Martin Rebne Farstad

Understanding the Key Performance Trends of Optimized Iterative Stencil Loop Kernels on High-End GPUs

Master's thesis in Computer Science
Supervisor: Magnus Jahre
June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Martin Rebne Farstad

Understanding the Key Performance Trends of Optimized Iterative Stencil Loop Kernels on High-End GPUs

Master's thesis in Computer Science
Supervisor: Magnus Jahre
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Problem Description

Iterative Stencil Loops (ISLs) are the performance bottleneck of many problems within computational science, and the objective of this thesis is to better understand the key performance trends of ISLs on high-end GPUs. The student is hence expected to conduct an extensive performance analysis of highly optimized ISL kernels and thereby identify and explain key performance trends. Meeting this objective will require significant implementation work. First, the student should implement ISL kernels for both 2D and 3D problem domains. Second, they should investigate the performance impact of commonly used optimizations such as using shared memory, thread coarsening, and autotuning. Third, the student should identify numerous stencil types from a wide range of ISL applications and explore to what extent performance trends are stencil-dependent. If time permits, the student should investigate how the implemented code scales on tightly-coupled multi-GPU systems such as the NVIDIA DGX-2.

Abstract

Iterative Stencil Loops (ISLs) are the main computational kernel of many applications within computational science. This thesis explores the key performance trends of ISLs on high-end GPUs through an extensive performance analysis of individual and combinations of optimizations. The optimizations include utilizing shared memory, thread coarsening, and autotuning thread block dimensions for optimizing Reverse Time Migration stencils in 2D and 3D domains. We benchmark our ISL application on two modern GPU architectures: Pascal (P100) and Volta (V100), observing up to 20.3x (12.7x) and 6.07x (3.34x) increased performance for Volta's shared memory kernel over Pascal's baseline kernel in 2D (3D) for 128 MiB and 8 GiB domains, respectively. We crucially find that device memory write (local memory) throughput bottlenecks our 2D (3D) kernels on Volta and suggests that cache utilization composes the performance bottleneck for Pascal. Our optimizations improve the kernels' performance over the architectures' respective baseline kernels by up to 1.47x (1.36x) on Volta and 3.64x (1.44x) on Pascal in 2D (3D) by improving the bottlenecks. Finally, we extend our optimization approach by utilizing up to 16 V100 GPUs. Our multi-GPU scheme achieves near-linear performance scaling by offloading the workload onto more devices, increasing performance over single-GPU by up to 14.8x.

Sammendrag

Iterative stensil-løkker (ISL) er hovedberegningskjernen til mange applikasjoner innen beregningsvitenskap. Denne masteroppgaven utforsker de viktigste ytelsestrendene for ISL-applikasjoner ved hjelp av spesielt kraftige grafikkprosessorer ved å gjennomføre en omfattende ytselsesanalyse av individuelle og kombinasjoner av optimaliseringer. Optimaliseringene inkluderer bruk av delt minne, trådforgrovning og autotuning av trådblokkdimensjoner for å optimalisere "Reverse Time Migration"-stensiler for to- og tredimensjonale domener. Vi evaluerer applikasjonen vår på to moderne arkitekturer: Pascal (P100) og Volta (V100), og observerer ytelsesforbedringer på opp til 20.3x (12.7x) og 6.07x (3.34x) for Volta's kjerne med delt minne i forhold til Pascal's enkleste kjerne for todimensjonale (tredimensjonale) domener med domenestørrelser henholdsvis på 128 MiB og 8 GiB. Vi oppdager at skrivehastigheten til hovedminnet (lokalt minne) er en flaskehals for applikasjonen vår med to (tre) dimensjoner på Volta og indikerer at hurtigminne utgjør flaskehalsen for Pascal. Optimaliseringene våre forbedrer kjerneytelsen i forhold til arkitekturenes respektive enkleste kjerner med opp til 1.47x (1.36x) på Volta og 3.64x (1.44x) på Pascal for todimensjonale (tredimensjonale) domener ved å forbedre flaskehalsene. Videre utvider vi optimaliseringsmetoden vår til å anvende opp til 16 V100 grafikkprosessorer. Ved å fordele arbeidsmengden mellom grafikkprosessorerene klarer vi å oppnå tilnærmet lineær ytelsesskalering som øker ytelsen i forhold til én grafikkprosessor med opp til 14.8x.

Preface

The thesis builds upon our previous work [1] for the specialization project performed in Fall 2020 for the TDT4501 course. The project applied the Jacobi method to approximate the numerical solution of Laplace's equation in 2D by iteratively calculating a five-point stencil. The project utilized shared memory, grid synchronization through Cooperative Groups, autotuned thread block dimensions, multiple GPUs (up to four V100 GPUs in a different system), and a technique to prevent unnecessary inter-GPU synchronization. However, we configured the baseline thread block dimensions naively and did not optimize occupancy. Crucially, the previous implementation was considerably more superficial, and the analysis was shallow and gave limited insights.

Our previous project's weaknesses motivated our decision to significantly extend our implementation, perform a more comprehensive performance analysis, and locate the performance bottlenecks. This thesis is an extensive extension of the previous work that includes optimizations that improve performance significantly for different stencils in 2D and 3D. Additionally, introducing 3D presents significant implementation complexities and different application behavior. This thesis also generalizes the concept of stencil computations from a specific iterative method to ISLs in general, targeting a broader audience.

As our previous work implemented the Jacobi method on GPUs, certain parts of the background material are relevant for this thesis. However, we do not expect the reader to have read our previous work, so we base certain parts of our background material on our previous work to include all overlapping aspects. This adaptation is standard practice at NTNU, and we presently present which parts of our background our thesis bases on and to which degree it overlaps with our previous work:

- Section 1.1 bases certain parts of the GPU architecture theory on our previous work.
- Section 2.2 bases certain parts of the execution model and thread synchronization theory on our previous work.
- Section 2.6 bases most parts on our previous work.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

1.1 Motivation

ISLs are the key computational kernel within a range of compute-intensive applications, including image processing, data mining, weather- and climate modeling, as well as physical simulations such as seismic or fluid simulations [2, 3]. ISLs are among the most important computational kernels of contemporary and future applications, residing within the category of structured grids [4]. In a multi-dimensional discretized grid, the method computes each element as a function of its neighbors. The number of neighbors calculated depends on the number of dimensions at hand and the stencil's size. This technique often simulates a time-step (e.g., Conway's Game of Life [5]) that can converge to a solution (e.g., iterative methods [6]). Initially, ISLs were studied widely for CPU applications [7–10]. ISLs often compose large parts of applications' total execution time (up to 100% for SPEC 2017's Lattice Boltzmann Method [3, 11]), motivating the interest for offloading such computations onto high-performant accelerators [12].

Although initially proposed as a hardware-accelerator for computing specialized graphics computations in parallel [13, p. 8], Graphics Processing Units (GPUs) extended their domain onto general-purpose computing by realizing high performance through a highly parallel execution model by extensive latency hiding. Utilizing the vast amount of threads for parallelism, GPUs became the de facto hardware accelerator for compute-intensive applications within High-Performance Computing. Meanwhile, energy-constrained environments make offloading stencil computations onto FPGAs also widely recognized [3, 14, 15]. Utilizing GPUs as accelerators for processing ISLs has been studied extensively in the last decade [2, 16–18]. ISLs' widespread importance sparks the need for efficient GPU implementations. However, as we will show later, the applications' performance depends on the underlying GPU architecture. Therefore, this thesis studies ISLs on two GPU architectures: Pascal and Volta.

As presented in Figure 1.1, the GPU differentiates itself from the CPU by allocating more on-chip area for execution units as a trade-off for cache and control flow logic like branch predictors and prefetchers. The GPU's programming model

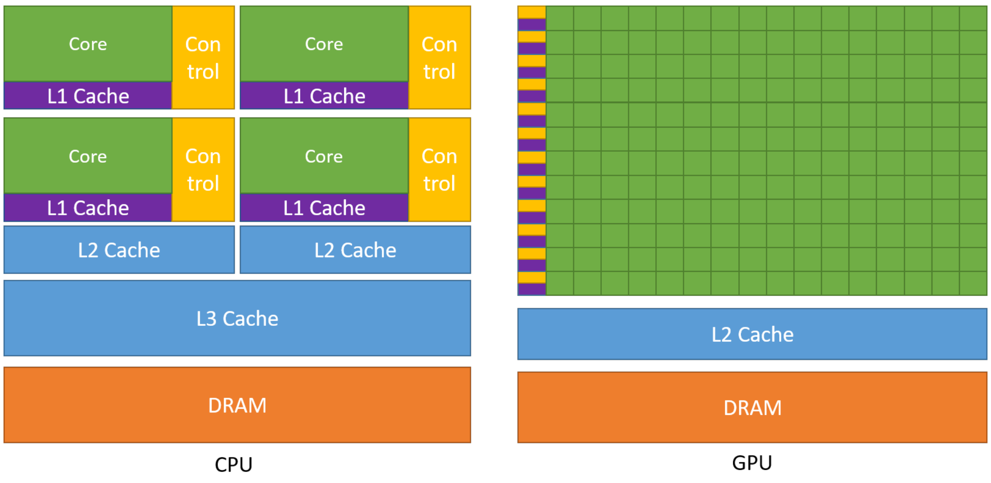


Figure 1.1: The figure displays an example distribution of a multi-core CPU vs. a many-core GPU. More transistors are devoted to computation in GPUs. From [19, p. 2]. Reprinted with permission from Nvidia.

does not require such resources to find runtime parallelism because this is the programmer's task. The GPU architectures emphasize high throughput through many lower-frequency cores compared to the CPU's few high-frequency cores. As a result, the GPU programming model's weak spot is high latency, making applications with limited parallelism better suited for CPUs than GPUs.

The GPUs employ an array of Streaming Multiprocessors (SMs) to achieve extensive hardware parallelism [13, p. 68]. Each row in Figure 1.1's simplified GPU architecture layout represents an SM comprising an L1/unified cache, control units, and numerous execution units. GPUs also achieve significant parallelism by employing fast context switches and lightweight threads [13, p. 90]. The ISLs' grid-structured problem domains map elegantly to the many-core GPU architectures. The cores' incredible parallelism composes a powerful tool to solve ISLs.

The Compute Unified Device Architecture (CUDA) [19] serves as the primary platform for Nvidia GPUs currently dominating the market, which we focus on in this work. CUDA enables implementing ISLs on GPUs at a relatively low abstraction level but enables the programmer more detailed optimizations to prevent redundant memory requests to the device memory (e.g., shared memory or register caching [20]) or increase the number of in-flight memory requests (e.g., thread coarsening [21, 22]). Furthermore, optimizing ISLs require significant attention to memory access patterns to avoid redundant accesses to the device memory [13, p. 158]. Moreover, using automatic code generation tools to aid optimization is possible [12, 23]. However, modern GPU architectures improve ISLs' performance inconspicuously for the programmer, making naive implementations highly competitive [24]. Therefore, we wish to investigate the performance gap between naive and more advanced implementations for contemporary architectures.

Modern domain workloads for ISLs rapidly exceed the capacity of a single

GPU, motivating the need to offload computation onto more GPUs. Distributing workload between multiple GPUs is an excellent optimization for improving performance. Additionally, multi-GPU also improves performance tremendously for moderate domain sizes, offering a great optimization to combine with one or more single-GPU optimizations. However, the addition of numerous GPUs comes at a cost. Operating multiple devices requires creating GPU contexts associated with each device. Furthermore, the GPUs' asynchronous execution implies a need for proper synchronization to ensure program correctness. The synchronization implies an overhead to the application and requires synchronization through the host or between the devices themselves. Finally, we need to make sure our multi-GPU scheme is scalable by making the application's source code handle an arbitrary number of GPUs. Scalable multi-GPU applications can potentially increase performance linearly with the number of GPUs added.

1.2 Assignment Interpretation

The problem description states that our main objective is to better understand the key performance trends of ISLs on high-end GPUs through an extensive performance analysis of highly optimized ISL kernels. To state our interpretation of the problem description, we define the following tasks:

- T1 Implement an ISL application in 2D and 3D and explain the key performance impacts of moving from 2D to 3D.
- T2 Optimize the ISL application's kernels through utilizing shared memory, thread coarsening, and autotuning. Identify and explain the optimizations' key performance-enhancing factor by localizing the application's performance bottlenecks on high-end GPUs.
- T3 Include numerous stencils from multiple ISL applications and explain to which extent the performance trends depend on the stencil type.
- T4 Compare the ISL application's performance on different high-end GPUs to learn whether our results are architecture-dependent.
- T5 Implement a scalable multi-GPU scheme by distributing the workload between more GPUs and analyze its scaling capabilities.

1.3 Contributions

We describe our contributions by highlighting their objective task in parentheses. This thesis implements an ISL application in 2D and 3D (T1) and contributes a thorough analysis into utilizing a combination of shared memory, thread coarsening, and autotuned TB dimensions (T2) for optimizing Reverse Time Migration (RTM) stencils (T3). Our analysis finds that introducing shared memory achieves the most stencil-dependent performance improvements (T3). Furthermore, we propose a simplistic but powerful scheme for choosing thread block (TB) dimensions heuristically. The approach maximizes occupancy, ensures correct execution, and provides a baseline to compare against autotuned TB dimensions. Moreover, we investigate the optimizations' key performance trends on two modern GPU architectures: Pascal and Volta, emphasizing the incredible performance improvements achieved by upgrading from Pascal to Volta (T4).

Introducing shared memory kernels improves performance over the architectures' respective baseline kernels by up to 1.22x (1.01x) in 2D (3D) on Volta and 3.27x (1.43x) on Pascal for an 8 GiB domain. Applying thread coarsening improves the kernels' performance over the architectures' respective baseline kernels by up to 1.30x (1.10x) on Volta and 3.64x (1.44x) on Pascal in 2D (3D) for an 8 GiB domain. Autotuning the coarsened kernels' TB dimensions improves performance over the baseline kernel by up to 1.47x (1.36x) on Volta. Comparing Volta's kernels to Pascal's baseline kernel exhibits an incredible 20.3x (12.7x) performance improvement for a 128 MiB domain and 6.07x (3.34x) for an 8 GiB domain in 2D (3D).

Our analysis crucially finds that DRAM write (local memory) throughput bottlenecks our 2D (3D) application on Volta (T2), indicating that moving from 2D to 3D introduces significant register spilling (T1). Unfortunately, we do not have permission to profile our application on Pascal. However, utilizing shared memory has a significant impact on Pascal, indicating that cache utilization composes the performance bottleneck for Pascal instead of DRAM throughput (T2).

Finally, we further improve the performance by partitioning the workload between multiple GPUs (T5), keeping the domain sizes equal to the single-GPU analysis. Consequently, the analysis focuses on strong scaling instead of weak scaling, enabling a more direct comparison of multi-GPU against single-GPU optimizations. Furthermore, the implementation utilizes strip partitioning in 2D and 3D to distribute the workload between devices, enlarging the partitions with ghost zones to ensure correct execution. Our multi-GPU scheme's performance improvements closely resemble the number of GPUs utilized for most executions (T5), increasing performance over single-GPU by up to 14.8x.

1.4 Outline

- Chapter 2 introduces the theoretical concepts required to understand the presented implementation and refers to related work.
- Chapter 3 presents and explains the approach for implementing the optimized ISL application and presents the source code.
- Chapter 4 presents the experimental setup for benchmarking the optimized ISL application.
- Chapter 5 presents measurements and analyses our optimized ISL application.
- Chapter 6 concludes our contributions and presents future work for improving the presented implementation.
- Appendix A presents the autotuned TB dimensions generated for our kernels on a single GPU.

Chapter 2

Background

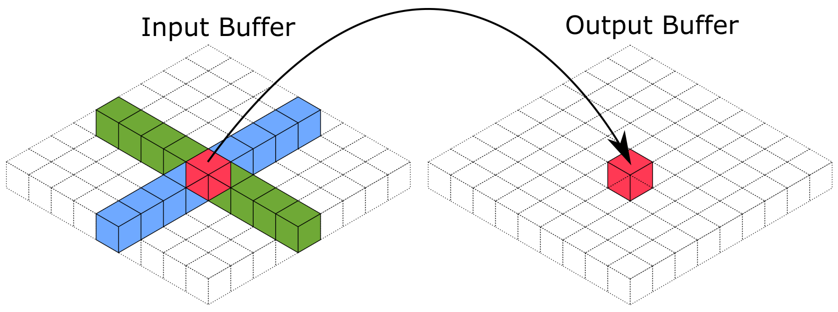
The following chapter introduces ISLs and describes multiple stencil patterns commonly used within scientific applications. Subsequently, we propose the motivation for using general-purpose GPU (GPGPU) computing to accelerate our ISL application and describe multiple approaches for optimizing the implementation.

2.1 Iterative Stencil Loops

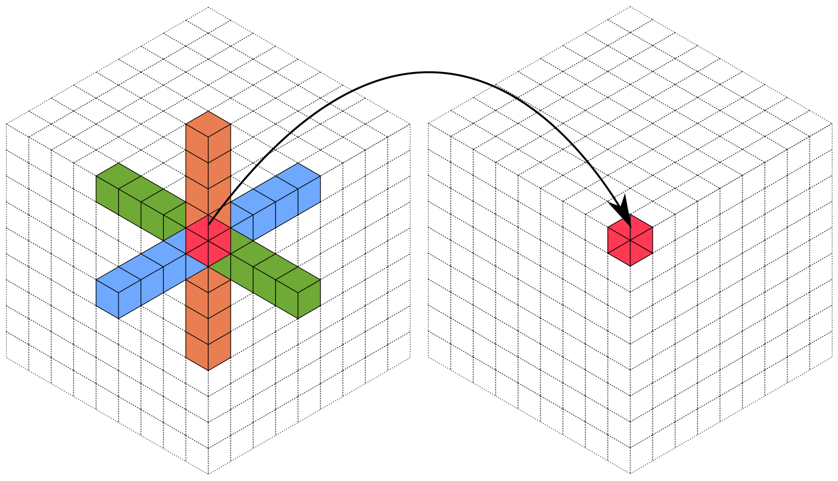
ISLs represent a significant number of compute-intensive applications within High-Performance Computing [3]. ISLs traverse grids and iteratively update cell values based upon their neighboring cells multiplied by a set of coefficients. For every iteration, input cells are read from one buffer and updated in the other before swapping buffers to prepare for the next iteration. The discretized domains range from one to three dimensions, representing a line (1D), plane (2D), or cube (3D). Higher dimensions require calculating more neighboring cells, increasing the GPU's workload. The distance to neighboring elements within the linear memory layout increases along with introduced dimensions, constituting offsets of a line (plane) in 2D (3D) for accessing a value along the last dimension. Additionally, increasing the number of dimensions decreases each domain dimension's size due to limited GPU memory size.

2.1.1 Stencil Patterns

Different stencil patterns exist as a result of the characteristics of widely different underlying methods. The vast number of patterns emerges as the number of underlying methods is extensive. Koraei et al. [3] used three stencil patterns to analyze ISLs on FPGAs: Jacobi, Gauss-Seidel, and RTM. In 2D (3D), their Jacobi pattern featured a 3x3(x3) square (cube), while Gauss-Seidel formed an X-shaped pattern within the same shape. Their RTM pattern featured a cross-shaped pattern within a 5x5(x5) square (cube). This thesis focuses on RTM patterns only, although we analyze the pattern differently; we utilize the same shape, although



(a) 2D



(b) 3D

Figure 2.1: Calculating a Jacobi iteration of the RTM stencil pattern in 2D and 3D. The green, blue, and brown elements represent the stencil's radius-sized neighborhood within the X-, Y-, and Z-dimension, respectively. The red element represents the value written to the output buffer.

we study more RTM stencil sizes, observing the stencils' impact on GPU resource usage.

We define the stencil's size as the stencil's *radius* and define it as the number of neighboring elements along each dimension from the center. Additionally, we denote a given stencil with stencil radius r by R_r. Our ISL application uses an input and output buffer for computing the stencil (Jacobi iterations [25, p. 12]). Figures 2.1a and 2.1b display the RTM stencil, R_4, in 2D and 3D, respectively. The figures compute a Jacobi iteration using the center and neighboring values from the input buffer and write an updated center value to the output buffer. The figures' colors emphasize the stencil's properties: green, blue, and brown elements represent the neighbors read from the input buffer along the X-, Y-, and Z-dimension, respectively. The red element represents the value written to the output buffer. This procedure is repeated for all elements within the input buffer's domain and represents a single ISL iteration.

2.1.2 Spatial and Temporal Blocking

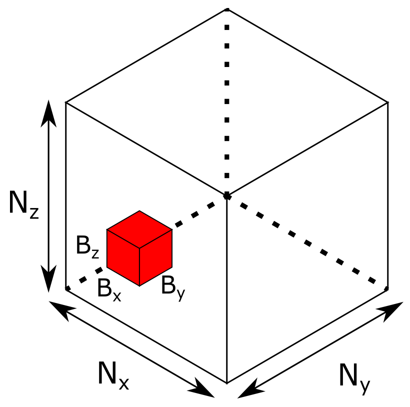
ISLs show considerable spatial locality utilizable by the GPU architecture's caches. Spatial blocking utilizes the caches more efficiently by processing subsets of the domain that fit well into the caches. This technique reduces the pressure on slower memory accesses, further increasing the performance. Temporal blocking reuse input data within the cache before writing to the output buffer. Spatial blocking and temporal blocking is prevalent for optimizing ISLs on CPUs [8, 26–28], GPUs [2, 12, 16, 29, 30], and FPGAs [15, 31, 32].

Spatial blocking provides limited reuse opportunities by reusing each grid point only a few times, depending on the stencil pattern. In general, spatial blocking is insufficient to transform memory-bound kernels into compute-bound [16]. Temporal blocking applies in combination with spatial blocking by calculating subsequent iterations within the spatial block before fully computing the first iteration for the whole domain [15].

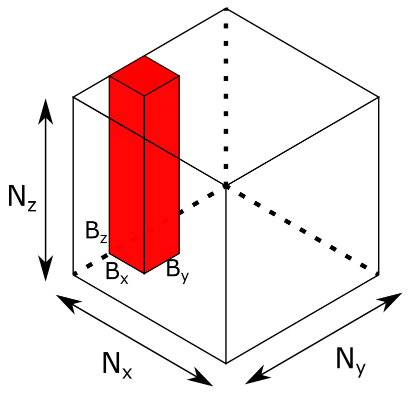
Figure 2.2a visualizes the 3D-blocking algorithm, a commonly used technique for dividing the input into overlapping axis-aligned three-dimensional blocks [16]. The scheme maps effortlessly onto TBs, with each thread handling a single element within the block. The argument is similar for 2D-blocking in 2D. Unfortunately, memory bandwidth essentially limits the scheme's performance. However, the technique imposes fewer requirements on the cache capacity compared to more sophisticated approaches. Therefore, this thesis implements 3D-blocking to ensure compatibility with an arbitrary domain size and proposes more sophisticated blocking algorithms as future work.

Nguyen et al. [16] presented a 3.5D-blocking algorithm for ISLs on CPUs and GPUs that combined 2.5D-spatial and 1D-temporal blocking using on-chip memory. Figure 2.2b visualizes their 2.5D-blocking algorithm, which utilizes the fact that every Z-value required grid values within a stencil radius' range in Z-direction to be resident within the cache. The approach performs blocking through the XY plane and *streaming* through the third dimension. The authors argued that streaming removes bandwidth requirements from the Z-direction by preventing the same elements from being loaded multiple times, leading to substantial performance improvements by achieving near-optimal memory bandwidth utilization. However, this approach requires the cache capacity to include the blocked data.

