just copy a "`.cubin`" file from one to the other. You have to specify the target GPU architecture when you create a binary file.

To support multiple GPU types in your application, you've got two main options for your build process:

- Manage lots of ".cubin" files (not recommended), or
- Compile to PTX assembly language

PTX is a text-based assembly language format that's much lower level than C++. The PTX assembly language files are further compiled to binary code by the GPU's device driver. What this really means is that every GPU device driver contains an assembler, and does "just-in-time compilation" to create machine code from PTX (really, shouldn't it be called "just-in-time assembling"?). The command-line version of the PTX assembler is called `ptxas`.

Note that the PTX language is not fully compatible across all GPU architectures. There are some options that control which level of "compute compatibility" need to be supported in the output PTX files. Hence, this adds another wrinkle to the build process, although maybe you won't be using any of the less powerful GPUs.

And just to confuse matters, there's a third option call "just-in-time compilation" of C++. This is where you can actually distribute the device code's CUDA C++ source code to multiple GPUs, rather than using binary or PTX assembly files. The NVRTC library can compile CUDA C++ files to PTX on the fly, which can then be assembled to binary code by the GPU device driver.

Summary. Let's wrap up this portability discussion with an overview of the various options.

- One CPU, one GPU — just use `nvcc` to build Linux executables and "`.cubin`" device binary files.
- One CPU, many GPUs — compile to PTX, or to binary, or use just-in-time NVRTC C++ compilation.
- Many CPUs, many GPUs — my head hurts; let's outsource.

Detecting Host versus Device Code

The simplest way to separate host and device code is to use different functions. It's a basic separation with "`__global__`" or "`__device__`" for device functions, and either no specifier or "`__host__`" for host code.

An even purer method is to separate the host code into its own source code file. In some cases, you could even have the basic C++ functions for host code in a non-CUDA C++ source file, or even link in a simple C++ non-CUDA library (e.g., via `g++` options).

But none of that is CUDA style! After all, the "U" in CUDA means "Unified" and we're supposed to smash it all into one source file. Hence, if you want to do different things on the host and the device, you need to detect it in the C++ code itself.

Preprocessor macro method. Whether the code is run on the host or the device can be detected at compile-time. The simplest way is to use a preprocessor macro.

```
#if __CUDA_ARCH__
   // Device code
#else
   // Host code
#endif
```

Another alternative way is:

```
#ifdef __CUDA_ARCH__
   // Device code
#endif
```

And for host code:

```
#if !defined(__CUDA_ARCH__)
   // Host code
#endif
```

Build your own symbols. Maybe you want it to look clearer in the code?

```
#if IS_DEVICE_CODE
  // kernel
#else
  // host
#endif
```

To permit this, you can define your own macros to hide these details.

Note that this idea won't work in a header file:

```
#ifdef __CUDA_ARCH__    // Fails!
#define IS_DEVICE_CODE 1
#else
#define IS_DEVICE_CODE 0
#endif
```

This above idea fails because the value of `__CUDA_ARCH__` will be evaluated by `nvcc` within your header file, where it is always host code, and the macro will always be empty.

Instead, this should work in a header, by making the expansion of your macro happen later:

```
#define IS_DEVICE_CODE ( __CUDA_ARCH__ > 0) // Better
#define IS_HOST_CODE   ( __CUDA_ARCH__ == 0)
```

Note that these will work in preprocessor expressions (e.g., `#if`), but not at runtime in "`if`" tests, where a compilation error will result. The undefined value of the `__CUDA_ARCH__` macro name in host code defaults to zero in preprocessor conditional expressions, but not elsewhere in C++ statements.

Detecting GPU Architectures in Device C++

You can detect the "compute capability" of your NVIDIA GPU within device code using the "__CUDA_ARCH__" preprocessor macro. This macro is not set in host code, which can be used to distinguish host versus device code, as already discussed above.

The main use of this macro is to use different code for more capable GPUs. Here's an example of how to use faster code with a higher compute capability, but also have code for a lower one on an older GPU. An example of the compile-time method:

```
#if __CUDA_ARCH__ >= 800
   // Compute capability 8.0 and above
#else
   // Less capable GPU
#endif
```

Is CUDA Installed?

You can check on a Linux box whether the CUDA Toolkit software is installed in various ways. Here's a selection of commands you can use. First, you can just try to run the compiler:

```
nvcc
```

Here's the output:

```
nvcc fatal: No input files specified; use option
  --help for more information
```

Use the `whereis` command on Linux:

```
whereis cuda
```

The output is:

```
cuda: /usr/local/cuda
```

You can list the CUDA file directory:

```
ls /usr/local/cuda/
```

Here's the output file listing:

```
bin compute-sanitizer  extras  include  nvml  res    src
compat      doc        gds     lib64    nvvm  share  targets
```

If CUDA is not installed, you get an error with most of these commands. Simples.

