Dynamic Orchestration of Massively Data Parallel Execution

by

Mehrzad Samadiarakhshbahar

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2014

Doctoral Committee:

Professor Scott Mahlke, Chair
Professor Robert Dick
Professor Trevor N. Mudge
Associate Professor Thomas F. Wenisch

To my family

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my adviser Prof. Scott Mahlke for his guidance, enthusiastic encouragement and useful critique of this research work. I consider myself truly lucky to have worked with him these past years. He has shown incredible patience, served as an excellent mentor, and provided me every opportunity to succeed in this field. I also owe thanks to the remaining members of my dissertation committee, Prof. Mudge, Prof. Dick and Prof. Wenisch. They all devoted their time to help shape this research into what it has become today.

I was lucky to be part of a research group whose members not only assisted me intellectually in my research but were also a comfort during those long nights before each deadline. Amir spent many hours to help me with this work. I can not imagine having this thesis in its current form without his support. Mojtaba provided signicant help with the Chapter III in this dissertation. Janghaeng also contributed signicantly, helping me with his amazing ideas. Anoushe Jamshidi did a great deal of work on the part of this thesis presented in Chapters V and VI. I want to thank Armin, Gaurav, Ankit, HK, Daya, Andrew and Shruti for all the discussions that we had and proofreading my papers.

I would like to thank all my fellow labmates in the CCCP research group for their social support: Amir, Mojtaba, Ganesh, Shuguang, Shantanu, Po-Chun, Yongjun, Hyoun

Kyu, Gaurav, Ankit, Anoushe, Daya, Janghaeng, Andrew, Shruti, Jason, John, and Silky. You folks made coming to work a lot more fun. My special thanks are extended to the staff in the fourth floor for puting up with our annoying traffic to the kitchen.

My time in Ann Arbor was made enjoyable mostly due to my large Iranian extended family of friends: Alireza, Amir, Mona, Mojtaba, Mehdi, Marjan, Hesam, Azadeh, Armin, Elnaz, Sara, Mohammad, Nasibeh, Mahdi, Narges, Roghayeh, Kaveh, Hadi, Shima, Fardokht, Gelareh, Morteza, Samira, Gelareh, Meysam, Ali, Mahta, Pedram, Saeed, Tayebeh, Sina, Mahya, Mohammadreza, Parisa, Yaser, Mozdeh, Mahmood, Azadeh, Maryam, Hamidreza, and many others. I couldn't have made it through without you people. I'll never forget the fun times we spent together. I would like to express my special thanks to all members of Washtenaw Toastmaster club for their encouragement and their support. I would like to thank creators, actors, and all cast members of different TV series such as the Office Show, Breaking Bad, Dexter, Walking Dead. Coding without watching these shows on the second monitor can be really boring.

Finally and most importantly, my family deserves major gratitude for everything they taught me. Whatever I am today, I owe it to my mother, Hermin. She taught me how to stay strong and handle challenges in my life. My father, Behrooz, who raised me with the love of science. My brother and my sister , Behzad and Samareh, taught me how to love what you do and do what you love. I also want to thank my in-laws, Farhad and Roshanak. They always supported my career and I really appreciate it. I would like to thank my niece, Helia, for bringing joy to our family. And above all, I really appreciate the love and support of my wife, Parisa. She taught me how to be hopeful and have fun even in the most difficult situations.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Dynamic Orchestration of Massively Data Parallel Execution

by

Mehrzad Samadi

Advisor: Scott Mahlke

Graphics processing units (GPUs) are specialized hardware accelerators capable of rendering graphics much faster than conventional general-purpose processors. They are widely used in personal computers, tablets, mobile phones, and game consoles. Modern GPUs are not only efficient at manipulating computer graphics, but also are more effective than CPUs for algorithms where processing of large data blocks can be done in parallel. This is mainly due to their highly parallel architecture.

While GPUs provide low-cost and efficient platforms for accelerating massively parallel applications, tedious performance tuning is required to maximize application execution efficiency. Achieving high performance requires the programmers to manually manage the amount of on-chip memory used per thread, the total number of threads per multiprocessor, the pattern of off-chip memory accesses, etc.

In addition to a complex programming model, there is a lack of performance portabil-

ity across various systems with different runtime properties. Programmers usually make assumptions about runtime properties when they write code and optimize that code based on those assumptions. However, if any of these properties changes during execution, the optimized code performs poorly. To alleviate these limitations, several implementations of the application are needed to maximize performance for different runtime properties. However, it is not practical for the programmer to write several different versions of the same code which are optimized for each individual runtime condition.