The authors further argued that temporal blocking is the only way to reduce bandwidth further after implementing 2.5D blocking and provided thread-level and data-level parallel algorithms to utilize computing resources fully. Furthermore, the authors argued that their 3.5D blocking scheme made the ISL application compute-bound, increasing performance significantly by removing the memory bandwidth dependency. Moreover, the authors argue that 4D blocking (3D-spatial + 1D-temporal) improves performance by reducing bandwidth through temporal reuse but emphasize that it introduces significant overhead compared to their 3.5D approach.



(a) 3D-blocking



(b) 2.5D-blocking

Figure 2.2: Spatial blocking in 3D. Based on [16].

2.2 General-Purpose GPU Computing Using CUDA

2.2.1 Execution Model

CUDA is a parallel computing platform developed by Nvidia containing a programming model extending multiple programming languages, including the C programming language [19, p. 2]. The programming model enables programmers to write kernels, essentially data-parallel functions using CUDA runtime libraries for procedure calls and GPU device manipulation [19, p. 15]. CUDA utilizes the Single Instruction Multiple Threads (SIMT) architecture to manage and execute groups of 32 threads called warps. The architecture differs from the well-known Single Instruction Multiple Data (SIMD) architecture by allowing multiple threads in the same warp to execute independently, having a private instruction counter, register state, and local memory space [13, p. 68-69]. The independent execution enables multiple threads to issue equal instructions to arbitrary data rather than a single thread issuing vector instructions across many data elements [33].

From the programmer's perspective, threads organize into TBs with per-thread private registers and access to per-block shared memory. The TBs compose an execution grid with access to a global memory space containing per-SM L1 and a global L2 cache [34, p. 6]. Figure 2.3 provides an overview of the execution model, where the programmer decides the TB dimensions and grid dimensions. When the GPU schedules TBs for execution to an SM, the warp scheduler dynamically groups 32 threads into warps and executes the warps in lockstep across available SMs. Three execution hierarchies exist from the hardware's perspective: Cooperative-Thread-Array (CTA), warp, and SIMD-lane [35]. Launching a kernel maps TBs and threads onto CTAs and SIMD-lanes, respectively. Warp-level execution remains transparent to the programmer unless the application utilizes warp-level primitives (e.g., via warp shuffle functions [19, p. 168]).

All active threads within a warp execute the same instruction on different

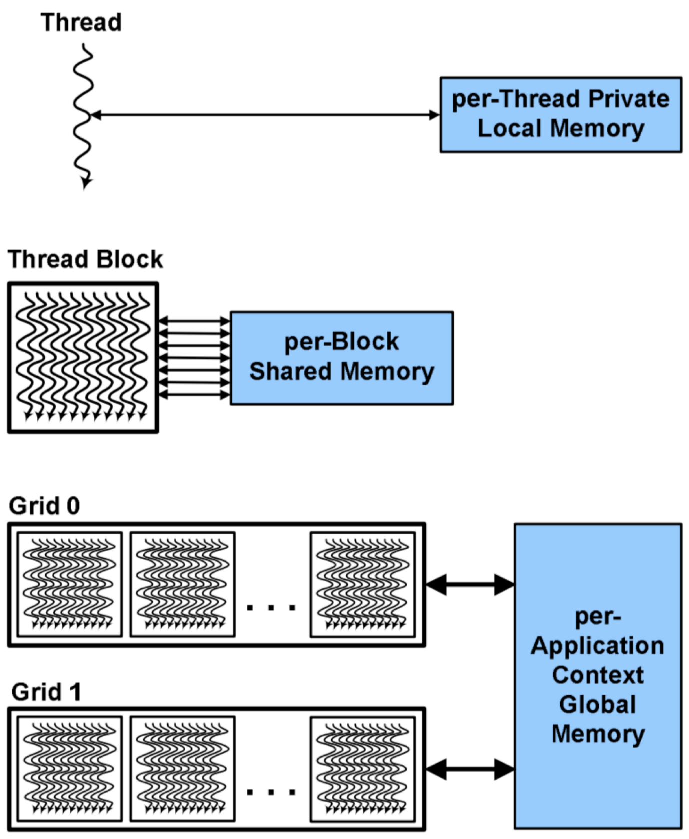

Thread
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
Grid 1
per-Application Context Global Memory

Figure 2.3: CUDA thread execution with per-thread local, per-block shared, and per-application global memory spaces. From [34, p. 6]. Reprinted with permission from Nvidia.

data elements. The coalescer combines the threads' memory requests into cache requests, quickly brought back into an SM's compute cores. If the request results in a cache miss, the system allocates one Miss Status Holding Register (MSHR) [36, 37, p. 2] and continues to search in the lower parts of the cache hierarchy. The warp scheduler schedules another warp if the active warp's request misses the last level cache, hiding the expensive memory operation's latency through Thread-Level Parallelism (TLP). However, the cache can maximally serve as many in-flight misses as the number of MSHRs, exposing the SMs to the total DRAM access latency should no MSHRs be available [38].

2.2.2 Thread Synchronization

Simultaneous execution of a substantial number of concurrent threads requires synchronization to prevent *race conditions* [13, p. 97]. A race condition implies non-serialized access by concurrently executing threads to the same memory location, resulting in unknown access orders. GPUs provide thread synchronization at different granularities to battle this issue. Threads synchronize either explicitly through barriers or implicitly when exiting kernels. CUDA 9 introduced Cooperative Groups (CG) [39] to provide synchronization capabilities at multiple granularities, extending from warps to multiple grids. Previously, TB synchronization was the only available synchronization method. However, synchronizing threads at a granularity coarser than TBs using CG generally does not improve performance [40], explaining our choice for sticking with implicit synchronization.

Li et al. [35] emphasized that current technology trends favor allocating more lightweight CTAs for processing individual tasks more independently in favor of fewer heavily-loaded CTAs with significant intra-CTA data reuse. By reducing the number of threads per TB (CTA) to the warp size, the authors increased performance significantly by replacing inter-warp communication (e.g., via shared memory), cooperation, and synchronizations (e.g., via bar primitives [41, p. 218]) with more efficient intra-warp communication (e.g., via warp shuffle functions [19, p. 168]), cooperation (e.g., via warp vote functions [19, p. 165]) and synchronizations (SIMT lockstep execution) across the SIMD-lanes within a warp.

2.3 The GPU Memory Architecture

2.3.1 Registers

Registers reside on the top of the memory hierarchy within GPUs. Each SM contains a register file partitioned among the active warps. Their scope is private to each thread and contains variables or arrays having compile-time-determined indices referring to it [13, p. 138]. The registers' lifetime is equal to that of a kernel, and the number of registers allocated by a kernel can severely limit a kernel's performance. Modern GPUs limit the number of registers per thread (block) to 255 (65536) [42]. The kernel spills registers over to *local memory* should the number of registers surpass the hardware limit. However, local memory resides within device memory, making accesses have high latency and low bandwidth. Local memory depends on memory coalescing in the same manner as global memory accesses with a layout organized as consecutive 32-bit words accessible by consecutive threads. Therefore, accesses coalesce as long as a warp's threads access the same relative address (e.g., same index in an array variable) [19, p. 118].

Warp-level primitives enable the use of registers as a high-end cache for stencil computations [20]. The warp-shuffle instructions enable threads to exchange values with other threads within the same warp. Pooling registers from a warp's threads creates a small cache with high performance. The number of registers per thread and the number of threads per warp limits the cache's size. The incredible register usage impacts the achieved occupancy but does not necessarily lead to worse performance [43]. Furthermore, register caching frees up shared memory usage, enabling using shared memory as a second-level cache [20] or creating more space for the L1 cache within the unified cache.

Falch and Elster [20] implemented register caching for a 1D 7-point and 2D 5-point stencil. As a single shuffle instruction could replace a shared memory load, store, and the in-between barrier, the authors argued that their implementation was best suited for applications reading and writing to shared memory. The authors argued that register caching indicated more redundant computation, more loads from global memory, and increased branch divergence than shared memory applications. Their work did not focus on optimizing ISLs particularly but instead evaluated register caching with a realistic ISL benchmark, showing promising results that increased performance up to 2.04x on a GTX 680 GPU.

2.3.2 Shared Memory

Shared memory is essentially a software-managed L1 cache (scratchpad) with much higher bandwidth and lower latency than global memory. The cache is either stand-alone or combined with the L1 cache into a unified cache. Using shared memory can hide the performance impact of global memory bandwidth and latency [13, p. 204-206]. The programmable shared memory differentiates itself from the L1 cache by controlling what data to store and where. For the L1 cache, the hardware loads and evicts data automatically [13, p. 213]. Every warp within a TB shares the same shared memory region, indicating the need for TB synchronization after fetching data into shared memory to ensure program correctness. Subsequently, each warp can fetch data from shared memory stored by other warps within the same TB.

Memory banks are physically separate memory modules that handle a subset of the memory address space [44]. Equally-sized shared memory banks allow shared memory to simultaneously serve multiple memory requests if the requests map to different banks. The Volta architecture employs 32 4-byte banks per SM, offering a theoretical bandwidth of 13,800 GiB/s across 80 SMs [45]. However, bank conflicts can decrease shared memory bandwidth utilization.

Bank conflicts occur when the addresses of multiple memory requests map to the same memory bank, requiring serialized access to serve the memory requests. The hardware splits the memory requests with bank conflicts into separate requests to perform conflict-free requests to the banks. This process can decrease shared memory throughput by a factor equal to the number of replayed memory bank accesses [19, p. 119]. However, it is crucial to note that the memory accesses must come from different threads within the same warp. Bank conflicts do not occur for threads in different warps or instructions arising from the same thread. This latter fact indicates that the stencil computation for RTM stencils can safely access elements along the Y-dimension, even though the offset between each element implies accessing the same memory bank. Therefore, bank conflicts are not a significant issue for RTM stencils.

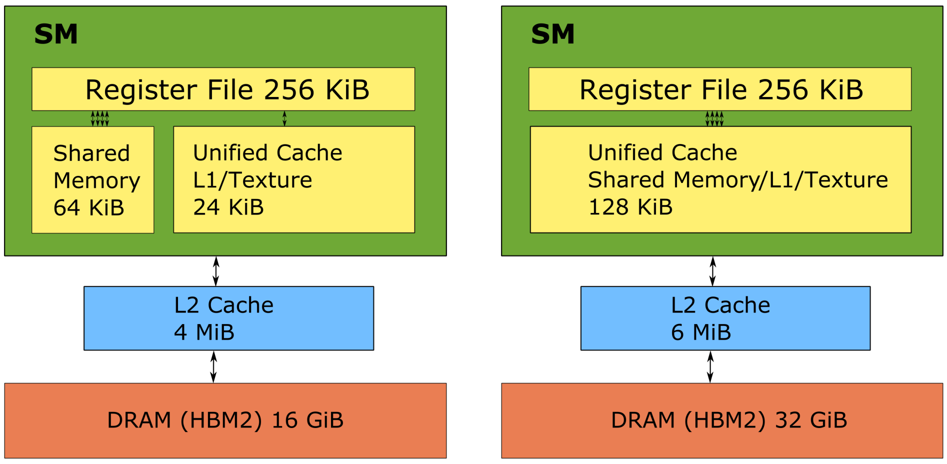
The amount of shared memory available for each SM is reconfigurable, and utilizing more than 48 KiB of shared memory per TB is architecture-specific, explicitly requiring dynamic shared memory allocations [19, p. 343]. Dynamic shared memory implies an overhead compared to static allocations and requires passing the allocation's size as a parameter to the launched kernel. The TBs cannot allocate the entire region of an SM's shared memory as the system retains a small allocation for system use. Furthermore, if the available shared memory amount per CTA is lower than per SM, a single TB can not allocate the whole SM's available region by itself. Lastly, as the Volta architecture combines shared memory and L1 into a unified cache, where configuring more shared memory for the SM is a trade-off for less L1 cache space available.

2.3.3 Unified Cache

Figure 2.4 presents simplified memory hierarchy diagrams of the Pascal and Volta architectures, showing the architectures' different cache configurations. The Volta architecture provides a unified shared memory/L1/texture cache where up to 96 KiB of the 128 KiB L1 cache storage can be configured dynamically as programmable shared memory, whereas the Pascal architecture provides separate L1 and shared memory. This hardware feature has significant consequences for the performance benefit of utilizing shared memory in ISL applications on Volta vs. Pascal.

Choquette et al. [24] benchmarked a set of applications designed to use shared memory and remapped their design to utilize the L1 cache instead for both reads and writes. They argued that Volta's cache unification caused the L1 cache's performance characteristics to be much closer to the shared memory's characteristics. The authors estimated that their remapped applications exhibited 30% performance decreases on average on Pascal, while the performance decrease was only 7% on Volta. They further argued that "programmers will be able to get reasonably high performance without the additional effort of programming to shared memory."

Furthermore, Volta's L1 caches drastically outperform Pascal by increasing the total size by a factor of 7.7 from 1.3 MiB up to 10 MiB while increasing the L2



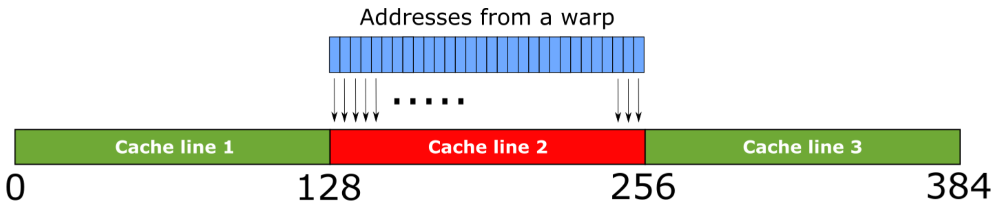
(a) Pascal (P100) **(b)** Volta (V100)

Figure 2.4: The figures present simplified memory hierarchy diagrams of the Pascal and Volta architectures. The main difference between the architectures lies in Pascal's separate shared memory and L1/texture cache versus Volta's unified shared memory/L1/texture cache. The diagrams simplify the layout by only presenting a single SM instead of replicating many SMs. Based on [13, p. 159]

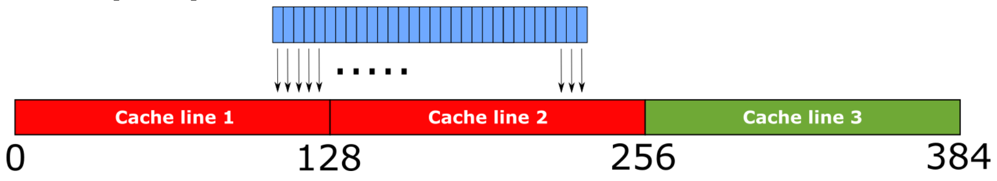
cache from 4 MiB to 6 MiB. Volta's L1 cache also has four times the bandwidth of Pascal's L1 cache. Additionally, Volta introduces a *streaming* L1 cache [42], substantially increasing the number of possible cache-misses in flight with a large number of MSHRs, reducing the severe performance impact of divergent memory accesses [38] and also making thread coalescing more viable as an optimization by enabling more in-flight memory requests.

2.3.4 High Bandwidth Memory

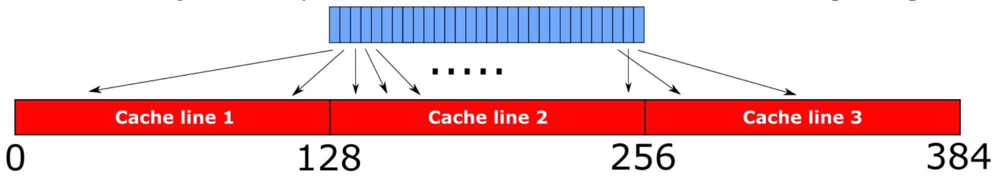
Both Pascal and Volta use the High Bandwidth Memory 2 (HBM2) memory system as their DRAM (device memory). The HBM2 memory system comprises memory stacks located on the same physical package as the GPU, providing substantial power and area savings compared to traditional GDDR5 memory designs [42]. Volta utilizes an improved HMB2 reaching up to 900 GB/s peak memory bandwidth, compared to 732 GB/s from Pascal's HBM2 system [46]. Volta's memory system also consists of an improved memory controller, which, when combined with the improved memory system, improves the theoretical memory bandwidth by 50% over Pascal, and Nvidia reports that the achieved bandwidth efficiency exceeds 95% bandwidth efficiency for specific workloads [46]. However, to fully utilize the HBM2's bandwidth over the GDDR5, more memory accesses must be kept in flight. In this thesis, we use thread coalescing to increase the number of in-flight memory requests. Furthermore, the advanced Volta system engages large numbers of SMs, increasing the number of concurrent threads and, thus, the in-flight memory requests compared to previous architectures.



(a) Aligned and coalesced memory access. The memory coalescer coalesces the memory accesses into a single cache line request. The load/store unit serves all warps' requests by fetching a single cache line per warp.



(b) Misaligned memory access. The load/store unit fetches two cache lines per warp.



(c) Uncoalesced memory access. The load/store unit fetches multiple cache lines per warp (up to 32 in the worst-case).

Figure 2.5: The following figures visualize memory access pattern characteristics. Figure 2.5a presents the ideal situation where fetching a single cache line is sufficient to satisfy a warp's memory requests. Figure 2.5b displays misaligned accesses' unfortunate consequence of requesting twice the number of cache lines required. Figure 2.5c presents the most severe situation fetching up to 32 cache lines per warp compared to Figure 2.5a's single cache line request per warp. Based on [13, p. 161-162].

2.4 Desirable GPU Characteristics

2.4.1 Aligned and Coalesced Memory Accesses

Memory bandwidth bottlenecks most GPU applications, making maximizing bandwidth efficiency crucial [13, p. 158]. Figure 2.5 presents three memory access patterns: coalesced, misaligned, and divergent, where divergent and misaligned memory access patterns decrease performance significantly. The GPU's L1 (L2) cache consists of 128-byte (32-byte) cache lines. Ideally, a memory request's first address should be an even multiple of the cache granularity to prevent fetching an extra cache line [13, p. 162]. As the cache-line sizes differ, the consequences for divergent L1 loads are more severe than for divergent L2 loads. Therefore, our following example focus on divergent L1 access patterns.

For the L1 cache, using 4-byte floats (single-precision) enables an ideal situation where the coalescer can combine 32 memory loads of a warp into a single cache request. This situation requires the loads' offsets to be contiguous within the 128-byte segment to ensure maximum efficiency. However, strided offsets incur large overheads for the memory system. The worst-case scenario occurs for 128-byte offsets, where the threads' loads request 32 different cache lines (4096 bytes) to satisfy a single warp's memory requests for 32 4-byte elements (128 bytes), reaching a meager 3.125% efficiency. Additionally, the requests allocate 32 MSHRs compared to a single MSHR in the coalesced example, and the enormous amount of memory requests essentially flood the GPU's Network-on-Chip and DRAM systems [38].

Vast domains incur large offsets for RTM's cross-shaped stencil in the Y-direction (2D) or Z-direction (3D). The last dimension offset for 3D (plane) quickly outgrows 2D's offset (line), decreasing the probability that the requested cache lines reside within the caches. As a result, the memory system allocates more MSHRs and saturates the NoC and DRAM systems more substantially in 3D than 2D to satisfy the memory requests, causing severe issues for maintaining TLP.

The streaming executing model of the GPU implies that applications can be memory-bound or compute-bound [47]. If compute-bound, the performance scales linearly with the number of occupied SMs, and if memory-bound, performance is strongly correlated with memory bandwidth. ISL applications are famously memory-bound [31], but reducing the memory-boundedness is possible through temporal blocking [15].

2.4.2 Effective Resource Utilization

Occupancy is a metric expressing the ratio of active warps per SM to the total number of hardware warp-slots per SM [35]. The metric focuses exclusively on the number of concurrent warps per SM and is, therefore, not the only goal for performance optimization [13, p.97]. Cheng et al. [13, p. 100] provide an excellent example showing that higher occupancy does not always mean higher performance. The authors show that increasing occupancy at the expense of decreasing other vital metrics, such as the efficiency of global memory loads, can decrease performance. For memory-bound applications, the consequences of decreasing memory efficiency can be fatal. Therefore, the programmer must be aware of the metrics' trade-offs. The metrics can be studied using the performance analysis tools Nvidia Visual Profiler (NVP) [48] or Nvidia Nsight Compute [49].

Each SM contains a finite set of registers and shared memory partitioned between the SM's executing warps. Additionally, the SMs have an upper bound on the number of warps eligible for simultaneous execution, causing the number of threads per block to decide how many blocks the GPU can execute simultaneously. Occupancy decreases if either resource is exhausted, making it crucial to adhere to these restrictions for optimizing the kernel's achieved occupancy. Therefore, manipulating the TB dimensions can expose sufficient parallelism to saturate the system resources. The CUDA Occupancy Calculator [50] is an excellent tool to guide the developer on how to handle these resources effectively.

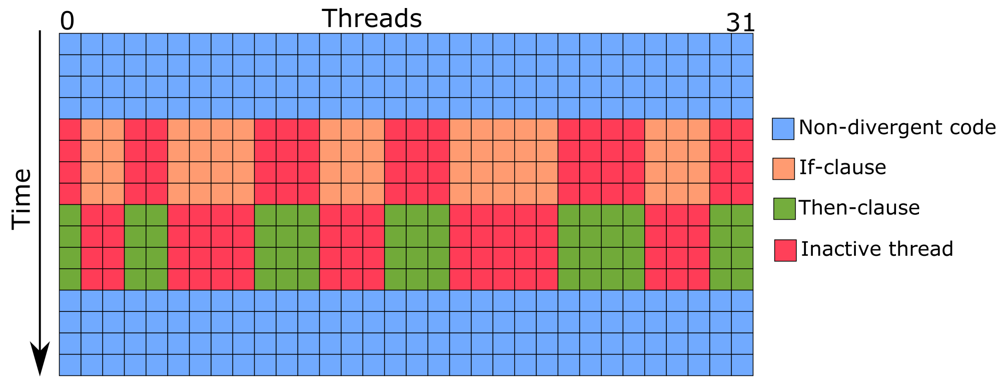


Figure 2.6: The figure displays threads' behavior when branch divergence occurs. The warp's threads remain active when executing non-divergent code. If a warp's threads take different paths when reaching a conditional branch, the warp executes each path serially, deactivating threads that do not follow the current execution path. Based on [13, p. 83].

2.4.3 Avoiding Branch Divergence

Warps execute single instructions at a time, realizing total efficiency only when all threads within the warp do not have divergent execution paths [19, p. 107]. Each execution path still executes if the path of threads within a warp diverges on a conditional branch. However, the scheduler disables the threads not following the executed path. Figure 2.6 presents a situation where branch divergence occurs. Branch divergence occurs within warps only and is therefore independent of the execution behavior of other warps. Traditionally, warps have had a single execution state shared by all threads. The Volta architecture introduced per-thread execution states, thus allowing independent thread scheduling, although the execution model remains SIMT. The feature improves the divergence and reconvergence of threads within warps by making them more flexible and efficient [35]. However, branch divergence does not disappear with the added feature as the hardware still restricts the execution to only a single instruction at a time [51].

2.5 Single-GPU Optimization Approaches

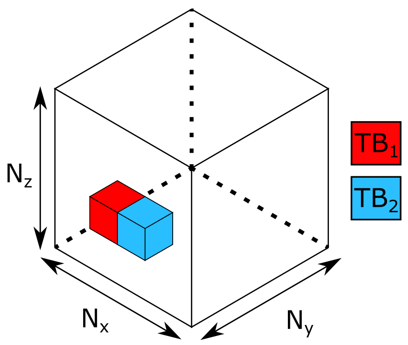
2.5.1 Thread Coarsening

Decomposing a problem into a set of fine-grained tasks distributed between the GPU's threads inevitably introduce overheads from different sources varying by the algorithm or method implemented. The implemented kernel might map unfavorably to the underlying hardware, causing poor compute resource utilization. Finer-grained decompositions incur inefficiencies by scheduling and communication overhead and needing to recalculate light operations in many threads, e.g., address offsets for memory requests [22].