Detecting CUDA Version

Using the `nvcc` compiler's version flag is one way:

```
nvcc --version
```

If you're running in Google Colab, you'll need to add a prefix "`!`" to the Cell command to run any of these Linux shell commands properly. The command in a new "+Code" cell is simply:

```
!nvcc --version
```

Here's the output I get, which shows "12.2" in various ways:

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
```

version.txt disappeared. According to the internet (i.e., Stack Overflow), the installed version of CUDA Runtime is stored in a text file on Linux:

```
cat /usr/local/cuda/version.txt
```

But it doesn't work. Although there is a directory `/usr/local/cuda/`, here's what I get on my Google Colab virtual box running CUDA 12:

```
cat: /usr/local/cuda/version.txt: No such file or
directory
```

So, it looks like `version.txt` is gone, at least by CUDA version 12.

nvidia-smi command. You cannot really also use the `nvidia-smi` command for this issue, because that is inspecting your GPU chip's capabilities, rather than the CUDA Toolkit software install.

The command is simply:

```
nvidia-smi
```

Here's the output:

```
Sun Sep 29 04:43:04 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   35C    P8              9W /  70W |      0MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

Although it says "CUDA Version 12.2" this is talking about the hardware, not software. It's not a reliable indicator of the software install, as it can refer to what version the GPU requires, rather than what's currently installed. (And wow, someone at NVIDIA really took some time to make the columns line up and pretty up the ASCII layout, because coding this is fiddly work!)

Mixing CUDA and Non-CUDA C++

There are times when you want to use the same shared C++ code in both CUDA and non-CUDA applications. How can you tell which type of compiler is running? One way is to detect preprocessor macros:

- `__NVCC__` is set when `nvcc` is compiling.
- `__CUDACC__` is also set when a CUDA C++ compiler is compiling (i.e., `nvcc`).
- `__CUDA_ARCH__` specifies host mode or various device architectures.

Hence, you can detect CUDA versus non-CUDA compilation via methods such as:

```
#ifdef __NVCC__
  // CUDA C++
#else
  // Non-CUDA C++
#endif
```

There are a number of other supporting preprocessor macros that can be used to further identify compiler versions:

- `__CUDACC_VER_MAJOR__` is the major version of the `nvcc` compiler.
- `__CUDACC_VER_MINOR__` is the minor version number.
- `__CUDACC_VER_BUILD__` is the build number.

There are also some preprocessor macros that indicate the "modes" that `nvcc` is compiling in:

- `__CUDACC_DEBUG__` for device-debug mode.
- `__CUDACC_RDC__` for relocatable device code mode.
- `__CUDACC_EWP__` for extensible whole program mode.

CUDA Portability Traps

There are a few traps in coding portable code:

- `__CUDA_ARCH__` is an undefined macro in host code.
- Preprocessor macros are not checked in C++

To the point about undefined C++ preprocessor macros, here's a bug:

```
#if NVCC  // BUG!
  // CUDA-only code
#endif
```

This is a typo of `__NVCC__`, but it's also a silent bug. C++ converts unknown symbols in `#if` expressions to 0, so this fails. Here's a little trick for your header file:

```
#define NVCC Maybe you meant __NVCC__ ?
#define __NVCC Maybe you meant __NVCC__ ?
#define __NVCC_ Maybe you meant __NVCC__ ?
#define _NVCC_ Maybe you meant __NVCC__ ?
#define NVCC__ Maybe you meant __NVCC__ ?
```

Now you'll get compiler errors if you typo them in an `#if` expression. Unfortunately, I don't have a trick for `#ifdef` or the `defined` operator, so this is still a silent bug:

```
#ifdef NVCC  // Wrong!
```

An alternative strategy would be to tolerate accidental typos of macro names by adding this in your header file:

```
#ifdef __NVCC__
#define NVCC __NVCC__
#define _NVCC_ __NVCC__
#endif
```

It might be easier to just use a `grep` command on your C++ source code files:

```
grep -r NVCC | grep -v __NVCC__
```

This doesn't actually catch all cases, such as mixing them, but it's probably good enough. Alternatively, you can directly scan for all the badly written versions, using regular expressions to avoid matching the correct one.

You need multiple versions for each of the other processor macros, such as `__CUDA_ARCH__` as well:

```
grep -r CUDA_ARCH | grep -v __CUDA_ARCH__
grep -r CUDAARCH
```

And then you have to add it to your build scripts.

C++ Operator Portability Pitfalls

Most of the low-level arithmetic code in C++ algorithms looks quite standardized. Well, not so much. The general areas where C++ code that looks standard is actually non-portable includes trappy issues such as:

- Arithmetic overflow of integer or `float` operators.
- Integer `%` remainder and / division operators on negatives.
- Right bitshift operator >> on a negative signed integer is not division.
- Divide-by-zero doesn't always crash on all CPUs and GPUs.
- Order of evaluation of expression operands (e.g., with side-effects).
- Order of evaluation of function arguments.
- Functions that should be Boolean are not always (e.g., `isdigit`, `isalpha`)
- Functions that don't return well-defined results (e.g., `strcmp`, `memcmp`, etc.)
- Initialization order for `static` or global objects is undefined.
- `memcmp` is not an array equality test for non-basic types (e.g., structures).

Note that these errors are not only portability problems, but can arise in any C++ program. In particular, different levels of optimization in C++ compilers may cause different computations, leading to insidious bugs.

Signed right bitshift is not division

The shift operators << and >> are often used to replace multiplication by a power of 2 for a low-level optimization. However, it is dangerous to use >> on negative numbers. Right shift is not equivalent to division for negative values. Note that the problem does not arise for unsigned data types that are never negative, and for which shifting is always a division.

There are two separate issues involved in shifting signed types with negative values: firstly, that the compiler may choose two distinct methods of implementing >>, and secondly, that neither of these approaches is equivalent to division (although one approach is often equivalent). It is unspecified by the standard whether >> on negative values will:

(a) sign extend, or

(b) shift in zero bits.