In this thesis, we propose a static and dynamic compiler framework to take the burden of fine tuning different implementations of the same code off the programmer. This framework enables the programmer to write the program once and allow a static compiler to generate different versions of a data parallel application with several tuning parameters. The runtime system selects the best version and fine tunes its parameters based on runtime properties such as device configuration, input size, dependency, and data values.

CHAPTER I

Introduction

Heterogeneous systems that combine traditional processors with powerful GPUs have become standard in most systems ranging from servers to cell phones. GPUs achieve their high performance and energy efficiency by providing a massively parallel architecture with hundreds of in-order cores while exposing parallelism and the memory hierarchy to the programmer. Different speedups from 2.5x [57] to 100x [71] have been achieved on the GPU architecture compared to the CPUs.

While GPUs provide an inexpensive and highly parallel system for accelerating massively parallel workloads, efficiently utilizing GPU resources is challenging mostly due to the *programming complexity* posed to application developers. Graphics chip manufacturers, such as NVIDIA and AMD, have tried to alleviate the complexity problem by introducing user-friendly programming models, such as CUDA [72] and OpenCL [46]. Although such programming models abstract away the underlying GPU architecture by providing a unified processor model, achieving high performance still requires the programmers to manually manage the amount of on-chip memory used per thread, the total number of threads per multiprocessor, and the pattern of off-chip memory accesses [89]. Often the

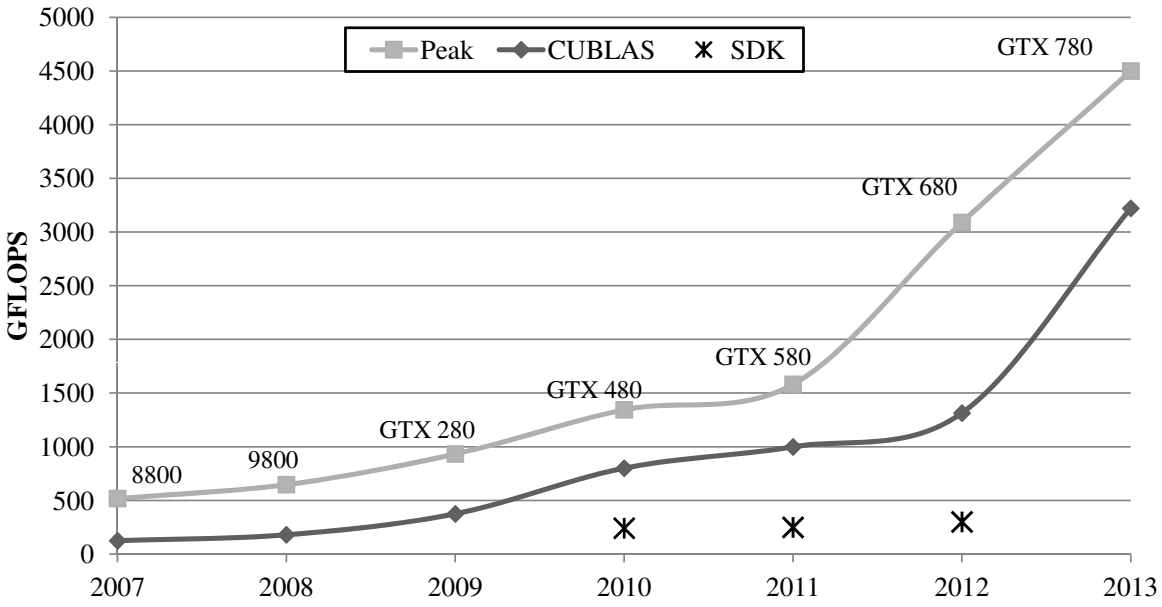


Figure 1.1: Comparison of peak and achieved performance for matrix multiplication on different GPUs

programmer must perform a tedious cycle of performance tuning to extract the desired performance. Figure 1.1 shows the theoretical peak performance , the performance achieved by highly optimized matrix multiplication from CUBLAS library and, the performance of matrix multiplication benchmark from NVIDIA SDK for different generations of NVIDIA GPUs. As shown in the figure, even the highly optimized code (CUBLAS) cannot efficiently utilize GPU resources and gain near peak performance. This gap is considerably larger for a moderately optimized code such as matrix multiplication from NVIDIA SDK.