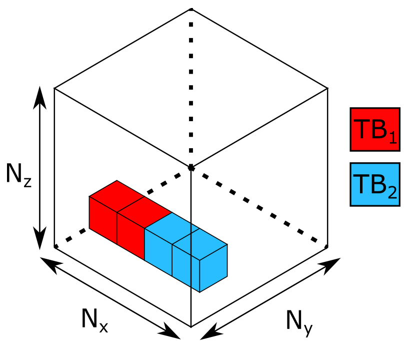
Loop unrolling is an optimization technique utilized by the preprocessor to increase the code size and register pressure as a trade-off for increased concurrency and data reuse [52]. The technique reduces the total number of branches and loop maintenance instructions for loops by duplicating the loop body instructions multiple times. Lack of branch prediction mechanisms makes branching expensive for GPU applications, indicating the available optimization potential. Additionally, the technique creates more independent instructions to schedule, enabling more in-flight operations to provide higher instruction and memory bandwidth efficiency [13, p. 114]. Utilizing the GPU's exceptionally lightweight warp scheduling enables the opportunity to hide latency by scheduling new warps when others stall for arithmetic instructions or memory operations. Therefore, more in-flight operations help the system hide more instruction and memory latency [13, p. 115]. The compiler provides automatic loop unrolling for loops of limited size with known loop conditions [19, p. 208].

Thread coarsening [21, 22] takes advantage of loop unrolling by merging code usually executed by separate threads into a single thread. The procedure effectively de-parallelizes a program, making threads more coarse-grained by increasing each thread's workload. The coarsening factor describes the factor by which we increase the amount of work per thread. Thread coarsening gets similar performance benefits as loop unrolling, combined with potentially improving problem decomposition. At some point, the reduced number of threads will limit parallelism and decrease performance. Furthermore, increasing the coarsening factor increases resource consumption (e.g., registers, shared memory), eventually reducing occupancy. Additionally, some kernels increase pressure on the caches with an increasing coarsening factor [21]. The performance trade-off motivates the search for an ideal coarsening factor or deciding that no coarsening should be applied at all. Exploring the coarsening factor can be explored manually [53, 54] or automatically through either autotuning [55] or machine-learning [56]. This thesis explores the ideal coarsening factor manually through benchmarking.

Two thread coarsening strategies remain dominant: thread- and block-level coarsening [21]. The former combines the threads' workload within a TB, while the latter merges multiple TBs' workload into one. Figure 2.7 visualizes the block-level coarsening approach utilized to optimize our ISL application. These approaches have different impacts on memory access patterns as each coarsening strategy imposes different memory access strides. Magni et al. [53] studied thread-level and Unkule et al. [55] explored block-level coarsening. Stawinoga and Field [21] scrutinized both strategies, concluding that block-level outperformed thread-level coarsening due to practical issues when deciding the memory stride for each thread, arguing that poor striding breaks memory coalescing. They further emphasize that block-level coarsening has no issues regarding memory coalescing. Furthermore, by utilizing a conservative approach for electing kernels for coarsening, the authors either increased or kept performance flat in most cases. The authors' conclusions motivate our inclination for focusing on block-level coarsening only.



(a) The figure presents two neighboring TBs executing without thread coarsening.



(b) The figure displays the effect of coarsening each TB with $CF = 2$.

Figure 2.7: The figures present block-level thread coarsening, which combines multiple blocks' workload into one TB. The coarsening factor, CF, decides how many blocks to merge. Figures 2.7a and 2.7b presents an example without ($CF = 1$) and with ($CF = 2$) thread coarsening. Furthermore, doubling the coarsening factor halves the number of scheduled TBs.

The coarsening strategy also impacts TB synchronization performance. As fewer threads (thread-level coarsening) or blocks (block-level coarsening) participate in the block-level synchronization barriers, performance increases for barrier-sensitive applications. Furthermore, Stawinoga and Field [21] observed that both strategies have a similar effect on barrier performance. Liu et al. [57] highlighted the importance of reducing cycles spent in barriers for barrier-sensitive applications as the number of cycles proliferates even though barriers employ lightweight hardware support. Therefore, coarsening impacts the performance of the kernels within this thesis that employ TB synchronizations (i.e., the kernels utilizing shared memory).

2.5.2 Optimizing Thread Block Dimensions

Section 2.4.2 emphasized the TB dimensions' importance for effectively using GPU resources and exposing sufficient parallelism. Choosing TB dimensions using the CUDA occupancy calculator is often suboptimal, as maximizing occupancy does not necessarily produce the best performance. However, testing all available TB dimension combinations is extremely difficult as the number of possible configurations is massive.

An autotuning framework enables TB dimension exploration either by brute-force or more sophisticated approaches using search algorithms. However, imposing restrictions on the parameters prevent complete domain exploration. Cheng et al. [13, p. 96] proposed guidelines for choosing grid and block sizes, emphasizing the importance of keeping the innermost dimension a multiple of the warp size to keep all threads within each warp active. Additionally, keeping the number of TBs much greater than the number of SMs generally improves parallelism.

Spencer [58] proposed the Flamingo autotuning framework, providing brute-force parameter exploration, enabling TB dimension and thread coarsening factor optimization. The author designed the framework to be very general-purpose, making few assumptions on user interaction, making it remarkably simple. This thesis utilizes Flamingo to optimize the TB dimensions for each thread coarsening factor, exploring more ideal configurations for improved resource consumption.

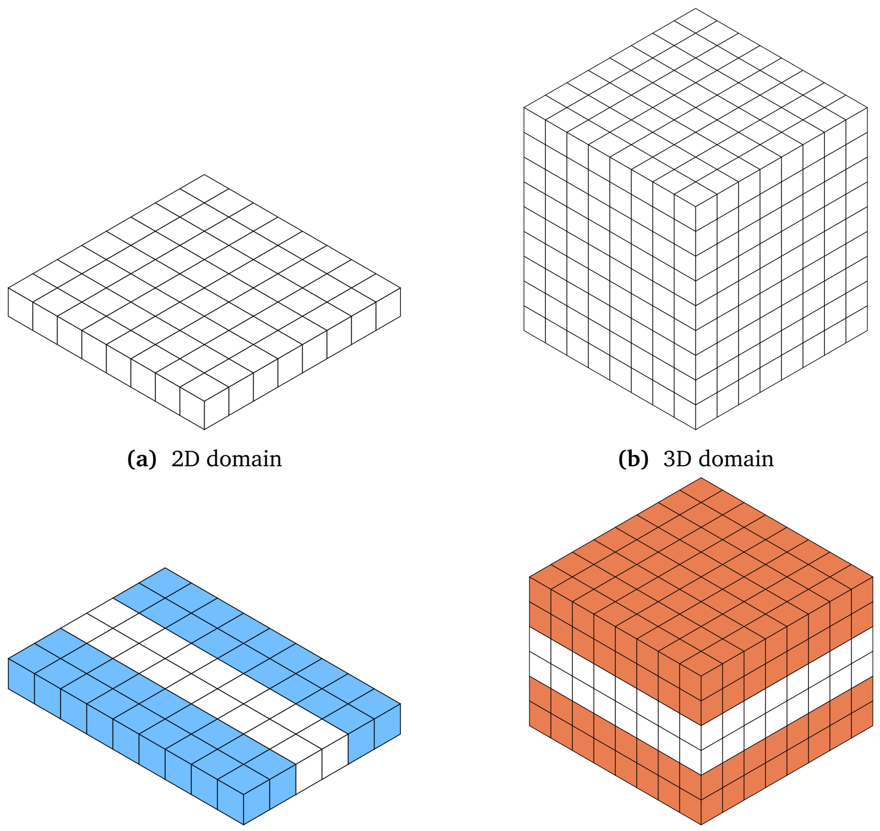
2.6 Utilizing Multiple GPUs

2.6.1 Domain Partitioning

Extracting the computation power added by utilizing multiple GPUs requires problem domain partitioning between the GPUs. Spampinato [59] contributed an investigation of multi-GPU data partitioning in 2D and the communication pattern required for each iteration in his scheme's iterative method. He specified the prevailing domains of both strip and block partitioning. If we assume a 2D (3D) square (cube) domain with equal dimension sizes, strip partitioning subdivides the grid along the y-axis (z-axis) into horizontal strips (planes). In comparison, block partitioning partitions the domain evenly into smaller squares (cubes). Assuming a system containing 4 GPUs, Spampinato argued that strip partitioning reduces communications by half for the GPUs having partitions with only a single neighbor. In comparison, block partitioning incurs the same number of communications for every GPU but needs to transfer fewer bytes per communication.

With differing communication requirements, data partitioning proposes a trade-off between latency and bandwidth. The optimal structure depends on the system architecture and the application at hand. Spampinato [59] refers to Wilkinson and Allen [60], stating that strip partitioning is better for high-latency communication systems, while block partitioning is better for low-latency communication. However, Spampinato concluded that this model was too general and not suited for multi-GPU systems because it does not include the incredible number of architectural hardware details necessary to provide an accurate model. Furthermore, the author's results showed that strip partitioning outperformed block partitioning for its setup containing 4 GPUs. Therefore, we choose to utilize strip partitioning in our multi-GPU scheme.

Another option not pursued in this thesis is to use frameworks for automatic problem domain partitioning onto multiple GPUs. Ben-Nun et al. [61] proposed a multi-GPU framework called MAPS-Multi, providing a programming abstraction that automatically allocates and distributes the workload among multiple GPUs and optimizes applications for the given architectures. The framework is also capable of providing memory transfers implicitly if required by the application. A host- and device-level API provides this abstraction and includes off-the-shelf functionality for various architectures containing an arbitrary number of GPUs. The API significantly reduces code length and achieves near-linear performance scaling by adding more devices for various applications.



(a) 2D domain

(b) 3D domain

(c) 2D domain partition

(d) 3D domain partition

Figure 2.8: The figure shows an example of strip partitioning in 2D and 3D for multi-GPU. Figures 2.8a and 2.8b display simplified grid layouts by setting each dimension to 8 elements. Figures 2.8c and 2.8d show each GPUs partition including their ghost zones when we utilize 4 GPUs in 2D and 3D, respectively.

2.6.2 Ghost Zones

Partitioning domains between multiple GPUs introduces a problem where stencil calculations along the boundary require elements residing in other devices' memory. A naive solution for ISL applications is communicating the required elements when necessary, but this approach implies a significant communication and synchronization overhead. Therefore, we augment the partitions to include *ghost zones* [2], essentially overlapping neighboring partitions causing redundant computations.

Figure 2.8 visualizes a strip partitioning of an 8x8(x8) domain between 4 GPUs in 2D (3D), including ghost zones applicable for computing the R_2 stencil. The figure's domains are superficially small for illustrational purposes. Partitioning the 2D (3D) domain between 4 GPUs creates an 8x2 (8x8x2) domain for each GPU shown in Figure 2.8c (Figure 2.8d), significantly reducing the workload per GPU. Figures 2.8c and 2.8d enlarge the second and third dimensions with additional ghost zone elements visualized with blue and brown cells, respectively.

Reducing inter-GPU communication is possible by increasing the computational workload by enlarging the partitions' ghost zones. Spampinato [59] argued that increasing the ghost zone's depth by a given size, n, reduces the communication factor by the same value. Thus, each iteration shrinks the ghost zone by one row (plane) for 2D (3D) until the number of iterations equals n. The author further argued that the optimal ghost zone size depends on the application and architecture and is a trade-off between the system's latency and transmission time. Therefore, it is beneficial as long as the new transmission overhead does not exceed the previous overhead. However, the concept belonged originally to compute clusters. At maximum, the author increased performance only by 1.1x for his setup using 4 GPUs. Our previous work [1] implemented a similar communication reduction scheme using the same number of GPUs but showed no significant performance improvements.

2.6.3 Handling Inter-GPU Communication

CUDA streams are sequences of operations executed in the order they are issued and can also provide a valuable mechanism for overlapping communication with computation [13, p. 268]. Our multi-GPU implementation utilizes a single stream per GPU to handle communications, enabling direct memory transfers between the devices asynchronously. Using multiple streams can significantly improve communication performance in some applications by overlapping communication with neighboring GPUs.

As device context switches are independent, host multi-threading allows parallel kernel launches, removing the single-threaded overhead imposed by switching GPU contexts and launching the kernel. Also, it prevents the main thread from stalling, as would be the case for only using a single thread. Preventing stalls can be significant in real-world applications where the thread needs to attend other tasks [62]. As a result, multi-threading provides improved task-parallelism for the CPU, resulting in increased system performance.

Sourouri et al. [62] provided an efficient inter-GPU communication scheme, utilizing multiple OpenMP [63] threads with multiple streams to overlap communication with computation. The additional streams facilitated inter-GPU communication, using two streams per neighboring device, each responsible for either the sending or receiving part of the communication. The authors emphasized the importance of utilizing lightweight OpenMP threads instead of MPI [64] processes to maximize performance. As a result, a device can simultaneously communicate with both neighbors, reducing the communication stages into a single stage, effectively doubling communication performance. The authors provided an example requiring half the original number of communications to emphasize the benefit of utilizing multiple streams. The authors also studied using either PCIe or GPU-Direct but did not look into the effects of having the novel NVLink, NVSwitch, or NV-SLI interconnects.

The main drawback of older multi-GPU implementations is the need to stage memory transfers through the host, a requirement imposed by PCIe interconnects, requiring twice the number of memory transfers as direct communication between the devices on a single node. However, modern multi-GPU systems use different NVLink, NVSwitch, or NVI-SLI versions, providing alternatives to the typical PCIe interconnect. This project applies direct P2P communication between the devices through an NVSwitch system on a single compute node.

Li et al. [65] studied numerous interconnect topologies in multi-GPU systems, contributing a thorough investigation into the execution behavior altered by the interconnects. The authors measured latency, bandwidth and studied the emerging performance factors of employing different topologies. The authors made a crucial observation by discovering that some modern GPU interconnects (e.g., NVLink) showed non-uniform memory access effects that increased latency, although previously claiming transparency. However, the NVSwitch interconnect utilized in this thesis showed no such deficiencies.

Chapter 3

Implementing Optimized ISL Kernels

The following chapter presents and discusses our ISL application by introducing multiple kernels. The kernels calculate the RTM stencil for equal inputs, whose workload can reside within both L1 or shared memory. We present the kernels' source code in 3D as it is more sophisticated than 2D, although the 2D-equivalent is similar. However, we visualize the shared memory layout of both 2D and 3D kernels. The kernels correctly execute on different architectures where L1 and shared memory do not necessarily reside within the same unified cache. Furthermore, we coarsen the kernels' threads to increase the workload per thread. Table 3.1 summarizes preprocessor macros used throughout the following sections' code listings. Crucially, this chapter presents essential parts of the source code only. However, the complete source code is available on Github[1].

[1]see `https://github.com/mrfarstad/thesis`

Table 3.1: Constants frequently used in our ISL application

Variable	Description
DIMENSIONS	Specifies 2D or 3D kernels and domains
NX, NY, NZ	Problem domain dimensions
RADIUS	The stencil radius
STENCIL_COEFF	The stencil calculation's coefficient
ITERATIONS	The number of stencil iterations
BYTES_PER_GPU	Domain partition size per GPU in bytes
COARSEN_X	The coarsening factor (in the X-direction)
GHOST_ZONE	Number of elements per ghost zone
GHOST_ZONE_BYTES	Ghost zone size in bytes
GHOST_ZONE_DEPTH	Number of rows (planes) per ghost zone
NGPUS	Number of GPUs utilized

Table 3.2: Kernels used in our ISL application

Kernel	Description
`base`	Baseline RTM kernel using global memory only.
`base_coarsened`	Extends `base`. Handles CF elements per thread.
`smem`	Extends `base` by adding TB sized shared memory.
`smem_coarsened`	Extends `smem` by multiplying the shared memory allocation by CF. Handles CF elements per thread.
`smem_padded`	Extends `smem` by padding the shared memory allocation by the stencil radius.
`smem_padded_coarsened`	Extends `smem_coarsened` by padding the shared memory allocation. Handles CF elements per thread.

3.1 ISL Kernels

Table 3.2 presents the kernels representing a combination of various shared memory implementations and thread coarsening. We greatly simplify our discussions throughout the subsequent sections by introducing some abbreviations, denoting the coarsening factor for a given kernel by CF and the TB dimensions as B_x, B_y, B_z.

3.1.1 The Baseline Kernel

Code listing 3.1 presents the `base` kernel providing the most straightforward ISL implementation without any further optimizations. Lines 7 to 11 calculate the threads' index within the domain (i.e., the global index). Line 12 checks if the computed indices fit within the domain dimensions minus (or plus) an offset equal to the stencil radius. Line 13 computes the stencil by calling a function that fetches all the elements needed for the stencil calculation directly from global memory. Code listing 3.2 presents this stencil function as a series of accumulations, summing the values of all neighbors in each direction of the stencil. Lines 5 to 11 present the functions used to perform this accumulation, consisting of function calls where all functions essentially call the function defined in Code listing 3.3 with different parameters depending on the accumulation direction.

The global memory is cached, providing excellent reuse capabilities if the cache capacity can include a TB's workload. The stencil includes the stencil's center, meaning that the elements in the X-direction are contiguous. Finally, the kernel multiplies the sum of the neighboring elements with a predefined constant and subtracts the stencil's center element. We choose to define the constants using a preprocessor macro. If the stencil requires different constants, then loading the values into constant memory maintains satisfactory performance.

Code listing 3.1: The base kernel

```
__global__ void base_3d(
    float* __restrict__ d_in,
    float* __restrict__ d_out,
    unsigned int kstart,
    unsigned int kend)
{
    unsigned int i, j, k, idx;
    i  = threadIdx.x + blockIdx.x*blockDim.x;
    j  = threadIdx.y + blockIdx.y*blockDim.y;
    k  = threadIdx.z + blockIdx.z*blockDim.z;
    idx = i + j*NX + k*NX*NY;
    if (check_stencil_border_3d(i, j, k, kstart, kend))
       stencil(d_in, d_out, idx);
}
```

Code listing 3.2: The base stencil

```
__device__ __host__ __inline__ void stencil(
    float *in, float *out, unsigned int idx)
{
    float sum = 0.0f;
    accumulate_global_i_prev(&sum, in, idx);
    accumulate_global_i_next(&sum, in, idx);
    accumulate_global_j_prev(&sum, in, idx);
    accumulate_global_j_next(&sum, in, idx);
#if DIMENSIONS>2
    accumulate_global_k_prev(&sum, in, idx);
    accumulate_global_k_next(&sum, in, idx);
#endif
    out[idx] = sum / STENCIL_COEFF - in[idx];
}
```

Code listing 3.3: Global memory stencil accumulators

```
__device__ __host__ __inline__
void accumulate_prev(float *sum, float *in, unsigned int idx, int offset)
{
#pragma unroll
    for (unsigned int d=RADIUS; d>=1; d--)
        *sum += in[idx-d*offset];
 }

 __device__ __host__ __inline__
 void accumulate_next(float *u, float *in, unsigned int idx, int offset)
 {
#pragma unroll
     for (unsigned int d=1; d<=RADIUS; d++)
        *sum += in[idx+d*offset];
 }
```

Code listing 3.4: The base_coarsened kernel

```
 1  __global__ void base_coarsened_3d(
 2      float* __restrict__ d_in,
 3      float* __restrict__ d_out,
 4      unsigned int kstart,
 5      unsigned int kend)
 6  {
 7      unsigned int i, j, k, i_off, idx, lidx;
 8      i  = threadIdx.x + blockIdx.x*blockDim.x*COARSEN_X;
 9      j  = threadIdx.y + blockIdx.y*blockDim.y;
10      k  = threadIdx.z + blockIdx.z*blockDim.z;
11  #pragma unroll
12     for (lidx=0; lidx<COARSEN_X; lidx++) {
13          i_off = i + lidx*blockDim.x;
14          idx = i_off + j*NX + k*NX*NY;
15          if (check_stencil_border_3d(i_off, j, k, kstart, kend))
16              stencil(d_in, d_out, idx);
17      }
18  }
```

3.1.2 Coarsening the Baseline Kernel

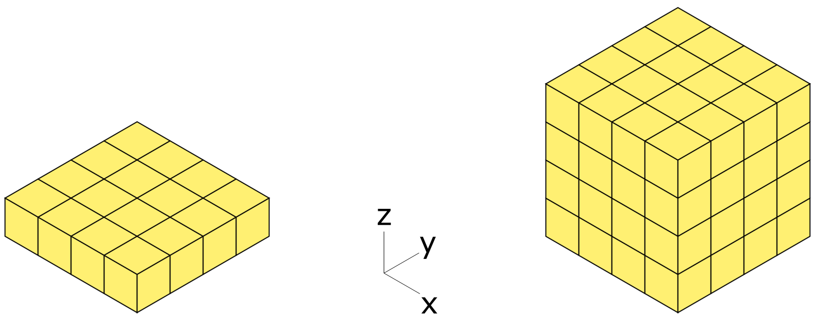
Code listing 3.4 presents the `base_coarsened` kernel, providing a direct extension of `base` by applying thread coarsening. Lines 12 to 17 show the kernel's *for*-loop that handles multiple data elements per thread. Line 11 forces the compiler to unroll the loop to expose more independent instructions. Before launching the kernel, dividing the total number of TBs in the grid by CF is essential for high performance as we need fewer threads to perform the calculation.

3.2 Shared Memory ISL Kernels

3.2.1 The Hybrid Shared Memory Kernel

The `smem` kernel shown in Code listing 3.5 is a naive shared memory implementation utilizing both shared and global memory when computing the stencil. The hybrid version employs the shared memory layout shown in Figures 3.1a and 3.1b for 2D and 3D, respectively. The kernel differentiates itself by its simplicity and without the need for intricacy: If an element does not reside within shared memory, fetch it from global memory. However, this scheme introduces thread-dependent control logic into the kernel, which implies branch divergence.

The kernel consists of two phases: the prefetching phase and the stencil calculation stage. The prefetching phase is straightforward and consists only of each thread fetching a single element from the global memory with the global index (i.e., Lines 16 and 17) into shared memory with the TB index calculated in Lines 13 to 15. The kernel allocates one shared memory entry per thread, causing the shared memory allocation to be the same size as the TB. TB synchronization separates the phases, synchronizing all threads within the TB.



(a) 2D **(b)** 3D

Figure 3.1: The figures show the shared memory layouts for the `smem` kernel in 2D and 3D. The layout is equal to the TB dimensions, and each thread within the TB fetches a single element into shared memory.

Code listing 3.6 presents the last stage that fetches the neighboring elements and calculates the stencil. Note that this shared memory layout does not contain all required elements for the stencil within the shared memory. Therefore the kernel fetches the elements outside the border from global memory instead, making it a hybrid using two memory types when calculating the stencil. Code listing 3.7 displays the hybrid accumulation procedure, which checks whether the stencil elements reside within shared memory in Lines 11 and 25.

