Using modern C++ to improve CUDA programs

By

MYTHREYA KURICHETI

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

John D. Owens, Chair

Jason Lowe-Power

Julian Panetta

Committee in Charge

2024

To my parents and my brother

CONTENTS

LIST OF FIGURES

List of Tables

ABSTRACT

Using modern C++ to improve CUDA programs

The classic style of writing and porting HPC applications to the GPU uses pointers to buffers or data-structures as kernel parameters. This style discards type information, leading to "flattening" of CPU-side data-structures before using them as kernel parameters, followed by a need to reconstruct them in GPU code to retain flexibility. In this thesis, we identify several major problems during the porting process, including lack of vectors or views into a GPU buffer, bounds checking, iterator support, macro-dependent function specialization on the GPU, and GPU allocators for arbitrary types. These are all features that are already supported by CUDA in kernel code, but programmers are generally unable to use them due to data-structures decaying to pointers in kernel invocations. We demonstrate these problems and present techniques to overcome them in an implementation in C++ and CUDA. We use modern C++ features to make CPU-side features (such as iterators, ranged-for loops, and bounds checking) first-class citizens in GPU kernel code while maintaining interoperability with existing libraries. The result is a new ability to use CPU-style coding patterns in GPU kernel code. We demonstrate that our abstractions generate equally good assembly as the classical implementations. As a case study, we use the library to simplify the porting process of accelerating a shallow-water simulation framework "HEC-RAS" to the GPU.

ACKNOWLEDGMENTS

I am extremely grateful to have been guided by Professor John Owens through my Master's programme and through the work that went into this thesis. Thanks to Professor John Owens for his kindness, patience, feedback, and support. His expertise and knowledge were incredibly valuable, and I am truly grateful for his guidance and mentorship.

I would like to thank Dr. Serban D. Porumbescu for his guidance and mentorship that helped me navigate academic, professional, and personal matters. I am really grateful for the extremely insightful discussions we had on these topics. I learnt a lot from the technical discussions we had with Dr. Matthew Drescher which helped me broaden my skills. I am also extremely thankful for the kindness and support offered by all the members of my research group, and for sharing their knowledge and expertise. I am also extremely thankful to all the members of the research group for their kindness, support offered, and for sharing their knowledge and expertise.

I would like to thank Professor Jason Lowe-Power and Professor Julian Panetta for their guidance, support, and excellent feedback on my thesis.

I would also like to thank the Computer Science administrative staff, especially Jessica Stoller, Alyssa Bates, and Lorena Galvan for helping me navigate academic requirements and responsibilities.

I would like to thank my friends for their incredible support, advice, and encouragement through the years.

Chapter 1

Introduction

Over the past few years, accelerators such as GPUs and ASICs have been developed to accelerate highly parallel workloads or specific applications such as matrix multiplication. These workloads have historically been implemented in C, C++, and Fortran and run on the CPU. Some parts of the data processing in these programs are parallel, and utilizing processors such as GPUs for those computations can provide a considerable gain in performance.

1.1 The Problem

GPUs are highly parallel co-processors designed for parallel data processing. They are driven by the CPU (the host) and are programmed using tools provided by the hardware vendor. GPUs (the device) have thousands of cores, significantly more than a CPU, and have fast device-local memory separate from the CPU. The GPU scheduler parallelizes GPU programs (kernels) by executing the same program across all threads, whereas CPU programs must explicitly spawn threads. Due to these architectural differences, we cannot use host-specific data-structures and functions in kernels without modifications.

Hardware vendors provide tools, APIs, and constructs within the confines of the language (or an extension of it) to let the user move data and specify the execution spaces in which a function may be used. The compiler uses this information to generate appropriate code for the host and device. Applications running on the CPU can initiate work on the GPU by using these APIs. Since host-local data is inaccessible from the GPU (under most circumstances [41]), we must use these APIs to copy data. This can lead to code duplication when replicating data-structures

across host and device, managing raw pointers, and manual copies to move data between execution spaces—all of which add to the maintenance cost of the application. Therefore, writing applications that target the GPU is not as straightforward as writing CPU programs under most circumstances due to the heterogeneity.

As a concrete example (which we further expand upon in chapter 3), when using CUDA with C++, users are often required to flatten their data-structures and have them decay to pointers or POD ("Plain Old Data") types that can be trivially copied to the GPU. As a consequence, standard C++ features such as iterators and range-based `for`-loops—which are, in fact, supported in host *and* device code by the compiler—cannot be used in device code. The fundamental problem is that we often have to reconstruct these higher-order types from the simplified or flattened-out types on the device to take advantage of these features.

All of these problems arise due to the existence of two different execution domains and their respective data spaces that are bridged minimally by the tools offered. This clear separation is in many ways an architectural decision by the manufacturer of the hardware that helps them establish a precise and well-defined programming and memory-model, which would otherwise be difficult and complicated to grasp if not for the "minimal bridge." This limits how expressive GPU programs can be. Our work focuses on improving the user experience when traversing this bridge between fundamentally different pieces of hardware. As a consequence of being driven by the CPU, we go through this "bridge" to interact with the GPU. This acts like a filter, limiting and guiding application architecture.

1.2 Prior work

Hardware vendors such as AMD and NVIDIA expose their APIs through C or C++. Apple exposes its hardware through the Metal API but uses Objective-C. We refer to these officially released tools by hardware vendors as "native." HPC applications are written in C, C++, and Fortran ("native languages"), so it stands to reason that the major vendors expose their hardware through APIs in these languages. These APIs are usually verbose, which allows the vendor to simplify their implementation and ensure minimal overhead. While this offers maximum control of the hardware, it is not user-friendly. Various tools have been developed to abstract

common use cases and simplify writing kernels. We will now take a look at common approaches for building such tools. We then present our reasoning as to why these tools do not entirely address the problem we describe.

1.2.1 Compilers

Programming APIs such as Vulkan, OpenGL, and DirectX allow the user to target the GPU, but programs (known as "shaders" in the computer graphics world) are written in a different language[1] (a domain-specific language—DSL) than the one used on the host. These shaders require separate compilation tools than the ones used for host code, and generally cannot be implemented in the same file as host code (the exception being the program is stored as string literal and is compiled at runtime). Compiler support is required to let the user implement shaders in the same language as the host implementation. For example, "Unified Shader Specialization" [46] taps into existing C++ features and adds custom attributes to the language by modifying the Clang compiler to express shader specialization, which is used to generate host and device code. CUDA takes a similar approach and extends the C++ language to support heterogeneous compilation from the same source file.

Similarly, for languages not supported by the hardware vendor, an approach is to implement a transpiler to the supported language (Hybridizer [1]) or a compiler (such as ILGPU [24] for C#, Rust-CUDA[2] for Rust) to generate GPU-native binaries. While in theory this allows them to accept any data-structure as an argument to a kernel, they generally accept only a limited subset offered by the language or the library itself (`ArrayView` in the case of ILGPU) to simplify the transformation. These supported structures are generally flat buffers, aggregates, or trivially-copyable value types. The compiler implementation limits the flexibility of this approach. A compiler cannot cover all possible use-cases. With adequate compiler support, we can implement libraries to overcome these limitations, which we discuss below.

[1] GLSL and HLSL are the most common ones, used by OpenGL and DirectX respectively.
[2] Rust-CUDA: https://github.com/Rust-GPU/Rust-CUDA

1.2.2 Libraries

Libraries such as oneAPI [21], SYCL,[3] and Thrust [4] allow the user to program the GPU in C++. They expose a set of highly optimized GPU routines that can be composed by the user to achieve the functionality they desire. If custom functionality is required, the user can pass a functor or a lambda function encapsulating the desired behavior to the library, which is parallelized and executed on the GPU. Essentially, these libraries approach the problem by hiding the complexity behind their own APIs. The user writes code that expresses their *intent* at a high level, and the library uses the most optimal implementation based on this information. While this offers customization and safety—goals we also claim to achieve, it does not necessarily offer *complete control* over both the GPU and CPU implementations. Thrust and stdgpu [47] address some of the issues we present (iterators and vectors in kernels). stdgpu also provides GPU counterparts for common CPU data-structures such as a stack, queue, vector, and maps; one can use them in both host and device code.

These libraries are indispensable when writing GPU-accelerated code. They address some of the issues we present, offer a large set of performant algorithms and data-structures that work on the host and device, and should be used where appropriate. We present additional techniques and tools that can be used alongside these libraries to provide similar flexibility to user-defined types and make them easily accessible in kernels. We focus on constructs that allow the user to control *both* the host and device implementations, and the structures that bridge them while staying within the confines of tools offered by hardware vendors. We do not aim to replace these libraries; quite the opposite—we aim to complement them by improving the experience of traversing the "bridge" we previously mentioned.

1.3 Approach

We use language features such as compile-time processing, type introspection, macros, allocators, inheritance, and iterators to build functions that can be used in both host and device code. This also allows us to build data-structures that work on both host and device code, overcoming code duplication problems. These data-structures allow us to avoid the fragile and repetitive

[3] SYCL: https://www.khronos.org/sycl/. Strictly speaking, SYCL is both a library and a compiler solution.

"glue code" that typically exists solely to chauffeur data between host and device code.

1.4 Goals

A primary goal of the techniques presented is to let users write GPU kernels with the same expressiveness and feature set as CPU implementations without sacrificing performance. We present abstractions and tools that can be utilized generally. We focus on users who intend to write kernels, and aim to simplify their kernel programming experience. As such, we focus on abstractions that allow users to utilize the same data-structures on the CPU and the GPU, with the expectation that the user is aware of any potential disadvantages of not using GPU-specific data-structures where appropriate.

We operate under the assumption that HPC applications spend a significant portion of their execution time and compute resources in tight loops that implement their algorithms. Therefore, we strive to ensure that the parts of our abstractions that implement tight loops generate optimal code.

The traditional approach to measuring the performance of a new tool or an algorithm is to run benchmarks with different datasets and compare the results against contemporary state-of-the-art implementations. This approach does not offer insights into the overhead of the abstraction for all use-cases.

Our methodology will be to compare the machine code that is generated for device code with and without using our implementation. If the generated machine code for the section that implements the compute loop on device code is identical, we claim to have achieved our goal—minimal overhead. We do not compare host code generation as the techniques we present are already used by tools such as Thrust. Since we are interoperable with Thrust, it can be used for host code if desired.

To that extent, we bring safety-oriented constructs such as spans and vectors to the GPU. Additionally, we move runtime errors to compile-time ones where possible. We try to address the gaps in features left (intentionally or otherwise) by existing libraries. We intend to *complement* existing libraries, not replace them.

We are not interested in building a new compiler or modifying existing ones to achieve our

goals. We instead focus on taking advantage of features offered by the officially supported tools ("native tools and languages") to simplify kernel development.

1.5 Contributions

We identify problems one may face when porting CPU code to the GPU. In particular, we address issues such as:

1. Function specialization for GPU and CPU leading to difficulties with code reuse.
2. Macro guards polluting the code—they textually eliminate code segments and are not checked for syntactic validity.
3. Having to simplify and flatten CPU data-structures into pointers and their associated metadata (such as buffer size) before passing them to kernels.
4. Lack of convenience features such as iterators and range-based for loops on the GPU as a consequence of the previous issue.
5. Incorrect implementations of the "Grid-stride loop" [19] (see section 3.5) leading to incorrect computations or runtime errors.
6. Mixing up of host and device pointers, and their sizes when using them as kernel parameters.
7. Trying to dereference GPU-only memory from the CPU.

To address these problems, we present a set of abstractions and techniques that allow users to use features taken for granted in host code (such as iterators) in device code too. We present a library that implements these techniques to exemplify the ideas discussed using CUDA and C++. We then demonstrate how these techniques assisted us in porting a C#-based shallow-water simulation framework to the GPU while giving examples with and without using these abstractions.

1.6 Benefits

These techniques allow one to write code that reads like CPU code, reducing cognitive load on the programmer. We demonstrate that our techniques enable a user to:

1. Use spans, iterators, ranged for-loops in device code.
2. Pass a structure by reference or as a pointer into kernels.
3. Leverage the RAII (Resource Acquisition is Initialization) paradigm to enable automatic allocation and cleanup of GPU memory.
4. Benefit from bounds-checking in debug mode on both host and device.
5. Share code between GPU and CPU implementations without macros, enabling compile checks for all code paths.
6. Maintain interoperability with existing libraries such as Thrust.

These features improve the expressiveness, maintainability, and readability of code without sacrificing performance or functionality. Some of the benefits mentioned above are brought forth by libraries such as Thrust but leave some features to be desired (e.g., Thrust vectors cannot be used in kernels, and stdgpu is focused on offering common CPU data-structures on the GPU).

1.7 Outline

We begin with some background on GPUs and their programming-model in chapter 2. In chapter 3, we describe the techniques being proposed. To begin with, we explain how compile-time processing simplifies writing device code in section 3.1. This also ties in with `concept`s in C++ (described in section 3.2), which enable us to enforce proper API usage at compile-time. We then present a simplified allocator and custom `new` and `delete` operators in section 3.3, which can be reused to simplify the implementation of user abstractions. Using this abstraction, we present an example of a vector in section 3.4 over a collection of elements. The feature that sets this apart from existing implementations is that it can be used in both host and device

code. We also demonstrate how iterators (section 3.5) can be added to improve user experience. Where appropriate, we show that our abstractions generate optimal code. In chapter 4, we demonstrate how our utilities helped us port a C# CPU codebase to the GPU and share a few examples on how it improves user experience.

Chapter 2

Background

In this chapter, we describe the GPU, how it differs from the CPU, and look at existing programming methods. We will then discuss CUDA, and give a high-level description of the problem.

2.1 The GPU

The GPU is designed with different goals in mind from that of a CPU—GPUs trade latency for throughput. High throughput is achieved through multiple cores, many more than typically found in CPUs. These cores are simpler compared to their CPU counterparts, which enable more of them to be packed together. These cores are connected to device local memory (separate from the CPU RAM) with wide buses to keep the bandwidth and throughput high. On a CPU, parallelism is explicit—a user is expected to create and launch threads explicitly to utilize the parallelism offered on the CPU, if any. CPU threads are independent units of execution, relatively expensive to create, and can implement logic completely separate from the main execution path of the program.

GPUs are SIMT machines (single instruction, multiple threads) [45]. Programs executed on the GPU are implicitly parallel, and the user writes a piece of code, typically known as a shader or a kernel, that is executed on all the cores at once. GPU threads are cheap to create and typically operate on a single element of the data. Each thread independently executes the kernel. The GPU assigns a unique identifier to each thread, which can be used to index into the elements buffer and process them in parallel. Consequently, thousands of threads execute at once on a GPU, processing items in parallel. Fundamentally, the GPU is a co-processor at heart

and any programming-model designed for the GPU must interact with, be controlled from, and be coordinated by the CPU. Therefore, it must, as a part of its design, support a heterogeneous execution model.

In the following section, we describe the major programming models at a high-level in their order of appearance, and their limitations at a high-level.

2.2 Programming model

GPUs and other application-specific hardware are driven by a CPU. The fundamental goal of a programming-model that exposes co-processors to the CPU (and the user) is to support constructs that submit commands to it and move data between the CPU and the co-processor. The programming-model is supported through the tools, libraries and APIs provided by the hardware vendor. Three common ways to program such hardware are:

1. Domain specific languages (DSLs), such as GLSL and HLSL, with their own compilers that generate hardware-specific machine code,

2. Extension of an existing language (such as `brcc` which extends C, and CUDA which extends C++), or

3. Through compiler transformations (or a new compiler "backend") that generates hardware-specific code from existing languages (such as OpenMP[1]).

2.2.1 Shaders

Well known examples of DSLs are shaders, which are extensively used in computer graphics. In such scenarios, data-structures on the CPU cannot be used directly in shaders, as not only are the execution spaces different, their languages are too. As such, a user must replicate their data representations in shaders to match those on the CPU and call the appropriate API functions to inform the driver to "bind" them together. The advantage of this approach is that the compilers designed for such languages can be specifically designed and optimized for the hardware. They are also small enough to embed in the programs themselves, affording the user the ability to change and recompile shaders on the fly.

