Directive-Based Data Partitioning, Pipelining and Auto-Tuning for High-Performance GPU Computing

Xuewen Cui

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Application

Wu-chun Feng, Chair
Calvin J. Ribbens
Adrian Sandu
Hao Wang
Thomas R. W. Scogland
Bronis R. de Supinski

September 11, 2020
Blacksburg, Virginia

Keywords: Pipelining, GPUs, Partitioning, OpenMP, Streaming Multiprocessor, Multi-core, Persistent Thread Blocks, Auto-Tuning, Machine Learning

Directive-Based Data Partitioning, Pipelining and Auto-Tuning for High-Performance GPU Computing

Xuewen Cui

(ABSTRACT)

The computer science community needs simpler mechanisms to achieve the performance potential of accelerators, such as graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and co-processors (e.g., Intel Xeon Phi), due to their increasing use in state-of-the-art supercomputers. Over the past 10 years, we have seen a significant improvement in both computing power and memory connection bandwidth for accelerators. However, we also observe that the computation power has grown significantly faster than the interconnection bandwidth between the central processing unit (CPU) and the accelerator.

Given that accelerators generally have their own discrete memory space, data needs to be copied from the CPU host memory to the accelerator (device) memory before computation starts on the accelerator. Moreover, programming models like CUDA, OpenMP, OpenACC, and OpenCL can efficiently offload compute-intensive workloads to these accelerators. However, achieving the overlapping of data transfers with computation in a kernel with these models is neither simple nor straightforward. Instead, codes copy data to or from the device without overlapping or requiring explicit user design and refactoring.

Achieving performance can require extensive refactoring and hand-tuning to apply data transfer optimizations, and users must manually partition their dataset whenever its size is larger than device memory, which can be highly difficult when the device memory size is not exposed to the user. As the systems are becoming more and more complex in terms of heterogeneity, CPUs are responsible for handling many tasks related to other accelerators, computation and data movement tasks, task dependency checking, and task callbacks. Leaving all logic controls to the CPU not only costs extra communication delay over PCI-e

bus but also consumes the CPU resources, which may affect the performance of other CPU tasks. This thesis work aims to provide efficient directive-based data pipelining approaches for GPUs that tackle these issues and improve performance, programmability, and memory management.

Directive-Based Data Partitioning, Pipelining and Auto-Tuning for High-Performance GPU Computing

Xuewen Cui

(GENERAL AUDIENCE ABSTRACT)

Over the past decade, parallel accelerators have become increasingly prominent in this emerging era of "big data, big compute, and artificial intelligence." In more recent supercomputers and datacenter clusters, we find multi-core central processing units (CPUs), many-core graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and co-processors (e.g., Intel Xeon Phi) being used to accelerate many kinds of computation tasks.

While many new programming models have been proposed to support these accelerators, scientists or developers without domain knowledge usually find existing programming models not efficient enough to port their code to accelerators. Due to the limited accelerator on-chip memory size, the data array size is often too large to fit in the on-chip memory, especially while dealing with deep learning tasks. The data need to be partitioned and managed properly, which requires more hand-tuning effort. Moreover, performance tuning is difficult for developers to achieve high performance for specific applications due to lack of domain knowledge. To handle these problems, this dissertation aims to propose a general approach to provide better programmability, performance, and data management for the accelerators. Accelerator users often prefer to keep their existing verified C, C++, or Fortran code rather than grapple with the unfamiliar code. Since 2013 [57], OpenMP has provided a straightforward way to adapt existing programs to accelerated systems. We propose multiple associated clauses to help developers easily partition and pipeline the accelerated code. Specifically, the proposed extension can overlap kernel computation and data transfer between host and device efficiently. The extension supports memory over-subscription, meaning the memory

size required by the tasks could be larger than the GPU size. The internal scheduler guarantees that the data is swapped out correctly and efficiently. Machine learning methods are also leveraged to help with auto-tuning accelerator performance.

Acknowledgments

First, I would like to thank my advisor, Professor Wu-chun Feng. Throughout my Ph.D. career, Wu has provided me with invaluable support and knowledge. He has been leading me to the HPC area and providing me with opportunities to learn techniques as well as better communication skills. I've learned a lot from him about how to write and review papers, how to present my work, and how to establish productive collaboration with others.

My sincere thanks to my other committee members: Professor Calvin J. Ribbens, Professor Adrian Sandu, Dr. Hao Wang, Dr. Thomas RW Scogland, and Dr. Bronis R. de Supinski for all their time, comments, suggestions and assistance about my research work during my Ph.D. journey.

I have also spent valuable time collaborating with Lawrence Livermore National Laboratory throughout my Ph.D. as an intern at LLNL, which also had a significant impact on my research and technique knowledge. Special thanks go to Dr. Thomas RW Scogland(again) and Dr. Bronis de Supinski(again).

I am so fortunate to have been around wonderful researchers during my Ph.D.. We lived and fought together during those hard but happy days. Especially thanks to Dr. Hao Wang(again), Kaixi Hou, Jing Zhang, Xiaodong Yu, Da Zhang, Sarunya (Kwang) Pumma, and Sajal Dash. I cannot forget those nights that we worked together to catch paper submission deadlines.

I owe a debt of gratitude to Dr. James Lin for leading me to the area of HPC when I was still a third-year undergraduate student.

Finally, and most importantly, I wish to thank my family for their unconditional love, support, and encouragement. I will be forever grateful to my parents, Yanjun Cui and Zhijie

Ma for teaching and supporting me during the last 29 years.

Funding Acknowledgment This thesis work was in part performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. It was also supported in part by AFOSR BRI Project and AFOSR Machine Learning Project.

Contents

List of Figures

List of Tables

List of Abbreviations

accelerator High performance accelerator hardware, e.g., GPUs, FPGAs, and APUs.

BLP Block-level data partitioning and pipelining.

device Same as *accelerator*.

DToH Device to host memory transfer.

FLOPS Floating point operations per second.

HPC High performance computing.

HtoD Host to device memory transfer.

IterML Iterative machine learning.

NVLink Nvidia's wire-based communications protocol.

PCIe Peripheral Component Interconnect Express.

UM Nvidia's Unified Memory for GPUs.

UVA Nvidia's Unified Virtual Addressing technique.

Chapter 1

Introduction

In this chapter, we will motivate our work on "Directive-Based Data Partitioning and Pipelining and Auto-Tuning for High-Performance GPU Computing." We will outline the challenges and existing drawbacks of accelerator programming models and also describe the contribution of this dissertation.

1.1 Motivation

High Performance Computing has been evolving fast for its wide application in a lot of domain areas, including healthcare, engineering, space research, urban planning, finance and business, machine learning and AI, among others. The success of high performance computing in the last two decades significantly improved the computation power, which highly pushed the related domain areas to evolve even further. One of these areas, machine learning and AI, has been really popular in the last couple decades. It has been changing all other domain areas including HPC, not only improving the consumer market of HPC, but also affecting the research and development of HPC.

If we look at the history of these theories (e.g., neural networks, backpropagation, LSTM), mostly utilized by current techniques in machine learning and AI, most of them were published in the early 1970s. It was not until the late 2000s that machine learning and AI started to become useful because of limited data storage and computation power. Neural networks

have the advantages of continuing to improve as more training data is added. Model training usually takes a lot computation power.

Back in the 1970s, the computers at that time were not capable of supporting that huge computation power requirement. One of the most successful supercomputers in history, Cray-1 [70], was announced in 1975 at Los Alamos National Laboratory. It could achieve 160 MFLOPS computation power with 303MB storage. The current fastest supercomputer, Summit [85] at Oak Ridge National Laboratory achieves 200 PFLOPS computer power with 250PB storage. That is 1.25 billion times computation power speedup and 8.25 million times capacity improvement. Being able to analyze behavior in real-time and act directly based on that is really a game-changer for machine learning and AI. Part of this significant computation power increase was due to the success of recent accelerators such as GPUs, Co-processors, APUs, FPGAs, TPUs. On the other part, the success of distributed systems (since Google released the key 3 papers about GFS, MapReduce, Bigtable) provides us scalability, throughput, and reliability for operating big data.

We can see that the recent success of HPC and machine learning and AI have been promoting each other reciprocally and finally made the era of "big data, big compute, and artificial intelligence." However, determining how to achieve high performance and portability remains difficult for developers without domain knowledge. Currently, we see that heterogeneous systems with accelerators (GPUs, co-processors, APUs, TPUs, FPGAs, etc.) are prominent on the Top500 list [25]. Supercomputers are becoming more and more complex to speed up different computational requirements. Accelerators have been widely used in recent computational tasks including scientific simulation, image processing, finance strategy prediction, data mining, deep learning, and reinforcement learning. Due to their huge performance/power efficiency benefit, a supercomputer may contain multiple different accelerator chips in one system. Mostly, operating systems treat these accelerators as input/output(I/O)

devices, which means these accelerators own their high-bandwidth memory. High-bandwidth memory chips are usually expensive and have limited capacity. Current computing tasks, especially simulation, machine learning, and deep learning tasks usually require a large size of input data, which can not fit into these high-bandwidth memory chips. Determining how to move these data efficiently between the host and device could be significantly important to get optimal performance. On the other side, the CPU needs to take responsibility for copying the input data to the accelerators, launch jobs to the accelerators to process these data, and then copy the results back to the main host memory to output. The CPU host memory and device accelerator memory is usually connected by Peripheral Component Interconnect Express (PCI-e), which is a high-speed serial computer expansion bus standard. Both computation power and data transfer bandwidth are important to achieve high-performance improvement while using accelerators.

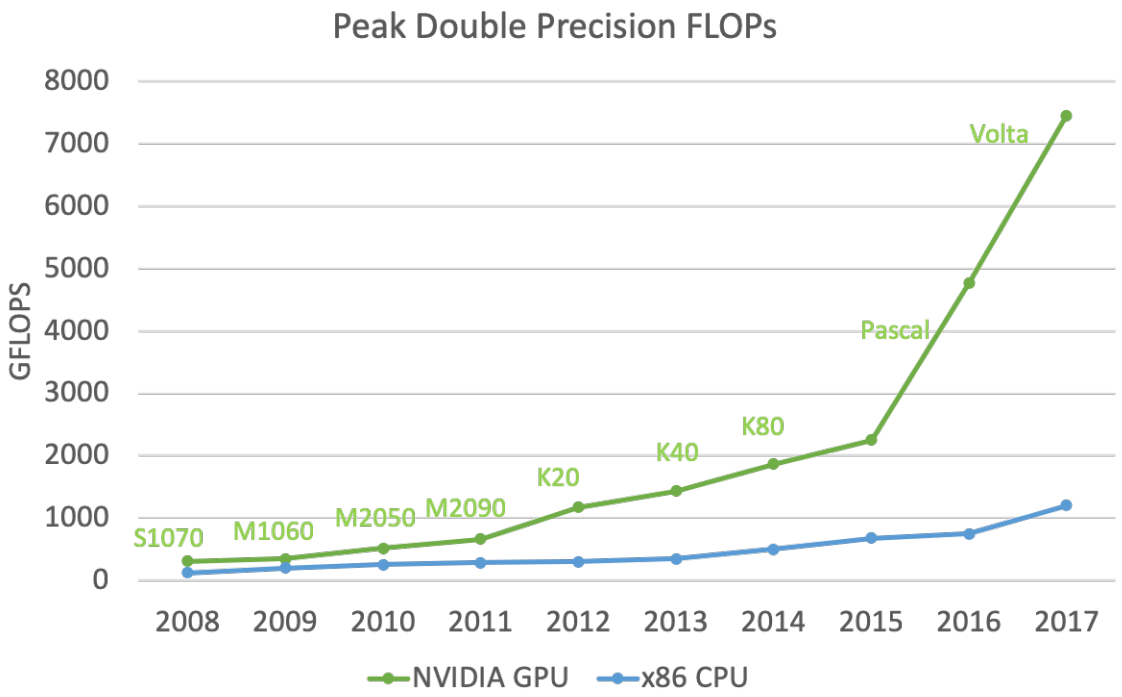

Figure 1.1: GPU Double-Precision Peak FLOPs (data from [56])

Figure 1.1 shows the peak double-precision FLOPs of Nvidia GPUs and CPUs since 2008 when the first few GPUs were released for general-purpose computing. We can observe that over the years, the GPU computation power grows significantly better than CPUs. The first GPU designed for server, Tesla S1070 (released on 2008) could achieve 311 GFLOPs performance on double-precision operations. Over the years, the GPU performance increased significantly. We also saw a huge increase around 2015. We think it probably was because of the high demand for computational power from the popular machine learning and AI tasks, which pushes the communities to grow even faster. We can also see that the most recent GPU in the Tesla product line, Tesla V100 can achieve 7450 GFLOPs. This means 24× performance improvement in the last 10 years.

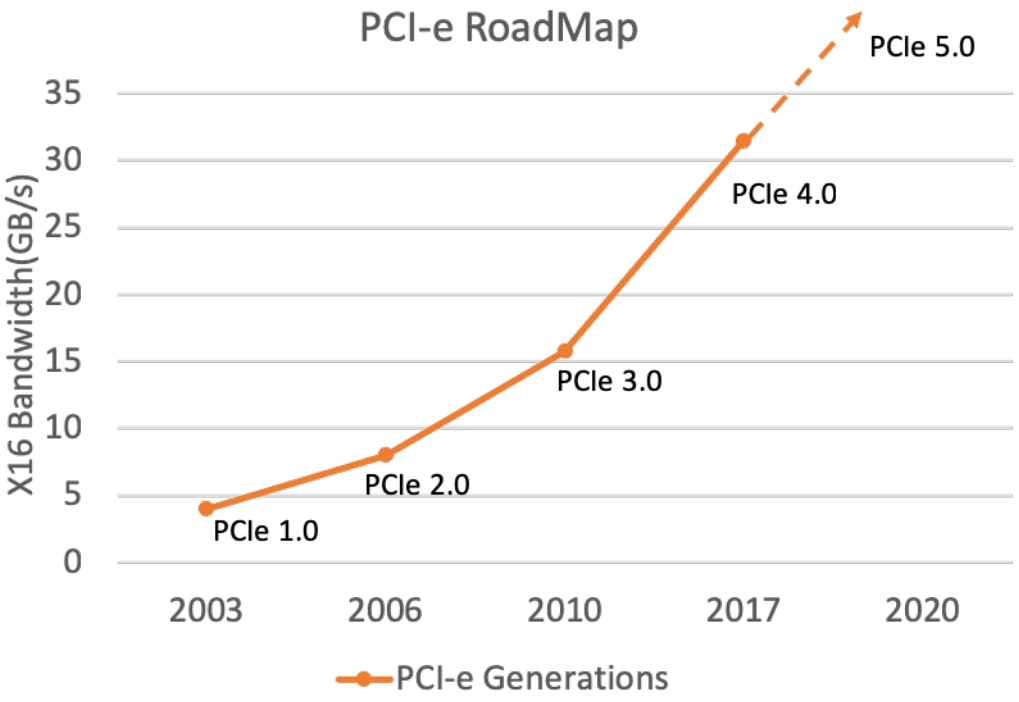

Figure 1.2: PCI-e generations (data from [61])

Figure 1.2 shows the peak data transfer bandwidth provided by the PCIe connection between host memory and device memory. The first generation of PCIe (1.0) was released in 2003.

PCIe Version	x1 Bandwidth	x4 Bandwidth	x8 Bandwidth	x16 Bandwidth
1.0	250 MB/s	1 GB/s	2GB/s	4 GB/s
2.0	500 MB/s	2 GB/s	4 GB/s	8 GB/s
3.0	984.6 MB/s	3.938 GB/s	7.877 GB/s	15.754 GB/s
4.0	1.969 GB/s	7.877 GB/s	15.754 GB/s	31.508 GB/s

Figure 1.3: PCI-e bandwidth details (data from [61])

Up to today, it has already been evolved by three generations. However, even though the recent release is PCIe 4.0 in 2017, most servers are still equipped with PCIe 3.0 to provide better compatibility to other devices. Figure 1.3 shows the peak transfer bandwidth provided by PCIe. We observe that PCIe 1.0 provides 4-GB/s bandwidth while PCIe 4.0 provides 31.5-GB/s bandwidth, approximately an 8X bandwidth improvement in 14 years.

We can clearly observe that computation power has grown significantly faster than the data connection during the past 10 years. Moreover, we also observe computing patterns (e.g, convolution, pooling, transpose) that cost as much time to compute as that of data movement. The data movement overhead could potentially become the bottleneck of the accelerators, especially in the near future. Therefore, efficiently handling data movement for accelerators is very important.

On the other side, the burden of the CPU increases a lot, due to the increasing number of accelerators. Since the systems treat these accelerators as I/O devices, the CPU has to handle the data movement, task launching, and even synchronization problems. This significantly increases the CPU resource usage, harming other tasks that might be handled on the CPU side. How to bypass the CPU control in most operations of the accelerators, thus releasing as much CPU resources as possible, has become a popular topic in recent years [63, 83].

While many new programming models are proposed to support accelerators, rather than

grapple with the unfamiliar, scientists often prefer to keep their existing verified C, C++, or Fortran code. Since version 4.0 [52, 58], OpenMP has provided directive-based accelerator support that is similar to OpenACC [86] and provides a straightforward way to adapt existing programs to these systems. These models provide multiple options to pipeline data but still have drawbacks in terms of performance, programmability, and memory management.

Even though "big data" is bringing us "magic" projects that improved our research methods using high-performance computing, we can see that the significantly increased data size is becoming a big hurdle to the development of these systems. It may become the bottleneck of overall performance even with faster computation power. Developers need a better way to store and move the data efficiently. The community needs a better mechanism for accelerators to achieve better overall performance, programmability, and portability.

This dissertation aims to provide efficient data transfer pipelining frameworks and extensions to help developers overlap kernel computation with data transfer in an efficient, heuristic, and easy-to-use way. Since 2013 [57], OpenMP has provided a straightforward way to adapt existing programs to accelerated systems. We aim to extend the current OpenMP standard to provide better data pipelining approaches for accelerators in terms of programmability, performance, and memory management. Detailed work will be discussed in the following chapters.

1.2 Directive-based Partitioning and Pipelining for Graphics Processing Units

In directive-based offload models, users annotate their data with mapping or copying directives to ensure that the accelerator can access the data. They then launch computation

on the accelerator and ensure that the results are available on the host when needed. If the accelerator cannot directly access host memory or if using the accelerator memory can improve performance, the data is copied to the device memory. The naive offload model synchronously performs these copies, which can take significant execution time. Furthermore, the data may not fit in the device memory because scientific applications frequently use huge data arrays or matrices. However, overlapping the transfers with computation and/or partitioning the data and computation can require extensive coding changes. To offload data in directive-based models, we associate the existing host variable name with a corresponding variable of the same name on the device. The result is that only one such mapping can exist for a given name or for a name and a slice range for arrays (in which case they must not overlap). Thus, users have limited control of memory mapping, partitioning, and pipelining with this mechanism.

To tackle these issues, we propose a directive-based pipelining extension for OpenMP specification on GPUs. Our extension supports automated data partitioning and overlapping of transfers through pipelining. It allows data to be mapped into a smaller buffer to reduce memory use and offers a simple interface to pipeline a parallel loop with an index handler and a kernel scheduler. We also provide a detailed evaluation including a comparison with state-of-the-art implementations.

1.3 Block-Level Data Pipelining for Graphic Processing Units

OpenMP [24, 52, 58] currently supports the data copy and computation pipeline in an asynchronous way but requires users to modify their code manually. Users must split the

task into chunks and launch multiple sub-kernels with different GPU streams. This manual task partitioning and sub-kernel launching entail extra code re-factoring and may introduce programming errors. Furthermore, splitting code to process multiple chunks causes extra function-call overhead. The hyper-parameters, e.g., #streams and #chunks, must be carefully tuned to achieve optimal performance; otherwise, poor choices may significantly harm performance. Moreover, as the growth of heterogeneity on supercomputers, systems are becoming more and more complex with various accelerators, co-processors, and other devices with discrete memory. These devices drastically increase the burden of the CPU to cooperate with all kinds of drivers for the data movement among these discrete memory spaces. Bypassing the CPU is an efficient way to deal with offloading tasks on accelerators [43, 81, 83, 88]. However, traditional kernel-level data pipelining still requires the CPU to be involved in the data movement, sub-kernel launch, and task dependency control between each task, with significant CPU resource utilization. The GPU-bound processes may hurt CPU processes and CPU-bound processes may hurt GPU performance in the other ways [69].

We propose a block-level pipelining mechanism to resolve these issues: Our block-level pipelining performs data communication and computation inside one GPU kernel and can totally bypass CPU control after this single kernel launch. The task dependencies between sub-tasks are maintained by flag arrays on the GPU and the sub-tasks are executed in the topology order of the program, with no extra requirement on developers to serialize the tasks to map to traditional GPU streams. We also implemented block-level pipelining mechanism in OpenMP by introducing a directive-based pipeline syntax to provide better programmability for block-level pipelining programming. We completed a detailed evaluation using our approach for various benchmarks, as well as providing a detailed comparison between our work and the current state-of-the-art implementations.

1.4 IterML: Iterative Machine Learning for Intelligent Parameter Pruning and Tuning in Graphics Processing Units

With the rise of graphics processing units (GPUs), the parallel computing community needs better tools to productively extract performance from the GPU. While modern compilers provide flags to activate different optimizations to improve performance, the effectiveness of such automated optimization has been limited at best.

Consequently, extracting the best performance from an algorithm on a GPU requires significant expertise and manual effort to exploit both spatial and temporal sharing of computing resources. In particular, maximizing the performance of an algorithm on a GPU requires extensive hyperparameter (e.g., thread-block size) selection and tuning. Given the myriad of hyperparameter dimensions to optimize across, the search space of optimizations is extremely large, making it infeasible to evaluate exhaustively.

As we mentioned, machine learning and AI have been successful these years. It has been changing all other domain areas including HPC, not only improving the consumer market of HPC, but also affecting the research and development of HPC. Most machine learning and AI tasks contain two stages, training and inference. Both stages require a large amount of data flow to go through the same neural network model iteratively, which means the computation patterns do not change through the process. This feature makes it possible to adjust the hyperparameters of the hardware accelerators during the iterations to achieve better performance, which could also be implemented using machine learning methods.

We propose an approach that uses statistical analysis with iterative machine learning (IterML) to prune and to tune hyperparameters to achieve better performance. During each itera-

tion, we leveraged machine-learning models to guide the pruning and tuning for subsequent iterations. We evaluated our IterML approach on the GPU thread-block size across many benchmarks running on an Nvidia P100 or V100 GPU. Our experimental results showed that our automated IterML approach reduced search effort by 40% to 80% when compared to traditional (non-iterative) ML and that the performance of our (unmodified) GPU applications can improve significantly — between 67% and 95% — simply by changing the thread-block size.

1.5 Dissertation Organization

The remaining dissertation chapters cover the following topics. Chapter 2 shows the background information of parallel accelerators, programming models and memory management techniques. It also provides the related work of existing solutions to achieve high performance data movement between discrete memory space (between CPUs and Accelerators) on supercomputers. Chapter 3 presents our directive-based partitioning and pipelining for graphics processing units. Chapter 4 covers our block-level data pipelining for graphic processing units work. Chapter 5 elaborates our performance auto-tuning work with machine learning methods. Chapter 6 summarizes the dissertation and future work.

Chapter 2

Background and Literature Review

In this chapter, we provide a brief background introduction regarding state-of-the-art heterogeneous computing and programming models for various accelerators. Related memory access techniques are also covered in this chapter. Overall, this chapter discuss the state-of-the-art research work related to this dissertation.

2.1 Programming Models of Accelerators

Supercomputers increasingly have accelerators, such as GPUs, FPGAs, APUs, and co-processors like the Intel Xeon Phi to increase their performance per watt and performance per dollar. Accelerators have been applied in many cutting-edge research areas, such as scientific simulations [60], medical solution development [94], machine learning [89], and regular expression [91, 92] matching. Programming these accelerators requires the use of alternate programming models or language extensions such as CUDA, OpenMP, OpenACC, and OpenCL.

OpenMP is a directive-based extension for Fortran, C and C++ that is best known for providing portable multithreading on shared-memory multicore systems. Insertion of an OpenMP directive can parallelize a loop. OpenMP 4.0 introduced device constructs that *target* offload to devices with potentially distinct memory spaces. OpenMP support for accelerators is still relatively nascent and, so, offers opportunities for improvement [20, 21,

34, 74].

OpenACC is a directive-based model that Cray, CAPS, Nvidia, and PGI developed specifically to address heterogeneous CPU/GPU systems. As with OpenMP, a user can annotate C, C++ and Fortran code with compiler directives to identify areas that should be accelerated. Current OpenACC implementations are more mature than existing support for OpenMP 4.X. Thus, several studies have compared its directive-based approach to CUDA in terms of performance, portability, and programmability [45, 84].

CUDA is an application programming interface (API) that Nvidia created. It supports general purpose processing on CUDA-enabled GPUs – an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). CUDA is designed to work with programming languages such as C, C++, and Fortran. It provides a powerful, flexible API with low-level GPU control. GPU threads are grouped as a grid of thread blocks, which are mapped to GPU streaming multiprocessors. CUDA often requires users to refactor their code significantly [11, 31, 34, 36, 90, 93].

OpenCL (Open Computing Language) [77] is a standard, low-level model similar to CUDA that the Khronos group maintains. OpenCL implementations offer portability across GPUs, multicore CPUs, DSPs, co-processors, and FPGAs. However, OpenCL's complex, low-level API, often requires significantly more code than even CUDA.

2.2 Memory Access of Accelerators and Data Pipelining Techniques

In this section, we will cover a few new features and techniques of memory access on accelerators and related data pipelining techniques.

2.2.1 Unified Virtual Addressing

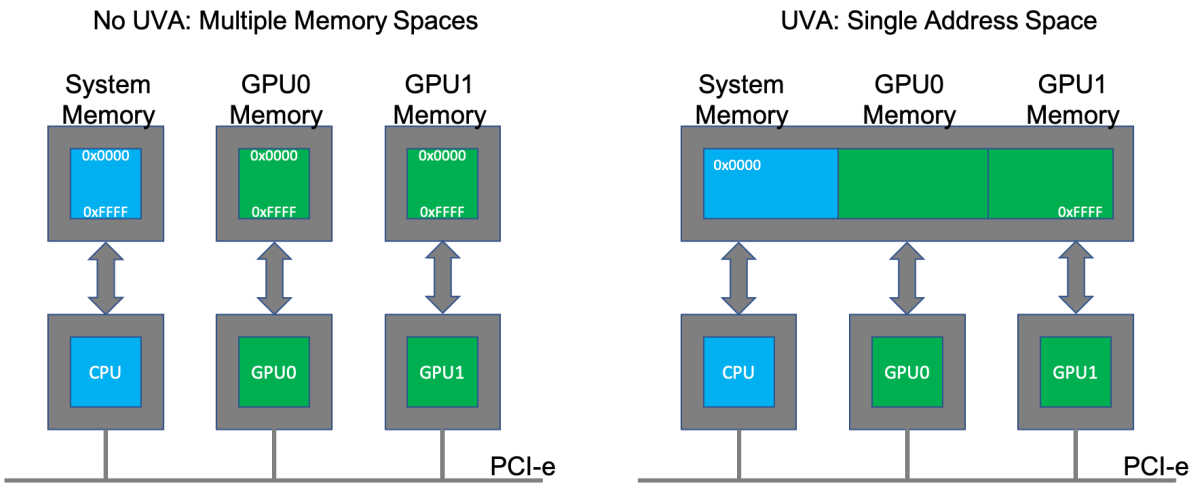

Figure 2.1: Unified Virtual Addressing

Unified Virtual Addressing (UVA) [56] was first introduced with CUDA 4. It is a memory address management system that enables Fermi and Kepler GPUs running 64-bit processes. UVA means that a single memory address is used for the host and all devices as shown in Figure 2.1. The physical memory location can be determined by the pointer itself. It also enables libraries to simplify their interfaces (e.g. cudaMemcpy). UVA also enables the Zero-Copy Memory feature, which means Pointers returned by cudaHostAlloc() can be used directly from within kernels running on UVA enabled devices (i.e, no need to obtain a device pointer via cudaHostGetDevicePointer()).

2.2.2 CUDA Streams

A CUDA Stream is a sequence of operations that execute on the device in the order in which they are issued by the host code. While operations within a stream are guaranteed to execute in the prescribed order, operations in different streams can be interleaved and, when possible, even run concurrently. As long as the device is capable of ”concurrent copy and execution,”

developers may launch the kernel computation and the data transfer in different, non-default CUDA streams asynchronously. The tasks should also be first partitioned to multiple chunks, then assigned to different GPU streams, which may entail extra code refactoring that often introduces significant programming errors.

2.2.3 Unified Memory

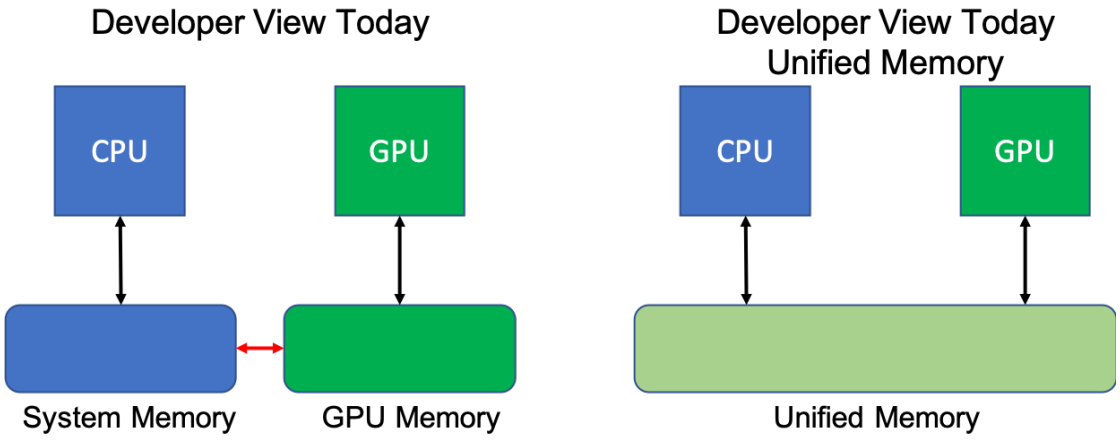

Figure 2.2: Developer View with/out Unified Memory

Unified Memory (UM) [56], as shown in Figure 2.2, which was introduced in 2014 with CUDA 6 defines a managed memory space in which all processors see a coherent memory address space. This hardware/software technology allows applications to allocate data that can be read or written from code running on either CPUs or GPUs (often called CUDA managed data). CUDA system software and/or the hardware automatically migrates memory pages to memory that the device can access. The Pascal GPU architecture is the first system with hardware support for virtual memory page faults and migration. Older GPUs based on the Kepler and Maxwell architectures support a more limited form of UM [49]. Since CUDA 9, the runtime supports a few optimization functions for Unified Memory to be more efficient in terms of data movement between host memory and device memory (e.g.,

prefetching). Nvidia claims that the data transfer occurs on demand for both GPU and CPU, requiring the use of only one array of data and no need for duplicate pointers as data is transferred between the two processors. Furthermore, they emphasize that this new method simplifies the programming model, as well as enables close to maximum bandwidth for the data transfer. On-demand migration is powerful in the way it enables fine-grain overlap between data transfers and kernel execution. However, on-demand data migration will never beat explicit memory copies in terms of large data regions. If the application's access pattern is well defined and structured, developers may use the prefetching technique of Unified Memory to tile the data and to overlap the data movement with computation kernel to achieve optimal performance [55].

2.3 Related Work

We first proposed an extension to support compiler-implemented pipelining for data transfer and compute overlap in directive-based models such as OpenMP [58] and OpenACC [86]. Previous work [21] presents some of the benefits of this approach. While double buffering, and pipelining in general, is a common manual optimization, it is not a common facility of either production programming models or research prototypes although some researchers have explored mechanisms that could show support.

Higher-level logically global models like Legion [6] encode the structure of their data and computation as part of the base model and can apply optimizations like those that we discuss in their runtimes. The challenge is that they cannot be incrementally applied to existing codes, thus requiring significant refactoring if not rewriting. Similarly, global models like Chapel [15] could logically support optimization of abstract loop computations through custom domain maps and other policies, but existing codes must be significantly modified

in order to use them.

Recent studies on MPI libraries such as MVAPICH2 [81, 82, 83], MPICH2 [2, 39], and OpenMPI [88] provide support to pipeline data transfers between PCIe with the data transfer on high performance interconnects to optimize bandwidth. Some of the custom data-type facilities of these libraries provide similar specification facilities to those that we propose, but differ in that they represent the data type as a whole rather than as something tied to computation and thus indexable as part of one. MPI-ACC [1] is an integrated and extensible framework. It allows end-to-end data movement in accelerator-based systems. The runtime system provides good performance by integrating support for an accelerator memory space into MPI with several optimizations, such as pipelining and parameter tuning.

On the system level, studies such as ADSM [27], CGCM [38], Spark-GPU [97], and RSVM [40] provide compiler-based optimizations for data management and movement between CPUs and GPUs, depending on static or dynamic compile-time analysis or on programmer supplied annotations. Instead of only focusing on data movement and management only, our research work focuses more on the techniques to overlap the data movement with computation efficiently.