In addition to complex programming model, a lack of performance portability across various systems with different runtime properties is another major challenge. Programmers usually make assumptions about runtime properties when they write a code and optimize it based on those assumptions. However, if any of these properties changes during execution, the optimized code performs poorly. We will explain how these runtime properties such

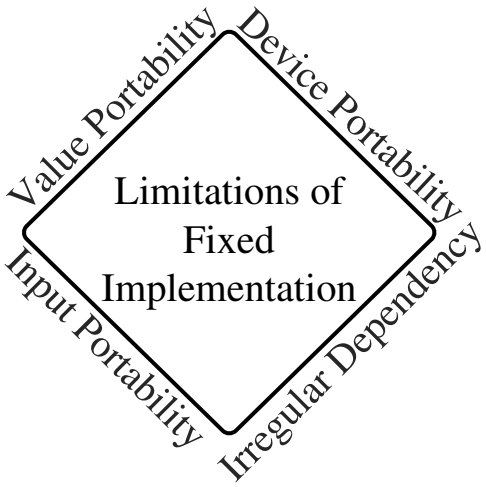


Figure 1.2: Limitations of fixed implementation code

as underlying architecture, input size and dimensions, data dependencies between threads, and data values (Figure 1.2) impact the performance of *fixed implementation code* in the following paragraphs.

Device Portability: Different GPUs vary in several key micro-architectural parameters such as number of registers, maximum number of active threads, and the size of global memory. These parameters will vary even more when newer high performance cards, such as NVIDIA's Kepler [73], and future resource-constrained mobile GPUs with less resources are released. This heterogeneity in hardware leads to a different set of optimization choices for each GPU. As a result, optimization decisions for one generation of GPUs are likely to be poor choices for another generation. We call this problem *device portability*.

Input portability: The portability issue is not specific to executing applications on different GPU targets. Even for a fixed GPU target, changing the problem size and dimensions can make an implementation of an algorithm sub-optimal, resulting in poor performance

portability. We refer to this problem as *input portability*. The main cause of this effect is that the workload of each thread in the application is set based on the input size. Therefore, a fixed implementation works well for a certain range of these values but for other input dimensions, either there would not be enough threads to run in parallel and hide memory latency , or the data chunk that each block is operating on would be too small to amortize the overhead of parallel execution.

Irregular Dependency: Irregular dependencies in data parallel codes are another limitation which prevents the fixed implementation code from performing efficiently. Common parallelization techniques cannot parallelize the applications that contain irregular dependencies that manifest infrequently, or statically-unresolvable dependencies that may not manifest during runtime at all. Therefore, ambiguous memory dependencies or control flow divergences in a small number of threads can negatively affect thousands of other threads on a GPU. The compiler analyses used for automatic parallelization are usually too conservative and fragile, resulting in small or no performance gains on commodity computer systems. One way to solve this problem is to use speculation.

Value Portability: Finally, data values also can have a great impact on the overall performance of a fixed implementation application. We refer to this problem as *value portability*. For example, performance of atomic operations is highly correlated with the addresses that those operations modify, which in turn depend on the input values. An atomic instruction performs a read-modify-write atomic operation on one element residing in global or shared memory. As the GPU serializes accesses to the same element, performance of atomic in-

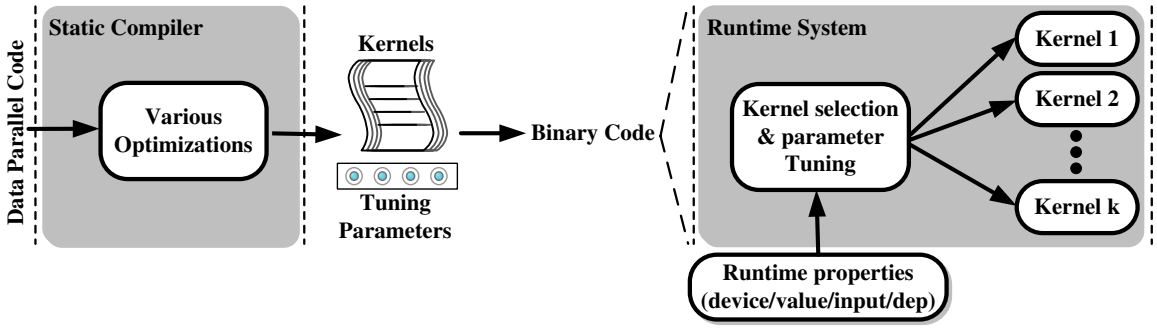


Figure 1.3: Dynamic framework for data parallel execution

structions is inversely proportional to the number of threads per warp that access the same address. If we prevent atomic operations to access the same address, overall performance will be improved.

