Performance Portability of CUDA Across NVIDIA GPU Architectures

Timothy P. Coyne

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science & Applications

Dimitrios S. Nikolopoulos, Chair

Adrian Sandu

Wu-chun Feng

May 5, 2025

Blacksburg, Virginia

Keywords: GPU, CUDA, Performance Portability

Performance Portability of CUDA Across NVIDIA GPU Architectures

Timothy P. Coyne

(ABSTRACT)

Graphics Processing Units (GPUs) provide impressive parallel performance that makes them invaluable to a number of computational workloads such as machine learning, simulations, and many others. NVIDIA GPUs currently outperform all of their competitors and thus make up the lion's share of today's market. Importantly, they are natively programmed using the proprietary framework Compute Unified Device Architecture (CUDA), which only compiles to machine code for NVIDIA hardware. Moreover, NVIDIA releases a new GPU with an updated architecture roughly every two to three years. Since CUDA is commonly forward compatible with the next generation of GPUs, it is natural to reuse CUDA code built for a previous architecture on a newer one. Unfortunately, the performance of CUDA applications from one architecture to the next does not necessarily benefit from the newer generation of GPUs. This work investigates a variety of CUDA workloads that fail to show a performance uplift moving from the V100 to A100 GPUs. While some kernels perform as expected, others exhibit up to a 700% performance drop when running on the newer architecture. For each, an analysis of the benchmarks is provided, and for some, a direct solution for improving performance portability is highlighted, where possible. These issues are also cross examined to provide a few holistic portability concerns. At the end, a set of programmer recommendations are made to assist developers in more easily maintaining performance portability between architectures.

Performance Portability of CUDA Across NVIDIA GPU Architectures

Timothy P. Coyne

(GENERAL AUDIENCE ABSTRACT)

Graphics Processing Units (GPUs) provide high parallel performance by executing instructions across many smaller internal compute units. This high parallel performance greatly benefits numerous workloads, including machine learning, simulations, and many others. NVIDIA currently has the largest market share of GPUs, which are natively programmed using Compute Unified Device Architecture (CUDA). The company typically releases a new family of GPUs with updated architectures every 2 to 3 years. Given that CUDA is the standard language for programming these GPUs and the relatively high frequency at which new architectures are released by NVIDIA, it is essential for CUDA applications to exhibit strong performance portability. In other words, a new GPU should provide uplift for pre-existing kernels proportional to its generational improvements. Unfortunately, this is not always the case which means developers must sometimes retrofit their code in order to obtain optimal performance. This research investigates the performance portability of a number of different workloads and provides a set of programmer recommendations to assist developers in maximizing performance portability.

Dedication

This thesis is dedicated to my loving father, Patrick J. Coyne.

You were my rock in this life,

and now you are my guardian angel.

You will be missed, dearly.

Acknowledgments

Firstly, my principal advisor Dr. Nikolopoulos has been instrumental in my research journey. I am deeply grateful for his suggestions, feedback, and critically valuable insights. I would also like to thank Dr. Sandu and Dr. Feng for serving on this committee. Specifically, Dr. Sandu's Advanced Parallel Programming course electrified my studies on CUDA as it was the first proper instruction I had on GPUs.

I am also immensely grateful to my family for always lending me their ears with their constant support. They were always innately curious about my ideas and progress on this work. Lastly, my father's support over the semesters was essential in reaching the finish line, and the success of my final semester is owed to both the countless hours we conversed and his belief in my work.

Contents

List of Figures

List of Tables

List of Abbreviations

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DGEMM Double-precision GEneral Matrix-Matrix Multiplication

GPGPU General Purpose Computing on Graphics Processing Units

GPU Graphics Processing Unit

PCIe Peripheral Component Interconnect Express

Slurm Simple Linux Utility for Resource Management

SM Streaming Multiprocessor

SXM Server PCI Express Module

Chapter 1

Introduction

1.1 Motivation

In the previous twenty years, the rise of general-purpose computing on graphics processing units (GPGPU) has been tremendous. A myriad of industries now utilize GPUs for fulfilling their computational needs, including engineering, biology, finance, and many more [1]. In order to harness this power, NVIDIA created Compute Unified Device Architecture (CUDA) to allow developers to build software on their new hardware. CUDA is a very mature and feature-rich framework which provides developers with a robust experience in programming NVIDIA GPUs.

CUDA supports compiling for NVIDIA GPU targets only. This presents the first portability concern where code written in CUDA cannot be reused for other capable GPUs like those provided by Advanced Micro Devices (AMD) or Intel. This has brought about a number of frameworks that provide this capability like OpenCL, Kokkos, and Raja. However, the problem is not exclusive to reusing CUDA code between GPUs from two different companies. In the spirit of Moore's Law [2], a new NVIDIA GPU is released roughly every two or three years with a brand new architecture. For a recent example, one of the most advanced and latest enterprise GPUs released by NVIDIA is the A100, which superseded the V100 about three years after its release [3] [4]. According to the press release, the A100 GPU can be up to 25x higher performing than the V100. While a boisterous claim that is likely only true in

specific workloads, the expectation from customers will be that this newer GPU is at least higher performing than its predecessor overall. Since CUDA natively supports both GPUs, one may assume that simply recompiling for the new architecture is all that is required to reap its benefits, but this is not the case.

A straightforward example of poor performance portability can be found in the turboSETI project put out by the Search for Extraterrestrial Intelligence (SETI) center at the University of California, Berkeley. TurboSETI excels at finding narrow band drifting signals amongst background filterbank data. In other words, it searches for radio waves that change frequency in a narrow range over time. This is similar to how communications are transmitted on earth, so the idea is that if signals originating from outer space mimic this pattern, it could indicate intelligent life broadcasting data. TurboSETI utilizes CUDA to accelerate this analysis.

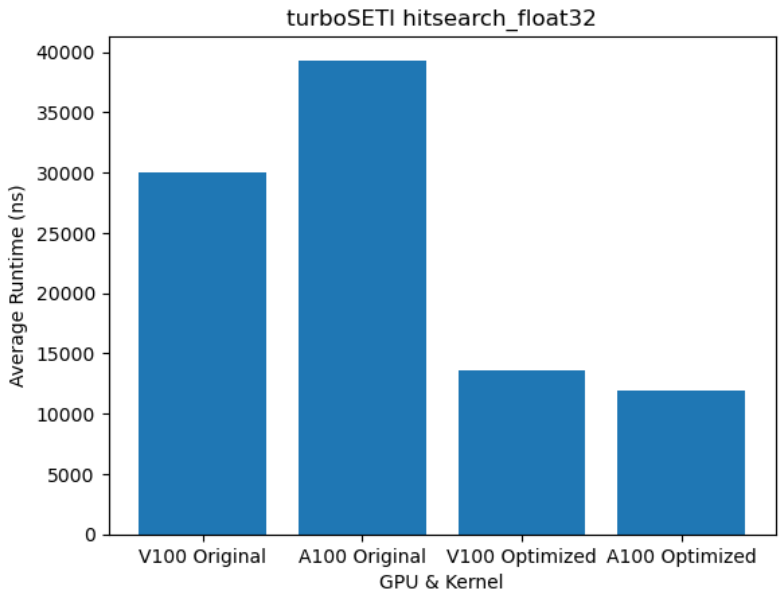


Figure 1.1: Performance of turboSETI

When examining its runtime, there is a notable drop in performance from the V100 to A100. Without making any changes to the code, one would be quite disappointed if their workload consisted of mostly turboSETI on their new GPU. This unexpected performance discrepancy

between the V100 and A100, prior to optimizations, is the crux of the problem. In this case, optimization successfully improved the performance of both GPUs, and it allowed for the A100 to take the lead. While proper tweaking can provide performance uplift in some cases, guidance for developers in this area is lacking for both identifying and remedying these problems.

1.2 Contributions

Since there is limited research on the performance portability of CUDA between NVIDIA architectures themselves, this work is meant to serve as an exploration of the variety of different codes and real-world applications that yield varying degrees of performance portability problems within the CUDA ecosystem. Given the regular releases of new NVIDIA GPUs, strong performance portability is important for customers who do not want to rewrite their kernels every generation. To improve the situation, this work aims to answer a few potent questions.

- What is the level of performance portability between successive modern NVIDIA GPUs?
- Can the demonstrated performance portability be improved if it is not already adequate?
- What recommendations can be provided to developers looking to maximize the intergenerational performance portability of their kernels?

Firstly, for each example, a step through of the code and performance discrepancies will be discussed. Next, some of the examples will include a solution to improve performance

portability or an explanation showing that there is no readily available alternative. Additionally, some level of architectural explanation is provided, if discovered. Lastly, a number of patterns evident across many of the examples are highlighted to showcase some higher-level performance portability concerns between the NVIDIA V100 and A100 GPUs. The combined analysis of these three components was used to build recommendations for developers to fully leverage the performance of new architectures.

Chapter 2

Background

2.1 NVIDIA Graphics Processing Units

A typical configuration of a GPU is as an add-on PCIe card or SXM module that acts as a co-processor. Communication is done through a driver running in the kernel space of the operating system. Before diving into CUDA and its performance characteristics, it is essential that the typical architecture of an NVIDIA GPU is discussed first.

2.2 Execution Model

2.2.1 Thread Hierarchy

NVIDIA GPUs consist of a number of Streaming Multiprocessors (SMs). Each SM can run many blocks, and each block contains a number of threads. Threads are additionally organized into groups of 32, previously 16 on older architectures, called a warp. A thread block is considered resident on an SM for the entire length of time it is being executed.

The aim of this architecture is to be able to parallelize thread execution as much as possible. Single instruction, multiple threads (SIMT) is the execution model that is being leveraged here with this design [5]. The scheduler will attempt to execute an entire warp per clock

cycle so long as the threads are ready to execute another instruction. For instance, a warp of threads waiting to execute a floating-point operation may be chosen for the next clock, but a warp of threads waiting on a read from global memory, known as a long scoreboard dependency, will not be.

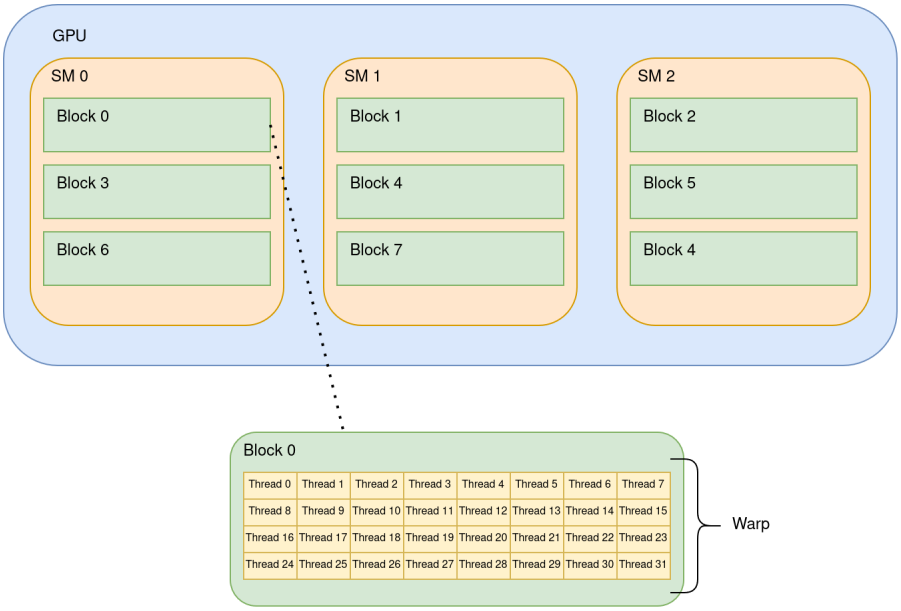


Figure 2.1: Thread Hierarchy Visual

Only a limited number of threads, warps, and blocks can be resident on a single SM at a time. If more blocks are scheduled to be executed than there is available capacity for them on the SMs, then some blocks will not execute until other resident blocks have finished. As a result, knowing the capacity of a GPU being programmed is essential for optimal performance. If too few blocks are being executed, the GPU may be underutilized, but if too many blocks are scheduled, some will have to wait. In practice, different workloads will require varying numbers of threads, which can limit potential optimization. As an example, a large matrix multiplication operation will inherently require a substantial thread count.

2.2.2 Memory Hierarchy

The memory hierarchy of an NVIDIA GPU provides a number of different levels of storage that each vary in performance, capacity, and ease of accessibility. At the lowest level, global memory provides a large amount of on-board memory that the GPU can access fairly quickly. For example, the NVIDIA V100 and A100 have as much as 32 and 80 GB of available global memory, respectively. While it is fully accessible by the GPU during kernel execution, the CPU is also able to access global memory before and after kernel execution, commonly for copying data. This is because global memory falls under the Unified Memory model that dictates global memory be accessible by both GPUs and CPUs.

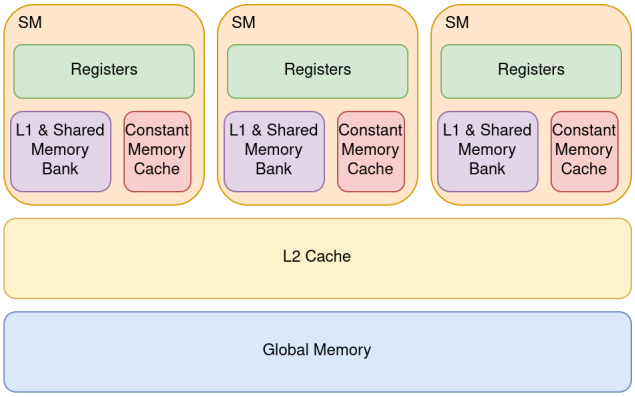


Figure 2.2: Memory Hierarchy Visual

After global memory, the performance of the higher levels is considerably faster. Next comes an L2 cache that is available to all SMs. Afterwards is shared memory which, according to NVIDIA, is about 100 times faster than uncached global memory [6]. Shared memory can be accessed directly in CUDA code unlike traditional caches, and it has the added benefit that it is as fast as an L1 cache. It is named shared memory because threads share an allocation at the block level. In other words, every resident block has its own separate shared memory allocation of which any of its threads may use. At the same level is the SM's L1 cache whose capacity is reduced by any shared memory allocations made.

There is also the constant memory space which consists of data the compiler deems it can optimize as a statically accessible constant. It is very small but very fast since each SM has its own cache. Lastly, each SM also has a register bank which provides the absolute fastest performing memory on the system. The amount of shared memory and registers required per thread limits how many of them can be present per block. If the total usage of either of these exceeds the capacity supported by the SM, the programmer must reduce the block's thread count. Importantly, this capacity is generally increased every generation which can lead to performance portability loss when older code is not aware of the increased available capacity.

2.3 Compute Unified Device Architecture

CUDA is the NVIDIA GPU programming framework built on top of modern C++. An important distinction made in CUDA is the difference between the host and device. The host represents the realm of the system's CPU and RAM while the device is the GPU and its memory. In order to execute a piece of device code, it needs to be organized into a kernel function. Kernel code must be launched by the host to start execution on a device. A kernel is sent to the GPU with launch parameters indicating how many blocks and threads should be spun up. Within the kernel, a thread can easily access its own numeric thread identifier based on its position in the block, its block's position within the grid, and the block count of the grid. Grids can be launched with between one and three dimensions. The following example illustrates a trivial compute kernel that simply calculates the square of each element within the *nums* array and outputs the value back to the same entry.