Existing literature [66, 67, 68] has studied and tackled the data movement problem—specifically the file I/O bottleneck—in large-scale deep learning systems. We can observe that data movement plays an important role in providing good overall performance. These research studies inspired some of our research work to focus on data movement to prevent it from becoming the hurdle of computation.

The Nvidia's Unified Memory (UM) mechanism might impact performance, compared to a standard implementation without it. Typically, solutions that increase flexibility and ease of programming impose some performance overhead. Li et al. [49] evaluated UM by comparing

a selected set of applications with and without UM run on the Nvidia K40 and Jetson TK1 GPU platforms. The applications tested were Diffusion3D Benchmark, Parboil Benchmark Suite and Matrix Multiplication. This paper shows that Unified Memory versions cause 10% performance loss on average for these benchmarks. They also validated that the performance loss is caused by redundant memory transfers and page faults when adopting the Unified Memory programming model. This performance loss turned out to be the cost for an easier programming model. Landaverde et al. [47] investigate the performance and behavior of UM on a variety of common memory access patterns, especially the communication behavior between a host CPU and GPU. They develop multiple customized microbenchmarks for the GPU architecture with the Rodinia benchmark suite. They categorize the benchmarks by their behaviors and apply UM to these benchmarks and evaluate the change in performance. They find that for the vast majority of applications, UMA generates significant overhead and results in notable performance loss. Furthermore, the UMA model only marginally simplifies the programming model for most applications. However, for individual tests, it was demonstrated that application of UM may bring performance benefits. Specifically, if a subset of data is queried by multiple kernels multiple times before some other data are accessed. The authors state that in such case the UM mechanism can place data favorably which brings benefits compared to the standard API.

CoreTSAR [72, 73] explores automated coscheduling between devices with potentially disjoint memory spaces. CoreTSAR uses mapping functionality that associates data to computation along a single dimension for certain specific patterns. Our specifications take similar information to the array association pattern employed by CoreTSAR. However, CoreTSAR uses this information to divide computation across devices rather than to overlap computation and communication and to reduce memory use. Task-based models like OmpSs [14] and StarPU [5] construct graphs of “tasks” composed of statically sized chunks of data and

computation, which are then scheduled. A user can achieve overlap by subdividing a given loop into multiple tasks, as long as they select the size and translate addresses manually.

Bauer et al. [7] study CudaDMA, an extensible API for efficiently managing data transfers between the on-chip shared memory and off-chip global memory of CUDA-enabled GPUs. They partitioned the warps in a thread block to compute warps and DMA warps. Both sequential and stride DMA patterns are supported to improve flexibility. The approach achieves good speedup (3.2x) for the SGEMV benchmark with small matrices due to the increased sustained memory bandwidth. Results with larger matrices or other benchmarks show 0.87X to 1.15X overall speedup because the benefit from reduced register pressure was not enough to overcome the overhead of the additional CudaDMA transfer through shared memory.

Aji et al. [3] proposed a scheduling attribute to OpenCL context and the runtime could automatically generate an ideal queue for different tasks. Part of our research work related to "dynamic queue" was inspired by this work.

Several papers focus on improving the computational task execution pipeline to utilize the hardware resource better. Wu et al. [87] proposed an SM-centric scheme for task scheduling. The tasks that a thread executes is based on the ID of the SM on which the thread runs. The approach provides more opportunities for optimizing GPU programs with scheduling strategies. Some of the research work in this dissertation is using similar techniques during the Persistent Thread Blocks implementations.

Zheng et al. [98] proposed VersaPipe, which leverages the combination of persistent threads and the SM-centric mechanism [87]. VersaPipe supports coarse-grain pipelining, fine–grain pipelining and a hybrid pipeline mode that assigns different pipeline stages to different SMs to utilize SM resources fully. It significantly improves the computation performance of

several applications including Pyramid, face detection, CFD, image rasterization and LDPC. Belviranli, et al. [8] introduced Juggler, a dependence-aware, in-device task execution runtime for GPUs. The runtime uses in-GPU dependence resolution and task placement. They also implemented a compiler transformation to integrate their runtime into OpenMP 4.5.

The Whippletree Megakernel [76] inherits some traits from persistent megakernels [46]. Unlike traditional persistent megakernels, Whippletree assigns available threads to work on the incoming tasks by implementing a dynamic queue and a new scheduling policy. It provides an abstraction to implement complex software pipelines, recursive algorithms and many other applications. The megakernel implementations inspired our implementation to use Nvidia's cooperative thread groups while implementing the block-level data partitioning and pipelining.

Nowadays, accelerators other than GPUs are also widely used. Bo et al. [10] proposed automata identify gRNA off-targets in Bioinfomatices and achieved over 10x speedups compared with the state-of-the art approaches. They also found that using spatial architectures (FPGAs and Micron's Automata Processor) could provide additional speedups. ANMLZoo and AutomataZoo [79, 80] are two benchmark suites that are designed for evaluating and validating automata processing engines. They are composed of various applications from different domains besides regular expressions. They also provide sample input files including the automata description files and the input streams. Researchers working on network or system security problems are also leveraging GPUs to seek better and more efficient solutions [95, 96]. We observe that the recent success of GPU has drawn more attention from researchers in various of areas. This observation also motivates some of the work in this dissertation to provide better programmability, performance and data management.

Choi et al. [18], Li et al. [51], and Tran et al. [78] auto-tune the performance of a particular algorithm or application on accelerators. However, their auto-tuning still requires exten-

sive expertise (or intuition) to select the key parameters *manually* as well as the compiler flags. To address this problem, we propose an approach that automatically identifies a much smaller (pruned) search space that contains a (near-)optimal setting, which can then be searched. Cui et al. [22, 23] utilized linear regression and decision tree approaches to help with tuning on accelerators. Specifically, given a large search space, our iterative machine-learning (IterML) approach gathers information during each iteration, builds models, and finds the best interaction between the parameters and performance. This, in turn, provides automated guidance as to how to tune performance in the context of a large parameter search space.

Various statistical or machine-learning (ML) methods have been applied to help with auto-tuning parameters to get better performance. Joseph et al. propose a linear regression model to predict processor performance based on micro-architectural parameters [42], however, it requires a large amount of processor profiling data as input to build the linear model. Li et al. utilized deep-reinforcement learning approaches to identify the optimal values of tunable parameters in computer systems — from a simple client-server system to a large data center [50]. While this approach can be deployed into a production system to collect training data and suggest tuning actions during the system's daily operation, it requires the system to be mostly static — less applicable to new algorithms or libraries that target new devices like GPUs. There are pheromone models based on the profiling data of GPUs [32, 71], but these models require large training sets across many programs and with a wide variety of performance counters. Moreover, they require developers to have intimate knowledge about the programs. Other related research focuses on designing coding machines to handle the programming tasks [48]. In contrast, our goal in this dissertation is to help developers productively tune their programs to achieve near-optimal performance with the least amount of effort and domain knowledge.

Chapter 3

Directive-based Partitioning and Pipelining for Graphics Processing Units

3.1 Introduction

Systems with accelerators, particularly GPUs, are now prominent on the Top500 [25]. While many programming models support these systems, rather than grapple with the unfamiliar, scientists often prefer to keep their existing verified C, C++ or Fortran code. OpenMP version 4.0 [52, 58] introduced directive-based accelerator support that is similar to that provided in OpenACC [86] and that provides a straightforward way to adapt existing code to use these systems.

In directive-based offload models, users annotate their data with mapping or copying directives to ensure that the accelerator can access the data. They then launch computation on the accelerator and ensure that the results are available on the host when needed. If the accelerator cannot directly access host memory or if using the accelerator memory can improve performance, the data is copied to device memory. The naive offload model synchronously performs these copies, which can take significant execution time. Furthermore, the data may

not fit in device memory because scientific applications frequently use huge data arrays or matrices. However, overlapping the transfers with computation and/or partitioning the data and computation can require extensive coding changes.

To offload data in directive-based models, we associate the existing host variable name (e.g., of an array) with a corresponding variable of the same name on the device. Result is, only one such mapping can exist for a given name, or a name and a slice range for arrays in which case they must not overlap. Thus, users have limited control of memory mapping, partitioning and pipelining with this mechanism.

Our extension to this model supports automated data partitioning and overlapping of transfers through pipelining. It allows data to be mapped into a smaller buffer to reduce memory use and offers a simple interface to pipeline a parallel loop with an index handler and a kernel scheduler.

We evaluate our partitioning and pipelining extension using four applications on multiple accelerators. Our results show that it significantly reduces memory use and improves performance. We also compare it to CUDA 9 Unified Memory (UM) [30], which supports GPU page faults with optimizations like prefetching and duplication, thus offering an automated alternative for memory oversubscription, on the Pascal P100 GPU [13] using multiple data sets. Our extension outperforms UM in most cases, especially when the data set size significantly exceeds the GPU memory limit.

This chapter makes the following contributions:

- A new directive-based pipelined extension that automates the overlap of data transfers and kernel computation and reduces GPU memory use;
- A prototype of the proposed extension;
- Evaluation of our prototype of the extension with four applications on Nvidia K40m,

P100, and AMD Radeon HD 7970 GPUs;

- Evaluation of CUDA-9 Unified Memory and its optimization methods and a comparison to the performance with our framework on the P100 GPU.

3.2 Pipeline Extension API and Usage

Figure 3.1 presents the syntax of our extension. The `pipeline` map-type modifier extends the semantics of the `map` clause, which makes all data available at the beginning and/or at the end of the region. Our modifier splits the data updates and subsequent loop computation into multiple subtasks.

The *list_items* parameters to map have a new option to take an expression based on an iterator as their *begin* parameter. The format of this parameter is `<var>[(iterator_expr|begin):len]`. The `<var>` is the variable or base pointer of an array. The `[iterator_expr:len]` parameter identifies the dimension to split and size required for the given iteration to function. The function defines the split starting offset in that dimension while the `len` defines the range. The split currently can be performed in one or two dimensions since our runtime system supports 1D and 2D memory copies. This `len` parameter helps us determine the array offset. We use different internal APIs for data movement based on the subsequent loop, which we discuss later.

The `chunk_size` is the number of indices in the subsequent loop that we handle in each device buffer (potentially fewer in the last chunk). The `overlap` parameter determines the number of GPU streams. This parameter determines the number of chunks that we launch asynchronously. We choose these two parameters as the key components of our framework not only because they provide information to improve the data transfer between host and device, but also because they can significantly affect performance and memory use.

The `overlap` and `chunk_size` parameters determine the size of the device buffer, which we tune before we allocate the buffer to fit the total memory usage within the available GPU memory size. The other target clauses, for example, `device` or `private`, work as previously.

```
#pragma omp target teams distribute
  map(pipeline(iterator[,chunk_size[,overlap]]):items…)
  iterator(ident = begin:end[:step])
```

pipeline() inputs

overlap	number of blocks or streams to overlap while copying
chunk_size	number of iterations in a chunk (possibly less in last)
iterator	iterator, defined on same directive, representing range and stride

map() inputs

list_items	array declaration *arr*[*iterator_expr[:len]* | *begin*:*len*]

iterator() inputs

ident	identifier for the iterator
begin/end/step	bounds and step of the iteration space

Figure 3.1: Our Proposed Pipeline Extension for OpenMP

Figure 3.2 shows a three-level nested loop that performs a stencil computation in which `pipeline(k,1,3)` sets `chunk_size` to 1 and the number of GPU streams to 3. The `to` specifies that the three-dimensional input array `A0` will be pipelined. `iterator(k = 1:nz-1)` indicates that the outermost loop is split and pipelined. Here we denote the outer loop variable as k. We use a function of k and `<num>` to indicate the data chunks that we must copy before launch of the kth chunk's kernel. For instance, the `[k-1:3]` indicates that we must copy the $k-1$, k and $k+1$ chunks in that dimension to the device before the kth kernel executes. The `[k-1:3]` in the first set of brackets on `A0` means that we split this array by its Z dimension. It defines the dependency relationship between the array and the outermost

loop. For example, before kernel iteration `k=t`, we must copy chunk `t-1`, `t`, and `t+1` of `A0` to device memory. The `[0:ny-1][0:nx-1]` defines the other dimensions of array `A0`. The `from` in the `map` clause defines the output array `Anext`. For this array, the `[k:1]` indicates that each iteration only stores its corresponding chunk. The `teams distribute parallel for` clause still parallelizes the nested loop i and j inside loop k.

```
#pragma omp target \
  map(pipeline(k,1,3),to:A0[k-1:3][0:ny-1][0:nx-1])\
  map(pipeline(k,1,3),from:Anext[k:1][0:ny-1][0:nx-1])\
  iterator(k = 1:nz-1) \
for(k=1;k<nz-1;k++) {
#pragma omp target teams distribute parallel for
  for(i=1;i<nx-1;i++) {
    for(j=1;j<ny-1;j++) {
      Anext[Index3D (i,     j,     k)] =
        (A0[Index3D (i,     j,     k + 1)] +
         A0[Index3D (i,     j,     k - 1)] +
         A0[Index3D (i,     j + 1, k)] +
         A0[Index3D (i,     j - 1, k)] +
         A0[Index3D (i + 1, j,     k)] +
         A0[Index3D (i - 1, j,     k)])*c1
       - A0[Index3D (i,     j,     k)]*c0;
    } }
}
```

Figure 3.2: A Stencil Benchmark Example

A powerful code analysis engine capable of deep analysis of code and dependencies [53] could significantly simplify our proposed extension. Potentially the compiler could determine the array definition information and even the data dependencies. However, the assumption of these capabilities would limit the applicability of our extension to code that can be analyzed completely at compile time and complicate its adoption into the OpenMP specification. Thus, our prototype allows all parameters to be passed explicitly.

3.3 Prototype Implementation and Technical Details

Our prototype runtime framework of our proposed extension splits each loop into configurable-sized chunks that are handled by different streams. Each chunk has data dependencies that

must be present on the device before its kernel executes. We map the data from the original data space to the buffer data space and copy each chunk to its corresponding location in the buffer. Currently, we use the `mod` operator (%) to get the offset of each chunk inside the buffer. For example, if we have a buffer that can hold four chunks, so it has positions 0, 1, 2, and 3, then, we copy chunk `i` to position `(i % 4)`. Once a data chunk is not needed for later partitions (kernels), we replace it. As long as the data is present, we schedule the corresponding subtask kernel to launch on the GPU.

Based on the stream numbers and array declaration information, we pre-allocate a fixed-size buffer. Each array defined by the `pipeline` modifier in the `map()` clause in the OpenMP region is associated with a data region. The array dimensions, `chunk_size` and `num_stream` determine the size of the device buffer for this data region. Once created, our runtime records the array's information for later use. We use a static pointer for the device buffer, which we allocate in GPU memory with `cudaMalloc()` on Nvidia devices. We use `cudaHostalloc()` to allocate pinned host memory, which avoids the data movement time from virtual to pinned buffer memory. The asynchronous memory copy is handled by `cudaMemcpyAsync()` for contiguous data movement; We also implement a 2D array interface using `cudaMallocPitch()` and `cudaMemcpy2DAsync()` to support non-contiguous data transfer. Currently, our prototype handles non-contiguous copies for 2D arrays, which means buffering a "Block" of a matrix. If `split_iter` is applied to both dimensions of a 2D array, we mark it as a 2D data region and record the corresponding information, e.g., `x_offset` and `y_offset`. Depending on the data dependencies of each subtask, we map the required data to this buffer and then pass the offsets in the buffer to the corresponding computation kernels.

To compare to Nvidia's Unified Memory (UM), we also need to apply UM to multiple benchmarks. We compile and link with the `-ta=tesla:managed` flag provided by the PGI compiler. This command line option allocates all dynamic memory in CUDA Unified (Man-

aged) Memory. To make sure there is no explicit data transfer, we also eliminate all data movement clauses and check the runtime information provided by the PGI compiler.

On Pascal and later GPUs, managed memory may not be physically allocated when allocation API is called; it may only be populated on access (or prefetching). In other words, pages and page table entries may not be created until they are accessed by the GPU or the CPU. So we also use `cudaMemPrefetchAsync()` to prefetch the data. To compare fairly, we make sure that the data are prefetched in the way that our pipelining does. In other words, the number of `cudaMemPrefetchAsync()` function calls is the same as that of `cudaMemcpyAsync()` for contiguous data movement. However, `cudaMemPrefetchAsync()` does not support non-contiguous data movement, we here simply prefetch the entire array.

We also notice that directive-based extensions are becoming more complex as we incrementally incorporate more functionality into the design to make it more powerful. On the other side, Unified Memory focus on reducing the complexity of the programmability while trying to offer compatible performance. However, Unified Memory requires underlying hardware/-software technology since it is transparent to the programmer. The goal of our work is to provide portable performance whether Unified Memory is available or not. In our experiments, which are discussed in the evaluation section, we also observe that the Unified Memory can provide good enough performance in some cases, especially when kernels are launched multiple times with little data motion. We are also considering implementing an auto-scheduler to help the developer to make choices under different circumstances.

3.4 Prototype Evaluation Results on Modern GPUs

In this section, we evaluate the performance and memory consumption of our proposed extension and compare to Nvidia's Unified Memory. We first evaluate our approach on four

benchmarks: a 3D Convolution benchmark; a stencil benchmark; a matrix multiplication benchmark; and a production Lattice QCD application. We run our experiments on three types of GPUs: Nvidia Tesla K40m, AMD Radeon 7970 and Nvidia Tesla P100.

Our Nvidia experiments run on one of two platforms: an x86_64 node containing two Nvidia Tesla K40m GPUs, each of which has 2880 stream cores and 12GB of on-board memory; and a Power-8 platform with NVLink containing four Nvidia Tesla P100 GPUs, each of which has 3584 stream cores and 16GB of on-board memory.

Our AMD experiments run on a node with an AMD Radeon HD 7970 GPU, which has 2048 stream processors and 3GB of on-board memory.

For each benchmark, we measure the performance in terms of the function that contains the GPU operations, including all transfers but ignoring time for code that is identical in all versions. We execute all test runs six times and use their average as the final result. The error in the performance numbers is within 0.2 seconds in all cases. For memory usage, we use the Nvidia System Management Interface `nvidia-smi` to inspect memory usage during execution.

3.4.1 Initial study of the pipeline technique

The naive offload model, i.e., synchronously copying and executing, is inefficient. Figure 3.3 shows a time distribution on an Nvidia K40m of different phases in a naive *Lattice Quantum Chromodynamics (QCD)* application written with OpenACC. Data transfers consume nearly 50% of execution time, during which no computation is performed. This execution model wastes GPU and CPU compute resources during data transfers. Thus, the current standard interface still has limitations in terms of performance, programmability, and memory usage. To understand the pipeline technique and the impact of stream counts and data sizes, we

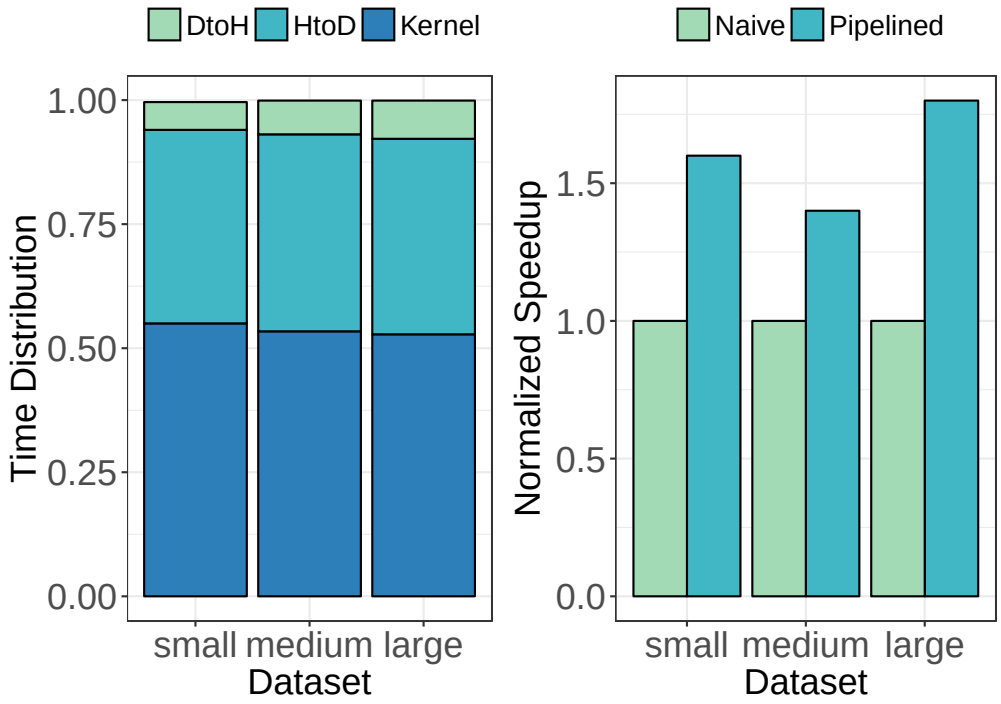

Figure 3.3: Lattice QCD Time Distribution (left) and Normalized Speedup (right) on Nvidia K40m

first use the Lattice QCD application as a case study on the Nvidia Tesla K40m GPU. From Figure 3.3, we observe that pipelining achieves a 1.6× speedup for the small test case. As the problem size grows, the speedup increases, indicating that larger cases may approach the theoretical upper bound of 2×, which would be achieved if data transfers and computation were perfectly overlapped.

We also vary chunk size and number of streams in Figure 3.4. These two parameters can significantly affect performance. The number of streams value is the number of GPU streams that we use in parallel, which is the number of transfers and kernels that may simultaneously be in flight. More GPU streams could potentially hide more "bubbles" in the pipeline. However, more GPU streams requires more scheduling overhead. As we divide the task into multiple chunks, chunk size determines the size of each chunk and, thus, the number of chunks. More chunks requires more API calls, and thus more overhead. Few chunks

mean that the transfers of the first input chunk and the last output chunk, which cannot be overlapped, are larger and account for more of the runtime which can degrade performance. Thus, we vary these parameters to explore the trade-off.

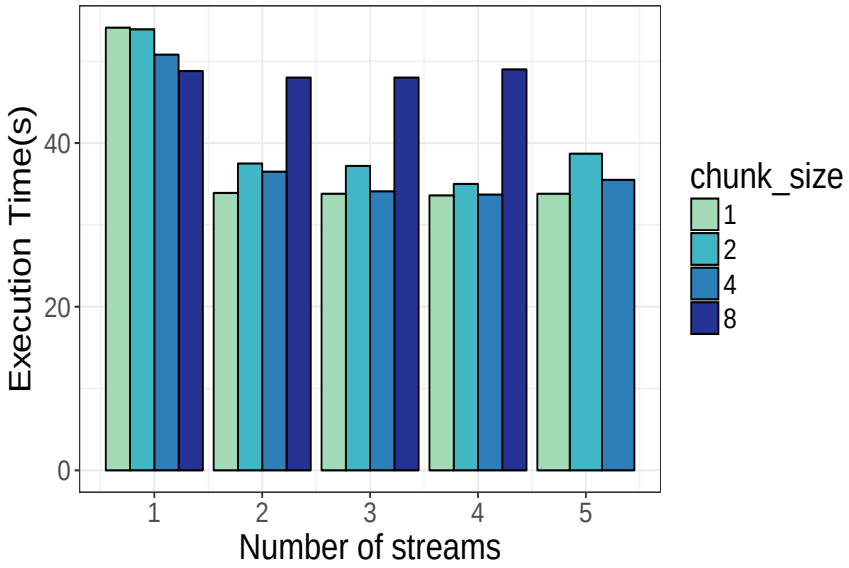

Figure 3.4: Different Chunk Sizes and GPU Stream Counts on Nvidia K40m

Figure 3.4 shows the results for the large test case on the K40m. Using two streams generally performs significantly better than one, showing the benefits of overlapping data transfers and computation. However, using more than four streams offers no further benefit due to increasing API and scheduling overheads while only slightly increasing potential overlap. Increasing the chunk size reduces API call and kernel launch overhead but makes load balancing harder. Increasing the chunk size usually does not adversely impact performance. Thus, we can ignore the additional overhead to use more chunks for this case. The K40m needs two streams to reach its best performance for this application.

3.4.2 Performance and Memory Evaluation

We evaluate our framework on four benchmarks on Nvidia and AMD GPUs. We denote the baseline version as "Naive". We overlap the data transfer and kernel computation and implement a "Pipeline". We then apply our framework to these benchmarks thus implement a "Framework". We have to note that the "Pipeline" version of the Matrix-Multiplication benchmark also benefits from the algorithm change. To utilize the idea of saving memory better, we replace the Naive Matrix-Multiplication algorithm by a Block Matrix-Multiplication.

3D Convolution

Many science and engineering applications use convolutions on multi-dimensional periodic data. Applications include deconvolving blurred images, signal and image processing, noise suppression, feature extraction, wave properties modeling and many others [12, 14, 28]. More specifically, convolution is used as a processing step in many electromagnetic (EM) problems involving scattering and radiation. Moreover, convolution neural networks are becoming important building blocks for most of deep learning networks [44]. Thus, fast convolution computation is becoming increasingly important for the training speed of deep learning tasks [17]. We use the 3D Convolution benchmark from the Polybenchmark set as an example on which to evaluate our approach.

Figure 3.5 presents its performance on the K40m. The Pipeline version achieves 1.45× speedup over the Naive version. Our prototype also delivers 1.46× speedup over the Naive version, which provides exactly the same performance compared to the hand-coded Pipeline version. Thus, the benefit from overlapping data transfer and kernel computation is still much larger than the overhead from extra index translation and API calls.

Figure 3.6 shows the memory usage across versions. Since the default test case of the

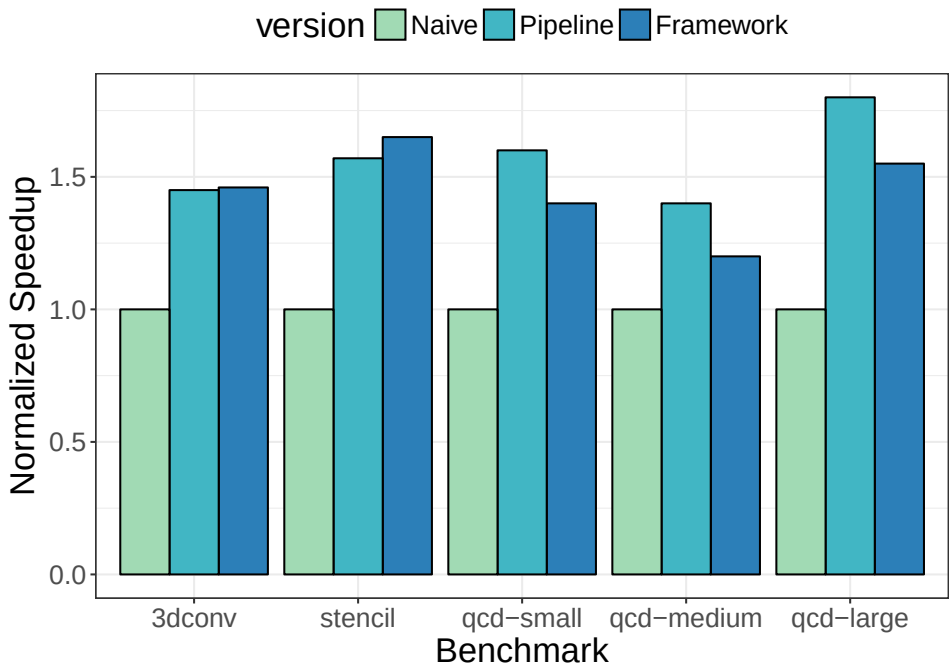

Figure 3.5: Performance Evaluation on Nvidia K40m

benchmark is relatively large, the Naive and Pipeline versions require about 3.5 GB of GPU memory. Our Framework version only consumes 93 MB of GPU memory, which means we could save 97% of device memory. With this huge memory savings, we could potentially run much larger datasets or keep other useful data structures in device memory for a larger application.

Figure 3.7 shows that the number of overlapping streams affects the performance of the Pipeline version. However, using two streams no longer delivers the best performance; we instead need up to eight streams to achieve the best performance. As our results show with our other benchmarks, the number of streams can significantly affect performance, but the ideal number of streams varies across benchmarks. We also find that our prototype uses slightly more memory as the number of streams increases, because we must pre-allocate a larger buffer as we increase the number of streams. Still, we reduce memory use 96% even with eight streams.

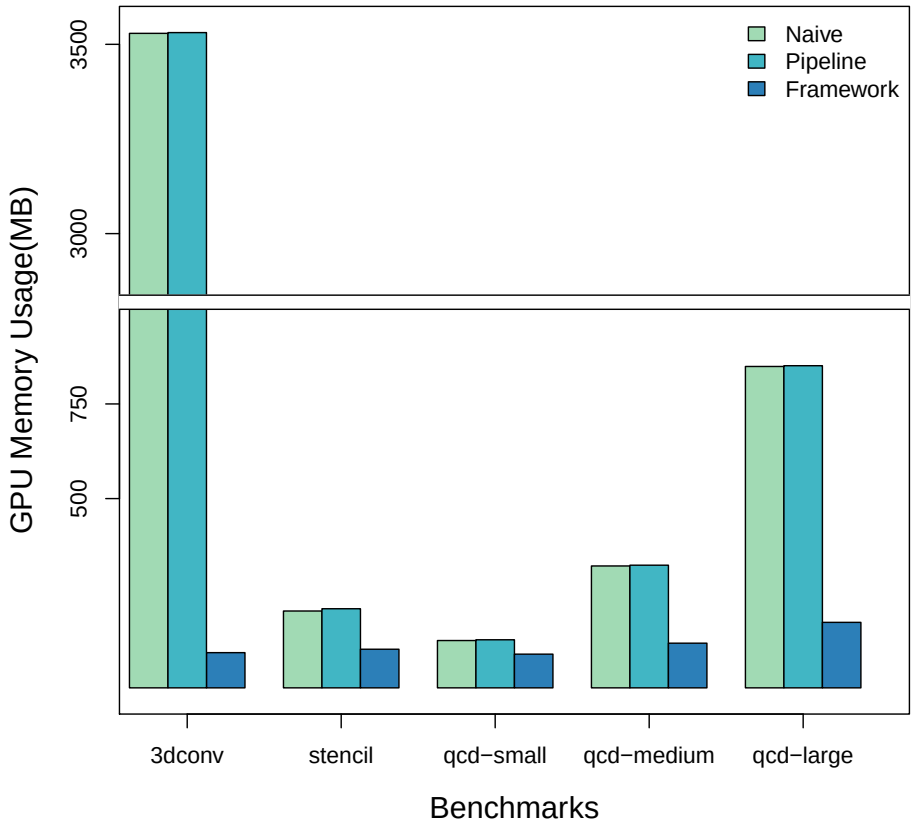

Figure 3.6: GPU Memory Usage Evaluation on Nvidia K40m

Figure 3.8 (left) shows the performance of the 3D Convolution benchmark on the AMD Radeon 7970 GPU. We first compare the Naive version with the Pipeline version. The Pipeline version is 57% slower than the Naive version, which is significantly different from our Nvidia K40 results. To understand this difference, we use the AMD APP Profiler to profile the Pipeline version, which reveals that data transfer times lead to the significant performance degradation. Although the data volume that is transferred is the same, the Pipeline version takes much longer to move it: the transfer rate for the Naive version is about 6 GB/s while it is only 2 GB/s for the Pipeline version.

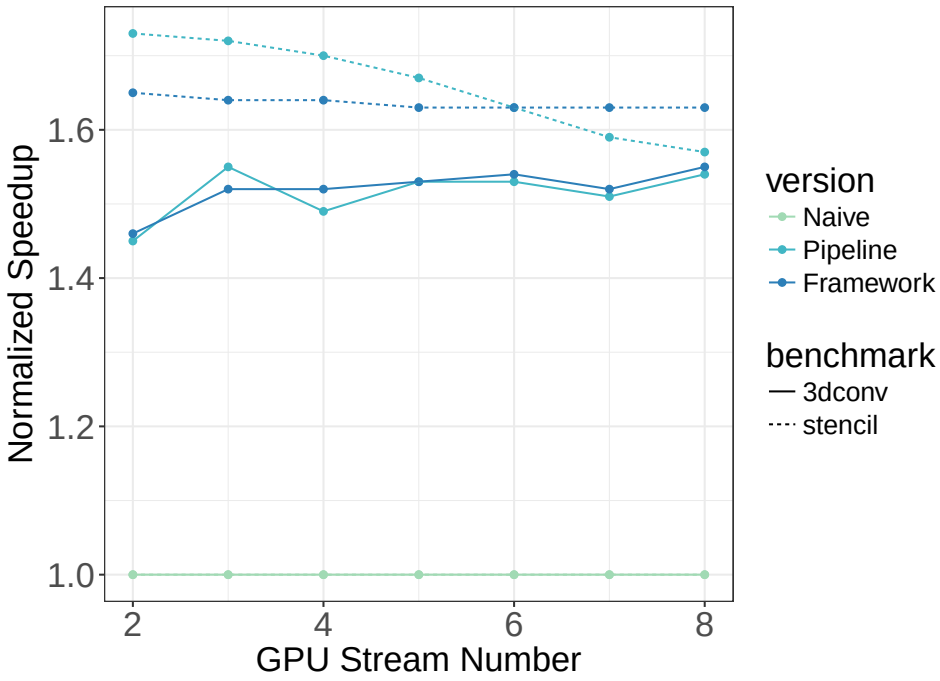

Figure 3.7: Execution Time Varying GPU Stream Count on Nvidia K40m

To ameliorate this issue, we vary the chunk size and number of streams. Our conclusions include that even if more chunks imply more API call overhead, it can be ignored on Nvidia GPUs. However, that overhead is more significant with the AMD GPU. The AMD APP Profiler results indicate that the performance degradation arises because:

- We split the task by the outer loop into small chunks, which means the chunk size is 1 and number of chunks is the problem size in that dimension, which requires many API calls and high scheduling overhead;
- Splitting the tasks into small chunks decreases the array size of each transfer, possibly to below the data transfer unit size for the AMD GPU, thus limiting bandwidth.