Code listing 3.5: The smem kernel

```
__global__ void smem_3d(
    float* __restrict__ d_in,
    float* __restrict__ d_out,
    unsigned int kstart,
    unsigned int kend)
{
    unsigned int i, j, k, idx, sidx;
    extern __shared__ float smem[];
    i   = threadIdx.x + blockIdx.x*blockDim.x;
    j   = threadIdx.y + blockIdx.y*blockDim.y;
    k   = threadIdx.z + blockIdx.z*blockDim.z;
    idx = i + j*NX + k*NX*NY;
    sidx = threadIdx.x
         + threadIdx.y*blockDim.x*COARSEN_X
         + threadIdx.z*blockDim.x*COARSEN_X*blockDim.y;
    if (check_domain_border_3d(i, j, k, kstart, kend))
        smem[sidx] = d_in[idx];
    this_thread_block().sync();
    if (check_stencil_border_3d(i, j, k, kstart, kend))
        smem_stencil(smem, d_in, d_out, sidx, idx);
}
```

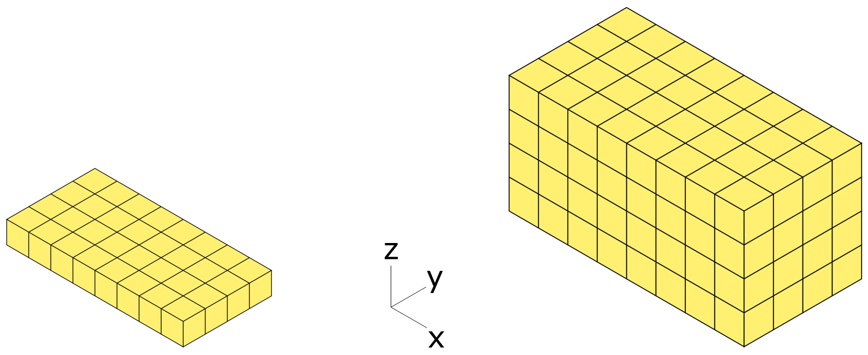
Code listing 3.6: The smem stencil

```
__device__ __inline__ void smem_stencil(
    float* smem, float* in, float* out, unsigned int sidx, unsigned int idx)
{
    float sum = 0.0f;
    accumulate_hybrid_i_prev(&sum, smem, in, sidx, idx);
    accumulate_hybrid_i_next(&sum, smem, in, sidx, idx);
    accumulate_hybrid_j_prev(&sum, smem, in, sidx, idx);
    accumulate_hybrid_j_next(&sum, smem, in, sidx, idx);
#if DIMENSIONS>2
    accumulate_hybrid_k_prev(&sum, smem, in, sidx, idx);
    accumulate_hybrid_k_next(&sum, smem, in, sidx, idx);
#endif
    out[idx] = sum / STENCIL_COEFF - smem[sidx];
}
```

Code listing 3.7: Hybrid global/shared memory stencil accumulators

```
 __device__ __inline__
 void accumulate_hybrid_prev(
     float *sum, float *smem, float *in,
     unsigned int sidx,
     unsigned int idx,
     unsigned int tidx,
     int soffset, int offset)
 {
#pragma unroll
     for (unsigned int d=RADIUS; d>=1; d--)
        *sum += (tidx >= d) ? smem[sidx-d*soffset] : in[idx-d*offset];
 }

 __device__ __inline__
 void accumulate_hybrid_next(
     float *u, float *smem, float *in,
     unsigned int sidx,
     unsigned int idx,
     unsigned int tidx,
     unsigned int tb_limit,
     int soffset, int offset)
 {
#pragma unroll
     for (unsigned int d=1; d<=RADIUS; d++)
        *sum += (tidx+d < tb_limit) ? smem[sidx+d*soffset] : in[idx+d*offset];
 }
```



(a) 2D **(b)** 3D

Figure 3.2: The figures show the shared memory layouts for the `smem_coarsened` kernel for $CF = 2$ in 2D and 3D. The coarsening factor multiplies the shared region's X-dimension.

3.2.2 Coarsening the Hybrid Shared Memory Kernel

The `smem_coarsened` kernel shown in Code listing 3.8 extends the previously proposed `smem` kernel, applying thread coarsening to handle more elements per thread. Thread coarsening expands the shared memory region in the X-dimension by a factor equal to CF. Therefore, the kernel's first phase consists of more global memory requests per TB compared to `smem`. Figures 3.2a and 3.2b display the expanded shared memory region with $CF = 2$ in 2D and 3D, respectively.

Increasing the shared memory usage per TB enables more shared memory reuse in the X-dimension for the elements near the center of the allocated region. After the synchronization stage, each thread calculates the stencil for CF elements using the hybrid global/shared memory stencil (i.e., Code listing 3.9). When combining TBs' work using thread coarsening, $CF > 2$ implies that some TBs' stencil computations in the X-direction can be performed entirely by fetching from shared memory. Therefore, Code listing 3.9 differs from Code listing 3.6 by checking whether the workload resides at the endpoints through the loop index in Lines 8 to 11. Figure 3.3 visualizes this circumstance by differentiating between green and yellow cubes. The green cubes represent the endpoints where computing the stencil along the X-dimension includes both global and shared memory values. The yellow cubes represent the in-between workloads that benefit from the expanded shared memory region by computing the X-dimension through fetching elements from shared memory only.

Code listing 3.8: The smem_coarsened kernel

```
__global__ void smem_coarsened_3d(
    float* __restrict__ d_in,
    float* __restrict__ d_out,
    unsigned int kstart,
    unsigned int kend)
{
    extern __shared__ float smem[];
    unsigned int i, j, k, idx, sidx, lidx, i_off;
    i  = threadIdx.x + blockIdx.x*blockDim.x*COARSEN_X;
    j  = threadIdx.y + blockIdx.y*blockDim.y;
    k  = threadIdx.z + blockIdx.z*blockDim.z;
#pragma unroll
    for (lidx=0; lidx<COARSEN_X; lidx++) {
        i_off = lidx*blockDim.x;
        idx = i+i_off + j*NX + k*NX*NY;
        sidx = threadIdx.x+i_off
             + threadIdx.y*blockDim.x*COARSEN_X
             + threadIdx.z*blockDim.x*COARSEN_X*blockDim.y;
        if (check_domain_border_3d(i+i_off, j, k, kstart, kend))
            smem[sidx] = d_in[idx];
    }
    this_thread_block().sync();
#pragma unroll
    for (lidx=0; lidx<COARSEN_X; lidx++) {
        i_off = lidx*blockDim.x;
        idx = i+i_off + j*NX + k*NX*NY;
        sidx = threadIdx.x+i_off
             + threadIdx.y*blockDim.x*COARSEN_X
             + threadIdx.z*blockDim.x*COARSEN_X*blockDim.y;
        if (check_stencil_border_3d(i+i_off, j, k, kstart, kend))
            smem_coarseneded_stencil(d_in, d_out, smem, lidx, idx, sidx);
    }
}
```

Code listing 3.9: The smem_coarsened stencil

```
__device__ __inline__ void smem_coarseneded_stencil(
        float *in, float *out, float *smem,
        unsigned int lidx,
        unsigned int idx,
        unsigned int sidx)
{
    float sum = 0.0f;
    if (lidx>0)           accumulate_smem_i_prev(&u, smem, sidx);
    else                  accumulate_hybrid_i_prev(&u, smem, in, sidx, idx);
    if (lidx+1<COARSEN_X) accumulate_smem_i_next(&u, smem, sidx);
    else                  accumulate_hybrid_i_next(&u, smem, in, sidx, idx);
    accumulate_hybrid_j_prev(&sum, smem, in, sidx, idx);
    accumulate_hybrid_j_next(&sum, smem, in, sidx, idx);
#if DIMENSIONS>2
    accumulate_hybrid_k_prev(&sum, smem, in, sidx, idx);
    accumulate_hybrid_k_next(&sum, smem, in, sidx, idx);
#endif
    out[idx] = sum / STENCIL_COEFF - smem[sidx];
}
```

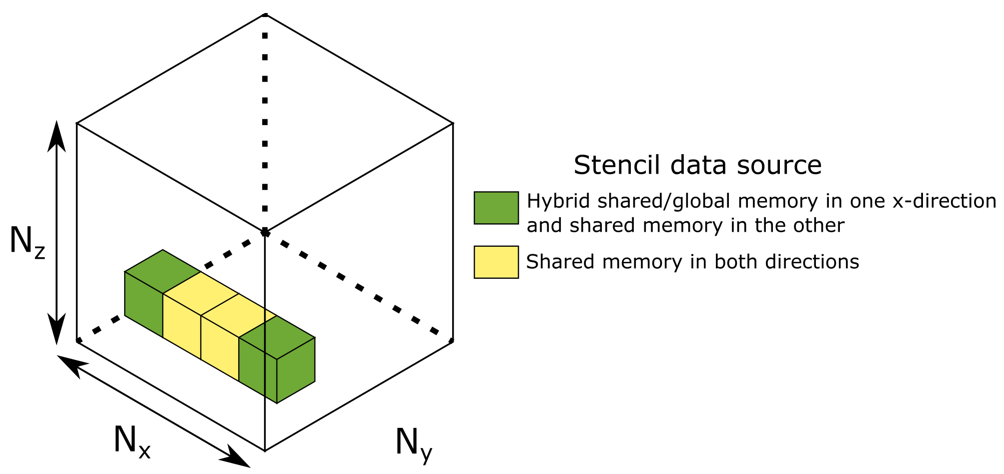

Stencil data source
🟩 Hybrid shared/global memory in one x-direction and shared memory in the other
🟨 Shared memory in both directions
N_z
N_x
N_y

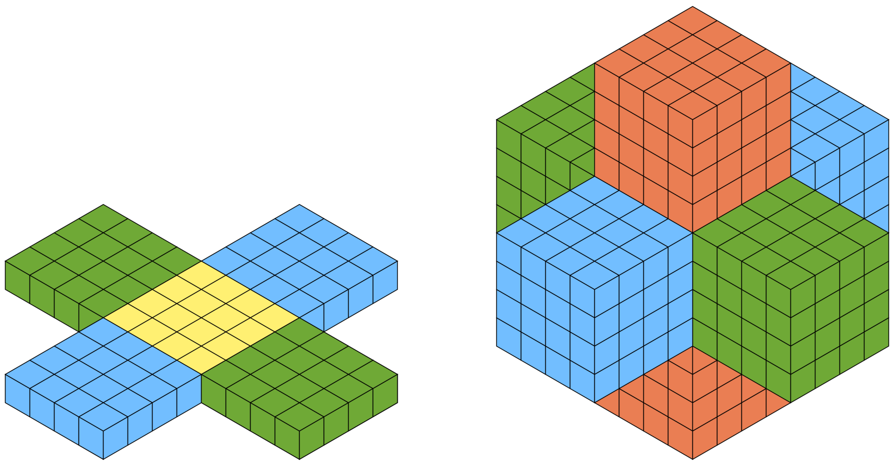
Figure 3.3: Thread coarsening increases the shared memory size, enabling stencil computations in some instances to reuse all neighboring values in the X-direction directly from shared memory without using the hybrid accumulation function.

3.2.3 The Padded Shared Memory Kernel

The `smem_padded` kernel shown in Code listing 3.10 extends the previously proposed `smem` kernel, differentiating itself by increasing the size of the shared memory region. The kernel adds a padding of elements around the shared memory region. Figures 3.4a and 3.4b visualizes this padding in 2D and 3D, respectively, where the stencil radius decides the padding size.

The kernel consists of two phases: the prefetching phase and the stencil computation stage. The main difference in this approach versus the `smem` kernel lies in these phases. The kernel requires every value within a stencil radius' offset outside the TBs' borders for the second phase to prevent global loads in the stencil calculation. Therefore, the threads that reside close to the TBs borders fetch an extra element per border. However, to execute this technique correctly, the TB dimensions must be larger than the stencil radius ($B_x, B_y, B_z \geq RADIUS$). Code listings 3.12 and 3.13 present the source code for this behavior in 3D, and Figure 3.5 visualizes this phase by showing a simplified example where the TB size is 4x4 in 2D. The procedure maps similarly to 3D but is harder to visualize. The yellow squares indicate the shared memory region directly mapped to the threads' indices, while the green and blue areas show the padding in the X- and Y-dimension, respectively. The figure presents the elements that each thread fetches into shared memory. In this simplified example, each thread fetches four elements, but the workload per thread declines with larger domains. With a larger 2D domain, most threads near the borders fetch two elements, and near the corners fetch three elements.

Finally, after the synchronization, the last phase has been dramatically simplified. The final stage fetches the neighboring elements from shared memory only and calculates the stencil.



(a) 2D **(b)** 3D

Figure 3.4: The figures show the shared memory layout for the `smem_padded` kernel for 2D and 3D. The regions include paddings with the same number of elements as the stencil radius in each dimension. The padding allows the kernel to apply the stencil using values from shared memory only.

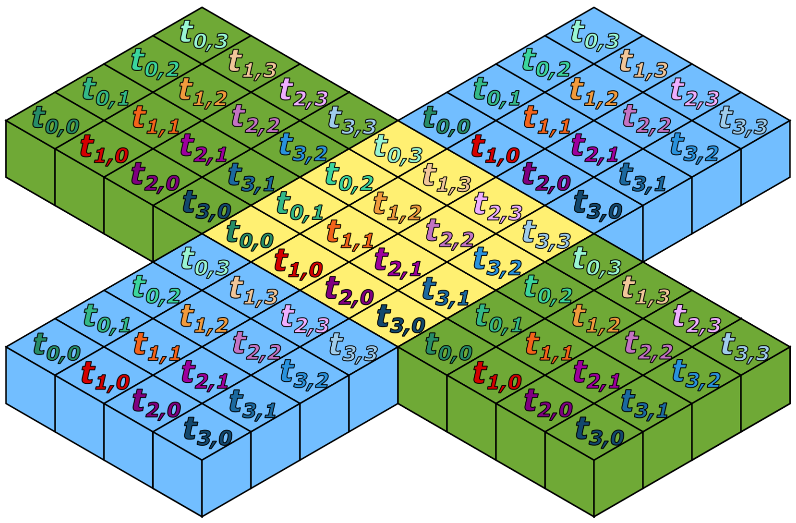

$t_{0,3}$ $t_{0,2}$ $t_{1,3}$ $t_{0,1}$ $t_{1,2}$ $t_{2,3}$ $t_{0,0}$ $t_{1,1}$ $t_{2,2}$ $t_{3,3}$ $t_{1,0}$ $t_{2,1}$ $t_{3,2}$ $t_{2,0}$ $t_{3,1}$ $t_{3,0}$
$t_{0,3}$ $t_{0,2}$ $t_{1,3}$ $t_{0,1}$ $t_{1,2}$ $t_{2,3}$ $t_{0,0}$ $t_{1,1}$ $t_{2,2}$ $t_{3,3}$ $t_{1,0}$ $t_{2,1}$ $t_{3,2}$ $t_{2,0}$ $t_{3,1}$ $t_{3,0}$
$t_{0,3}$ $t_{0,2}$ $t_{1,3}$ $t_{0,1}$ $t_{1,2}$ $t_{2,3}$ $t_{0,0}$ $t_{1,1}$ $t_{2,2}$ $t_{3,3}$ $t_{1,0}$ $t_{2,1}$ $t_{3,2}$ $t_{2,0}$ $t_{3,1}$ $t_{3,0}$
$t_{0,3}$ $t_{0,2}$ $t_{1,3}$ $t_{0,1}$ $t_{1,2}$ $t_{2,3}$ $t_{0,0}$ $t_{1,1}$ $t_{2,2}$ $t_{3,3}$ $t_{1,0}$ $t_{2,1}$ $t_{3,2}$ $t_{2,0}$ $t_{3,1}$ $t_{3,0}$
$t_{0,3}$ $t_{0,2}$ $t_{1,3}$ $t_{0,1}$ $t_{1,2}$ $t_{2,3}$ $t_{0,0}$ $t_{1,1}$ $t_{2,2}$ $t_{3,3}$ $t_{1,0}$ $t_{2,1}$ $t_{3,2}$ $t_{2,0}$ $t_{3,1}$ $t_{3,0}$

Figure 3.5: The figure presents a simplified example that visualizes `smem_padded`’s prefetch stage by showing the elements each thread fetches into shared memory by arranging the threads by their X- and Y-index within the TB.

Code listing 3.10: The smem_padded kernel

```
__global__ void smem_padded_3d(
    float* __restrict__ d_in,
    float* __restrict__ d_out,
    unsigned int kstart,
    unsigned int kend)
{
    extern __shared__ float smem[];
    unsigned int i, j, k, idx, sidx, smem_p_x, smem_p_y;
    i  = threadIdx.x + blockIdx.x*blockDim.x*COARSEN_X;
    j  = threadIdx.y + blockIdx.y*blockDim.y;
    k  = threadIdx.z + blockIdx.z*blockDim.z;
    idx = i + j*NX + k*NX*NY;
    smem_p_x = blockDim.x*COARSEN_X+2*RADIUS;
    smem_p_y = blockDim.y+2*RADIUS;
    sidx = (threadIdx.x + RADIUS)
         + (threadIdx.y + RADIUS)*smem_p_x
         + (threadIdx.z + RADIUS)*smem_p_x*smem_p_y;
    if (check_domain_border_3d(i, j, k, kstart, kend))
        prefetch_3d(smem, d_in, i, j, k, 0, idx, sidx, kstart, kend);
    this_thread_block().sync();
    if (check_stencil_border_3d(i, j, k, kstart, kend))
        smem_padded_stencil(smem, d_out, idx, sidx);
}
```

Code listing 3.11: The smem_padded stencil

```
__device__ __inline__ void smem_padded_stencil(
    float *smem,
    float *out,
    unsigned int idx,
    unsigned int sidx)
{
    float sum = 0.0f;
    accumulate_smem_i_prev(&sum, smem, sidx);
    accumulate_smem_i_next(&sum, smem, sidx);
    accumulate_smem_j_prev(&sum, smem, sidx);
    accumulate_smem_j_next(&sum, smem, sidx);
#if DIMENSIONS>2
    accumulate_smem_k_prev(&sum, smem, sidx);
    accumulate_smem_k_next(&sum, smem, sidx);
#endif
    out[idx] = sum / STENCIL_COEFF - smem[sidx];
}
```

Code listing 3.12: Prefetching values into shared memory for smem_padded

```
__device__ void prefetch_3d(
    float *smem, float *d_in,
    unsigned int i, unsigned int j, unsigned int k,
    unsigned int lidx, unsigned int idx, unsigned int sidx,
    unsigned int kstart, unsigned int kend)
{
    if(lidx==0)           prefetch_i_prev(smem, d_in, sidx, idx, i);
    if(lidx==COARSEN_X-1) prefetch_i_next(smem, d_in, sidx, idx, i);
    prefetch_j_prev(smem, d_in, sidx, idx, j, 0);
    prefetch_j_next(smem, d_in, sidx, idx, j, NY-1);
    prefetch_k_prev(smem, d_in, sidx, idx, k, kstart);
    prefetch_k_next(smem, d_in, sidx, idx, k, kend);
    smem[sidx] = d_in[idx];
}
```

Code listing 3.13: Prefetch functions

```
__device__ void prefetch_prev(
    float *smem,
    float *d_in,
    unsigned int sidx,
    unsigned int idx,
    unsigned int thread_id,
    unsigned int domain_idx,
    unsigned int domain_idx_limit,
    unsigned int soffset,
    unsigned int offset)
{
    if (thread_id < RADIUS && domain_idx >= domain_idx_limit)
    {
        smem[sidx-RADIUS*soffset] = d_in[idx-RADIUS*offset];
    }
}

__device__ void prefetch_next(
    float *smem,
    float *d_in,
    unsigned int sidx,
    unsigned int idx,
    unsigned int thread_id,
    unsigned int thread_id_limit,
    unsigned int domain_idx,
    unsigned int domain_idx_limit,
    unsigned int soffset,
    unsigned int offset)
{
    if (thread_id >= thread_id_limit && domain_idx <= domain_idx_limit)
    {
        smem[sidx+RADIUS*soffset] = d_in[idx+RADIUS*offset];
    }
}
```

3.2.4 Coarsening the Padded Shared Memory Kernel

The `smem_padded` kernel shown in Code listing 3.14 combines the `smem_coarsened` and `smem_padded` features into a single kernel. This kernel has the most significant shared memory region by increasing its size and padding the allocation with extra elements to handle the stencil radius. Figures 3.6a and 3.6b visualizes the shared memory allocations for $CF = 2$ in 2D and 3D, respectively. This kernel's substantial shared memory size enables the most considerable potential for shared memory reuse for stencil calculations. The prefetching and stencil calculation stages utilize a combination of the methods described for `smem_coarsened` and `smem_padded`.

Code listing 3.14: The smem_padded_coarsened kernel

```
__global__ void smem_padded_coarsened_3d(
    float* __restrict__ d_in,
    float* __restrict__ d_out,
    unsigned int kstart,
    unsigned int kend)
{
    extern __shared__ float smem[];
    unsigned int i, j, k, si, sj, sk, i_off, si_off, lidx, idx, sidx,
                 smem_p_x, smem_p_y;
    i  = threadIdx.x + blockIdx.x*blockDim.x*COARSEN_X;
    j  = threadIdx.y + blockIdx.y*blockDim.y;
    k  = threadIdx.z + blockIdx.z*blockDim.z;
    si = threadIdx.x + RADIUS;
    sj = threadIdx.y + RADIUS;
    sk = threadIdx.z + RADIUS;
    smem_p_x = blockDim.x*COARSEN_X+2*RADIUS;
    smem_p_y = blockDim.y+2*RADIUS;
#pragma unroll
    for (lidx=0; lidx<COARSEN_X; lidx++) {
        i_off  = i+lidx*blockDim.x;
        si_off = si+lidx*blockDim.x;
        idx    = i_off+j*NX+k*NX*NY;
        sidx   = si_off + sj*smem_p_x + sk*smem_p_x*smem_p_y;
        if (check_domain_border_3d(i_off, j, k, kstart, kend))
            prefetch_3d(smem, d_in, i_off, j, k, lidx, idx, sidx, kstart, kend);
    }
    this_thread_block().sync();
#pragma unroll
    for (lidx=0; lidx<COARSEN_X; lidx++) {
        i_off  = i+lidx*blockDim.x;
        idx    = i_off+j*NX+k*NX*NY;
        si_off = si+lidx*blockDim.x;
        sidx   = si_off + sj*smem_p_x + sk*smem_p_x*smem_p_y;
        if (check_stencil_border_3d(i_off, j, k, kstart, kend))
            smem_padded_stencil(smem, d_out, idx, sidx);
    }
}
```

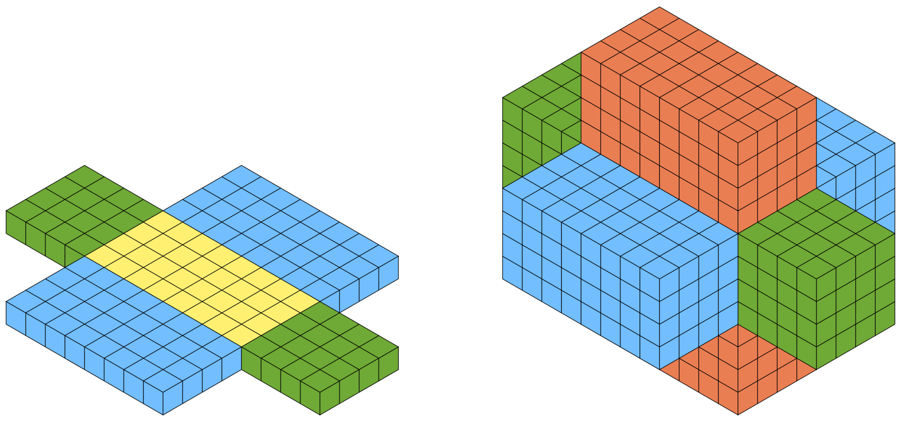

(a) 2D **(b)** 3D

Figure 3.6: The figures show the shared memory layout for the `smem_padded_coarsened` kernel for 2D and 3D for $CF = 2$. Thread coarsening increases the shared region's X-dimension significantly. Similar to Figure 3.4, the regions include paddings with the same number of elements as the stencil radius in each dimension. The padding allows the kernel to apply the stencil using values from shared memory only.

3.3 Launching ISL Kernels

3.3.1 Allocating Resources

Creating a high-performance ISL application supporting more than a single GPU requires defining a single CUDA stream per GPU context. The streams enable asynchronous memory transfers between the CPU and GPU and between the GPUs. The memory transfers would otherwise be serialized, incurring significant performance penalties. To allocate streams for a specific GPU, the host must explicitly switch GPU contexts. The `cudaSetDevice` CUDA API [66, p. 45] switches GPU context before invoking GPU-specific commands.