Listing 2.1: Trivial Squaring Function

```
__global__ void simple_square(long *nums, const int n)
{
    int tid = gridDim.x * blockIdx.x + threadIdx.x;

    if (tid < n) {
        long n = nums[tid];
        nums[tid] = n * n;
    }
}
```

In order to provide data to the device for use during execution, memory must first be allocated on the device and initialized by the host. Unified Memory allows for seamlessly creating the input data on the host and copying it over to the device's global memory.

Listing 2.2: Global Memory Allocation

```
int nums_host[N] = {2, 3, 7, ...}
int *nums_device = NULL;

cudaMalloc(&nums_device, sizeof(int) * N); // Allocate device
    memory

// Copy over the host values
cudaMemcpy(nums_device, nums_host, sizeof(int) * N,
    cudaMemcpyHostToDevice);
```

Launching the kernel is quite straightforward once the number of threads and blocks needed is known. Blocks cap out at 1024 threads on the latest GPUs, but they can contain as few as one thread each.

Listing 2.3: Kernel Launch

```
1  simple_square<<<CEIL(N / MIN(N, 1024)), MIN(N, 1024)>>>(
       nums_device, N);
2  cudaDeviceSynchronize(); // wait for GPU to finish execution
```

While it is not a hard and fast requirement to call *cudaDeviceSynchronize()* to wait for kernel completion, it is needed if the newly computed values are to be read by the host. They can be copied back over to the host the same way they were copied initially with *cudaMemcpy* by instead passing in *cudaMemcpyDeviceToHost* and swapping the pointers.

Chapter 3

Review of Literature

There is generally limited preexisting research focusing on the performance portability of CUDA between two different NVIDIA GPU architectures. Most other work is about benchmarking and analyzing the portability benefits of frameworks like SYCL through porting specific applications for execution on a diverse array of hardware.

3.1 Performance Portability

In order to understand and quantify *performance portability* between two separate systems, it is necessary to first define it. According to Pennycook, Sewall, and Lee, who have published a number of works on performance portability, their definition is as follows: "A measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set" [7]. The architectural efficiency of an application is defined as the ratio between the observed performance and the peak theoretical performance of the hardware. For example, if an application achieves 90% architectural efficiency on the V100 but 50% on the A100, this would indicate poor performance portability as the A100's architecture is not being leveraged properly. In general, a highly portable application would ideally be meeting the same architectural efficiency on all platforms. However, the work presented in this research does not quantify performance portability in the same way. Instead, the difference in mean runtime was the fundamental metric used to measure the performance

portability of a kernel as it provides a clear metric for comparison between runtimes. For example, if the mean runtime difference of the A100 against the V100 changes from 10% to -30% after optimizations, it's clear that performance portability increased tremendously since the A100 began exhibiting substantially higher performance than the V100. Pennycook et al.'s metric for performance portability was considered as a replacement, but it tends to yield low values for kernels with small launch grid sizes, which may distract from the performance portability problem at hand and the subsequent improvement gained through optimization. This is also true for computational problems that lack a highly efficient solution. In short, strong performance portability can be characterized as a rise in runtime from one architecture to the next that is proportional to the uplift the new architecture is expected to provide. Even without this information, an increase in the difference of mean runtime indicates improved performance portability compared to the contrary.

3.2 Intra-CUDA Portability

Published shortly after the release of the A100, a paper examined the performance portability of matrix operations within the MAGMA library [8], with many of its authors being heavily involved in the project. MAGMA is a high-performance linear algebra library developed for NVIDIA GPUs written in CUDA. The paper focuses on benchmarking and analyzing the performance of MAGMA compared to the highly optimized NVIDIA-produced cuBLAS library. The V100 and A100 GPUs were used since, at the time, the library was heavily optimized for the V100 and newly updated for the A100. The results are rather intriguing as they vary tremendously between applications. MAGMA generally outperformed cuBLAS in most of the tests, with only some exceptions including small-batched DGEMM.

The most troubling results presented in the paper are the sometimes very limited performance

gains evident with some of the workloads. For example, large batched DTRSV showed essentially the same performance between the A100 and V100. This is the crux of the problem that haunts CUDA developers: not achieving expected performance increases with newer hardware.

Shifting towards more of a hardware level focus, the amount of shared memory can generally be expected to increase with every new GPU architecture. For example, suppose a code is created that requires a large amount of shared memory such that each thread has 160 bytes to work with for this hypothetical computation. On the V100, which permits a maximum of 96 KB per SM, a block would be limited to 600 threads to fully occupy the SM. If this same code were to be run on an A100, it would be unnecessarily hampering itself by running with only 600 threads since the A100 can support up to 164 KB per SM. A compiler for automatically optimizing shared memory utilization and adjusting thread workload was built by Ivanov et al. to mitigate this performance portability concern [9]. Similarly, static analysis of code was explored to optimize kernel launch parameters by Lim, Norris, and Malony [10]. This is important since an underutilized GPU will leave performance on the table, and newer architectures typically contain more SMs.

3.3 Portability Frameworks

Having not to write application code multiple times for every different hardware vendor can save a lot of time. However, doing so without sacrificing performance portability is not without its challenges. There have been many frameworks developed over the years to combat this trade-off, but one of the larger and most mature ones is certainly OpenCL. It has native driver support on NVIDIA, AMD, and Intel GPUs [11]. Going back to the creation of OpenCL in 2009, portability between devices was the major motivation for developing

the framework as the Khronos Group states that OpenCL is meant to "... enable parallel processing for maximum performance and portability across devices" [12]. As is still the case today, CUDA could only run on NVIDIA GPUs which left other capable hardware behind. Back then, OpenCL was just beginning to be tested for its performance portability potential, and research from that time suggested the framework sometimes even outperformed CUDA on both the GTX 280 and GTX 480, so long as both were similarly optimized [13]. In more recent times, however, the portability of OpenCL has come into question, specifically between different vendors [14].

There has been a plethora of research published surrounding the portability of additional compute frameworks too. Some examples include a paper implementing the Smith-Waterman Algorithm in CUDA and SYCL [15], one implementing Particle-In-Cell simulation code in CUDA and SYCL [16], and another implementing multi-material simulation kernels in CUDA, Kokkos, OpenACC, OpenMP, and SYCL [17]. The overall question of whether these cross-platform frameworks provide high levels of performance portability is complex. In some cases, these papers, like the aforementioned ones, showcase good performance portability between device vendors, while other times not so much [18]. Of course, any framework which can correctly execute a compute kernel across a range of devices provides better performance portability than NVIDIA-only CUDA. Nonetheless, it's typical that device-specific optimizations are necessary to maximize throughput, so the initial state of a kernel is commonly not performance portable [19]. For the papers that include multiple NVIDIA GPUs in their testing, such as the Smith-Waterman and multi-material works, frameworks like SYCL and Kokkos, when compared against CUDA, provide unimproved performance portability between NVIDIA GPU architectures.

Benchmarking the performance portability of fundamental CUDA operations and concepts—such as butterfly reduction, scanning algorithms, and bank conflicts—sets this research apart

from similar works. Given the lack of comprehensive performance portability research on CUDA, this work seeks to fill the void by not only providing new data but also recommendations to aid developers in improving performance portability between NVIDIA GPUs.

Chapter 4

Design

4.1 System

This work focuses on benchmarking two of NVIDIA's enterprise-grade GPUs: the V100 and A100. These two successively released GPUs were the top of their respective generations. They are also heavily used in datacenters, making them a quintessential pair for studying the performance portability of NVIDIA GPUs. They each come in multiple different configurations, but the ones available for this research were the V100 PCIe with 16 GB of global memory and A100 SXM with 80 GB of global memory. A chart is provided to illustrate some key differences between the two, which are important to understand when comparing performance discrepancies.

Table 4.1: Reference Specifications for the NVIDIA V100 & A100

Specification	V100	A100
SMs	80	108
Global Memory	16 GB	80 GB
Shared Memory per SM	96 KB	192 KB
Max Threads per SM	2048	2048
Memory Bandwidth	900 GB/s	1555 GB/s

4.1.1 Slurm & Code Benchmarking

This research was conducted using systems owned by Virginia Tech that are shared with the entire university. In order to effectively share GPUs, a reservation system known as Slurm is employed on all of these systems. Users must first interact with Slurm in order to run a job on one of these GPUs. Since there are a number of A100 GPUs that Slurm can select for a job, these results may have been taken from different GPUs. The same is true for the V100 that was instead accessed from my research group's server. Therefore, these results are not taken from identical underlying systems. However, such as is the case with Berkeley National Laboratory too, different GPUs do not typically have identical system specifications in high performance computing environments [20]. Logically, this makes sense since the GPU would be sold with the cutting-edge servers available at the time. Since GPUs are being benchmarked and not the underlying host hardware, this should have little effect on the results.

Table 4.2: NVIDIA V100 & A100 Systems

Specification	V100 Cluster	A100 Cluster
Operating System	Ubuntu 22.04.5	Cent OS 7
Driver	550.120	535.129.03
NVCC	12.3/compiler.33567101_0	12.3/compiler.33567101_0
C++ Standard	11	11
CPU	AMD EPYC 7251	AMD EPYC 7742

4.2 Profiling

Benchmarking GPUs necessitates greater care when compared to CPUs. The time it takes to launch, execute, and await completion cannot simply be measured in-line like host code. For instance, simply measuring the difference in time between right before a kernel is launched

and right after yields a value that includes the time it takes to schedule the kernel execution on the GPU, synchronization after execution finishes, and much more. Therefore, GPU-specific software is required to get accurate results, especially when analyzing performance differences at the nanosecond level.

4.2.1 Nsight Systems

Kernel runtime was measured using NVIDIA's Nsight Systems utility. While vastly powerful, it excels at simply obtaining accurate nanosecond-resolution runtime measurements. It does not attempt to modify the behavior of the GPU in any way, so benchmarking runtime this way can be helpful to determine real-world performance. NVIDIA generally recommends Nsight Systems to be the first line of defense in investigating performance issues with CUDA code.

Depending on the specific application or code being tested, all kernels were run for anywhere between 500 and 100000 iterations, since benchmarking with an insufficient number of runs can introduce data integrity issues. More specifically, patterns that show up later on in execution will require the longer profiling time more iterations provide. Benchmark results that indicate differing behavior at the start of execution versus later on are split into two parts. The beginning of the kernel's execution will be referenced as its initial state and the rest of the time will be its steady state. The exact delineation between these two will vary per code but is generally where the performance significantly changes after some amount of time.

Given that the difference in mean runtime between two GPUs is the metric used in this work to quantify performance portability, the extensive runtime data provided by Nsight Systems makes it the quintessential tool for determining a kernel's level of performance portability.

4.2.2 Nsight Compute

Once runtime is fully analyzed, more detailed profiling information about how a kernel is executed on a GPU can be obtained with NVIDIA's Nsight Compute. This utility provides a plethora of information like memory access patterns, warp stall reasons, throughput information for roofline analysis, and much more. All of these values will be used in evaluating the performance of an application.

Memory access information provided by Nsight Compute that will be of particular use includes when memory is not coalesced and not free of bank conflicts. Stall reasons are also incredibly useful for determining where a code is getting stuck. In 2.2.2, it is described that a warp of threads will be executed together if they are all ready for the next instruction. When a warp cannot be executed due to some dependency, this is known as a stall. These can occur for a number of different reasons, but a non-exhaustive list is provided below.

Table 4.3: Selected Warp Stalling Causes

Stall Reason	Description
Barrier	Stuck at synchronization point such as a ___syncthreads()
Drain	Pending thread exit waiting on memory operation to complete
Long Scoreboard	Waiting for higher latency operation like accessing global memory
Short Scoreboard	Waiting for lower latency operation like accessing shared memory

Compute and memory throughputs are provided by Nsight Compute. The compute throughput is defined as the measured utilization of the SMs over their theoretical maximum,, and memory throughput is the measured utilization of all on-board memory systems. L1 cache, L2 cache, and DRAM throughput are provided separately.

By analyzing the different metrics that Nsight Compute provides, pinpointing potential areas that could be contributing to poor performance portability becomes significantly easier. As a result, it provides crucial aid in developing alternative kernels with improved performance

portability where static analysis of the code is impractical.

Chapter 5

Evaluation

5.1 Evaluation Process

Determining the performance portability of a given code is a multi-step process. The very first objective is to determine if there is an unexpected runtime discrepancy between the two GPUs on the same code. Nsight Systems was used to quickly and effectively profile the runs. It provides the mean, median, standard deviation, and individual runtime statistics for each kernel invocation inside the tested application. Next, this data is plugged into a custom Python script which reads the raw runtime data points provided by Nsight Systems to provide a table and graph. The graph contains the individual runtime data points over time for each kernel, and the table provides the aggregated runtime average. With these two visuals, performance as the GPU warms up is measured and analyzed by comparing the mean runtime between the GPUs.

For simulated codes that are not standalone applications, they were, for the most part, run with 1024 threads and 8, 16, 32, 64, 128, 160, 224, 256, 512, and 1024 blocks. Additionally, a V100 achieves full occupancy of all SMs when a kernel is launched with 160 blocks and 1024 threads, since each of the 80 SMs can house up to 2048 threads. Consequently, any launch smaller than this will result in all threads residing in SMs, but larger launches will force some threads to wait for SM availability. For many benchmarked algorithms that are outside the scope of an application, the number of threads used is always 1024. Previous work indicates

that the dimension of a kernel's grid has a significant impact on performance and requires per-architecture tuning [21]. However, the aim of this study is to examine algorithms and applications that exhibit unexpectedly low performance portability. In order to maintain consistency, kernels are usually launched with the same grid size on both GPUs. Importantly, code exhibiting poor performance portability with 1024 threads per block remains notable even if results vary with different grid dimensions. Therefore, benchmarking with the most optimal per-architecture grid size is left for future work.

After looking at runtime data, Nsight Compute was used to search deeper into why performance portability was not as expected for a given kernel. Warp stall reasons, cache hits, bank conflicts, and a number of other statistics are available here to help determine what's going wrong with the A100. More specifically, stall counts are available on a per-instruction basis that significantly improves the resolution at which performance portability comparisons can happen.

After thoroughly investigating a kernel that exhibited poor performance portability, the specific area generating the observed performance discrepancies had to be identified. The first line of defense was identifying any intrinsic, library call, or other code strategy that may have a modern replacement in the newer architecture. A more general approach that was also effective was using Nsight Compute to find lines of code producing higher rates of stalling on the newer architecture and investigating relevant alternatives or optimization strategies. Once all of these areas are identified, new code can be written that uses different strategies.

5.2 Butterfly Reduction

This section examines the performance portability of the hallmark butterfly reduction operation. GPUs require parallelized workloads in order to provide the best performance the hardware has to offer. As a result, the data that is needed to find the final solution of a computational problem is often split up and shuffled between the many different threads during execution. This presents a problem because actively synchronizing threads and exchanging values between all of them can be a tricky process that is often prone to bottlenecking.

Butterfly reduction has been a widely used algorithm for exchanging primitives between threads within the same warp for many years now [18] [22] [23]. The high level of efficiency provided is because each thread swaps with the thread that equals its own id *xor* an iteratively increasing operand equal to 1, 2, 4, 8, and 16. CUDA provides an intrinsic operation *___shfl_xor_sync* which allows threads to efficiently combine intra-warp held values. It also takes care of any synchronization necessary because threads need to have all reached the same instruction to correctly exchange memory.