To test our theory, we modify our code to decrease the number of chunks. We then evaluate the performance of the Pipeline version versus the Naive version as we vary the number of chunks.

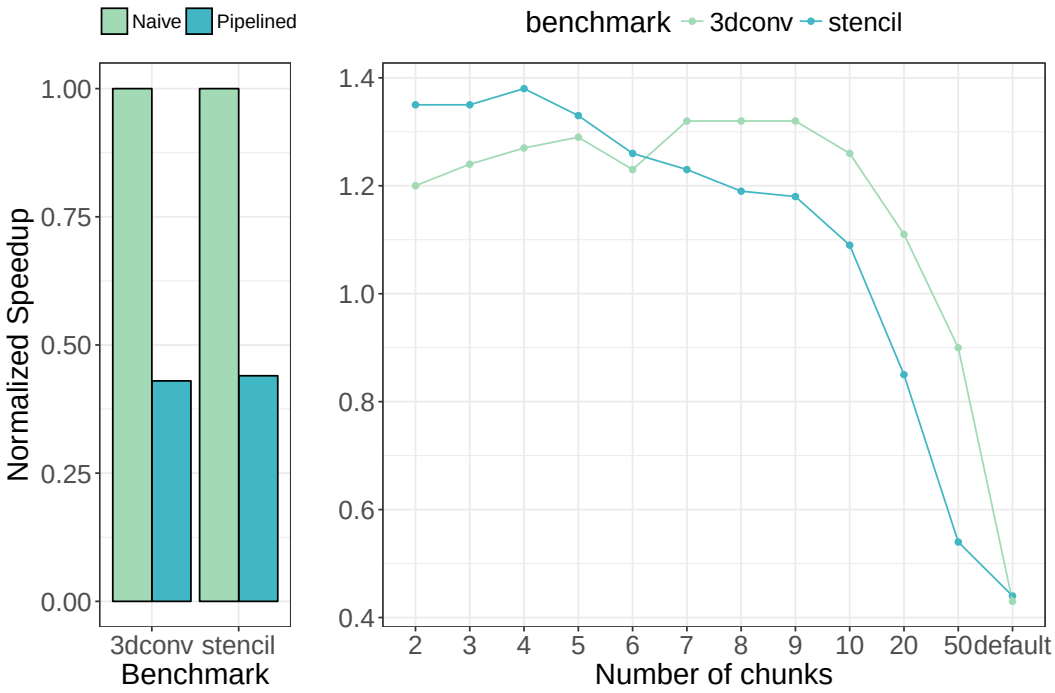

Figure 3.8: 3D Convolution and Stencil Performance Degradation (Left) and Normalized Speedup , Varying Number of Chunks (Right) on AMD HD 7970

Figure 3.8 (right) shows that if we split the problem into only two chunks that we achieve 1.2× speedup over the Naive version. Performance improves as we increase the number of chunks until we use nine chunks, after which it degrades sharply. When we use between 20 to 50 chunks, performance is worse than the Naive version, it continues to decline to the default chunk count.

Stencil

The Parboil Stencil benchmark represents an iterative Jacobi solver of the heat equation on a 3-D structured grid, which can also be used as a building block for more advanced multi-grid PDE solvers. We implement a prototype of the stencil benchmark using our approach.

Figure 3.5 shows the performance evaluation for the stencil benchmark on the K40m GPU.

The Pipeline version, which uses native OpenACC pragmas to pipeline the kernel computation and data transfer, achieves 1.57× speedup over the Naive version. Our Framework version is even faster than the Pipeline version, even including the time to handle array indexing and function calls. Our analysis finds that we only use two streams to implement the Framework version. However, we assign one stream to handle each subtask with the OpenACC `async()` clause, which indicates that it uses the maximum number of available GPU streams by default. Although more GPU streams could potentially hide more "bubbles" in the pipeline, they require more scheduling and API calls and can create contention overhead. Overall, these effects have more overhead than the benefit from overlapping data transfer and kernel computation. Since these parameters are building blocks of our schedules, we evaluate their impact, as Figure 3.7 shows.

We observe that the Pipeline version uses eight streams by default, which explains its execution time of 6.48 seconds in Figure 3.5. Further, as we increase the number of streams, the execution time of the Pipeline version increases dramatically while our Framework version remains stable. If we limit the number of streams to two instead of using the default eight streams, the Pipeline version performs best. However, as the stream count increases, the performance crosses over: with over six streams, the Framework version is faster. Either pipeline version provides at least 1.5× speedup over the Naive version.

Figure 3.6 shows the memory usage of our prototype for the Stencil benchmark. Our Framework version reduces memory consumption nearly 50% compared with the Pipeline version. Further, the GPU runtime and scheduler, rather than the data set, consume a large portion of the memory for this small test case.

The stream count can significantly affect memory use. The Pipeline version requires more memory to schedule the streams and to maintain the corresponding information. Our Framework version also requires a larger buffer allocation. Also, memory use increases slightly as

we increase the stream count. Our approach always reduces memory consumption nearly 50% for the Stencil benchmark.

Figure 3.8 (left) shows the poor performance of Parboil Stencil on the AMD HD 7970 with the default number of chunks. For the Stencil benchmark, the Naive version is 56% faster than the Pipeline version. We again verify that reduced effective transfer bandwidth leads to the performance loss. Figure 3.8 (right) shows that with two chunks, the Pipeline version achieves 1.35× speedup over the Naive version. As we increase the number of chunks to four, performance improves slightly. With more chunks, performance degrades until it is the same as the Naive version between 10 and 20 chunks, after which it becomes worse. The results with the 3D convolution and Stencil benchmarks demonstrate that data transfer bandwidth and API overhead limit the benefit of pipelining on the AMD GPU. More chunks require more API calls and scheduling overhead and reduce the chunk size below that required to maximize data transfer bandwidth.

Overall for the Stencil benchmark, our approach significantly reduces memory use while performing competitively with a hand-coded OpenACC solution. Further, our approach automates index translation and scheduling, which improves programmability, thus increasing the key motivation to use directive-based extensions. We find that stream count can impede the OpenACC solution. Using too many GPU streams reduces performance of the Pipeline version, while our prototype is not sensitive to stream count.

Lattice QCD

Quantum Chromodynamics (QCD) is the component of the standard model of elementary particle physics that governs the strong interactions. Our Lattice QCD benchmark is a larger application from the SciDAC Lattice Group. The main computational subroutine has several

parallel regions, which operate on a high-dimensional lattice. These features complicate a hand-coded implementation, which indicates that programmability is particularly important. The problem size can be formalized by $O(Cn^4)$ where C is a relatively large constant. We evaluate our prototype with three data sets where n is equal to: 12 (small), 24 (medium), and 36 (large).

Figure 3.5 shows the performance of the Lattice QCD code. In the large test case, our prototype delivers 1.54× speedup over the Naive version. The huge indexing operation to map the high-dimensional space to the pre-allocated buffer probably leads to the performance difference. We pass the offset variables into the OpenACC kernel region to point to the corresponding location inside the pre-allocated buffer. Since the kernel is much larger and contains many more array element accesses, the index calculations, additional operations inside the kernel, reduce performance compared to the hand-coded version. Nonetheless, the Framework version significantly outperforms the Naive one.

Figure 3.6 shows that compared with the Pipeline version, our Framework version significantly reduces memory use. As we increase the problem size, the memory savings also increase. For the largest test case, our approach reduces GPU memory use up to 79% and achieves competitive performance.

Matrix Multiplication benchmark

Matrix multiplication is a fundamental building block for many scientific computing applications. Moreover, the algorithmic patterns of matrix multiplication are representative. In our previous benchmarks, all data transfers are contiguous. In this section, we use the Matrix-Multiplication benchmark from the Polybenchmark suite as a case study to investigate the performance of our approach with non-contiguous data transfers.

We use a naive OpenACC matrix multiplication implementation from the Polybenchmark suite as our "baseline". With the matrix multiplication $A \times B = C$, this Naive implementation assigns one GPU thread to each element in matrix C. Each GPU thread gathers a line of A and a column of B and then calculates the corresponding element in C.

Many optimization methods have been developed to improve matrix multiplication performance; matrix blocking or tiling is an important one. By splitting the matrix into tiles, the size of each sub-matrix can be controlled to fit in shared memory. We assign one GPU thread block to each sub-matrix multiplication after loading the elements into shared memory. We accumulate these results into C. Ensuring shared memory use with OpenACC is difficult; we use the `private()` and `cache()` clauses. We denote this version as the "block-shared" version. Each task only needs data from a column of blocks in matrix A and a row of blocks in matrix B. We then apply our previous approach to this benchmark, partitioning the inputs and tasks into chunks by columns in A and rows in B. We assign one GPU stream to each task and copy the necessary data to a pre-allocated buffer. Mapping columns of blocks in Matrix A requires non-contiguous data transfers. After that we launch the computation kernel, and finally pipeline these GPU streams. This version is our "pipeline-buffer" version.

Figure 3.9 shows matrix multiplication performance across versions on an Nvidia K40 GPU. We observe that the block-shared version, which uses block partition and shared memory, can achieve up to 3× speed up over the baseline: Using shared memory significantly reduces global memory access. We also observe that our pipeline-buffer version achieves almost the same performance as the block-shared version. We then use Nvidia Visual Profiler to profile these two versions. We find that since the matrix multiplication is compute bound, the data transfer takes little time compared to kernel computation. Although non-contiguous data transfers take more time, it can be completely overlapped with the kernel computation. Thus, the two versions achieve nearly the same performance.

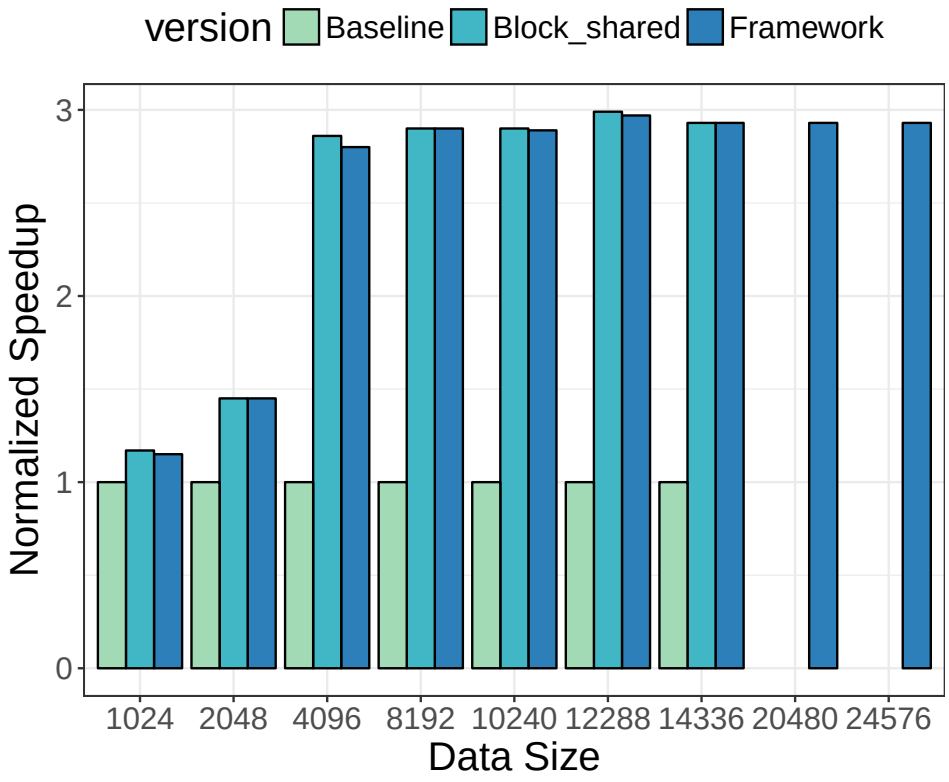

Figure 3.9: Matrix Multiplication Performance on Nvidia K40m

Figure 3.10 shows the memory usage of our Matrix Multiplication versions. Pipelined-buffer significantly reduces memory consumption. As we increase the problem size, the memory savings also increase. Since we only split the Matrix A and B, if the data size is large enough, it reduces memory use nearly 66% while delivering competitive performance. This savings allow the Pipelined-buffer to compute, with no performance loss, problem sizes that exceed GPU memory for the other two versions, as shown by the two rightmost problem sizes in Figures 3.9 and 3.10.

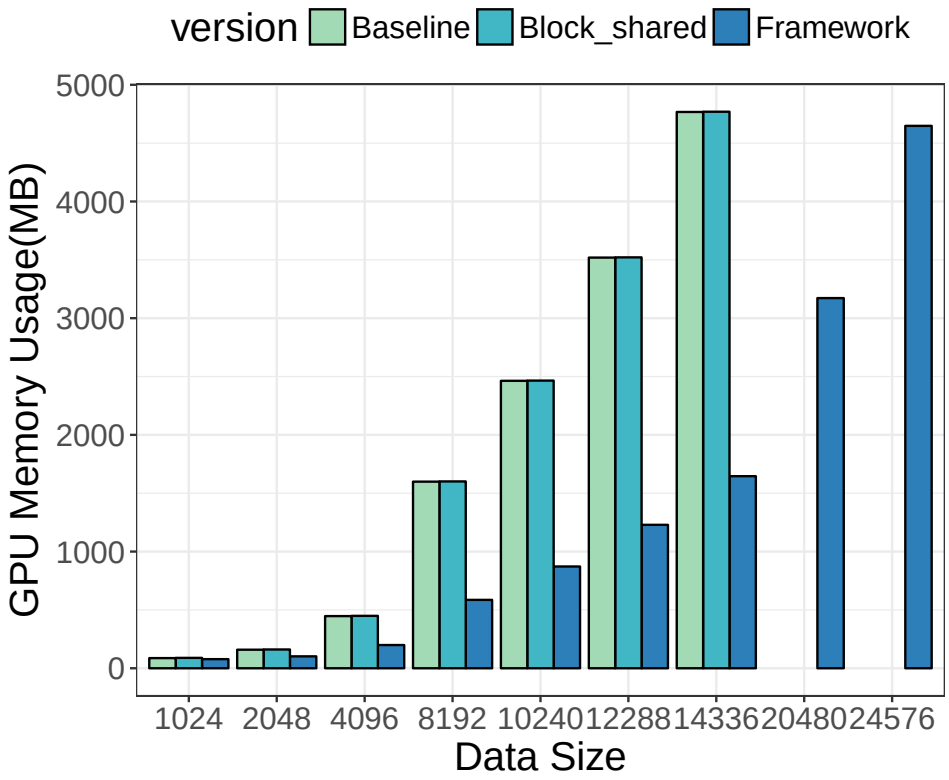

Figure 3.10: Matrix Multiplication Memory Consumption on Nvidia K40m

3.4.3 Comparison to the Unified Memory Technique

Unified Memory is a feature of the CUDA programming model that simplifies porting applications to GPUs by providing a single, unified, virtual memory space. It was first introduced in CUDA 6 on the Kepler GPU family. However, that version was still in early stages and the performance was relatively poor [49]. Nvidia's Pascal GPUs incorporate significant improvements to simplify programming and sharing of memory between CPUs and GPUs.

First, it extends the GPU addressing capabilities to enable 49-bit virtual addressing. Therefore, the P100 Unified Memory allows programs to access the full address spaces of all CPUs and GPUs in the system as a single virtual address space, unlimited by the physical memory size of any one processor.

Second, Unified Memory (UM) for Pascal GPUs supports device memory page faults. Combined with the system-wide virtual address space, page faulting provides several benefits. One of the most important benefits that is related to our directive-based extension is that Pascal GPUs support GPU memory oversubscription. We designed experiments to compare our proposed extension prototype to UM on the P100.

Based on our proposed extension, we implement a version of our runtime to add data transfer hints to unified memory to exploit the data used for the distributed memory case. We also extend two benchmakrss to exploit it: the 3D Convolution benchmark, and the Matrix Multiplication benchmark. We apply the Unified Memory technique to these benchmarks. We also apply the newest optimization techniques provided by CUDA 9 for UM to provide better performance, such as "prefetching". There exists two types of prefetching: (1) device-to-host (DtoH) data copy prefetching and (2) host-to-device (HtoD) data copy prefetching. We denote these two kinds of prefetching as "CPU Prefetching" and "GPU Prefetching," respectively.

Another reason to use prefetching is to compare the performance of the "UM + Explicitly Memory Prefetching" with the traditional data copy launched by `CudaMemCpyAsync()`. We also link the UM memory prefetching to multiple different non-default GPU streams, like what we do with the `cudaMemcpyAsync()`, to overlap the kernel computation with the data movement for better pipeline performance.

We run our experiments on compute nodes equipped with Nvidia TESLA P100 GPUs. They have 3,584 Nvidia CUDA Cores and 16 GB HBM2 Stacked Memory with NVLink support.

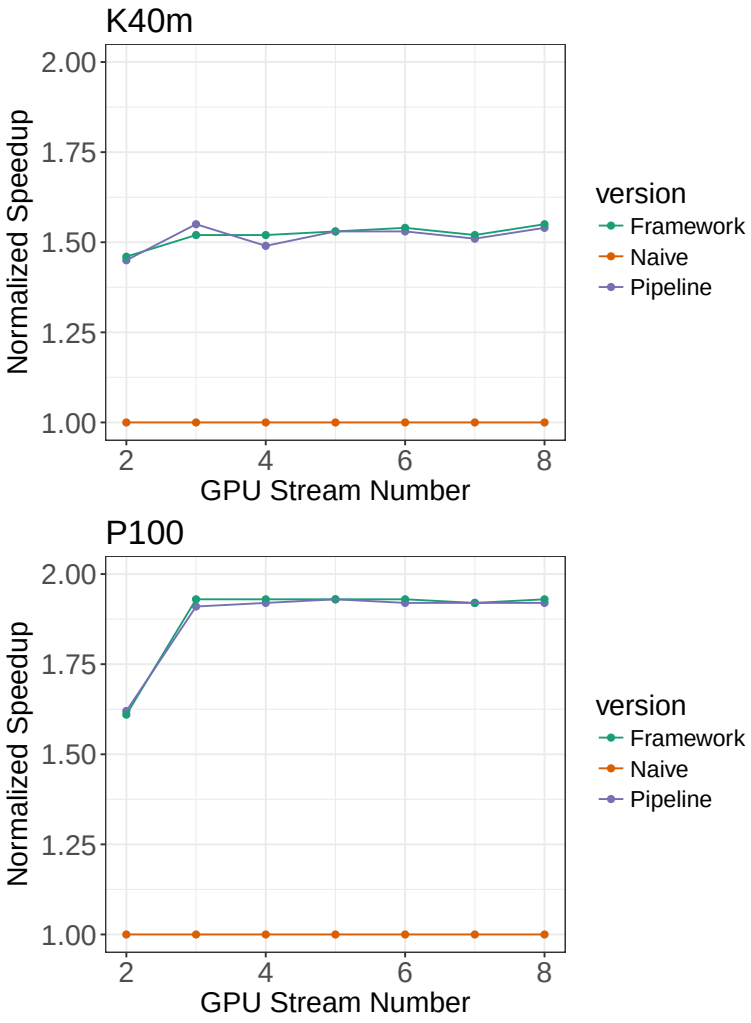

Figure 3.11: 3D Convolution on K40m vs. P100

Performance comparison K40m vs. P100

We compare the performance of our framework to Nvidia's Unified Memory. Specially we target Nvidia's P100 GPU on a Power-8 platform with NVLink and CUDA 9. We will first evaluate the performance difference between the K40m GPU and P100 GPU.

Figure 3.11 shows the performance of the 3D Convolution benchmark on these two devices. We can first observe that the average normalized performance speedup on the P100 is much higher than on the K40m, which is about 1.2×. We also vary the number of GPU streams

on both devices.

We can observe that the number of overlapping streams affects the performance of the Pipeline version on the K40m GPU. However, using two streams no longer delivers the best performance; we instead need up to eight streams to achieve the best performance.

On the P100 GPU, however, we first find a huge performance speedup after switching the number of GPU streams from 2 to 3, which is about 1.18× speedup. The performance stays stable if we continue to increase the GPU stream count, which is significantly different from that on K40m GPU.

Initial Study of Unified Memory

We first apply Unified Memory to the 3D Convolution benchmark from the Polybenchmark set [65] using the standard dataset. This default version basically does an iterative job on a 3D matrix. We denote this baseline as the "Naive" version. We first apply the pipelining technique to the baseline version in the "Pipeline" version. After that we use our proposed extension to implement a "Framework" version. At last, we apply the Unified Memory technique to it and denote this version as "UM" version. Figure 3.12 shows the execution time of different versions.

We observe that using pipelining to overlap the data transfer and kernel computation achieves a 1.91× speedup. Our framework provides exactly the same performance compared to the manually pipelined, "Pipeline," version. However, here we find that the "UM" version provides 2.97× speedup over the baseline. To determine the reason, we use the Nvidia Profiler to profile the program. We find that the data transfer size of the two different version is extremely different. Figure 3.13 shows the data transfer size of the "Pipeline" version and "UM" version for both device to host and host to device direction. We can see that the

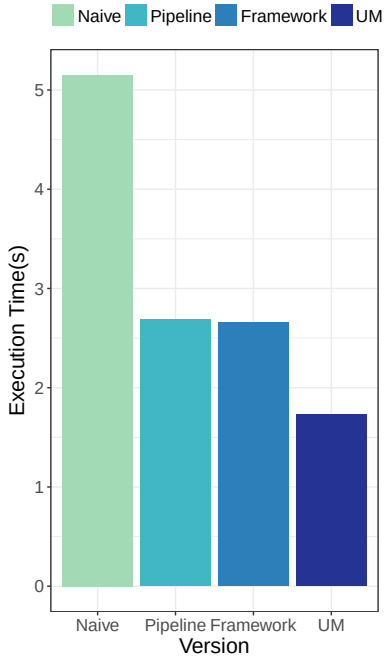

Figure 3.12: Performance of UM

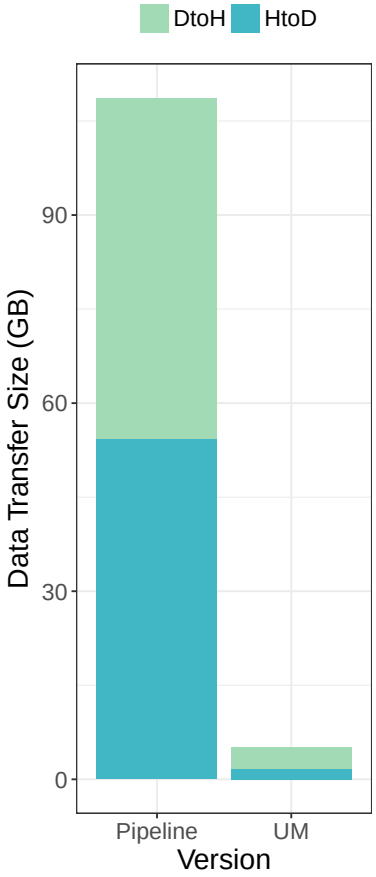

Figure 3.13: Data Transfer

data transfer size of "Pipeline" is 27.5× more than the "UM" version. We find that the UM runtime keeps the 3D matrix inside the GPU despite the array pointer exchange. However, in real world applications, data changes between iterations. For instance, if we use multiple nodes to compute, we need MPI exchange between the iterations. We cannot always keep the data inside GPU. Deep learning is another good example since we usually use a "small batch" from the entire super large dataset to train the neural networks. To simulate this kind of process, we add additional host data updates between each iteration to the data matrices on the GPU in the following evaluation studies.

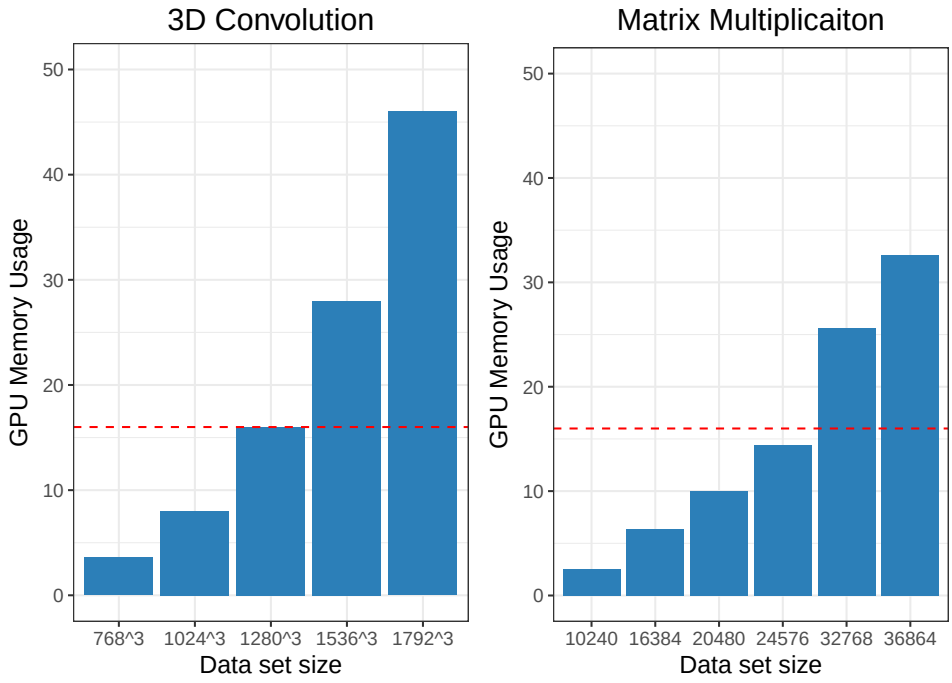

Figure 3.14: Data Set Size

3D Convolution Benchmark

We implemented six 3D Convolution versions. First, our baseline GPU version, which is denoted as "Naive", uses that model's naive offload method. Our "Pipeline" version uses OpenACC APIs to pipeline the data transfer and kernel computation by splitting the task into multiple chunks. The "Framework" version uses our prototype runtime framework. We also implement three versions that use UM; "UM" uses the managed memory for the baseline version. The other two use GPU prefetching and CPU prefetching.

Figure 3.14 shows the data set we use with the 3D Convolution benchmark. The red dash line shows that the P100 GPU has 16GB GPU memory. Our two small data sets fit into GPU memory. The third data set size is exactly 16 GB but does not fit into the P100 memory because of runtime memory overhead. The input data of our two large problems exceed the memory limit.

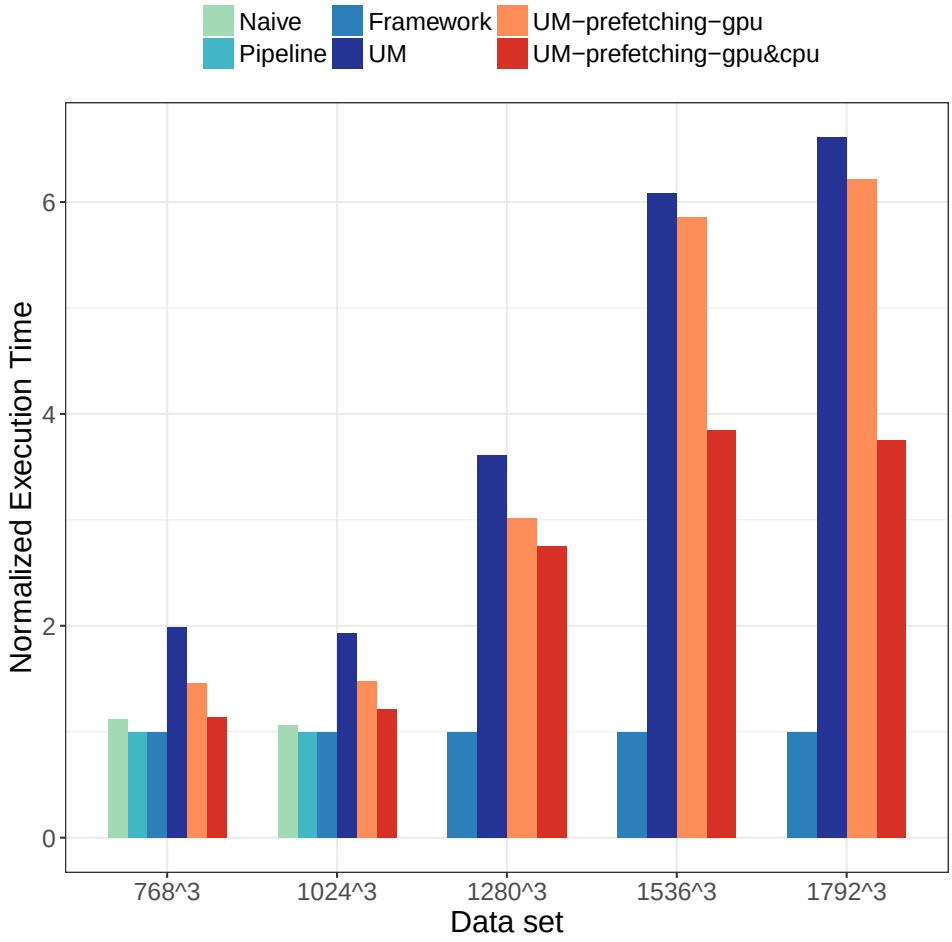

Figure 3.15: 3D Convolution Normalized Execution Time

Figure 3.15 shows the normalized execution time of 3D Convolution benchmark across different versions. We normalize all the execution time to the "Framework" version because it fit all data sets and its performance is stable. First, we can observe that pipelining the computational kernel and data transfer can provide 1.2× speedup over Naive, which is a relative high performance improvement. Also, the "Framework" version provides exactly the same performance compared to the "Pipeline" version when data size can fit into GPU memory. However, the "Naive" and "Pipeline" versions fail with larger data sets. After applying the Unified Memory technique, we find that the UM executes all test cases correctly but GPU and/or CPU prefetching improve its performance significantly. For small test

cases, the UM version is about 1.9× slower than the framework version. For large test case, where memory eviction is required, it is 3.6× to 6.5× slower than our framework version. Both CPU prefetching and GPU prefetching provide performance benefit here, because it is a iterative computation. If we apply both prefetching technique, we could get about 1.6× improvements.

Matrix Multiplication Benchmark

Figure 3.14 shows the data set we choose for Matrix Multiplication benchmark. We have two small and two medium data sets that fit into the GPU memory, one large data set that slightly exceeds the GPU memory and one super large data set that significantly exceeds the GPU memory. We note that the Matrix Multiplication Benchmark here is not a iterative computation, which means we have only one iteration here.

Figure 3.16 shows the normalized execution time of the Matrix Multiplication benchmark. First, we can observe that the block matrix multiplication algorithm can provide 1.6× to 2.2× speedup over the Naive matrix multiplication algorithm. We also find that prefetching is not effective as we observe little performance improvement after applying the prefetching technique, which is usually less than 2%.