To alleviate these limitations, several implementations of the application are needed to maximize performance under different runtime properties. However, it is not practical for the programmer to write different versions of the same code and optimize them separately. Furthermore, as most of these runtime properties are not predictable statically, a dynamic solution is necessary to choose the best implementation to maximize the performance during runtime. In this thesis, we propose a static/dynamic compiler framework to take the burden of fine tuning different implementations of the same code off the programmer as shown in Figure 1.3. The static compiler generates different versions of the data parallel application with several tuning parameters. The runtime system selects the best version and fine tunes its parameters based on runtime properties such as device configuration, input size, dependency, and data values. The remainder of this chapter describes different frameworks that are specifically designed for each of these runtime properties explained above.

1.1 Sponge

To overcome the *device portability* problem, we propose *Sponge* [44], a streaming compiler for the StreamIt language that is capable of automatically producing customized CUDA code for a wide range of GPUs. Sponge consists of stream graph optimizations which optimizes the organization of the computation graph and an efficient CUDA code generator to express the parallelism for the target GPU. Producing efficient CUDA code is a multi-variable optimization problem and can be difficult for software programmers due to the unconventional organization and the interaction of computing resources of GPUs. Sponge is equipped with a set of optimizations to handle the memory hierarchy of GPUs and also to efficiently utilize the processing units.

1.2 Adaptic

In order to mitigate the *input portability* problem, we propose an adaptive input-aware compilation system, called *Adaptic* [93], which is capable of automatically generating optimized CUDA code for a wide range of input sizes and dimensions from a high-level algorithm description. Adaptic decomposes the problem space based on the input size into discrete scenarios and creates a customized implementation for each scenario. Decomposition and customization of the problem space are accomplished through a suite of optimizations. These include a set of memory optimizations which coalesce memory access patterns employed by the high-level streaming model and efficiently execute algorithms that access several neighboring memory locations at the same time. Adaptic uses an additional group of optimizations to effectively break up the work in large program segments for efficient

execution across many threads and blocks. The final optimizations combine the work of different functions to reduce the memory overhead so that their execution overhead can be reduced.

Adaptic uses these optimizations to generate different versions of the code. At runtime, based on the provided input to the program, the best version of the generated code is selected and executed to maximize performance. This method frees application developers from the tedious task of fine-tuning and possibly changing the algorithm for each input range as shown in Figure 1.3.

1.3 Paragon

To overcome *dependency limitation*, we propose cooperative speculative loop execution on GPUs and CPUs using *Paragon* [92, 91] for implicitly data-parallel programs written in C/C++. Paragon, using data-parallel speculation and distributed conflict detection engines designed, enables programmers to transparently take advantage of GPUs for pieces of their applications that are possibly-data-parallel. The programmers does not need to manually change the application or rely on complex compiler analyses, thus reducing the cost of porting to GPUs. Further, the set of applications that can be mapped onto a GPU is broadened beyond loops that exclusively use arrays with affine indices. Paragon's use of cooperative execution between the GPU and CPU increases the performance of the overall system in the presence of conflicts since the CPU is not left idle while the GPU is speculatively running an application.

The static phase of Paragon mainly performs loop classification and generates CUDA

code for the runtime system which monitors the loops on the GPU for dependency violations. The runtime phase also performs light-weight one-time loop monitoring and decides which loops are more likely to benefit from executing on the GPU. Therefore, for each loop, runtime system decides to run it on GPU, CPU or both.

1.4 Sage

In order to maximize the performance for different *data values*, we use approximate computing to simplify or skip processing on the computationally expensive input data. We propose *Sage [95]* a framework for performing systematic runtime approximation on GPUs that enables the programmer to implement a program once in CUDA, and depending on the *target output quality (TOQ)*, trade accuracy for performance based on the evaluation metric provided by the user. SAGE has two phases: offline compilation and runtime kernel management. During offline compilation, SAGE performs approximation optimizations on each kernel to create multiple versions with varying degrees of accuracy. At runtime, SAGE uses a greedy algorithm to tune the parameters of the approximate kernels to identify configurations with high performance and a quality that satisfies the TOQ. This approach reduces the overhead of tuning as measuring the quality and performance for all possible configurations can be expensive. Since the behavior of approximate kernels may change during runtime, SAGE periodically performs a calibration to check the output quality and performance and updates the kernel configuration accordingly.