Kernels can often exchange data frequently, so even a small performance decrease can significantly alter overall runtime. Butterfly reduction's high versatility and widespread usage make it mission-critical for developers, and any performance loss would be a troubling architectural deficiency.

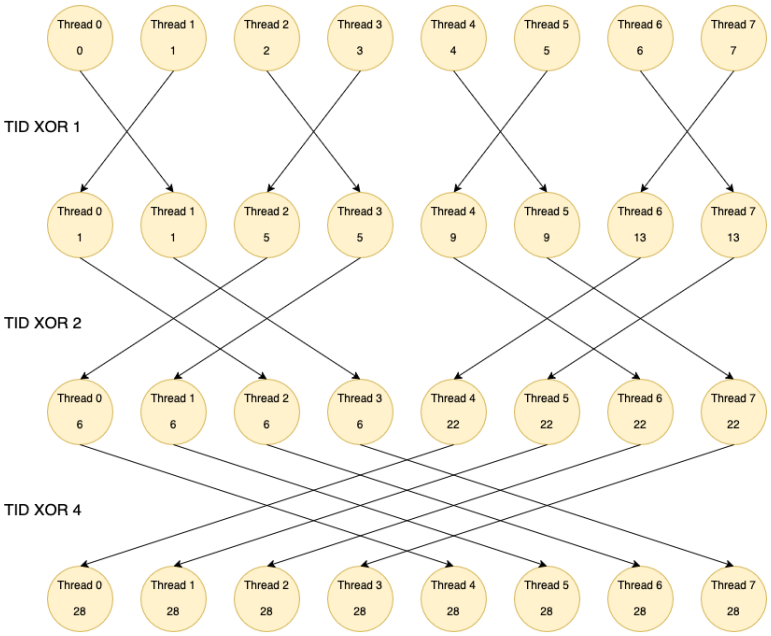


Figure 5.1: Butterfly Reduction Visual

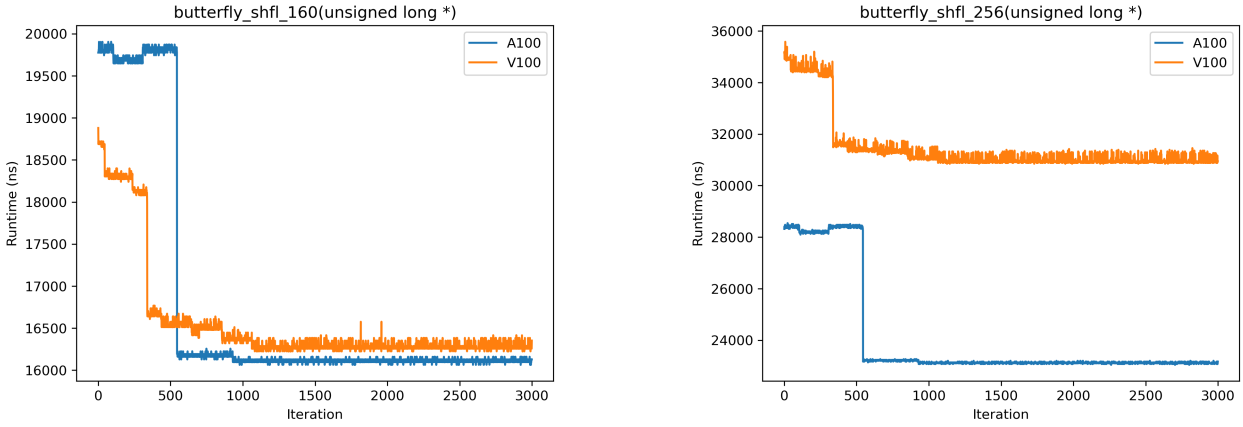
5.2.1 Shuffled Sync Performance

In order to compare the performance of butterfly reduction on the two GPUs, a kernel was written that runs the routine many times in a row to effectively measure its performance due to the low runtime per iteration. For all butterfly reduction benchmarks, each block contains 1024 threads.

Listing 5.1: Snippet of Benchmarked Butterfly Reduction

```
1    unsigned int sum = 0;
2    unsigned int inc = (index + (index / 32)) % 32;
3
4    // Repeated 4 times
5    inc += __shfl_xor_sync(0xFFFFFFFF, inc, 1, 32);
6    inc += __shfl_xor_sync(0xFFFFFFFF, inc, 2, 32);
7    inc += __shfl_xor_sync(0xFFFFFFFF, inc, 4, 32);
8    inc += __shfl_xor_sync(0xFFFFFFFF, inc, 8, 32);
9    inc += __shfl_xor_sync(0xFFFFFFFF, inc, 16, 32);
```

```
10
11      sum += inc;
```



(a) 160 Blocks (b) 256 Blocks

Figure 5.2: Butterfly Reduction Benchmark Partial Results

The diagram exhibits two important characteristics. First, there is clearly a warm-up period required to get from the initial burst of execution to a more predictable steady state behavior. This warm-up behavior could be caused by a number of GPU nuances such as cache warmup, clock speed adjustment, or scheduler changes. Secondly, the steady state behavior exhibited after roughly 500 iterations or roughly ten milliseconds shows that butterfly reduction performance is unimpressive on the newer A100 GPU with an increase in performance of only 0.63% at 160 blocks.

To determine the reproducibility of this pattern across varying load intensity, a number of different grid sizes were tested to see how this pattern changes with varying amounts of GPU load. With few exceptions, the A100 delivers underwhelming performance compared to the V100 for workloads that do not exceed the V100's simultaneous block residency capacity of 160. This is most pronounced in the initial burst of the kernel, but is still present in the steady state. However, for larger workloads, the A100 performs notably better than the

Table 5.1: Butterfly Reduction Mean Performance A100 vs. V100

Grid Size (Blocks)	Initial Mean Runtime Diff. (%)	Steady Mean Runtime Diff. (%)
8	13.19	2.52
16	13.49	2.78
32	12.98	2.88
64	11.57	2.22
96	-38.32	-43.5
128	7.86	-0.63
160	7.37	-1.03
192	-25.39	-31.6
224	5.48	-2.6
256	-18.25	-25.36
288	-18.1	-24.96
512	-21.97	-28.55
1024	-17.55	-24.08

V100. Pivoting to Nsight Compute, a partial architectural explanation can be seen in the SM frequency. The V100's SMs were operating at 1.20 GHz and the A100's at 1.13 GHz. While this 5.8% difference does not completely explain the up to 13% performance gap, it is potentially a contributing factor.

5.2.2 Warp Reduce Primtive

Since the release of the Ampere architecture, a new array of synchronization primitives is available on the A100 or newer GPUs. They allow for warp-wide add, max, min, and, or, and xor operations. Importantly, they can be significantly higher performing than their older counterparts. For example, the multi-step butterfly reduction code that utilizes *__shfl_xor_sync* can be simplified to a single *__reduce_add_sync* call. Two kernels were written to compare their performance. The first is a classic butterfly reduction kernel, and the second is a modern version utilizing the new *__reduce_add_sync*. Importantly, these two versions use 32-bit integers since the newer warp reduce functions do not support

wider integers.

Listing 5.2: Snippet of Benchmarked Butterfly Reduction

```
1    unsigned int sum = 0;
2    unsigned int inc = (index + (index / 32)) % 32;
3
4    // Repeated 4 times
5    unsigned int val = __reduce_add_sync(FULL_MASK, inc);
6    sum += val;
7    inc = val;
```

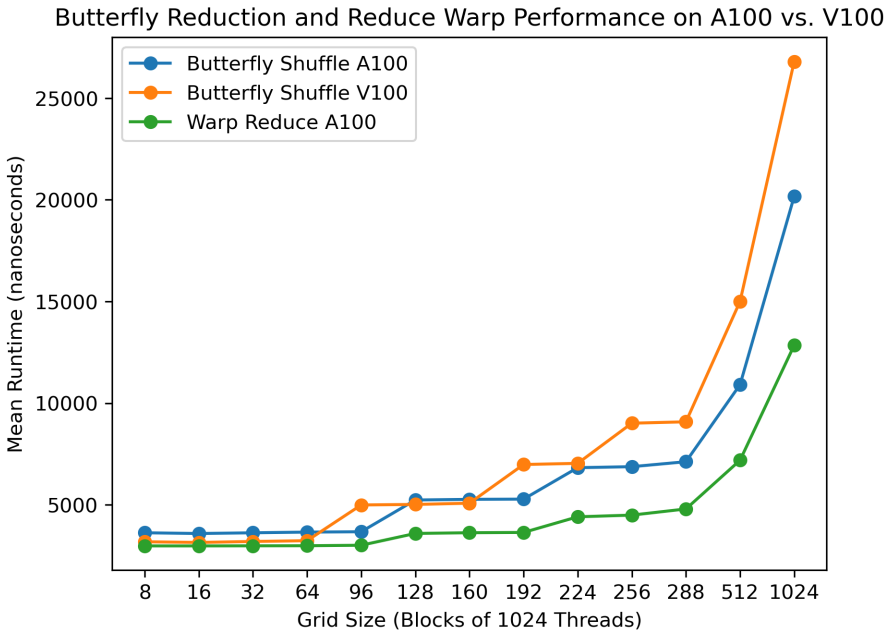


Figure 5.3: Butterfly vs. Reduce Warp Performance

On one hand, reduce functions do not support types other than signed and unsigned 32-bit integers unlike the previous warp primitives. This presents a problem when trying to retrofit older code with newer reduce functions if types like longs, floats, or doubles are needed. On the other hand, the performance is measurably and significantly improved for 32-bit integers. There was also no measured warp-up period to obtain steady-state performance with the

32-bit warp reduce and butterfly shuffle codes.

In summary, butterfly reduction sees performance struggles on the A100 for smaller grid sizes with more severe discrepancies present at the start of execution. Modern warp reduce functions alleviate this issue for 32-bit integers, but other types are left behind. On one hand, developers are left having to use the slower butterfly *__shfl_xor_sync* reduction code for other types, which hampers potential performance gains. On the other, if reduction of 32-bit integers is desired, high performance can be achieved on the A100, but this is still not portable whatsoever to the V100 due to the lack of backward compatibility of warp reduce functions.

5.3 Synchronized Neighbor Shuffle

Exchanging data within a warp doesn't have to use butterfly reduction; in fact, there are a variety of warp-level primitives in the same family as *__shfl_xor_sync* which can power a variety of different reduction algorithms [24] [25] [26]. For instance, the *__shfl_sync* intrinsic provides native functionality for copying data from a single thread, thus making it an incredibly flexible operation for intra-warp data exchange. Given that butterfly reduction utilizing *__shfl_xor_sync* exhibited performance portability problems and the continued importance of efficient intra-warp communication in applications like Sparse Matrix-Vector Multiplication [27] [28], a natural question arises: do other warp-level reduction primitives also have similar performance issues? To explore this, a kernel was developed that implements a simple exchange protocol where each thread updates a variable by swapping the value held in a neighboring thread. The offset from the current thread increases iteratively in a round-robin fashion.

The algorithm was designed to make simple yet effective use of the *__shfl_sync* intrinsic,

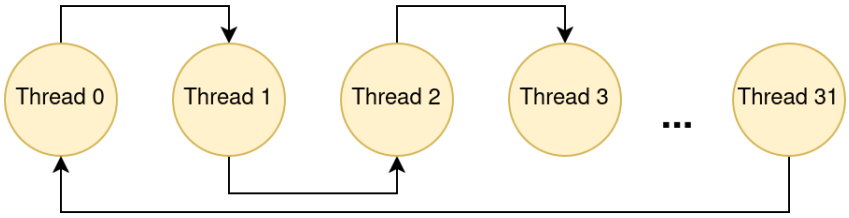


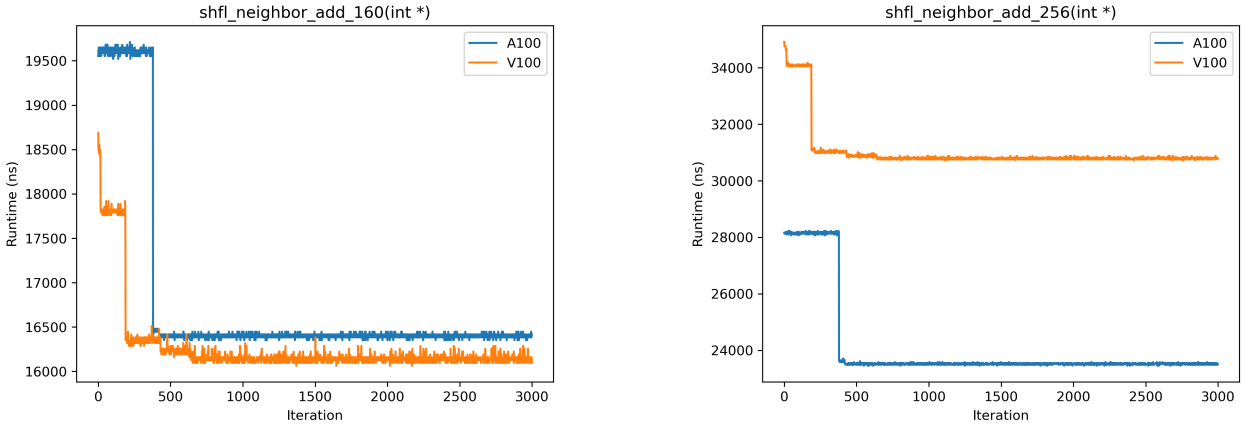
Figure 5.4: Synchronized Neighbor Shuffle

which retrieves the value of the desired variable from the provided thread ID. It will wrap around any out of bounds value, so thread 31 of a warp will read thread 0's variable when lane 32 is requested. Each block contains 32 warps concurrently executing this algorithm.

Listing 5.3: Snippet of Benchmarked Synchronized Neighbor Shuffle

```
1         const int warpLevelThreadId = (int) threadIdx.x & 0x1f;
2         int value = warpLevelThreadId;
3         int sum = 0;
4
5     #pragma unroll
6         for (int i = 0; i < 32; i++)
7         {
8     #pragma unroll
9             for (int j = 0; j < 32; j++)
10                sum += __shfl_sync(0xffffffff, value, warpLevelThreadId + i +
                      j);
11        }
```

Interestingly, the performance characteristics of this algorithm were strikingly similar to the butterfly reduction code. This makes sense since the two intrinsics applied are each part of the same family. A warm-up period is clearly visible that lasted around 8 ms, which preceded a steady-state non-volatile performance zone. Once there, the A100 ran notably slower than the V100 when compared to its performance in the butterfly reduction code.



(a) 160 Blocks

(b) 256 Blocks

Figure 5.5: Neighbor Shuffle Benchmark Partial Results

Table 5.2: Synchronized Neighbor Shuffle Mean Performance A100 vs. V100

Grid Size (Blocks)	Initial Mean Runtime Diff. (%)	Steady Mean Runtime Diff. (%)
8	12.04	3.74
16	13.06	4.72
32	13.08	4.69
64	12.76	4.39
96	-38.99	-43.79
128	8.45	-0.01
160	10.32	1.63
192	-23.7	-29.73
224	7.76	-0.55
256	-17.11	-23.60
288	-15.99	-22.49
512	-20.78	-26.82
1024	-16.26	-22.47

As for the overall pattern when comparing different launch grid sizes, the same trend from the butterfly reduction code is visible here as well. Smaller grid sizes exhibited worse performance portability, and larger grid sizes showed the opposite. Butterfly reduction code performance can be improved when using 32-bit integers through newer warp reduce functions, but intrinsics like *___shfl_sync* do not have a modern analog to improve performance portability. Regardless, intra-warp reductions exhibited degraded performance on the A100. Given the importance of reductions in many kernels, this is a principal performance portability challenge for the architecture.