[1] OpenMP: The OpenMP API specification for parallel programming—https://www.openmp.org/specifications

But this approach has a major disadvantage—there is no type checking for types that are used across the host and device domains. The user is responsible for ensuring that the data types in the shader program and the CPU program have identical binary representations; this is not validated by the compiler. Incorrect binding of resources on either the shader or host code leads to runtime errors or data corruption. Users are required to implement a lot of "boilerplate" code on the host before running a kernel—initialize the rendering API, check for support for shader features and enable them, set up "bindings" to expose resources to shaders, handle the render loop, and compile shaders either ahead of time or on-time (depending on available features). All of this boilerplate code makes it cumbersome to quickly iterate and test new approaches and techniques, not to mention the effort required to debug these programs and shaders. To simplify programming, earliest GPGPU approaches used a custom compiler that transformed the user's code into shaders and a host program that launched the shader and retrieved the results.

For example, Ian Buck et al. in "Brook for GPUs: Stream Computing on Graphics Hardware" [5] use a source-to-source compiler `brcc` that generates C++ code, which invokes the shaders (either in DirectX or OpenGL). These are just some of the disadvantages [44] of using shaders and a graphics API to essentially trick the programming-model into performing computations that—one can argue—are completely divorced from the goals of a graphics API (rendering images to a framebuffer). Techniques such as using depth testing to discard work or for efficient branches were employed to improve the performance of early GPGPU applications. A user should not have to deal with the intricacies of a graphics API and shading language to implement applications that just want to utilize the parallelism in the hardware.

These efforts eventually led to the development of CUDA—it addresses many of the problems that plagued GPGPU programming using shaders and a graphics API. We discuss the advantages and disadvantages at a high-level in the next section (subsection 2.2.2), with some examples in section 2.3.

2.2.2 CUDA

CUDA was released by NVIDIA in 2007 [28] to expose the parallelism of the GPU much more readily to the programmer through a simpler programming-model [38]. It addresses many of the challenges of GPGPU programming, offering a single-source model that allows the user to

implement both host and device code in the same file, simple API calls to launch a kernel and wait for its completion, and next to no boilerplate code (as opposed to using a graphics API and shaders) to write an end-to-end saxpy application. CUDA is an API and programming-model that is an extension of the C++ language [32], allowing one to write CPU and GPU code in C and C++, within the same file—a *massive* simplification compared to the shader programming-model. While it is simpler by orders of magnitude to implement GPGPU applications in CUDA (when compared to using graphics APIs and shaders), it is still challenging to write a non-trivial application, which we address in section 2.3.

CUDA is primarily exposed as a C API and provides parallelization and synchronization primitives that expose the underlying hardware parallelism more readily to the programmer—fine-grained and coarse-grained parallelism constructs establish a clear hierarchy that makes it easy to reason about computations running in parallel. CUDA, being an extension of the C++ language, makes it easy to extend; new hardware and API features can just be exposed as part of the software stack.[2] This affords rapid evolution of the language and the programming-model, making it easy to extend and respond to the needs to the broader community. Since it is not based on a graphics API, it takes a simple API call to launch a kernel that is parallelized on the GPU. Host and device code can use the same data types without modifications under most circumstances—users no longer have to worry about ensuring that host and device data layouts are identical, without assistance from the compiler.

2.3 Problem

CUDA has addressed many of the challenges of using shaders for GPGPU programs, but programming in CUDA still presents a number of challenges, some of which we cover in this section to motivate the problem (with details in chapter 3). This thesis focuses on using the tools available in CUDA and C++ to simplify some of the common usage patterns and overcome their pitfalls—we are not interested in building a new tool (such as a new compiler), but wish to use existing language and API features to address these concerns.

CUDA is primarily exposed as a C API—all resources allocated must be explicitly managed

[2]While this is possible with shaders, they require an extension mechanism which requires the user to check for and enable the feature in both host code and shader code.

and freed appropriately. This pattern of allocating, using, and freeing resources is repetitive and error prone and could lead to memory leaks and hard-to-diagnose out-of-bounds accesses, some of which may lead to security issues [8]. Even when using Thrust (Listing 2.1) or similar libraries, the abstractions do not carry over into kernel code.

For example, data-structures such as `std::vector` in C++ cannot be used in kernel code due to various restrictions (enforced by the compiler—functions not explicitly annotated as callable from device cannot be used in device code), in no small part due to the hardware differences and device-local memory heaps. The code, driver, and the runtime environment that exposes the hardware to the user cannot account for all possible use-cases to enable transparent usage of CPU-native data-structures in device code.

This style of writing kernels leads to raw pointers and their sizes being passed around within device code, which discards type-information. Therefore, abstractions that exist on the CPU and are taken for granted—collections (`std::vector`, `std::span`), allocators (`std::allocator`), and iterators—cannot be directly used on the GPU.

Consequently, CUDA kernel parameters take raw pointers and their associated sizes. For example, if a kernel operates on two memory regions of different sizes, it takes four parameters—two pointers, and size of the memory region for each of the respective pointers. This means that convenience features such as iterators and ranged `for`-loops are not available in kernel code.

In this thesis, we present ways to implement similar abstractions that can be used across CPU and GPU code. While we use C++ as the language of choice since it's officially supported by NVIDIA, similar constructs can be utilized in other programming languages that support equivalent constructs, such as Rust.

```
1  __global__ void kernel(
2          Vertex* verts, int num_verts,
3          Edge* edges, int num_edges) {
4
5      while (/* process vertices and edges */) {
6          process(verts[i]); // no bounds checking
7      }
8
9      // no iterator support
10     for(auto& vert : verts) {  }
11 }
12
```

```
thrust::device_vector<Vertex> verts {};
thrust::device_vector<Edge>   edges {};

kernel<<<b, t>>>(
    thrust::raw_pointer_cast(verts.data()),
    verts.size(),
    thrust::raw_pointer_cast(edges.data()),
    edges.size());

void load_data(/**/) {
    // Load data into a CPU buffer
    std::vector<data> cpu_data {...};
    thrust::device_vector<data> gpu_data {};
    // Then meticulously copy it to the GPU buffer
    ...
}

__host__ __device__ auto func() {
    #ifdef __CUDA_ARCH__
        #if __CUDA_ARCH__ >= 700
            // ARCH-specific code
        #endif
    /* generic device logic */
    #else
    /* host logic */
    #endif
}
```

Listing 2.1: Motivating examples: Macro pollution, lack of iterators in device code, and the inability to manipulate data in a thrust `device_vector` from the host in a performant way.

Since CUDA is an extension of the C++ language, it supports most modern C++ features, even in kernel code. Despite this, a user cannot just pass in host data-structures into kernel code due to the aforementioned issues, leading to fewer opportunities where they can utilize all the features that C++ offers.

As a motivating example of the kinds of problems we seek to address, in Listing 2.1, the compiler does not warn us about swapped and incorrectly passed in parameters, if we swap the sizes. An equivalent C++ function would have accepted `std::vector<T>` as parameters for the respective types.

We bring forth and discuss more examples in chapter 3, where we present various sections that explain the problem and come up with ways to address the problem. In particular, we

- explain some modern C++ features that enable us to create a few building blocks that can be used in a generic fashion to improve CUDA programming.

- demonstrate how compile-time programming can be used to replace macros and simplify writing host and device-specific code (see Listing 2.1, line 30).
- present a way to allocate objects on device-local and managed-memory with the `new` keyword.
- show how a custom allocator backed by managed memory dramatically simplifies writing and loading data into GPU-native data-structures.
- demonstrate the need for owning and non-owning container types that can be used both on the host and in device code, and how it enables us to use iterators.
- explore coroutines and how they can be utilized to abstract concurrent kernel launches.

2.4 Prior work

There exist first-party (Thrust [4], libcu++ [29]) and third-party (cuda-api-wrappers [16]) libraries that simplify some of the resource management and also provide a set of highly optimized routines for common operations such as *reduce* and *scan*. But none of these libraries extend functionality into kernel code, i.e., when the user finally starts to write a CUDA kernel, all of the parameters decay into raw pointers and their associated sizes, and the kernel body drops down to C-like abstractions. For example, it is not possible to use a `thrust::device_vector` in kernel parameters. Programming forums such as StackOverflow[3] cite examples that outline a simple implementation of a vector that can be used in device code.

Libraries such as Thrust and RAJA [3, 23] typically follow a functional programming style, where operations such as loops are written as either lambda functions or functors. When using lambda functions, local variables have to be either copied or passed as references.

Our work takes a different approach, as described in section 1.3 and section 1.4. In the following chapter, for each section, we address a specific problem, present a solution, and conclude with considerations that users must be aware of.

[3] Stackoverflow: Using `std::vector` in device code: https://stackoverflow.com/a/45671310

Chapter 3

Implementation

In this chapter, we will take a look at the specific techniques proposed, examples of their implementation, and their potential drawbacks. We first describe the technique and language feature being used to assist with finding the appropriate equivalent in other languages. We then describe the problem being addressed and proceed to give an example implementation in C++ to demonstrate its utility. Finally, we conclude with potential problems that one might face when using these approaches, and potential alternatives where possible. In all of the examples that follow, `cup` is used as the top-level `namespace` for the set of utilities we propose.

3.1 `constexpr` and `consteval`

Some programming languages (such as C++) provide the ability to evaluate the result of a function call or an expression at compile-time. This allows one to avoid the runtime overhead, provided such an expression can be computed at compile-time. We can tap into this machinery to specialize parts of a function's implementation based on the compilation context—introspect the types of function arguments and the compilation target (host or device), and specify specialized implementations. In this section, we will take a look at how C++ `constexpr` and `consteval` can be used to our advantage to simplify code specialization and improve code maintainability and readability, Listing 3.1 (a) vs. (b).

```
__host__ __device__
auto func(/**/) {
  if constexpr ( cup::device_code() ) {
    // GPU code
    if constexpr ( cup::
    compiling_architectures(500) ) {
      // specialized code for compute_50
    }
  }
  else {
    // Host code
  }
}
```

(a) `if constexpr` usage

```
__host__ __device__
auto func() {
#if defined(__CUDA_ARCH__)
  // GPU code
  # if __CUDA_ARCH__ > 860
    // arch is compute_86 or newer, can use
     those features
  #endif
  // fallback to common features
# else
  // CPU code
#endif
}
```

(b) Using `__CUDA_ARCH__` macros

Listing 3.1: Using `if constexpr` allows one to express the compile-time decisions using constructs that are similar to classical control-flow statements, improving readability.

3.1.1 Introduction

There are instances when we would like to use the same function (when writing libraries consumed by others, for example) across both host and device, but specialize the implementation—a different codepath for host vs. device code—so that we take advantage of the hardware effectively. In other words, we want an abstracted function that does the right thing and takes advantage of the hardware transparently on behalf of the user. For example, both the GPU and CPU offer hardware-accelerated or highly-optimized math intrinsics for fused-multiply-add, trigonometric, and logarithmic functions. Utilizing these could dramatically speed up programs that are math-heavy in tight loops. A library author could then expose a single matrix-multiply routine that uses the GPU if possible, or falls back to a CPU implementation, making the API easy to use for the end user.

3.1.2 The problem

CUDA allows one to implement host and device code in the same source file. Since code generation for device is opt-in, it offers a way to inform the compiler of sections (of source code) where device code needs to be generated. In C++, the canonical way to achieve this is to check for the `__CUDA_ARCH__` macro [31] (Listing 3.1), or by using function decorators (`__host__` and `__device__`). In GPU code, the `__CUDA_ARCH__` macro is set to the current virtual architecture being compiled for and can be used to take advantage of modern

CUDA hardware features. While this works, we claim that we can do better due to the various issues that macros present. Macros textually eliminate sections of source code in the preprocessor stage, meaning that the eliminated section is not checked for syntactical correctness by the compiler.

Essentially, we are making a decision at *compile-time*—whether a section of code is processed by the compiler machinery responsible for host or device code generation—to implement host and device-specific code-paths. C and C++ macros do not introduce variable scoping. If the user does not explicitly introduce a new scope, all the variables in `func` are within the same scope. Failure to implement the code common for all architectures, as shown in the example below (Listing 3.1, line 7), can lead to hard-to-diagnose errors during runtime.

```
1  auto func() {
2    #if defined(__CUDA_ARCH__)
3    // GPU code
4      # if __CUDA_ARCH__ > 860
5      // arch is compute_86 or newer, can use those features
6      #endif
7      // fallback to common features
8    # else
9    // CPU code
10   #endif
11 }
```

Listing 3.1: Macros do not introduce a variable scope and the compiler is unable to check the code for syntactic validity in the textually-eliminated branches.

We want to use C++ language constructs—`if` statements in place of macros—to detect device and host code. This approach leads to code that reads like the rest of the code and clearly expresses the intent of the programmer. Before we explain the usage of such constructs, we will explain the language features that make this possible in subsection 3.1.3, an overview of how the CUDA compiler processes source files in subsection 3.1.4, and finally, in subsection 3.1.5, we use this understanding to implement constructs demonstrated in Listing 3.1.

3.1.3 Compile-time evaluation

`constexpr` and `consteval` are C++17 and C++20 features respectively that allow one to hint and require computations be performed at compile-time, respectively, rather than at runtime. To enable this feature on `nvcc`, use the command-line flag `--expt-relaxed-`

`constexpr` [34].

Since these computations can be performed at compile-time rather than at runtime, the compiler can evaluate the result of this expression during the compilation process and just store the computed value at each call-site.

```
template <typename T>
constexpr T square(T a) {
    return a * a;
}

__global__ void kernel(float* f) {
    *f = square(2.8);
}

int main() {
    auto res = square(3.14);
    printf("Res: %.3f", res);
}
```

Listing 3.2: Calls to `square` are usually eliminated in release mode, as the results of the call are computed at compile-time.

In Listing 3.2, an optimizing compiler completely eliminates the call to `square` in both host and device code.

3.1.4 NVCC compilation path

Both host and device code may be present in the same file, yet the compiler has to generate different instructions for each of them. To achieve this, the CUDA compiler (`nvcc`) splits host and device code into temporary files (since the functions are decorated with `__host__`, `__device__`, `__global__` attributes), and inserts code to facilitate the copy of function-call parameters from host to device code—from CPU to GPU. During this code-splitting process [33], the compiler defines `__CUDA_ARCH__` macro in device files but not in host files. This can be used to distinguish host and device compilation paths, as in Listing 3.1.

```
consteval bool device_code() {
    #ifdef __CUDA_ARCH__
    return true;
    #else
    return false;
    #endif
}
```

```
8
9  __host__ __device__
10 void func() {
11     if constexpr (device_code()) {
12         // GPU-specific code
13     }
14     else {
15         // CPU-specific code
16     }
17 }
```

Listing 3.3: `consteval` functions are *required* to be evaluated at compile-time. We can abstract the macros behind a helper function and use it instead.

3.1.5 Using `constexpr`

By wrapping this macro `__CUDA_ARCH__` into a utility function, we can clean up and replace most macro guards. This (Listing 3.3) allows one to use typical function-call syntax to write host and device-specific code. The advantage of this approach is that both branches are verified to be syntactically correct, since they are not eliminated by the preprocessor, leading to more robust code. This also introduces appropriate variable scoping, as opposed to variables being in the same scope when using macros.

An alternative approach is to explicitly define two different functions with the same name for host and device, but this feature is supported only on the clang compiler [6] (at the time of writing).

This idea can be extended further, using `if constexpr` to conditionally compile device code based on the architecture (Listing 3.4).

```
1  consteval bool compiling_architectures(int arch) {
2      constexpr std::array archs { __CUDA_ARCH_LIST__ };
3      return std::find(archs.begin(), archs.end(), arch)
4          != archs.end();
5  }
6
7  __device__
8  auto func() {
9    // ...
10   if constexpr (compiling_architectures(500)) {
11       // code for compute_50 virtual architecture
12   }
13 }
```

Listing 3.4: We can check at compile-time if we are compiling for a specific architecture using standard C++ code that isn't polluted by macros.

An alternate approach is to template the functions on the architecture so that the right function is picked.

```
1  #ifdef __CUDA_ARCH__
2  constexpr int DEVICE_ARCH { __CUDA_ARCH__ };
3  #else
4  constexpr int DEVICE_ARCH { 0 }; // host
5  #endif
6
7  template <int arch = DEVICE_ARCH>
8  __device__
9  void func(); // define this to use as fallback
10
11 template <>
12 __device__
13 void func<860>() {
14     printf("Arch 860 specific code!\n");
15 }
16
17 __global__ void kernel(float* f) {
18     // default template parameter picks specialization for
19     // current architecture
20     func();
21 }
```

Listing 3.5: Template specialization can be used to implement architecture-specific code, and a generic fallback can be implemented in the first definition.