1.5 Paraprox

One of the main challenges to use approximation to provide good performance for different *data values* is generating approximate programs. Since there is no single approximation method that works for all applications, we propose a software framework called *Paraprox [94]*. Paraprox identifies common patterns found in data-parallel programs and uses a custom-designed approximation technique for each detected pattern. Paraprox is applicable to a wide range of applications as it determines the proper approximation optimizations that can be applied to each input program.

To automatically create approximate kernels, Paraprox utilizes four optimization techniques which target six data parallel patterns: Map, Scatter/Gather, Reduction, Scan, Stencil, and Partition. Paraprox applies approximate memoization to map and scatter/gather patterns where computations are replaced by memory accesses. For reduction patterns, Paraprox uses sampling plus adjustment to compute the output by only computing the reduction of a subset of the data. The stencil & partition approximation algorithm is based on the assumption that adjacent locations in an input array are typically similar in value for such patterns. Therefore, Paraprox accesses a subset of values in the input array and replicates that subset to construct an approximate version of the array. For scan patterns, Paraprox only performs the scan operation on a subset of the input array and uses the results to predict the results for the rest of the array.

CHAPTER II

Data Parallel Programming Model

The CUDA programming model is a multi-threaded SIMD model that enables implementation of general purpose programs on heterogeneous GPU/CPU systems. There are two different device types in CUDA: the host processor and the GPU. A CUDA program consists of a host code segment that contains the sequential portion of the program, which is run on the CPU, and a parallel code segment which is launched from the host onto one or more GPU devices. Recent generations of NVIDIA's GPUs, Fermi and Kepler, can support concurrent kernel execution, where different kernels of the same application context can execute on the GPU at the same time. Concurrent kernel execution allows programs that execute a number of small kernels to utilize the whole GPU. It is also possible to overlap data transfers between CPU and GPU, and kernel execution. The threading and memory abstraction of the CUDA model is shown in Figure 2.1.

The threading abstraction in CUDA consists of three levels of hierarchy. The basic block of work is a *thread*. A group of threads executing the same code are combined together to form a *thread block* or simply a *block*. Together, these thread blocks combine to form the parallel segments called *grids* where each grid is scheduled onto a GPU at a

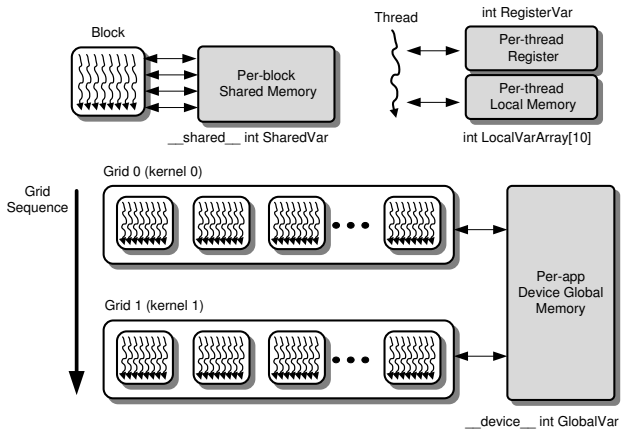


Figure 2.1: *CUDA/GPU Execution Model*

time. Threads within a thread block are synchronized together through a barrier operation ($\_\_syncthreads()$). However, there is no explicit software or hardware support for synchronization across thread blocks. Synchronization between thread blocks is performed through the global memory of the GPU, and the barriers needed for synchronization are handled by the host processor. One way to communicate between threads of different thread blocks is using atomic instructions. An atomic construct performs a read-modify-write atomic operation on one element residing in global or shared memory. For example, *atomicInc()* reads a 32-bit word from an address in the global or shared memory, increments it, and writes the result back to the same address. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete [72].

NVIDIA GPUs use a single instruction multiple thread (SIMT) model of execution where multiple thread blocks are mapped to streaming multiprocessors (SM). Each SM contains a number of processing elements called Streaming Processors (SP). A thread executes on a single SP. Threads in a block are executed in smaller execution groups of threads called *warps*. All threads in a warp share one program counter and execute the same in-

structions. If conditional branches within a warp take different paths, causing *control path divergence*, the warp will execute each branch path serially, stalling the other paths until all the paths are complete. Such control path divergences severely degrade the performance.