5.4 Bank Conflicts

Bank conflicts are a common nuisance that hampers the performance of kernels that make use of shared memory, so their performance portability is examined in this section. Shared memory is often utilized in kernels due to its quick access times on par with L1 cache. However, it's important for programmers to understand the details of the implementation behind this programmable cache in order to obtain maximum performance. All shared memory is split into thirty-two banks that are each thirty-two bits wide. Older cards which used warp sizes of 16 threads had 16 banks. For example, if an array of sixty-four 32-bit integers is stored in shared memory, index 0 will be in bank 0, index 31 in bank 31, index 32 in bank 0, and so on. Shared memory is fast, but it's important to understand that each bank can only serve one thread at a time. Suppose thirty-two threads of a warp wish to access a value in the shared memory array. If each thread accesses the array index equal to their thread ID, then each thread's load will be spread out evenly across each bank. If there are only thirty-two threads in that block accessing the array, threads may not have to queue at all to read the data.

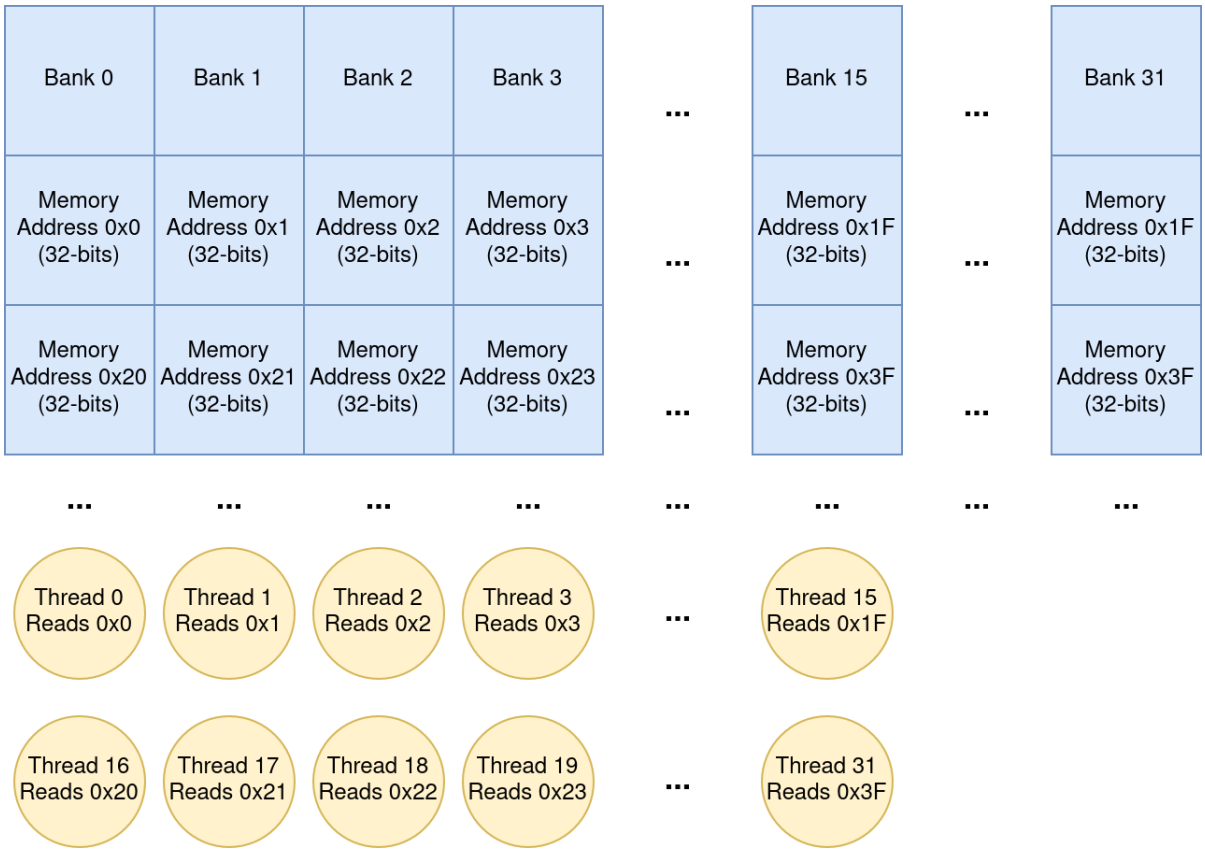


Figure 5.6: A 2-way Bank Conflict

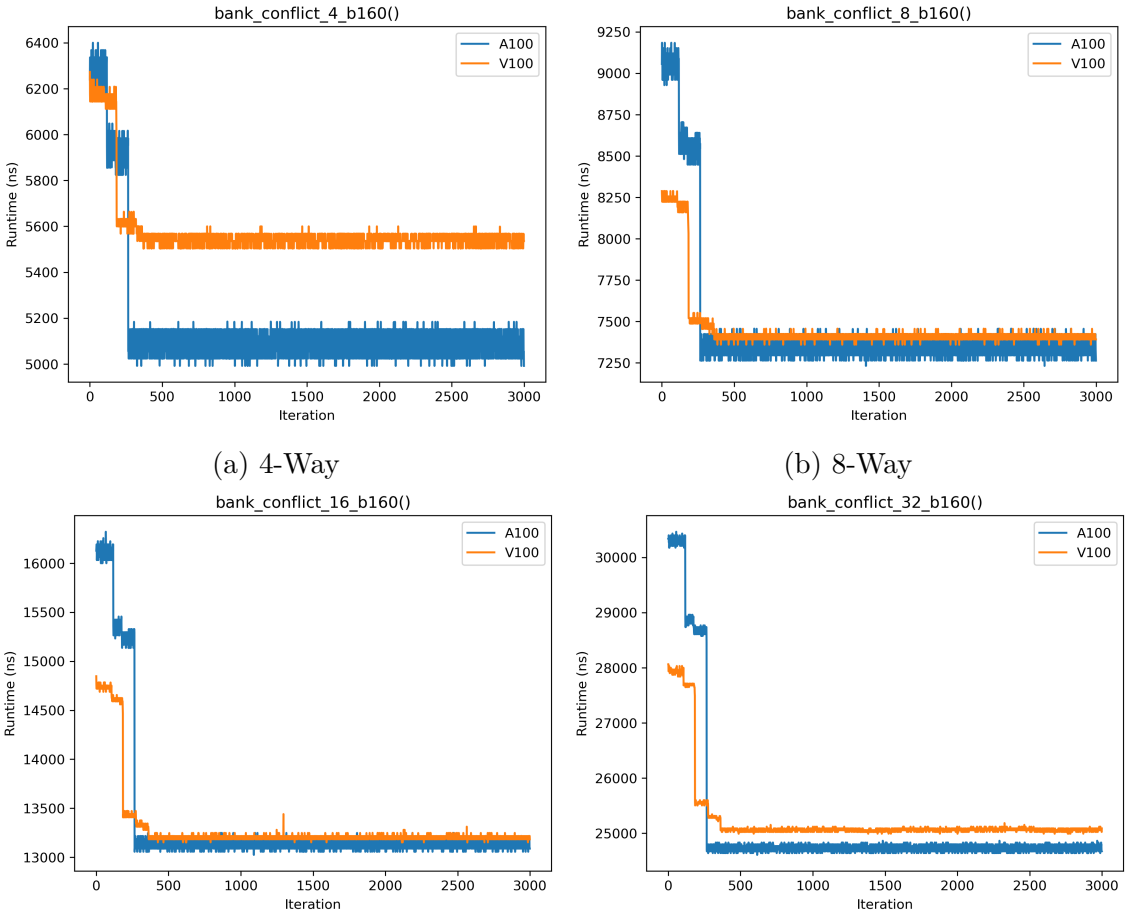
Unfortunately, it can be easy to write code that results in n-way bank conflicts. An n-way bank conflict means that there are n threads concurrently accessing the same bank. In the diagram, two threads per warp are accessing the first sixteen banks concurrently, so the second thread to make the read request has to wait for the first thread to complete its operation. This represents a 2-way bank conflict, but these can become as severe as 32-way where every thread in the warp is accessing memory within the same bank.

In order to accurately test the performance of the A100 and V100 under bank conflict scenarios, a simulation kernel was written and benchmarked. Each thread writes to shared memory and calculates which index to write to that will produce the desired n-way bank conflict. Each block here contains only 256 threads due to shared memory allocation constraints on

both GPUs.

Listing 5.4: Snippet of Benchmarked Bank Conflict Simulation Code

```
1     constexpr int BANKWAY = 32; // n-way bank conflict of 32
2     __shared__ int cache[BANK_THREADS * BANKWAY];
3     constexpr int NUM_COLS = (BANKS / BANKWAY);
4
5     const int tid = threadIdx.x;
6     const int btid = threadIdx.x;
7
8 #pragma unroll 32
9     for (int i = 0; i < 32; i++) {
10        const int row = (btid / NUM_COLS) % BANKWAY;
11        const int col = (btid % NUM_COLS) + ((((btid + (32 * i)) %
              BANK_THREADS) / 32) * (NUM_COLS));
12        const int pos = row * BANK_THREADS + col;
13
14        cache[pos] += tid;
15    }
```



(a) 4-Way

(b) 8-Way

(c) 16-Way

(d) 32-Way

Figure 5.7: Bank conflicts of Varying Severity With 160 Blocks

Table 5.3: 4, 8, 16, 32-way Bank Conflicts A100 vs. V100

(a) 4-Way

Grid Size	Initial Diff. (%)	Steady Diff. (%)
8	-36.39	-41.53
16	-35.79	-40.96
32	-34.56	-39.86
64	-32.89	-38.24
96	-34.59	-39.85
128	0.52	-7.62
160	0.11	-8.14
256	-13.61	-20.81

(b) 8-Way

Grid Size	Initial Diff. (%)	Steady Diff. (%)
8	-13.26	-20.29
16	-11.51	-18.75
32	-11.46	-18.65
64	-9.9	-17.06
96	-33.54	-38.86
128	9.82	0.94
160	8.4	-0.79
256	-15.55	-22.48

(c) 16-Way

Grid Size	Initial Diff. (%)	Steady Diff. (%)
8	10.21	1.07
16	9.88	0.94
32	9.52	0.72
64	8.64	-0.16
96	-39.28	-44.13
128	8.71	0.15
160	8.25	-0.47
256	-17.65	-24.25

(d) 32-Way

Grid Size	Initial Diff. (%)	Steady Diff. (%)
8	8.68	-0.25
16	8.87	0.07
32	8.74	-0.03
64	8.16	0.55
96	-42.94	-47.51
128	7.54	-1.03
160	7.35	-1.31
256	-21.12	-27.86

Like with the reduction kernels, a warm-up and steady state feature is present in these performance numbers too. Bank conflicts exhibited the worst performance portability with grid sizes below the V100's simultaneous block residency capacity of 160, with less severe bank conflicts producing better scaling. Additionally, bank conflicts exhibited poorer performance portability as they became more severe. Because larger conflicts result in more serialization, Ampere is discernibly struggling to keep up with Volta when serializing these shared memory accesses. Nsight Compute supports this theory as the A100 stalls for a total of 2415 cycles versus the V100's 1141 cycles when hitting 32-way bank conflicts with 160 blocks. Therefore, developers should optimize bank conflicts out of their code or transition to another algorithm without them to maximize performance portability.

5.5 Prefix Sum

This section investigates the performance portability of computing a prefix sum on the V100 and A100. It's a widely used concept in GPU programming that plays a critical role in a number of use cases such as statistical analysis and radix sort [29], so poor performance portability would have wide-ranging effects. Thankfully, the concept behind a prefix sum is fairly straightforward to grasp. Given an array of numbers n and an index i, the prefix sum p_i can be calculated as follows:

$$\sum_{j=0}^{i-1} n_j = p_i$$

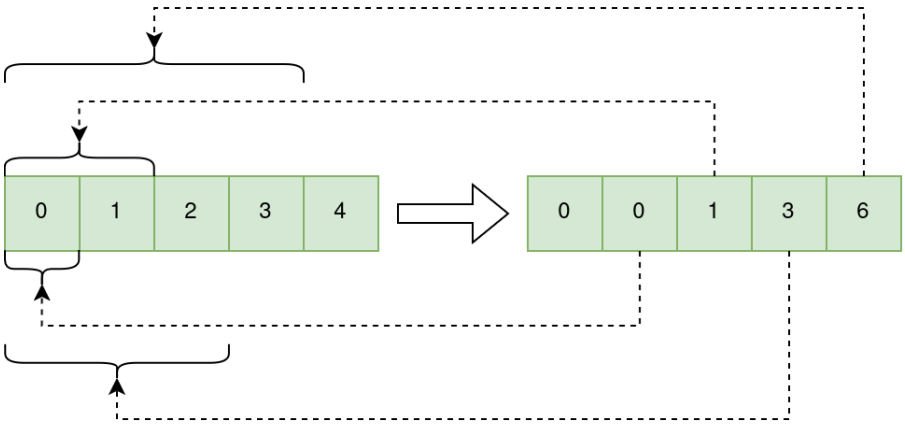


Figure 5.8: Elementary Exclusive Prefix Sum

Notably, there are two types of prefix sums. A prefix sum p_i can be represented exclusively as $\sum_{j=0}^{i-1} n_j = p_i$ or inclusively as $\sum_{j=0}^{i} n_j = p_i$. More plainly, inclusive scans include the value at n_i whereas exclusive ones do not.

Optimizing this for a single threaded environment is trivial. The optimized algorithm walks the array and determines each p_i by keeping track of a cumulative sum of the traversed elements. However, in the context of a massively parallel environment like that of a GPU,

this non-parallel algorithm becomes useless. To combat this issue, Blelloch designed a scan algorithm that is highly efficient on GPUs, which can be used to determine an array's prefix sum [30]. The algorithm starts with creating partial sums in strides that begin at 2 and double after each step. For example, if a thread is creating a partial sum at index i, the other addend will be located at $i - \frac{s}{2}$ where s is the stride. This is done until s exceeds the length of the array, l, when the down-sweep phase begins. In this final phase, the addition that took place in the up-sweep phase is continued except s begins at $\frac{l}{2}$ and is steadily decreased by the same factor of 2. Additionally, the value at index i replaces the value at index $i - \frac{s}{2}$ after the sum is computed.