Different compilers must choose one of these methods, document it, and use it for all applications of the >> operator. The use of shifting in zero bits is never equal to division for a negative number, since it shifts a zero bit into the sign bit, causing the result to be a nonnegative integer (dividing a negative number by two and getting a positive result is not division!). Shifting in zero bits is always used for unsigned types, which explains why right shifting on unsigned types is a division.

Divide and remainder on negative integers

Extreme care is needed when the integer division and remainder operators / and % are applied to negative values. Actually, no, forgot that, because you should never use division or remainder in a kernel, and if you must, then you choose a power-of-two and use bitwise operations instead. Division is unsigned right bitshift, and remainder is bitwise-and.

Anyway, another reason to avoid these operators occurs with negatives. Problems arise if a program assumes, for example, that -7/2 equals -3 (rather than -4) . The direction of truncation of the / operator is undefined if either operand is negative.

Order of Evaluation Errors

Humans would assume that expressions are evaluated left-to-right. However, in C++ the order of the evaluation of operands for most binary operators is not specified and is undefined behavior. This makes it possible for compilers to apply very good optimizing algorithms to the code. Unfortunately, it also leads to some problems that the programmer must be aware of.

To see the effect of side effects, consider the increment operator in the expression below. It is a dangerous side effect.

```
y = (x++) + (x * 2);
```

Because the order of evaluation of the addition operator is not specified, there are two orders in which the expression could actually be executed. The programmer's intended order is left-to-right:

```
temp = x++;
y = (temp) + (x * 2);
```

The other incorrect order is right-to-left:

```
temp = x * 2;
y = (x++) + (temp);
```

In the first case, the increment occurs before `x*2` is evaluated. In the second, the increment occurs after `x*2` has been evaluated. Obviously, the two interpretations give different results. This is a bug because it is undefined which order the compiler will choose.

Function-call side effects

If there are two function calls in the one expression, the order of the function calls can be important. For example, consider the code below:

```
x = f() + g()
```

Our first instinct is to assume a left-to-right evaluation of the "+" operator. If both functions produce output or both modify the same global variable, the result of the expression may depend on the order of evaluation of the "+" operator, which is undefined in C++.

Order of evaluation of assignment operator

Order of evaluation errors are a complicated problem. Most binary operators have unspecified order of evaluation — even the assignment operators. A simple assignment statement can be the cause of an error. This error can occur in assignment statements such as:

```
a[i] = i++;   // Bug
```

The problem here is that "`i`" has a side effect applied to it (i.e., ++), and is also used without a side effect. Because the order of evaluation of the = operator is unspecified in C++, it is undefined whether the increment side effect occurs before or after the evaluation of `i` in the array index.

Function-call arguments

Another form of the order of evaluation problem occurs because the order of the evaluation of arguments to a function call is not specified in C++. It is not necessarily left-to-right, as the programmer expects it to be. For example, consider the function call:

```
fn(a++, a);  // Bug
```

Which argument is evaluated first? Is the second argument the new or old value of *a*? It's actually undefined in C++.

Order of initialization of static objects

A special order of evaluation error exists because the order of initialization of static or global objects is not defined across files. Within a single file the ordering is the same as the textual appearance of the definitions. For example, the Chicken object is always initialized before the Egg object in the following code:

```
Chicken chicken; // Chicken comes first
Egg egg;
```

However, as for any declarations there is no specified left-to-right ordering for initialization of objects within a single declaration. Therefore, it is undefined which of `c1` or `c2` is initialized first in the code below:

```
Chicken c1, c2;
```

If the declarations of the global objects "chicken" and "egg" appear in different files that are linked together using independent compilation, it is undefined which will be constructed first.

memcmp cannot test array equality

For equality tests on many types of arrays, the memcmp function might seem an efficient way to test if two arrays are exactly equal. However, it only works in a few simple situations (e.g., arrays of int), and is buggy in several cases:

- Floating-point has two zeros (positive and negative zero), so it fails.
- Floating-point also has multiple numbers representing NaN (not-a-number).
- If there's any padding in the array, such as arrays of objects or structures.
- Bit-field data members may have undefined padding.

You can't skip a proper comparison by looking at the bytes.

Data Type Sizes

There are a variety of portability issues with the sizes of basic data types in C++. Some of the problems include:

- Fundamental data type byte sizes (e.g., how many bytes is an "int").
- Pointer versus integer sizes (e.g., do void pointers fit inside an int?).
- size_t is usually unsigned long, not unsigned int.

Typical AI engines work with 32-bit floating-point (float type). Note that for 32-bit integers you cannot assume that int is 32 bits, but must define a specific type. Furthermore, if you assume that short is 16-bit, int is 32-bit, and long is 64-bit, well, you'd be incorrect. Most platforms have 64-bit int types, and the C++ standard only requires relative sizes, such as that long is at least as big as int.