We claim that these approaches are robust because given a list of device architectures to compile for, if the user forgets to implement any architecture-specific specialization, the compilation will fail. In Listing 3.5, a specialization for `compute_60` has not been defined. When attempting to compile for that architecture, we get a linker error due to an unresolved external to a mangled name (for example, `_Z4funcILi600EEvv`). To get a more user-friendly diagnostic, a `static_assert` (example in Listing 3.6) dependent on the template parameter can be used as the body of `func` (Listing 3.5, line 9). To use a fallback (features supported on all devices), it is sufficient to define `func` in Listing 3.5, line 9.

```
1  template <int arch = DEVICE_ARCH>
2  __device__
```

```
3 void func() {
4   static_assert(0 == arch, "Function not defined for this
      architecture");
5 }
```

Listing 3.6: Unused template functions are not instantiated. Their body can contain a `static_assert` to ensure architecture-specific code is always implemented.

Functions decorated with `constexpr` can be utilized in device code unmodified in most cases. For example, the `ratio` and `chrono` headers can be used unmodified in device code.

3.1.6 Problems with this approach

A potential disadvantage of this approach is that some device-only functions (such as math intrinsics) cannot be called in the device codepath, as they are syntactically invalid in the host codepath (Listing 3.7, line 4). A macro would textually eliminate these calls in the host codepath, but with the `if constexpr` approach, the compiler terminates compilation due to missing definitions. C++ allows the `false` branch of `if constexpr` to be discarded under certain conditions [13]. The usual workaround is to wrap these intrinsics behind functions (which internally use macros, limiting their scope), and use them instead in these code blocks.

```
1  __host__ __device__
2  float fma_round_down1(float x, float y, float z) {
3      if constexpr (cup::device_code()){
4          return ::__fmaf_rd(x, y, z);
5      }
6      else {
7          return x * y + z;
8      }
9  }
10
11 __host__ __device__
12 float fma_round_down2(float x, float y, float z) {
13     #ifdef __CUDA_ARCH__
14         return ::__fmaf_rd(x, y, z);
15     #else
16         return x * y + z;
17     #endif
18 }
19
20 // Error message from the compiler
21 main.cu: In function 'float fmamul_round_down1(float, float, float)':
22 main.cu:11:10: error: '::__fmaf_rd' has not been declared; did you
       mean '__fmaf64x'?
23     11 |          return ::__fmaf_rd(x, y, z);
24        |                  ^~~~~~~~~
```

```
25          |           __fmaf64x
```

Listing 3.7: `if constexpr` does not work with intrinsics since they are syntactically invalid in host code.

Replacing instances of `__CUDA_ARCH__` macro with compile-time checks is not always viable, as some device functions textually exist in the files generated in codepath for host, and vice-versa. In instances where replacement with this approach doesn't work, one can try to diagnose the issue by asking the compiler not to delete the intermediate generated files with the `--keep` parameter and inspecting them.

3.1.7 Summary

In this section, we

- described the disadvantages of using macros, and problems they cause;
- explored compiler features `constexpr` and `consteval` that let us make decisions at compile-time;
- described how this compiler machinery can be taken advantage of to replace most macro-based (e.g., `__CUDA_ARCH__`) host and device code specialization;
- contrasted the advantages of our approach with classic macro-based implementations (code is easier to read and maintain); and
- laid out the shortcomings of our approach (we may not be able to completely replace macros), and potential workarounds.

3.2 `concept`s usage

The ability to inspect types and make decisions based on those types is a powerful feature, but a library author cannot know all the types a user may define, making it restrictive. Specifically, the library author can implement logic based on a closed set of types, and once distributed, the set cannot be expanded by the user. An alternative approach is to specify the *behaviors* of the types, rather than the types themselves. Specifying the behavior of a type or the properties

allows one to operate on the behaviors offered by the types. For example, an integer is a type whose members can be ordered, whereas a type that represents complex numbers cannot.

By decoupling these behaviors from the types themselves, it is possible to implement decision-making on an open set of types. As such, a library author can specify that their implementation requires certain behaviors (say, *the members of the type can be ordered*) of the types passed in. The end user has to specify what behaviors their custom types affords. This allows any set of libraries and their types to interoperate. C++ 20 and above provide this through a language feature knows as concepts—a named set of requirements. Similar functionality is provided by Rust with the trait system.

3.2.1 Introduction

C++20 introduces a new language feature called concepts. A `concept` is a way to specify restrictions and requirements that are checked at compile-time. For example, Listing 3.9 is an example of a concept that enforces that a type `T` must support hashing. Without concepts, the compiler prints the full traceback that is often multiple pages long, earning C++ quite a notoriety around templates and error messages. For example, Listing 3.8 generates a wall of error messages during compilation that must be parsed through; sifting through them requires some familiarity with C++ diagnostics to identify the root cause of the issue.

```
1 struct S {
2     int p;
3     bool operator==(const S&) const = default;
4 };
5
6 int main() {
7     std::unordered_map<S, int> maps {};
8 }
```

Listing 3.8: An example that generates a wall of error messages.

```
1 template<typename T>
2 concept hashable = requires(T a) {
3     { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
4 };
5
6 auto func(hashable auto obj) {}
```

Listing 3.9: Concept to check if a type supports hash.

When an object that does not satisfy the `hashable` concept is passed as a parameter to `func` in Listing 3.9, the error message is much more descriptive and easily actionable. Apart from being more user-friendly, concepts also serve as an early-exit mechanism if a constraint is not satisfied [11].

3.2.2 The problem

Issues such as,

- compile-time error messages that are often hard to read and take action against; and
- common and easy-to-repeat errors (e.g., passing host pointers to device function, double free, leaking memory) that cause either undefined behavior or runtime crashes

reduce the developer's productivity. Time spent diagnosing these issues is better spent optimizing and implementing the application.

Succinct but descriptive error messages by the compiler, and the ability to catch more errors at compile-time improves the efficiency of the user. This can dramatically improve user experience as it allows them to stay in the tight test-implement iteration loop. Runtime errors must be explicitly tested for with manual or automated tests. For example, one of the most common issues that users face when porting or writing new kernels is passing in host pointers into kernel code. This leads to crashes at runtime. Ideally, we want to be able to detect this at compile-time where possible. We also want to improve compiler diagnostics to overcome the issue described in Listing 3.8.

Concepts make it easier to identify and fix errors during compilation, and decouple behaviors from the types themselves, making it easier to use across libraries. This mechanism allows us to convert some runtime errors (or the corresponding checks) to compile-time ones, which we also explore in this section.

3.2.3 Constraints for kernels

We can implement concepts that require kernel parameters and buffers to point to valid GPU memory, which prevents a user from launching a kernel with a buffer that points to CPU heap during compilation.[1] This implementation(e.g., Listing 3.11) requires that the allocators return typed pointers—for example, `dptr<float>` instead of `float*`. This involves three steps:

- Create a discriminator (typically an `enum`) to identify the memory type (host, device, pinned, etc.)—Listing 3.10.
- Create a user-defined type templated on the type of the pointer (e.g., `float`, `int`) and the discriminator.
- Wrap the native allocator in a function that returns the native pointer wrapped in the type created in the previous step.

We demonstrate how this technique can be implemented in C++ to illustrate its utility.

```
enum class pointer_loc : uint8_t {
    HOST,
    DEVICE,
    MANAGED,
    // PINNED,
    // other types
};
```

Listing 3.10: Discriminator using an enum

We can now tie these discriminators with the pointer types to create a new type that, in its type information, includes the kind of allocation. This is useful for compile-time decision-making. For example, we can use `device_ptr<float>` in place of `float*`—the latter does not have information on the *kind* of memory (host vs. device) it points to (Listings 3.11, 3.12). We can use this type information to implement rich decision-making at compile-time, and by implementing concepts that utilize this information, provide descriptive diagnostic messages (Listing 3.13).

```
template <typename T, pointer_loc T1 = pointer_loc::HOST>
struct pointer_t {
    static constexpr pointer_loc type { T1 };
    T* value { nullptr };

    [[nodiscard]]
    constexpr T* operator*() noexcept {
```

[1]The initial design by Dr. Ian Buck et al. in "Brook for GPUs: Stream Computing on Graphics Hardware" [5] exposes a way to use custom data types created from a superset of language-native primitives (such as `float2` alongside `float`, but is still limited by what C and the shading language allowed). This approach is flexible and allows a user to construct arbitrary types. We extend this approach by retaining additional metadata—source of a memory allocation—by using the C++ type-system to afford additional type-safety at compile-time.

```
8           return value;
9       }
10
11      [[nodiscard]]
12      constexpr T& operator[](size_t pos) noexcept {
13          return value[pos];
14      }
15  };
16
17  static_assert(sizeof(pointer_t<float>) == sizeof(float*));
18  static_assert(sizeof(pointer_t<int>) == sizeof(int*));
```

Listing 3.11: A new template type `pointer_t` that wraps the underlying pointer with a discriminator allows us to track the allocation type, which can be inspected later to implement custom routines.

Since this type info exists *only* at compile-time, this pointer-like type has zero runtime overhead—it is the same size as the underlying pointer it encapsulates. We can create handy aliases for commonly used types, such as `host_ptr<T>` (Listing 3.12, line 5). We now implement a concept that inspects the types for the constraints we want. For example, for a pointer to be addressable from the device, it must be one of device-local or managed (amongst others, such as zero-copy, and mapped host-pinned), see Listing 3.12, line 11.

```
1  template <typename T>
2  using device_ptr = pointer_t<T, pointer_loc::DEVICE>;
3
4  template <typename T>
5  using host_ptr = pointer_t<T, pointer_loc::HOST>;
6
7  template <typename T>
8  using managed_ptr = pointer_t<T, pointer_loc::MANAGED>;
9
10 template <typename T>
11 concept device_addressable_ptr = requires {
12    T::type;
13    T::type == pointer_loc::DEVICE || T::type == pointer_loc::MANAGED;
14 };
15
16 // using the concept
17 __global__
18 void kernel(device_addressable_ptr auto ptr) {
19     //
20 }
21
22 // alternate way to check for concept
23 template <typename T>
24 auto foo() {
```

```
25     if constexpr (device_addressable_ptr<T>) {
26         // T satisfies constraint
27     }
28 }
```

Listing 3.12: Aliases and Concepts that use the discriminator to enforce the requirement that only device-accessible types are passed as kernel parameters.

If the constraint is not satisfied, the compiler gives us a descriptive error message (Listing 3.13). Not all compilers currently support all the features of concepts as a consequence of it being a fairly new feature, but one can expect support in their compiler of choice in future releases. We present our examples with the `clang`, compiler which supports concepts.

```
1 file:loc: error: no matching function for call to 'kernel'
2     kernel<<<1, 1>>>(cpu_mem);
3     ^~~~~~~~~~~~~~~~~~~~~~~~~
4 file:loc: note: candidate template ignored: constraints not satisfied
      [with T = int *]
5 void kernel(T ptr) {
6      ^
7 file:loc: note: because 'int *' does not satisfy '
    device_addressable_ptr'
```

Listing 3.13: A user-friendly error message that points the user to the constraint that failed.

This approach moves a runtime error to a compile-time one, and clearly points the user to where and what the error is, making it more actionable. Concepts can also be used to ensure that kernels that call lambda functions pass in the correct parameters. This is especially useful for libraries such as Thrust that make use of functors and lambda functions to customize the functionality of their algorithms (Listing 3.14). Concepts can be used in conjunction with `if constexpr` to implement conditional checks in a more declarative way (Listing 3.12, line 25).

```
1 template<typename LambdaFunc, typename... Args>
2 requires std::invocable<LambdaFunc, Args...>
3 __global__ void launch(LambdaFunc func, Args... args) {
4     func(std::forward<Args>(args)...);
5 }
```

Listing 3.14: concepts paired with lambda functions can be used to ensure that the argument types are compatible.

3.2.4 Problems with this approach

Concepts take template parameters as arguments. Therefore, all the disadvantages of using templates carry over. They must be defined and implemented in headers. There are workarounds, such as forwarding the parameters to the implementation, to get the best of both worlds—modern compilers are exceptionally good at stripping away layers of abstractions. There may be an impact on compile times, but the "early-exit" mechanism, where the compiler terminates compilation as soon as the constraint is not satisfied, could help too—this depends on the application.

An alternate approach is to wrap kernel launches (just like the workaround above) and at runtime, check if the pointers point to valid GPU memory with `cudaPointerGetAttributes` when using the CUDA Unified Memory API. On some system configurations, it is legal to dereference memory allocated on the host (with `malloc`, for example) in device-code [41].

On such systems, this approach might nevertheless be useful as there is a performance impact when not using device-local memory—the memory is either "streamed" to the GPU incurring large latency, or may stall the kernel on startup as the memory pages are migrated to the GPU.

This approach requires that memory-allocation APIs and allocators return strongly-typed pointers that include additional type information (such as the heap location). Therefore, CUDA APIs cannot be used directly. One could argue that this approach isn't as flexible as initially claimed, as this implementation depends on a custom type (pointer with additional type information) we introduced—the claimed advantage of concepts. This is completely accurate. But first-party libraries (such as Thrust) are indispensable and used widely enough when writing most CUDA programs that such concepts can be built over its types without sacrificing cross-library portability. As we will demonstrate in section 3.3, a custom allocator that wraps CUDA APIs can be written that ties into this system, which can be integrated into tools that Thrust provides.

3.2.5 Summary

In this section, we

- briefly explored what C++20 concepts are, and how they simplify existing C++ code;
- presented a problem—how do we ensure that the user passes pointers that are valid in device code at compile-time instead of trying to validate them at runtime;
- described why this is useful—it saves time and allows the user to quickly iterate and ensure correctness without having to run the app and deal with runtime crashes;
- combined concepts with `constexpr`, `consteval`, and templates to ensure correctness at compile-time, with legible and easy-to-read error messages; and
- laid out the shortcomings and challenges that one may face when implementing or using this approach—cross-library portability, compiler support, and having to reimplement some of the internals to support this feature.

3.3 Allocator

Applications typically have varying memory requirements at runtime, and acquire memory with system calls. Operating systems expose these APIs through system headers. While these APIs allow one to allocate memory, they do not necessarily help with managing it. Some languages allow one to specify allocation behavior. We can tap into this functionality to allocate objects on various heaps—on both host and device. Advanced allocation strategies (such as arena allocation and memory pools) can also be implemented within these allocators to tune application performance.

Strictly speaking, allocators are not *necessarily* a language feature, but a technique or an abstraction that utilizes language features (destructors and constructors in C++, for example) to simplify and assist with memory management. Their feature-set may differ across languages, but the underlying idea is the same—a tool to assist the user with memory management without the pitfalls that often accompany handling raw pointers and manual memory management.

3.3.1 Introduction

C++ programmers typically follow a paradigm known as RAII (resource acquisition is initialization) [14] to manage memory allocations. RAII is a technique in which the initialization

of a variable is not distinct from acquiring the resources it manages (as opposed to creating a variable and then populating it with resources to manage). Initialization and acquisition of resources happens at once, allowing for the destructor to clean those resources up when the variable falls out of scope.

In this section, we describe how similar functionality can be implemented over CUDA memory management APIs. We can combine this with the "typed pointers" described in the previous section to allow for succinct code (Listing 3.15, and Listing 3.16 vs. Listing 3.17) that is not littered with `mallocs` and their corresponding `frees`—which must be in the reverse order of the allocations when dealing with nested data-structures to prevent leaks. For example, `std::unique_ptr` and `std::vector` are automatically cleaned up at the end of their scope. We want to use similar features and techniques to manage GPU memory, and improve interoperation with host code (e.g., a `std::vector` of integers backed by GPU-addressable memory). This could be ideal for porting legacy CPU-only applications to the GPU in steps.