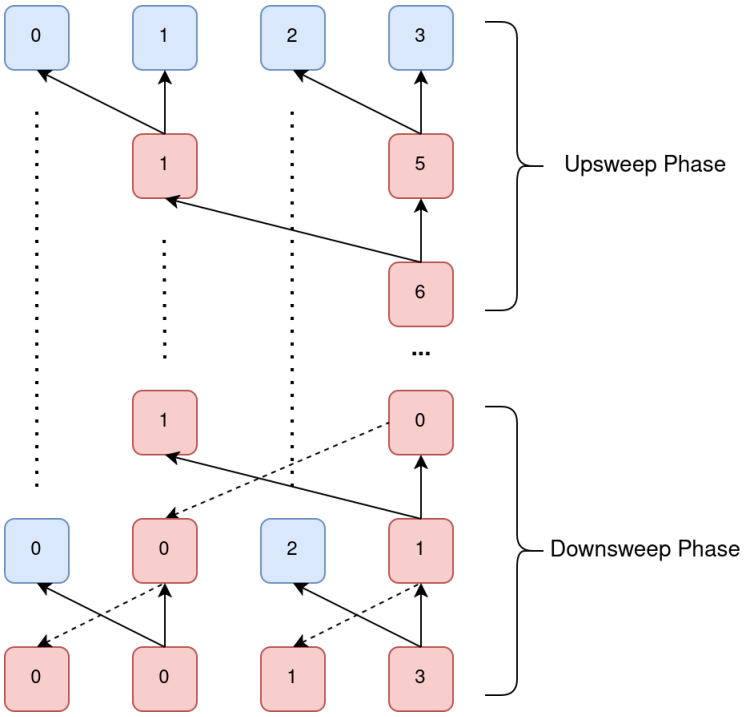


Figure 5.9: Blelloch Scan for Exclusive Prefix Sum

5.5.1 Bank Conflict Kernel

In order to assess the performance of computing a Blelloch-based prefix sum in CUDA, a kernel was developed and benchmarked. Every block calculates its own partial sum on 1024 elements using 1024 threads. A shared memory cache array was used to consecutively store all partial sums, but it is accessed in a way that generates, on average, 6-way bank conflicts for both reads and writes.

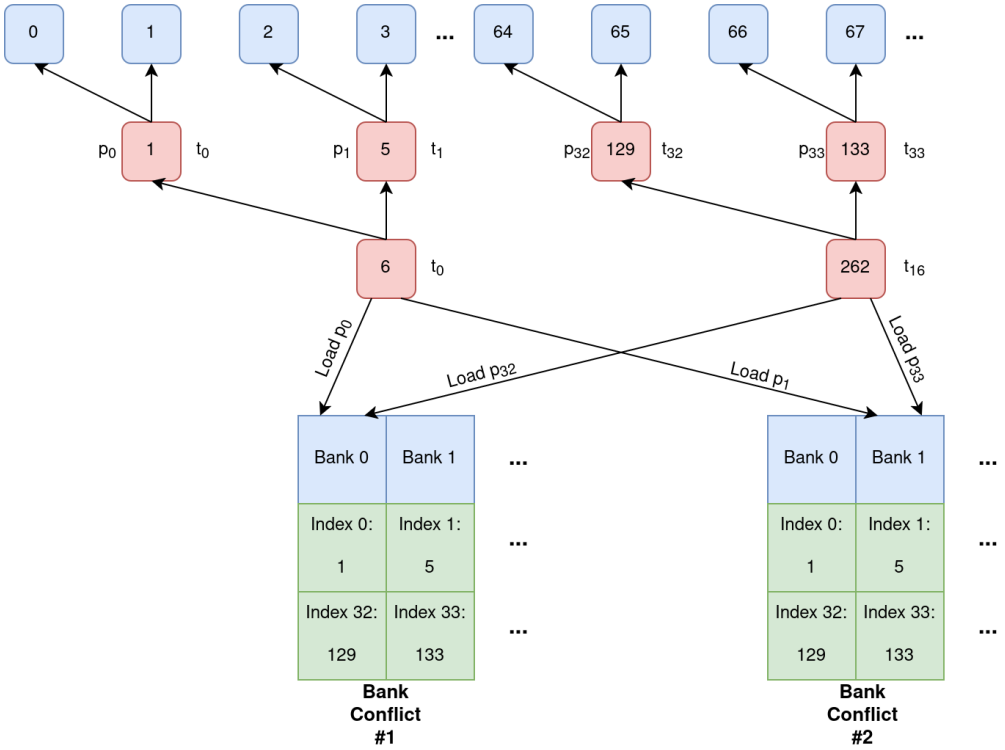
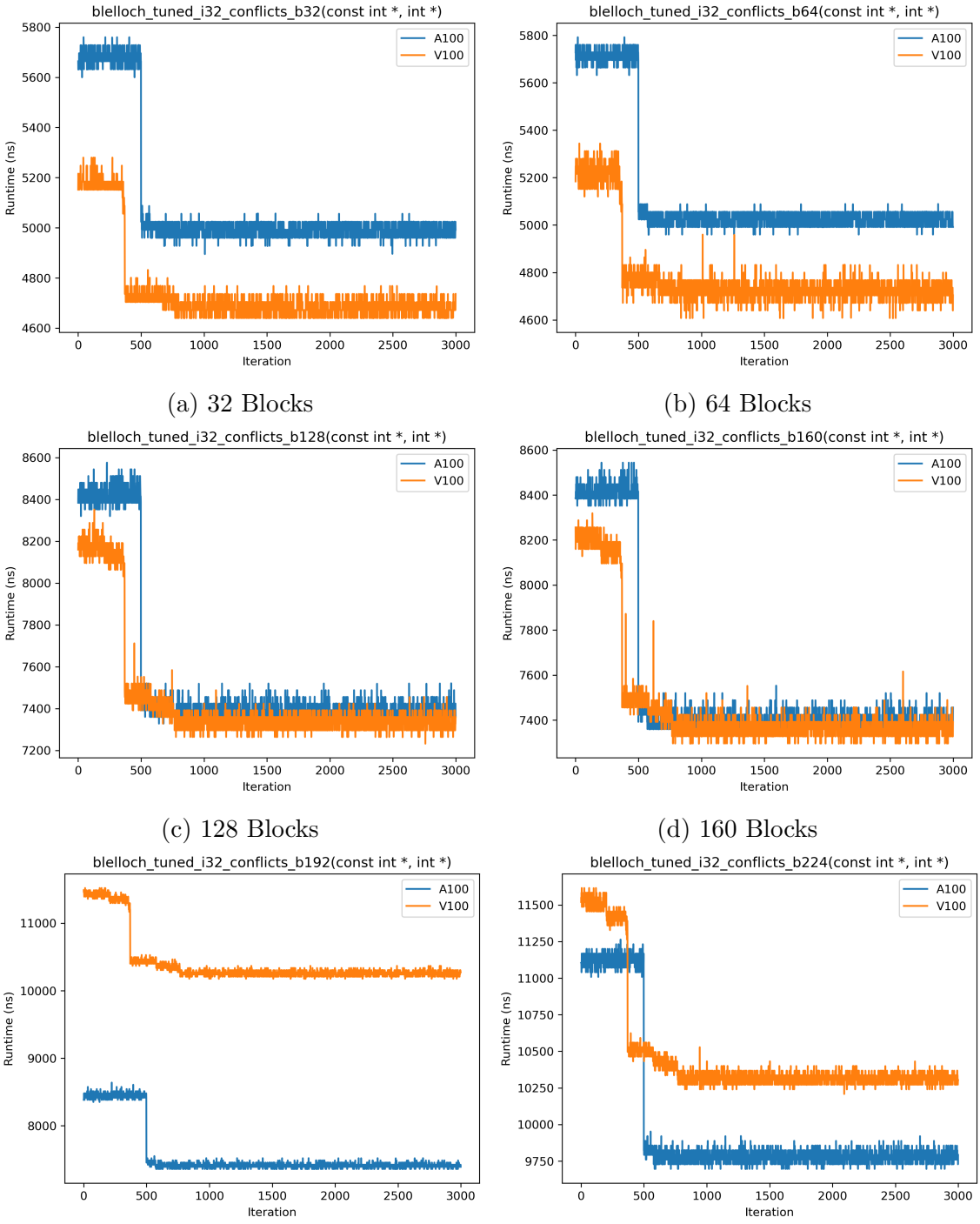


Figure 5.10: Bank Conflict During Upsweep Phase

These bank conflicts occur during the upsweep phase when previously computed partial sums are accessed by threads with iteratively lower thread IDs. Eventually, threads within the same warp access partial sums that were previously computed in different warps. This causes one bank conflict when loading the partial sum and when writing the updated value. The downsweep phase has a similar problem as it exhibits the same access patterns.



(a) 32 Blocks

(b) 64 Blocks

(c) 128 Blocks

(d) 160 Blocks

(e) 192 Blocks

(f) 224 Blocks

Figure 5.11: Blelloch Scan Prefixsum With Varying Block Counts

Table 5.4: Mean Runtime of Exclusive Prefix Sum with Bank Conflicts on A100 vs. V100

Grid Size (Blocks)	Initial Mean Runtime Diff. (%)	Steady Mean Runtime Diff. (%)
8	9.58	7.0
16	9.78	6.78
32	9.54	6.68
64	9.39	6.37
96	-29.21	-30.85
128	2.83	0.75
160	2.45	0.39
192	-26.12	-27.73
224	-3.47	-5.18
256	-22.1	-23.98
288	-20.09	-21.76
512	-25.62	-27.2
1024	-23.01	-24.34

The kernel exhibits an initial burst of slower performance akin to other codes presented like butterfly reduction. The A100's runtime is notably higher than the V100 by around 10% for most block counts less than the V100's simultaneous block residency capacity of 160. After a few milliseconds, the performance changes to a more predictable steady state. In this steady state, the V100 takes a near 7% lead over the A100 with some of the smaller block sizes like thiry-two and sixty-four. While there were fewer performance issues present with the A100 in the steady state of the simulated bank conflict code, these benchmarks indicate that at least some applications that generate bank conflicts will exhibit degraded performance portability.

5.5.2 Bank-Conflict-Free Kernel

Because of the performance portability issues discovered with the bank-conflict-generating kernel, another Blelloch-based prefix sum code, which was inspired by a similar approach on much older hardware [31], was developed to assess if a more optimized, bank-conflict-free

version would improve performance portability.

Shared memory's high performance means that it's the optimal data store for handling all of the partial sums generated during a Blelloch scan, so it is still needed. However, in order to make full use of its high performance, the bank conflicts needed to be removed. To do this, padding was added every 32 indices in the partial sum cache. A simple function models both the index j and bank b of the shared memory cache where some partial sum p_i, with i meaning the ith partial sum generated in the first round, will be stored.

$$j = \lfloor \frac{i}{32} \rfloor + i$$

$$b = i \bmod 32$$

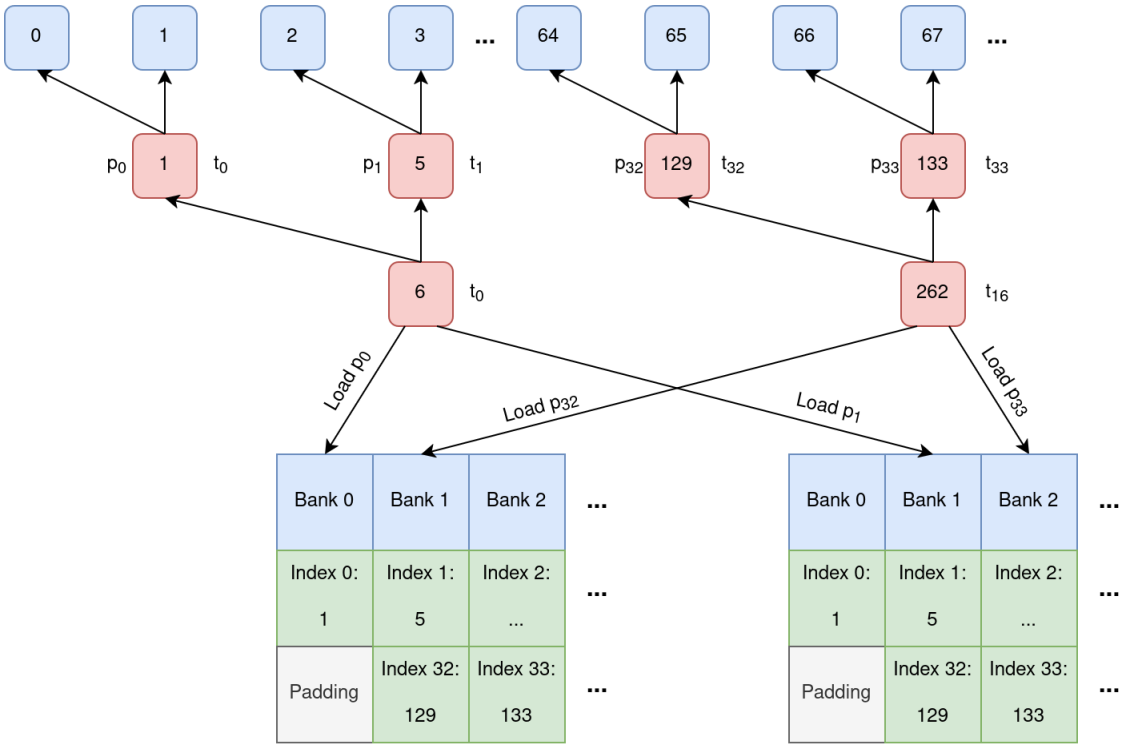
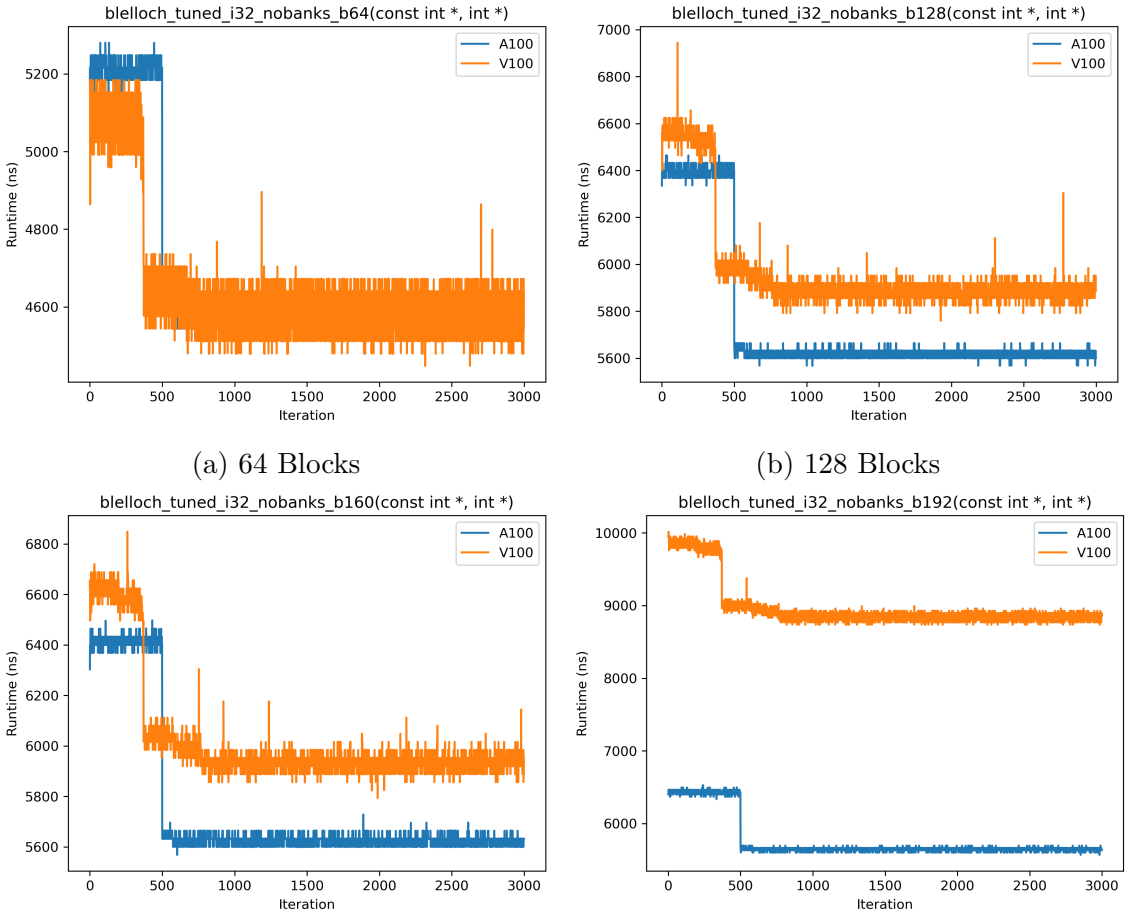


Figure 5.12: Parallel Shared Memory Access During Upsweep Phase

Unlike in the bank-conflict-generating version where the A100 sometimes lagged behind the

V100, they are either functionally tied or the A100 takes the lead. Removing bank conflicts for Blelloch-powered prefix sum noticeably increased performance portability, with the V100 only pulling ahead by no more than 2.22% compared to 9.78% with the bank conflict version. Since bank conflicts do not exhibit good performance portability and this version does not produce bank conflicts, this is an expected result.