If the data set is small enough (the first two datasets), the Unified Memory versions provide about 96% to 99% performance compared to the corresponding versions without using Unified Memory, which indicates that the compiler applies heuristic optimization that automatically alleviate the GPU page-fault overhead, making prefetching technique less effective at the same time. If we increase the data set size, where over 10 GB GPU memory is required, although it still fits into GPU memory, we observe 1.3× performance degradation for UM versions. So, the heuristic optimization appears to have data set size limitations. As we

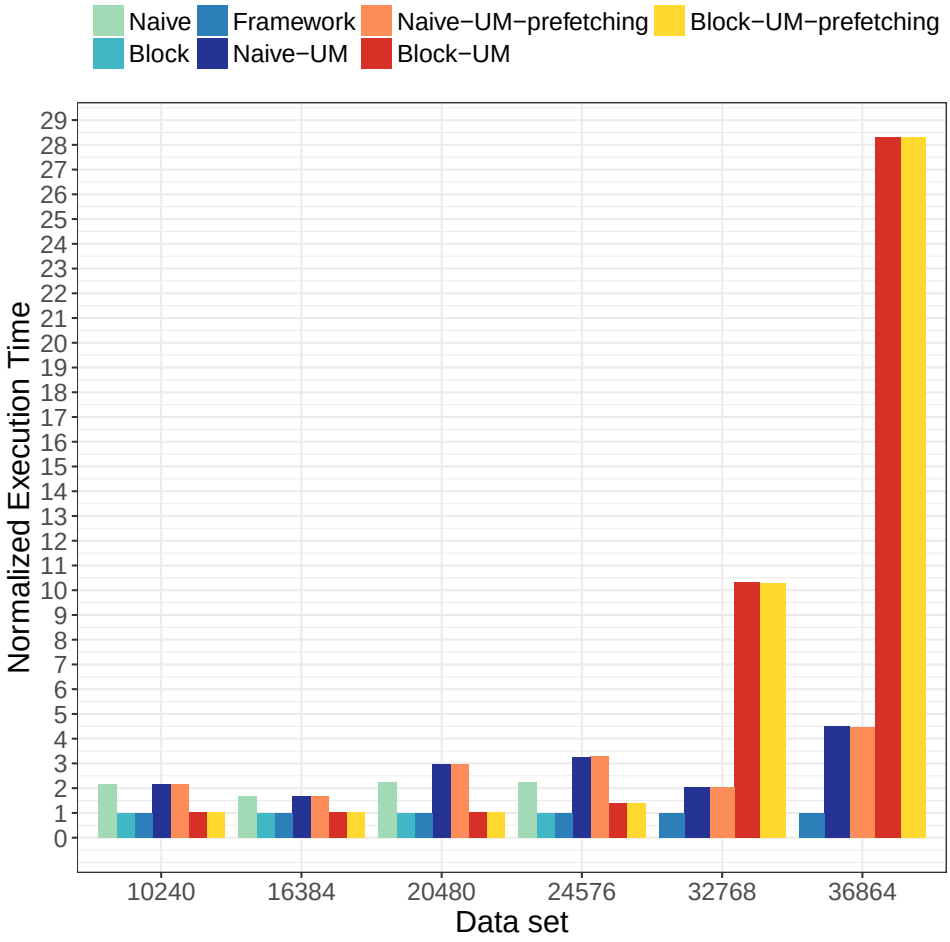

Figure 3.16: Matrix Multiplication Normalized Execution Time

continue to increase the data set size, making it much larger than the 16 GB GPU memory available, the last two data sets show us significantly different results. We observe that when data sets exceed the GPU memory limit, the UM version for naive matrix multiplication algorithm performs normally, maintaining the same performance difference as before. However, the Block UM version shows very huge (10.3× to 28.3×) performance degradation, which is significantly different from that of the Naive UM version (2× to 5× performance degradation). Based on the previous assumption, the non-contiguous data transfer may cause some extra data transfer due to the heuristic software level optimization by the compiler.

We also notice that the prefetching technique does not provide performance benefit for both Naive and Block versions in this case. The Matrix Multiplication is just a one iteration computation, we also evaluate another version which has multiple iterations with host updates between iterations. The results still show that the UM versions provide almost the same performance compared to the versions without UM, which is the same as our results discussed above. Also, prefetching still does not provide significant improvements.

We have to note that these experiments use the PGI 17.9 compiler with CUDA 9.0.176 for the UM versions, because previous PGI compilers only used one GPU stream. In those versions, PGI disabled asynchronous support for P100 and older GPUs when managed memory is used to avoid segment faults that can arise if the host and device access the same address. We may lose the benefit from prefetching technique for UM, because we are not able to overlap the prefetching with the kernel computation if we have only one GPU stream.

To sum up, we show that the UM memory engine has compiler-level optimizations for small data set movement between host and device that are effective for small test cases. However, for large datasets especially datasets that exceed the GPU memory with non-contiguous memory access patterns, it may cause performance degradation.

3.4.4 Summary and Discussion of Experiments

We implement a prototype framework for our proposed extension and apply it to four benchmarks. We show the performance on Nvidia K40m GPU, P100 GPU and AMD Radeon HD 7970 GPU. Our approach significantly reduces GPU memory use while delivering competitive performance to a hand-written pipeline version. Also we find that the K40m GPU is sensitive to GPU stream count, but the P100 GPU is not. Using three GPU streams is much better than using two GPU streams for P100 GPU. We observe that pipeline performance

on the AMD device is sensitive to the number of chunks due to API calls launch overhead and data transfer bandwidth limitations.

We then compare our framework to Nvidia's newest Unified Memory technique provided by CUDA 9 on Pascal P100 GPU. The experimental results shows that our extension outperforms Nvidia's Unified Memory by 25% to 40% for data sets that fit into GPU memory and 260% to 2830% for those that do not. We also explain the potential reason is that the Nvidia's compiler has optimizations for non-contiguous data movement, which results in extra data movement overhead.

Our evaluation results show that directive-based programming models for accelerators should include our partitioning and pipelining extension. This approach significantly improves performance and programmability. It also supports running applications with huge data sizes without complex coding changes.

3.5 Conclusion

We propose a directive-based pipelining extension for offload models such as OpenMP and OpenACC. Our extension allows GPU programmers to pipeline data transfers without major refactoring, thus automating overlap of computation and communication. Further, mapping subsections of the host array to a device buffer can reduce memory requirements and increase portability. To show the benefits of our design, we choose four benchmarks: the Stencil benchmark from the Parboil suite, the 3D Convolution and Matrix Multiplication benchmarks from the Polybenchmark set, and a SciDAC Lattice QCD application. We extend them with our prototype runtime and present a detailed evaluation that compares the programmability, performance and GPU memory consumption. We also compare it with the state-of-the-art Unified Memory technique released by Nvidia on PASCAL GPUs. Our

results show that our implementations hav better overall performance, particularly while dealing with non-contiguous data movement.

Chapter 4

Block-Level Data Pipelining for Graphic Processing Units

4.1 Introduction

GPU-accelerated systems are prominent on the Top500 list [25]. While many new and proposed programming models support them, scientists often prefer to keep their existing verified C, C++, or Fortran code rather than grapple with the unfamiliar. Since 2013, OpenMP has provided a straightforward way to adapt existing programs to accelerated systems [57].

The directive-based offload mode includes mapping annotations that ensure the accelerator can access the data and that the results are available on the host after the computation. If the accelerator cannot directly access host memory or if using accelerator memory can improve performance, the data is copied to device memory. The naive offload model copies data synchronously, which can adversely impact performance.

OpenMP [59] currently enables asynchronous data copy and computation pipelines. However, users must *manually* split the task into chunks and that they must launch with different GPU streams. This manual approach requires error-prone code refactoring and incurs extra function-call overhead. The hyper-parameters, e.g., stream (i.e., #streams) and chunk (i.e.,

#chunks) counts, must be carefully tuned to achieve optimal performance; poor choices may significantly reduce it.

As system heterogeneity grows more complex, the burden increases on the CPU to manage data movement among discrete memory spaces. Bypassing the CPU can efficiently offload tasks to accelerators [43, 81]. However, traditional kernel-level data pipelining still involves CPU data movement, sub-kernel launch, and task dependency control, which still consume significant CPU resources. The GPU-bound processes may adversely impact the CPU processes and vice versa [69]. Further, the data may not fit in device memory because scientific applications frequently use huge arrays or matrices. The user must then manually split the data and the associated computation, which can involve significant code changes since the user must partition large data arrays on the host and separately pass each array pointer.

We propose a block-level data pipelining (BLP) mechanism to resolve these issues. It performs data communication and computation inside one GPU kernel and can completely bypass CPU control after a single kernel launch. Persistent thread blocks occupy the GPU streaming multiprocessors (SMs). We partition these thread blocks into groups to handle various sub-tasks, including data communication. Our novel GPU flag-array mechanism monitors sub-tasks for completion. BLP can map data into a small buffer to reduce memory usage and to support GPU memory oversubscription.

Recent studies use GPU persistent threads to reduce kernel launch overhead and to share a GPU between multiple programs [16]. However, these approaches suffer from limited GPU occupancy and scheduling uncertainty, especially for programs that use locks. Our approach leverages cooperative thread groups [19] to implement efficient synchronization across persistent thread blocks, resulting in three major advantages. First, bypassing the CPU for data movement and sub-task management releases many CPU resources, which significantly benefits CPU-bound processes. Second, we enforce dependencies between sub-

tasks with our GPU flag arrays and execute the sub-tasks in the program's topological order, which eliminates development effort to serialize tasks. Third, our OpenMP pipeline syntax improves BLP programmability.

We find that a few reserved thread blocks for memory copying can provide equal or better bandwidth compared to host-initiated memory copies. Our results for multiple benchmarks on Nvidia V100 GPUs show that BLP without any parameter-tuning delivers stable and competitive performance compared to traditional kernel-level stream-based pipelining.

This chapter makes the following contributions:

- An examination of the challenges and shortcomings of traditional kernel-level pipelining with GPU streams;
- A comparison of the memory bandwidth achieved by direct access and that achieved by traditional runtime copies on Nvidia V100 GPUs;
- Our BLP mechanism to overlap communication and computation within a single kernel launch;
- A demonstration that with appropriate index mapping, BLP can support GPU memory oversubscription while providing competitive performance; and
- A detailed BLP performance analysis of several application benchmarks.

4.2 Design and Implementation

4.2.1 Proposed OpenMP Syntax

Fig. 4.1 shows our extended OpenMP syntax to support BLP that avoids the pitfalls of kernel-level pipelining. The `pipeline` map-type modifier extends the `map` clause, which

makes data available at the beginning and/or at the end of the region. We use the OpenMP 5.0 iterator concept to parameterize task splitting and the range and stride of the associated data movement. Thus, we add an `iterator` clause that we use in parameters of the `pipeline` modifier. Those parameters are `dirc`, which specifies the memory copy direction (the existing OpenMP `to`, `from` and `tofrom`) and `num_block`, which specifies the number of thread blocks to use to move data. Other thread blocks handle computation tasks.

```
#pragma omp target teams distribute
  map(pipeline(iterator[,dirc[,num_block]]):items…)
  iterator(ident = begin:end[:step])
```

pipeline inputs	
dirc	*copy direction*
num_block	number of streaming multiprocessors to copy in/out data
iterator	iterator, defined on same directive, representing range and stride
map inputs	
items	array declaration *arr*[*iterator_expr[:len]* | *begin*:*len*]
iterator inputs	
ident	identifier for the iterator
begin/end/step	bounds and step of the iteration space

Figure 4.1: Proposed OpenMP Extension for Block-Level Data Pipelining

We add an option to map array sections that uses an iterator expression as their *begin* parameter. The format is `<var>[(iterator_expr|begin):len]` where `<var>` is the variable or base pointer of an array and `[iterator_expr:len]` identifies the dimension to split and the data size (e.g., `len`) accessed by a sub-task.

Fig. 4.2 shows a two-level nested, stencil computation loop. The `iterator(i = 1:_PB_NI-1)` splits the i loop with range and step 1. The `pipeline(i,to:4)` inside the `map` clause splits the data by the dimension that contains a function of i and that four thread blocks handle the host-to-device copies of the two-dimensional input array `A`. A function of i and `<size>` indicates the data chunks that we must copy before a compute thread block executes the ith

loop sub-task. For instance, the `A[i-1:3][0:_PB_NI-1]` indicates that we split this array by its most significant dimension and must copy the $i-1$, i and $i+1$ chunks in that dimension to the device before executing the ith computation sub-task. The `[0:_PB_NJ-1]` defines the other dimensions of array `A`. Similarly, the `from:4` and `B[i:1][0:_PB_NJ-1]` clause defines the output array `B`, its dependency for each `i` iteration and that four thread blocks handle the device-to-host copies. The `[i:1]` indicates that each sub-task `i` only stores its corresponding chunk i in the most significant dimension.

```
1
2  #pragma omp target teams distribute  \
3          map(pipeline(i,to,4):A[i-1:3][0:_PB_NI-1]) \
4          map(pipeline(i,from,4):B[i:1][0:_PB_NJ-1]) \
5          iterator(i = 1:_PB_NI-1)
6  for (i = 1; i < _PB_NI - 1; ++i){
7    #pragma omp parallel for
8    for (j = 1; j < _PB_NJ - 1; ++j)
9    {
10     B[i][j] =  0.2 * A[i-1][j-1] +  0.5 * A[i-1][j]
11             + -0.8 * A[i-1][j+1] + -0.3 * A[ i ][j-1]
12             +  0.6 * A[ i ][j]   + -0.9 * A[ i ][j+1]
13             +  0.4 * A[i+1][j-1] +  0.7 * A[i+1][j]
14             +  0.1 * A[i+1][j+1];
15   }
16 }
```

Figure 4.2: A Stencil Code with our Proposed Extension

A powerful code analysis engine capable of deep logic analysis of the code and dependencies [53] could significantly simplify our proposed extension and back-end implementation. Potentially, the compiler could determine the array definition and the data dependencies for each sub-loop to construct sub-tasks for each thread block. However, the assumption of these capabilities would limit the applicability of our extension to code that can be analyzed completely at compile time and complicate its adoption into the OpenMP specification. Thus, we allow developers to specify this information.

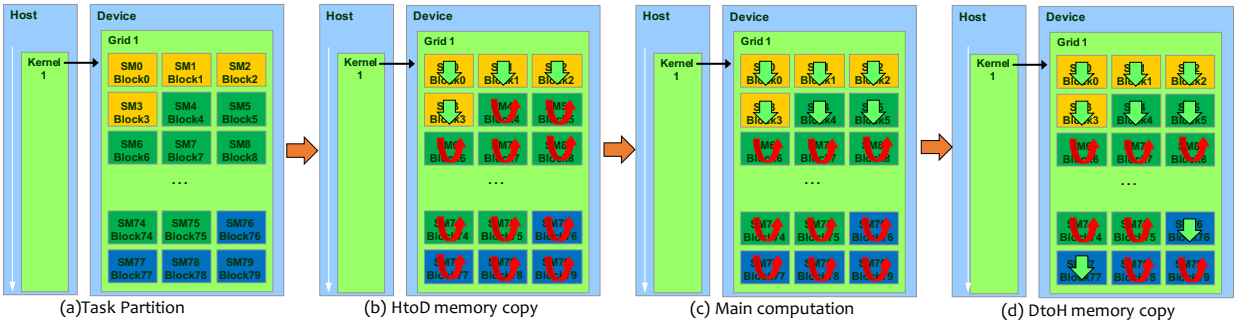

Figure 4.3: Block-level Pipelining Workflow

4.2.2 Block-Level Data Pipelining

Our BLP approach partitions GPU thread blocks into three groups to handle host-to-device memory copies, computation, and device-to-host memory copies. Developers may specify the partition sizes or allow our scheduler to choose them automatically. The best choice varies with several factors, including memory copy bandwidth as Section 4.3 shows.

We launch persistent thread blocks on the SMs for the entire program execution. Each thread of the data movement blocks reads data from pinned host memory and then writes it to GPU memory for the host-to-device (device-to-host) direction. We use the CUDA 9 cooperative group feature to address any issues with limited occupancy or thread block scheduling. It also guarantees that the thread blocks run concurrently, which avoids potential deadlocks.

The host buffer must be pinned so that the CUDA kernels can use zero-copy reads from host memory. This feature supports on-demand memory access for each operation. To hide latency and to utilize available bandwidth fully, our *direct access* mechanism launches multiple thread blocks that read host input data and store it in GPU global memory.

Our flag arrays in GPU global memory track when data movement and computation sub-tasks finish. The corresponding flags trigger dependent computation or data-copy thread blocks. Atomic operations or global memory operations read or write the flag arrays to

guarantee that modifications are visible across SMs. We call `__threadfence()` after storing the data but before the atomic flag update to guarantee that thread blocks that observe the update read correct data.

Fig. 4.3 shows a BLP example on an Nvidia V100 GPU. In subfigure (a), the CPU launches a kernel on 80 persistent thread blocks. Cooperative groups can support more thread blocks, in which case multiple thread blocks would run concurrently on the SMs. In our example, four thread blocks copy input data to the GPU (yellow), 72 thread blocks perform the computation (green) and four thread blocks copy output data to the CPU (blue). Subfigure (b) shows that the yellow thread blocks continually read input data from the CPU, which they store in GPU global memory. They update the flags after each round to indicate that the corresponding slice of the input data is ready. The green thread blocks spin-check the appropriate input flags before executing each task. After the flags are updated, those blocks execute the computation on that data, as subfigure (c) shows. Subfigure (d) shows that when a green thread block finishes its task, it updates the output flag to trigger the blue thread blocks to copy the results to the host.

If only one kernel is launched, the GPU handles all data movement, computation and the logic between each task. Thus, we do not use the CPU to enforce sub-task dependencies and do not need to tune the chunk size and number of GPU streams. Moreover, since the GPU handles all operations, call-back instructions before/after each sub-task (including computation and data movement) could easily be added without involving the CPU. The GPU thread block size is always the key performance parameter, which is important to both traditional stream-based pipelining and BLP. The work distribution of the thread blocks could be determined by a system bandwidth test. Section 4.3 shows that the number of copy threads limits BLP memory-copy bandwidth.

4.3 Experimental Evaluation

4.3.1 Experimental Hardware

We use two state-of-the-art GPU platforms for our experiments. The first has IBM Power 9 CPUs and Nvidia Volta V100 GPUs that are interconnected with Nvidia NVLink2. The second has Intel Xeon Gold 6136 CPUs and Nvidia V100 GPUs that are interconnected by PCI-E 3.0. The V100 GPU has 5120 stream cores and 16GB of HBM2 memory.

4.3.2 Benchmarks

Our evaluation uses multiple GPU benchmarks from PolyBench GPU V1.0 [64] written in CUDA.

2D Convolution and 3D Convolution

Convolutions on multi-dimensional periodic data are used to deconvolve blurred images, to process signals and images, to suppress noise, to extract features, and to model wave properties [28]. It is a step in many electromagnetic (EM) problems involving scattering and radiation. Also, convolution neural networks (CNNs) are important building blocks for deep learning (DL).

Generic Matrix-Matrix Multiplication (GEMM)

Matrix multiplication is a building block of many scientific computing and machine learning algorithms. We use the GEMM benchmark to investigate BLP performance with non-contiguous data transfers. Many optimization methods improve GEMM performance, in-

cluding matrix tiling. The block GEMM algorithm divides each matrix into multiple blocks, then separately multiplies the blocks, and finally accumulates the results. The block size is usually small enough to fit into the shared memory on GPUs to reduce global memory access latency.

2-D Finite-Difference Time-Domain Kernel

A two-dimensional finite-difference time-domain (2D-FDTD) method can accurately find the source location of an acoustic wave in an urban area [4]. Accelerators like GPUs excel at problems that involve many independent computations such as the FDTD method [26]. This particular 2D-FDTD benchmark actually consists of three separate compute kernels, each representing an inter-dependent step in the overall 2D-FDTD kernel computation. These kernels have separate input arrays and depend not only on their own input data but also on the output of the other two kernels. These complex dependencies make partitioning and pipelining difficult to map to an iterative procedure on sequential GPU streams. Our BLP approach only needs to set the task dependencies, which the GPU manages.

Matrix Transpose and Vector Multiplication

Matrix transpose and vector multiplication (ATAX) form the basis of fundamental algorithms such as linear solvers and eigenvalue solvers on symmetric matrices and are used in many scientific areas, such as quantum chemistry [75] and solid-state and nuclear physics. Optimization typically focuses on cache utilization on CPUs and shared memory on GPUs.

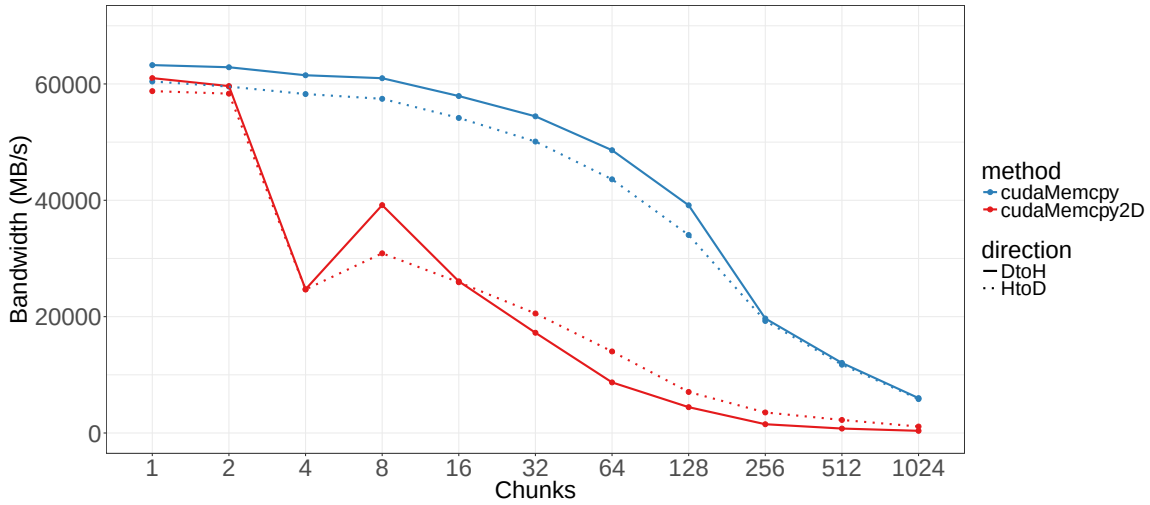

Figure 4.4: cudaMemcpy and cudaMemcpy2D Bandwidth with NVLink2

4.3.3 Comparison of Memory Copy Bandwidth

Fig. 4.4 shows `cudaMemcpy()` and `cudaMemcpy2d()` bandwidth on a Power9/V100 NVLink2 node as the number of chunks varies. The maximum bandwidth is around 60 GB/s with only one chunk. Splitting the array into chunks drops the bandwidth substantially – a 99% reduction with 1024 chunks. The 2D-copy decrease is more precipitous, with a 90% reduction with only 64 chunks. The blue line in Fig. 4.5 shows the direct access memory bandwidth as the number of copy threads varies. Direct access provides similar bandwidth compared to the maximum bandwidth provided by `cudaMemcpy()` when enough GPU threads are used.

Fig. 4.6 shows the direct access bandwidth with a non-contiguous stride. We use a 32×32 thread block and copy the array chunk by chunk column-wise, which is similar to 64 chunks with `cudaMempcy2d()` with the same data set. Direct access with four or more SMs achieves nearly the maximum bandwidth or about 4× the `cudaMempcy2d()` bandwidth with the same stride. We show later that this result is critical to GEMM performance.

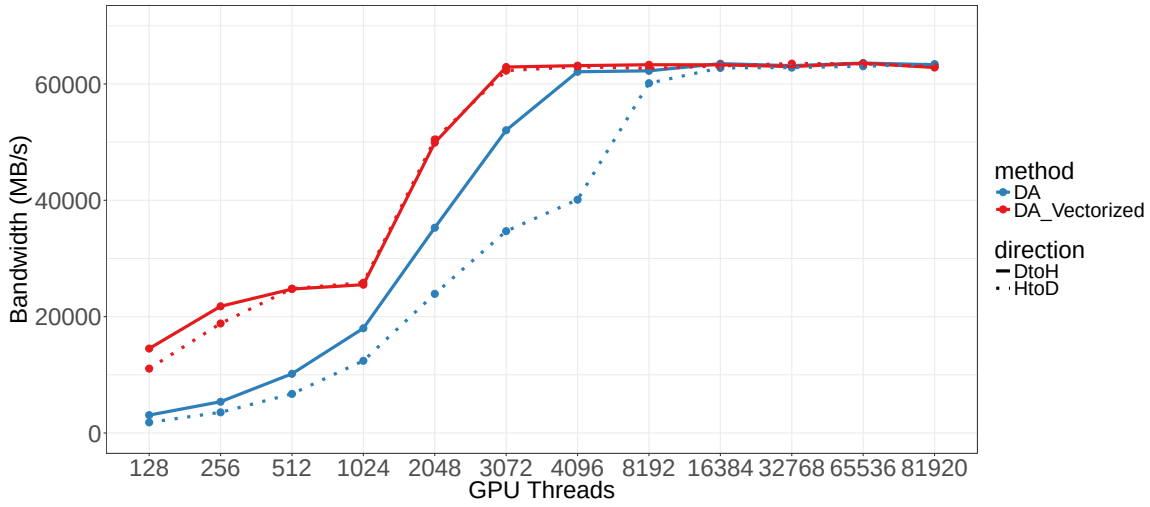

Figure 4.5: Direct Access Bandwidth with NVLink2

4.3.4 Block-Level Pipelining Optimizations

BLP uses persistent thread blocks to eliminate repeated kernel launch costs. Several persistent thread blocks run on the SMs. The number of persistent thread blocks depends on the number of SMs on the GPU, e.g., 80 persistent thread blocks on 80 SMs on a V100 GPU. We partition the thread blocks to handle different tasks (i.e., host-to-device data transfer, main computation, and device-to-host data transfer). However, to utilize the bandwidth and computation power fully, partition sizes should be evaluated for each system. To track the completion of each task, we maintain flag arrays for data movement or computation sub-tasks. The implementation of flag read/write operations can also significantly affect performance.

Memory Copy Optimization

Each thread in the memory copy thread block reads an element from the source memory space and stores it in the destination memory space. By default, a single instruction may load and store 32 bits from those addresses for data types such as `int` or `float`. We can use

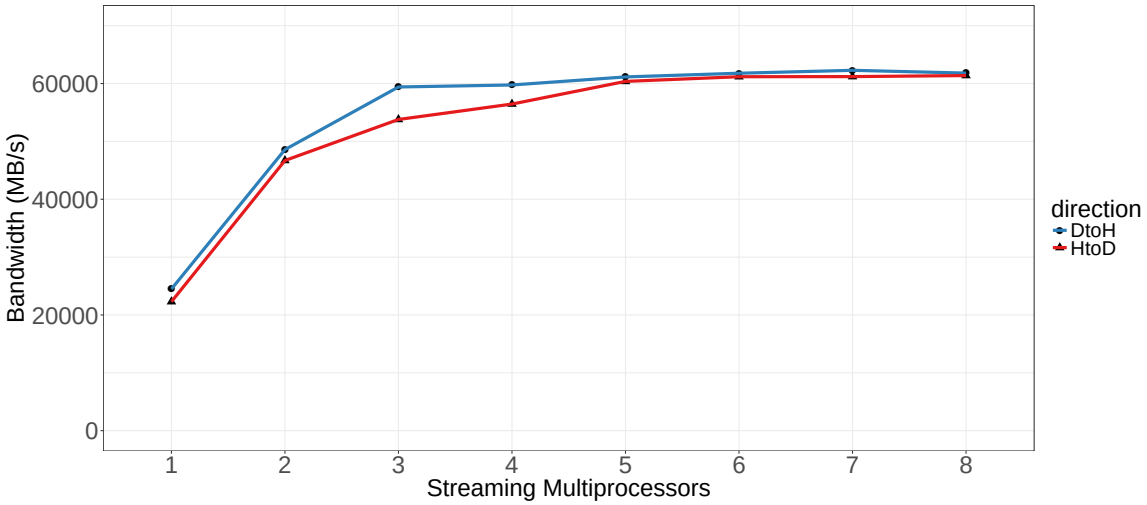

Figure 4.6: Direct Chunk Access Bandwidth with NVLink2

Table 4.1: Flag Read/Write Operations

Partition	Operation	Functions
HtoD	write	atomicAdd, atomicOr, atomicExch
Compute	spin read	atomicAnd, atomicCAS, global read
Compute	write	atomicAdd, amtoicInc
DtoH	spin read	atomicAnd, atomicCAS, global read

vectorized instructions to increase the bit count per instruction. For most benchmarks, we merge four 32-bit load/store operations into a 128-bit load/store operation.

Fig. 4.5 shows the direct access bandwidth on a Power9/V100 NVLink2 node. The red line shows that this optimization does not increase the maximum peak bandwidth. However, it significantly reduces the number of GPU threads required to achieve it. Only three or more thread blocks (each has 1024 threads) are required to achieve the maximum bandwidth. Fig. 4.7 shows the direct access bandwidth on the PCI-E-connected node. Since the PCI-E peak bandwidth is much lower, one thread block is sufficient.

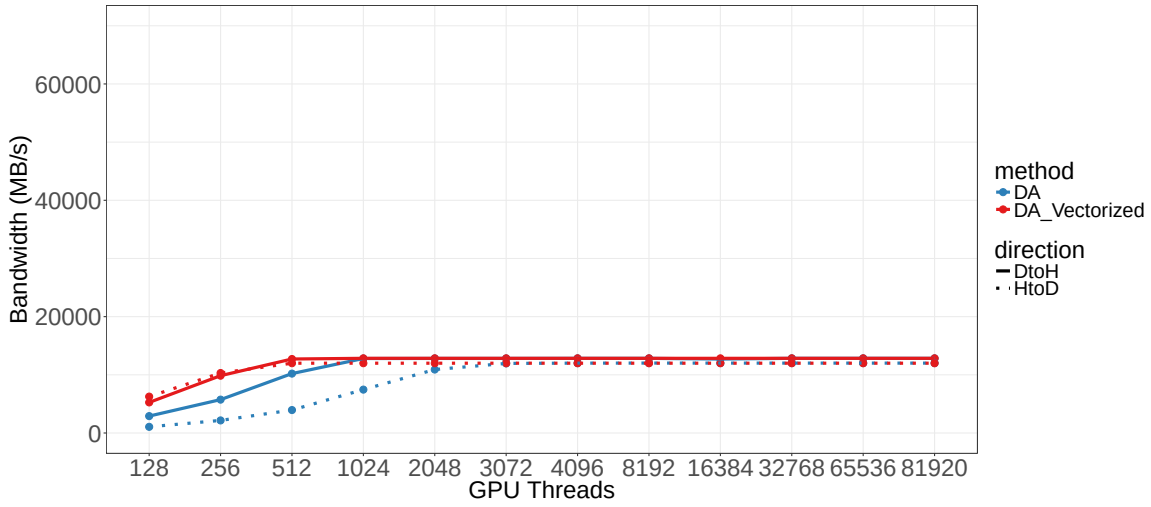

Figure 4.7: Direct Access Bandwidth with PCI-E

Flag Read/Write Optimization

Multiple implementations of the flag read/write operations are possible. We explore two approaches, atomic operations and global memory read/write with the `volatile` keyword, to guarantee that flag modification by one thread block is visible to other thread blocks. Table 4.1 shows the available instructions for each flag read/write. For most atomic operations, `atomicAdd`, `atomicAnd`, and `atomicOr` outperform other atomic operations in terms of throughput and latency. Due to concurrent flag write updates on the same data, we currently only use atomic operations for flag write operations.

Persistent Thread Discussion

CUDA 9 introduces cooperative groups, which extend CUDA to allow kernels to organize groups of threads dynamically. The cooperative groups programming model describes synchronization patterns, both within and across thread blocks. It provides device APIs that define, partition, and synchronize groups of threads. It also provides host APIs to launch thread grids that execute concurrently to enable synchronization across thread blocks. This

feature can significantly increase occupancy [29]. Even though some thread blocks are used for direct access memory copy, we can still launch more thread blocks than the number of SMs on a GPU to utilize its compute resources fully.

4.3.5 Application Performance

We now present our evaluation of BLP for the benchmarks described earlier. We compare performance with two other data pipelining implementations and two simpler choices based on Unified Memory (UM). In the following, we denote our BLP approach as *blp*. We also implement a traditional data pipelining approach by partitioning the tasks to multiple sub-kernels and mapping them to multiple GPU streams asynchronously, which we denote as *async*. Alternatively, we do not partition the tasks and instead execute activities synchronously, which we denote as *sync*.

We also compare to using UM to overlap data transfer and computation. The CUDA runtime can automatically handle data movement on demand, which we denote as *um*. CUDA also supports manual UM prefetching, which can guide the runtime to move data more efficiently. We denote this prefetching procedure, which is mapped to asynchronous GPU streams. as *um-prefetch*. We report all performance as the normalized speedup compared to the *sync* version.

Convolution Benchmarks

BLP achieves 1.67× speedup on 2D convolution on the NVLink2 node, which is 95% of the best hand-tuned *async* performance. Fig. 4.8 shows normalized *async* speedup on exhaustively enumerated hyper-parameter combinations (#streams, #chunks). While it achieves speedup as high as 1.77×, poorly chosen hyper-parameters can reduce performance by up to

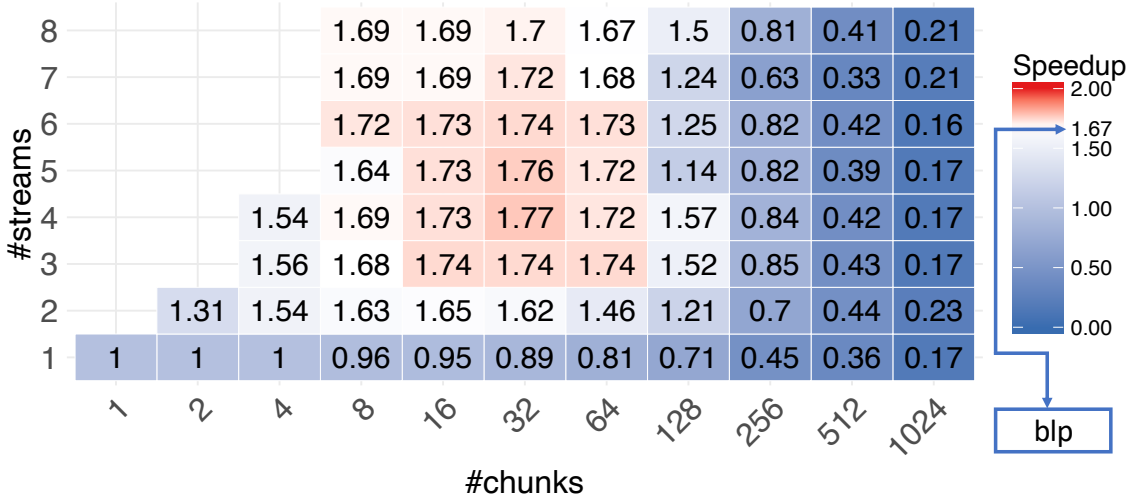

Figure 4.8: Normalized Speedup of Asynchronous 2D Convolution (NVLink2)

83%. Colors in the figure indicate relative *asynch* performance to *blp*. Recall that *sync* splits the persistent thread blocks into three partitions. On the NVLink2 node, we use 4 persistent thread blocks to handle the data transfers for each direction and the other 72 thread blocks handle the computation sub-tasks. In Fig. 4.8, the blue blocks indicate that *async* perform worse than *blp* while red ones indicate it is faster, which occurs rarely. Since Fig. 4.9 shows no red blocks, the speedup with *blp* (1.62) is higher than all *async* hyper-parameter settings on the PCI-E node.