In this section, we start by describing what allocators are, present a basic implementation of an allocator to understand the features it offers, and then we augment it with the features described in the previous section. We conclude with how they are useful to the user.

```
1  void load_data(const auto& file) {
2    // use device local memory (cudaMallocManaged)
3    std::vector<int, cup::managed_allocator<int>>  gpu_ints {};
4    gpu_ints.reserve(...);
5
6    while(file.has_more_data()) {
7      // read data from file
8      // construct GPU-native data-structures directly in
9      // GPU memory instead of copying it over from the CPU
10     gpu_ints.emplace_back(data);
11   }
12
13   // process data
14   kernel<<<...>>>(gpu_ints.data(), gpu_ints.size(), ...);
15
16   // gpu_ints deallocated at end of scope
17 }
```

Listing 3.15: What we want to achieve. The memory for the `vector`'s elements is backed by GPU memory and is cleaned up automatically. This abstraction is much simpler to follow than using raw CUDA API calls.

3.3.2 Allocators and containers

An allocator encapsulates strategies for access or addressing, allocation and deallocation, and construction and destruction of objects [9]. One of the template parameters of a C++ container is an allocator, which is used for allocating its members (for example, the items in a `std::vector`).

By following a set of guidelines, we can implement our own allocator and container to suit our needs [10]. In particular, an allocator must expose methods such as `allocate` and `deallocate`. We want to use similar features to manage GPU memory.

A container library is a generic collection of class templates and algorithms. Prominent examples of containers include `std::vector` and `std::array` [12]. Containers are not unique to C++. Other languages offer similar equivalents. A typical container exposes methods such as `begin`, `end`, and the indexing operator to access elements. They automatically handle resizing and reallocation, using an allocator to allocate and deallocate memory.

Libraries such as Thrust provide a set of primitives that the user can use to construct the functionality we demonstrate in this section. We add to it in that we use concepts (described in the previous section) to enhance and augment these constructs and features. Before we do so, we give an overview of how such abstractions work, to better understand the integration of user-implemented and language-provided machinery.

3.3.3 The problem

CUDA applications primarily allocate memory on the GPU through a rich set of API functions, of which `cudaMalloc`, `cudaMallocManaged`, and `cudaFree` are the most common. Allocating memory for an object in CUDA follows a repetitive pattern that can be abstracted away into a function. Consider the example in Listing 3.16—a mesh data-structure must be copied onto the GPU. It is easy for the user to forget the factor `sizeof(T)`.

Common errors that users run into include: forgetting to free memory and leaking it; calling free twice on the same pointer; dereferencing a null pointer or a pointer that is invalid on the GPU; and out-of-bounds access. We can wrap this allocation and deallocation into a type that automates this for the user, preventing a whole class of errors. Existing libraries such as Thrust offer such allocators. By combining them with the ideas demonstrated in the previous section,

we can improve their utility. Let us take a look at an example to motivate and describe the features we would like to have—automatic allocation and cleanup of GPU memory, restrict kernel parameters to pointers that are valid on the GPU, automatic utilization of optimal routines to copy memory between various heaps, and allocators that are compatible with the C++ container paradigm.

```
struct point {
    double x;
    double y;
};

struct mesh {
    std::vector<point> edge_points;
    std::vector<point> face_points;
};

struct mesh_gpu {
    point* edge_points;
    point* face_points;
};

std::vector<mesh> meshes { ... };
mesh_gpu* d_mesh; // to store the device address

static_assert(sizeof(mesh) == sizeof(mesh_gpu));

CUDA_CHECK(cudaMallocManaged(
    &d_mesh,
    sizeof(point) * mesh.size()));

// allocate pointers in d_mesh
CUDA_CHECK(cudaMallocManaged(
    &(d_mesh->edge_points),
    sizeof(point) * mesh.edge_points.size()));

CUDA_CHECK(cudaMallocManaged(
    &(d_mesh->face_points),
    sizeof(point) * mesh.edge_points.size()));

// copy data to the GPU
for (int mesh = 0; mesh < meshes.size(); ++mesh) {
    for (int j = 0; j = meshes.face_points.size(); ++j) {
        d_mesh[i]->face_points[j] = meshes[i].face_points[j];
    }
    // same for edge points
}

kernel<<<>>>(d_mesh->face_points, meshes.size());
```

Listing 3.16: Copying data from CPU to GPU—it is hard to figure out what is happening and check for correctness

The code in Listing 3.16 looks correct at first glance, but there are a few errors.

- Listing 3.16, line 23 should be `sizeof(mesh_gpu)`
- Listing 3.16, line 32 should be `mesh.face_points.size()`
- Listing 3.16, line 37 is correct, but `cudaMemcpy` is faster
- Listing 3.16, line 42 should be `meshes.face_points.size()`

If a structure has many vectors, as in Listing 3.16, it is necessary to first allocate the root object, and then recursively allocate and copy the containing elements. Had the root object `d_mesh` been allocated using `cudaMalloc`, this program would crash, as the pointer should not be dereferenced in host code. Consequently, all the assignments should be replaced with `cudaMemcpy`. Each time any new member is added to this structure, one should not forget to copy the appropriate elements to the GPU. This process is error-prone, distracts the user from implementing the core of the algorithm, and is sometimes hard to debug, especially when the memory is inaccessible from the host. One must remember to appropriately free these regions too, starting with the innermost object. Freeing just the root object leaks memory of all the contained objects. These problems can be avoided by abstracting all the allocation and deallocation behavior into an allocator, and ensuring that the types follow the RAII paradigm.

```
1  struct mesh {
2      std::vector<point> edge_points;
3      std::vector<point> face_points;
4  };
5
6  struct mesh_gpu {
7      cup::vector<point> edge_points;
8      cup::vector<point> face_points;
9
10     mesh_gpu& operator=(const mesh& cpu_mesh) {
11         // uses cudaMemcpy
12         edge_points = cpu_mesh.edge_points;
13         face_points = cpu_mesh.face_points;
14         // or some other implementation that optimizes
```

```
        // memory layout for the GPU
    }

    // similar copy constructor
};

std::vector<mesh> cpu_meshes { ... };
cup::vector<mesh_gpu> gpu_meshes { ... };

mesh_gpu gpu_meshes = cpu_meshes;

kernel<<<>>>(gpu_meshes);
```

Listing 3.17: Convenient but performant memory management that also ensures code reuse, thereby improving maintainability.

3.3.4 Allocator using CUDA APIs

Libraries (Thrust, CUDA API wrappers [16]) provide C++ abstractions over the plain CUDA C API, but we will use the C API to demonstrate the ideas.

We template the allocator on the *kind* of the heap too, as it allows us to perform compile-time introspection and implement optimized routines depending on the types of source and destination for move and copy-construction and assignment, both in host and device code. In Listing 3.18, we use template parameters, to determine, at compile-time, the type of allocator to use. It also includes a codepath to allocate memory from device code.

```
namespace cup {
enum class memory_type : uint8_t {
    managed,
    device_local,
    // pinned
};

template<typename T, memory_type MT>
class allocator {
  public:
    static constexpr memory_type memory_t { MT };

    // type traits
    using value_type = T;
    // implement rebind

    constexpr allocator() = default;

    // other constructors

```

```
21      [[nodiscard]] __host__ __device__
22      T* allocate(size_t byte_count) {
23          T* ret = nullptr;
24
25          if constexpr ( MT == memory_type::managed ) {
26  #           ifdef __CUDA_ARCH__
27                  CUDA_CHECK(cudaMallocManaged(&ret, byte_count));
28  #           else
29                  assert(0&& "Cannot call cudaMallocManaged in device
        code");
30  #           endif
31          }
32          else if constexpr ( MT == memory_type::device_local ) {
33              CUDA_CHECK(cudaMalloc(&ret, byte_count));
34          }
35
36          return ret;
37      }
38
39      __host__ __device__
40      void deallocate(T* ptr) noexcept {
41          CUDA_CHECK(cudaFree(ptr));
42      }
43
44      __host__ __device__
45      void deallocate(T* ptr, size_t n) noexcept {
46          CUDA_CHECK(cudaFree(ptr));
47      }
48  };
49
50  // equality comparison operators
51
52  template<typename T>
53  using managed_allocator = allocator<T, memory_type::managed>;
54
55  template<typename T>
56  using device_allocator = allocator<T, memory_type::device_local>;
57  }
58
59  auto cells = std::vector<Cell, cup::managed_allocator<Cell>>{ /**/ };
```

Listing 3.18: A simple allocator using CUDA API

Calls to the default allocator `std::allocator` used by containers in the standard library use the global `new` and `delete`, and place the object in an implementation-defined manner in a heap accessible only by the CPU (usually—unified memory support may change this). This simple allocator conforms to the requirements of the C++ language, meaning it can be used instead of the default allocator in containers. For example, Listing 3.18, line 59 places all the

cells in a CUDA-managed heap. They are accessible in both host and device memory.

This allocator makes use of neither the virtual memory APIs offered by the CUDA driver API [43], nor the stream-ordered memory allocator [39] that provides asynchronous allocations, and is completely stateless. Thrust exposes a set of higher-order memory allocation primitives, which can be used to implement a production-ready allocator with all or some of the aforementioned features, as needed. A library author can combine the techniques presented here with the allocators provided by Thrust (or extend them) to implement complex memory management and reuse strategies in an expressive way without the mental overhead of working through the complexity and verbosity of the C API.

On systems where it is supported, allocations from classic system allocators (such as plain old `malloc`) are transparently available to the GPU [41]. On such systems, it is valid to dereference CPU-allocated pointers on the GPU, without having to invoke any of the `cudaMalloc*` APIs.

An allocator as a template parameter can change the allocation behavior of a collection of items but sometimes we might want to change it for a single object. In such cases, we can override class-specific `new` and `delete` operators to use our allocator and place the objects in CUDA heaps.

```
struct Cell {
    /* members */
    [[nodiscard]] __host__ __device__
    static void* operator new(size_t byte_count) noexcept {
        // use cuda allocator
    }

    // delete calls cudaFree
};

// cell_gpu is a pointer to GPU memory
auto cell_gpu = std::make_unique<Cell>(); // smart pointer
kernel<<<>>>(cell.get(), ...); // kernel can modify cell
cudaDeviceSynchronize();
// cell is automatically freed when it goes out of scope
```

Listing 3.19: Custom `new` and `delete` operators for custom user-defined types allow allocation on the device-accessible heap.

This process of implementing `new` and `delete` for every such struct can become cumber-

some. To alleviate this issue, an empty struct with the custom implementation can be written, and any other struct requiring them can simply inherit from them, as in Listing 3.20.

```
template<typename T, typename allocator>
struct new_delete {
    public:
    [[nodiscard]] __host__ __device__
    static void* operator new(size_t byte_count) noexcept {
        allocator alloc {};
        return alloc.allocate(byte_count);
    }

    __host__ __device__
    static void operator delete(void* ptr) noexcept {
        allocator alloc {};
        alloc.deallocate((T*)ptr);
    }
    // other functions as needed
    // https://en.cppreference.com/w/cpp/memory/new/operator_new
};

struct Cell :
    public new_delete<Cell, cup::managed_allocator<Cell>> {

};
auto cell_gpu = std::make_unique<Cell>(); // smart pointer
kernel<<<>>>(cell.get(), ...); // kernel can modify cell
```

Listing 3.20: Generic `new` and `delete` operators that user-defined types can inherit from.

It is possible to implement an allocator to manage shared memory (`__shared__` memory within a compute unit), which can be used in conjunction with the vector class we describe in section 3.4 to ensure data alignment requirements are met.

3.3.5 Potential disadvantages

While inheriting from `new_delete` helps with making CPU data-structures available on the GPU, it does not allow one to call virtual functions on the GPU (CPU virtual function calls work as usual). If virtual function call support is desired on the GPU as well [42], libraries such as CHAI [2, 20] can be used. Calling `cudaMalloc*` APIs causes the GPU to synchronize across all executing CUDA streams. This is undesirable when the application allocates and frees memory at various points during runtime, not just at initialization. In such cases, stream-ordered memory allocators [39] and virtual memory APIs [43] might be of interest, together

with tools offered by Thrust.

3.3.6 Summary

In this section, we

- explored the common ways to allocate and manage host and device memory in CUDA applications;
- described the difficulties, common mistakes, and pitfalls of using the C API directly to manage memory;
- described language features in C++ that we can use to abstract away allocation and deallocation of memory by wrapping the C API with C++ constructs;
- demonstrated how these abstractions can help us in writing expressive and maintainable code without the disadvantages of using the C API directly; and
- laid out potential disadvantages and synchronization issues that may arise if one is not careful in implementing and utilizing these abstractions.

3.4 Vector

CUDA programs typically have a buffer of some sort containing a collection of items to be processed on the GPU. These buffers are allocated with memory-allocation functions provided by the CUDA API. CUDA offers a few APIs to allocate memory based on a user's needs, with the most common ones being `cudaMalloc` and `cudaMallocManaged`. The usual approach to allocating a buffer containing `num_items` of some type `T` could look like,

```
1  std::vector<T> cpu_items { /* */ };
2  T* gpu_items;
3  auto error_code = cudaMalloc(&gpu_items, sizeof(T) * num_items);
4
5  if (error_code != cudaSuccess) {
6      /* handle error, exit program */
7  }
8
9  // Copy items from CPU buffers into GPU memory
10 auto error_code = cudaMemcpy(gpu_items,
11     cpu_items.data(),
```

```
12      cpu_items.size(),
13      cudaMemcpyHostToDevice);
14
15  /* launch kernel */
16  cudaDeviceSynchronize();
17  cudaFree(gpu_items);
```

Listing 3.21: Copying items using the CUDA C API is error-prone, and is often hard to follow for nested types.

3.4.1 The problem

While Thrust offers `thrust::device_vector` and `thrust::host_vector`, the methods of the class aren't decorated with `__device__`, and therefore cannot be used in kernel code (Listing 3.22), and users resort to passing raw pointers into kernel functions. In this implementation, we focus on dynamic arrays—those whose sizes are only determined at runtime. While static arrays (`int cells[100]`) do have sizes associated with them in their type information, they decay to pointers when the full type is not specified (in function arguments, for example), their size must be known at compile-time, and are not dynamic.

```
1  // This example will not compile
2  __global__ void kernel(thrust::device_vector<int> ints) {
3      for (int i = 0; i < ints.size(); ++i) { }
4      for (auto& i : ints) { }
5  }
```

Listing 3.22: Thrust vectors cannot be used in device code.

Our design of a vector aims to bring these crucial convenience features—indexing `[]` operator, iterators, ranged `for`-loops, convenience methods such as `.size()`, `.front()`, `.data()`—we take for granted in host code into device code as well. We will also demonstrate that iterators can be used in device code too, with automatic parallelization using grid-stride [19] loops (see subsection 3.5.2). Our design mostly replicates the design of `std::vector` for simplicity and compatibility. We will use the allocators discussed in section 3.3 to allocate objects.

```
1  template<typename T,
2           typename allocator_t = managed_allocator<T>,
3           memory_type heap_t   = memory_type::managed>
4  class vector {
```

```
5   __restrict__ T* mem { nullptr };
6   int size { 0 };
7   int capacity; { 0 };
8
9   static constexpr auto heap_type { heap_t };
10  [[no_unique_address]] allocator_t alloc { };
11  // type traits
12  // constructors
13  // swap
14  // iterator methods
15
16  // Assignment from vector. Similar logic for copy-assignment
17  explicit vector(const std::vector<T>& vec) {
18    reserve(vec.size());
19    CUDA_CHECK(cudaMemcpy(mem,
20                          vec.data(),
21                          vec.size() * sizeof(T),
22                          cudaMemcpyHostToDevice));
23    // set size to capacity, indicating buffer is full
24    m_size = m_capacity;
25  }
26
27  __host__ __device__ __forceinline__
28  T& operator[](const size_t inx) {
29    #ifdef __CUDACC_DEBUG__
30    assert(inx < size && "Out of bounds access");
31    #endif
32    return mem[inx];
33  }
34
35  __host__ __device__ __forceinline__
36  const T operator[](const size_t inx) const {
37    const T* __restrict__ l = mem;
38    #ifdef __CUDACC_DEBUG__
39    assert(inx < size && "Out of bounds access");
40    #endif
41    return l[inx];
42  }
43
44  // other members not illustrated here for brevity
45 };
```

Listing 3.23: Our vector abstraction that can be used both in host and device code.