(a) 64 Blocks

(b) 128 Blocks

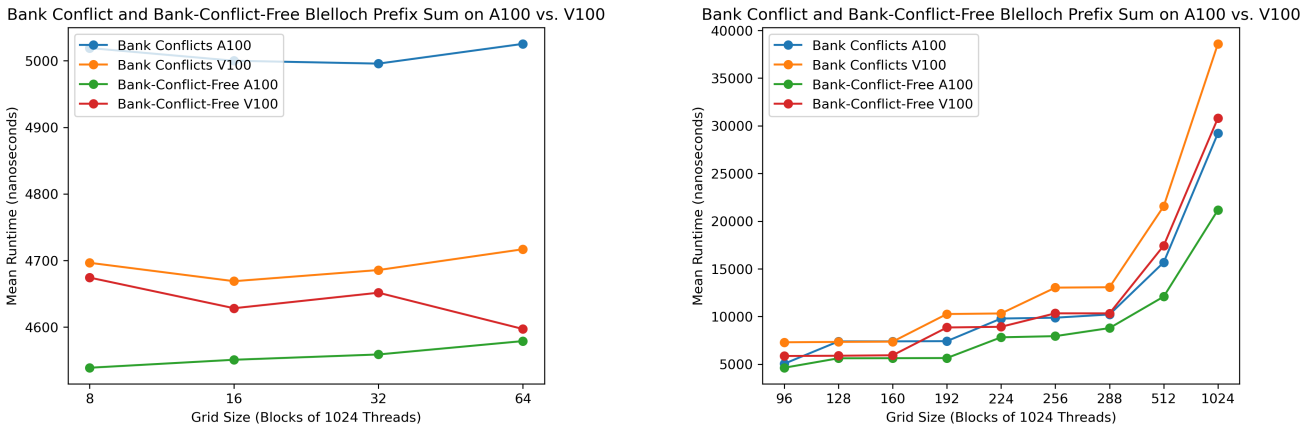
(c) 160 Blocks

(d) 192 Blocks

Figure 5.13: Blelloch Scan Bank-Conflict-Free Prefixsum With Varying Block Counts

Table 5.5: Exclusive Prefix Sum Bank Conflict Version: Mean Performance of A100 vs. V100

Grid Size (Blocks)	Initial Mean Runtime Diff. (%)	Steady Mean Runtime Diff. (%)
8	-0.45	-2.93
16	0.82	-1.81
32	0.47	-1.9
64	2.22	-0.53
96	-19.13	-21.1
128	-2.51	-4.51
160	-3.16	-5.14
192	-34.79	-36.21
224	-10.63	-12.44
256	-21.37	-23.09
288	-12.74	-14.66
512	-29.11	-30.7
1024	-30.0	-31.31



(a) Smaller Grid Sizes

(b) Larger Grid Sizes

Figure 5.14: Prefix Sums Overall Performance Comparison

Aside from performance portability improving, the bank-conflict-free version also decreased mean runtime across the board. Because threads are not waiting for one another when accessing shared memory, the transaction can execute rapidly, which reduces overall runtime. Given everything discussed, it is important for developers to optimize code that produces bank conflicts to improve both performance and performance portability.

5.6 Column Permutation

The performance portability of generating a permutation of a column-major matrix is examined in this section, as this is an operation that challenges some typical programming practices like coalescing global memory accesses. The Reusable Accelerated Functions and Tools for Vector Search and More (RAFT) library provides C++ and Python programmers with CUDA-accelerated machine learning and data mining algorithms. While there are many available functions, *make_regression* was specifically analyzed. It's built for generating random regression datasets of varying dimensions. One of its helper methods used in the process of generating the dataset is a CUDA kernel named *permuteKernel*. Its job is to either shuffle around rows or columns of a matrix depending on whether it's row-major or column-major based. The performance of permuting the columns within a column-major context was analyzed.

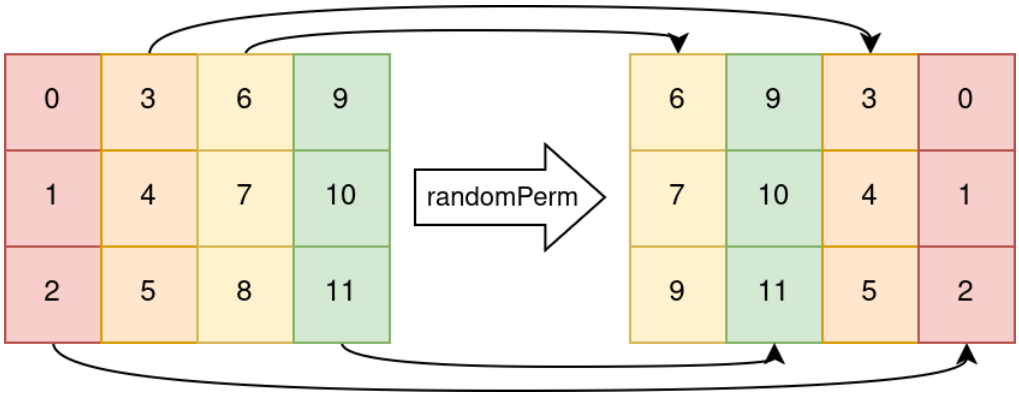
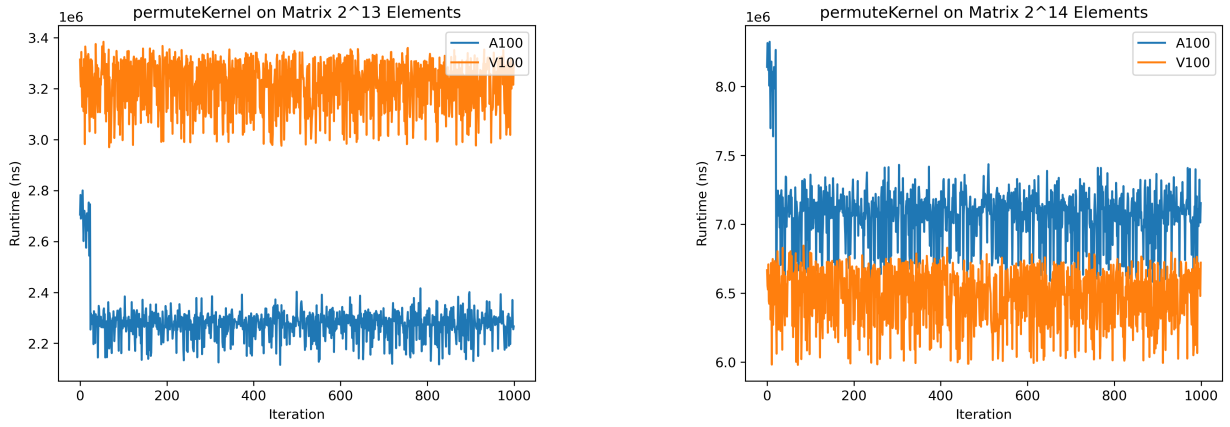


Figure 5.15: Column Permutation

For all regression problem sizes requested, there was no visible warm-up period on the V100, yet this feature was present on the A100. This is an interesting change because both GPUs thus far usually exhibited some kind of initial state where performance was worse for some segment at the start of execution, but now it is only the A100 showing such behavior.



(a) 2^{13} Elements (b) 2^{14} Elements

Figure 5.16: Partial permuteKernel Benchmark Results of A100 vs. V100

Table 5.6: Mean Performance of *permuteKernel* on A100 vs. V100

Elements	Mean Runtime Diff. (%)	Launched Grid	A100 Mem. Throughput (%)
2^4	-6.84	«2, 256»	0.04
2^8	-11.07	«2, 256»	0.17
2^{10}	-5.74	«2, 256»	0.17
2^{12}	-43.71	«2, 256»	0.17
2^{13}	-28.87	«2, 256»	0.17
2^{14}	9.25	«2, 256»	0.21
2^{15}	18.14	«2, 256»	0.25
2^{16}	19.07	«2, 256»	0.26
2^{17}	18.43	«2, 256»	0.27

RAFT's *permuteKernel* clearly has poor performance portability from the V100 to A100. Runtime on the newer GPU is higher by more than 19% at times. Interestingly, this is the first investigated example where performance portability worsened as the problem size grew. Typically, smaller problem sizes exhibited weaker performance portability, whereas larger ones showed the opposite. However, this kernel differs in that its launch grid size remains unchanged as the problem size increases by a factor of *8192* from 2^4 to 2^{17}. By only using a small grid size for large problem sizes, performance scaling is limited since more threads cannot help to offset the increased work size. Therefore, performance portability is likely to

increase with better grid scaling to match the problem size, but this theory requires further investigation and relies on the suboptimal memory access patterns not being too detrimental to performance.

5.7 L2 Normalized k-Nearest Neighbors

The k-nearest neighbors (kNN) algorithm is fundamental in machine learning, so it is essential for it to exhibit strong performance portability. It works by finding the k-closest points to a location in some n-dimensional space. A simple use case is teaching a computer how to recognize a drawn letter by figuring out which other reference letter drawings it is closest to.

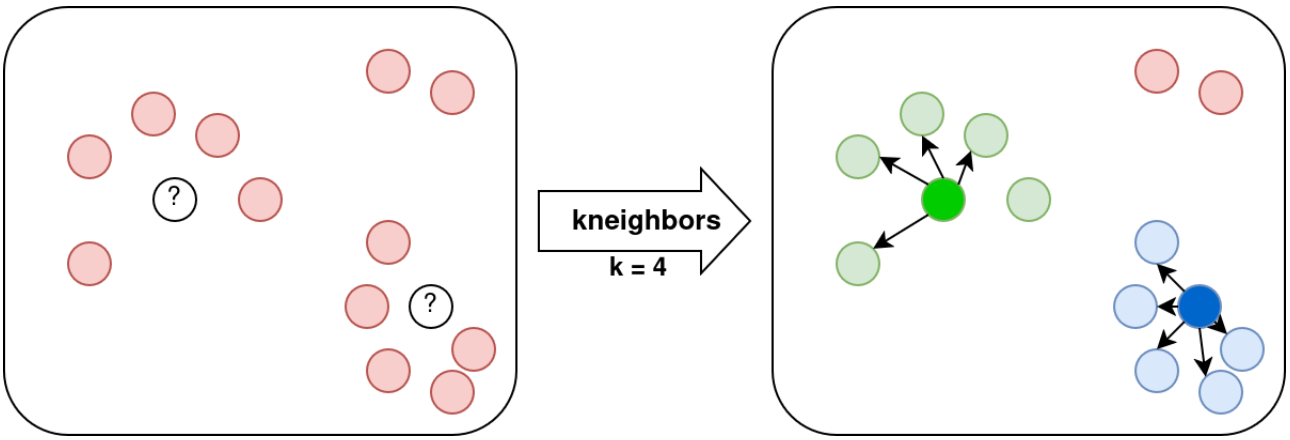
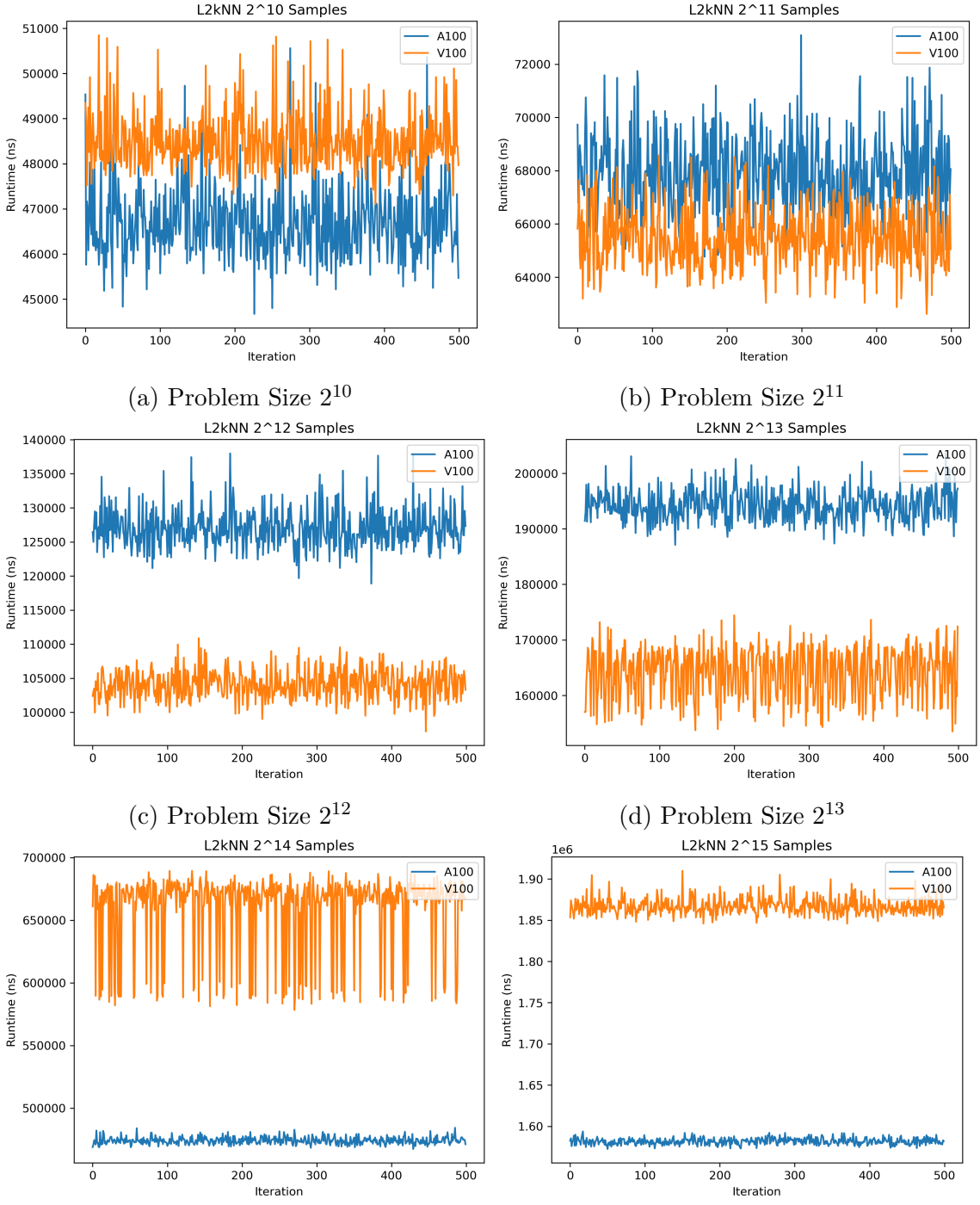


Figure 5.17: Sample kNN Execution with $k = 4$

The CUML library is both popular and mature, with strong support for computing the k-nearest neighbors of data sets, so its implementation was benchmarked. Data was first generated with the *make_blobs* function. For each problem set size, CUML's *kneighbors* function was executed 500 times. To ensure the results were not representative of only a single random state, the data points were regenerated for each run with the random seed being set to the increasing iteration value. The 4-nearest neighbors were identified, with Euclidean distance being used for comparisons.