Your startup portability check should check that sizes are what you want:

```
// Test basic numeric sizes
aussie_assert(sizeof(int) == 4);
aussie_assert(sizeof(float) == 4);
aussie_assert(sizeof(short) == 2);
```

Another more efficient way is the compile-time `static_assert` method:

```
static_assert(sizeof(int) == 4);
static_assert(sizeof(float) == 4);
static_assert(sizeof(short) == 2);
```

And you should also print them out in a report, or to a log file, for supportability reasons. Here's a useful way with a macro that uses the "#" stringize preprocessor macro operator and also the standard "adjacent string concatenation" feature of C++.

```
#define PRINT_TYPE_SIZE(type) \
    printf("Config: sizeof " #type \
       " = %d bytes (%d bits)\n", \
    (int)sizeof(type), 8*(int)sizeof(type));
```

You can print out whatever types you need:

```
PRINT_TYPE_SIZE(int);
PRINT_TYPE_SIZE(float);
PRINT_TYPE_SIZE(short);
```

Here's the output on my Windows laptop with MSVS:

```
Config: sizeof int = 4 bytes (32 bits)
Config: sizeof float = 4 bytes (32 bits)
Config: sizeof short = 2 bytes (16 bits)
```

Standard Library Types: Other data types to consider are the builtin ones in the standards. I'm looking at you, `size_t` and `time_t`, and a few others that belong on Santa's naughty list. People often assume that `size_t` is the same as "`unsigned int`" but it's actually usually "`unsigned long`".

Here's a partial solution:

```
PRINT_TYPE_SIZE(size_t);
PRINT_TYPE_SIZE(clock_t);
PRINT_TYPE_SIZE(ptrdiff_t);
```

Data Representation Pitfalls

Portability of C++ to platforms also has data representation issues such as:

- Floating-point oddities (e.g., negative zero, `Inf`, and `NaN`).
- Whether "`char`" means "`signed char`" or "`unsigned char`"
- Endian-ness of integer byte storage (i.e., do you prefer "big endian" or "little endian"?).
- Whether zero bytes represent zero integers, zero floats, and null pointers.

Zero is not always zero? You probably assume that a 4-byte integer containing "0" has all four individual bytes equal to zero. It seems completely reasonable, and is correct on many platforms, but not all. There's a theoretical portability problem on a few obscure platforms. There are computers where integer zero or floating-point 0.0 is not four zero bytes. If you want to check, here's a few lines of code for your platform portability self-check code at startup:

```
int i2 = 0;
unsigned char* cptr2 = (unsigned char*)&i2;
for (int i = 0; i < sizeof(int); i++) {
    assert(cptr2[i] == 0);
}
```

Are null pointers all-bytes-zero, too? Here's code to check `NULL` in a "`char*`" type:

```
// Test pointer NULL portability
char *ptr1 = NULL;
unsigned char* cptr3 = (unsigned char*)&ptr1;
for (int i = 0; i < sizeof(char*); i++) {
    assert(cptr3[i] == 0);
}
```

What about 0.0 in floating-point? You can test it explicitly with self-testing code:

```
// Test float zero portability
float f1 = 0.0f;
unsigned char* cptr4 = (unsigned char*)&f1;
for (int i = 0; i < sizeof(float); i++) {
    assert(cptr4[i] == 0);
}
```

It is important to include these tests in a portability self-test, because you're relying on this whenever you use `memset` or `calloc`.

Pointers versus Integer Sizes

You didn't hear this from me, but apparently you can store pointers in integers, and vice-versa, in C++ code. Weirdly, you can even get paid for doing this. But it only works if the byte sizes are big enough, and it's best to self-test this portability risk during program startup. What exactly you want to test depends on what you're (not) doing, but here's one example:

```
// Test LONGs can be stored in pointers
aussie_assert(sizeof(char*) >= sizeof(long));
aussie_assert(sizeof(void*) >= sizeof(long));
aussie_assert(sizeof(int*) >= sizeof(long));
// ... and more
```

Note that a better version in modern C++ would use "`static_assert`" to test these sizes at compile-time, with zero runtime cost.

```
static_assert(sizeof(char*) >= sizeof(long));
static_assert(sizeof(void*) >= sizeof(long));
static_assert(sizeof(int*) >= sizeof(long));
```

In this way, you can perfectly safely mix pointers and integers in a single variable. But don't tell the SOC compliance officer.

References

1. Horton, Mark, *Portable C Software,* Prentice Hall, 1990, https://www.amazon.com/Portable-Software-Mark-R-Horton/dp/0138680507.
2. Jaeschke, Rex, *Portability and the C Language*, Hayden Books, 1989, https://www.amazon.com/Portability-Language-Hayden-Books-library/dp/0672484285.
3. Lapin, J. E., *Portable C and UNIX System Programming*, Prentice Hall, 1987, https://www.amazon.com/Portable-Systems-Programming-Prentice-hall-Processing/dp/0136864945.
4. Rabinowitz, Henry, and SCHAAP, Chaim, *Portable C,* Prentice Hall, 1990, https://www.amazon.com/Portable-C-Prentice-Hall-Software/dp/0136859674.
5. David Spuler, March 2024, *Generative AI in C++*, https://www.amazon.com/Generative-AI-Coding-Transformers-LLMs-ebook/dp/B0CXJKCWX9/.

Appendix: CUDA Puzzles

Instructions: Here are some puzzles on CUDA C++ debugging for your full and total enjoyment, or to use for tormenting CUDA C++ job applicants. The choice is entirely yours to make!

Every one of these code sequences has a bug in them, and usually a serious one. Catch all the bugs if you can!