3.4.2 Implementation considerations

Since our vector implementation is an "owning" type (i.e., it manages memory), the copy-constructor and copy-assignment operators must implement a deep copy. Passing our vector "by value" as a kernel parameter therefore results in a full copy being generated. This is undesirable if the size of the buffer is large, and wastes PCIe bandwidth and system resources. In fact, it

is dangerous when the data-type has a non-trivial destructor due to how the kernel invocation syntax is implemented [35]. Excluding the copy-assignment and copy-constructors results in a shallow copy, but this leads to double-free errors, and does not follow the RAII paradigm.

To prevent a deep copy, we could pass the vector to the kernel as a reference or as a pointer to GPU-addressable memory, but this prevents the compiler from performing certain optimizations. The compiler cannot guarantee that the value pointed to, even if declared `const`, might be changed by a different thread. `const` only indicates to the compiler that users of this reference must not modify the value, not that the value itself does not change. A copy offers stronger guarantees than a value passed by reference. Care must be taken, however, as passing a local *stack* variable by reference (or pointer) will lead to a runtime error. As a workaround, the vector *itself* can be placed on the CUDA managed or pinned heap using overloaded `new` and `delete` operators (section 3.3), and using `std::unique_ptr<vector>` for automatic cleanup.

To get the best of both worlds (compiler optimizations through pass by-value as kernel parameters, and a vector-like API in device code), a non-owning container such as `cup::span` can be used as a kernel parameter. Custom conversion operators from Thrust's device vector `thrust::device_vector` to `cup::vector` allow seamless interoperability. Thrust's algorithms are container-agnostic and operate on iterators into containers. Our vector exposes iterators and can often be used as a drop-in replacement in place where Thrust's vectors are used. Codebases already using Thrust can continue utilizing Thrust on host code, and use a non-owning view into it (with `cup::span`, for example) as kernel parameters for maximum flexibility. The only difference between the vector and span is that the latter offers a non-owning view into the former—i.e., it does not allocate or free memory— and must not outlive the former.

Implementing a simple saxpy loop in both the C-style way and with our abstractions, we can see (Listing 3.25 and Listing 3.26) that the PTX generated is identical, and are identical in terms of resource usage.

```
1 // Assuming a non-owning span (a cup::vector without RAII)
2 __global__ void kern1(const cup::span<float> a, const cup::span<float
      > b, cup::span<float> c) {
3     int maxsz = c.size();
4
```

```
5       for(int i = blockIdx.x * blockDim.x + threadIdx.x; i < maxsz; i
        += blockDim.x * gridDim.x) {
6           c[i] = a[i] + b[i] * 3.2f;
7       }
8   }
9
10  __global__ void kern2(
11      float* __restrict__ a, int as, int ac,
12      float* __restrict__ b, int bs, int bc,
13      float* __restrict__ c, int cs, int cc) {
14
15      for(int i = blockIdx.x * blockDim.x + threadIdx.x; i < cs; i +=
        blockDim.x * gridDim.x) {
16          c[i] = a[i] + b[i] * 3.2f;
17      }
18  }
```

Listing 3.24: Kernels implemented using abstractions and the typical way.

```
1  $L__BB11_2:
2    mul.wide.s32    %rd7, %r10, 4;
3    add.s64         %rd8, %rd1, %rd7;
4    add.s64         %rd9, %rd2, %rd7;
5    ld.global.nc.f32        %f1, [%rd9];
6    ld.global.nc.f32        %f2, [%rd8];
7    fma.rn.f32      %f3, %f1, 0f404CCCCD, %f2;
8    add.s64         %rd10, %rd3, %rd7;
9    st.global.f32   [%rd10], %f3;
10   add.s32         %r10, %r10, %r3;
11   setp.lt.s32     %p2, %r10, %r6;
12   @%p2 bra        $L__BB11_2;
```

Listing 3.25: PTX for Listing 3.24 (line 6)

```
1  $L__BB8_
2    mul.wide.s32    %rd7, %r22, 4;
3    add.s64         %rd8, %rd3, %rd7;
4    add.s64         %rd9, %rd2, %rd7;
5    ld.global.nc.f32        %f1, [%rd9];
6    ld.global.nc.f32        %f2, [%rd8];
7    fma.rn.f32      %f3, %f1, 0f404CCCCD, %f2;
8    add.s64         %rd10, %rd1, %rd7;
9    st.global.f32   [%rd10], %f3;
10   add.s32         %r22, %r22, %r4;
11   setp.lt.s32     %p2, %r22, %r17;
12   @%p2 bra        $L__BB8_2;
```

Listing 3.26: PTX for Listing 3.24 (line 16)

The Clang compiler[2] allows a user to overload functions and methods based on the `__host__` and `__device__` attributes, and isn't available in the `nvcc` official compiler. This feature allows one to disambiguate functions with the same name and arguments based just on the attributes, allowing one to implement host and device logic in separate functions. While this doesn't seem that significant at first glance, when combined with templates and concepts, it can be used to restrict function usage. For example, it can be used to restrict the availability of `operator[]` to only in contexts where the memory can be safely dereferenced. This is extremely powerful as it moves a runtime error (dereferencing memory in the wrong execution space) to a compile-time one, dramatically improving user productivity and safety. An equivalent with partial functionality can be implemented with `nvcc` (Listing 3.27).

[2]Clang: Compiling `CUDA` with clang: https://llvm.org/docs/CompileCudaWithLLVM.html

```
auto func() {
    cup::vector<int, cup::device_allocator<int>>  ints_d {};
    cup::vector<int, cup::managed_allocator<int>> ints_h {};
    ints_d[19] += 100; // compile-time error in host code
}
```

Listing 3.27: Compile-time detection of incorrect access into buffers.

3.4.3 Potential disadvantages

As can be seen in Listing 3.23, the indexing operator had to be tweaked to coerce the compiler into generating the most optimized version of PTX, identical to the simple implementation. Abstractions are not free. They take up resources, either at runtime, or at compile-time (compilation takes longer). The compiler can retain only so much of the program's local context to generate optimized code, and in extreme cases, it might miss out on some optimizations. Strong attention to detail and deep understanding of C++ specification is required to ensure these abstractions generate optimal code.

During profiling, all profiler data for a memory load through the `operator[]` in device code is attributed to the location where it is defined (in `vector.cuh`), instead of the call-site. This makes it hard to track down the actual location of calls that contribute to all the aggregated metrics (which are typically one up the call-stack). Some debuggers offer a way to mark third-party code as such, and prevent stepping into it during debugging (called "just-my-code" in Visual Studio[3]). Similar functionality can perhaps be implemented and used in profilers to attribute metrics to the call-site of such functions, rather than the functions themselves.

We demonstrated that there is potential for our abstractions to generate optimal code. But depending on the specific circumstances under which they are used, it may not always be optimal, as can be seen in Listing 3.28. Tools such as godbolt.org make it easy to inspect the assembly generated. We urge the reader to use such tools to assist with the decision-making process.

Before we conclude, let us look at an example of missed optimizations that may happen when one does not inspect or realize the semantics of the code.

[3]Visual Studio Documentation: Just my code debugging
https://learn.microsoft.com/en-us/visualstudio/debugger/just-my-code

```
1     void foo(int* a, const int& b) {
2         for (int i=0; i<10; i++) {
3             a[i] += b;
4         }
5     }
6
7     // generates: https://godbolt.org/z/T7h4nK3G7
8     foo(int*, int const&):                          # @foo(int*, int
    const&)
9         mov     eax, dword ptr [rsi]
10        add     dword ptr [rdi], eax
11        mov     eax, dword ptr [rsi]
12        add     dword ptr [rdi + 4], eax
13        mov     eax, dword ptr [rsi]
14        add     dword ptr [rdi + 8], eax
15        mov     eax, dword ptr [rsi]
16        add     dword ptr [rdi + 12], eax
17        mov     eax, dword ptr [rsi]
18        add     dword ptr [rdi + 16], eax
19        mov     eax, dword ptr [rsi]
20        add     dword ptr [rdi + 20], eax
21        mov     eax, dword ptr [rsi]
22        add     dword ptr [rdi + 24], eax
23        mov     eax, dword ptr [rsi]
24        add     dword ptr [rdi + 28], eax
25        mov     eax, dword ptr [rsi]
26        add     dword ptr [rdi + 32], eax
27        mov     eax, dword ptr [rsi]
28        add     dword ptr [rdi + 36], eax
29        ret
```

Listing 3.28: The compiler generates unnecessary reads from `b` at each iteration.

In Listing 3.28, the function `foo` increments each member of array `a` by `b`. Each `mov` (Listing 3.28, line 9) loads `b` into a register, and each `add` (Listing 3.28, line 10) adds the loaded `b` value to `a[i]` and then stores the result into `a[i]`. It could very well be that `b` is a reference into one of the elements in `a`; this is called "aliasing." To account for this possibility, the compiler has to load `b` after each iteration.

To prevent this, the argument `a` could be decorated with `__restrict__`, which informs the compiler that two pointers cannot point to overlapping memory regions. One would immediately pose the question—and rightfully so—"Why would the compiler load `b` over and over again, is it not `const`?" A `const` reference (`const T&`) only guarantees that the reference cannot be modified using that "name" (we cannot use `b` to modify the value it exposes), not that the reference *itself* does not change—another thread, which holds a mutable reference to the

underlying value represented by `b`, could have modified `b`. We therefore must load `b` at every iteration to ensure logical correctness, even if that was not the intent of the programmer (the code expresses a different intent than what the programmer intended and expects). Passing our vector as a kernel parameter by-reference could generate suboptimal code due to this reason. Non-owning containers can be used to prevent this, as they can safely be passed by-value.

3.4.4 Summary

In this section, we

- demonstrated the need for and the convenience of a vector-like type that can be used in both host and in device code;
- looked at Thrust's vector, and realized that it cannot be used in device code (which may be a design decision);
- proposed and constructed a vector type that utilizes the allocator types presented in previous section to simplify usage across both host and device code;
- stressed the non-standard semantics that CUDA uses for passing kernel arguments from host to device code and how it impacts us and our implementation;
- proposed alternative implementations and design considerations to work around the problems it presents; and
- explained the disadvantages and shortcomings of this approach (missed optimization opportunities), ways to overcome them, and how additional language features or compiler hints may address some of these concerns.

3.5 Iterators

An iterator enables one to traverse (i.e., iterate) over containers such as vectors and lists. This allows one to iterate over the elements of a container without explicitly indexing into it (for example, with the `[]` operator). Most programming languages offer various means to do so. Using iterators prevents runtime errors such as out-of-bounds accesses, and simplifies implementation

of higher-order primitives and algorithms that process a collection of items. By exposing a common interface, algorithms can be implemented agnostic to the underlying container (Listing 3.29).

3.5.1 Introduction

Thrust offers iterators, but since the underlying container cannot be used in device code, iterators on host code, when passed to kernels, must "decay" to the underlying pointer types. In this case, iterators are usually type aliases to a pointer of the underlying type (`float*`, for example). While this retains much of the iterator-like functionality, the associated metadata such as the size, capacity, and bounds-checking of the container must be passed as separate arguments to the kernel, due to the aforementioned reason. We want to be able to use iterators and the associated containers as well in device code, along with the convenience language and library features that they enable. Iterators enable succinct but expressive implementation of loops, and we want to bring this to device code as well. Let us take a look at a few examples that demonstrate the utility of iterators.

In Listing 3.29, the user need only define a single function that is flexible to the types of arguments passed in. The function `double_all` is actually a template over its arguments—the `auto` keyword is just syntactic sugar for writing the full template. Thrust offers similar features that can be used on the host and device (for example, `thrust::transform`). With access to iterators in device code too, we are able to complement the library's feature set. For better error messages during compilation, one could define a concept (Listing 3.29, line 8) that ensures that the function arguments meet the requirements of a container (Listing 3.29, line 14).

```
1  void double_all(auto& items) {
2    for(auto& item : items) {
3      item = item * 2;
4    }
5  }
6
7  template <typename container_t>
8  concept container = requires(container_t& a, container_t& b) {
9    // provides a .begin function
10   { a.begin() } -> std::same_as<typename container_t::iterator>;
11   // and so on
12 };
13
```

```
void double_all(container auto& items) {
  for(auto& item : items) {
    item = item * 2;
  }
}


void foo() {
  std::vector<int>   a{ 1, 2, 3 /*...*/ };
  std::vector<float> b{ 1.1, 2.2, 3.3 /*...*/ };
  std::list<double>  c{ 1.2, 2.3, 3.4 /*...*/ };

  double_all2( a );
  double_all2( b );
  double_all2( c );
  thrust::transform(items.begin(), items.end(), /**/);
  // and so on
}
```

Listing 3.29: Example of an iterator. We don't need to explicitly specify the start and end index—they are inferred.

A classic iterator goes over elements one at a time. We should not use the same iteration mechanism in the GPU, as this would cause *all* threads to iterate over *all* the elements in the container, starting at index zero. Instead, we use a form of parallelization (discussed in the next section) where we use the current thread's index to process different pieces of data with the same instructions.

Without iterators, both host and device code may be more verbose than preferred. Assuming that `mesh`'s `cell` member is a type that provides the appropriate iterators, one could implement mesh processing as follows,

```
__device_
void process_mesh_a(mesh& gpu_mesh) {
    for (auto& cell : to_gpu_iterator(gpu_mesh.cells)) {
        process_cell(cell);
    }
}

__device_
void process_mesh_b(mesh& gpu_mesh) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    for (int inx = tid; inx < gpu_mesh.cells.size(); i += stride) {
        process_cell(gpu_mesh.cells[inx]);
    }
```

```
16 }
```

Listing 3.30: GPU code that is as expressive and readable as CPU code using custom iterators—we can use ranged for-loops in GPU code.

Let us look at how we can implement `to_gpu_iterator`.

3.5.2 Grid-Stride loop

CUDA exposes threads in a hierarchy of grids, blocks, and threads [18]. A grid is made up of multiple blocks, and a block consists of multiple threads. For best performance, neighboring threads from within a compute unit must access elements from neighboring memory locations. A common iteration strategy to maintain coalesced accesses while processing each item exactly once is called the "Grid-Stride loop" [19]—note the initialization and increment of the loop variable `i` (Listing 3.31). This ensures that neighboring threads access neighboring elements and process all items in sets of warp-size.[4] Consequently, each thread index is incremented by the total number of threads that were launched in parallel (block size times the grid size).

```
1 __global__ void kernel(cup::vector<int>& items) {
2   const auto tid { threadIdx.x + blockIdx.x * blockDim.x };
3   const auto stride { blockDim.x * gridDim.x };
4
5   for (int inx = tid; inx < n; inx += stride) {
6     // ...
7   }
8 }
```

Listing 3.31: CUDA Grid-Stride loop to parallelize processing of items on the GPU

3.5.3 The problem

In subsection 3.4.1, we saw that abstractions such as Thrust's vector cannot be used in device code. A vector is just one half of the solution. The other half is providing iterators to help implement convenient parallelization strategies in kernel code. One of the most common issues encountered is incorrect work parallelization using grid-stride loops. For example, it is good exercise to try and find the errors in Listing 3.32.

```
1 __global__ void kernel1(int* mem, int size) {
```

[4]The smallest executable unit of parallelism on a CUDA device comprises 32 threads (termed a warp of threads) [37].

```
    const int tid    { threadIdx.x + blockIdx.x * blockDim.x };
    const int stride { blockDim.x * gridDim.x };

    for (int i = 0; i < size; ++i) {  }
}

__global__ void kernel2(int* mem, int size) {
    const int tid    { threadIdx.x + blockIdx.x * gridDim.x };
    const int stride { blockDim.x * gridDim.x };

    for (int i = threadIdx.x; i < size; i += stride) {  }
}

__global__ void kernel3(int* mem, int size) {
    const int tid    { threadIdx.x + blockIdx.x * gridDim.x };
    const int stride { blockIdx.x * gridDim.x };

    for (int i = tid; i < size; i += stride) {  }
}

__global__ void kernel4(int* mem, int size) {
    const int tid    { threadIdx.x + blockIdx.x * gridDim.x };
    const int stride { blockDim.x * gridDim.x };

    for (int i = tid; i < size; i += stride) {  }
}
```

Listing 3.32: Incorrect grid-stride implementations.