(a) Problem Size 2^{10}

(b) Problem Size 2^{11}

(c) Problem Size 2^{12}

(d) Problem Size 2^{13}

(e) Problem Size 2^{14}

(f) Problem Size 2^{15}

Figure 5.18: Performance of CUML's *fusedL2kNN* with Varying Problem Set Size

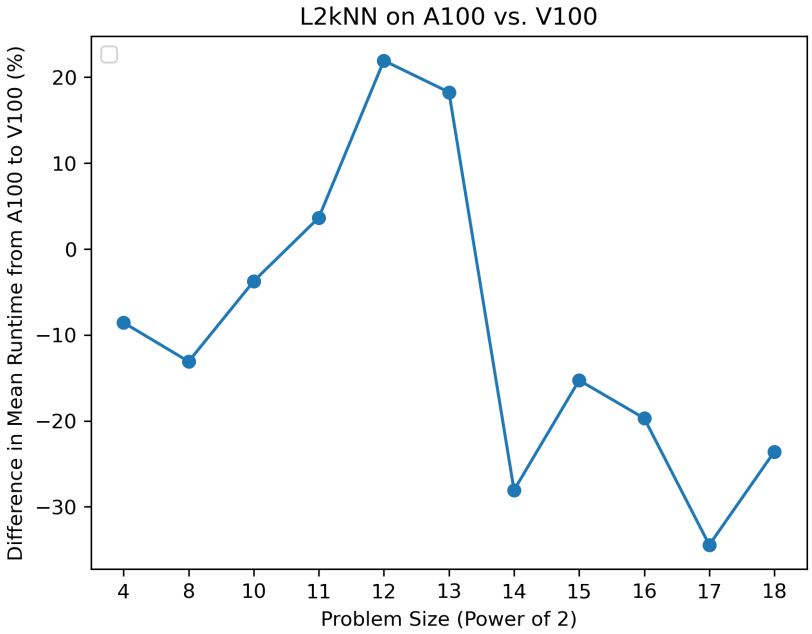


Figure 5.19: L2kNN Mean Performance

Unlike what was typically exhibited in most of the other presented examples like butterfly reduction or prefix sum, there is no visible warm-up period in any of the L2kNN runs. Oddly, it is only for a specific middle-road range of problem sizes where performance portability is poor. For smaller and larger numbers of elements, performance portability is much higher with mean runtimes on the A100 improving over the V100 by as much as roughly 22%.

5.8 Atomic Operations

When multiple threads need to write to a memory location concurrently, atomic operations save the day with their high performance and robust protection for correctness. This section examines the performance portability of atomic operations at both the application and intrinsic levels, as they are invaluable for inter-warp reductions where warp-level primitives like *__shfl_xor_sync* cannot be used. Given their importance, wide-ranging performance

portability issues could have deep effects on high-level reductions.

5.8.1 turboSETI

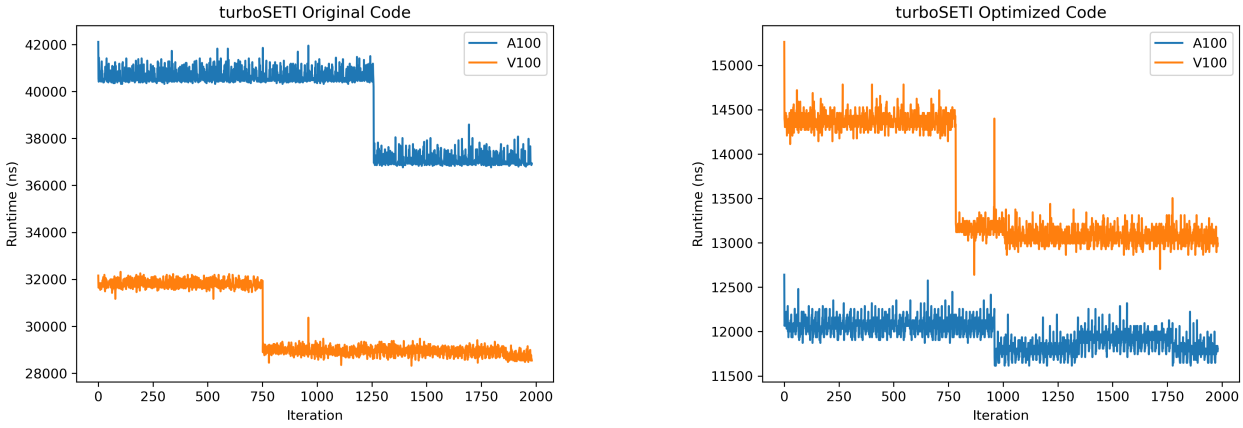
TurboSETI is a CUDA-accelerated Python application developed by the SETI Research Center at the University of California, Berkeley. The application is tailored toward finding narrow band drifting signals in background filterbank data. These wave patterns mimic communication on Earth, so the hope is that extensive surveying of signals from outer space may reveal communications from intelligent extraterrestrial life. The kernel *hitsearch_float32* is used within the library to efficiently search for these patterns by utilizing the strong parallel performance of NVIDIA GPUs.

Listing 5.5: Snippet of turboSETI hitsearch_float32 Kernel [32]

```
1     int index = blockIdx.x * blockDim.x + threadIdx.x;
2     int stride = blockDim.x * gridDim.x;
3     int count = 0;
4     for (int i = index; i < n; i += stride) {
5         const double bin = (spectrum[i] - median) / stddev;
6         if (bin > threshold) {
7             count++;
8             if (bin > maxsnr[i]) {
9                 maxsnr[i] = bin;
10                maxdrift[i] = drift_rate;
11            }
12        }
13    }
14    atomicAdd(&tot_hits[0], count);
```

From a purely static perspective, this code's central performance scalability issue is the

atomic add operation at the end. Every thread in the launched grid will be adding to the same global memory location located at *tot_hits*. This causes significant contention for access to the memory location. Even if the load was optimized to be split between two different locations, the amount of contention would be reduced by half. This is clearly an area where optimization would almost certainly increase performance. To determine this code's performance portability, it was benchmarked on both the V100 and A100. The background filterbank data used for the benchmark was collected by the West Virginian Green Bank Telescope in 2020 [33].



(a) Original Code (b) Optimized Code

Figure 5.20: turboSETI Runtime Performance

Table 5.7: hitsearch_float32 Mean Performance on A100 vs. V100

Version	V100 (95% Conf.)	A100 (95% Conf.)	Mean Runtime Diff. (%)
Original	30022 ± 2821	39331 ± 3477	31.01
Optimized	13600 ± 1274	11951 ± 298	-12.13

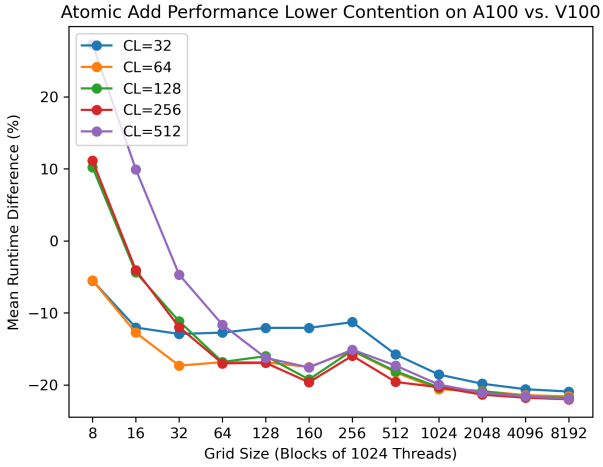
To optimize the original code, the contention was significantly reduced. All threads still need to have their result added to the original singular memory location as before, so a new reduction algorithm was developed. Firstly, each warp performs a butterfly reduction so that the sum of all threads within that warp is held by each thread. Secondly, the first thread in

each warp writes its partial sum into a unique index of a shared memory array. Thirdly, the first thread of each block computes the total of all partial sums held within its block's shared memory array. Lastly, each of these threads from the previous step performs an atomic add of its block's partial sum on the global memory location. Since the kernel is launched with a grid consisting of 2048 blocks with 1024 threads each, this algorithm reduces contention by a factor of 1024.

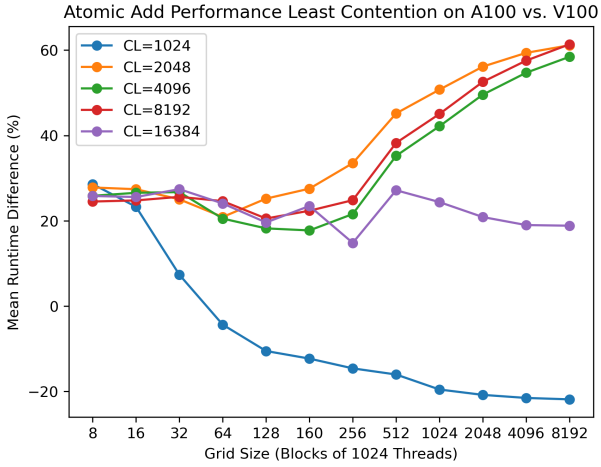
From an optimization standpoint, performance on the GPUs is increased by roughly 55-70%. As for performance portability, the A100 goes from performing 31% worse to 12% better than the V100. While high levels of contention on an atomic operation are clearly not optimal, it is also demonstrably less performance portable.

5.8.2 Atomic Add

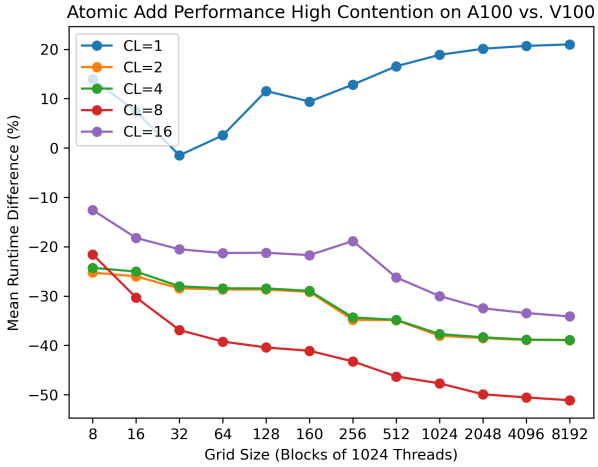
Due to the lower performance portability exhibited by the atomic add operation in the highly contentious turboSETI environment and the importance of the operation such as in reductions [34] and histogram generation [35], benchmarks were written to assess its performance under varying levels of contention. Threads in the benchmark all write to an array of global memory at the index equal to their thread ID modulo the contention level. For example, if the contention level is 1, all threads write to index 0 of the array, or if the contention level is 2, even threads will write to index 0 and odd ones to index 1. The higher the contention level, the lower the amount of contention there is since the contention level is directly related to the number of available memory locations. Each contention level and grid size combination was tested with 10,000 iterations, and there are 1024 threads per block.



(a) Lower Contention



(b) Lowest Contention



(c) Highest Contention

Figure 5.21: Atomic Add Benchmark Results

Starting with the most contention at level one, performance is similar to that of turboSETI with a $0 - 20\%$ performance drop on the A100 over the V100, depending on the grid size. Under this high contention scenario, performance portability generally decreases as the grid size increases, with the V100 and A100 being functionally tied at a grid size of thirty-two blocks. As the contention is decreased, the A100 decisively pulls ahead with as much as a 50% performance uplift. Strangely, this pattern is suddenly broken for the higher contention levels of 1024, 2048, 4096, 8192, and 16384. The contention is theoretically 4096 times lower from

CL=2 to CL=8192, yet the A100 is 60% slower than the V100. The Nsight Compute results were examined for some clues, and some of the reviewed metrics differentiated significantly between the two GPUs. A summary of some discrepancies found in Nsight Compute is presented below.

Table 5.8: Selected Atomic Add Nsight Compute Statistics (CL=8192, 8192 Blocks)

Metric	V100	A100
Compute (SM) Throughput (%)	21.91	10.45
Memory Throughput (%)	44.17	22.60
L1/TEX Cache Throughput (%)	66.63	17.06
Average Stall Drain Warp States (cycles)	15.53	57.69
L2 Hit Rate (%)	99.90	99.93

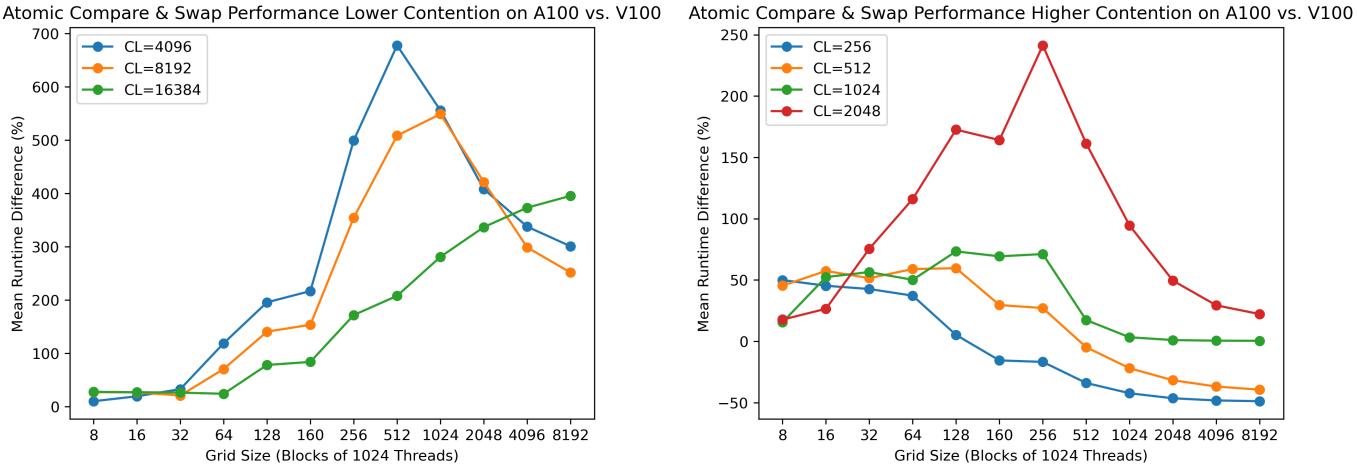
Runtime performance statistics are worse on the A100 with most being at least half as strong compared to the V100. The most significant difference is the more than 3-fold increase in the number of pending drainage stall cycles. While turboSETI gave a glimpse into the performance portability challenges of high-contention atomic adds on the A100, there are also similar struggles with lower contention environments as well.