Pinning (page-locking) the pageable host memory ensures adequate performance for an arbitrary number of GPUs in our application. Pinning ensures excellent performance for asynchronous memory transfers between the host and device by preventing the OS from moving the allocated data [13, p. 148]. However, pinning can impact the performance of other processes on the same system by leaving less pageable memory available for other processes.

Before initializing any kernels, we allocate the input and output buffers in the GPUs' memory space. Independent context switches make spawning a new host thread per GPU advantageous. This process removes the overhead of single-threaded context switches and kernel launches [62]. Our implementation utilizes shared memory multi-threading on the host through the OpenMP API, and Code listing 3.15 shows the process of allocating the buffers on multiple GPUs.

Allocating the domain buffers on the devices enable memory transfers between the host and devices. Code listing 3.16 shows the memory transfers before (Lines 4 to 8) and after (Lines 26 to 30) dispatching the kernels. Notice that we do not include the memory transfers when timing the kernels, as Lines 10 to 14 and Lines 19 to 24 surround the kernel launches only. Finally, Line 32 synchronizes the execution before proceeding to free the allocated resources.

Code listing 3.15: Allocating GPU memory in parallel

```
unsigned long size = BYTES_PER_GPU;
if (NGPUS>1) size += 2*GHOST_ZONE_BYTES;
#pragma omp parallel for num_threads(NGPUS)
for (int i = 0; i < NGPUS; i++) {
    cudaSetDevice(i);
    CU(cudaMalloc((void **)&d_in[i], size));
    CU(cudaMalloc((void **)&d_out[i], size));
}
```

Code listing 3.16: Host multi-threading for handling memory transfers between host and GPUs.

```
1   int offset;
2   if (NGPUS==1) offset=0;
3   else          offset=GHOST_ZONE;
4   #pragma omp parallel for num_threads(NGPUS)
5   for (int i = 0; i < NGPUS; i++) {
6       cudaSetDevice(i);
7       CU(cudaMemcpyAsync(&d_in[i][offset], &d_ref[i * OFFSET], BYTES_PER_GPU,
         cudaMemcpyHostToDevice, streams[i]));
8   }
9
10  cudaSetDevice(0);
11  cudaEvent_t start, stop;
12  CU(cudaEventCreate(&start));
13  CU(cudaEventCreate(&stop));
14  CU(cudaEventRecord(start));
15
16  if(NGPUS==1) dispatch_kernels(d_in[0], d_out[0]);
17  else         dispatch_multi_gpu_kernels(d_in, d_out, streams);
18
19  cudaSetDevice(0);
20  cudaEventRecord(stop);
21  cudaEventSynchronize(stop);
22  cudaEventElapsedTime(&milli, start, stop);
23  cudaEventDestroy(start);
24  cudaEventDestroy(stop);
25
26  #pragma omp parallel for num_threads(NGPUS)
27  for (int i = 0; i < NGPUS; i++) {
28      cudaSetDevice(i);
29      CU(cudaMemcpyAsync(&d_ref[i * OFFSET], &d_in[i][offset], BYTES_PER_GPU,
         cudaMemcpyDeviceToHost, streams[i]));
30  }
31
32  for (int s=0; s<NGPUS; s++) CU(cudaStreamSynchronize(streams[s]));
```

3.3.2 Kernel Configuration

Shared Memory

The amount of shared memory available for each CTA is crucial for optimizing occupancy and ensuring correct execution behavior for our proposed shared memory kernels. The shared memory amount utilized within the kernels is reconfigurable through the `cudaFuncAttributeMaxDynamicSharedMemorySize` attribute, configurable by the `cudaFuncSetAttribute` CUDA API [66, p. 103]. Allocating more than 48 KiB shared memory per CTA on the Volta GPUs requires dynamic shared memory allocations. Code listing 3.17 presents the source code for maximizing shared memory per CTA.

Configuring the amount of shared memory available per SM impacts the ideal number of threads to include in each TB. Naively setting the amount to the closest value of each TB prevents the SMs from issuing multiple TBs in parallel. We maximize the amount available for all kernels utilizing shared memory as we believe our shared memory kernels benefit more from extra shared memory than L1 cache space. The `smem` kernel could potentially benefit from a more even ratio of shared memory and L1 cache, as the kernel calculates the stencil from shared and global memory. However, introducing different shared memory capacities per SM would make the analysis harder, and therefore, we avoid this intricacy.

Thread Block Dimensions

Choosing a default configuration is essential for later evaluating the impact of autotuning TB dimensions. The default configuration selects the TB dimensions using a heuristic approach. Predetermining the TB dimensions is a simple approach and gives satisfactory performance for many configurations in 2D. However, the approach imposes problems for performance and ensuring correctness in 3D as `smem_padded` requires $B_x, B_y, B_z \geq RADIUS$, and the shared memory allocations grow more rapidly than in 2D, motivating the need for a more sophisticated approach.

The `cudaOccupancyMaxPotentialBlockSizeVariableSMem` CUDA API [66, p. 443] returns the ideal number of threads per TB for maximizing occupancy given the register and shared memory usage of a kernel. For 2D, we naively set $B_x = 32$ and divide the total thread count by this value. For 3D, the process is more intricate, and only some configurations ensure good performance. We observed that the kernels with $B_y, B_z \in [2, 8]$ generally performed well. Therefore, Code listing 3.18 proposes the following heuristic approach to distribute the number of threads between the dimensions in 3D. Ideally, we try to keep the value $B_x = 32$, but Lines 6 and 7 halve the value for `smem_padded` should the $B_x, B_y, B_z \geq RADIUS$ restriction enforce it. Line 10 sorts the dimensions in ascending order, setting B_x as the highest value to maximize memory coalescing should B_x decrease below B_y or B_z.

Code listing 3.17: Maximizing dynamic shared memory allocation size for the Volta and Pascal architectures

```
void set_max_dynamic_shared_memory_size() {
    if (SMEM) {
        const char* arch = STR(ARCH);
        if (strcmp(arch, "volta")==0)
            cudaFuncSetAttribute(
                get_kernel(), cudaFuncAttributeMaxDynamicSharedMemorySize, 98304);
        else if (strcmp(arch, "pascal")==0)
            cudaFuncSetAttribute(
                get_kernel(), cudaFuncAttributeMaxDynamicSharedMemorySize, 49152);
    }
}
```

Code listing 3.18: Heuristically distributing the number of threads between the TB dimensions to maximize occupancy

```
    __host__ __device__ void set_max_occupancy_block_dimensions(
        int *bx, int *by, int *bz, int threads)
    {
    if (DIMENSIONS==3) {
        int b0 = 32;
        while (SMEM && PADDED && threads / (b0*RADIUS*RADIUS) == 0 && b0 > 1)
            b0 = b0/2;
        int b1 = MIN(MAX(2, RADIUS), 8);
        int b2 = threads/(b0*b1);
        sort3_desc(&b0, &b1, &b2);
        *bx = b0, *by = b1, *bz = b2;
    } else
        *bx = 32, *by = threads/32, *bz=1;
}
```

Code listing 3.19: Configuring shared memory allocation size

```
void set_smem(
    unsigned int *smem,
    unsigned int bx,
    unsigned int by,
    unsigned int bz)
{
    if (!SMEM) {*smem = 0; return;}
    unsigned int smem_x   = bx*COARSEN_X;
    unsigned int smem_p_x = smem_x + 2*RADIUS;
    unsigned int smem_p_y = by + 2*RADIUS;
    unsigned int smem_p_z = bz + 2*RADIUS;
    if (DIMENSIONS == 3) {
        if (PADDED)         *smem = smem_p_x*smem_p_y*smem_p_z*sizeof(float);
        else                *smem = smem_x*by*bz*sizeof(float);
    } else {
        if (PADDED)         *smem = smem_p_x*smem_p_y*sizeof(float);
        else                *smem = smem_x*by*sizeof(float);
    }
}
```

3.3.3 Launching Single-GPU Kernels

Dispatching CUDA kernels involves the two preceding steps: shared memory and TB dimension configuration. Code listing 3.20 presents the source code for launching kernels, which performs both steps and launches the kernels for a certain number of iterations. All threads within the execution grid implicitly synchronize when exiting the kernels, ensuring that every TB has finished executing the iteration step before proceeding onto the next iteration. Line 2 defines a utility function needed by `cudaOccupancyMaxPotentialBlockSizeVariableSMem` for calculating each kernel's shared memory usage, and Code listing 3.21 presents its implementation, which uses the functions in Code listings 3.18 and 3.19. Line 12 exits the application if any block dimension is smaller than the stencil radius for the `smem_padded` kernel. Furthermore, Line 14 ensures that fewer TBs execute when applying thread coarsening to the kernels.

Code listing 3.20: Launching single-GPU kernels

```
1  void dispatch_kernels(float *d_in, float *d_out) {
2      calculate_smem calc_smem;
3      int g, b, bx, by, bz;
4      unsigned int smem;
5      if (SMEM) set_max_dynamic_shared_memory_size();
6      if (HEURISTIC)
7          cudaOccupancyMaxPotentialBlockSizeVariableSMem(
8              &g, &b, get_kernel(), calc_smem
9          );
10     set_block_dims(&bx, &by, &bz, b);
11     print_program_info(bx, by, bz);
12     check_early_exit(bx, by, bz);
13     dim3 block(bx, by, bz);
14     dim3 grid((1+(NX-1)/bx)/COARSEN_X);
15     if (DIMENSIONS>1) grid.y = 1+(NY-1)/by;
16     if (DIMENSIONS>2) grid.z = 1+(NZ-1)/bz;
17     float *d_tmp;
18     set_smem(&smem, bx, by, bz);
19     for (int i=0; i<ITERATIONS; i++) {
20         get_kernel()<<<grid, block, smem>>>(d_in, d_out, 0, NZ-1);
21         getLastCudaError("kernel execution failed\n");
22         d_tmp = d_in; d_in = d_out; d_out = d_tmp; // swap input and output buffers
23     }
24 }
```

Code listing 3.21: Utility function for configuring shared memory allocation size

```
struct calculate_smem: std::unary_function<int, int> {
    __host__ __device__ int operator()(int threads) const {
        if (!SMEM) return 0;
        int bx, by, bz;
        set_max_occupancy_block_dimensions(&bx, &by, &bz, threads);
        unsigned int smem;
        set_smem(&smem, bx, by, bz);
        return smem;
    }
};
```

3.3.4 Launching Multi-GPU Kernels

Dispatching multi-GPU kernels is a more sophisticated procedure than for single-GPU kernels. Code listing 3.23 presents the approach for launching 2D and 3D kernels on multiple GPUs, where the main difference from single-GPU lies within the iterative loop of Lines 23 to 40. The procedure requires switching GPU contexts before launching a kernel onto a specific GPU. Furthermore, each iteration requires communication between the GPUs for transferring ghost zone elements.

Lines 24 and 25 present the communication pattern for strip partitioning where each GPU communicates their borders onto their neighbors' ghost zone and vice versa. Code listing 3.22 presents the communication source code, utilizing indices for the internal region and ghost zone. The `cudaMemcpyPeerAsync` CUDA API [66, p. 179] handles P2P communication between the GPUs. The function operates asynchronously with respect to the host and the other streams running on the same GPU, enabling simultaneous transfers between all participants. Finally, Line 39 synchronizes the GPUs before performing the next iteration using the `cudaStreamSynchronize` CUDA API [66, p. 75].

Code listing 3.22: Transferring ghost zone elements between neighboring GPUs

```
void send_upper_ghost_zone(
    float **d_u1, unsigned int device, cudaStream_t* streams)
{
    CU(cudaMemcpyPeerAsync(
        d_u1[device+1],
        device+1,
        d_u1[device] + (INTERNAL_END-GHOST_ZONE_DEPTH) * BORDER_SIZE,
        dev,
        GHOST_ZONE_BYTES,
        streams[device]));
}

void send_lower_ghost_zone(
    float **d_u1, unsigned int device, cudaStream_t* streams)
{
    CU(cudaMemcpyPeerAsync(
        d_u1[device-1] + INTERNAL_END * BORDER_SIZE,
        device-1,
        d_u1[device] + INTERNAL_START * BORDER_SIZE,
        device,
        GHOST_ZONE_BYTES,
        streams[device]));
}
```

Code listing 3.23: Launching multi-GPU kernels

```
void dispatch_multi_gpu_kernels(
    float **d_u1, float **d_u2, cudaStream_t *streams)
{
    calculate_smem calc_smem;
    float **d_tmp;
    int g, b, bx, by, bz, s;
    unsigned int i, kstart, kend, smem;
    if (SMEM) set_max_dynamic_shared_memory_size();
    if (HEURISTIC)
        cudaOccupancyMaxPotentialBlockSizeVariableSMem(
            &g, &b, get_kernel(), calc_smem
        );
    set_block_dims(&bx, &by, &bz, b);
    print_program_info(bx, by, bz);
    dim3 block(bx, by, bz);
    dim3 grid((1+(NX-1)/bx)/COARSEN_X);
    if (DIMENSIONS==2) grid.y = 1+(NY/NGPUS+2*GHOST_ZONE_DEPTH-1)/by;
    else if (DIMENSIONS==3) {
        grid.y = 1+(NY-1)/by;
        grid.z = 1+(NZ/NGPUS+2*GHOST_ZONE_DEPTH-1)/bz;
    }
    set_smem(&smem, bx, by, bz);
    for (i=0; i<ITERATIONS; i++) {
        for (s=0; s<NGPUS-1; s++) send_upper_ghost_zone(d_u1, s, streams);
        for (s=1; s<NGPUS; s++)   send_lower_ghost_zone(d_u1, s, streams);
        for (s=0; s<NGPUS; s++)   CU(cudaStreamSynchronize(streams[s]));
        for (s=0; s<NGPUS; s++) {
            CU(cudaSetDevice(s));
            kstart = 0;
            kend   = INTERNAL_END-1+GHOST_ZONE_DEPTH;
            if      (s==0)       kstart = INTERNAL_START;
            else if (s==NGPUS-1) kend   = INTERNAL_END-1;
            get_kernel()<<<grid, block, smem, streams[s]>>>(
                d_u1[s], d_u2[s], kstart, kend
            );
            getLastCudaError("kernel execution failed\n");
        }
        d_tmp = d_u1; d_u1 = d_u2; d_u2 = d_tmp; // swap d_u1 and d_u2
        for (s=0; s<NGPUS; s++) CU(cudaStreamSynchronize(streams[s]));
    }
}
```

Chapter 4

Experimental Setup

The following chapter presents our approach for generating the results discussed in the next chapter. Furthermore, we discuss the physical hardware and technical decisions taken throughout the project to ensure the reader understands our approach for generating the measurements.

4.1 Hardware Setup

We conducted the experiments on two different systems, a DGX-2 [67] and a node within the Idun cluster [68] at NTNU. Table 4.1 presents the specifications for both systems and provides a side-by-side comparison. The DGX-2 was the primary system where we gathered measurements for all optimizations on the Volta architecture, while the Idun cluster provided measurements for the shared memory and thread coarsening optimizations on the Pascal architecture. Table 4.2 compares the Volta and Pascal GPUs used in our experiments.

The DGX-2 utilizes the modern NVSwitch interconnect topology, capable of handling 16 GPUs executing simultaneously. The interconnect is essentially a fully connected non-blocking crossbar, enabling all simultaneous communication between the devices at full NVLink bandwidth. The system connects two baseboards, each supporting 8 GPUs. The baseboards' GPUs can communicate with one another at full 300 GB/s GPU-to-GPU bandwidth. Furthermore, each GPU

Table 4.1: DGX-2 vs. Idun cluster node. Based on [67, 68].

	DGX-2	Idun cluster node
GPUs	16x NVIDIA Tesla V100	2x NVIDIA Tesla P100
CPU	Dual Intel Xeon Platinum 8168	2x Intel Xeon E5-2650 v4
	2.7 GHz, 24-cores	2.2 GHz, 24-cores
CUDA	CUDA 11.1	CUDA 10.1
Software	NVIDIA DGX Server 4.8.0	CentOS Linux 8 (Core)
Interconnect	NVSwitch	PCIe

Table 4.2: Comparing the Tesla V100 and Tesla P100 GPUs. Based on [35, 42].

	Tesla V100	Tesla P100
No. SMs	80	56
FP32 Cores per SM	64	64
FP32 Cores per GPU	5120	3584
Peak FP32 TFLOPS	15.7	10.6
GPU Memory Size	16 GiB	16 GiB
Unified Cache	Shared Memory/L1/Texture	L1/Texture
Unified Cache Size	128 KiB	24 KiB
L2 Cache Size	6 MiB	4 MiB
Shared Memory Size per CTA	Up to 96 KiB	48 KiB
Shared Memory Size per SM	Up to 96 KiB	64 KiB
Register File Size per SM	256 KiB	256 KiB
Register File Size per GPU	20480 KiB	14336 KiB
Memory system	HBM2	HBM2

can communicate with any GPU on the opposite baseboard, although at half the bandwidth as the communication requires two NVSwitch traversals [69].

Various challenges appeared with each system as they were different. The DGX-2 system restricted CUDA applications to NVIDIA Docker, contrary to Idun, where we had direct access to the NVIDIA CUDA Compiler. Additionally, Idun employed Slurm [70], a job scheduling manager for collaboratory executing jobs. Meanwhile, the collaborators of the DGX-2 did not use Slurm and instead relied on mutual trust in not interfering with each other's scripts. Combined with the fact that all scripts shared the same CPU, these deficiencies impose potentially unwanted contention causing slowdowns when gathering results. The CPU contention partly motivated our decision to time the program's part where the GPU is active only. As a response, the following section provides a density plot to legitimize our results.

4.2 Measurements

4.2.1 Kernel Timing

To evaluate the performance of the proposed kernel optimizations, multiple possibilities for timing the kernels appear. Evaluating the whole applications' performance from end to end seems like the natural choice but does not provide sufficient insights into the application's behavior. For example, our previous work [1] did not increase performance for the simple stencil application when applying multiple GPUs, even with a substantial domain size. Furthermore, our previous work failed to investigate the reasons behind this deficiency.

However, in this thesis's early stage, the error soon became evident: creating and destroying multiple streams and execution contexts for each device composes substantial startup and shutdown overheads for multi-GPU applications. Nevertheless, these overheads only have to be counted once in real-world applications, as the applications can idle while waiting for kernels to execute on-demand. Furthermore, memory transfers occupy a significant part of the application. Meanwhile, most of our optimizations target kernels instead of memory transfers, making it more convenient to time the kernels only for observing the optimizations' impact. Therefore, we choose to time the kernels only, leaving the resource allocation and memory transfers outside our measurements.

4.2.2 Metric Collection

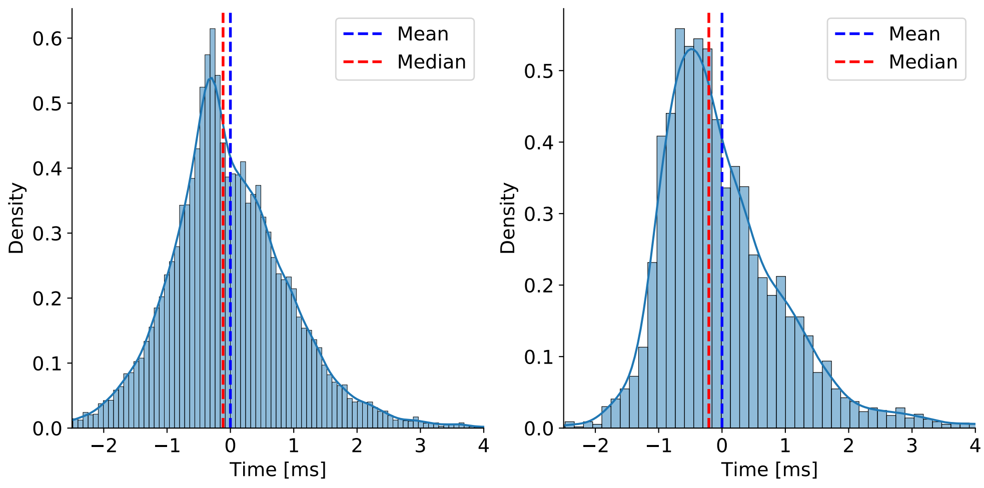
We used the `nvprof` command-line tool for gathering metrics and events for each kernel executed in the application on the Volta architecture. Executing `nvprof --metrics all --events all` generated detailed metrics regarding the kernels' execution behavior. Each ISL iteration incurred a kernel launch outputting varying metrics for each run. `nvprof` outputted the minimum, maximum and average value of the metrics observed for all kernels launched. For events, the output included the total number of times the event occurred. Our analysis extracted the average metric values and total event count to benchmark the kernel's general performance.

We utilized three approaches for analyzing the metrics. Firstly, executing `nvprof --analysis-metrics` enables profiling with NVP. NVP guides the analysis by showing charts with the metrics generated by `nvprof`, giving thorough insights into the application. Secondly, we utilized a Python script to create a table that compares the raw metric data. This approach is much less time-consuming than the first approach for quick comparisons. Finally, when we found a metric of interest, we plotted the metric using the Seaborn [71] statistical data visualization tool. We noticed if the execution time had a similar pattern as the metric to detect if the metric constituted a computational bottleneck.

We did not have permission to gather metrics for our application on the Idun cluster. This deficiency limited our analysis of the Pascal architecture. The Idun cluster also introduced significant contention from other students and employees of NTNU, causing substantial queueing delays. Therefore, we limited our analysis of Pascal to executing the kernels on a single GPU with thread coarsening but omit the time-intensive autotuning. However, executing the kernels with thread coarsening covers essential aspects for comparing Volta and Pascal's caches.

4.2.3 Reporting Measurements

Hoefler and Belli [72] emphasized twelve rules for reporting performance results after investigating 120 papers across three top conferences. One rule stated the importance of emphasizing if the baseline is a serial or parallel process and reporting the baseline's absolute execution performance. We report the `base` kernel using our heuristic approach for choosing TB dimensions and no thread coarsening as the baseline kernel in all measurements. Furthermore, we add the absolute



(a) Single-GPU measurement distribution **(b)** Multi-GPU measurement distribution

Figure 4.1: The graphs display the distribution of our single-GPU and multi-GPU measurements through a z-score normalized density plot, representing the standard score of our measurements. The x-axis represents the number of standard deviations by which the measurements' values differ from the mean of all measurements. The graph differentiates itself from a Gaussian distribution, indicating that we can not assume a normal distribution in our measurements. As a result, the median resides closer to the densest density region than the mean, as visualized by the graph's red and blue bars.

execution performance of our measurements before reporting performance improvements relative to the baseline.

Runtime variance in contemporary supercomputers convolutes performance evaluation [73]. Seemingly identical runs might output noticeably different results, making it harder to determine if minor optimizations indeed boost performance. Therefore, Hoefler and Belli [72] presented another rule highlighting the need for reporting that measurements are either deterministic or nondeterministic. As we gathered the measurements using two of NTNU's supercomputers, the measurements in the following chapter are nondeterministic. We explain our measurements' variance by providing a density plot.

Figure 4.1 visualizes the distribution of our measurements by plotting their density. As the graph differs significantly from a Gaussian distribution, we can not assume normality in our analysis. Therefore, using the mean is suboptimal, as the mean does not represent our measurements' densest part. The mean includes too many outliers, moving its value away from the densest region, while the median provides a value that resides closer to the densest region, improving the quality of our analysis. Therefore, we include the median in our analysis instead of the mean. Moreover, we choose to present the next chapter's measurements in bar plots over box plots as our measurements' variance is slight.