Results for 3D convolution are similar. BLP achieves 1.95× speedup, which is 95% of the best tuned *async* version on the NVLink2 node and always outperforms it on the PCI-E node, as shown in Fig. 4.10 and Fig. 4.11. When we enumerate all hyper-parameter combinations (#streams, #chunks) to tune *async* performance, we achieve speed up as high as 2.05×. We observe that *blp* provides a speedup of 1.07 over the best hand-tuned *async* settings on the PCI-E node.

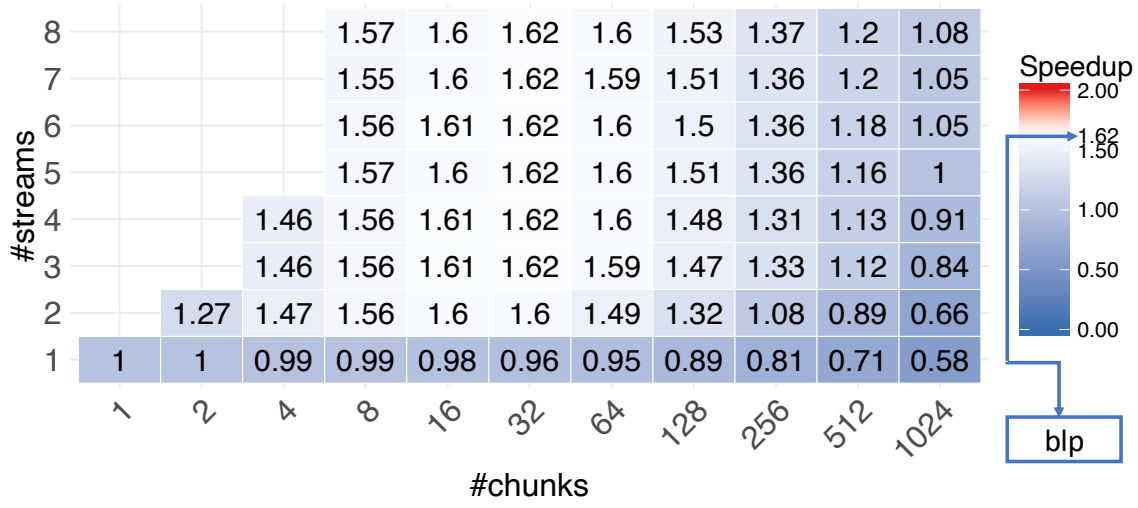

Figure 4.9: Normalized Speedup of Asynchronous 2D Convolution (PCI-E)

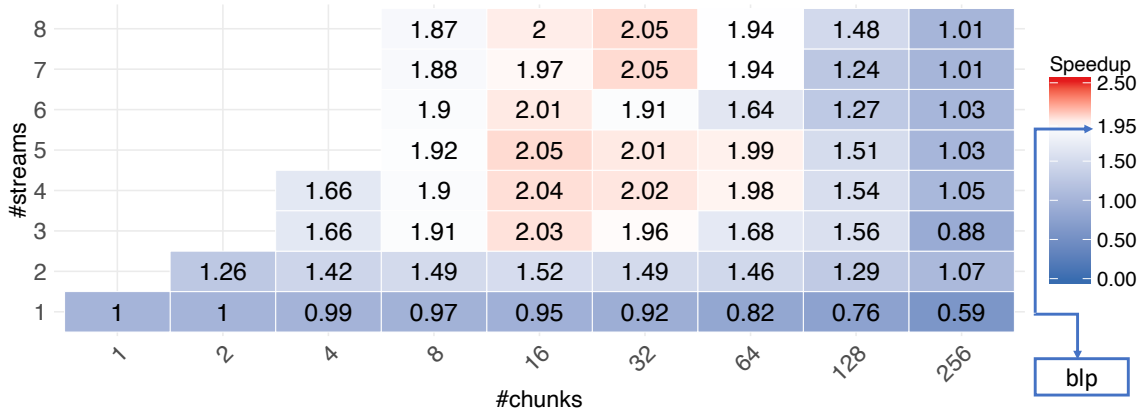

Figure 4.10: Normalized Speedup of Asynchronous 3D Convolution (NVLink2)

GEMM

Fig. 4.12 shows GEMM performance. We again compare *blp* and *async*. We leverage the block matrix multiplication algorithm such that shared memory improves performance, which requires non-contiguous memory copies to pipeline the tasks. We use a 32X32 block size so the default chunk size is 32 elements. So we can only vary one parameter, the #streams, to tune *async* performance and compare it to *blp*. The speedup of *async* is limited to under 2% on the NVLink2 node due to the huge bandwidth drop caused by using `cudaMemcpy2d()` for

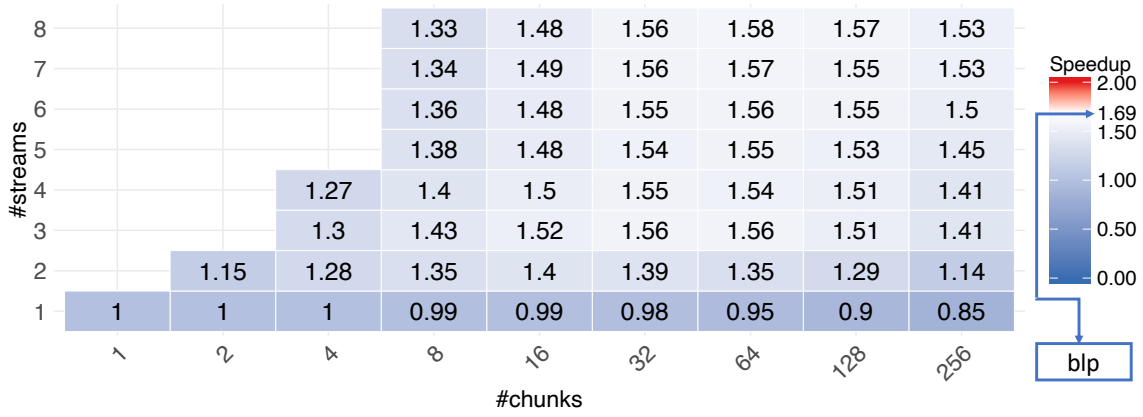

Figure 4.11: Normalized Speedup of Asynchronous 3D Convolution (PCI-E)

non-contiguous memory copy. Our prototype provides about 5% stable speedup over *sync*. On the PCI-E node, the speedup increases because the ratio of the data transfer and kernel computation is larger, which increases the benefit of pipelining. Moreover, the bandwidth drop caused by `cudaMempcy2d()` is also limited by PCI-E bandwidth. In this case, our prototype still outperforms the best hand-tuned *async* version by about 4%.

2-D Finite Different Time Domain Kernel

The structure of the 2D-FDTD task dependencies leads us to a separate bandwidth test that reads three input arrays and writes one output array slice by slice to guide data movement choices. We consider two implementations. One assigns a thread block to each array, which is our original memory copy method. The other, *Interleaved*, assigns each thread to handle one index of all arrays simultaneously.

Fig. 4.13 shows the *Interleaved* access bandwidth as we vary the number of GPU threads. For the NVLink2 node, *Interleaved* requires at least 5 thread blocks to achieve peak bandwidth. Fig. 4.5 shows that if we assign one thread block to handle each input array, three thread blocks already achieve the peak bandwidth. So, *blp* on the NVLink2 node uses one thread

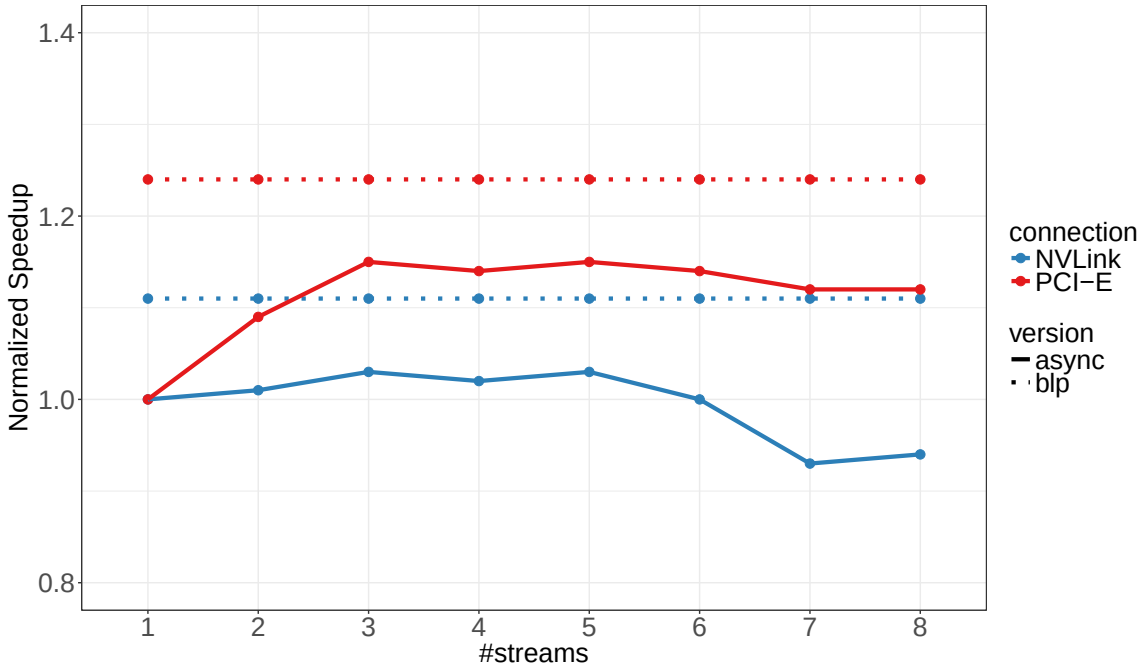

Figure 4.12: Normalized Speedup of GEMM

block to handle each input array. However, on the PCI-E node, *Interleaved* requires only two thread blocks to achieve peak bandwidth, so we use that approach for *blp* on that node.

Fig. 4.14 shows the 2-D FDTD *async* performance on the NVLink2 node. Exhaustive tuning can achieve as high as 1.26× speedup. With *blp*, we observe 1.34× speedup, which is a 6.3% improvement over the best hand-tuned *async* result. Fig. 4.15 shows *async* performance on the PCI-E node. The best tuned case yields 1.24× speedup while *blp* achieves 1.31× speedup, a 5.6% improvement.

Matrix Transpose and Vector Multiplication

Fig. 4.16 shows the matrix transpose and vector multiplication *async* performance on the NVLink2 node. Exhaustive tuning can achieve as high as 1.14× speedup while *blp* achieves 1.21× speedup, which is a 6.1% improvement over the best hand-tuned *async* result. Fig. 4.17

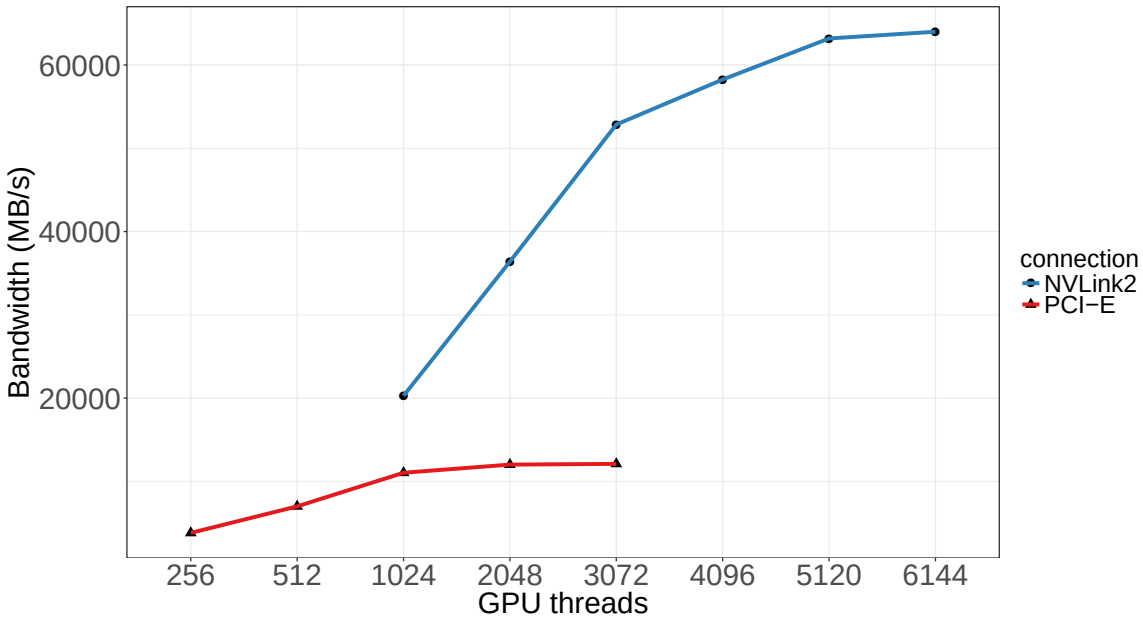

Figure 4.13: Memory Bandwidth of Interleaved Access

shows *async* performance on the PCI-E node, where *blp* achieves 1.13× speedup, an 8.6% improvement over the best tuned *async* performance.

4.3.6 Performance with Larger Datasets

Our previous results use the default Polybench datasets, as shown in Table 4.2, which shows the GPU memory usage as measured by *nvidia-smi*. We now increase the dataset sizes.

Fig. 4.18 and Fig. 4.19 show the speedups with varied dataset sizes. The results of the medium datasets are consistent with the default datasets. For medium datasets, BLP can provide 0.95× to 1.14× speedup over the best tuned *async* version on the NVLink2 node. We also see a speedup of 0.98× to 1.1× speedup on the PCI-E node.

Because BLP uses flag updates to trigger tasks, we can track the data blocks based on these flags. With additional index mapping, BLP can support GPU memory over-subscription by mapping large host arrays to smaller device memory buffers. To evaluate BLP in this mode,

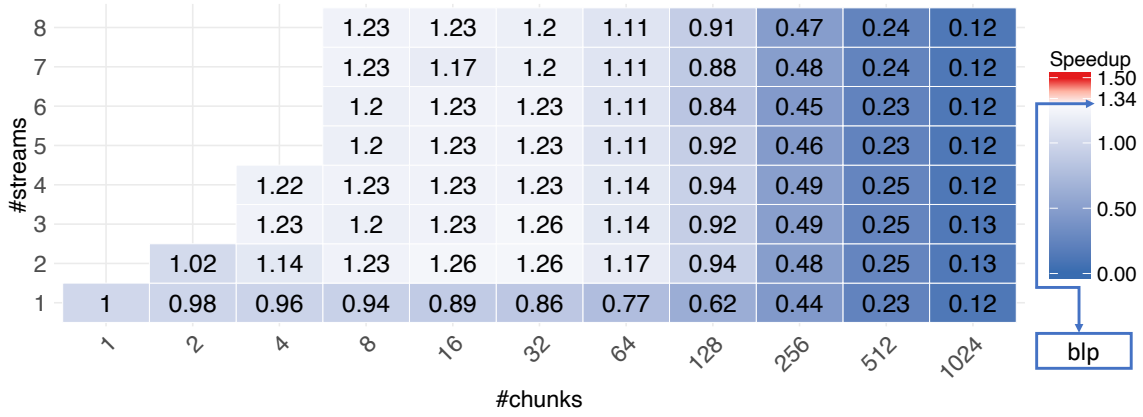

Figure 4.14: Normalized Speedup of Asynchronous 2D FDTD (NVLink2)

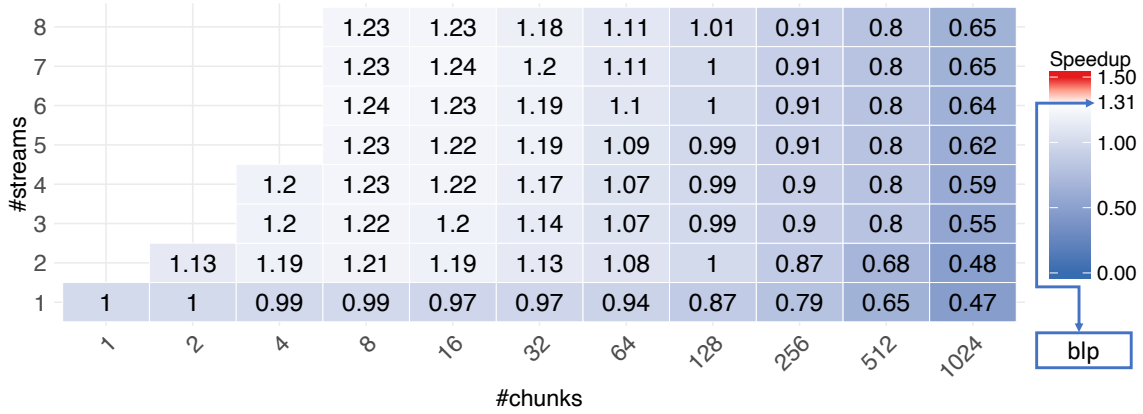

Figure 4.15: Normalized Speedup of Asynchronous 2D FDTD (PCI-E)

we use two benchmarks, 2D convolution and matrix transpose and vector multiplication, with datasets that exceed V100 GPU memory (16GB). We extend BLP with additional flag handling and buffer index mapping. Because the dataset exceeds the V100 GPU memory, we cannot use the *sync* and *async* versions. With BLP, the flags are allocated to label the data block use time. Based on the data dependency topology, before execution, we can calculate how many times the data are used. We initialize a flag with this number after the host-to-device memory copy and decrement the flag each time it is used with atomic operations. When the flag becomes zero, other data copy thread blocks can overwrite the data block to reuse the buffer space. We currently use round-robin order mapping, but other orders

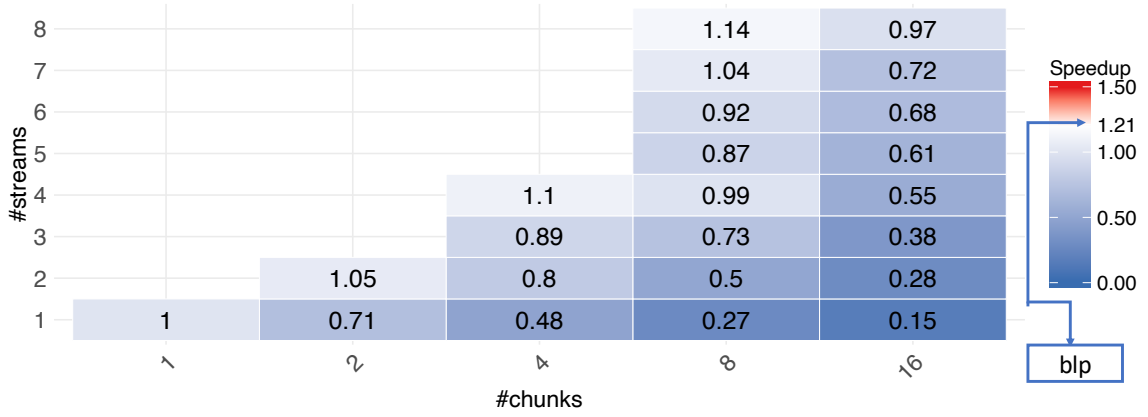

Figure 4.16: Normalized Speedup of Asynchronous ATAX (NVLink2)

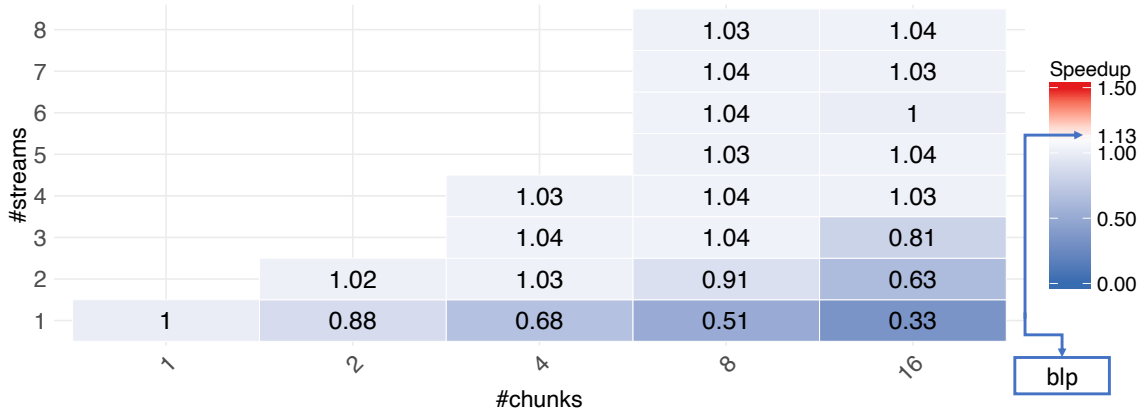

Figure 4.17: Normalized Speedup of Asynchronous ATAX (PCI-E)

can be supported. The buffer size could also be varied. We compare this approach to the latest update of UM on Volta GPUs, which supports GPU page faults and, thus, supports memory over-subscription. We implemented the BLP buffer version and compare it to UM with prefetching optimizations.

Table 4.3 shows the GPU execution time for 2D convolution and matrix transpose and vector multiplication the NVLink2 and PCI-E nodes. For 2D convolution, BLP achieves 1.89× and 1.63× speedup over *UM* with prefetching on the NVLink2 and PCI-E nodes. For matrix transpose and vector multiplication, BLP achieves 4.55× speedup on the PCI-E node. We observe a huge performance degradation for the *UM* version on the NVLink2 node, which

Table 4.2: Dataset GPU Memory Usage

Dataset	GPU Memory Usage(GB)
2DCONV_default	0.60
3DCONV_default	0.56
GEMM_default	0.30
FD2D_default	0.51
ATAX_default	1.30
2DCONV_medium	4.50
3DCONV_medium	4.40
FD2D_medium	4.42
ATAX_medium	4.50

Table 4.3: GPU Execution Time (sec) when Data Set Exceeds GPU Memory

	2dconv_nvlink	2dconv_pci-e	atax_nvlink	atax_pci-e
blp	0.22	0.90	1.60	1.78
UM	0.41	1.47	266.00	8.10

leads to BLP achieving nearly 166× speedup. Fig. 4.20 shows the GPU memory usage for these two versions. We can observe that using BLP with buffer mapping reduces GPU memory usage by 86% and 75% for the 2D convolution benchmark and matrix transpose and vector multiplication, respectively.

Since the 2D convolution performance is linear, we can predict the theoretical performance of the *sync* version, which shows that BLP provides about 1.3× speedup over the theoretical *sync* version. However, UM with prefetching only provides about 0.68× the theoretical *sync* performance. Perhaps UM prefetching would improve with different hyper-parameters but its out-of-the-box performance is disappointing.

4.3.7 Static vs. Dynamic Task Scheduling

In the previous experiments, our BLP approach uses static task partitioning. The GPU thread blocks are partitioned into thread groups to handle host-to-device memory copies, main computation, and device-to-host memory copies. However, this static partition may not

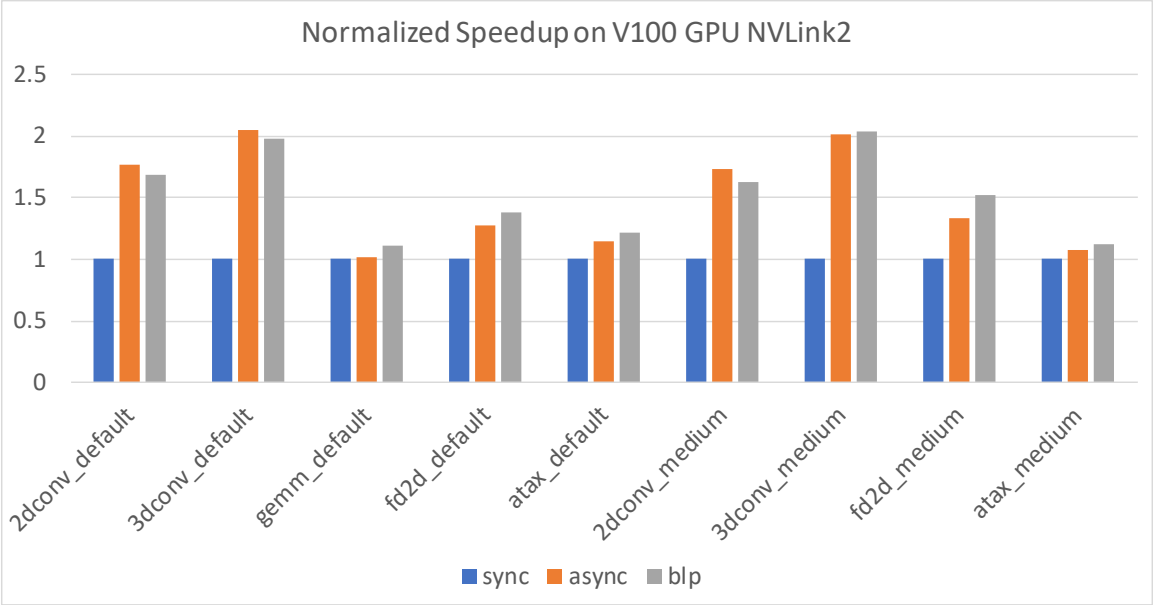

Figure 4.18: Overall Normalized Speedup (NVLink2)

fully utilize the resource, which may lead to load balance issues. Thus, we also implemented a preliminary version using dynamic thread block scheduling, which we denote as *blp-dynamic*. We first identify the data dependencies for each compute thread block. We then group them and assign a task ID for each group. After launching a persistent kernel, each thread block obtains its corresponding task ID using atomic operations, which determines whether it performs memory operations or computation.

Fig. 4.21 shows the normalized performance of matrix transpose and vector multiplication (ATAX) on the NVLink2 and PCI-E nodes using the default and medium data sets. Overall, we observe performance degradation for *blp-dynamic*. It only achieves 0.75× to 0.93× the performance of the *sync* version, which appears to be due to using too many thread blocks to copy data. In the previous experiments, BLP used only four thread blocks on the NVLink2 node and only one thread block on the PCI-E node for memory operations. We observe performance degradation using more thread blocks. Because of the limited bandwidth, these thread blocks compete to access memory and could instead be more productively used for

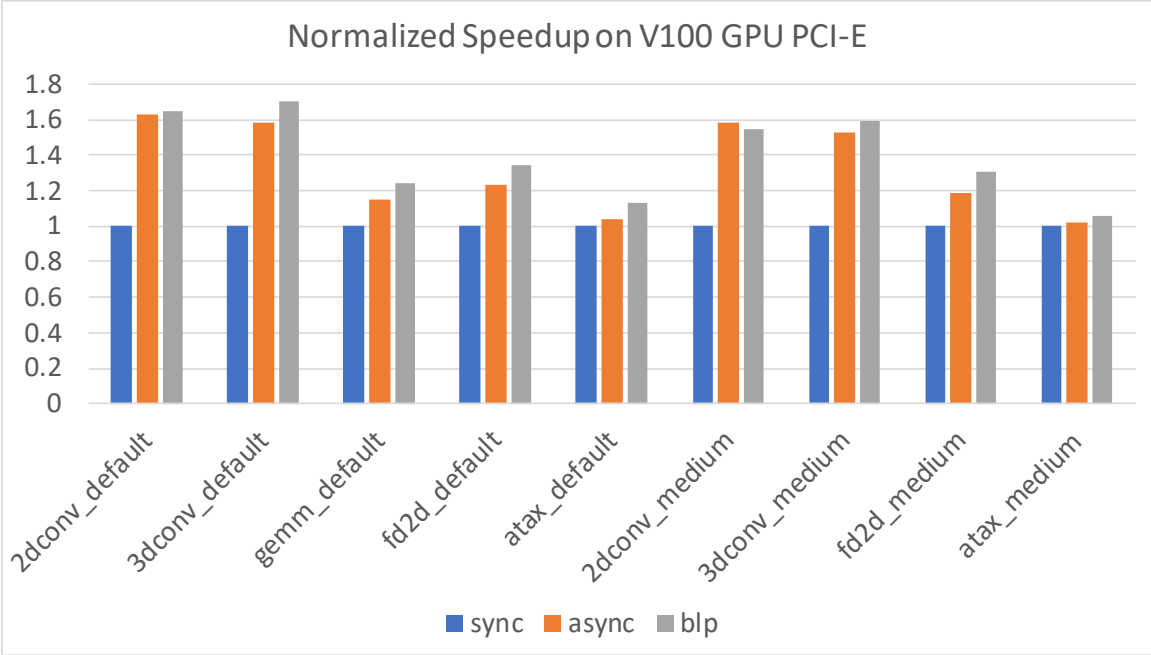

Figure 4.19: Overall Normalized Speedup (PCI-E)

computation. Our bandwidth experiments show that memory copy bandwidth with direct access is limited regardless of the total number of GPU threads. Thus, we can calculate the optimal number of thread blocks for data movement based on a system-wise bandwidth test. This choice avoids dynamic overhead and contention with more thread blocks while ensuring enough thread blocks perform memory operations.

4.3.8 Summary and Discussion

With cooperative thread groups, the number of thread blocks may exceed the number of SMs on the GPU. These thread blocks will execute concurrently. So we increase the number of thread blocks from 80 to 160 and observe about 2% average performance degradation for BLP. Our experiments show that we achieve peak data transfer bandwidth with four (NVLink2) or two (PCI-E) thread blocks for each direction, which is only 5% to 10% of the thread blocks without over-subscription. In general, we can determine the number of

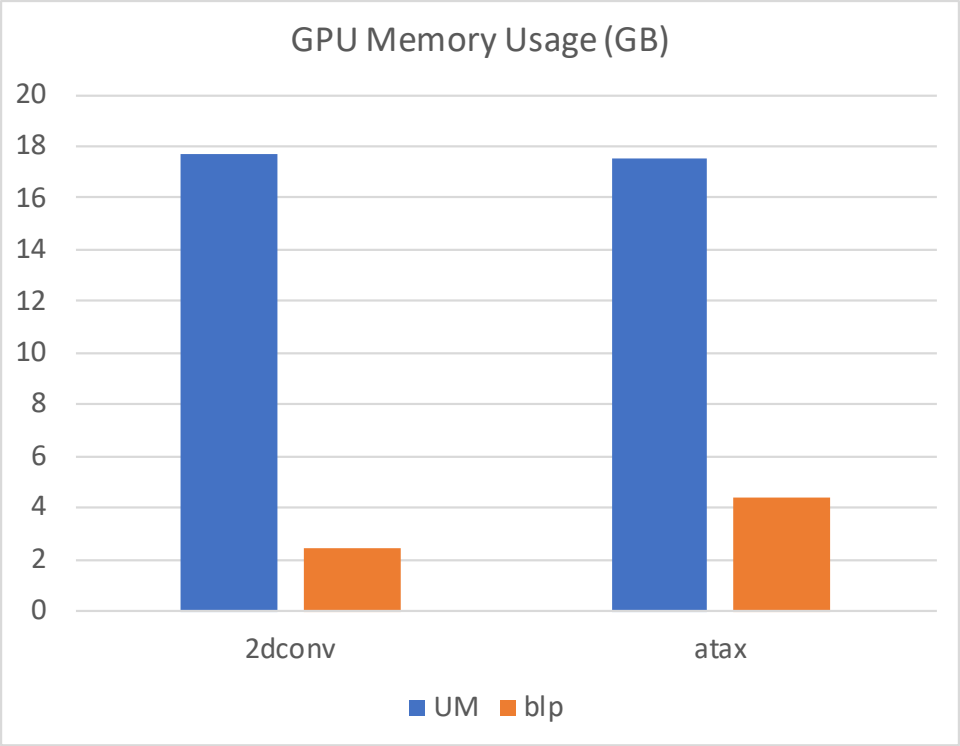

Figure 4.20: GPU Memory Usage Comparison

thread blocks required to achieve peak bandwidth for all applications on a given node with a simple experiment. As long as we require few thread blocks, using over-subscription will not improve data transfer bandwidth, but will use more compute resources, which is not a good choice. However, memory-latency-bound kernels, which are generally well-suited to GPUs, might benefit from the additional thread blocks.