Mostly these are insidious run-time errors, but a few might get a helpful warning, or even a compile-time error. Note that `#include` lines have been removed from some for brevity, so that's not the answer! Also excluded are common things such as the definition of any idiomatic `CUDACHK` runtime error checking macros or other types of runtime error checking code, or self-testing unit test functions that add up vector elements. If a missing declaration is all you can find, keep looking!

CUDA Puzzle #1: Where's the Bug?

Puzzle Code: Here's the device kernel:

```
__global__ void aussie_add_vec_puzzle1(
      const float*v1,
      const float*v2,
      float* vout, int n
    )
{
    // Compute offset
    int id = threadIdx.x;
    if (id < n) { // Safety
        vout[id] = v1[id] + v2[id]; // Add one element
    }
}
```

And here's the host code that launches the kernel:

```
// Kernel launch sequence
int nthreads = 256;
int blocks = (n + nthreads - 1) / nthreads;
aussie_add_vec_puzzle1<<<blocks,nthreads>>>(dv1,dv2,dv3,n);
CUDACHK( cudaDeviceSynchronize() );
```

CUDA Puzzle #2: Where's the Bug?

Puzzle Code: Here is the device code:

```
__global__ void aussie_vec_scale_puzzle2(
       float* vout,
       int n,
       float scalar
    )
{
    // Compute offset
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < n) { // Safety
        vout[id] *= scalar;  // Scale element
    }
}
```

And here is the host code with kernel launch:

```
// Kernel launch sequence
float rec = 1.0f / divisor;
int nthreads = 2048;
int blks = (n + nthreads - 1) / nthreads;
aussie_vec_scale_puzzle2<<<blks,nthreads>>>(dv, n, rec);
CUDACHK( cudaDeviceSynchronize() );
```

CUDA Puzzle #3: Where's the Bug?

Puzzle Code: Here's the device code for a 2D kernel:

```
__global__ void matrix_add_puzzle3(
    float *m3, const float *m1, const float *m2,
    int nx, int ny)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int id = x + y * nx; // Linearize
    if (x < nx || y < nx) {  // Safety
        m3[id] = m1[id] + m2[id];
    }
}
```

CUDA Puzzle #4: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void matrix_add_safe_puzzle4(
     float *m3, const float *m1, const float *m2,
     int nx, int ny)
 {
    int x = blockIdx.x + blockDim.x + threadIdx.x;
    int y = blockIdx.y + blockDim.y + threadIdx.y;
    if (x < nx && y < ny) {  // Safety
        int id = x + y * nx; // Linearize
        m3[id] = m1[id] + m2[id];
    }
 }
```

CUDA Puzzle #5: Where's the Bug?

Puzzle Code:

```
__global__ void matrix_add_safe_puzzle5(
     float *m3, const float *m1, const float *m2,
     int nx, int ny)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.x + threadIdx.y;
    if (x < nx && y < ny) {  // Safety
        int id = x + y * nx; // Linearize
        m3[id] = m1[id] + m2[id];
    }
}
```

CUDA Puzzle #6: Where's the Bug?

Puzzle Code: Here's the code for the 2D kernel:

```
 __global__ void matrix_add_safe_puzzle6(
    float *m3, const float *m1, const float *m2,
    int nx, int ny)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   if (x < nx && y < ny) {  // Safety
       int id = x + y * ny; // Linearize
       m3[id] = m1[id] + m2[id];
   }
}
```

CUDA Puzzle #7: Where's the Bug?

Puzzle Code: Here's the device code:

```
__global__ void aussie_clearvec_puzzle7(
      float* v, int n )
{
    // Compute offset
    int id = blockIdx.x* blockDim.x + threadIdx.x;
    if (id < n) { // Safety
        v[id] = 0.0;  // Clear element
    }
}
```

And here's the host code with the kernel launch:

```
// Kernel launch sequence
int nthreads = 32;
int blocks = 1;
aussie_clearvec_puzzle7 <<< blocks, n>>>(dv, n);
CUDACHK( cudaDeviceSynchronize() );
```

CUDA Puzzle #8: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
 __global__ void matrix_add_safe_puzzle8(
    float *m3, const float *m1, const float *m2,
    int nx, int ny)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   if (x < nx /*X* / && y < nx /*Y*/ ) {
       int id = x + y * nx; // Linearize
       m3[id] = m1[id] + m2[id];
   }
}
```

CUDA Puzzle #9: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void matrix_hadamard_safe_puzzle9(
    float *m3, const float *m1, const float *m2,
    int nx, int ny)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   if (x < nx /*X*/ && y /*Y*/ ) {  // Safety
       int id = x + y * nx; // Linearize
       m3[id] = m2[id] * m1[id];
   }
}
```

CUDA Puzzle #10: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void matrix_diff_safe_puzzle10(
    float *m3, const float *m1, const float *m2,
    int nx, int ny)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   if (x >= nx /*X*/ && y >= ny /*Y*/ ) {
       int id = x + y * nx; // Linearize
       m3[id] = m1[id] - m2[id];
   }
}
```

CUDA Puzzle #11: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void matrix_add_safe_puzzle11(
    float *m3, const float *m1, const float *m2,
    int nx, int ny)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   if (x < nx /*X*/ & y < ny /*Y*/ ) {
       int id = x + y * nx; // Linearize
       m3[id] = m1[id] + m2[id];
   }
}
```

CUDA Puzzle #12: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void matrix_clear_safe_puzzle12(
    float *m, int nx, int ny)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   if (x < nx && y < ny) {
       m[x][y] = 0.0;
   }
}
```