The memory abstraction in CUDA consists of multiple levels of hierarchy. The lowest level of memory is *registers*, which are on-chip memories private to a single thread. The next level of memory is *shared memory*, which is an on-chip memory shared only by threads within the same thread block. On devices with compute capability 2.0 and higher, there is also an L1 cache for each SM and an L2 cache shared by all SMs, both of which are used to cache accesses to local or global memory, The same on-chip memory is used for both L1 and shared memory: It can be configured by the programmer as 48 KB of shared memory and 16 KB of L1 cache or as 16 KB of shared memory and 48KB of L1 cache.

Finally, the last level of memory is *global memory*, which is an off-chip memory accessible to all threads in the grid. This memory is used primarily to stream data in and out of the GPU from the host processor. Three other memory levels exist on-chip called the *Local memory*, *texture memory* and *constant memory*. Local memory resides in the device memory and has high latency like global memory accesses. Local memory is mainly used as spill memory for local arrays and is private to a single thread. Mapping arrays to shared memory instead of spilling to local memory can provide much better performance. Texture memory is accessible through special built-in texture functions and constant memory is accessible to all threads in the grid.

Because off-chip global memory access has high latency, GPUs support *coalesced memory accesses*. Coalescing memory accesses allows one bulk memory request from multiple threads in a half-warp (full warp in Fermi and Kepler architecture) to be sent to

global memory instead of multiple separate requests. In order to coalesce memory accesses in recent generations of GPUs, all accesses of a warp should be adjacent and in the same cache line. Effective memory bandwidth is an order of magnitude lower than using non-coalesced memory accesses which further signifies the importance of memory coalescing for achieving high performance.

In modern GPUs, such as the NVIDIA GTX 560, there are 8 SMs each with 48 SPs. Each SM processes warp sizes of 32 threads. The memory sizes for this GPU are: 48K of registers and 64 KB configurable shared/L1 per SM and 1GB of global memory shared across all threads in the GPU.

CHAPTER III

Adaptive Input-aware Compilation

3.1 Introduction

GPUs are specialized hardware accelerators capable of rendering graphics much faster than conventional general-purpose processors. They are widely used in personal computers, tablets, mobile phones, and game consoles. Modern GPUs are not only efficient at manipulating computer graphics, but also are more effective than CPUs for algorithms where processing of large data blocks is done in parallel. This is mainly due to their highly parallel architecture. Recent works have shown that in optimistic cases, speedups of 100-300x [71], and in pessimistic cases, speedups of 2.5x [57], can be achieved using modern GPUs compared to the latest CPUs.

While GPUs provide inexpensive, highly parallel hardware for accelerating parallel workloads, the programming complexity remains a significant challenge for application developers. Developing programs to effectively utilize GPU's massive compute power and memory bandwidth requires a thorough understanding of the application and details of the underlying architecture. Graphics chip manufacturers, such as NVIDIA and AMD, have

tried to alleviate part of the complexity by introducing new programming models, such as CUDA [72] and OpenCL [46]. Although these models abstract the underlying GPU architecture by providing unified processing interfaces, developers still need to deal with many problems such as managing the amount of on-chip memory used per thread, total number of threads per multiprocessor, and the off-chip memory access pattern in order to maximize GPU utilization and application performance [89]. Therefore, programmers must manually perform a tedious cycle of performance tuning to achieve the desired performance.

Many prior efforts have tried to address this programmability challenge mostly along three interrelated angles. The works in [20, 44, 21, 45, 54, 55] provide high-level abstractions at the language level to enable easier expression of algorithms. These abstractions are later used by the compiler to generate efficient binaries for GPUs. Adding annotations to current models (CUDA or OpenCL) or popular languages (C or Fortran) to guide compiler optimizations is another method used in [39, 122, 117]. Finally, works in [106, 13, 121] try to automatically generate optimized code from a basic, possibly poor performing, parallel or sequential implementation of an application.

The hard problem of finding the optimal implementation of an algorithm on a single GPU target is further complicated when attempting to create software that can be run efficiently on multiple GPU architectures. For example, NVIDIA GPUs have different architectural parameters, such as number of registers and size of shared memory, that can make an implementation which is optimal for one architecture sub-optimal for another. The situation is even worse if the goal is to have an optimal implementation for GPUs across multiple vendors. We call this effect the *device portability* problem.