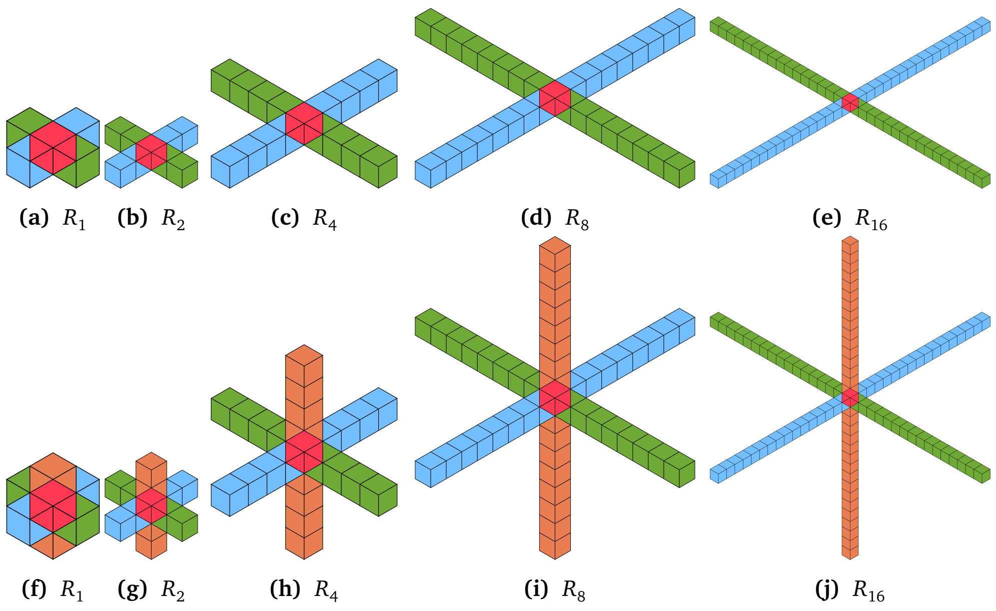


Figure 4.2: The figures show each RTM stencil configurations measured in Chapter 5, denoting each RTM stencil with radius r by R_r. The color characteristics is equal to Figure 2.1.

4.3 Execution Configurations

4.3.1 Stencils

To evaluate the key performance trends of our application, we emphasize five 2D stencils and five 3D stencils. The stencils follow the RTM stencil pattern with different radiuses. Figures 4.2a to 4.2e and Figures 4.2f to 4.2j present the setup covering 2D and 3D stencils, respectively. Technically, R_1 does not relate to the concept of RTM, but we include it as it is a critical stencil for iterative methods. These stencils cover a range of computational demands for the GPU. Additionally, they also cover various amounts of redundant work with differing shared memory allocation sizes and multi-GPU ghost zone sizes. Consequently, we expect the stencil's size to have impacts for both single-GPU and multi-GPU.

The domain size is an essential aspect of the experiments. For simplicity, we keep each dimension equal. To simplify the comparison between 2D and 3D, we choose dimensions that compose equal total domain buffer sizes. The application uses an input and output buffer for computing the stencil, doubling the storage requirement. We wish to evaluate the application with both a small and a large domain. We evaluate a small domain by setting each dimension $NX = NY = 4096$ (2D) and $NX = NY = NZ = 256$ (3D), composing a total device buffer size of 128 MiB. Furthermore, we evaluate a large domain by setting $NX = NY = 32768$ (2D) and $NX = NY = NZ = 1024$ (3D), composing a total buffer size of 8 GiB.

4.3.2 Stencil Configurations

The following subsection explains the configurations for all the optimizations presented in the next chapter. All of the presented optimizations include the median of 30 executions to ensure credible results. Additionally, all measurements run the ISL application with eight stencil iterations unless otherwise specified. Furthermore, all performance improvements are relative to the given dimension's baseline kernel, normalized by the respective stencil radius.

Firstly, we evaluate the `base`, `smem`, and `smem_padded` kernels on Volta utilizing our heuristic approach for choosing TB dimensions. Our evaluation presents side-by-side measurements for 2D and 3D, displaying their absolute performance. Followingly, we augment the analysis for Volta by performing thread coarsening within each kernel and autotuning the TB dimensions to expose sufficient parallelism. We evaluate the optimizations for $CF \in \{1, 2, 4, 8\}$ as further coarsening decreases performance for our setup. Furthermore, we compare the runtime performance with the metric our analysis described as the most likely performance bottleneck. We present the measurements as performance and metric improvements to reveal the relationship between performance and bottleneck metrics.

The autotuning includes all TB dimension permutations, $B_x, B_y, B_z \in \{2^x \mid x \in [0, 10]\}$, with the added restrictions $B_x \geq B_y \geq B_z$ and $B_x * B_y * B_z \in [32, 1024]$ after realizing that most satisfactory configurations followed this pattern. The autotuning framework executed each configuration 30 times for extracting the ideal TB dimensions. It is crucial to note that this set does not cover the whole TB dimension space, potentially leading to sub-optimal TB dimensions. As the Flamingo autotuning framework applies brute-force search in this space, the number of tested configurations grows large. This set took the system several days to run, even when executing the kernels in parallel.

Subsequently, we evaluate the kernels on Pascal using thread coarsening before concluding our analysis for single-GPU optimizations by comparing Pascal's performance to Volta. In this manner, we evaluate Volta's improved caches and gain insights into the implications of combining the L1 and shared memory into a unified cache for our ISL application.

Finally, we amplify the analysis by utilizing multiple GPUs to study our application's scaling capabilities. Our goal is to scale performance linearly with added devices, keeping the domain size fixed to indicate strong scaling. Therefore, we present the performance improvements achieved when offloading computations onto more GPUs using equal domain dimensions as the single-GPU measurements. By evaluating the application's strong scaling capabilities, we can easily compare the results to the single-GPU optimizations.

Chapter 5

Results

The following chapter explores the performance characteristics of our ISL application. We present runtime performance charts and metrics explaining our observations. To simplify our discussions, we introduce abbreviations for domain size, e.g., $D = 8$ GiB, and the number of GPUs utilized, e.g., $G = 16$. Additionally, we denote a kernel's TB dimension strategy in parentheses after the kernel name, e.g., `smem` (heuristic).

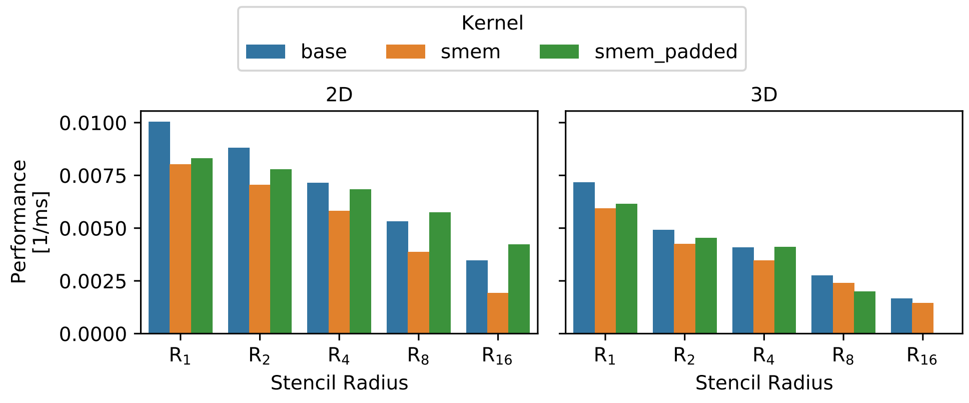
5.1 Optimizing ISL Kernels on the Volta Architecture

5.1.1 Shared Memory

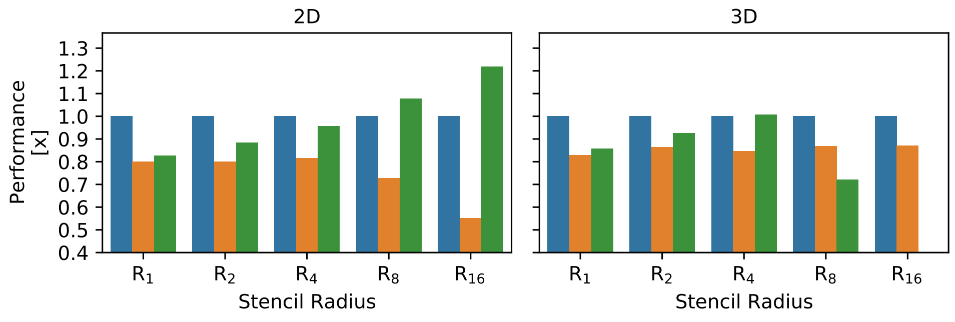
Figure 5.1a presents the measured results from executing the kernels on a single V100 GPU on the DGX-2. Figure 5.1b displays the kernels' performance improvement relative to the given dimension's baseline kernel. Introducing shared memory kernels improves performance over the baseline kernel by up to 1.22x (1.01x) in 2D (3D) on Volta. In 2D, `base` outperforms `smem` and `smem_padded` for R_1, R_2, and R_4. However, `smem_padded` becomes slightly superior for R_8 and is significantly better for R_{16}.

In contrast to 2D, `base`'s performance in 3D is remarkable compared to `smem` and `smem_padded`. Interestingly, `base` strictly dominates the other kernels in 3D in all cases except for R_4. At this point, `smem_padded` matches `base`'s incredible performance. However, `smem_padded`'s performance soon drops as the 3D TB dimensions cause troubles for `smem_padded`'s $B_x, B_y, B_z \geq RADIUS$ restriction. This restriction forces `smem_padded`'s B_x value below 32 for R_8, which causes troubles for coalescing memory accesses. There exists no available TB dimensions adhering to this restriction for R_{16}, causing our application to exit execution.

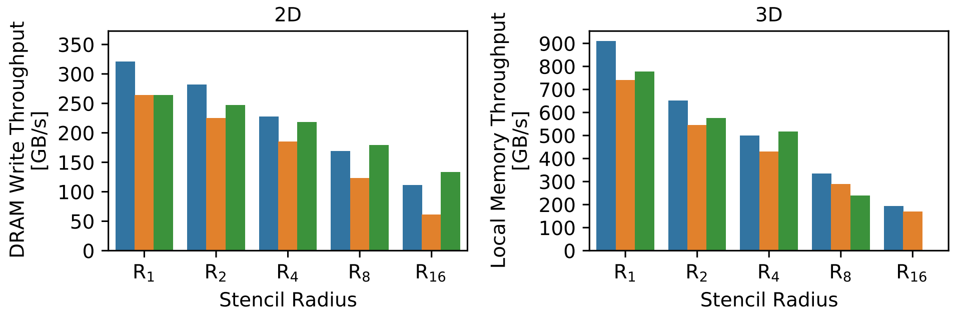
Figure 5.1c displays the kernels' DRAM write throughput and local memory throughput for 2D and 3D, respectively. The metrics closely resemble Figure 5.1a's performance measurements, indicating that DRAM write (local memory) throughput limits our kernels in 2D (3D). This exciting result indicates that optimizing DRAM write throughput is more critical than increasing shared memory reuse in



(a) The kernels' performance

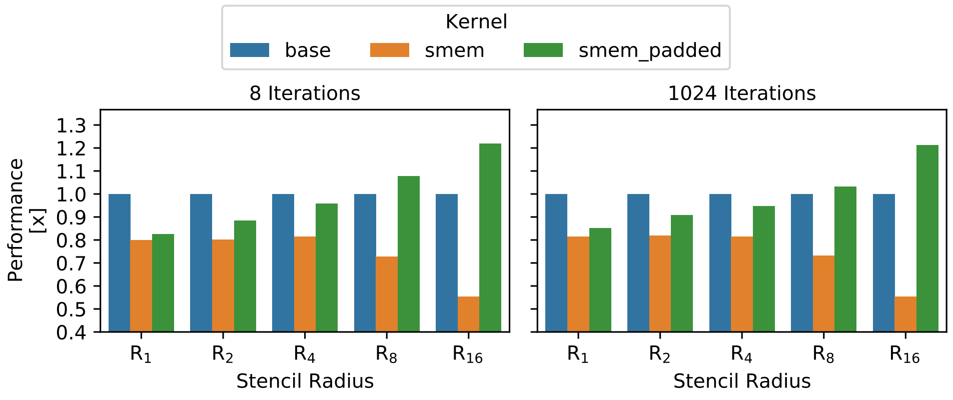


(b) Comparing each kernel's performance improvement relative to the given dimension's baseline kernel, normalized by the stencil radius

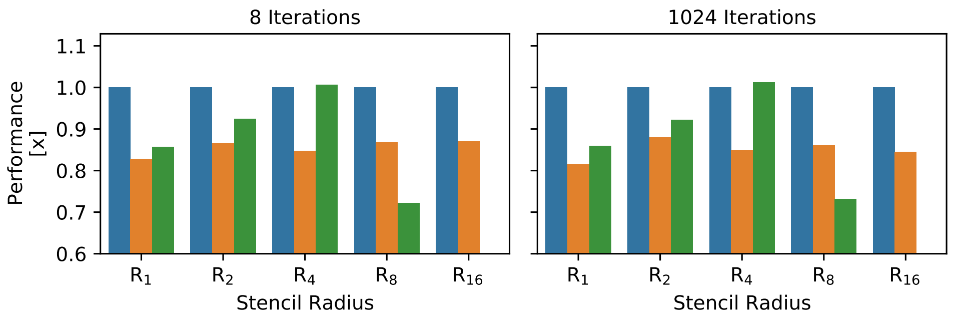


(c) DRAM write throughput and local memory throughput as a function of the stencil radius

Figure 5.1: The figures present the kernels' performance, performance improvement over the given dimension's baseline kernel normalized by the stencil radius, DRAM write throughput, and local memory throughput as a function of stencil radius for the Volta architecture (D = 8 GiB). The kernels utilize heuristic TB dimensions.



(a) 2D



(b) 3D

Figure 5.2: The graphs compare the kernels' performance improvement over the respective baseline kernel normalized by the stencil radius for 8 and 1024 iterations in 2D and 3D. The similar improvements indicate that the number of iterations does not impact the kernels' performance relationships. The kernels utilize heuristic TB dimensions.

2D on the Volta architecture. In contrast to 2D, register spilling strongly limits the 3D kernels. Therefore, optimizing the 3D kernels will depend on register usage over bandwidth utilization.

We include a brief sensitivity analysis showing the kernels' sensitivity to the number of iterations calculated in the ISL application. Figures 5.2a and 5.2b display the kernels' performance improvements over the respective baseline kernel for 8 and 1024 iterations in 2D and 3D, respectively. The columns' similarity indicate that the kernels' performance improvements are independent of the number of stencil iterations used in the computation. As a result, we can justify that sticking to 8 iterations is sufficient to display our ISL application's behavior for single-GPU kernels.

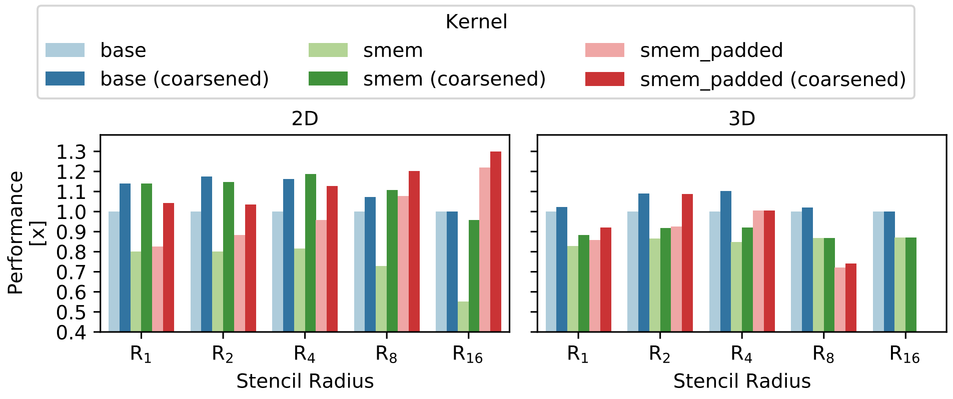


Figure 5.3: The graphs compare the performance improvements of uncoarsened kernels against kernels having the best observed coarsening factor relative to the given dimension's baseline kernel normalized by the stencil radius. The kernels utilize heuristic TB dimensions.

5.1.2 Thread Coarsening

Figure 5.3 presents our results for optimizing the kernels with thread coarsening and selecting the best-performing coarsening factor. The figure compares uncoarsened kernels to coarsened kernels. Applying thread coarsening improves the kernels' performance by up to 1.30x (1.10x) over the baseline kernel in 2D (3D), indicating that coarsening impacts performance on Volta for heuristic TB dimensions.

Figures 5.4 and 5.5 present more detailed graphs for thread coarsening in 2D and 3D, respectively. The graphs display the performance and DRAM write throughput (2D) or local memory throughput (3D) improvements of the coarsened kernels relative to the baseline kernel normalized by the stencil radius. The throughput improvements are exceedingly similar to the performance improvements, indicating that thread coarsening improves performance by exposing more parallel memory operations per thread for 2D kernels and decreasing register usage for 3D kernels.

Thread coarsening has a significant impact on the underperforming `smem` kernel in 2D, making it competitive to the other kernels and outperforming them for R_4. It is tempting to credit the significantly increased shared memory size that comes with coarsening as it enables more reuse. For this example, the shared memory load throughput increases from 3526 GB/s to 5672 GB/s and shared memory store throughput from 187.1 GB/s to 276.9 GB/s. However, since Figure 5.5 shows that DRAM write throughput improvement closely resembles the performance improvement, the critical performance-enhancement factor is exposing more parallel I/O per thread.

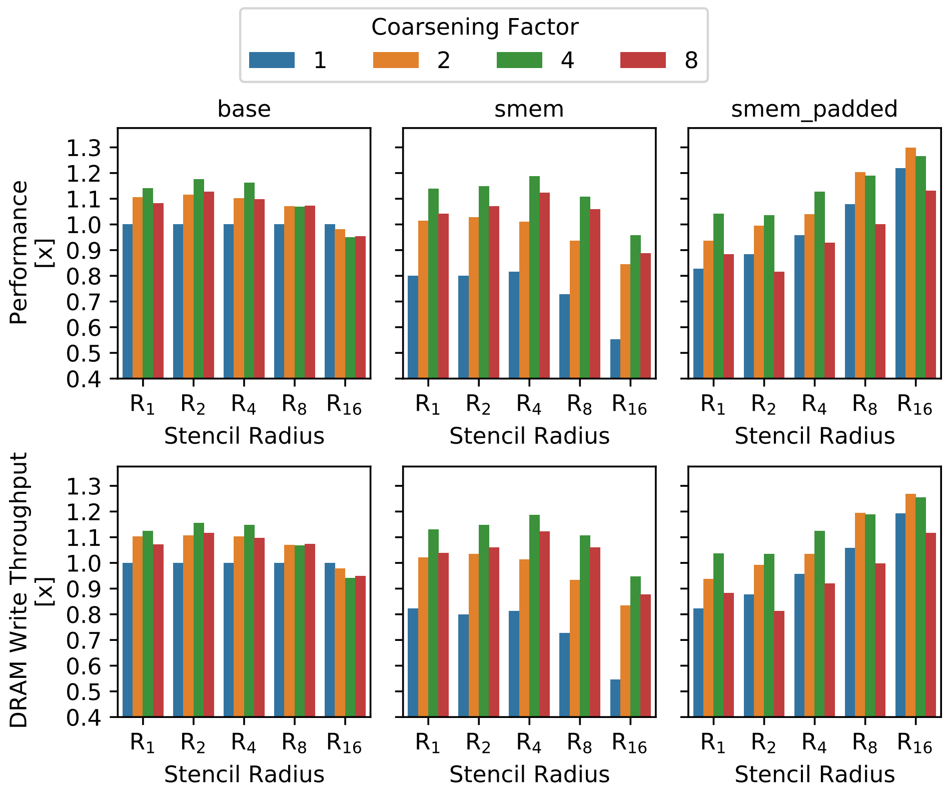


Figure 5.4: The graph displays coarsened kernels (heuristic) in 2D. The rows show performance and DRAM write throughput improvement as a function of stencil radius, displaying each coarsened kernel's metric relative to the 2D baseline kernel normalized by the stencil radius. The DRAM write throughput improvement closely resembles the performance improvement for all configurations, implying that thread coarsening improves performance by increasing this throughput.

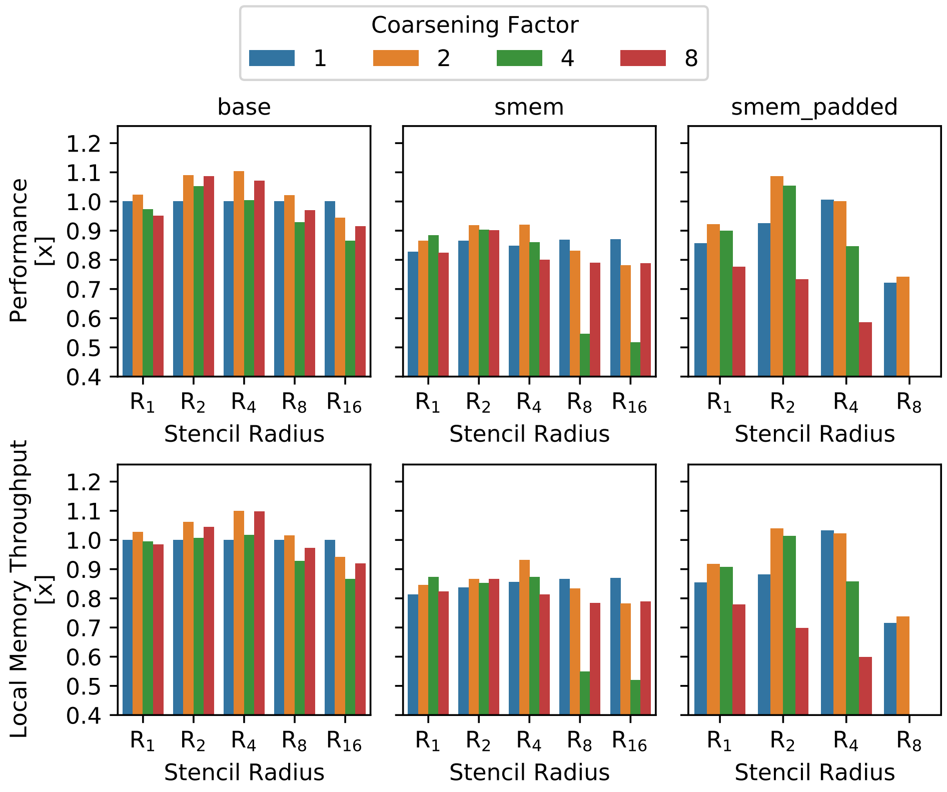


Figure 5.5: The graph displays coarsened kernels (heuristic) in 3D. The rows show performance and local memory throughput improvement as a function of stencil radius, displaying each coarsened kernel's metric relative to the 3D baseline kernel normalized by the stencil radius. The local memory throughput improvement closely resembles the performance improvement for all configurations, suggesting that thread coarsening improves performance by reducing register spilling.

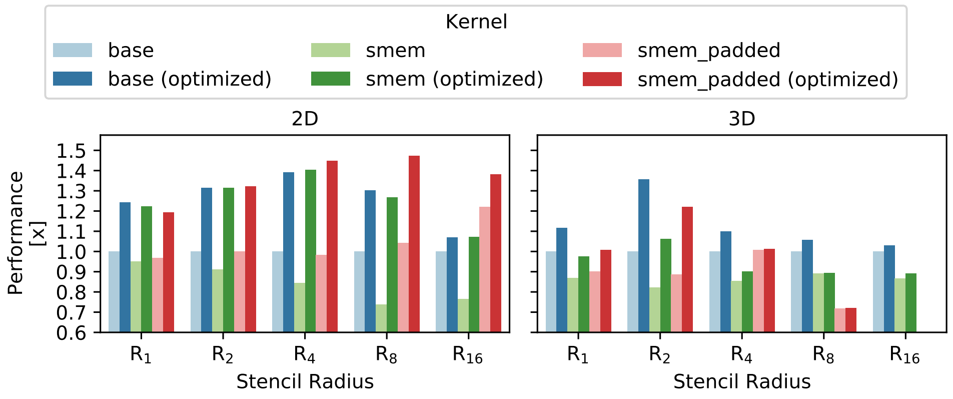


Figure 5.6: The graph compares uncoarsened kernels (heuristic) against the coarsened kernels (autotuned) on Volta by calculating their performance improvement relative to the given dimension's baseline kernel normalized by the stencil radius.