3.5.4 Prior work

Hemi [25] exposes in-kernel iterators that automatically implement the grid-stride loop, allowing one to write Listing 3.33. Alternatively, CUDA offers cooperative groups, which provide a higher-order abstraction over grids and blocks. They also offer higher-order synchronization primitives to synchronize between threads, blocks, or across the whole system.

```
void saxpy(int n, float a, const float *x, float *y) {
  hemi::parallel_for(0, n, [=] HEMI_LAMBDA (int i) {
      y[i] = a * x[i] + y[i];
  });
}

// alternatively,
__global__
void saxpy(int n, float a, float *x, float *y) {
    for (auto i : hemi::grid_stride_range(0, n)) {
        y[i] = a * x[i] + y[i];
    }
}
```

Listing 3.33: Hemi example that implements a grid-stride loop.

3.5.5 Grid-stride loop using iterators

We need to construct an iterator that emulates the grid-stride loop. An iterator following the same approach with initialization and increment of the iterating variable fulfills our criteria—each thread initializes the iterator with its global thread ID, and increments it by `stride`, and uses a sentinel value to terminate the loop. The adaptor in Listing 3.35 changes the start and end iterators, and returns new iterators that implement the grid-stride iteration strategy.

Before we implement our iterator, we need to understand how ranged `for`-loops [15] work. For our current implementation, a rough mental-model of what the compiler generates could look like Listing 3.34,

```
1  for (auto& i : items) {
2      // user's code
3  }
4
5  // can be visualized as,
6  {
7      auto begin = items.begin();
8      auto end   = items.end();
9      for (auto ptr = begin; ptr != end; ++ptr) {
10         auto& i = *ptr;
11         {
12             // user's code
13         }
14     }
15 }
```

Listing 3.34: Range based for loop—compiler approximation.

Since the exit criteria is not based on a less-than-or-equal ($\leq$) comparison, but rather a strict equality comparison (Listing 3.34, line 9), we should explicitly set the end iterator one stride away from the last valid element *per-thread*. This is acceptable, as the last element is never dereferenced. In the code listing below (Listing 3.35), we present constructs that encapsulate:

- the construction of a special range-view into a collection called `grid_stride_adaptor`, whose iterators implement the grid-stride loop. This object takes in any container that exposes a contiguous iterator and uses its `begin()` and `end()` methods to generate the

corresponding iterator objects that bound the ranged-for loop; and

- the logic for the grid-stride iterator itself (`grid_stride_iterator`), that implements the grid-stride loop. It simply initializes the iterator based on the thread index, and implements a method that increments the iterator by the stride value.

```
template<typename T>
struct grid_stride_iterator {
  __device__
  grid_stride_iterator(T pos)
    : m_pos { std::to_address(pos) +
              threadIdx.x + (blockIdx.x * blockDim.x) } {}

  constexpr grid_stride_iterator& operator++() {
    m_pos += blockDim.x * gridDim.x;
    return *this;
  }

  // other methods
  private:
    pointer m_pos {};
};

template <typename T>
struct grid_stride_adaptor {
  using underlying_iterator = T::iterator;
  using iterator = grid_stride_iterator<underlying_iterator>;

  underlying_iterator pstart {};
  underlying_iterator pend {};

  __device__
  grid_stride_adaptor(T& ctr)
    : pstart { ctr.begin() } {
    const size_t sz     = ctr.size();
    const size_t stride = blockDim.x * gridDim.x;
    const size_t remd   = sz % stride;
    const size_t tid = threadIdx.x + (blockIdx.x * blockDim.x);

    pend = pstart + sz - remd;
    if (remd != 0 && tid < remd)
      pend += stride;
  }

  constexpr iterator begin() {
    return { pstart };
  }

  constexpr iterator end() {
```

```
44      return { pend };
45    }
46 };
47
48
49 __global__ void kernel(std::span<int> items) {
50     for (auto& i : grid_stride_adaptor(items)) { ... }
51 }
```

Listing 3.35: A minimal example of a grid-stride iterator and adaptor that enables efficient grid-stride iteration over any iterable (Listing 3.35, line 50).

We can see that the loop in Listing 3.36 not only states the intent of the programmer—modifying (or adapting) the iterator to optimize access patterns for the GPU—but also prevents errors in implementing the grid-stride loop. It is arguably more readable and intuitive to understand than the classic implementation. Our approach follows the DRY (do not repeat yourself) principle, and prevents one of the most common errors that a user can make—incorrect implementation of the grid-stride loop, a common source of frustration and confusion.

```
1 __device_
2 void process_mesh(mesh& gpu_mesh) {
3     for (auto& cell : grid_stride_adaptor(gpu_mesh.cells)) {
4         process_cell(cell);
5     }
6 }
```

Listing 3.36: Using the CUDA grid-stride adaptor. Each `cell` is processed in parallel, allowing for generic and error-free implementation. Only one implementation of grid-stride technique needs to be maintained.

In some cases, we want an index sequence rather than a reference to the elements themselves. To support such a case, the iterator can be initialized with just the indices rather than the pointers. Using iterators helps us prevent the most common class of errors when trying to index into buffers on the GPU—out-of-bounds accesses and incorrect implementation of the grid-stride loop.

3.5.6 Potential disadvantages

While the PTX for the loop is nearly identical, in release mode, this implementation uses 16 registers, whereas the simple equivalent implementation uses 12 registers. When compared to the simple implementation, we perform extra work initially to compute the exact end iterator

and store it in a local variable, which explains the extra register overhead. For applications that suffer from register pressure, this increase in register usage might reduce occupancy, which may lead to a decrease in performance. This extra register pressure may be completely unacceptable for some users, and in cases where it is not, we urge the user to measure (among other things, register usage) and benchmark their code to precisely determine if the tradeoff is acceptable.

We can see (below) that the PTX that implements the loop of our examples is almost identical. The iterator example has one fewer instruction, and the load-multiply-store sequence of instructions appear earlier in the loop.[5] The extra `add` instruction (on line 7) is for indexing into the buffer `a`—we increment the iterating variable `i` which is then used to update the current pointer into the buffer `a`.

```
1  __global__
2  void kern_normal(int* __restrict__ a, int sz) {
3
4    const auto tid {
5      threadIdx.x + (blockIdx.x * blockDim.x)};
6    const auto stride{
7      blockDim.x * gridDim.x};
8
9    for (size_t i = tid; i < sz; i += stride) {
10     a[i] *= 3;
11   }
12 }
```

Listing 3.37: Grid-stride loop *without* iterator.

```
1  __global__
2  void kern_span(std::span<int> a) {
3    // A span is equivalent to
4    // struct {
5    //   int* memory;
6    //   size_t size;
7    // } + other methods;
8
9    for (auto& i : cup::grid_stride_adaptor(a)) {
10     i *= 3;
11   }
12 }
```

Listing 3.38: Grid-stride loop *with* iterator.

```
1  $L__BB0_2:
2    shl.b64         %rd8, %rd10, 2;
3    add.s64         %rd9, %rd4, %rd8;
4    ld.global.u32   %r7, [%rd9];
5    mul.lo.s32      %r8, %r7, 3;
6    st.global.u32   [%rd9], %r8;
7    add.s64         %rd10, %rd10, %rd3;
8    setp.lt.u64     %p2, %rd10, %rd2;
9    @%p2 bra        $L__BB0_2;
```

Listing 3.39: PTX for Listing 3.37 (line 10)

```
1  $L__BB1_5:
2    ld.global.u32   %r11, [%rd27];
3    mul.lo.s32      %r12, %r11, 3;
4    st.global.u32   [%rd27], %r12;
5    shl.b64         %rd25, %rd1, 2;
6    add.s64         %rd27, %rd27, %rd25;
7    setp.ne.s64     %p4, %rd27, %rd9;
8    @%p4 bra        $L__BB1_5;
```

Listing 3.40: PTX for Listing 3.38 (line 10)

3.5.7 Summary

In this section, we

[5] A godbolt.org example is available to test at https://godbolt.org/z/efj34nM1r

- explored the advantages of being able to use a type that exposes an interface similar to `std::vector` in both host and device code;
- demonstrated the need for an abstraction for iterating over a range of elements;
- expanded upon the vector described in the previous section and how iterators offer flexibility;
- explored the advantages and disadvantages of using iterators over implementing them using raw loops—this may introduce additional register pressure, which may be completely unacceptable in certain cases, but makes the code easier to read and maintain; and
- presented resources and ways to inspect generated code to fine-tune the performance-maintainability trade off.

3.6 Coroutines

A coroutine is a function that can suspend execution to be resumed later. This makes them ideal for implementing lazy generators, infinite lists, cooperative scheduling, and event loops. In this section, we will take a look at how coroutines can be used by a library author to wrap common CUDA programming patterns in more familiar user-friendly C++ ones, simplifying consumption by end users.

3.6.1 Introduction

The main difference between a function and a coroutine is that the latter can persist its state across subsequent invocations. The state—usually collectively referred to as an "activation frame"—includes information such as function arguments, local variables, and return addresses, to name a few. The activation frame is essentially a block of memory that holds all this state. As function invocations are strictly nested, their activation frames can be stored and manipulated using efficient data-structures such as a stack (usually referred to as the "call-stack" or "stack frame"). In contrast, coroutine activation frames may not be strictly nested, and therefore are usually stored on the heap. It is important to note that coroutines do not require multiple threads. Coroutines and multithreading impart concurrency and parallelism respectively to a

process. This implies that a single-threaded program can have concurrent blocks of code. For example, single-core machines of the past supported concurrency by time-slicing between running processes.

3.6.2 Why coroutines

Before we delve further, let us try and understand why coroutines may be useful. CUDA programs, from the point of view of the CPU, involve running some host code, preparing work for the GPU, launching the kernel, perhaps executing more host code, and then waiting for the GPU work to complete. If the work is still pending, the program can resume processing more host code before checking if GPU work is complete. This in essence, from the point of view of the CPU, is concurrency—similar to how a single-threaded program can periodically check on a file descriptor with non-blocking system calls such as `poll` or `epoll`. Coroutines therefore offer a natural abstraction over scheduling GPU work.

3.6.3 Coroutines in C++

Coroutines were introduced to the C++ language in the C++20 standard. C++20 coroutines—as a language feature—offer the basic building blocks and compiler support necessary to build and implement user-facing coroutines. Implementing a coroutine without a helper library (such as cppcoro [22]) entails a fair bit of boilerplate code, which some examples exclude for the sake of brevity. A coroutine consists of three fundamental building blocks that offer points where the user can customize functionality—a task that encapsulates the work to run; a handle to the coroutine to suspend, resume, and complete execution; and a way to retrieve a result at various points in the coroutine's lifetime. A coroutine is associated with

- a *promise* type to store and retrieve the result, modified from within the coroutine;
- a *handle* to the coroutine to suspend, resume, and complete the execution of coroutine by destroying its activation frame; and
- the *coroutine state*, which contains the local variables, coroutine arguments, storage for the promise type, and an implementation-defined piece of data that represents the suspension and continuation point.

Before we delve further, let us take a look at a simple coroutine that showcases the boilerplate to better understand the details of these building blocks.

```
1   struct promise_type {
2     T value{};
3
4     generator<T> get_return_object() {
5       return {handle_type::from_promise(*this)};
6     }
7     std::suspend_always initial_suspend() noexcept { return {}; }
8     std::suspend_always final_suspend()   noexcept { return {}; }
9     // void return_void() {}
10
11    T return_value() { return value; }
12    std::suspend_always yield_value(T &&v) {
13      value = std::move(v);
14      return {};
15    }
16    std::suspend_always yield_value(const T &v) {
17      value = v;
18      return {};
19    }
20    std::suspend_always return_value(const T &v) {
21      value = v;
22      return {};
23    }
24    T await_transform() {}
25    void unhandled_exception() {}
26  };
27
```

Listing 3.41: Promise type of the generator that stores and gets the value.

The promise type (Listing 3.41) is required to expose a few functions that the compiler uses to generate an internal state-machine for the coroutine. Of particular interest are the ones that set and return the value which correspond to `co_yield` and `co_return` in the coroutine respectively. We can use these to implement a lazy generator that, for example, generates the sequence of Collatz[6] numbers for a given initial value.

```
1 generator<uint64_t> collatz_sequence(uint64_t val) {
2   while (val != 1) {
3     co_yield val;
4     if (val % 2 == 0)
5       val /= 2;
6     else
```

[6]Collatz conjecture—wikipedia.org/wiki/Collatz_conjecture

```
7         val = (3 * val) + 1;
8     }
9     co_return val; // value is 1
10 }
11
12 int main() {
13
14   auto seq = collatz_sequence(120);
15   int c = 0;
16
17   while (seq) {
18     printf("Iter[%5d]: %5lu\n", c++, seq());
19   }
20 }
```

Listing 3.42: A coroutine that lazily generates the Collatz sequence for a given input.

3.6.4 Coroutines and CUDA

The customization points from Listing 3.41 can be used to launch and wait on kernels, which allows us to write,

```
1 auto func() {
2     // ...
3     auto work = compute_gpu(/* ... */);
4
5     // perform some other tasks as GPU task runs
6
7     co_await work(); // proceed only after GPU work returns
8     // If work is not yet complete, control flow is returned
9     // to caller of 'func'
10
11    // rest of the function
12 }
```

Listing 3.43: A potential example of CUDA kernel launch and how coroutines fit into the programming paradigm.

This may seem inconsequential, but this allows a completely new and succinct way to express complex CUDA features and kernel-launch dependency chains of both CPU and GPU work.

Kernel launches are not free—they take up CPU time. When quite a few kernels are launched to do some intermediate work before launching the large compute kernel, the delay due to the submission of work (kernel launch) starts to add up and may even dominate the time taken to run the kernel itself. CUDA graphs were introduced to alleviate this issue. In-

stead of the CPU submitting each kernel individually, CUDA exposes an API that allows us to instantiate a graph, record all the kernels we want to launch, and submit all of them to the GPU at once. CUDA graphs can be created in one of two ways—stream capture, and explicit graph creation. An example used on the CUDA documentation[7] is given below,

```
auto func(/* */) {
    //   A
    //  / \
    // B   C
    //  \ /
    //   D
    cudaStreamBeginCapture(stream1);

    kernel_A<<< ..., stream1 >>>(...);

    // Fork into stream2
    cudaEventRecord(event1, stream1);
    cudaStreamWaitEvent(stream2, event1);

    kernel_B<<< ..., stream1 >>>(...);
    kernel_C<<< ..., stream2 >>>(...);

    // Join stream2 back to origin stream (stream1)
    cudaEventRecord(event2, stream2);
    cudaStreamWaitEvent(stream1, event2);

    kernel_D<<< ..., stream1 >>>(...);

    // End capture in the origin stream
    cudaStreamEndCapture(stream1, &graph);
}
```

Listing 3.44: CUDA graph using the C API

The dependency chain in the example is difficult to parse and reason about. With coroutines, we can rewrite the example above as follows,

```
auto func() {
    // Launch A and wait
    co_await launch_kernel_A(...);

    // Launch B and C when A is done
    auto kern_B = launch_kernel_B(...);
    auto kern_C = launch_kernel_C(...);
```

[7]CUDA Graphs—Cross-stream Dependencies and Events:
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=graph#creating-a-graph-using-graph-apis

```
8
9       // Wait for B and C to finish before launching D
10      co_await wait(B, C); // or co_await B; co_await C;
11      // Launch D, wait, and return
12      co_await launch_kernel_D(...);
13  }
```

Listing 3.45: The same graph as above (Listing 3.44), but using coroutines

Listing 3.45 is arguably significantly easier to build a mental model of and reason about. A lot of the complexity is implemented in the awaitables and the coroutine object. In particular, one must implement a scheduler that launches these kernels on a stream and wakes them up after checking that the kernel has finished executing.

The implementation we just described faces the same issue the CUDA graphs solve—launching kernels takes time, and launching many can cause a lot of latency. Coroutines are flexible enough in that we can take advantage of CUDA graphs for launching kernels too. Instead of actually launching the kernel, the coroutine implementation could just as well capture the launches with stream capture and lazily submit all the work at the end of the capture, just as seen in Listing 3.44.

A coroutine's allocator can be overloaded to specify a custom allocator. This allows one to use the CUDA managed-heap for allocating the coroutine frame. This allows one to allocate all the local variables in the coroutine frame on the CUDA managed heap, allowing them to be freely passed by-reference (either through a reference, or as a pointer with the address-of operator `&value`) to kernel parameters. This could be useful for prototyping implementations before implementing the final API.