However, the portability issue is not specific to moving applications across different

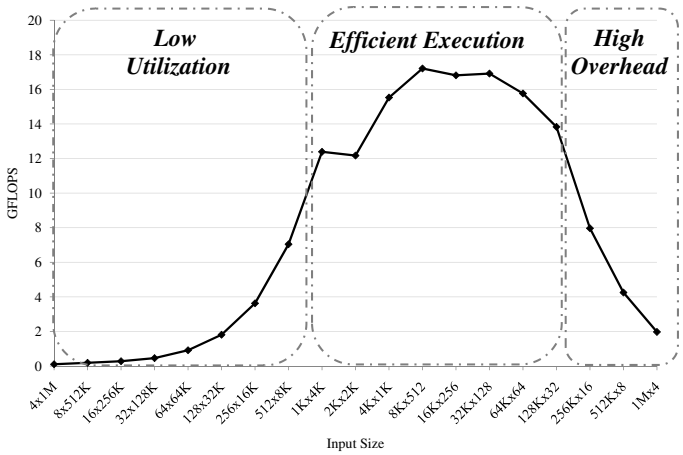


Figure 3.1: *Performance of the transposed matrix vector multiplication benchmark from the CUBLAS library on an NVIDIA Tesla C2050. The X-axis shows the input dimensions in number of rows x number of columns format.*

GPU targets. Even for a fixed GPU target, changing the problem size and dimensions can make an implementation of an algorithm sub-optimal, resulting in poor performance portability. Figure 3.1 illustrates this issue for the transposed matrix vector multiplication (TMV) benchmark from the CUBLAS library [70]. The benchmark performs consistently between 12 and 17 GFLOPs over the input dimension range of 1Kx4K to 128Kx32 on an NVIDIA Tesla C2050 GPU. However, when input dimensions fall out of this range, the performance degrades rapidly by upto a factor of more than 20x. The main reason for this effect is that the number of blocks and threads in the application are set based on the number of rows and columns in the input matrix. Therefore, this benchmark works well for a certain range of these values and for other input dimensions, either there would not be enough blocks to run in parallel and hide memory latency (towards the left end of X-axis in the figure), or the data chunk that each block is operating on would be too small to amortize the overhead of parallel block execution (towards the right end of X-axis in the figure).

In general, there are various reasons for this *input portability* problem such as unbal-

anced workload across processors, excessive number of threads, and inefficient usage of local or off-chip memory bandwidth. Unbalanced workloads occur when a kernel has a small number of blocks causing several processors to be idle during execution, which leads to under-utilization of GPU resources and poor performance. Excessive number of threads result in sequential thread execution due to lack of enough resources in the GPU to run all threads in parallel. Finally, memory access patterns in some program segments are determined based on the size or dimensions of the input. Therefore, memory optimizations must be adapted based on the input to efficiently utilize the memory bandwidth.

One solution to the input portability problem is to have the programmer design and develop different algorithms for each input range and size. However, this would impose a considerable implementation effort and verification overhead as applications become larger, more complex, and need to work across a vast range of inputs. For instance, as we show later, five kernel implementations are created to sustain high performance across the complete input spectrum in the TMV benchmark. Multi-kernel applications complicate matters as programmers must deal with the cross-product of choices for each kernel as the input is varied. Clearly, automatic tools will become essential to guarantee high performance across various input sizes.

Figure 3.2 shows a classification of prior works that have focused on improving GPU programmability, based on their support for portability across different targets (horizontal dimension) or inputs (vertical dimension). The entries in the lower left region focus on a combination of higher level programming paradigms and optimizing compilers for programming GPUs. The entries in the lower right focus on device portability as well, and use machine description to generate optimized code for various hardware targets. However,no

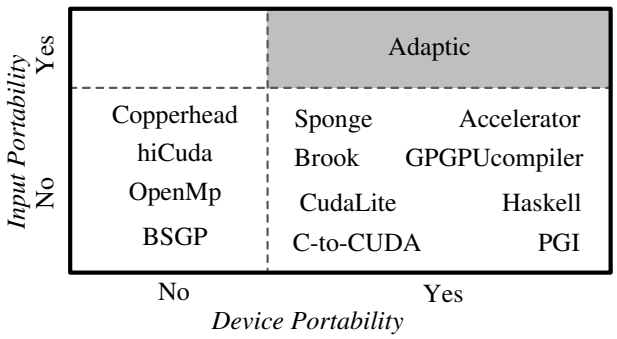


Figure 3.2: *Classification of prior works that have focused on improving GPU programmability based on their support for device portability and input portability.*

prior work has looked into ways to solve this problem.