CUDA Puzzle #13: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void aussie_add_vector_puzzle13(
      const float*v1,
      const float*v2,
      float* vout,
      int n
    )
{
    // Compute offset
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id <= n) { // Safety
      vout[id] = v1[id] + v2[id]; // Add one element
    }
}
```

CUDA Puzzle #14: Where's the Bug?

Puzzle Code: Here's the GPU kernel:

```
__global__ void aussie_add_vector_puzzle14(
      const float*v1,
      const float*v2,
      float* vout,
      int n
    )
{
    // Compute offset
    int id = blockIdx.x * blockDim.x * threadIdx.x;
    if (id <= n) { // Safety
        vout[id] = v1[id] + v2[id];  // Add element
    }
}
```

CUDA Puzzle #15: Where's the Bug?

Puzzle Code: Here's the kernel device code for the GPU:

```
__global__ void aussie_addvec_puzz15(
        const float*v1,
        const float*v2,
        float* vout, int n
      )
  {
      // Compute offset
      int lane = threadIdx.x & 1F;
      int id = blockIdx.x * blockDim.x + lane;
      if (id <= n) { // Safety
          vout[id] = v1[id] + v2[id];  // Add element
      }
  }
```

And here's the kernel launch code:

```
// Kernel launch sequence
int nthreads = 32;
int blocks = (n + nthreads - 1) / nthreads;
aussie_addvec_puzz15<<<blocks,nthreads>>>(dv1,dv2,dv3,n);
CUDACHK( cudaDeviceSynchronize() );
```

CUDA Puzzle #16: Where's the Bug?

Puzzle Code: Here's the kernel device code for the GPU:

```
  __global__ void aussie_clear_puzzle16 (
        float* v, int n)
  {
      // Compute offset
      int id = blockIdx.x * blockDim.x + threadIdx.x;
      if (! (id >= n)) { // Safety
          v[id] = 0.0;  // Clear element
      }
  }
```

And here's the kernel launch code:

```
  int nthreads = (1<<6);
  int blocks = n + nthreads - 1 / nthreads;
  aussie_clear_puzzle16<<<blocks, nthreads>>> (dv, n);
```

CUDA Puzzle #17: Where's the Bug?

Puzzle Code: Here's the kernel device code for the GPU:

```
__global__ void aussie_clearvec_puzzle17(
      float* v, int n) {
    // Compute offset
    int id = blockIdx.x* blockDim.x + threadIdx.x;
    if (id < n)  // Safety
        v[id] = 0.0;  // Clear element
    }
}
```

And here's the kernel launch:

```
int nthreads = 27;
int blocks = (n + nthreads - 1) / nthreads;
aussie_clearvec_puzzle17<<<blocks,nthreads>>> (dv, n);
```

CUDA Puzzle #18: Where's the Bug?

Puzzle Code: Here's some GPU code:

```
__global__ void aussie_clear_puzzle18(
      float* vout, int n)
{
    // Compute offset
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < n) { // Safety
        vout[id] = 0.0;  // Clear element
    }
}
```

And here's the kernel launch:

```
int nthreads = 032;
int blocks = (n + nthreads - 1) / nthreads;
aussie_clear_puzzle18 <<<blocks, nthreads>>>(dv, n);
```

CUDA Puzzle #19: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void aussie_clear_vector_puzzle19(
      char* v,
      int n
    )
{
    // Compute offset
    int id = blockIdx.x* blockDim.x + threadIdx.x;
    if (id < n) { // Safety
        v[id] = 0.0;  // Clear element
    }
}
```

CUDA Puzzle #20: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void aussie_clrv_puzzle20(
      float* v,
      int n
    )
{
    int id = blockIdx.x* blockDim.x + threadIdx.x;
    if (id < n) v[id] = 0;
}
```

This is the host code:

```
int nthreads = 64;
int blocks = (n + (nthreads - 1)) / nthreads;
aussie_clrv_puzzle20<<<blocks, nthreads>>>(dv, sz);
```

CUDA Puzzle #21: Where's the Bug?

Puzzle Code: Here's the kernel:

```
__global__ void aussie_clrv_puzzle21(
      float* v,
      int n
    )
{
    int id = blockIdx.x* blockDim.x + threadIdx.x;
    if (id < n) { // Safety
        v[id] = 0.0;  // Clear element
    }
    else {
        assert(id >= n);
    }
}
```

And here's the host C++ code:

```
int nthreads = 32;
int blocks = (n + nthreads - 1) / n;
aussie_clrv_puzzle21<<<blocks, nthreads>>> (dv, n);
```

CUDA Puzzle #22: Where's the Bug?

Puzzle Code: Here's the kernel in 2D:

```
__global void matrix_add_safe_puzzle22(
    float *m3, const float *m1, const float *m2,
    int nx, int ny)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   if (x < nx /*X*/ && y < ny /*Y*/ ) {  // Safety
       int id = x + y * nx; // Linearize
       m3[id] = m1[id] + m2[id];  // Add
   }
}
```

CUDA Puzzle #23: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void aussie_clearvec_puzzle23(
      float* v, int n)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    (id < n || (v[id] = 0.0));
}
```