5.8.3 Atomic Compare & Swap

Due to the troubling performance portability of CUDA's *atomicAdd*, the *atomicCas* intrinsic was also benchmarked to see if atomics as a whole may exhibit poor performance portability. The operation is necessary for preserving synchronization when memory is updated by different threads concurrently, such as in Joint Mutual Information kernels [36]. The same concept of a contention level was used here, with threads determining which address to perform a compare-and-swap operation at based on their thread ID modulo the contention level. All combinations were tested 10,000 times with 1024 threads per block.

Listing 5.6: Snippet of *atomicCas* Benchmark Kernel

```
1    const int tid = blockIdx.x * blockDim.x + threadIdx.x;
2    int read_val, current_val;
3    do {
4        read_val = nums[tid % CONTENTION_LEVEL];
5        current_val = atomicCAS(&nums[tid % CONTENTION_LEVEL], read_val, tid);
6    } while(read_val != current_val);
```



(a) Lower Contention

(b) Higher Contention

Figure 5.22: Atomic Compare & Swap Benchmark Results

Just like with the atomic add results, decreasing contention results in low performance portability, but it is significantly worse with up to a near *700%* difference in mean runtime at a contention level of 4096 with 512 blocks. However, performance portability does generally increase across the board as the grid size increases, but the difference in performance with the lower contention scenarios is still extreme.

Nsight Compute supports this with significantly lower performance metrics. It is evident that the atomic compare and swap operation suffers from significant to severe performance portability problems. The large discrepancy in average long scoreboard stalls indicates dif-

Table 5.9: Selected Atomic Add Nsight Compute Statistics (CL=4096, 512 Blocks)

Metric	V100	A100
Compute (SM) Throughput (%)	11.39	1.23
Memory Throughput (%)	38.60	6.81
L1/TEX Cache Throughput (%)	56.08	3.34
Average Stall Long Scoreboard Warp States (cycles)	66.05	1793.00
L2 Hit Rate (%)	99.80	88.38

ficulty in finalizing the transactions to global memory, even under relatively low contention scenarios. Developers should tread carefully when using atomic compare and swap operations in their kernels targeting the A100, as its performance may vary depending on the contention level.

Chapter 6

Conclusion

6.1 Summary

This work explored a variety of CUDA intrinsics and applications which demonstrated poor performance portability. For such a popular datacenter GPU, it is surprising how such a variety of workloads not only exhibited performance portability issues, but the A100 was often unable to reach the raw performance of the V100. Many of the observed challenges surrounded workloads that did not take advantage of all of the A100's SMs. This indicates that the A100 may be relying on its larger SM count for performance gains alongside newer features like tensor cores. In spite of this, atomics, for example, showed weakened performance portability even at very large grid sizes indicating trouble with highly contentious global memory transactions. This raises a number of fundamental questions concerning the underlying architecture and its inherent limitations. Therefore, it is clear that there remain some adaptational challenges preventing Ampere from completely supplanting Volta.

6.2 Programmer Recommendations

Through the course of developing this work, a number of suggestions that would have improved performance portability of the examined codes came to light. These are not foolproof as there are limitations to improving performance in the face of architectural obstacles.

Investigating New GPU Features Understanding the details of a new architecture can be a cumbersome and time consuming process. Developers want to build code and not necessarily become experts on a new architecture every two years. However, NVIDIA may provide clues to what has changed through new features that take advantage of the architecture or circumnavigate known issues. This is evident in the butterfly reduction code where the brand new reduce warp functions provided substantially higher performance. Additionally, benchmarking any code that has a fresh equivalent on a newer GPU is important to ensure that the change is not due to architectural challenges. This is especially important for released features that lack cross-compatibility with their older counterparts, such as in the case of reduce warp functions only supporting 32-bit integers.

Ensuring Optimized Code While highly optimized code is always the objective for any serious high performance computing, the prefix sum analysis is a lesson for how performance portability is reduced with suboptimal code. In this case, it was the presence of bank conflicts that was hurting performance portability. Therefore, algorithms which lack strong optimization may produce poor performance portability, especially if they contain bank conflicts.

Use Larger Grids A consistent theme that was present throughout most of the benchmarks was that smaller grid sizes showed lower performance portability. Therefore, scaling up programs to make use of all available threads will likely provide stronger performance portability.

GPU Performance Porting Step-by-Step With all of these ideas in mind, a more detailed step-by-step analysis can be put together on how to take a kernel running on one GPU and port it to another while maintaining performance.

1. Benchmark the kernel with Nsight Systems and Nsight Compute to obtain runtime information. Be sure to look for glaring runtime differences.
2. Examine Nsight Compute results line-by-line to look for differing stall counts of code. If a particular segment is consistently exhibiting longer stall times than the previous architecture, it may be in need of replacement. The key areas to be paying special attention to are anywhere that interacts with memory or performs a reduction at any scale. These are very typical bottlenecks in code, and they are much more likely to create performance portability challenges over a typical floating point operation.
3. Replace any algorithm that isn't meeting performance expectations. Changing out bank conflicts or highly-contentious atomic operations is a must, and the new features of the architecture can help provide inspiration on how to retrofit other areas. For example, butterfly reduction using *__shfl_xor_sync* should be replaced with the newer *__reduce_add*.

As for programmers specifically seeking to port code from the V100 to the A100, more detailed recommendations are provided.

Use Reduce Warp Functions Butterfly reduction, along with other warp-level intrinsics, indicated difficulty adapting to the Ampere architecture, but the reduce warp functions exhibited strong performance. If possible, making the change to these would be ideal.

Avoid Bank Conflicts Performance portability is sensitive to code that contains bank conflicts on the A100. If benchmarks indicate they are present during execution, alternative algorithms may need to be explored.

Be Wary of Atomics For some problem sizes, atomics on the A100 demonstrated great performance portability with more than a 50% uplift over the V100. However, several

others exhibited some of the worst performance portability measured in this work. As a result, be wary of using atomics in any A100 code unless the warps are not excessively stalling. Analyzing the performance of atomic operations within a kernel is essential for determining if that specific scenario is scaling well with the new architecture.

6.3 Future Work & Limitations

This work aimed to open the door to discussions surrounding performance portability between modern NVIDIA GPUs. It is not a complete evaluation of all applications and CUDA language features. However, it seems reasonable to assume that more performance portability challenges would be uncovered in a larger study. Firstly, the grid sizes of applications explored in this work remained mostly consistent, with many of the benchmarks having 1024 threads per block. Since performance varies with grid size per architecture, testing performance portability against differing grid dimensions would provide additional insight into this work's results, particularly as it relates to memory access patterns. Secondly, other work can also be leveraged to ensure a truly broad range of applications that uniquely stress the GPU is incorporated [37]. Lastly, different primitives could be more broadly surveyed to compare their performance portability. Butterfly reduction already provided some insight, with the newer reduce warp functions only supporting 32-bit integers. With more examples of poor performance portability, these could be documented together and turned into an application that automatically detects and mends code found to underperform on a given architecture. This hypothetical benchmark would test applications with a variety of grid sizes, including what is computed to be the most optimal for the problem [21]. Eliminating the struggle to maintain large applications between generations, through automated remediation, would be of tremendous benefit to developers and organizations worldwide.

Bibliography

[1] CUDA In Action - Research Apps. URL https://developer.nvidia.com/cuda-action-research-apps.

[2] D. Englebart. Microelectronics and the art of similitude. In *1960 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, volume III, pages 76–77, 1960. doi: 10.1109/ISSCC.1960.1157297.

[3] NVIDIA's New Ampere Data Center GPU in Full Production. URL https://nvidianews.nvidia.com/news/nvidias-new-ampere-data-center-gpu-in-full-production.

[4] NVIDIA Launches Revolutionary Volta GPU Platform, Fueling Next Era of AI and High Performance Computing. URL https://nvidianews.nvidia.com/news/nvidia-launches-revolutionary-volta-gpu-platform-fueling-next-era-of-ai-and-high-performance-computing.

[5] CUDA C++ Programming Guide. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/#programming-model.

[6] Using Shared Memory in CUDA C/C++. URL https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

[7] S. J. Pennycook, J. D. Sewall, and V. W. Lee. A metric for performance portability, 2016. URL https://arxiv.org/abs/1611.07409.

[8] Hartwig Anzt, Yuhsiang M. Tsai, Ahmad Abdelfattah, Terry Cojean, and Jack Dongarra. Evaluating the performance of nvidia's a100 ampere gpu for sparse and batched

computations. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 26–38, 2020. doi: 10.1109/PMBS51919.2020.00009.

[9] Ivan R. Ivanov, Oleksandr Zinenko, Jens Domke, Toshio Endo, and William S. Moses. Retargeting and respecializing gpu workloads for performance portability. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 119–132, 2024. doi: 10.1109/CGO57630.2024.10444828.

[10] Robert Lim, Boyana Norris, and Allen Malony. Autotuning gpu kernels via static and predictive analysis. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 523–532, 2017. doi: 10.1109/ICPP.2017.61.

[11] OpenCL Conformance. URL https://www.khronos.org/conformance/adopters/conformant-companies.

[12] YetiWare Inc. AJ Guillon. An introduction to opencl c++. Technical report, The Kronos Group, 2015. URL https://www.khronos.org/assets/uploads/developers/resources/Intro-to-OpenCL-C++-Whitepaper-May15.pdf.

[13] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *2011 International Conference on Parallel Processing*, pages 216–225, 2011. doi: 10.1109/ICPP.2011.45.

[14] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407, 2012. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2011.10.002. URL https://www.sciencedirect.com/science/article/pii/S0167819111001335. APPLICATION ACCELERATORS IN HPC.

[15] Manuel Costanzo, Enzo Rucci, Carlos García-Sánchez, Marcelo Naiouf, and Manuel Prieto-Matías. Comparing performance and portability between cuda and sycl for protein database search on nvidia, amd, and intel gpus. In *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 141–148, 2023. doi: 10.1109/SBAC-PAD59825.2023.00023.

[16] Ivona Vasileska, Pavel Tomšič, Leon Kos, and Leon Bogdanović. Unveiling performance insights and portability achievements between cuda and sycl for particle-in-cell codes on different gpu architectures. In *2024 47th MIPRO ICT and Electronics Convention (MIPRO)*, pages 1115–1120, 2024. doi: 10.1109/MIPRO60963.2024.10569866.

[17] István Z. Reguly. Performance portability of multi-material kernels. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 26–35, 2019. doi: 10.1109/P3HPC49587.2019.00008.

[18] Zheming Jin and Jeffrey S. Vetter. Performance portability study of epistasis detection using sycl on nvidia gpu. In *Proceedings of the 13th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, BCB '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393867. doi: 10.1145/3535508.3545591. URL https://doi.org/10.1145/3535508.3545591.

[19] Matthias Noack, Florian Wende, and Klaus-Dieter Oertel. Chapter 19 - opencl: There and back again. In James Reinders and Jim Jeffers, editors, *High Performance Parallelism Pearls*, pages 355–378. Morgan Kaufmann, Boston, 2015. ISBN 978-0-12-803819-2. doi: https://doi.org/10.1016/B978-0-12-803819-2.00001-X. URL https://www.sciencedirect.com/science/article/pii/B978012803819200001X.

[20] Es1 (Einsteinium) GPU Cluster. URL https://scienceit-docs.lbl.gov/hpc/systems/einsteinium/.

[21] Xuewen Cui and Wu-chun Feng. Iterml: Iterative machine learning for intelligent parameter pruning and tuning in graphics processing units. *J. Signal Process. Syst.*, 93 (4):391–403, April 2021. ISSN 1939-8018. doi: 10.1007/s11265-020-01604-4. URL https://doi.org/10.1007/s11265-020-01604-4.

[22] Ganesh Bikshandi and Jay Shah. A case study in cuda kernel fusion: Implementing flashattention-2 on nvidia hopper architecture using the cutlass library, 2023. URL https://arxiv.org/abs/2312.11918.

[23] Zhongming Yu, Guohao Dai, Guyue Huang, Yu Wang, and Huazhong Yang. Exploiting online locality and reduction parallelism for sampled dense matrix multiplication on gpus. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 567–574, 2021. doi: 10.1109/ICCD53106.2021.00092.

[24] Eric Nielsen Gabriel Nastac Christopher Stone, Aaron Walden and Mohammad Zubair. Performance optimization methods for a memory-bound, unstructured-grid cfd application on massively parallel gpu platforms. https://ntrs.nasa.gov/citations/20220002949, 2022. Presentation.

[25] Jou-An Chen, Hsin-Hsuan Sung, Xipeng Shen, Nathan Tallent, Kevin Barker, and Ang Li. Bit-graphblas: Bit-level optimizations of matrix-centric graph processing on gpu. *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 515–525, May 2022. doi: 10.1109/ipdps53621.2022.00056.

[26] Ruipeng Li and Chaoyu Zhang. Efficient parallel implementations of sparse triangular solves for gpu architectures. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*, pages 106–117. doi: 10.1137/1.9781611976137.10. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611976137.10.

[27] Yuechen Lu and Weifeng Liu. Dasp: Specific dense matrix multiply-accumulate units accelerated general sparse matrix-vector multiplication. In *SC23: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2023. doi: 10.1145/3581784.3607051.

[28] Yizhuo Wang, Fangli Chang, Bingxin Wei, Jianhua Gao, and Weixing Ji. Optimization of sparse matrix computation for algebraic multigrid on gpus. *ACM Trans. Archit. Code Optim.*, 21(3), September 2024. ISSN 1544-3566. doi: 10.1145/3664924. URL https://doi.org/10.1145/3664924.

[29] Andy Adinets and Duane Merrill. Onesweep: A faster least significant digit radix sort for gpus, 2022. URL https://arxiv.org/abs/2206.01784.

[30] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[31] Hubert Nguyen. *Gpu gems 3*. Addison-Wesley Professional, first edition, 2007. ISBN 9780321545428.

[32] turboSETI. python based seti search algorithm. https://github.com/UCBerkeleySETI/turbo_seti.

[33] Elan Lavie. Voyager 1 turboSETI Tutorial. https://github.com/elanlavie/VoyagerTutorialRepository/blob/master/VoyagerTutorial.ipynb.

[34] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on gpus. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 73–84, 2019. doi: 10.1109/CGO.2019.8661187.

[35] Jolly Chen, Monica Dessole, and Ana-Lucia Varbanescu. Migrating cuda to sycl: A hep case study with root rdataframe. In *Proceedings of the 12th International Workshop on OpenCL and SYCL*, IWOCL '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400717901. doi: 10.1145/3648115.3648122. URL https://doi.org/10.1145/3648115.3648122.

[36] Bieito Beceiro, Jorge González-Domínguez, Laura Morán-Fernández, Verónica Bolón-Canedo, and Juan Touriño. Cuda acceleration of mi-based feature selection methods. *Journal of Parallel and Distributed Computing*, 190:104901, 2024. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2024.104901. URL https://www.sciencedirect.com/science/article/pii/S0743731524000650.

[37] Vignesh Adhinarayanan and Wu-chun Feng. An automated framework for characterizing and subsetting gpgpu workloads. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 307–317, 2016. doi: 10.1109/ISPASS.2016.7482105.