Our approach focuses on overlapping device computation and data transfers between the host and device. We show that direct access and flag-based dependence tracking support an easy-to-use and efficient approach for both contiguous and non-contiguous data transfers. Our block-level pipelining approach can outperform the best hand-tuned traditional overlapping technique, which is based on GPU streams. With our approach, the CPU launches a single kernel and the GPU performs all data transfer and dependence checking. Our method also provides a better way to handle callbacks generated by the compute kernel (e.g., some data

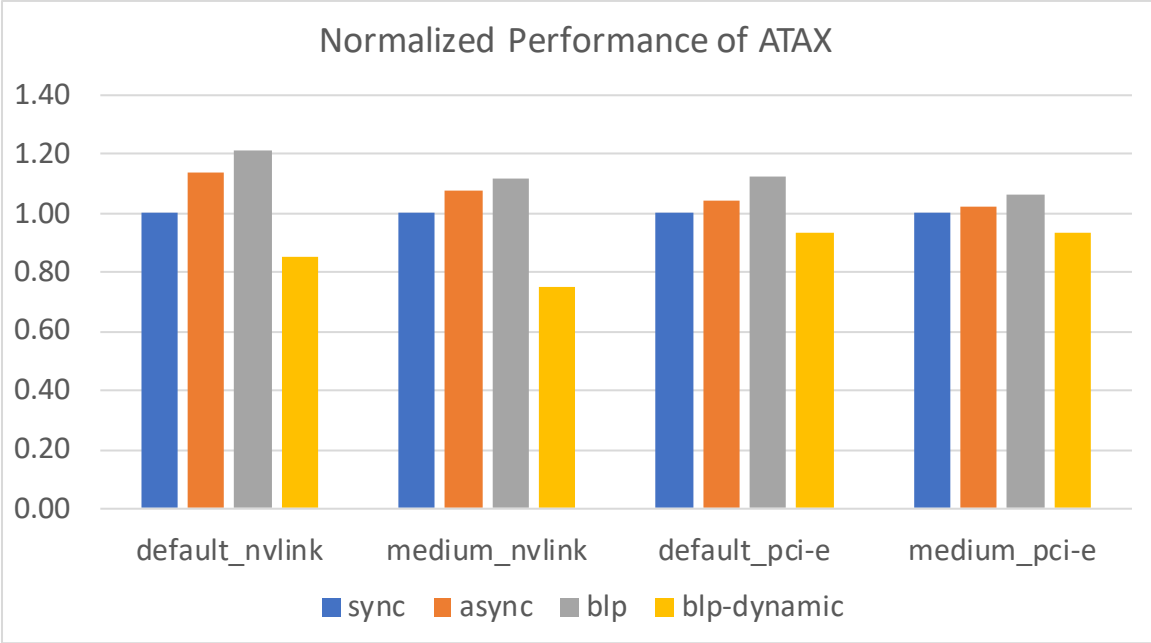

Figure 4.21: Performance Evaluation of Dynamic Task Scheduling for BLP

transfers are only conditionally required). Although we use some thread blocks to handle data transfers, CUDA cooperative thread groups could support more GPU thread blocks for latency-bound kernels. However, our results show that that support is often unnecessary.

Our BLP approach launches one kernel that handles the data transfer, computation, and task dependencies. The user does not need to split the task into multiple sub-tasks with extra API calls, which may cause significant launch overhead. Fig. 4.8 and Fig. 4.15 show that *async* with only 1 GPU stream suffers significant performance degradations as we split the task to more chunks. We see 50% to 90% performance slowdown with 1024 chunks. Even with 32 to 128 chunks, performance decreases 10% to 20%. Even though we still achieve good speedup from pipelining after mapping these sub-tasks to different GPU streams, the increased launch overhead consumed part of the benefit of pipelining. As the task may become large and complex, launch overhead may significantly increase and overwhelm the benefit of pipelining and impact the performance of unrelated CPU work. Our approach

removes the extra launch overhead while direct access can provide equal or even better bandwidth than CUDA memory copy APIs issued by the CPU, especially for non-contiguous accesses. Further, our approach enables overlap of finer-grained computation. With GPU thread blocks, we can control the order of the memory block copies. As soon as the data blocks are ready, we can start the corresponding computation tasks. CUDA memory copy APIs do not guarantee the memory copy order for large data blocks while breaking the copies into chunks decreases bandwidth significantly.

4.4 Conclusion

This work presents our block-level data pipelining (BLP) approach for GPUs. The GPU handles all tasks, including data transfers, computation, and tracking of task dependencies. The CPU only launches one kernel. We leveraged multiple techniques to optimize BLP. Our results show that, with no manual tuning, our approach provides performance comparable to the best hand-tuned manual pipelining on Nvidia V100 GPUs. With proper index mapping, our flag mechanism also significantly reduces GPU memory usage while delivering competitive performance, enabling BLP to support GPU memory over-subscription. We also demonstrated that direct memory access can provide equal or higher bandwidth than CUDA memory copy APIs, especially when dealing with non-contiguous data. We also designed an OpenMP extension that supports our approach, which allows developers to pipeline data transfers without major code refactoring.

Chapter 5

IterML: Iterative Machine Learning for Intelligent Parameter Pruning and Tuning in Graphics Processing Units

5.1 Introduction

Heterogeneous computing with accelerators, particularly GPUs, has become increasingly prominent in the Top500 List [25] as well as in embedded high-performance computing (HPC) systems, like those found in smartphones and smart cars. In such systems, the host CPU manages the execution context while computation is offloaded to an accelerator. Leveraging accelerators not only enables high performance, but it also improves energy efficiency [54]. However, extracting the optimal performance and energy efficiency from these accelerators can be extraordinarily difficult for a software developer [35]. Thus, developers need simpler abstractions and underlying mechanisms to program these accelerators [21, 49] as well as significant domain knowledge to tune the performance of the code on these accelerators [33, 37].

Because heterogeneous architectures with accelerators expose many software and hardware parameters for developers to tune to achieve optimal performance, the different combinations of parameters result in an enormous search space, making it infeasible for developers to test

each combination of parameters exhaustively. Furthermore, choosing the wrong combination of parameters can result in severe performance degradation. As such, this paper presents IterML, our iterative parameter pruning and tuning approach with machine learning (ML). During each iteration, we use ML models to assist with the pruning (and tuning) of the search space by their predicted performance [22, 23].

In all, our research contributions are as follows:

- The design of an iterative machine-learning (IterML) approach that automatically determines nearly optimal parameter settings for the GPU thread-block size to achieve high performance.
- An empirical study that illustrates how our IterML approach consistently delivers better search speed over non-iterative ML methods and achieves nearly optimal performance while sampling only 1.5% of the search space on average and, in turn, reducing the search effort by 40%-80%. In addition, when compared to the PGI 17.5 compiler, IterML also delivers about a 50% improvement in performance by automatically identifying a nearly optimal GPU thread-block size.

5.1.1 Motivating Example

Figure 5.1 shows a performance heatmap of our lid-driven cavity (LDC) code,[1] where the GPU thread-block size is varied (i.e., $\mathsf{blockDim.x} \times \mathsf{blockDim.y} \leq 512$) when running on an Nvidia V100 GPU. The x- and y-dimensions are limited to 64, and each thread block contains at most 512 threads. We observe that the performance varies significantly across different thread-block sizes. At the ideal thread-block size of 4x32 for *this* code on *this* GPU, the V100 achieves 893.3 GFLOPS. On the other hand, the performance can be 33% worse

[1] A well-known computational fluid dynamics (CFD) problem for viscous incompressible fluid flow.

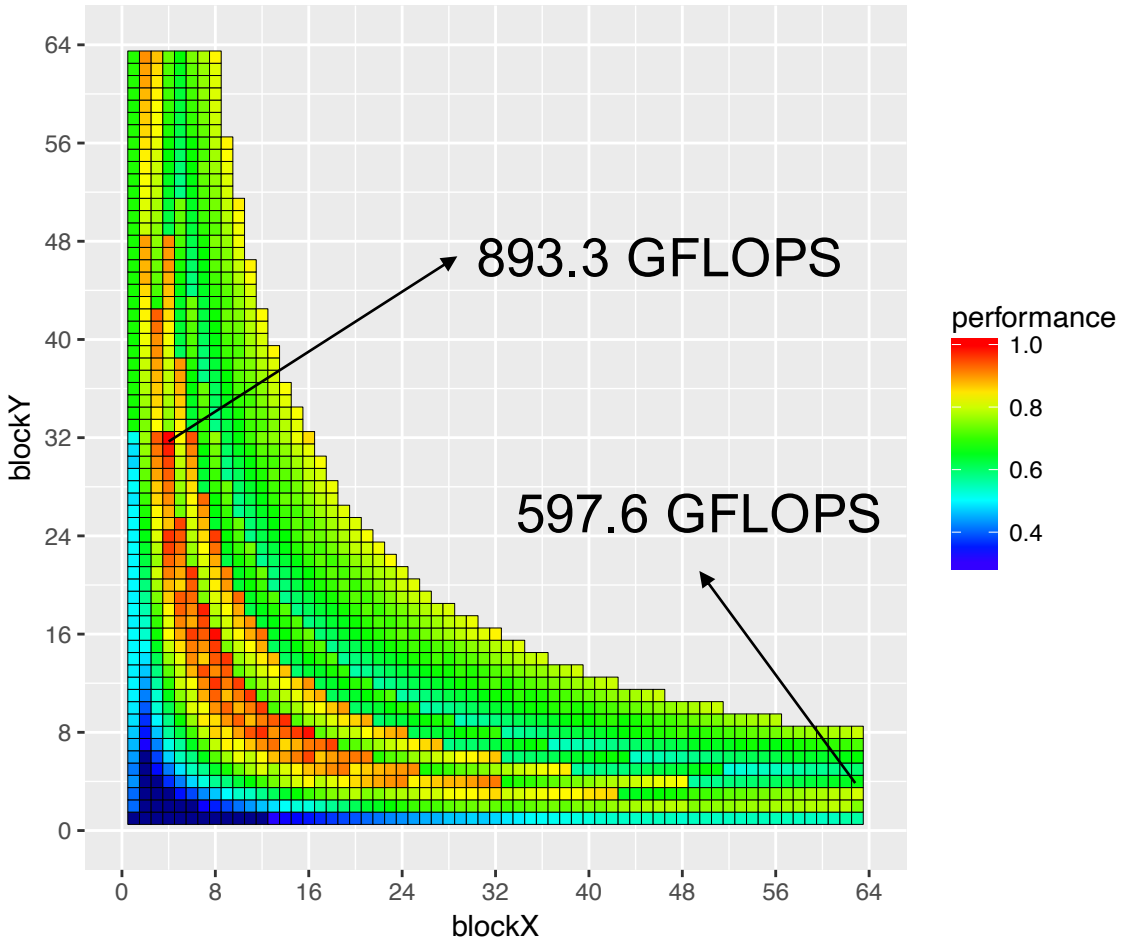

Figure 5.1: Performance of a Lid-Driven Cavity Code with Varying GPU Thread-Block Size on an Nvidia V100 GPU

at 597.6 GFLOPS if the default thread-block size of the PGI 17.5 OpenACC compiler for this LDC code is chosen, namely 64x4.

This 64x4 thread-block size only delivers approximately 67% of the optimal performance. In addition, exhaustively generating the performance heatmap in Figure 5.1 is tedious and time-consuming.

Figure 5.2 shows the performance heatmap of a two-dimensional (2D) convolution benchmark. This heat-map looks significantly different from that in Figure 5.1. Thus, different

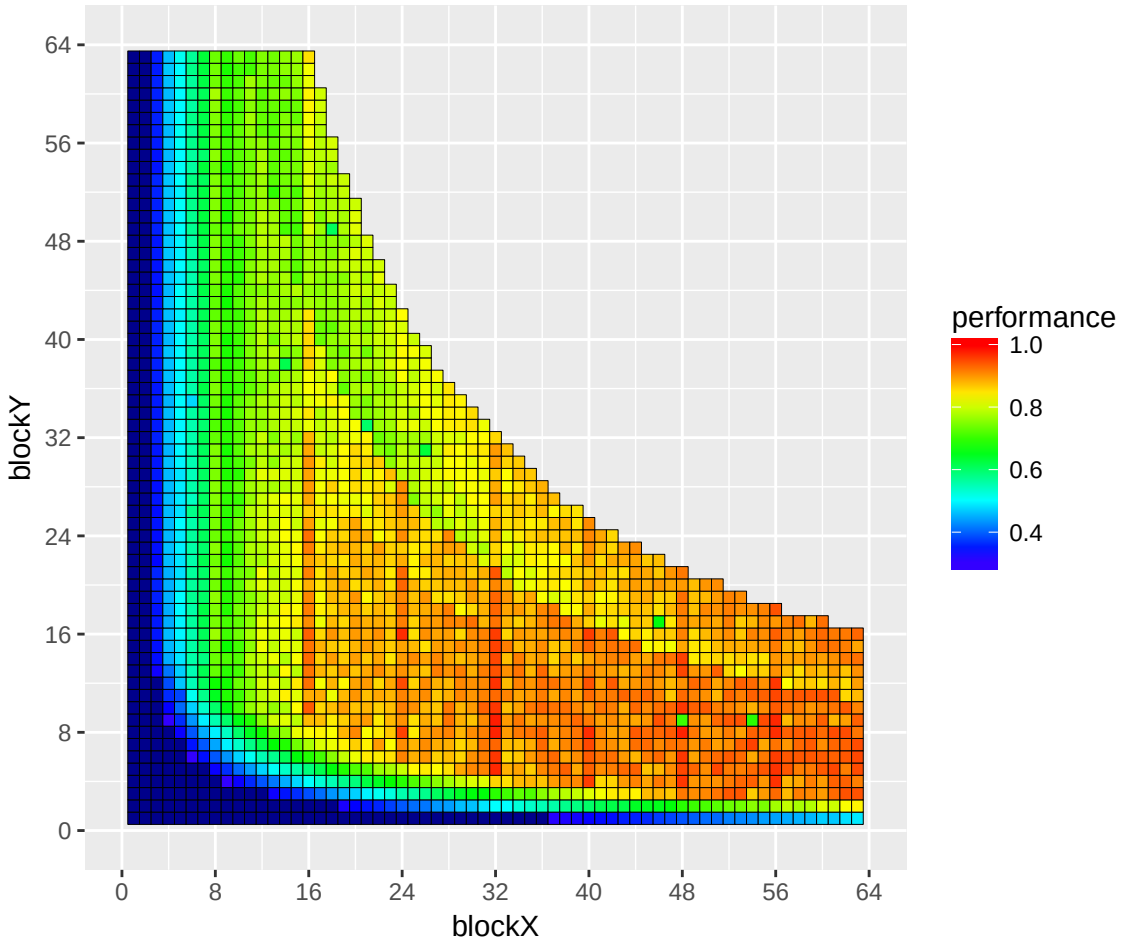

Figure 5.2: Performance of the 2D Convolution Benchmark with Varying GPU Thread-Block Size on an Nvidia V100 GPU

codes on the same GPU can produce widely different performance characteristics, relative to GPU thread-block size.

The GPU thread-block size is just one parameter that can be tuned. Many other potential parameters that could be tuned, e.g., GPU block size, degree of loop unrolling, and register usage limitation. These assortment of parameters massively increase the search space, making it infeasible to exhaustively enumerate every combination. Thus, we need a simpler and more efficient approach to identify ideal parameter settings for (near-)optimal performance.

5.2 Approach and Design

Here we articulate the approach and design of our iterative machine learning (IterML) [22, 23], including the selection of the parameter search space, the iterative machine-learning (pruning) algorithm itself, and the regression models to predict the rest of the search space.

5.2.1 Choosing the Parameter Search Space

For microbenchmarks or libraries, a set of hyperparameters that define the dimensions of the tuning search space must be identified. The hyperparameters may (1) relate to software (e.g., input partition chunks and thread count) or hardware (e.g., active core count, GPU thread-block size, and compiler optimization options) and (2) be either binary in nature (e.g., turning on/off a compiler flag) or multi-valued across a range (e.g., thread-block size or number of partition chunks). Our iterative machine-learning (IterML) approach builds knowledge based on machine-learning (ML) models as it uses samples from one iteration to then look for potentially better samples in subsequent iterations.

5.2.2 Iterative Machine-Learning (IterML) Algorithm for Pruning and Tuning

In order to reduce (i.e., to prune and to tune) the search space quickly and effectively, we propose an iterative machine-learning (IterML) algorithm, as shown in Algorithm 5.3.

As inputs, the algorithm takes the *search space* D, specified by multiple design parameters (e.g., thread-block dimension); a *pick ratio*, which is the sample ratio that needs to be tested in each iteration; a *cut ratio*, which sets the ratio of the space to be pruned in each iteration; and a *model*, which is the regression model used for prediction. Once the pick

ratio is selected, the number of samples that we pick to test in each iteration stays constant. For each iteration, we first apply the regression model to the samples. We then predict the performance of the residual search space and drop the lowest performing ones by the cut ratio, e.g., 50%. This process repeats until a stopping criteria is met. For example, if the cut ratio is 50%, then after every iteration, the search space is halved, which, in turn, means that the number of iterations is log_2(size of search space). We may adjust the cut ratio based on the size of the original search space.

Algorithm 1: Iterative Machine-Learning Algorithm (IterML)

Input : **D:** search space specified with n design parameters
Pick ratio: sample ratio taken in each iteration
Cut ratio: ratio of the space pruned each iteration
Model: regression model chosen for prediction

Result: Best parameter combination currently found

initialization;
while *while not meeting stopping criteria* **do**
  pick sample set **S** randomly from remaining **D**, based on the pick ratio;
  gather performance **P** of **S**;
  build model M_i, based on **P**;
  predict the remaining **D** based on model M_i;
  prune from the remaining **D** the samples with low predicted performance by an amount specified by the **cut ratio**;
end

Figure 5.3: Iterative Machine-Learning Algorithm (IterML)

Figure 5.4 shows basic workflows for non-iterative ML and our iterative ML (IterML) with three iterations. We observe that a small portion of samples are randomly chosen in iteration 0 (i.e., *iter*0). Then the model is generated to use as a guide for subsequent iterations for data point selection in the residual search space. The corresponding performance of these data points are measured and then utilized to further build models for next iterations.

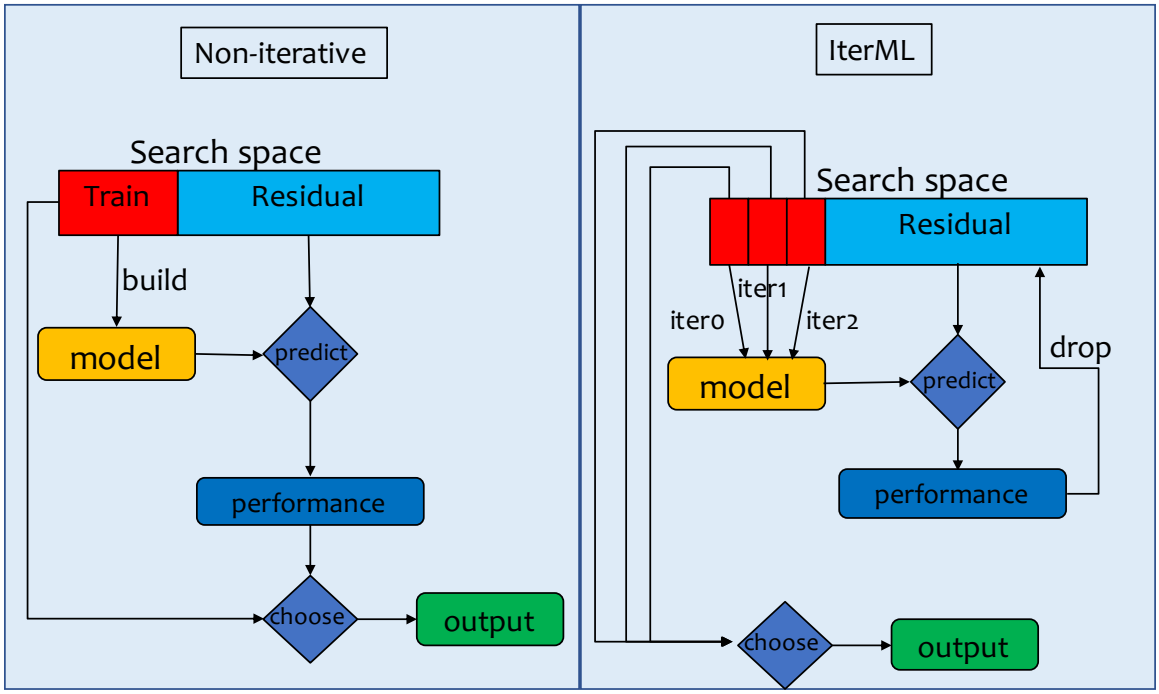

Figure 5.4: Comparison of Non-Iterative and Iterative Machine Learning

Finally, the best data point should be selected as the output result.

5.2.3 Regression Models

Based on our iterative machine-learning (IterML) approach [22, 23], we need to build a model during each iteration to predict the *rest* of the search space. We study and explore the use of the following five popular ML models to support our IterML algorithm.

Classification and Regression Trees (CART): Decision trees can be represented as a binary tree, where each node represents a single input variable (x) and a split point on that variable (assuming the variable is numeric). CART is typically fast to train and very fast to make predictions. It requires no data pre-processing and can be accurate for a broad range of problems.

K-Nearest Neighbors (KNN): Predictions are made for a new data point by searching through

the entire training set for the K most similar instances (i.e., neighbors) and summarizing the output variable for those K instances. We use the mean output variable as the result for regression problems.

Support Vector Machine (SVM) Regression: SVMs use a hyperplane to split the input variable space. The support vector regression (SVR) uses the same principles as the SVM for classification, with only a few minor differences. In the case of regression, a margin of tolerance (i.e., ε) is set in approximation to the SVM, which would have already requested from the problem.

Random Forest (RF): This type of ensemble ML algorithm is called bootstrap aggregation or bagging. Multiple samples of training data are taken; models are then built for each data sample. When a prediction for new data needs to be made, each model makes a prediction, and the predictions are averaged to give a better estimate of the true output value. Combining predictions from these models results in a better estimate of the true underlying output value.

Multilayer Perceptron (MLP):

This model is a neural network that connects multiple layers in a directed graph, which means that the signal path through the nodes only goes one direction. Each node, apart from the input nodes, has a nonlinear activation function. MLP utilizes a supervised learning technique called back propagation for training, which has drawn significant interest recently due to its success in deep learning.

5.3 Experiments

To evaluate our iterative machine-learning (IterML) approach, we leverage different ML models while pruning the search space. To demonstrate the efficacy of our approach, we focus on the *GPU thread-block size* as the hyperparameter of interest. The GPU thread block is typically composed of an X-dimension and Y-dimension. Each dimension ranges between 1 and 1024, inclusive. The product of the two dimensions, which is the thread number of each GPU thread block, is also limited by 1024. To identify the GPU thread-block size that delivers the best (optimal) performance as a reference point to which to compare, we exhaustively test the performance of different benchmarks using all possible combinations of GPU thread-block size, a process that takes *days* to complete. We then demonstrate the speed and efficacy of our IterML pruning and tuning approach on the GPU thread-block size and evaluate its subsequent performance relative to the optimal performance.

5.3.1 Benchmarks Studied

As shown in Table 5.1, we use nine (9) GPU kernels from the Polybench suite [64], an OpenACC kernel from the EPCC benchmark suite [41], and an OpenACC kernel from our lid-driven cavity (LDC) code to conduct our experiments. The kernels use various GPU functional units and exhibit diverse behavior. For CUDA benchmarks, relevant design parameters are substituted by C macros so that our design can easily modify and recompile them. For the OpenACC benchmarks, we pass variables using the compiler flags to modify the OpenACC pragma.

Each kernel is executed 10 times and the average execution time reported. Only the GPU time of each kernel execution is measured and used, thus excluding any data transfer overhead. Because we think the thread block size only affects kernel execution time on GPU.

Table 5.1: Benchmarks Used

Benchmark	Description
2dconv	2-D convolution
3dconv	3-D convolution
2mm	2 matrix multiplications
3mm	3 matrix multiplications
gemm	Matrix-multiply C=alpha.A.B+beta.C
gesummv	Scalar, vector and matrix multiplication
mvt	Matrix vector product and transpose
syr2k	Symmetric rank-2k operations
syrk	Symmetric rank-k operations
epcc	EPCC 27Stencil benchmark
ldc	Lid-driven cavity code

5.3.2 Hardware Platform

We conduct our experiments on two Nvidia GPUs: Tesla P100 and Tesla V100. Our P100-based node contains two 2.4-GHz Intel E5-2680v4 CPUs for a total of 28 CPU cores per node. The V100 node pairs two 3.0-GHz Intel Skylake Xeon Gold CPUs for a total of 24 cores per node, In addition, there is 384 GB of memory and two Nvidia V100 ("Volta") GPUs per node. Both V100 and P100 GPUs are connected to CPUs using PCI-E 3.0.

5.3.3 Dataset Analysis

We measure the performance of 11 benchmarks while varying the GPU thread-block size. We then generate a performance heatmap for each benchmark and conduct a preliminary data analysis. We find that the performance distribution of these benchmarks typically fall into two major categories: clustered or banded.

Clustered. Most of these benchmarks achieve higher performance when a specific GPU thread-block dimension gets higher (or lower) values, i.e., "clustered" high performance. As shown in Figure 5.2 and Figure 5.5, we observe better performance with higher blockX or

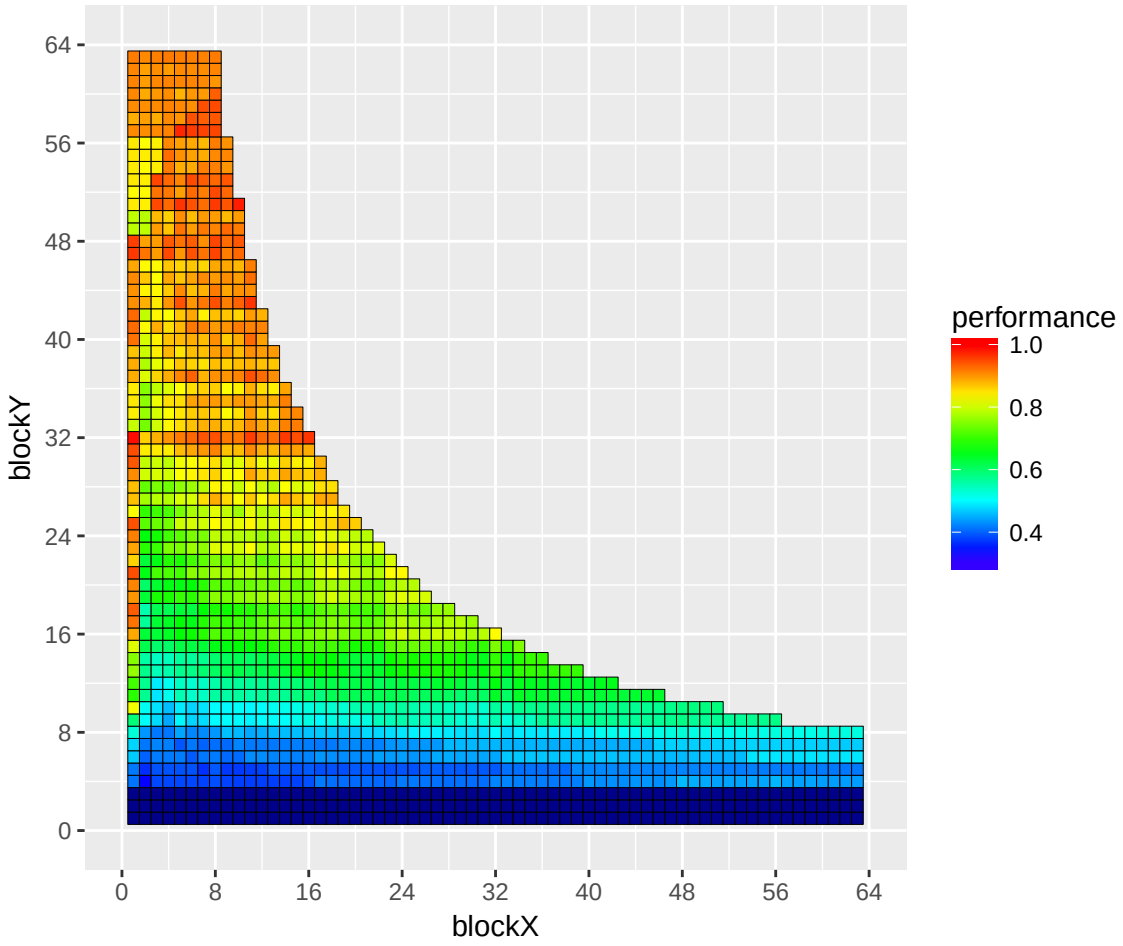

Figure 5.5: Performance of the EPCC Benchmark with Varying GPU Thread-Block Size on an Nvidia V100 GPU

blockY, respectively. In Figure 5.6, we see performance improvement with small blockY values.

Figure 5.7 shows the SYR2K benchmark on the V100 GPU. The highest-performing configurations are those with a blockX equal to eight (8). The performance then degrades gradually as one moves away from the blockX = 8 line. When the performance of a benchmark changes gradually along one axis (e.g., blockX or blockY) like this, we refer it as a "1D Cluster."

Similarly, other benchmarks deliver high performance when the *product of blockX and blockY*

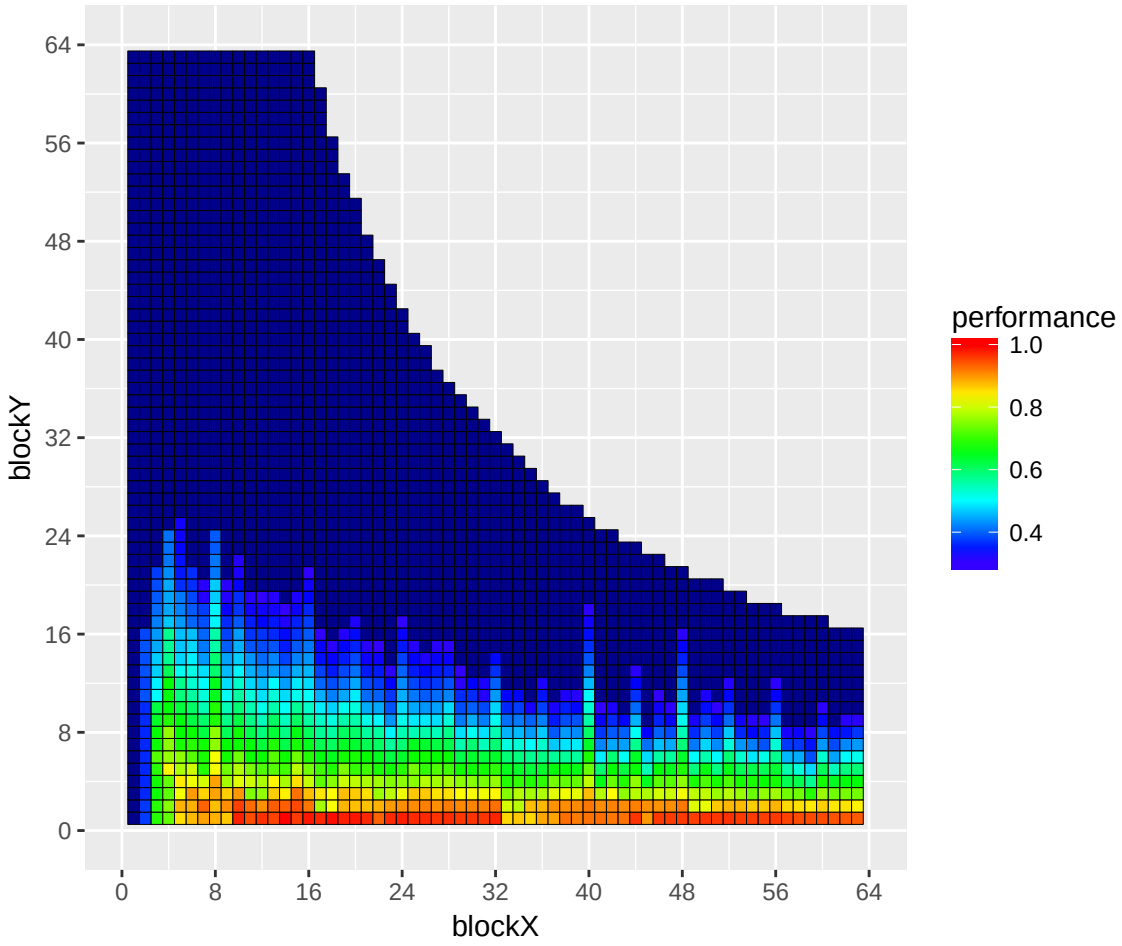

Figure 5.6: Performance of the MVT Benchmark with Varying GPU Thread-Block Size on an Nvidia P100 GPU

reaches a specific number or range, meaning that the total number of GPU threads within a thread block should be limited in order to achieve the best performance. Visually, this translates into a "locus" of high performance, where performance degrades as thread-block size moves away from the "center of the locus." For example, as shown in Figure 5.1, the best-performing thread-block configurations occur when the total number of GPU threads in a thread block is about 128.