5.1.3 Autotuning

Figure 5.6 presents the measured performance improvements by coarsening the kernels with the best-performing coarsening factor and autotuning TB dimensions. Autotuning the coarsened kernels' TB dimensions improve performance by up to 1.47x (1.36x) over the baseline kernel in 2D (3D). The autotuning scheme covers many values, but the set does not cover all heuristic combinations. Therefore, although the autotuning scheme outperforms the heuristic approach in most cases, this outcome does not always happen.

Figures 5.7 and 5.8 present more detailed graphs for autotuning in 2D and 3D, respectively. The graphs display the kernels' performance and DRAM write throughput (2D) or local memory throughput (3D) improvements with autotuned TB dimensions relative to the baseline kernel. In the same fashion as thread coarsening, the metrics are exceedingly similar to the performance improvement, indicating that autotuning TB dimensions improves performance by exposing more parallel memory operations per thread for 2D kernels and decreasing register usage for 3D kernels.

Interestingly, although thread coarsening worked best for the `smem` kernel, autotuning the TB dimensions improves performance significantly for all the kernels in most cases in 2D and R_2 in 3D. By comparing Figure 5.7 to Figure 5.4, we notice that autotuning is excellent for uncoarsened kernels in 2D, particularly for the `smem` kernel. However, this event does not occur for 3D. We observe that autotuning works better in general for 2D compared to 3D, except for `smem` for R_8 and R_{16} with $CF = 4$. In Figure 5.5, these configurations underperform significantly, decreasing performance by more than 40% compared to the baseline. By comparing Figure 5.8 to Figure 5.5, we observe that for `smem`, autotuning increases performance significantly by 1.45x and 1.40x for R_8 and R_{16}, respectively.

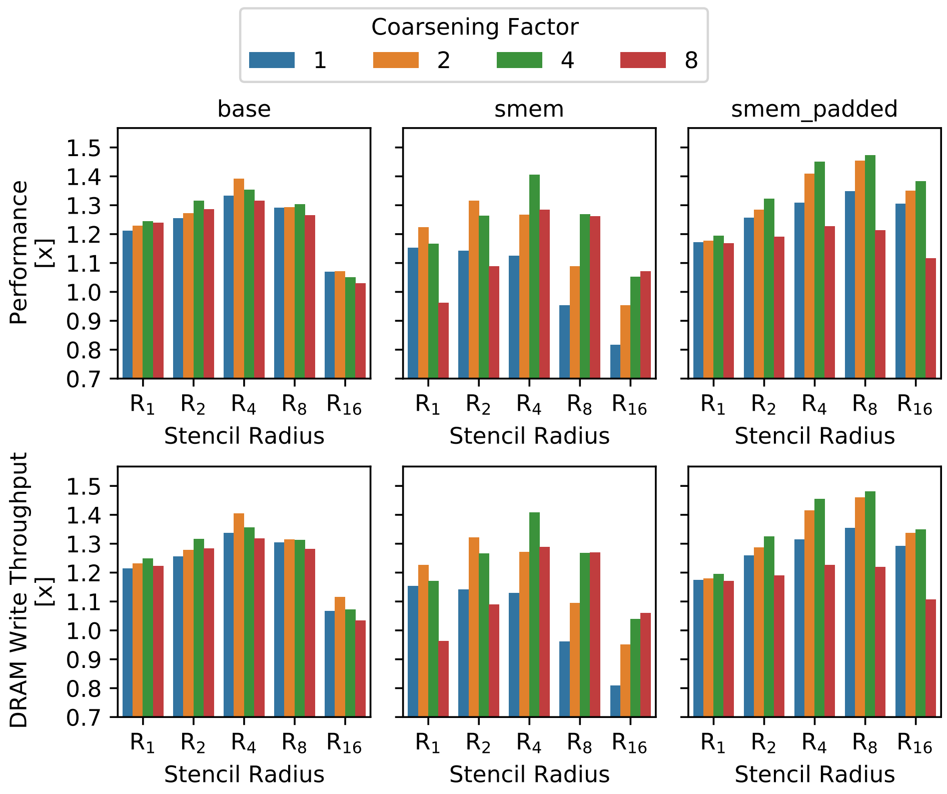


Figure 5.7: The graph displays coarsened kernels (autotuned) in 2D on Volta. The rows show performance and DRAM write throughput improvement as a function of stencil radius, displaying each improvement relative to the given dimension's baseline kernel normalized by the stencil radius. The DRAM write throughput improvement closely resembles the performance improvement for all configurations, implying that autotuning improves performance by increasing this throughput.

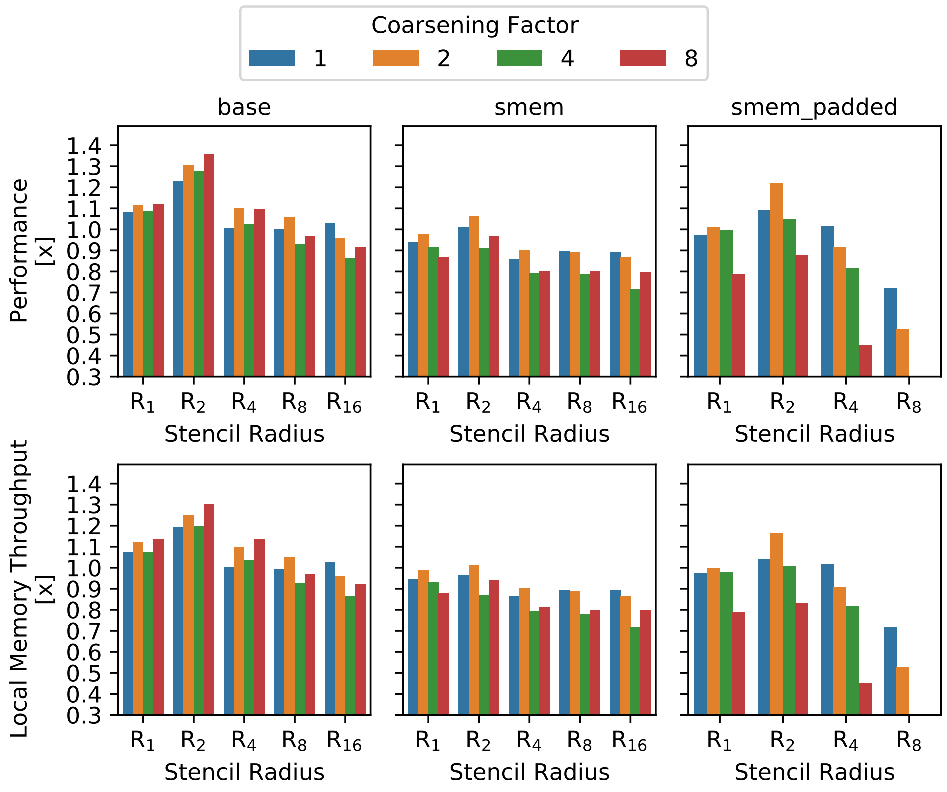


Figure 5.8: The graph displays coarsened kernels (autotuned) in 3D on Volta. The rows show performance and local memory throughput improvement as a function of stencil radius, displaying each coarsened kernel's metric relative to the given dimension's baseline kernel normalized by the stencil radius. The local memory throughput improvement closely resembles the performance improvement for all configurations, implying that autotuning improves performance by reducing register spilling.

5.2 Optimizing ISL Kernels on the Pascal Architecture

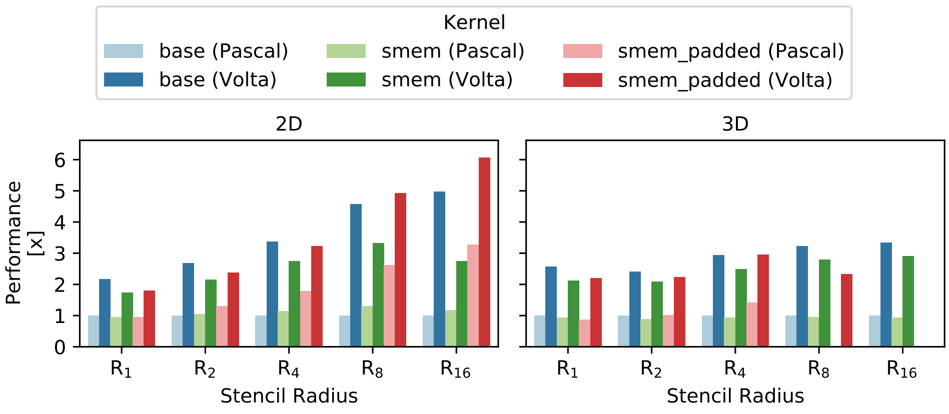
5.2.1 Shared Memory

Figure 5.9 presents the measurements for executing the kernels on the Pascal architecture in 2D and 3D and compares the results against executing on Volta. Figure 5.9a display that `smem_padded` outperforms the other kernels significantly on Pascal for $D = 8$ GiB when $R > 1$ in 2D and $R > 2$ in 3D, showing that its performance improvement over `base` proliferates with increasing stencil radius up to 3.27x (1.43x) for R_{16} (R_4) in 2D (3D). However, this radius' performance improvement shrinks to 2.20x (1.00x) for R_{16} (R_4) in 2D (3D) for a $D = 128$ MiB domain.

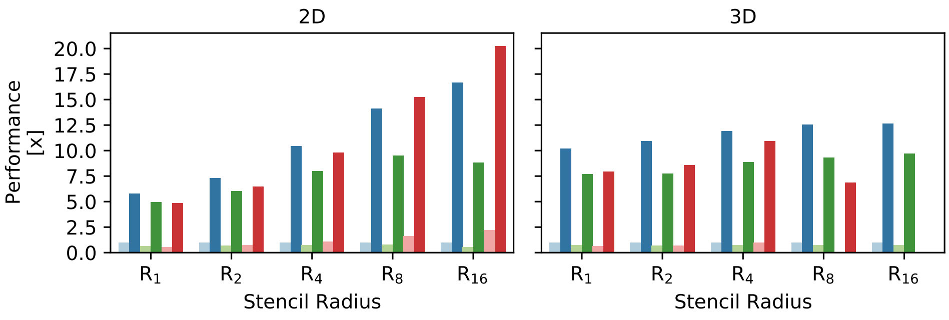
The uncoarsened `smem_padded` kernel's dominance over the uncoarsened `base` kernel on Pascal contrasts Volta's results, where `smem_padded` struggled to compete with `base` due to the DRAM write throughput bottleneck. In contrast to Volta, Pascal's dominant `smem_padded` kernel implies that the kernel benefits significantly from increased shared memory reuse, indicating that cache utilization composes the performance bottleneck for Pascal. The performance difference between $D = 8$ GiB and $D = 128$ MiB implies that increasing shared memory reuse has significantly less impact for smaller domains.

By comparing the columns of Figure 5.9a, we explore that utilizing shared memory within the kernels in 2D is slightly superior to 3D, increasing performance by up to 1.78x (R_4) for `smem_padded` over `base` in 2D compared to 1.43x (R_4) for 3D. The advantage decreases for a smaller domain in Figure 5.9b, where the performance improvement is 1.09x and 1.00x for 2D and 3D, respectively. `smem_padded`'s shared memory allocation per TB increases significantly from 2D to 3D to include the required elements from the third dimension. The Pascal architecture restricts shared memory allocations to 48 KiB per CTA, severely limiting occupancy when each TB requires large allocations. Additionally, the shared memory per CTA limit causes the application to exit for R_8 in 3D on Pascal, as the allocation requires 90 KiB.

Furthermore, Figure 5.9 shows that Volta significantly outperforms Pascal with a performance gap increasing proportionally with the stencil radius, indicating that Pascal's caches do not have sufficient capacity to handle the ISL application's demanding workload. Volta achieves remarkable performance improvements over Pascal, increasing performance by up to 20.3x (12.7x) for $D = 128$ MiB and 6.07x (3.34x) for $D = 8$ GiB for Volta's `smem_padded` kernel compared to Pascal's `base` kernel. These results indicate that the unified cache's improved capacity, bandwidth, number of MSHRs, and shared memory per CTA/SM impact performance significantly.



(a) $D = 8$ GiB



(b) $D = 128$ MiB

Figure 5.9: The graphs display the performance improvement of Volta and Pascal kernels relative to the given dimension's baseline kernel on Pascal normalized by the stencil radius baseline kernel. By comparing the figures, we recognize Volta's massive advantage over Pascal for different domain sizes. Furthermore, the figures show that the benefit increases with larger stencil radiuses.

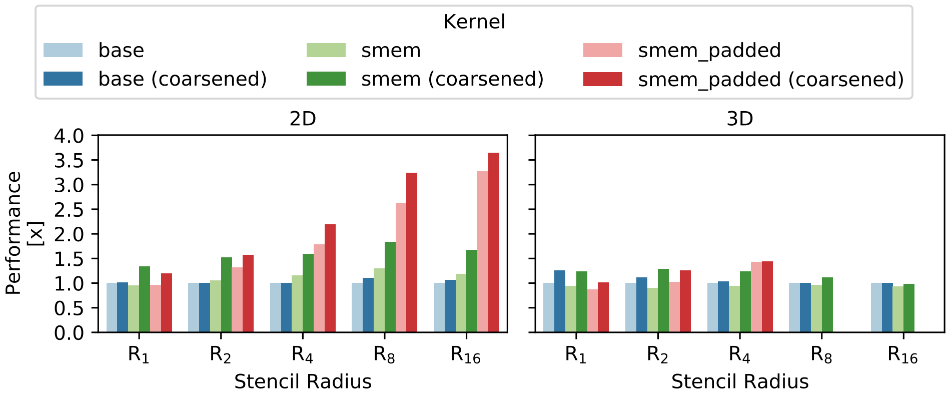


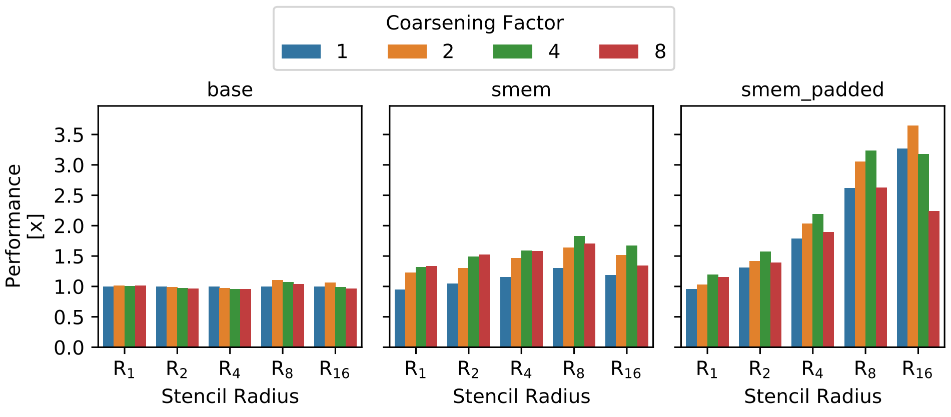
Figure 5.10: The graph compares the performance improvement of uncoarsened kernels (heuristic) against the coarsened kernels with the best observed coarsening factor on Pascal relative to the given dimension's baseline kernel normalized by the stencil radius.

5.2.2 Thread Coarsening

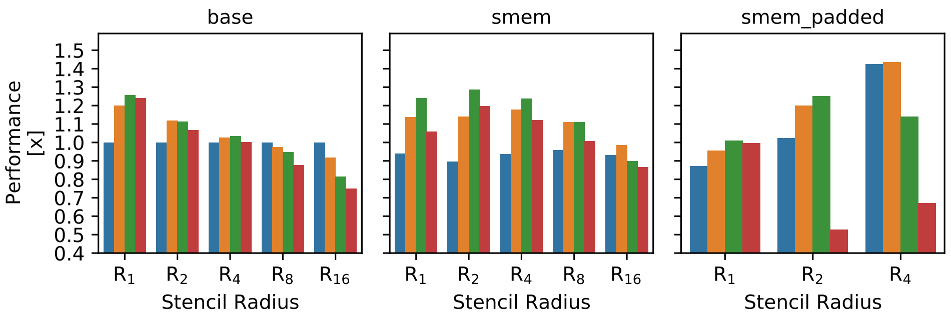
Figure 5.10 presents our results for optimizing the kernels with thread coarsening and selecting the best-performing coarsening factor on Pascal. The figure compares uncoarsened kernels to coarsened kernels. Applying thread coarsening improves the kernels' performance by up to 3.64x (1.44x) over the baseline kernel in 2D (3D), indicating that coarsening impacts performance significantly on Pascal for heuristic TB dimensions. Pascal's shared memory per CTA limit prevents further performance improvements in 3D as R_4 is the largest applicable stencil.

In contrast to Volta, thread coarsening has a limited impact on performance for the `base` kernel on Pascal, except for R_1 and R_2 in 3D. If DRAM utilization bottlenecked Pascal's performance, we would see performance improvements for all the kernels, as thread coarsening would expose more parallel memory operations per thread. Therefore, this result indicates that the performance improvement comes from the large shared memory allocation by enabling more reuse. As we do not have permission to profile the application on Pascal, we can not present the shared load and store throughput. However, by looking at Volta's incredible shared memory throughput through larger allocations, we hypothesize that this is the crucial performance-enhancing factor.

Figures 5.11a and 5.11b present a more detailed graph for thread coarsening on Pascal in 2D and 3D, respectively. In general, coarsening the kernels with $CF = 2$ and $CF = 4$ gives the best performance, and the performance drops for $CF = 8$. As the previous discussion indicated that shared memory reuse was crucial for increasing performance, we believe that $CF = 8$ introduces too large shared memory allocations that map unfavorably to the total amount available, decreasing the performance by reducing occupancy.



(a) 2D



(b) 3D

Figure 5.11: The graphs present coarsened kernels (heuristic) in 2D and 3D on Pascal. The rows show performance improvement as a function of stencil radius, displaying each coarsened kernel's performance improvement relative to the given dimension's baseline kernel normalized by the stencil radius.

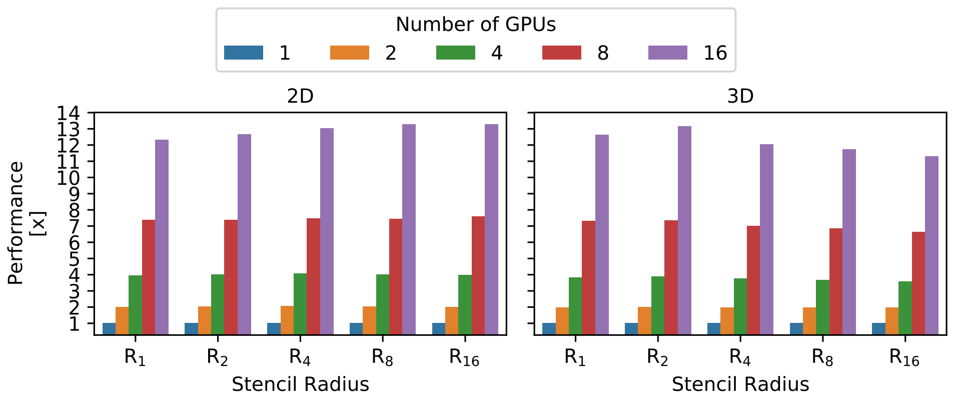
5.3 Multi-GPU

Figures 5.12a and 5.12b present the measured performance improvements for utilizing up to 16 V100 GPUs on the DGX-2 to offload our ISL application's calculations in 2D and 3D for $D = 8$ GiB and $D = 128$ MiB, respectively. Figure 5.12a signifies significant scaling in 2D and 3D for $D = 8$ GiB by achieving performance improvements similar to the number of GPUs utilized except for $G = 16$. Nonetheless, the performance improvements reach an incredible 13.3x (13.2x) for R_{16} (R_2) in 2D (3D). Arguing that $G = 16$ increases variance does not work as $G = 16$'s variance closely resembles the other multi-GPU executions' variance. This deficiency is most likely due to the communication between the GPUs with index 7 and 8 in the strip partitioning approach, as this communication needs to travel between the NVSwitch's two baseboards at half throughput.

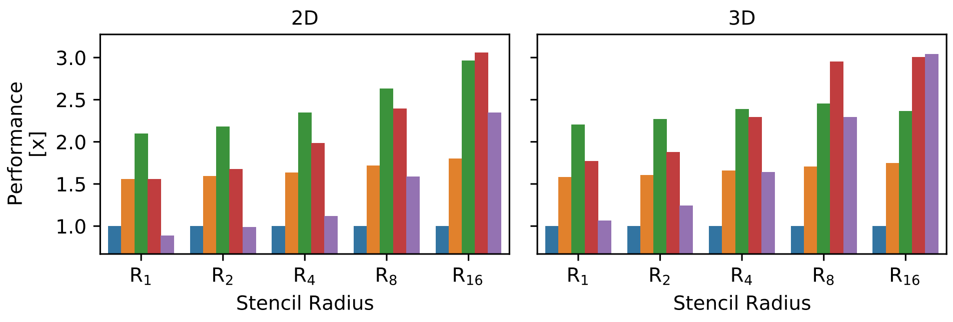
Furthermore, the performance declines slightly by 9.4% from R_1 to R_{16} for $G = 8$ in 3D. The performance declines more substantially by 14% from R_2 to R_{16} for $G = 16$. The stencil radius impacts the number of bytes transferred between GPUs by increasing the ghost zone size, indicating that communication becomes a bottleneck in 3D. We do not observe this effect in 2D, most likely because the inter-GPU communication requirements are less severe. The bottleneck increases from $G = 8$ to $G = 16$ as we introduce communication between baseboards and increase the number of GPUs participating in the communication.

Figure 5.12b shows that our application requires a sufficient domain size to improve performance substantially by utilizing more GPUs. When the $D = 512$ MiB, the performance improvements already diminishes when $G = 2$, indicating that the domain is too small to expose sufficient parallelism from each GPU. Furthermore, the performance peaks for either $G = 4$ or $G = 8$ before rapidly declining. The rate of decline is faster for smaller stencil radiuses. Therefore, multi-GPU requires sufficient domain sizes to offload computation efficiently.

Figures 5.13a and 5.13b compare our ISL application with 8 and 1024 stencil iterations for determining whether the number of stencil iterations affects multi-GPU's scaling capabilities. Both figures show significantly similar measurements in most cases except for $G = 16$. The most significant difference occurs for R_{16} in 2D, where the performance improvements compared to single-GPU are 13.3x and 14.8x for 8 and 1024 iterations, respectively. The difference amounts to 10%, indicating that $G = 16$ is somewhat sensitive to the number of iterations.