CUDA Puzzle #24: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void aussie_clearvec_puzzle24(
      float* v, int n)
{
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    assert(id < n && (v[id] = 0.0));
}
```

CUDA Puzzle #25: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void aussie_clearvec_puzzle25(
      float* v, int n )
{
    // Compute offset with lane
    int lane = threadIdx.x & 0x1F;
    int id = blockIdx.x * blockDim.x + lane;
    if (id < n) { // Safety
        v[id] = 0.0;  // Clear element
    }
}
```

CUDA Puzzle #26: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void aussie_clearvec_puzzle26(
      float* v, int n )
{
    // Compute offset with lane
    int lane = threadIdx.x & 0x1F;
    int id = blockIdx.x * blockIdx.x + lane;
    if (id < n) { // Safety
        v[id] = 0.0;  // Clear element
    }
}
```

And here's the kernel launch:

```
int nthreads = 32;
int blocks = (n + nthreads - 1) / nthreads;
aussie_clearvec_puzzle26<<<blocks, nthreads>>>(dv, n);
CUDACHK( cudaDeviceSynchronize() );
```

CUDA Puzzle #27: Where's the Bug?

Puzzle Code: The kernel code is:

```
__global__ void aussie_clearvec_puzzle27(
      float* v, int n )
{
    int lane = threadIdx.x & 0x1F;
    int id = blockIdx.x * blockDim.x + lane;
    assert(id < n);
    v[id] == 0.0;
}
```

And here's the kernel launch:

```
// Kernel launch sequence
int n = 256*32;  // multiple of 32
int nthreads = 32;
int blocks = (n + nthreads - 1) / nthreads;
aussie_clearvec_puzzle27<<<blocks, nthreads>>>(dv, n);
CUDACHK( cudaDeviceSynchronize() );
```

CUDA Puzzle #28: Where's the Bug?

Puzzle Code: Here's the GPU kernel:

```
__global__ void matrix_add_safe_puzzle28(
    float *m3, const float *m1, const float *m2,
    int nx, int ny)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   if (x < nx /*X*/ && x < ny /*Y*/ ) {  // Safety
       int id = x + y * nx; // Linearize
       m3[id] = m1[id] + m2[id];
   }
}
```

CUDA Puzzle #29: Where's the Bug?

Puzzle Code: The kernel code is:

```
__global__ void aussie_clearvec_puzzle29(
      float* v,
      int n
    )
{
    // Compute offset using threadIdx
    int id = blockIdx.x* blockDim.x + threadIdx.x;
    if (id <=> n) { // Safety
        v[id] = 0.0;  // Clear element
    }
}
```

CUDA Puzzle #30: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void aussie_clearvec_puzzle30(
      float* v, int n )
{
    int id = blockIdx.x* blockDim.x + threadIdx.x;
    if (id < n) { // Safety
        v[id] = 0.0;  // Clear element
    }
}
```

And here's the kernel launch code:

```
int n = 1>>12;  // multiple of 32
int nthreads = 32;
int blocks = (n + nthreads - 1) / nthreads;
aussie_clearvec_puzzle30<<<blocks, nthreads>>>(dv, n);
CUDACHK( cudaDeviceSynchronize() );
```

CUDA Puzzle #31: Where's the Bug?

Puzzle Code:

```
#define BITS 5

__global__ void aussie_clearvec_puzzle31(
      float* v, int n )
{
    assert(blockDim.x == 32);
    int id = blockIdx.x << BITS + threadIdx.x;
    if (x < n) { // Safety
        v[id] = 0.0; // Clear element
    }
}
```

And here's the launch code:

```
// Kernel launch sequence
int n = 1u << 15;  // multiple of 32
int nthreads = 1 << BITS;  // 32
int blocks = ( n + nthreads - 1) / nthreads;
aussie_clearvec_puzzle31<<<blocks, nthreads>>>(dv, n);
CUDACHK( cudaDeviceSynchronize() );
```

CUDA Puzzle #32: Where's the Bug?

Puzzle Code: Here's the kernel code:

```
__global__ void aussie_clearvec_puzzle32(
      float* v, int n )
{
    // Compute offset
    int id = blockIdx.x* blockDim.x
             + (threadIdx.x &0x1F);
    if (id < n) { // Safety
        v[id] = 0.0;  // Clear element
    }
}
```

And here's the launch code:

```
// Kernel launch sequence
int n = 1u << 15;  // multiple of 32
int nthreads = 32;
int blocks = ( n + nthreads - 1) / blocks;
aussie_clearvec_puzzle32<<<blocks, nthreads>>>(dv, n);
CUDACHK( cudaDeviceSynchronize() );
```

Answers

Answer #1: The kernel does not use `blockIdx` in the computation of the index, so it won't ever set the higher elements of a vector. This will only add vector elements 0..31, probably many times over in parallel across different blocks and warps. The kernel will not crash, but won't work correctly for vectors with more than 32 elements.

Answer #2: The block size in `nthreads` is 2048, but more than 1024 threads exceeds the limits allowed for block size. Hence, the kernel will fail to launch, with a synchronous failure.

Only part marks if you thought the only problem was that the `divisor` reciprocal calculation was not protected against divide-by-zero errors. The `blocks` calculation should really be capped at a maximum, too, as this code will exceed maximum limits for very large `n` values. But it won't work with capped blocks because there's no loop in the kernel. Does the code need more comments?

Answer #3: The `||` operator should be `&&`. The safety test is not very safe.