3.6.5 Potential disadvantages

As of writing, the coroutine support library is still in development, and is expected to ship with C++23 or C++26 which includes commonly used abstractions over the raw-coroutine compiler-interface. This should help simplify writing custom coroutines that integrate and interoperate with CUDA.

When using CUDA managed memory for allocating a coroutine frame, care must be taken. Managed memory works through a mechanism known as paging [40]. Assume a case where two variables are present in the same page of memory and are initially resident on the CPU.

When the GPU tries to access a variable, a page fault is generated and is migrated to the GPU. At the same time, if the CPU tries to access it (since kernel launches are asynchronous, or from a different thread), the program will be terminated with a platform-dependent runtime-error. A workaround is to explicitly synchronize with the CPU or to use pinned memory.

3.6.6 Summary

In this section, we

- looked at a new C++ feature—coroutines—and how they can be used to expose and abstract asynchronous operations, including CUDA kernel launches;
- described how the native API for parallel kernel launches may be cumbersome to use;
- proposed how coroutines can tap into this API machinery and be used to launch kernels and asynchronously wait for the results;
- described the alternatives that exist (CUDA graphs) and how they solve the issue of kernel launch latency, and how this abstraction also fits nicely into coroutines; and
- explained the shortcomings of this approach, and potential issues that one might encounter—memory consistency and synchronization issues that might arise if one is not careful with memory access patterns in the application.

Chapter 4

Case study

Now that we have described various techniques, we bring them to bear on a widely used critical piece of software. In this chapter, we present a case study through which we demonstrate the advantages and disadvantages of the techniques we presented so far. We will first introduce the software under study, describe its various characteristics to understand and reason about the constraints we are bound by, and then proceed to describe how our tools helped the development of this piece of software.

4.1 Introduction

The Hydrologic Engineering Center of the US Army Corps of Engineers has been developing computer software for hydrologic engineering and planning analysis procedures since its inception in 1964. The Hydrologic Engineering Center's River Analysis System (HEC-RAS) is a piece of software that allows the user to perform one-dimensional steady flow simulation, one and two-dimensional unsteady flow calculations, sediment transport/mobile bed computations, and water temperature/water quality modeling [49]. HEC-RAS is used to simulate disasters such as flooding and visualize the scale and extent of its impact. Due to the extensive insight it offers into the behavior of water over large land masses (e.g., cities near a river), it plays a key role in planning of hydrological projects, prevention and preparation for disasters such as floods and dam bursts.

Due to the nature of the software—improper use could potentially cost lives—concerns such as the ability to audit and inspect the codebase and its dependencies, maintainability of the

codebase, documentation, and testing are paramount. They help ensure that the team developing the code can, at all times, reason about the behavior of the implementation and minimize human errors and errors introduced into computations.

In this chapter, we will cover two different aspects of the implementation.

- Implementation of the solver in C++
- Exposing this solver to C# though an API

Before we describe the internal details, we first describe the project being ported, and why it is a good case study to field-test our ideas.

4.2 Background

HEC-RAS can be described in simple terms as a shallow-water simulation framework. Shallow-water simulation typically concerns itself with the behavior of large bodies of water where the horizontal scale is much larger than the vertical scale (depth)—shorelines, rivers, lakes, for example. We can make the assumption that the vertical velocities are small and that the pressure gradient is hydrostatic [48]. The general mathematical models are therefore based on depth-averaged version of classic three-dimensional conservation laws [17]. Consequently, this problem can be approximated with a one or two-dimensional model (depending on the setting) that may store additional data to better approximate a full three-dimensional simulation. Such approximation relies on unstructured grids to cover large areas of land as the simulation domain, with higher-resolution meshes used around key areas of interest such as dams and levees. This also allows computational flexibility—different timesteps (or adaptive timesteps) can be used at different parts of the mesh to tune the tradeoff between accuracy of the solution and time taken.

The current iteration of the software suite is built using C#. This is part of an ongoing effort to modernize the codebase and migrate it away from Fortran. C# is a garbage-collected language by Microsoft, and is often compared to Java due to the similarities they share. Both are garbage-collected languages and allow a user to write a program once and run them anywhere (WORA). It uses the .NET Core runtime to execute users' programs. This makes it easier to develop (once) and run the non performance-critical parts of the application (such as the UI) on any system.

Water simulation frameworks are often excellent candidates for extensive parallelization due to the nature of the computations involved. The simulation domain containing land and water is divided into cells separated by faces. Fundamentally, the state of water level in a cell is a function of the water level and velocity of water in neighboring cells. This computation pattern is ripe for parallelization. Computations for a given cell can often be performed independent of computations in any other—an ideal candidate to offload onto the GPU. There are various mathematical methods that describe the steps to take to achieve a realistic simulation, and are collectively known as "solvers."

4.3 Design decisions

While C# can be used to write performant code that takes advantage of the hardware, it is mostly limited to the CPU. It does not officially provide support for offloading computations onto the GPU. The garbage collector (GC) is a piece of the problem. While there are various techniques to minimize the impact of a garbage collector, its behavior cannot be precisely controlled. At some point, the garbage collector must pause all executing threads and clean up unreachable memory, or risk leaking it. This may introduce unacceptable latency into the time-critical areas of the application. Techniques such as memory pools which reuse memory preventing repeated allocations and deallocations, and pinning memory which informs the GC that it should not clean up the referenced memory, can be used to lower the impact. In fact, these approaches are used in the CPU implementation of the solvers.

A few potential approaches exist to offload computations onto the GPU, which we describe below, to establish the context and reasoning for our implementation choices.

- Use an existing C#[1] library that compiles C# functions into GPU kernels.
- Use CUDA directly in C# by using wrapper libraries that expose the C API of CUDA as a C# library. These are libraries typically known as "native bindings."
- Implement the core computational logic in C++, a language officially supported by CUDA, and expose only the computational interface to the C# codebase.

[1] Altimesh-Hybridizer [1], ILGPU [24], ManagedCUDA [26], to name a few

4.3.1 Existing libraries

Typically, it is recommended that one use existing libraries to implement missing functionality. In the case of HEC-RAS, this may be a potential liability. For such an important piece of software that has real-life consequences, minimizing dependencies not only ensures that the development team does not need to depend on the library author for bugs, features, and support, but also makes it simpler to audit the software. It also maximizes licensing options for the software and the source code, and minimizes licensing liability. Another major consideration is that there are no officially supported libraries published by NVIDIA for C#, meaning that one has to rely on third party libraries. This also implies that official support channels from NVIDIA generally cannot be used.

4.3.2 Native bindings

In this approach, the C API of a library is exposed through a wrapper in the guest language. The wrapper library forwards all API calls to and from the library and essentially provides a translation layer for incompatible types. For example, since the string representations in C++ and C# are different, the library, in the wrapper function, can transform to and from these representation formats to allow seamless interoperability. The wrapper library usually loads the native library (.dll or .so on Windows and Linux, respectively) at runtime through system calls such as `LoadLibrary` and `dlopen`. When using such techniques, one must be extremely careful with memory management and ensure that memory managed by one (language or library) isn't freed by the other. This approach is flexible, as it offers the same API as the underlying native library offered by the vendor, but each API call has to go through this "marshalling" process, which may introduce expensive bottlenecks.

4.3.3 Native solver libraries

The approaches described above have their own sets of advantages and disadvantages. We can instead take an approach that is a compromise between the two extremes. The solver is implemented in native C++, and only the parts of the solver that require external input or configuration and calls to exchange data are exposed as a C API to C#. HEC-RAS can then load the native library at runtime when a GPU is detected on the system and configure the solver with the

appropriate runtime parameters. The advantage of this approach is that we marshal data only when crossing API boundaries, which happens only during configuration and exchange of data. This approach also does not require manual GC intervention, as only the marshalling interface uses manually-managed memory and pointers. Overall, we only pin the pointers that point to structures that are long-lived and generally do not change through the lifetime of the simulation, completely eliminating the concerns of manual GC configurations. Since much of the simulation data is managed by the library, the driver application can simply read and write to that memory through the interface it exposes. This is exponentially easier than the inverse—using memory managed by C# in a native environment such as C++ requires manual intervention to prevent the GC from relocating memory in use by native code (which would invalidate the pointer). In simulation loops, pinning and unpinning various memory locations can introduce significant performance bottlenecks.

4.4 Internals

HEC-RAS operates on unstructured grids of cells—the cells and faces of the grid are not identical in dimension and can vary depending on the resolution and precision required. For example, a high resolution and fine grid may be used around non-uniform and rapidly changing terrain elevations and near dams. Due to the nature of the data involved, storing cells and faces separately is often the best approach, and cell-face relationships are stored as indices to the corresponding data—a cell stores the indices of the faces (in the faces array) that define the cell.

At runtime, HEC-RAS loads the solver library if a CUDA-capable GPU is found, and passes the configuration parameters to it through the marshalling interface [27] offered by C#. Solver methods to run a single iteration or a variable number of iterations are exposed through the C API, which allows the user to configure the solver and inspect its state at different points of the simulation, improving user experience. This is especially useful for visualization of water movement over time. All CUDA memory is allocated by the CUDA library and is exposed as a pointer and size pair to the driver application, which allows it to safely read the contents of memory. The allocators we describe play a crucial role in making this as frictionless as possible.

For the initial proof of concept, we implemented a native library to demonstrate the potential

performance gains a GPU can bring forth. The tools we developed enabled quick iteration of ideas, minimized the lost productivity due to common CUDA programming mistakes, and allowed us to focus on application and implementation considerations, while being reasonably performant.

4.5 Implementation details

We will take a look at some of the data-structures representative of their HEC-RAS counterparts, and demonstrate how our abstractions improve maintainability, readability, and expressiveness of the code.

As previously mentioned, HEC-RAS operates on unstructured grids which makes using matrix-based methods to arrive at solutions difficult due to the the organization of data. It operates on buffers that represent the grid and current water state (Listing 4.1). These buffers are populated by the C# code through the API exposed by the C++ shared object.

```
namespace cpu {
    struct mesh {
        std::vector<cell>   cells {};
        std::vector<face>   faces {};
        std::vector<point>  facepoints {};
        // other vectors
    };

    struct sim_state {
        std::vector<double> cell_area {};
        std::vector<double> cell_inflow {};
        // other vectors
    };

    struct solver {
        mesh      m_mesh {};
        sim_state m_state {};
        // other vectors
    }
}
```

Listing 4.1: Simulation data-structures

Without our abstractions, a simple implementation would use raw pointers and store their corresponding sizes with paired-up variables. Manually implementing a loop to copy data from the vectors to the GPU structures is repetitive and error-prone (Listing 4.1). This also requires

implementing a copy back to the host after processing is complete.

We can use our vector replacement (Listing 4.2) in place of `std::vector`, and have `solver` inherit from `new_delete` (Listing 4.3). We can then use `std::unique_ptr` to store our solver object. Copying these vectors onto the GPU is a simple case of assignment, since we have implemented these operators in our custom vector. For other types, such as Thrust's vectors, equivalent operators can be defined by the user once and then reused everywhere.

```
namespace gpu {
    struct mesh {
        cup::vector<cell>   cells {};
        cup::vector<face>   faces {};
        cup::vector<point>  facepoints {};
        // other items

        mesh& operator=(const cpu::mesh& cpu_mesh) {
            this->cells = cpu_mesh.cells;
            // and so on!
            return *this;
        }
    };

    struct solver {
        mesh      m_mesh {};
        // other items

        solver(const cpu::solver& cpu_solver) {
            this->m_mesh = cpu_solver.m_mesh;
            // copy-assign other members too
        }
    };
}
```

Listing 4.2: Copying data to GPU

```
struct solver;
using managed_allocator = cup::managed_allocator<solver>;

struct solver
    : public cup::new_delete<solver, managed_allocator> {
    mesh      m_mesh {};
    sim_state m_state {};
    // other vectors
}

auto gpu_solver = std::make_unique<solver>(/* args */);
```

```
12 kernel<<<>>>(gpu_solver.get());
```

Listing 4.3: RAII solver initialization

Ideally, we want a kernel that can just accept this object and access the various vectors to process the simulation. CUDA allows `__device__` methods to be class members, but not `__global__` kernel functions. This allows the solver object to be passed as-is to the GPU, and methods to be invoked on it. To iterate over all the cells, we can use our adaptor that implements the grid-stride technique, and we now have code that reads like code on the CPU, and clearly expresses our intent. We have bounds checking, convenient iteration over all items, and we were trivially able to pass in our solver object into the GPU kernel. If users prefer customization of allocation strategies, they only need to implement an allocator and reuse it everywhere. To implement a different iteration strategy, they only need to implement an adaptor and reuse it where needed. This leads to robust, self-documenting code that improves maintainability (Listing 4.4).

```
1  __global__ void update_velocity(solver* solver) {
2      // we have access to all the state here
3      auto& cells = solver->m_mesh.cells;
4
5      // robust parallelization
6      for(auto& cell : cup::grid_stride_adaptor(cells)) {
7          // process cell
8          auto faces = cell.faces;
9          cell.update_velocity();
10     }
11 }
```

Listing 4.4: Kernel to update velocity

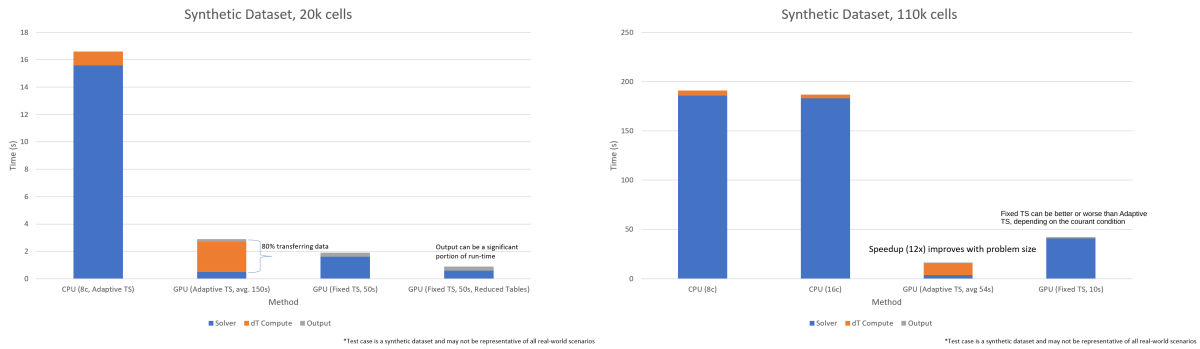
By implementing kernel logic as methods of a class, users can use all of the state of the class object if needed, which drastically simplifies writing kernels. This minimizes the need for kernels that take multiple parameters, and offers one ground-truth where the workings of the solver can be inspected. Kernels, then, simply parallelize this over all items to be processed.

4.6 Implementation results

We implemented a successful port of one of the solvers in HEC-RAS, and preliminary tests show performance gains over using the CPU. It is important to note that the comparisons here

are between CPU code using C#, and GPU code with C++ and CUDA. This does not offer insight into the overhead of the abstractions, but does offer some insight into the advantages of using the GPU.

We tested on a system with an Intel Xeon W-2265 CPU and an RTX 5000 GPU. The simulation advances in timesteps; we tested with adaptive and fixed timesteps on the GPU, and with adaptive timesteps on the CPU. CPU simulation with fixed timesteps take significantly longer to complete, and are hence not compared here. When adaptive timesteps are used, the simulation computes the time-delta to use for the next timestep based on the current state (e.g., velocity of water in all the cells). The adaptive timestep approach takes fewer steps overall to complete the simulation, but there is some cost associated with evaluating the prior timestep to compute the time-delta for the next adaptive step. Since the CPU and the GPU use identical mathematical models, the average timestep ("avg. TS" in the figure) is identical when adaptive timesteps are used. We tested both implementations with a synthetic dataset (not fully representative of real-world models) containing 20,000 and 110,000 cells with regularities that are naturally advantageous for the GPU implementation, and noticed a 3x and 12x improvement respectively (Figure 4.1[2]) in end-to-end execution-time of the simulation, which also demonstrates performance scaling with problem size.



(a) Synthetic dataset, 20,000 Cells (b) Synthetic dataset, 110,000 Cells

Figure 4.1: CPU vs. GPU performance

[2]"Reduced tables" is a technique that stores fewer data-points in the metadata for the unstructured grid, sacrificing some accuracy for reduced memory usage.