Finally, Figure 5.8 shows that the peak performance for the GESUMMV benchmark on a V100 GPU occurs near the lower-left corner of the heatmap, which means that many GPU

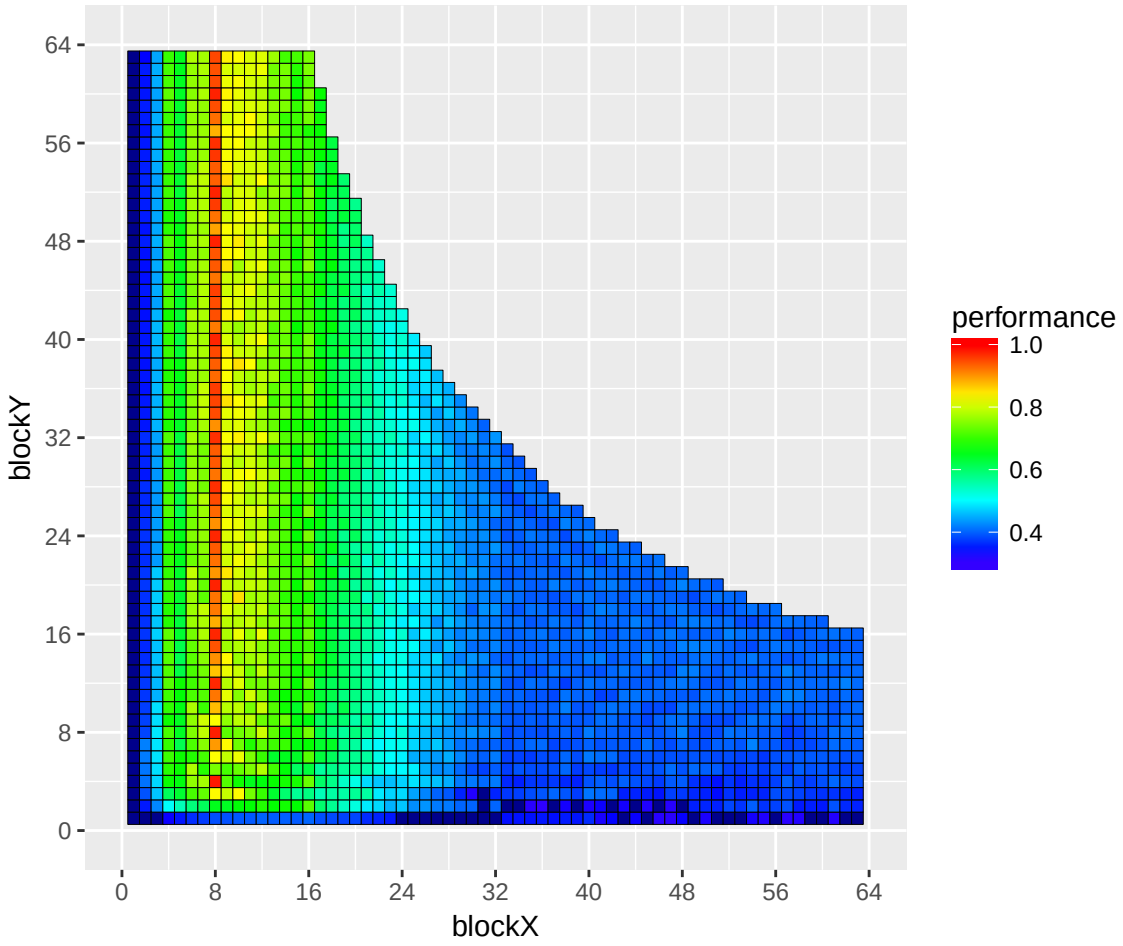

Figure 5.7: Performance of the SYR2K Benchmark with Varying GPU Thread-Block Size on an Nvidia V100 GPU

thread blocks with a small number of GPU threads in each thread block delivers the best performance.

Banded. The banded performance distribution occurs for GPU programs that deliver peak performance when a specific GPU thread-block dimension reaches a specific number or multiple of it. As the thread-block size moves away from these specific numbers, performance degrades significantly. As shown in Figure 5.9 and Figure 5.10, the 2MM and GEMM benchmarks deliver the best performance only when blockX is a multiple of 16 and 8, respectively. For other values of blockX, the performance achieved is always below 60% of

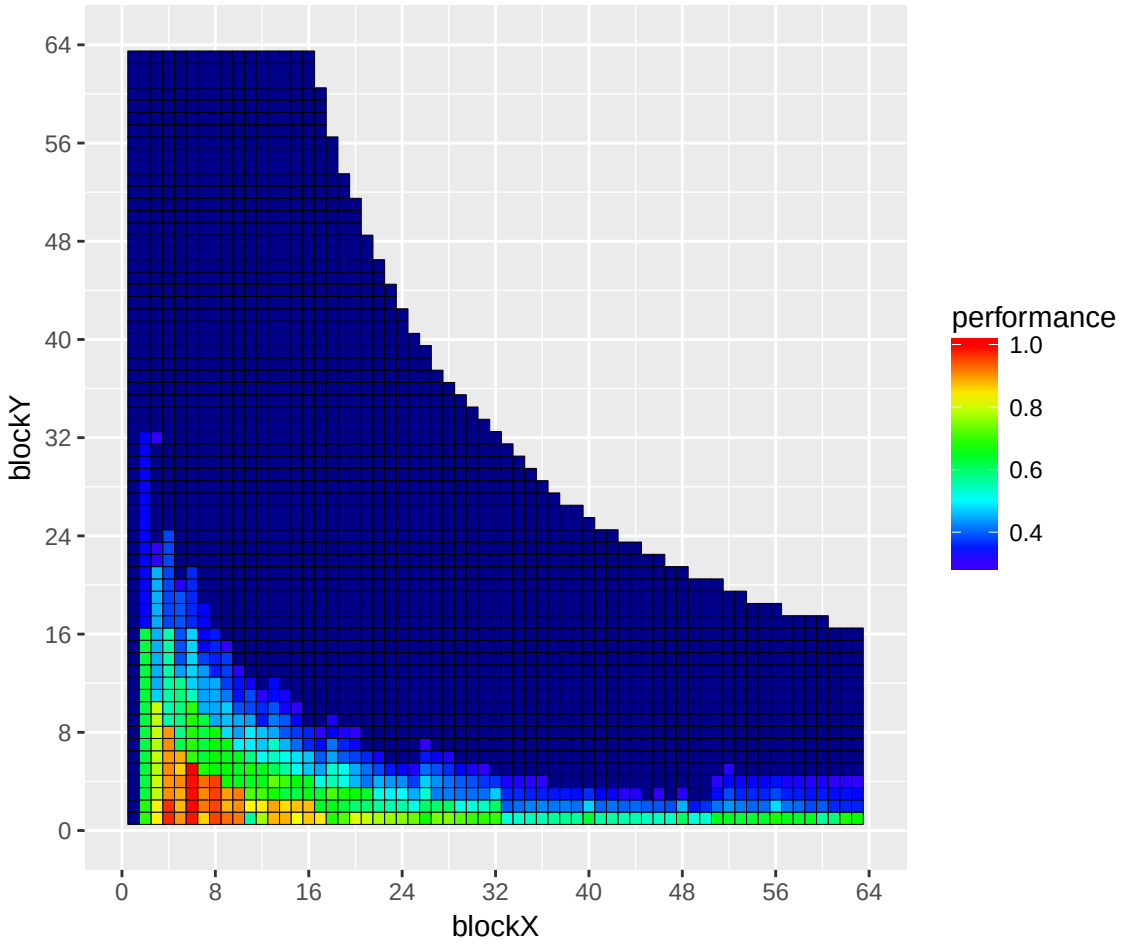

Figure 5.8: Performance of the GESUMMV Benchmark with Varying GPU Thread-Block Size on an Nvidia V100 GPU

the performance achieved when blockX is a multiple of 16 and 8 for the 2MM and GEMM benchmarks, respectively.

Using the performance classifications of "clustered" and "banded," Table 5.2 shows the overall distribution of application performance when varying the GPU thread-block size. Interestingly, the performance distribution across the P100 GPU and V100 GPU is consistent. As a consequence, this feature might be useful as a guideline for further experimental work on future accelerators.

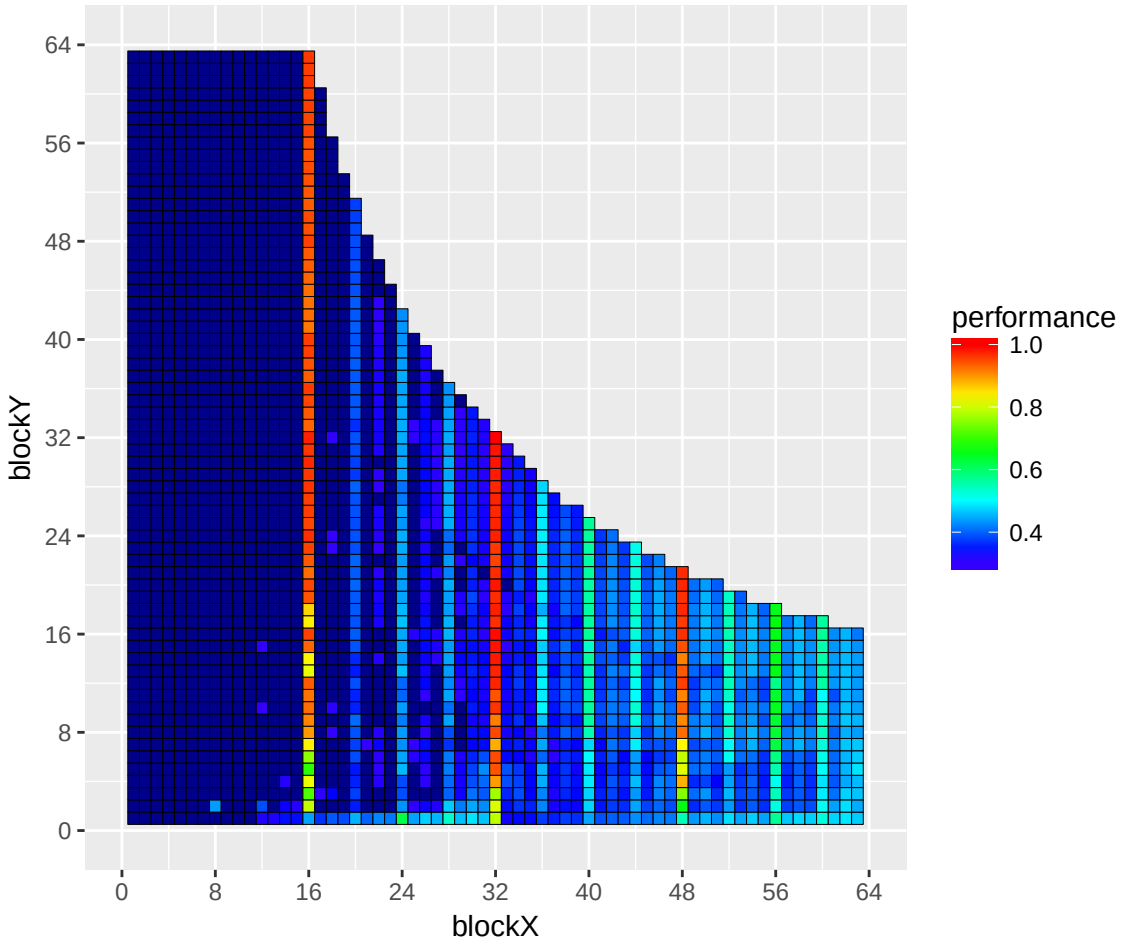

Figure 5.9: Performance of the 2MM Benchmark with Varying GPU Thread-Block Size on an Nvidia P100 GPU

5.3.4 Pruning Procedure

We evaluate our iterative pruning approach [22, 23] after we collect all performance data of the 10 benchmarks and applications by varying the GPU thread-block size. The pruning approach is implemented in Python with the Scikit-learn machine learning (ML) libraries [62]. We select five (5) commonly used models to predict the performance based on previous samples; these models include CART, KNN, SVM, RF, and MLP, as presented in §5.2.3. Because the thread-block search space is relatively modest, we use 0.5 as the cut ratio, which

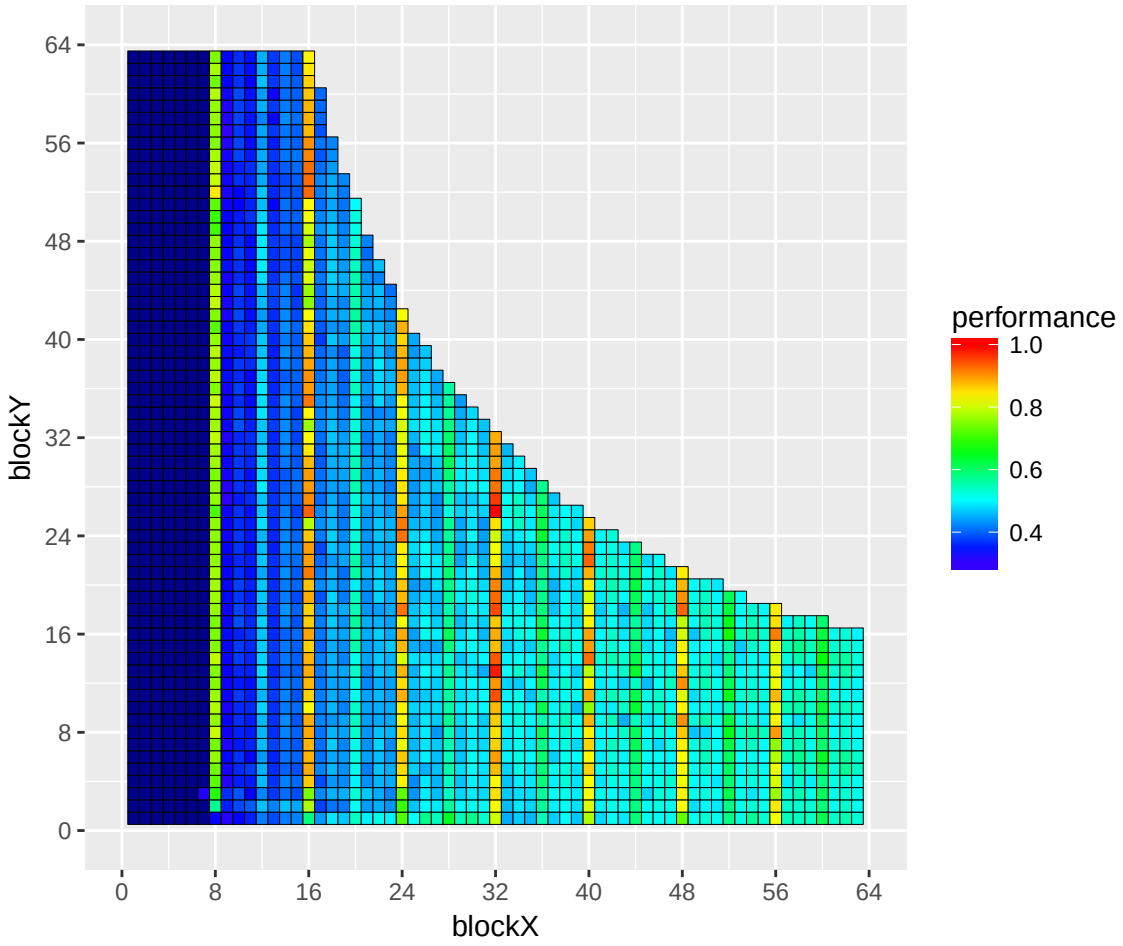

Figure 5.10: Performance of the GEMM Benchmark with Varying GPU Thread-Block Size on an Nvidia V100 GPU

means that IterML drops 50% of the search space that has low predicted performance in each iteration. We can tune this number depending on the scale of the search space. Each time, we pick a portion of the sample based on the pick ratio and keep this sample number consistent until end. Due to the random selection of the samples, we repeat this process at least 100 times, thus drawing a distribution of the results using our iterative pruning approach. We then compare the result of different models based on this distribution. We normalize the samples to the result of the baseline, which entirely randomly selected the samples.

Table 5.2: Distribution of Application Benchmark Performance

	P100	V100
Clustered	2dconv epcc 3dconv ldc gesummv mvt syrk syr2k	2dconv epcc 3dconv ldc gesummv mvt syrk syr2k
Banded	2mm 3mm gemm	2mm 3mm gemm

5.3.5 Case Studies

Here we present case studies to show how the selection of the GPU thread-block size affects performance.

How does the manual selection of the thread-block size by an experienced developer impact performance? In real-world GPU coding, developers set the GPU thread-block size based on their experience (rather than relying on a default setting chosen by the compiler). Typically, the chosen block sizes are 64, 128, and 256. Figures 5.11 and 5.12 show the normalized performance of benchmarks, where the GPU thread-block size is set by experienced developers, relative to the optimal performance. We observe that none of the manually chosen thread-block sizes deliver consistently high performance across all benchmarks. In fact, some settings achieve as low as 20% of the optimal performance across the benchmarks.

Does a universal thread-block size "rule them all"? Figures 5.13 and 5.14 show the minimum normalized performance across all benchmarks tested on the V100 and P100 GPUs, respectively. For brevity, we only show the cases where the total number of threads ≤ 512

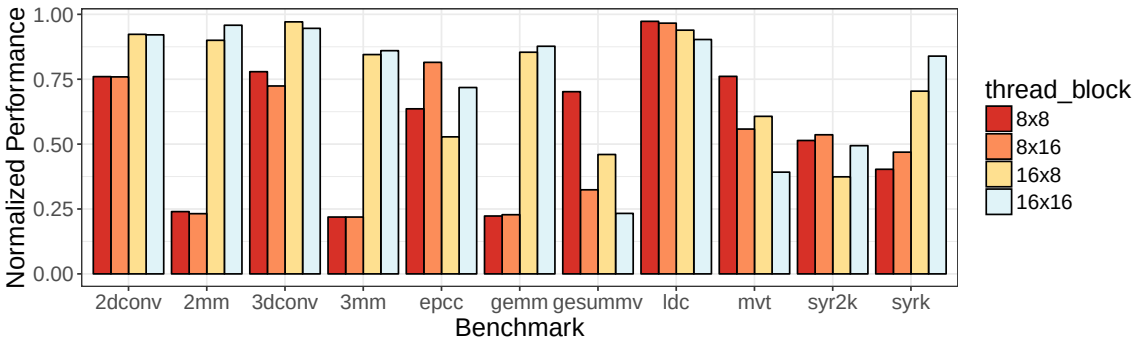

Figure 5.11: Normalized Performance Across Varying Thread-Block Sizes on the P100 GPU

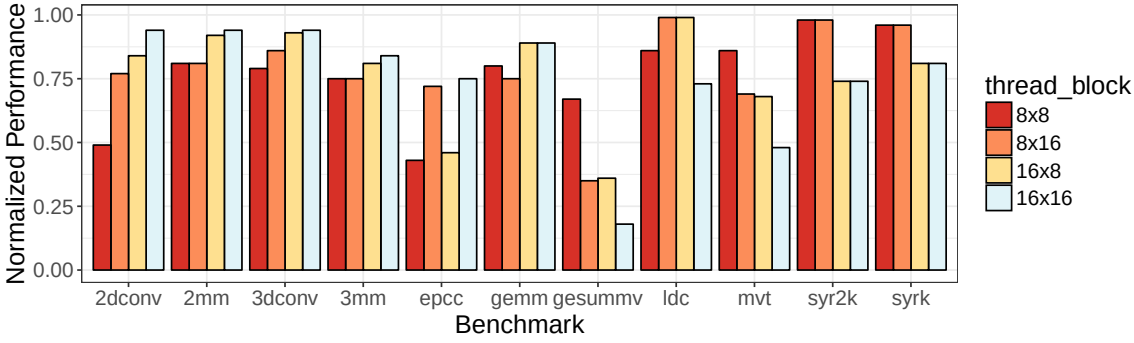

Figure 5.12: Normalized Performance Across Varying Thread-Block Sizes on the V100 GPU

and where each dimension ≤ 128. (Note: The omitted parts of the graphs show similar results.) Overall, we observe that all blocks in the heatmap achieve less than 30% of optimal performance across the tested application benchmarks. Thus, no such universal thread-block size can provide high performance across applications.

Is the ideal thread-block size for one device good enough for other devices with similar architectures? We first identify the ideal thread-block size that achieves the best performance on one device (e.g., P100) and then see if that thread-block size is also good enough for another device (e.g., V100) with a similar architecture.

Table 5.3 shows how the ideal thread-block size on the P100 GPU performs on the V100 GPU while Table 5.4 shows the converse. For some benchmarks (e.g., 2mm), the ideal block size on the P100 delivers 99% of the optimal performance on the V100. However, a significant

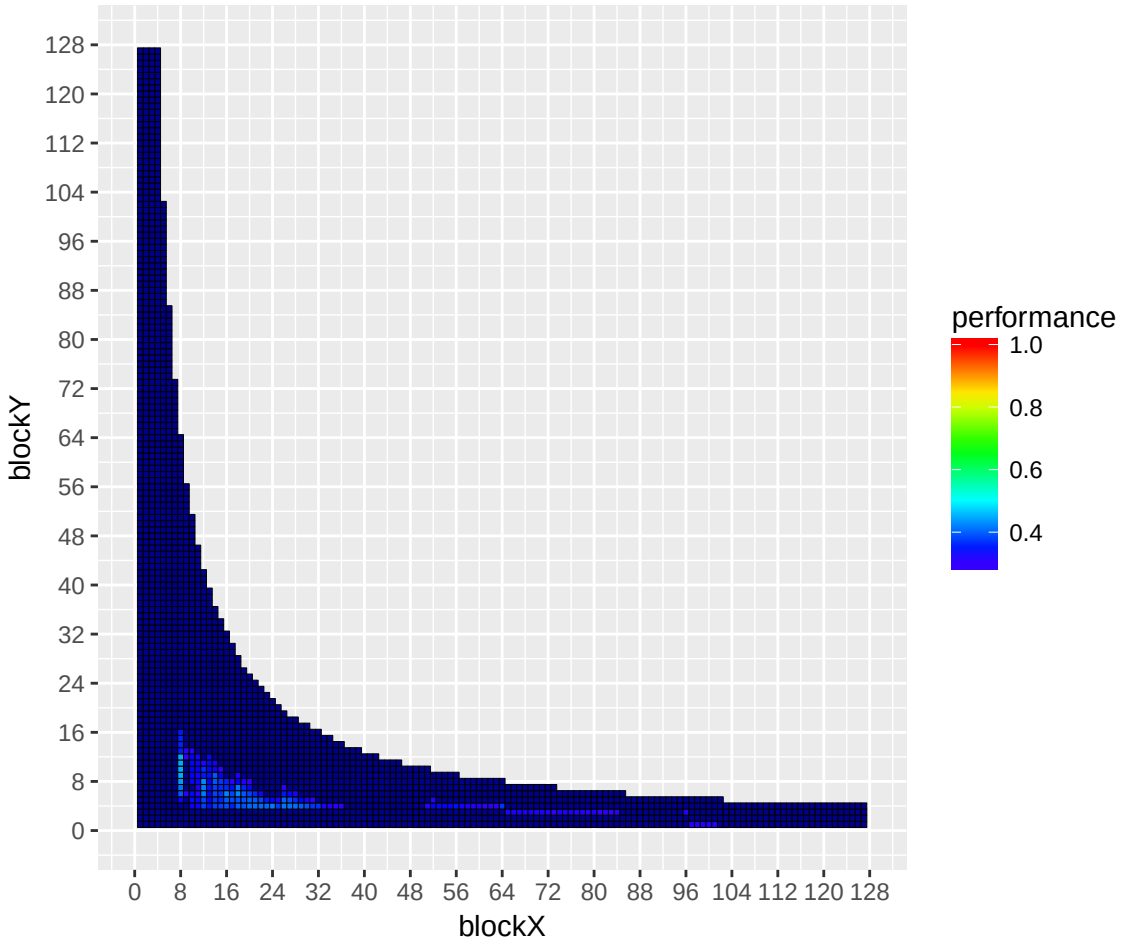

Figure 5.13: Minimum Performance Heatmap Across All Benchmarks on V100 GPU

number of the tested benchmarks (e.g., SYRK) achieve as little as 44%-47% of the optimal performance. Thus, the ideal thread-block size for one device may not be good enough for other devices with similar architectures.

5.4 Evaluation

We evaluate the total sample ratio needed to achieve high performance using our iterative machine-learning (ML) approach for pruning and tuning. We vary the ML models used for

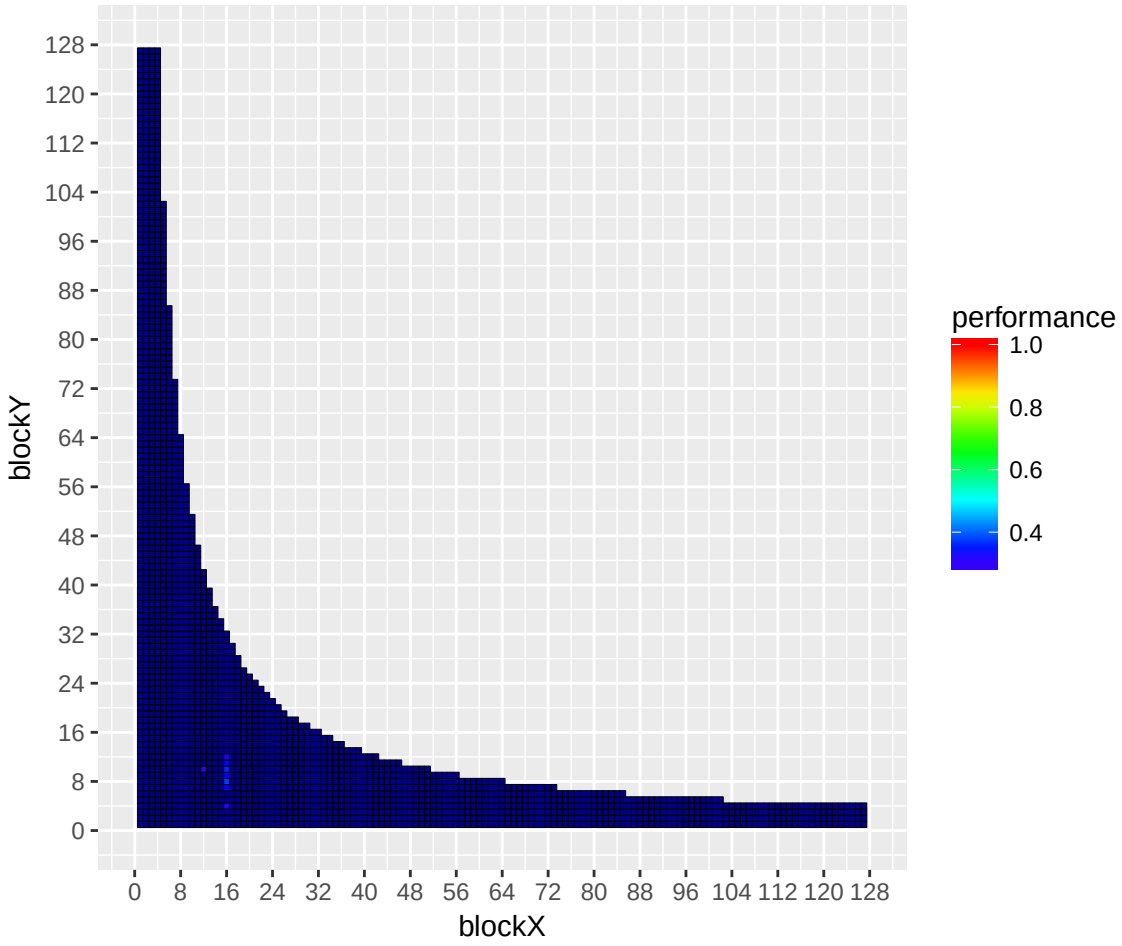

Figure 5.14: Minimum Performance Heatmap Across All Benchmarks on P100 GPU

prediction and use random search sampling as our baseline. To quantify the goodness of the performance compared to the optimal, we utilize the following two standards:

1. **Standard 1:** The sample ratio required to achieve a median that is higher than 95% of optimal performance. It means that the result is expected to be at least better than 95% of the optimal performance on average.

2. **Standard 2:** The sample ratio required to achieve 5-percentile *higher* than 95% of the optimal performance. Standard 2 is a more difficult standard to achieve than Standard 1. It means that there is at least a 95% probability to get a result that is better than

Table 5.3: P100 Ideal Thread-block Size on V100

Benchmark	Thread-X	Thread-Y	Performance
2dconv	512	2	0.974
3dconv	96	2	0.946
2mm	32	32	0.99
3mm	16	48	0.76
gemm	16	48	0.775
gesummv	16	1	0.868
mvt	14	1	0.903
syr2k	16	36	0.751
syrk	32	24	0.473
epcc	1	128	0.968
ldc	16	6	0.954

Table 5.4: V100 Ideal Thread-block Size on P100

Benchmark	Thread-X	Thread-Y	Performance
2dconv	64	5	0.941
3dconv	88	10	0.846
2mm	64	16	0.994
3mm	32	26	0.938
gemm	32	26	0.938
gesummv	6	5	0.675
mvt	8	1	0.883
syr2k	8	124	0.601
syrk	8	124	0.447
epcc	8	64	0.778
ldc	4	32	0.945

95% of the optimal performance.

Figure 5.15 shows a box-and-whisker plot for the performance of the SYR2K benchmark on a V100 GPU. The performance is tuned by IterML using the random forest model and varying the total sample ratio used. Each bar in the box plot consists of at least 500 data points. The middle line of each bar represents the median (for *Standard 1*); the lower-bound labels the 5th percentile (for *Standard 2*). We only show the first three iterations in this plot as the data set is relatively small in this case. The red color represents the baseline (non-iterative)

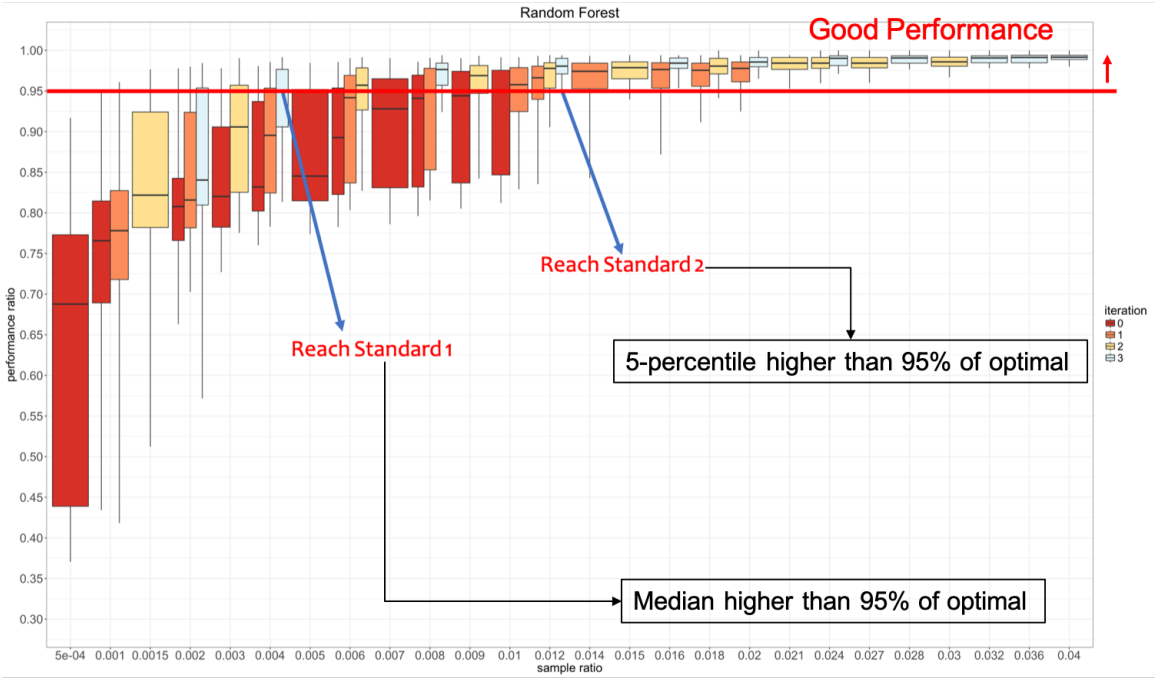

Figure 5.15: Performance of SYR2K Benchmark with Random Forest (RF) and Varying the Number of Iterations (see legend) Using IterML and the Total Sample Ratio (X-axis) on the V100 GPU

results, which is based on the random selection of samples. The figure clearly shows that, with the same number of total samples used, those using more iterations generally produce better performance. Thus, *our IterML approach performs better by using fewer samples per iteration but with more iterations.* However, the caveat is that some ML models may require a certain minimum number of samples (per iteration) in order to be accurate for prediction.