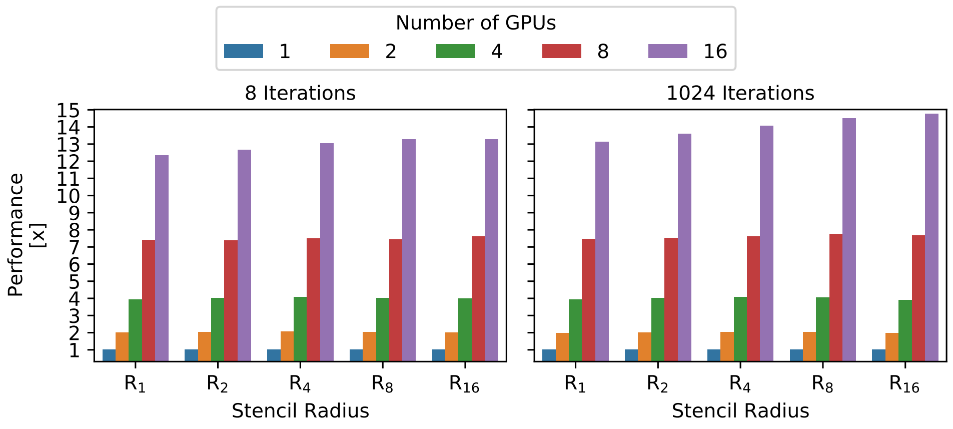


(a) $D = 8$ GiB

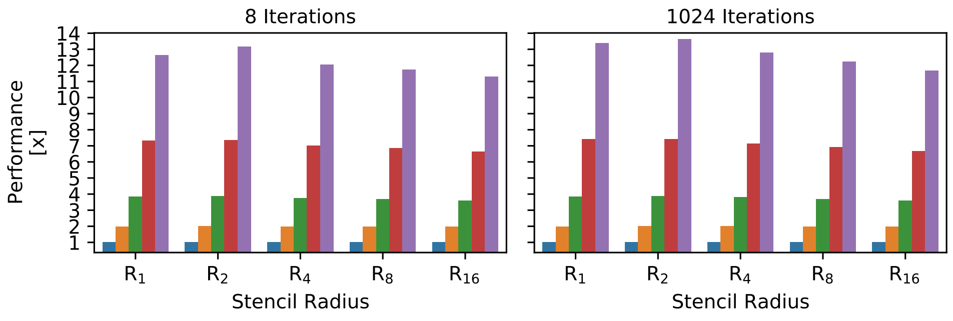


(b) $D = 128$ MiB

Figure 5.12: The figures display the performance improvements by utilizing multiple GPUs relative to a single GPU for the `base` kernel normalized by the stencil radius in 2D and 3D for $D = 8$ GiB and $D = 128$ MiB on the DGX-2.



(a) 2D



(b) 3D

Figure 5.13: The figures display the performance improvements by utilizing multiple GPUs relative to a single GPU for the `base` kernel normalized by the stencil radius in 2D and 3D for 8 and 1024 stencil iterations on the DGX-2.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We conclude our contributions by highlighting their objective task from Chapter 1 in parentheses. This thesis implemented an ISL application in 2D and 3D (T1) and contributed a comprehensive analysis into utilizing a combination of shared memory, thread coarsening, and autotuning TB dimensions (T2) for optimizing RTM stencils with various stencil radiuses for 2D and 3D domains (T3). We found that introducing shared memory achieves the most stencil-dependent performance improvements (T3). Furthermore, we discovered the optimizations' key performance bottlenecks by discovering that DRAM write (local memory) throughput bottlenecks performance in 2D (3D) on Volta (T2), indicating that moving from 2D to 3D introduces significant register spilling (T1). Additionally, we argued that utilizing shared memory has a significant impact on Pascal, indicating that cache utilization composes this architecture's performance bottleneck instead of DRAM throughput (T2). Moreover, we observed immensely increased performance for our ISL application by upgrading the GPU architecture from Pascal to Volta (T4). Finally, offloading workload onto more GPUs incurred tremendous performance improvements closely resembling the number of GPUs utilized (T5).

6.2 Future Work

The thesis has given a great introduction to the topic of ISLs on GPUs. However, there are many aspects of ISLs on GPUs we have not covered. Therefore, we wish to mention specific ways to continue our work.

6.2.1 Spatial and Temporal Blocking

Chapter 2 compared our ISL application's 3D blocking scheme to Nguyen et al.'s [16] 3.5D scheme combining 2.5D spatial blocking and 1D temporal blocking. We argued that 3D blocking is more flexible than 2.5D blocking by not imposing restrictions on the cache capacity for achieving good performance. However, the authors' hardware (Nvidia GTX 285) is aged, and extending our ISL application with 2.5D blocking would be interesting for experimenting with V100's enormous cache capacities.

Furthermore, our ISL application can potentially benefit from introducing temporal blocking techniques to transform our memory-bound kernels into compute-bound. However, this technique is feasible only if we keep all redundant elements cached for extracting substantial performance, which can be intricate for larger RTM stencils. The implementation can either utilize shared memory or even register caching, although using shared memory would be the most straightforward approach. However, reducing the number of threads per CTA to the warp size to use warp-level instruction instead of block-level instructions in combination with register caching could remove much of the overhead associated with block-level instructions. Therefore, we argue that utilizing both shared memory or register schemes for temporal blocking is an exciting continuation of our implementation.

6.2.2 Stencil Patterns

Our ISL application emphasizes an RTM stencil pattern for various stencil radiuses in 2D and 3D. Chapter 2 covered two other approaches: square-shaped (cube-shaped) Jacobi patterns and X-shaped Gauss-Seidel patterns. Implementing kernels following such patterns would incur different results as their memory access patterns differ from the RTM pattern. The Jacobi pattern includes a significantly more considerable amount of neighboring values when calculating the stencil, putting more pressure on the GPU's resources.

Furthermore, the Jacobi pattern would cause bank conflicts in shared memory as the threads' access to shared memory would overlap across banks. Solving the issue would require bank conflict mitigation approaches (e.g., shared memory padding [13, p. 211]).

6.2.3 Iteration Patterns

Our ISL application utilized Jacobi iterations for computing the stencil, doubling the storage requirements for the domain by including an input and output buffer. The incredible domain size demands of modern workloads motivate using a different iteration pattern. Utilizing Gauss-Seidel iterations or Gauss-Seidel Red-Black iterations halves the storage requirement for the domain by using a single buffer [25, 59, 74].

The Gauss-Seidel iteration pattern improves the convergence rate for specific applications, requiring fewer iterations to complete. However, the Gauss-Seidel approach is less parallelizable and is therefore not attractive for GPUs. This deficiency is due to dependencies between the simultaneously calculated values, as the method emphasizes the current iteration over previous iterations. Gauss-Seidel Red-Black iterations improve Gauss-Seidel's parallelism issues by dividing the grid into red and black nodes. The method performs the iterative refinement for one of the colors first, followed by computing the other color. However, the pattern only works for R_1 stencils as neighboring nodes require a different color.

6.2.4 Case Studies

Koraei et al. [3] highlighted that ISLs often compose large parts of applications' total execution time and mentions SPEC 2017's Lattice Boltzmann Method as a particular application spending 100% of the execution time calculating ISLs. However, the authors emphasized numerous applications with differing amounts of time spent calculating ISLs and mention other SPEC 2017 applications like the Parallel Ocean Program that only spends 26% of the execution time calculating ISLs. Therefore, future work should focus not only on including specific stencil and iteration patterns but also on implementing real-world applications that use specific stencil configurations to provide broader practical insights.

6.2.5 Autotuning

Our autotuning approach utilized brute-force search to explore a set of possible TB dimension configurations on Volta. This set was a subset of all possible configurations, creating a significant potential for sub-optimal configurations. However, enlarging the set incurs massive time requirements for executing the application, which is unfavorable when using a shared system. Meanwhile, we believe we have covered a crucial part of the configuration space for improving bandwidth utilization. Therefore, further search should not incur massive performance improvements but can give interesting results. Nonetheless, if we were to implement our kernels in a real-world application with time limits but substantial performance demands, we would expand our configuration set and utilize an improved autotuner that uses more effective search algorithms as brute-force search becomes intractable for large configuration sets.

6.2.6 Improving Inter-GPU Communication

Chapter 5 concluded that communication was an issue when offloading computations onto all GPUs within the DGX-2. The NVSwitch interconnect system's characteristics halved the throughput between a pair of GPUs when utilizing more than 8 GPUs and strip partitioning. Therefore, we argue that improving inter-GPU communications should improve performance when utilizing more than 8 GPUs.

Chapter 2 covered two approaches for improving inter-GPU communication: Reducing communication by enlarging the partitions' ghost zones and utilizing multiple OpenMP threads with multiple CUDA streams to overlap communication with computation. These techniques might improve our issue with utilizing more than 8 GPUs on the DGX-2. Therefore, we propose combining both approaches as an exciting approach to continue our work.

Bibliography

[1] M. R. Farstad and A. C. Elster, 'Using Modern Optimization Techniques to Study Iterative Methods on GPUs,' unpublished.

[2] J. Meng and K. Skadron, 'A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations,' *International Journal of Parallel Programming*, no. 1, pp. 115–142, 2011.

[3] M. Koraei, O. Fatemi and M. Jahre, 'DCMI: A Scalable Strategy for Accelerating Iterative Stencil Loops on FPGAs,' *ACM Trans. Archit. Code Optim.*, vol. 16, no. 36, 2019.

[4] K. Asanovi, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams and K. A. Yelick, 'The Landscape of Parallel Computing Research: A View from Berkeley,' EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

[5] M. Gardner, 'Mathematical Games - The Fantastic Combinations of John Conway's New Solitaire Game "life",' en, *Natural Science*, vol. 6, no. 13, pp. 120–123, 1970.

[6] M. R. Rodriguez, B. Philip, Z. Wang and M. A. Berrill, 'Block-Relaxation Methods for 3D Constant-Coefficient Stencils on GPUs and Multicore CPUs,' *CoRR*, 2018. arXiv: `1208.1975 [cs.DC]`.

[7] Z. Li and Y. Song, 'Automatic Tiling of Iterative Stencil Loops,' *ACM Trans. Program. Lang. Syst.*, no. 6, pp. 975–1028, 2004.

[8] R. Strzodka, M. Shaheen, D. Pajak and H.-P. Seidel, 'Cache Oblivious Parallelograms in Iterative Stencil Computations,' in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10, 2010, pp. 49–59.

[9] H. Stengel, J. Treibig, G. Hager and G. Wellein, 'Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model,' in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15, 2015, pp. 207–216.

[10] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf and K. Yelick, 'Auto-Tuning the 27-point Stencil for Multicore,' in *In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*, 2009.

[11] SPEC, *SPEC CPU 2017*, 2017. [Online]. Available: `https://www.spec.org/cpu2017/`.

[12] J. Holewinski, L.-N. Pouchet and P. Sadayappan, 'High-Performance Code Generation for Stencil Computations on GPU Architectures,' in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12, 2012, pp. 311–320.

[13] J. Cheng, M. Grossman and T. McKercher, *Professional CUDA C Programming*, 1st. Wrox Press Ltd., 2014, ISBN: 1118739329.

[14] Y. Chi and J. Cong, 'Exploiting Computation Reuse for Stencil Accelerators,' in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '20, 2020.

[15] H. R. Zohouri, A. Podobas and S. Matsuoka, 'Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL,' *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018.

[16] A. Nguyen, N. Satish, J. Chhugani, C. Kim and P. Dubey, '3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs,' in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–13.

[17] P. Micikevicius, '3D Finite Difference Computation on GPUs Using CUDA,' in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2, 2009, pp. 79–84.

[18] S. Tabik, M. Peemen, N. Guil and H. Corporaal, 'Demystifying the 16 x 16 Thread-Block for Stencils on the GPU,' *Concurrency and Computation: Practice and Experience*, vol. 27, no. 18, pp. 5557–5573, 2015.

[19] Nvidia, *CUDA C++ Programming Guide*, 2021. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[20] T. L. Falch and A. C. Elster, 'Register Caching for Stencil Computations on GPUs,' in *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2014, pp. 479–486.

[21] N. Stawinoga and T. Field, 'Predictable Thread Coarsening,' *ACM Trans. Archit. Code Optim.*, vol. 15, no. 2, 2018.

[22] J. A. Stratton, N. Anssari, C. Rodrigues, I.-J. Sung, N. Obeid, L. Chang, G. D. Liu and W.-m. Hwu, 'Optimization and Architecture Effects on GPU Computing Workload Performance,' in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[23] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch and C. Dubach, 'High Performance Stencil Code Generation with Lift,' in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018, 2018, pp. 100–112.

[24] J. Choquette, O. Giroux and D. Foley, 'Volta: Performance and Programmability,' *IEEE Micro*, vol. 38, no. 2, pp. 42–52, 2018.

[25] K. Datta, 'Auto-Tuning Stencil Codes for Cache-Based Multicore Platforms,' Ph.D. dissertation, University of California at Berkeley, 2009.

[26] D. Wonnacott, 'Achieving Scalable Locality With Time Skewing,' *International Journal of Parallel Programming*, vol. 30, Mar. 1999.

[27] R. Strzodka, M. Shaheen, D. Pajak and H.-P. Seidel, 'Cache Accurate Time Skewing in Iterative Stencil Computations,' in *2011 International Conference on Parallel Processing*, 2011, pp. 571–581.

[28] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev and P. Sadayappan, 'Effective Automatic Parallelization of Stencil Computations,' in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, 2007, pp. 235–244.

[29] K. Matsumura, H. R. Zohouri, M. Wahib, T. Endo and S. Matsuoka, 'AN5D: Automated Stencil Framework for High-Degree Temporal Blocking on GPUs,' in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020, 2020, pp. 199–211.

[30] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet and P. Sadayappan, 'Effective Resource Management for Enhancing Performance of 2D and 3D Stencils on GPUs,' in *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, ser. GPGPU '16, 2016, pp. 92–102.

[31] R. Cattaneo, G. Natale, C. Sicignano, D. Sciuto and M. D. Santambrogio, 'On How to Accelerate Iterative Stencil Loops: A Scalable Streaming-Based Approach,' *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, 2015.

[32] V. Rana, I. Beretta, F. Bruschi, A. A. Nacci, D. Atienza and D. Sciuto, 'Efficient Hardware Design of Iterative Stencil Loops,' *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, no. 12, pp. 2018–2031, 2016.

[33] Y. Lin and V. Grover, *Using CUDA Warp-Level Primitives*, 2018. [Online]. Available: `https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/`.

[34] Nvidia, *NVIDIAs Next Generation CUDA Compute Architecture: Fermi*, 2009. [Online]. Available: `https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[35] A. Li, W. Liu, L. Wang, K. Barker and S. L. Song, 'Warp-Consolidation: A Novel Execution Model for GPUs,' in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18, 2018, pp. 53–64.

[36] D. Kroft, 'Lockup-Free Instruction Fetch/Prefetch Cache Organization,' in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ser. ISCA '81, 1981, pp. 81–87.

[37] M. Jahre and L. Natvig, 'A High Performance Adaptive Miss Handling Architecture for Chip Multiprocessors,' in *Transactions on High-Performance Embedded Architectures and Compilers IV*, P. Stenström, Ed. 2011, p. 2.

[38] L. Wang, M. Jahre, A. Adileho and L. Eeckhout, 'MDM: The GPU Memory Divergence Model,' in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 1009–1021.

[39] M. Harris, *CUDA 9 Features Revealed: Volta, Cooperative Groups and More*, 2017. [Online]. Available: `https://developer.nvidia.com/blog/cuda-9-features-revealed/`.

[40] L. Zhang, M. Wahib, H. Zhang and S. Matsuoka, 'A Study of Single and Multi-Device Synchronization Methods in Nvidia GPUs,' in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 483–493.

[41] Nvidia, 'Parallel Thread Execution ISA,' 2021. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/ptx_isa_7.3.pdf`.

[42] Nvidia, *NVIDIA TESLA v100 GPU ARCHITECTURE*, 2017. [Online]. Available: `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`.

[43] V. Volkov, 'Better Performance at Lower Occupancy,' 2010. [Online]. Available: `https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf`.

[44] Y. Liu, X. Zhao, M. Jahre, Z. Wang, X. Wang, Y. Luo and L. Eeckhout, 'Get out of the Valley: Power-Efficient Address Mapping for GPUs,' in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18, 2018, pp. 166–179.

[45] Z. Jia, M. Maggioni, B. Staiger and D. P. Scarpazza, 'Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking,' *CoRR*, 2018. arXiv: `1804.06826`.

[46] Nvidia, *Volta Tuning Guide :: CUDA Toolkit Documentation*, 2021. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/Volta_Tuning_Guide.pdf`.

[47] X. Zhao, M. Jahre and L. Eeckhout, 'HSM: A Hybrid Slowdown Model for Multitasking GPUs,' in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, 2020, pp. 1371–1385.

[48] Nvidia, *NVIDIA Profiler User's Guide*, 2021. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf`.

[49] Nvidia, *NVIDIA Nsight Compute User Manual*, 2021. [Online]. Available: `https://docs.nvidia.com/nsight-compute/pdf/NsightCompute.pdf`.

[50] Nvidia, 'CUDA Occupancy Calculator,' 2021. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/CUDA-Occupancy-Calculator.pdf`.

[51] H. Lin, C.-L. Wang and H. Liu, 'On-GPU Thread-Data Remapping for Branch Divergence Reduction,' *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, 2018.

[52] P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L.-N. Pouchet, A. Rountev and P. Sadayappan, 'Register Optimizations for Stencils on GPUs,' *SIGPLAN Not.*, no. 1, pp. 168–182, 2018.

[53] A. Magni, C. Dubach and M. F. P. O'Boyle, 'A Large-Scale Cross-Architecture Evaluation of Thread-Coarsening,' in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, 2013.

[54] A. Magni, C. Dubach and M. O'Boyle, 'Automatic Optimization of Thread-Coarsening for Graphics Processors,' in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14, 2014, pp. 455–466.

[55] S. Unkule, C. Shaltz and A. Qasem, 'Automatic Restructuring of GPU Kernels for Exploiting Inter-thread Data Locality,' in *Compiler Construction*, M. O'Boyle, Ed., 2012, pp. 21–40.

[56] V. Volkov and J. W. Demmel, 'Benchmarking GPUs to Tune Dense Linear Algebra,' in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08, 2008.

[57] Y. Liu, Z. Yu, L. Eeckhout, V. J. Reddi, Y. Luo, X. Wang, Z. Wang and C. Xu, 'Barrier-Aware Warp Scheduling for Throughput Processors,' in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16, 2016.

[58] B. Spencer, 'A General Auto-tuning Framework for Software Performance Optimisation,' Balliol College, University of Oxford, 2011.

[59] D. Spampinato, 'Modeling Communication on Multi-GPU Systems,' M.S. thesis, NTNU, Department of Computer Science, 2009.

[60] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)*. Prentice-Hall, Inc., 2004, ISBN: 0131405632.

[61] T. Ben-Nun, E. Levy, A. Barak and E. Rubin, 'Memory Access Patterns: the Missing Piece of the Multi-GPU Puzzle,' in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[62] M. Sourouri, T. Gillberg, S. B. Baden and X. Cai, 'Effective Multi-GPU Communication Using Multiple CUDA Streams and Threads,' in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2014, pp. 981–986.

[63] L. Dagum and R. Menon, 'OpenMP: An Industry-Standard API for Shared-Memory Programming,' *IEEE Computational Science and Engineering*, no. 1, pp. 46–55, 1998.

[64] L. Clarke, I. Glendinning and R. Hempel, 'The MPI Message Passing Interface Standard,' *Int. J. Supercomput. Appl.*, vol. 8, 1996.

[65] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent and K. J. Barker, 'Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect,' *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2020.

[66] Nvidia, *CUDA Runtime API :: CUDA Toolkit Documentation*, 2021. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf`.

[67] Nvidia, *NVIDIA DGX-2: The Worlds Most Powerful Deep Learning System for the Most Complex AI Challenges*, 2019. [Online]. Available: `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-2-datasheet-us-nvidia-955420-r2-web-new.pdf`.

[68] M. Själander, M. Jahre, G. Tufte and N. Reissmann, *EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure*, 2019. arXiv: `1912.05848 [cs.DC]`.

[69] Nvidia, 'NVIDIA NVSwitch: The World's Highest-Bandwidth On-Node Switch,' 2018. [Online]. Available: `https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf`.

[70] SchedMD, *Slurm Workload Manager - Quick Start User Guide*. [Online]. Available: `https://slurm.schedmd.com/quickstart.html`.

[71] Seaborn, *Statistical Data Visualization seaborn 0.11.1 documentation*. [Online]. Available: `https://seaborn.pydata.org/`.

[72] T. Hoefler and R. Belli, 'Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results,' in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, 2015.

[73] N. Wright, S. Smallen, C. Olschanowsky, J. Hayes and A. Snavely, 'Measuring and Understanding Variation in Benchmark Performance,' *HPCMP Users Group Conference*, pp. 438–443, 2009.

[74] D. Evans, 'Parallel S.O.R. Iterative Methods,' *Parallel Computing*, no. 1, pp. 3–18, 1984.

Appendix A

Autotuned Thread Block Dimensions

Tables A.1 and A.2 present the autotuned TB dimensions for each kernel configuration, showing that the kernels generally prefer a larger X-dimension. Furthermore, the tables adhere to the $B_x, B_y, B_z \geq RADIUS$ requirement for the `smem_padded` kernel, outputting "N/A" for configurations that exit early.

Table A.1: Autotuned 2D TB dimensions (`blockDim.x`, `blockDim.y`) for every combination of kernel, stencil radius, and coarsening factor when applying eight stencil iterations on a single V100 GPU.

Kernel	Radius	CF=1	CF=2	CF=4	CF=8
base	1	(256, 1)	(128, 8)	(64, 1)	(32, 1)
base	2	(256, 1)	(128, 8)	(64, 8)	(128, 4)
base	4	(256, 4)	(128, 8)	(128, 8)	(128, 8)
base	8	(64, 8)	(128, 8)	(64, 8)	(64, 8)
base	16	(256, 4)	(256, 4)	(128, 8)	(128, 8)
smem	1	(256, 1)	(128, 1)	(64, 8)	(32, 16)
smem	2	(256, 1)	(256, 1)	(64, 8)	(32, 16)
smem	4	(256, 1)	(64, 8)	(64, 2)	(64, 16)
smem	8	(256, 1)	(32, 16)	(32, 16)	(32, 16)
smem	16	(256, 2)	(64, 4)	(32, 32)	(32, 16)
smem_padded	1	(256, 1)	(128, 1)	(64, 8)	(64, 8)
smem_padded	2	(256, 2)	(128, 2)	(128, 4)	(32, 8)
smem_padded	4	(128, 4)	(128, 4)	(64, 4)	(32, 32)
smem_padded	8	(64, 8)	(32, 8)	(32, 8)	(16, 16)
smem_padded	16	(32, 16)	(32, 16)	(32, 32)	(32, 32)

Table A.2: Autotuned 3D TB dimensions (`blockDim.x`, `blockDim.y`, `blockDim.z`) for every combination of kernel, stencil radius, and coarsening factor when applying eight stencil iterations on a single V100 GPU.

Kernel	Radius	CF=1	CF=2	CF=4	CF=8
base	1	(32, 4, 4)	(32, 4, 4)	(64, 4, 4)	(64, 4, 4)
base	2	(32, 4, 4)	(32, 4, 4)	(64, 4, 4)	(64, 4, 4)
base	4	(32, 4, 4)	(32, 8, 4)	(64, 4, 4)	(64, 4, 4)
base	8	(16, 8, 8)	(16, 8, 8)	(32, 8, 4)	(32, 8, 4)
base	16	(16, 8, 8)	(16, 8, 8)	(32, 8, 4)	(32, 8, 4)
smem	1	(64, 2, 2)	(64, 2, 2)	(64, 2, 2)	(32, 8, 4)
smem	2	(32, 4, 4)	(32, 4, 4)	(32, 4, 4)	(32, 8, 4)
smem	4	(32, 4, 4)	(32, 4, 4)	(32, 4, 4)	(32, 8, 4)
smem	8	(32, 4, 4)	(32, 4, 4)	(16, 8, 8)	(64, 4, 4)
smem	16	(16, 8, 8)	(32, 4, 4)	(16, 8, 8)	(64, 4, 4)
smem_padded	1	(64, 2, 2)	(32, 4, 2)	(32, 4, 4)	(32, 4, 4)
smem_padded	2	(32, 4, 4)	(32, 4, 4)	(16, 8, 4)	(16, 8, 8)
smem_padded	4	(32, 8, 4)	(32, 4, 4)	(16, 8, 8)	(16, 4, 4)
smem_padded	8	(16, 8, 8)	N/A	N/A	N/A
smem_padded	16	N/A	N/A	N/A	N/A