4.7 Summary of implementation

To conclude, let us summarize the implementation which added GPU acceleration to HEC-RAS, from a high-level overview to how the techniques were used.

We exposed the solver as a C API—symbols are exported from a shared object, which are then imported into C# using its marshalling interface. Though exposed as a C API, the underlying state is managed in a C++ structure. A pointer to the class (or structure) instance is returned to the C# code as a handle, which must be passed as a function argument to the C API.

To simplify memory management, we allocate and free all memory within the shared library. C# code uses the solver API to allocate (and free) memory, which is freed at the end of the application. This also overcomes the challenges of memory management across shared-object boundaries and ensures binary compatibility across versions.

In C++, we manage the state of the solver through a structure that contains data required by the solver. This uses the custom allocator (see Listing 3.20) to ensure that all of its member objects are allocated on the CUDA managed heap.

Conditional compilation (or code-generation) to implement optimized routines for specific GPU architectures can be enforced (or a fallback implementation can be used) without macros (see section 3.1), making the code that much more easier to reason about. Our technique does not use macros in the codepath, ensuring that the compiler can verify syntactic validity of all branches.

Vectors that are referenced both on the host and the GPU use the vector abstraction descried in section 3.4. This allows the host code to construct and initialize data directly on GPU-addressable memory, simplifying the implementation. Where applicable, we use the grid-stride adaptor (see section 3.5) to parallelize processing of items in GPU kernels. This enables us to enforce strict application-driven bounds checking when necessary. Our vector also conforms to the requirements of a C++ random-access container, ensuring interoperability with libraries such as Thrust.

While not explicitly presented here, a non-owning container can be implemented with the same techniques presented in section 3.4, allowing one to use iterators in kernels while ensuring performance; discussed in subsection 3.4.2 and subsection 3.4.3.

Since the entire state of the solver is available on the GPU, we can use advanced techniques such as GPU-driven kernel-launches, affording performant conditional kernel scheduling. In particular, we can use adaptive time-steps—kernels can be launched again with a smaller timestep if the computation is deemed unstable.

Chapter 5

Future work

In this chapter, we discuss future work that we planned to implement in the abstraction library and in HEC-RAS.

5.1 Future work in abstractions

Our work has been implemented as a library[1] and can be integrated into any project that uses CMake[2] as its build system. While the library currently implements simplified examples of the techniques presented that can be used in any project, a few implementation details were left out, in the effort to make a viable proof-of-concept and demonstrate the utility by using it in a real-world project.

The C++ standard is the reference against which compilers and the "standard library" are built upon and is designed by experts. The vector, iterator, and allocator do implement the minimum required components to interoperate with abstractions that use the interface, and further work is required to implement the remaining subset of features. The iterator, as implemented, requires that the it be convertible to a pointer-like type. This is not strictly necessary and any iterator satisfying the `std::random_access_iterator` concept will work but may come with performance overhead when not convertible to a pointer. Similarly, our vector implementation does not currently offer a constructor to facilitate aggregate initialization, and convenience-methods to `insert`, `emplace`, or `push_back` items. Implementing

[1] Available on GitHub: github.com/mythreyak/cup

[2] CMake: CMake is an open source, cross-platform family of tools designed to build, test, and package software. cmake.org

`push_back` in a performant way requires picking one of many different approaches, each with different performance characteristics; when a vector is backed by device-local memory, a `push_back` operation from the CPU could either cache the values in an intermediate buffer on the CPU and then append all values at once, or copy them immediately to the GPU. We do not support resizing the vector from kernel code. While CUDA does offer `cudaMalloc` and `malloc` in device code, the memory allocations are not inter-compatible [36]—memory allocated from one cannot be freed by the other. Moreover, memory allocated with `cudaMalloc` on the device must not be freed with `cudaFree` on the host and vice-versa [30]. This implies that resizing a vector allocated with CPU-side CUDA APIs due to an operation in a kernel is not possible, unless host code is invoked alongside a retry mechanism in the kernel. For users looking for a more feature-complete implementation, stdgpu [47] might be of interest.

As discussed in section 3.3, our allocator is synchronous—it does not utilize asynchronous allocation strategies, and does not incorporate techniques to reuse memory allocations across kernel launches. Many HPC applications use memory pools to improve memory reuse, prevent systems calls and their associated performance penalties, and improve issues related to memory fragmentation. While we do not implement such techniques, our allocator can be used as the underlying allocator that allocates memory managed by these higher-order techniques. Stream-ordered memory-allocators [39] and the virtual-memory-management API [43] can be used to implement more advanced memory-management strategies.

The Clang compiler allows users to implement overloads based just on the execution space of the function declaration. For example, a variation of this technique using the CUDA `nvcc` compiler allows us to prevent users from using the indexing operator on the host with our custom vector when it is backed by device-local memory. Further work is required to explore how this technique can be leveraged to solve other common problems that CUDA users face.

Coroutines were not implemented as it was not required for porting HEC-RAS. Using coroutines to launch and retrieve the results of a kernel requires further investigation. This approach could be used to either launch kernels eagerly or construct a CUDA graph that is executed for maximum performance.

5.2 Future work in HEC-RAS

While the port was successful and demonstrated the advantages and brought forth clear benefits of using the GPU, there definitely are improvements that can be made to make it suitable long-term.

We currently do not have an implementation that uses the classical approach of writing kernels. Since our abstractions themselves do not impose any requirements on the kinds of algorithms or data-layouts to use, they can be freely changed. Our abstractions make it easier to build these data-structures—e.g., our vector can be used in any part of the data-structure that requires a contiguous buffer, simplifying implementation.

To adequately test the runtime and compiler performance overhead of our abstractions, a full implementation using the classical approach may be required, even though we have shown that the abstractions generate identical assembly in simple test cases. The current CPU implementation uses C#, so the performance comparisons we presented do not offer insight into the overhead of the abstraction themselves. Nevertheless, our GPU port does offer a significant speedup over the parallelized CPU implementation.

The port currently does not use GPU-optimized data-structures in all hot-paths of the simulation and therefore has many opportunities for improvement. In particular, indexing the faces of the cell does not follow the coalesced access-pattern—completely antithetical to the programming principles of writing applications for the GPU. Future work includes optimizing the access-patterns of the data by improving spatial locality, moving to a GPU-first data-structure, and perhaps using it on the CPU as well to prevent expensive data-format changes during data-exchange.

We have successfully used the library in HEC-RAS to port the application and expose an interface to C#, simplifying maintenance. We currently synchronize the application (with `cudaDeviceSynchronize`) after every "frame" of the simulation, or every few frames for metrics reporting and visualization. This has major performance implications, as we are essentially waiting for the GPU to idle before scheduling more work. Furthermore, HEC-RAS can use the data present on the GPU to render the visualization to prevent the synchronization step. Graphics APIs such as Vulkan offer interoperation capabilities with CUDA that allow them to

reference CUDA buffers in traditional graphics pipelines.

Chapter 6

Conclusion

In this thesis, we demonstrated a few techniques that make use of the CUDA C++ language features to simplify CUDA programming. We evaluated the advantages and disadvantages of each approach by discussing the nature of the code generated, which offers insight into hardware-limits and efficiency. While high-level performance benchmarks are extremely useful (which we expect to conduct in future work), we wanted to acknowledge the finer details such as register pressure and assembly overhead that are immediately relevant to the porting process—one that empowers the user to make decisions on implementation details.

We saw how modern C++ can be utilized to make a good and worthwhile attempt at solving and addressing some of the major sharp-edges of a language built upon C++—a language known for its nuances. We saw how tools such as godbolt.org can be used to inspect the generated assembly and make an informed decision on whether an abstraction is capable of generating optimized assembly. These tools can be used to regularly inspect the generated assembly as the codebase evolves and ensure that any code changes or addition of abstractions that reduce code-generation quality can be caught and addressed.

C++ has evolved over the years to include features requested by the community to improve productivity and maintainability of code. The CUDA programming-model introduces more opportunities for programmers to make mistakes, on top of the C++ language, which already has notoriety for having a lot of edge-cases that one must be careful about. We explored how existing C++ language and library features, along with the changes CUDA makes to the language can be utilized to improve user experience.

In addition to the CUDA compiler `nvcc`, the Clang compiler also supports compiling CUDA applications, though it often lags behind in the version of the CUDA toolkit it officially supports. Clang has introduced additional features (such as overloads based on execution space) that may be incompatible with CUDA libraries, but since the driver framework can utilize PTX for kernel objects, interoperability is feasible (depending on the compiler options). The Clang compiler also offers a set of tools based on the `llvm` framework that might be of great interest to users—especially their support for "optimization remarks" [7] which gives descriptive diagnostics on why certain optimizations were not performed. The ability to use two different compilers offers interesting insight into the language (and compiler) evolution—one driven by an organization and another driven by the community, already diverging slightly in the features they offer. We may see more features being offered in both compilers, perhaps convergent in some aspects and divergent in others, which opens up interesting opportunities for implementations using CUDA.

REFERENCES

[1] Altimesh. Compiler targeting NVIDIA GPU from C#. URL https://www.altimesh.com/hybridizer-essentials/.

[2] D. A. Beckingsale, M. J. McFadden, J. P. S. Dahm, R. Pankajakshan, and R. D. Hornung. Umpire: Application-focused management and coordination of complex hierarchical memory. *IBM Journal of Research and Development*, 64(3/4):00:1–00:10, 2020. doi: 10.1147/JRD.2019.2954403.

[3] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas RW Scogland. RAJA: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81, 2019. doi: 10.1109/P3HPC49587.2019.00012.

[4] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2, chapter 26, pages 359–371. Morgan Kaufmann, October 2011. doi: 10.1016/B978-0-12-385963-1.00026-5.

[5] Ian Buck, T. Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004. doi: 10.1145/1015706.1015800.

[6] clang. Clang—overloading based on __host__ __device__ attributes, . URL https://llvm.org/docs/CompileCudaWithLLVM.html#overloading-based-on-host-and-device-attributes.

[7] clang. Remarks—LLVM documentation, . URL https://llvm.org/docs/Remarks.html.

[8] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129 vol.2, Jan 2000. doi: 10.1109/DISCEX.2000.821514.

[9] CppReference. Cppreference documentation—allocator, . URL https://en.cppreference.com/mwiki/index.php?title=cpp/named_req/Allocator&oldid=155496.

[10] CppReference. Cppreference documentation—allocator requirements, . URL https://en.cppreference.com/mwiki/index.php?title=cpp/named_req/Allocator&oldid=155496#Allocator_completeness_requirements.

[11] CppReference. Cppreference documentation—concept, . URL https://en.cppreference.com/mwiki/index.php?title=cpp/language/constraints&oldid=153014.

[12] CppReference. Cppreference documentation—container, . URL https://en.cppreference.com/mwiki/index.php?title=cpp/container&oldid=154488.

[13] CppReference. Cppreference documentation—if constexpr, . URL https://en.cppreference.com/mwiki/index.php?title=cpp/language/if&oldid=154736#Constexpr_if.

[14] CppReference. Cppreference documentation—raii, . URL https://en.cppreference.com/mwiki/index.php?title=cpp/language/raii&oldid=140349.

[15] CppReference. Cppreference documentation—range-based for loop, . URL https://en.cppreference.com/mwiki/index.php?title=cpp/language/range-for&oldid=151897.

[16] Eyal Rozenberg. cuda-api-wrappers. URL https://github.com/eyalroz/cuda-api-wrappers.

[17] P. García-Navarro, J. Murillo, J. Fernández-Pato, I. Echeverribar, and M. Morales-Hernández. The shallow water equations and their application to realistic cases. *Environmental Fluid Mechanics*, 19(5):1235–1252, Oct 2019. ISSN 1573-1510. doi: 10.1007/s10652-018-09657-7. URL https://doi.org/10.1007/s10652-018-09657-7.

[18] Pradeep Gupta. CUDA refresher: The CUDA programming model. URL https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/.

[19] Mark Harris. CUDA pro tip: Write flexible kernels with grid-stride loops. URL https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/.

[20] Holger Jones, David Poliakoff, Peter Robinson , David Beckingsale, Riyaz Haque, Adam Kunen. CHAI: Copy-hiding array abstraction to automatically migrate data between memory spaces. URL https://github.com/LLNL/CHAI.

[21] Mariia Krainiuk, Mehdi Goli, and Vincent R. Pascuzzi. oneAPI open-source math library interface. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 22–32, Nov 2021. doi: 10.1109/P3HPC54578.2021.00006.

[22] Lewis Baker. CppCoro—a coroutine library for c++. URL https://github.com/lewissbaker/cppcoro.

[23] LLNL. RAJA performance portability layer. URL https://github.com/LLNL/RAJA.

[24] Marcel Koester. ILGPU JIT compiler for high-performance .net GPU programs. URL https://github.com/m4rs-mt/ILGPU.

[25] Mark Harris. Hemi: Simpler, more portable CUDA c++. URL https://github.com/harrism/hemi.

[26] Michael Kunz. ManagedCUDA aims an easy integration of nvidia's cuda in .net applications written in C#, visual basic or any other .net language. URL https://github.com/kunzmi/managedCuda.

[27] Microsoft. Type marshalling in .NET. URL https://learn.microsoft.com/en-us/dotnet/standard/native-interop/type-marshalling.

[28] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, March/April 2008. doi: 10.1145/1365490.1365500.

[29] NVIDIA. libcu++: The c++ standard library for your entire system, . URL https://github.com/nvidia/libcudacxx.

[30] NVIDIA. CUDA docs—api reference, . URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#api-reference.

[31] NVIDIA. NVIDIA documentation—virtual architecture macros, . URL https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#virtual-architecture-macros.

[32] NVIDIA. NVIDIA documentation—c++ language extensions, . URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#c-language-extensions.

[33] NVIDIA. NVIDIA documentation—cuda compilation trajectory, . URL https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#the-cuda-compilation-trajectory.

[34] NVIDIA. NVIDIA documentation—relaxed constexpr, . URL https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#expt-relaxed-constexpr-expt-relaxed-constexpr.

[35] NVIDIA. CUDA docs—__global__ function argument processing, . URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global-function-argument-processing.

[36] NVIDIA. CUDA docs—dynamic global memory allocation and operations, . URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#dynamic-global-memory-allocation-and-operations.

[37] NVIDIA. CUDA docs—differences between host and device, . URL https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#differences-between-host-and-device.

[38] NVIDIA. NVIDIA documentation—programming model, . URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model.

[39] NVIDIA. NVIDIA documentation—stream ordered memory allocator, . URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#stream-ordered-memory-allocator.

[40] NVIDIA. NVIDIA documentation—unified memory, . URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#unified-memory-programming.

[41] NVIDIA. NVIDIA documentation—unified memory with system allocator, . URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-system-allocator.

[42] NVIDIA. NVIDIA documentation—classes and virtual functions, . URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#virtual-functions.

[43] NVIDIA. NVIDIA documentation—virtual memory management, . URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#virtual-memory-management.

[44] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005. URL https://escholarship.org/uc/item/4nq8h63h.

[45] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008. doi: 10.1109/JPROC.2008.917757. URL http://escholarship.org/uc/item/0cv1p1nc.

[46] Kerry A. Seitz, Jr., Theresa Foley, Serban D. Porumbescu, and John D. Owens. Supporting unified shader specialization by co-opting C++ features. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 5(3):25:1–25:17, July 2022. doi: 10.1145/3543866. URL https://escholarship.org/uc/item/3127f66s.

[47] P. Stotko. stdgpu: Efficient STL-like data structures on the GPU. arXiv:1908.05936, August 2019. URL https://arxiv.org/abs/1908.05936.

[48] U.S. Army Corps of Engineers Hydrologic Engineering Center (HEC). HEC-RAS: 2D unsteady flow hydrodynamics, . URL https://www.hec.usace.army.mil/confluence/rasdocs/ras1dtechref/latest/theoretical-basis-for-one-dimensional-and-two-dimensional-hydrodynamic-calculations/2d-unsteady-flow-hydrodynamics.

[49] U.S. Army Corps of Engineers Hydrologic Engineering Center (HEC). HEC-RAS, . URL https://www.hec.usace.army.mil/software/hec-ras/.