Figure 5.15 also shows that IterML with three iterations (i.e., light blue-grey bars) reaches *Standard 1* with a total sample ratio of only 0.004 while the traditional, non-iterative, baseline approach (i.e., red bars) requires a total sample ratio of at least 0.01, an order of magnitude more samples than IterML, to reach the same standard. IterML achieves *Standard 2* by using three iterations of 0.004 (0.4%) samples for a total of 0.012 (1.2%) samples, as shown by the light blue-grey bars. We then compare the IterML results to (1) the median and (2)

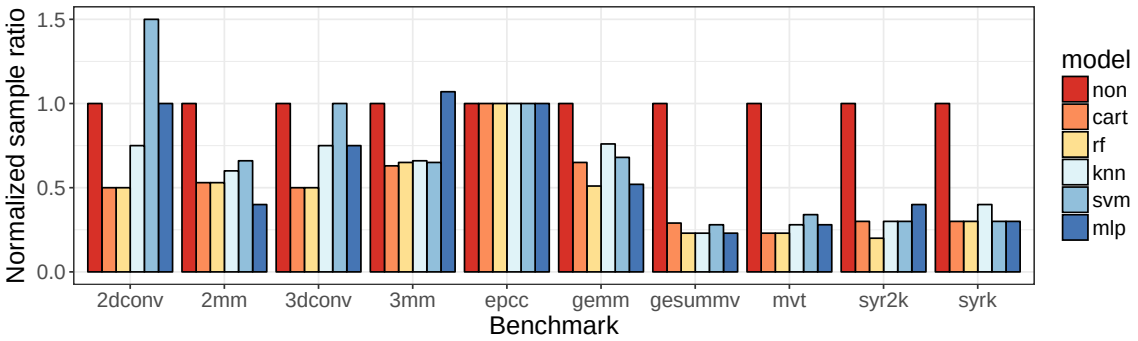

Figure 5.16: Normalized Sample Ratio to Achieve Standard 1 on the V100 GPU (Lower is Better.)

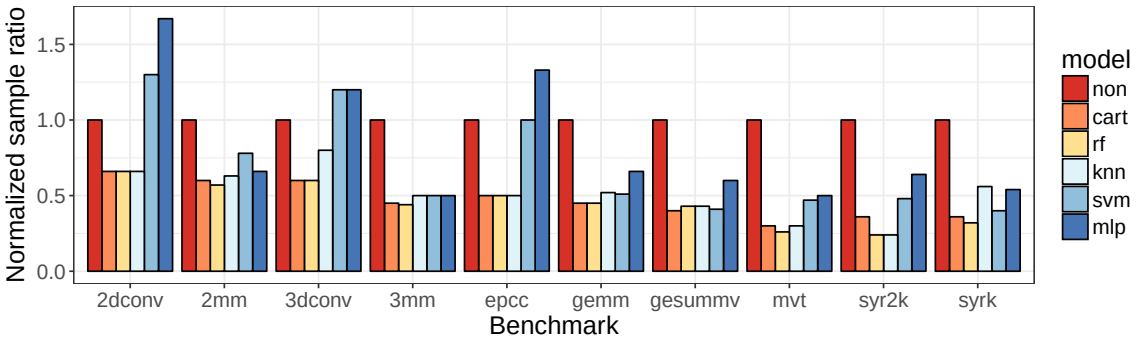

Figure 5.17: Normalized Sample Ratio to Achieve Standard 2 on the V100 GPU (Lower is Better.)

the 5th-percentile bar to 0.95 and collect the sample ratio required by each model to achieve *Standard 1* or *Standard 2*, respectively.

Figures 5.16, 5.17, 5.18, and 5.19 show the normalized total samples to reach *Standard 1* or *Standard 2* on the V100 and P100 GPUs, respectively. Depending on the performance distribution of different benchmarks, we need different sample ratios to achieve high performance. For comparison purposes, we normalize the number of the samples required by different models to the baseline (denoted as the "non" model, short for non-iterative ML model).

Figure 5.18 shows that the CART and RF models only require 0.2 (or 20%) of the samples compared to the baseline. The worst case is the 3MM benchmark, which takes approximately

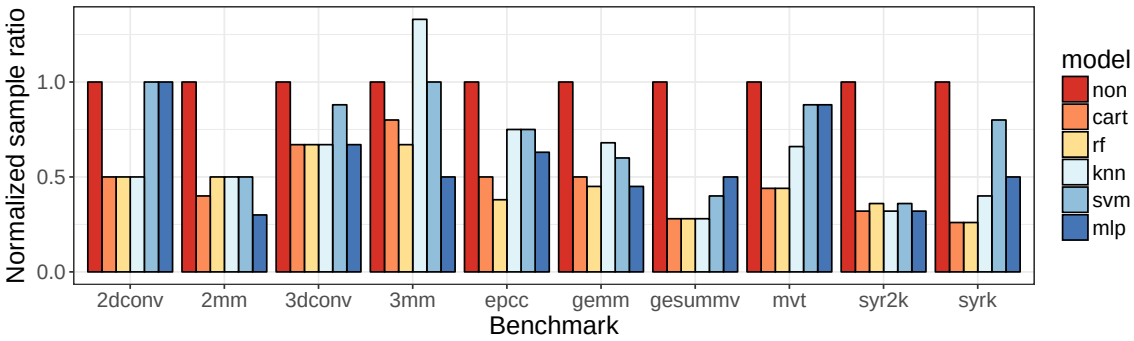

Figure 5.18: Normalized Sample Ratio to Achieve Standard 1 on the P100 GPU (Lower is Better.)

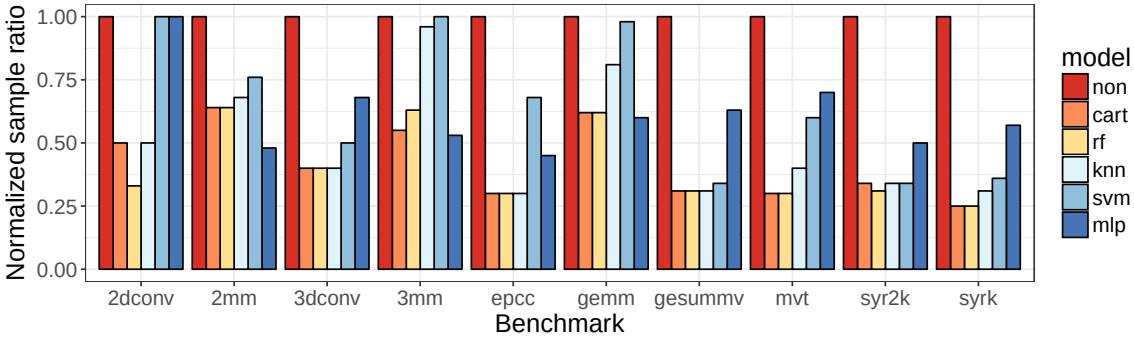

Figure 5.19: Normalized Sample Ratio to Achieve Standard 2 on the P100 GPU (Lower is Better.)

0.6 (or 60%) of the samples while using the RF model. Overall, in most cases, using our iterative machine-learning (IterML) approach saves 40% to 80% search effort when choosing the best model.

We also observe that the performance of SVM and MLP, respectively, are not stable. Sometimes these two models perform even worse than the baseline, especially when the sample size is relatively small. We conjecture that these two models require a certain amount of data to be effective.

On the other hand, the other three models (CART, RF, and KNN) always require fewer samples to achieve high performance. We note that we used a cut ratio of 50% by default in this case because the search space (i.e., GPU thread-block size) is relatively small. When

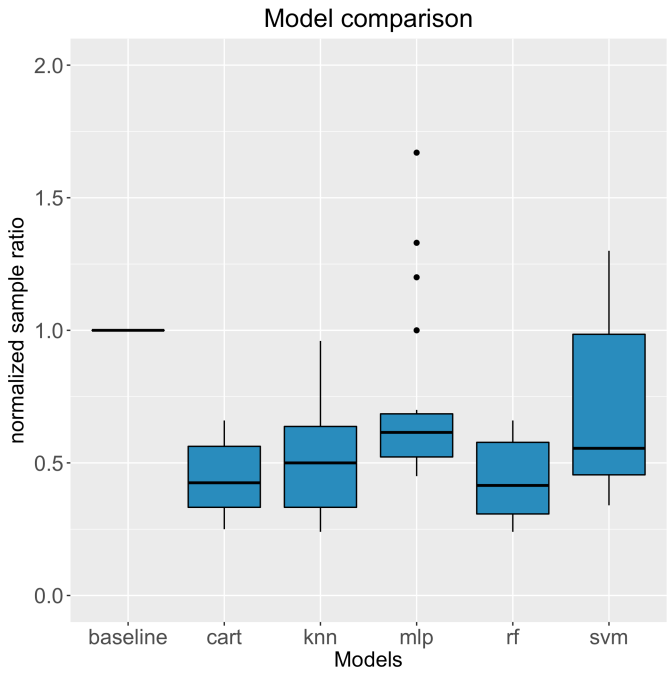

Figure 5.20: Comparison of Machine-Learning (ML) Models for IterML, Relative to the Normalized Sample Ratio (Lower is better.)

dealing with a larger search space, we may increase this value to achieve higher search speed. However, this higher search speed could compromise the search quality; hence, there is a tradeoff here, which we discuss in our future work.

Figure 5.20 shows the box plot of the overall average normalized sample required to achieve *Standard 1* and *Standard 2* for different machine-learning (ML) models. The results are normalized to the baseline, which exhaustively randomly chooses the parameter combinations as samples. For the five popular ML models, random forest (RF) performs the best and produces more stable results than all other models. Compared to the non-iterative approach (baseline), it saves $\sim$ 40% to 80% required samples to achieve good performance. Overall, it saves $\sim$60% required sample, which only requires $\sim$1.5% of the search space on average.

Previously, we categorized the benchmarks into two groups based on the performance distribution. Figure 5.21 shows their average raw sample ratio to achieve Standard 1, which means

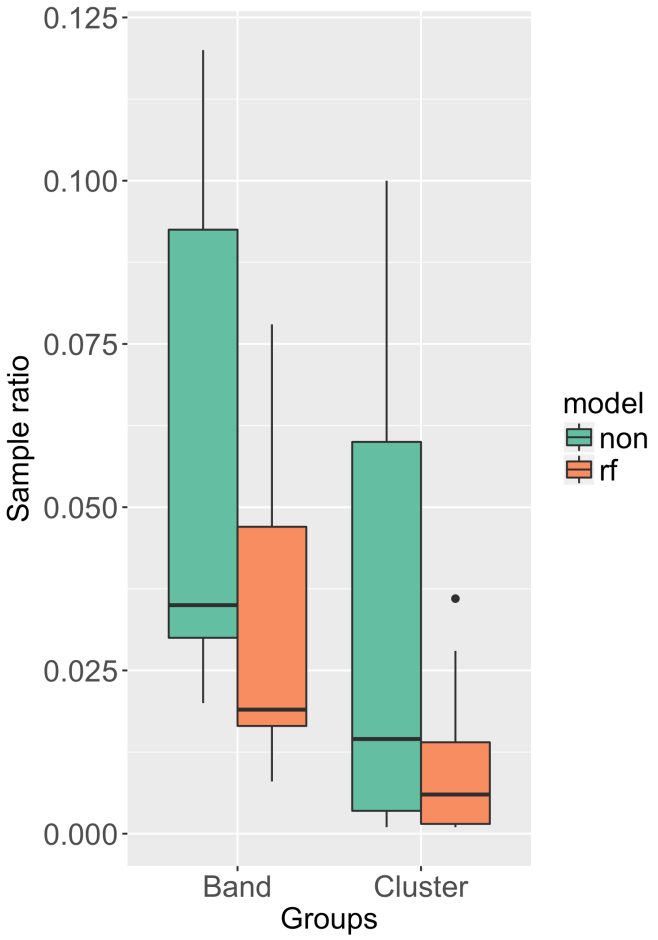

Figure 5.21: Comparison of Benchmark Performance Distribution Group, Relative to the Raw Sample Ratio to Achieve Standard 1 (Lower is Better)

the performance expectation is higher than 95% of the optimal. We only compare between the non-iterative approach and our IterML with Random Forest model. We see that "The Band" obviously requires a lot more samples to achieve good performance, which is around 3% on average. Our IterML approach can significantly reduce the sample ratio , which is 1.7% on average. For both groups, the IterML always provides more stable performance than the non-iterative approach.

In conducting this empirical study on our iterative machine learning (IterML) algorithm and comparing its performance to that of traditional non-iterative ML, we note that we used the

default model functions provided by the scikit-learn library.

5.5 Conclusion

In this work, we presented our iterative pruning approach with machine learning models (IterML) to auto-tune the performance of code running on accelerators, in particular, Nvidia GPUs. In each iteration, we used machine-learning (ML) models to assist with pruning the rest of the parameter search space. Specifically, we focused on auto-tuning the GPU thread-block size.

Overall, our experimental results showed that IterML can significantly reduce the search effort by 40% to 80% compared to the traditional non-iterative ML approach. We also showed that the random forest (RF) model, in particular, better fits our IterML design than other models like SVM or MLP.

Chapter 6

Conclusion and Future Work

6.1 Summary

Demand for extreme performance has been growing significantly in the last decades. We have seen the computation power of top supercomputers increase by tens of billion times during the last 40 years. We have also witnessed that companies are willing to develop new hardware accelerators (e.g., GPUs, co-processors, FPGAs, and TPUs) to speed up a specific category of applications. At the same time, we identify the following issues: How can we easily port existing code to new hardware platforms? How can we extract the most performance out of these new accelerators for different applications? How can we efficiently leverage the high-bandwidth memory chips on the hardware while the big data can not fit into it? How can we auto-tune the performance without much domain knowledge? This dissertation handles the above issues by introducing more directive-based extensions to the OpenMP standard in order to unify the API of accelerators (particularly GPUs) and to provide better performance, programmability, and memory management.

First, we proposed a directive-based partitioning and pipelining extension for GPUs. We noticed that developers and scientists are willing to keep their original C or Fortran code. Compared to CUDA/OpenCL, OpenMP makes the procedure much easier without large code refactoring and debugging. In light of the fact that OpenMP was originally intended for a shared-memory programming model and "target" is new to OpenMP, we proposed

associated clauses to improve pipelining efficiency. Our extension allows GPU programmers to pipeline data transfers without major refactoring, thus automating overlap of computation and communication. Further, mapping subsections of the host array to a device buffer can reduce memory requirements and increase portability.

Secondly, we noticed that the previous pipelining implementation still has multiple potential to be improved. First, the tasks are partitioned into multiple chunks, and these chunks are bound to different GPU streams. The memory copy API calls and kernel launch overhead have been magnified during this process. Secondly, we observed that the hyper-parameters (e.g., block size, chunk size, and #streams) still significantly affect performance, which makes tuning for optimal performance difficult. Finally, the CPU has to be involved during these API launches. As we see the heterogeneity is increasing tremendously and we will see more and more accelerators. This process will significantly increase the burden of the CPU. Moreover, task order logic controls are handled by the CPU, which may also delay the whole process because the information has to go through the network connection between CPU and the accelerators. To solve this issue, we present our block-level data pipelining approach for GPUs. The GPU handles all tasks, including data transfers, computation, and tracking of task dependencies, while the CPU only needs to launch one kernel. We leverage multiple techniques to optimize block-level data pipelining.

At the end, we found that the work focused on more enhanced programmability, performance, and memory management. However, after successfully porting the code to accelerators, performance hand-tuning is an important stage towards achieving high performance in specific applications, which always creates a hard task for developers without domain knowledge about accelerators. This dissertation then focused more on performance auto-tuning. We presented an iterative pruning approach with machine learning models (IterML) to auto-tune the performance of code running on accelerators, in particular, Nvidia GPUs. We noticed

that most AI tasks use the iterative approach and the computation pattern actually does not change during each iteration, only the data does. We then came up with IterML to auto-tune the parameters as it runs iteratively, using as little samples as possible to achieve relatively good performance.

6.2 Future Work

The dissertation focuses on proposing new directive-based extensions for accelerators to provide better performance, programmability, and memory management. We have several open research and engineering questions for future work.

6.2.1 Source-to-Source Translator

Current work focuses on the design, experiments, and evaluation. We proposed the extension syntax, implemented the framework, and evaluated it with multiple benchmarks. The translation from traditional code with the extension to the real accelerated code still requires extra source-to-source translator. This translator requires a lot of engineering work and depends on the compiler and hardware. How we might accurately and efficiently translate the code is still an open question for both research and engineering. A powerful code analysis engine capable of deep analysis of code and dependencies could significantly simplify our proposed extension. Potentially the compiler could determine the array definition information and even the data dependencies. How to track the dependencies effectively and partition the loops automatically is also a good research direction, which may significantly simplify the extension and improve programmability.

6.2.2 Inter-Node Pipelining

In our current work, the focus is on a single node with a single GPU or a single node with multiple GPUs. In real-world applications or tasks, multiple nodes with Message Passing Interface (MPI) are still required to achieve high performance. Most computational jobs can not be partitioned to pure independent groups, which means communication between nodes is required. How to overlap the communication with the computation efficiently is still an open question in this area. Traditionally, MPI is the most popular way to handle communication. However, the CPU has to be part of the process. While "GPU Direct" [9] or RDMA [81] improves this issue, the CPU still needs to be involved for a lot of API calls and logic control. Moreover, developers without domain knowledge have challenging time to program a multi-node accelerator code. As more and more new members are contributing the development of accelerators, people really need an effective and easy-to-use abstraction or unified API to program on these accelerators without much background knowledge.

Bibliography

[1] A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W. Feng, K. R. Bisset, and R. Thakur. MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-Based Systems. In *14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems*, pages 647–654, 2012.

[2] A. M. Aji, L. S. Panwar, F. Ji, K. Murthy, M. Chabbi, P. Balaji, K. R. Bisset, J. Dinan, W. Feng, J. Mellor-Crummey, X. Ma, and R. Thakur. MPI-ACC: Accelerator-Aware MPI for Scientific Applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1401–1414, 2016.

[3] A. M. Aji, A. J. Peña, P. Balaji, and W. Feng. Automatic Command Queue Scheduling for Task-Parallel Workloads in OpenCL. In *IEEE International Conference on Cluster Computing*, pages 42–51, 2015.

[4] Donald G Albert, Lanbo Liu, and Mark L Moran. Time Reversal Processing for Source Location in an Urban Environment. *The Journal of the Acoustical Society of America*, 115(2):2596–619, 2005.

[5] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. StarPU: A Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. 2010.

[6] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.

[7] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011.

[8] Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Laxmi N Bhuyan. Juggler: A Dependence-Aware Task-Based Execution Framework for GPUs. In *23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–67, 2018.

[9] Ray Bittner, Erik Ruf, and Alessandro Forin. Direct GPU/FPGA Communication via PCI Express. *Cluster Computing*, 17(2):339–348, 2014.

[10] C. Bo, V. Dang, E. Sadredini, and K. Skadron. Searching for Potential gRNA Off-Target Sites for CRISPR/Cas9 Using Automata Processing Across Different Platforms. In *24th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 737–748, 2018.

[11] Chunkun Bo, Vinh Dang, Ted Xie, Jack Wadden, Mircea Stan, and Kevin Skadron. Automata Processing in Reconfigurable Architectures: In-the-Cloud Deployment, Cross-Platform Evaluation, and Fast Symbol-Only Reconfiguration. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(2):1–25, 2019.

[12] S. Boll. Suppression of Acoustic Noise in Speech using Spectral Subtraction. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 27(2):113–120, 1979.

[13] Ian Buck. Nvidia's Next-Gen Pascal GPU Architecture to Provide 10x Speedup for Deep Learning App, 2015.

[14] Javier Bueno, Xavier Martorell, Rosa M Badia, Eduard Ayguadé, and Jesús Labarta. Implementing OmpSs Support for Regions of Data in Architectures with Multiple Ad-

dress Spaces. In *27th International ACM Conference on International Conference on Supercomputing*, pages 359–368, 2013.

[15] B L Chamberlain, D Callahan, and H P Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[16] Guoyang Chen and Xipeng Shen. Free Launch: Optimizing GPU Dynamic Kernel Launches Through Thread Reuse. In *48th International Symposium on Microarchitecture*, pages 407–419, 2015.

[17] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759*, 2014.

[18] Jee W Choi, Amik Singh, and Richard W Vuduc. Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. *ACM SIGPLAN Notices*, 45(5):115–126, 2010.

[19] J. Choquette, O. Giroux, and D. Foley. Volta: Performance and Programmability. *IEEE Micro*, 38(2):42–52, 2018.

[20] X. Cui, T. R. W. Scogland, B. R. d. Supinski, and W. Feng. Directive-Based Pipelining Extension for OpenMP. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 481–484, 2016.

[21] X. Cui, T. R. W. Scogland, B. R. De Supinski, and W. Feng. Directive-Based Partitioning and Pipelining for Graphics Processing Units. In *31st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 575–584, 2017.

[22] Xuewen Cui and Wu-chun Feng. Iterative Machine Learning (IterML) for Effective

Parameter Pruning and Tuning in Accelerators. In *ACM International Conference on Computing Frontiers*, pages 16–23, 2019.

[23] Xuewen Cui and Wu-chun Feng. IterML: Iterative Machine Learning for Intelligent Parameter Pruning and Tuning in Graphics Processing Units. *Journal of Signal Processing Systems*, pages 1–13, 2020.

[24] B. R. de Supinski, T. R. W. Scogland, A. Duran, M. Klemm, S. M. Bellido, S. L. Olivier, C. Terboven, and T. G. Mattson. The Ongoing Evolution of OpenMP. *Proceedings of the IEEE*, 106(11):2004–2019, 2018.

[25] Jack J Dongarra, Hans W Meuer, and Erich Strohmaier. Top500 Supercomputer Sites, 1994.

[26] P. Eller, J. C. Cheng, and D. Albert. Acceleration of 2-D Finite Difference Time Domain Acoustic Wave Simulation Using GPUs. In *DoD High Performance Computing Modernization Program Users Group Conference*, pages 350–356, 2010.

[27] Isaac Gelado, John E Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wenmei W Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–358, 2010.

[28] Rafael C Gonzalez and Richard E Woods. *Digital Image Processing*. Prentice Hall Upper Saddle River, 2002.

[29] K. Gupta, J. A. Stuart, and J. D. Owens. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In *Innovative Parallel Computing (InPar)*, pages 1–14, 2012.

[30] Mark Harris. Unified Memory in CUDA 6. *GTC On-Demand, NVIDIA*, 2013.

[31] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. *ACM SIGPLAN Notices*, 46(8):267–276, 2011.

[32] Sunpyo Hong and Hyesoon Kim. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. In *36th Annual International Symposium on Computer Architecture*, pages 152–163, 2009.

[33] K. Hou, W. Feng, and S. Che. Auto-Tuning Strategies for Parallelizing Sparse Matrix-Vector (SpMV) Multiplication on Multi- and Many-Core Processors. In *31st IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 713–722, 2017.

[34] K. Hou, H. Wang, and W. Feng. AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-Based Multi-and Many-Core Processors. In *30th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 780–789, 2016.

[35] K. Hou, H. Wang, W. Feng, J. S. Vetter, and S. Lee. Highly Efficient Compensation-Based Parallelism for Wavefront Loops on GPUs. In *32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 276–285, 2018.

[36] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. Fast Segmented Sort on GPUs. In *International Conference on Supercomputing*, pages 1–10, 2017.

[37] Kaixi Hou, Hao Wang, and Wu-chun Feng. GPU-Unicache: Automatic Code Generation of Spatial Blocking for Stencils on GPUs. In *ACM International Conference on Computing Frontiers*, pages 107–116, 2017.

[38] Thomas B Jablin, Prakash Prabhu, James A Jablin, Nick P Johnson, Stephen R Beard, and David I August. Automatic CPU-GPU Communication Management and Opti-

mization. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, 2011.

[39] J. Jenkins, J. Dinan, P. Balaji, T. Peterka, N. F. Samatova, and R. Thakur. Processing MPI Derived Datatypes on Noncontiguous GPU-Resident Data. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2627–2637, 2014.

[40] F. Ji, H. Lin, and X. Ma. RSVM: A Region-Based Software Virtual Memory for GPU. In *22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 269–278, 2013.

[41] N Johnson. EPCC OpenACC Benchmark Suite, 2013.

[42] P. J. Joseph, Kapil Vaswani, and M. J. Thazhuthaveetil. Construction and Use of Linear Regression Models for Processor Performance Analysis. In *12th International Symposium on High-Performance Computer Architecture, 2006.*, pages 99–108, 2006.

[43] B. Klenk, H. Fröening, H. Eberle, and L. Dennison. Relaxations for High-Performance Message Passing on Massively Parallel SIMT Processors. In *31st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 855–865, 2017.

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[45] K. Krommydas, T. R. W. Scogland, and W. Feng. On the Programmability and Performance of Heterogeneous Platforms. In *International Conference on Parallel and Distributed Systems*, pages 224–231, 2013.

[46] Samuli Laine, Tero Karras, and Timo Aila. Megakernels Considered Harmful: Wave-

front Path Tracing on GPUs. In *5th High-Performance Graphics Conference*, pages 137–143, 2013.

[47] R. Landaverde, Tiansheng Zhang, A. K. Coskun, and M. Herbordt. An Investigation of Unified Memory Access Performance in CUDA. In *18th IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2014.

[48] R. Lee, H. Wang, and X. Zhang. Software-Defined Software: A Perspective of Machine Learning-Based Software Production. In *38th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 1270–1275, 2018.

[49] W. Li, G. Jin, X. Cui, and S. See. An Evaluation of Unified Memory Technology on NVIDIA GPUs. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1092–1098, 2015.

[50] Yan Li, Kenneth Chang, Oceane Bel, Ethan L Miller, and Darrell DE Long. CAPES: Unsupervised Storage Performance Tuning Using Neural Network-Based Deep Reinforcement Learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[51] Yinan Li, Jack Dongarra, and Stanimire Tomov. A Note on Auto-Tuning GEMM for GPUs. In *International Conference on Computational Science*, pages 884–892. Springer, 2009.

[52] Chunhua Liao, Yonghong Yan, Bronis R de Supinski, Daniel J Quinlan, and Barbara Chapman. Early Experiences with the OpenMP Accelerator Model. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 84–98. Springer, 2013.

[53] Yuan Lin and David Padua. Compiler Analysis of Irregular Memory Accesses. *ACM SIGPLAN Notices*, 35(5):157–168, 2000.

[54] Sparsh Mittal and Jeffrey S Vetter. A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. *ACM Computing Surveys (CSUR)*, 47(2):19, 2015.

[55] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. Unified Memory in CUDA 6: A Brief Overview and Related Data Access/Transfer Issues. 2014.

[56] NVIDIA. CUDA C/C++ Basics Supercomputing 2011 Tutorial. *Supercomputing 2011*, 27.

[57] OpenMP ARB. OpenMP Application Program Interface Version 4.0, 2013.

[58] OpenMP ARB. OpenMP Application Program Interface Version 4.5, 2015.

[59] OpenMP ARB. OpenMP Application Program Interface Version 5.0, 2018.

[60] Manolis Papadrakakis, George Stavroulakis, and Alexander Karatarakis. A New Era in Scientific Computing: Domain Decomposition Methods in Hybrid CPU–GPU Architectures. *Computer Methods in Applied Mechanics and Engineering*, 200(13-16):1490–1508, 2011.

[61] PCI-SIG. PCI-E RoadMap, 2017.

[62] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[63] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda. Efficient Inter-Node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *42nd International Conference on Parallel Processing*, pages 80–89, 2013.

[64] Louis-Noël Pouchet. Polybench: The Polyhedral Benchmark Suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[65] Luis-Noel Pouchet et al. Polybenchmarks Benchmark Suite, 2013.

[66] S. Pumma, M. Si, W. Feng, and P. Balaji. Parallel I/O Optimizations for Scalable Deep Learning. In *23rd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 720–729, 2017.

[67] S. Pumma, M. Si, W. Feng, and P. Balaji. Towards Scalable Deep Learning via I/O Analysis and Optimization. In *19th IEEE International Conference on High Performance Computing and Communications & 15th IEEE International Conference on Smart City & 3rd IEEE International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 223–230, 2017.

[68] Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. Scalable Deep Learning via I/O Analysis and Optimization. *ACM Transactions on Parallel Computing (TOPC)*, 6(2):1–34, 2019.

[69] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *23rd ACM Symposium on Operating Systems Principles*, pages 233–248, 2011.

[70] Richard M Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, 1978.

[71] Shane Ryoo, Christopher I Rodrigues, Sam S Stone, Sara S Baghsorkhi, Sain-Zee Ueng, John A Stratton, and Wen-mei W Hwu. Program Optimization Space Pruning for a Multithreaded GPU. In *6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 195–204, 2008.

[72] T. R. W. Scogland, B. Rountree, W. Feng, and B. R. de Supinski. Heterogeneous Task Scheduling for Accelerated OpenMP. In *26th IEEE International Parallel and Distributed Processing Symposium*, pages 144–155, 2012.

[73] Thomas RW Scogland, Wu-chun Feng, Barry Rountree, and Bronis R de Supinski. CoreTSAR: Core Task-Size Adapting Runtime. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):2970–2983, 2015.

[74] Thomas RW Scogland, Jeff Keasler, John Gyllenhaal, Rich Hornung, Bronis R de Supinski, and Hal Finkel. Supporting Indirect Data Mapping in OpenMP. In *OpenMP: Heterogenous Execution and Data Movements*, pages 260–272. Springer, 2015.

[75] Isaiah Shavitt and Rodney J Bartlett. *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory*. Cambridge University Press, 2009.

[76] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. Whippletree: Task-Based Scheduling of Dynamic Workloads on the GPU. *ACM Transactions on Graphics (TOG)*, 33(6):228, 2014.

[77] John E Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[78] Nhat-Phuong Tran, Myungho Lee, and Jaeyoung Choi. Parameter Based Tuning Model for Optimizing Performance on GPU. *Cluster Computing*, 20(3):2133–2142, 2017.

[79] J. Wadden, V. Dang, N. Brunelle, T. T. II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron. ANMLzoo: A Benchmark Suite for Exploring Bottlenecks in Automata Processing Engines and Architectures. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2016.

[80] J. Wadden, T. Tracy, E. Sadredini, L. Wu, C. Bo, J. Du, Y. Wei, J. Udall, M. Wallace, M. Stan, and K. Skadron. AutomataZoo: A Modern Automata Processing Benchmark Suite. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 13–24, 2018.

[81] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda. GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2595–2605, 2014.

[82] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda. Optimized Non-Contiguous MPI Datatype Communication for GPU Clusters: Design, Implementation and Evaluation with MVAPICH2. In *IEEE International Conference on Cluster Computing*, pages 308–316, 2011.

[83] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K Panda. MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters. *Computer Science-Research and Development*, 26(3-4):257–266, 2011.

[84] Yichao Wang, Qiang Qin, Simon Chong Wee SEE, and James Lin. Performance Portability Evaluation for OpenACC on Intel Knights Corner and Nvidia Kepler. *HPC China*, 2013.

[85] Jack Wells, Buddy Bland, Jeff Nichols, Jim Hack, Fernanda Foertter, Gaute Hagen, Thomas Maier, Moetasim Ashfaq, Bronson Messer, and Suzanne Parete-Koon. Announcing Supercomputer Summit, 2016.

[86] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC–First Experiences with Real-World Applications. In *Euro-Par Parallel Processing*, pages 859–870. Springer, 2012.

[87] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations. In *29th ACM International Conference on Supercomputing*, pages 119–130, 2015.

[88] Wei Wu, George Bosilca, Rolf Vandevaart, Sylvain Jeaugey, and Jack Dongarra. GPU-Aware Non-Contiguous Data Movement in OpenMPI. In *25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 231–242, 2016.

[89] LC Yan, B Yoshua, and H Geoffrey. Deep Learning. *Nature*, 521(7553):436–444, 2015.

[90] X. Yu, H. Wang, W. Feng, H. Gong, and G. Cao. cuART: Fine-Grained Algebraic Reconstruction Technique for Computed Tomography Images on GPUs. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 165–168, 2016.

[91] Xiaodong Yu and Michela Becchi. Exploring Different Automata Representations for Efficient Regular Expression Matching on GPUs. *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[92] Xiaodong Yu and Michela Becchi. GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space. In *ACM International Conference on Computing Frontiers*, pages 1–10, 2013.

[93] Xiaodong Yu, Hao Wang, Wu-chun Feng, Hao Gong, and Guohua Cao. An Enhanced Image Reconstruction Tool for Computed Tomography on GPUs. In *ACM International Conference on Computing Frontiers*, pages 97–106, 2017.

[94] Xiaodong Yu, Hao Wang, Wu-chun Feng, Hao Gong, and Guohua Cao. GPU-Based

Iterative Medical CT Image Reconstructions. *Journal of Signal Processing Systems*, 91(3-4):321–338, 2019.

[95] Xiaodong Yu, Fengguo Wei, Xinming Ou, Michela Becchi, Tekin Bicer, and Danfeng (Daphne) Yao. GPU-Based Static Data-Flow Analysis for Fast and Scalable Android App Vetting. In *34th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020.

[96] Xiaodong Yu, Ya Xiao, Kirk Cameron, and Danfeng (Daphne) Yao. Comparative Measurement of Cache Configurations' Impacts on Cache Timing Side-Channel Attacks. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, Santa Clara, CA, August 2019. USENIX Association.

[97] Y. Yuan, M. F. Salmi, Y. Huai, K. Wang, R. Lee, and X. Zhang. Spark-GPU: An Accelerated In-Memory Data Processing Engine on Clusters. In *IEEE International Conference on Big Data*, pages 273–283, 2016.

[98] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. VersaPipe: A Versatile Programming Framework for Pipelined Computing on GPU. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 587–599, 2017.