Answer #4: The index computation should use multiplication, `blockIdx.x * blockDim.x`, not addition (+).

Answer #5: Typo. One of the `blockDim.x` should be `blockDim.y`.

Answer #6: Typo. `ny` should be `nx` in the `id` calculation. Works fine if it's a square matrix!

Answer #7: The launch uses `n` as the block size, rather than `nthreads`. This will only work for vectors of sizes up to 1024. If `n` is ever larger, there will be more than 1024 threads, the hard limit on block sizes for a GPU. Hence, the kernel will fail to launch with a synchronous error.

Answer #8: There's a nested comment problem that will comment-out the "`y < nx`" test, because there's a space between "`*`" and "`/`". You'd probably get a compiler warning, and hopefully you pay attention to them!

Answer #9: Should be "`&& y < ny`" not just "`&& y`".

Answer #10: The two Boolean safety tests have the reverse condition with >= operators, and the kernel will only do invalid assignments. If this is called only with correct indices by correctly grid dimensions, it will simply do nothing.

Answer #11: Should be "`&&`" (logical-and operator) not "`&`" (bitwise-and operator), with lots of operator precedence problems occurring in the `if` test. It would still work if you added enough parentheses.

Answer #12: Two-dimensional array syntax `v[x][y]` won't work on a linearized array. The computation of the linearized index is also missing. Fortunately, this should be a compiler error, albeit a confusing one.

Answer #13: Safety test is off-by-one, and should be "`id < n`" not "`id <= n`".

Answer #14: Should be "`+ threadIdx.x`" (addition) not "`*`" (multiplication).

Answer #15: The constant "`1F`" is accidentally a `float` constant (`1.0`), but should be "`0x1F`" (hexadecimal integer). Hence, it has the wrong value, and does bitwise-and on a `float` type, which is a compile error (luckily!). Note that the use of `lane` in this way is dubious (should use `threadIdx.x`), but it's not a bug here because `nthreads` is only 32.

Answer #16: Could you do `1<<6` in your head? But that's not the bug. There's missing parentheses in the calculation of `blocks`. The code should be "`(n + nthreads - 1) / nthreads`" calculation.

Answer #17: Surely, this is an easy one. The number of threads per block should be a multiple of 32, not 27.

Answer #18: The initializer for `nthreads` is `032`, which is an octal constant in C++, and does not equal decimal `32`.

Answer #19: The device parameter should be "`float*`" not "`char*`". You'd get a compiler error, but then, without thinking about it much, you might just add a pointer cast to the argument, right?

Answer #20: Yes, it's the wrong zero, but that won't crash it. Kernel parameter "`sz`" should be "`n`" and that will probably crash. Presumably, `sz` is the byte size used for memory allocation and equals `n*sizeof(float)`, which is too large. The kernel could overflow its array bounds if `n` is not a clean multiple of 64, because the safety test has a threshold that's four times too high.

Answer #21: The assertion is wrong, but is harmless. The part that isn't harmless is that the calculation of "`blocks`" mixes up "`nthreads`" and "`n`" in the divisor. The value for `blocks` will always be 1.

Answer #22: Specifier "`__global`" is invalid and should be "`__global__`" (with suffix underscores). It's a harmless problem as there's a compiler error to remind you.

Answer #23: It's a tricky try, aiming to use an expression instead of a safety `if` test. Perhaps the idea is to avoid `if` statements for branch coherence? However, the short-circuiting of the "`||`" operator is the wrong logic, with its "or else" meaning. The assignment operator only executes invalid assignments. This idea would work for "`&&`" (with "and then" logic), but it wouldn't really do anything to change branch divergence anyway (if that was the intention, rather than just showing off fancy coding skills).

Answer #24: The assertion always fails because the second operand is equal to zero. Furthermore, the kernel will do nothing if assertions are ever "compiled out" for production mode. Also, this will return an error code for the whole kernel code if even one of the threads fails the assertion, so it's not really a good way to combine the safety test with assertions.

Answer #25: If the block size is more than 32 threads, this will miss data for the threads with a higher thread index, because "`lane`" is always 0..31 here. Portions of the vector won't be processed.

Answer #26: The second "`blockIdx`" should be "`blockDim`." This use of `lane` is dubious, but works here for a block size of 32.

Answer #27: No, the assertion should not fail. But the "==" operator on `v[id]` should be "=". It's a null-effect statement, not an assignment, and should get a compiler warning.

Answer #28: The "`if`" condition is testing "`x`" twice.

Answer #29: The spaceship operator "<=>" (three-way comparison) should be just "<". But this is valid in modern C++ and should run.

Answer #30: "`1>>12`" is zero. Should be "`1<<12`" presumably.

Answer #31: Operator precedence error. Here, `blockDim.x` is 32, and `x<<5` would be the same as `x*32`, but the << operator has a lower precedence

than the + operator, whereas multiplication has higher precedence. Parentheses are needed around "`blockIdx.x << 5`".

Answer #32: Typo. The variable `blocks` is actually used in its own initializer, which is an uninitialized use with undefined results and could be a divide-by-zero. In any case, it's an incorrect calculation for the number of blocks required.

Final Words: How did you go with a full warp of puzzles? Was it fun? Or was it fully warped? Remember that if you're ever having trouble debugging your CUDA kernels, make like an Aussie and turn your C++ code upside-down.