In this work, we focus on tackling the input portability problem while providing GPU device portability. We employ a high-level streaming programming model to express target algorithms. This model provides explicit communication between various program kernels and its structured and constrained memory access lets the compiler make intelligent optimization decisions without having to worry about dependences between kernels. An *adaptive input-aware compilation system*, called *Adaptic*, is proposed that is capable of automatically generating optimized CUDA code for a wide range of input sizes and dimensions from a high-level algorithm description. Adaptic decomposes the problem space based on the input size into discrete scenarios and creates a customized implementation for each scenario. Decomposition and customization are accomplished through a suite of optimizations that include a set of memory optimizations to coalesce memory access patterns employed by the high-level streaming model and to efficiently execute algorithms that access several neighboring memory locations at the same time. In addition, a group of optimizations are introduced to effectively break up the work in large program segments for efficient execution across many threads and blocks. Finally, two optimizations are in-

troduced to combine the work of two segments so that execution overhead can be reduced.

An enhanced version of the performance model introduced in [42] is employed to predict application behaviour for each range of input size and dimensions. Based on these predictions, optimizations are applied selectively by the compiler. At runtime, based on the provided input to the program, the best version of the generated code is selected and executed to maximize performance. This method frees application developers from the tedious task of fine-tuning and possibly changing the algorithm for each input range.

The specific contributions offered by this work are as follows:

- We introduce input portability as a first class programmability challenge for GPUs and provide means to solve it.
- We propose input-aware optimizations to overcome memory related performance deficiencies and break up the work fairly between working units based on the input size and dimensions.
- We develop an adaptive compilation and runtime system that optimizes performance for various input ranges by conforming to the user input and identifying and adjusting required optimizations.

The rest of the chapter is organized as follows. In Section 3.2, the stream programming model is discussed. An overview of the Adaptic compiler is given in Section 3.3, while Section 3.4 describes the proposed input-aware optimizations in detail. Experiments are presented in Section 3.5. Related works are discussed in Section 3.6, and finally, Section 3.7 concludes the chapter.

3.2 Background

Exposed communication and an abundance of parallelism are the key features making stream programming a flexible and architecture-independent solution for parallel programming. In this work, we employ a stream programming model based on Synchronous Data Flow (SDF) models [53]. In SDF, computation is performed by *actors*, which are independent and isolated computational units, communicating only through data-flow buffers such as FIFOs. SDF, and its many variants, expose input and output processing rates of actors. This provides many optimization opportunities that can lead to efficient scheduling decisions for assignment of actors to cores, and allocation of buffers in local memories.

One way of writing streaming programs is to include all the computation performed in an actor inside a *work* method. This method runs repeatedly as long as the actor has data to consume on its input port. The amount of data that the work method consumes is called the *pop* rate. Similarly, the amount of data each work invocation produces is called the *push* rate. Some streaming languages, including StreamIt [109], also provide non-destructive reads, called *peek*, which do not alter the state of the input buffer. In this work, we use the StreamIt programming language to implement streaming programs. StreamIt is an architecture-independent streaming language based on SDF and allows the programmer to algorithmically describe the computational graph. In StreamIt, actors can be organized hierarchically into *pipelines* (i.e., sequential composition), *split-joins* (i.e., parallel composition), and *feedback loops* (i.e., cyclic composition).

To ensure correct functionality in StreamIt programs, it is important to create a steady state schedule which involves rate-matching of the stream graph. There is a buffer between

each two consecutive actors and its size is determined based on the program's input size and pop and push rates of previous actors. Rate-matching assigns a repetition number to each actor. In a StreamIt schedule, an actor is enclosed by a *for-loop* that iterates as many times as this repetition number.

Finally, since StreamIt programs are incognizant of input size and dimensions, Adaptic's input code is the same for all inputs but the output implementation will be different for various input sizes.

3.3 Adaptic Overview

The Adaptic compiler takes a platform-independent StreamIt program, ranges of its possible input size and dimension values, and the target GPU as input, and generates optimized CUDA code based on those ranges and the target. A StreamIt program consists of several actors that can be described as fine-grained jobs executed by each thread. Each actor in the StreamIt graph is converted to a CUDA kernel with a number of threads and blocks. By performing input-aware stream compilation, Adaptic decides how many threads and blocks to assign to the CUDA kernel generated for each actor. Figure 3.3 shows Adaptic's compilation flow that consists of four main components: baseline input-unaware optimizations, performance model, input-aware optimizations, and CUDA code generation. In addition, a matching runtime system selects appropriate kernels and sets their input parameters according to the program input at execution time. This section gives an overview of these four components as well as the runtime kernel management, while Section 3.4 details our proposed input-aware optimizations.

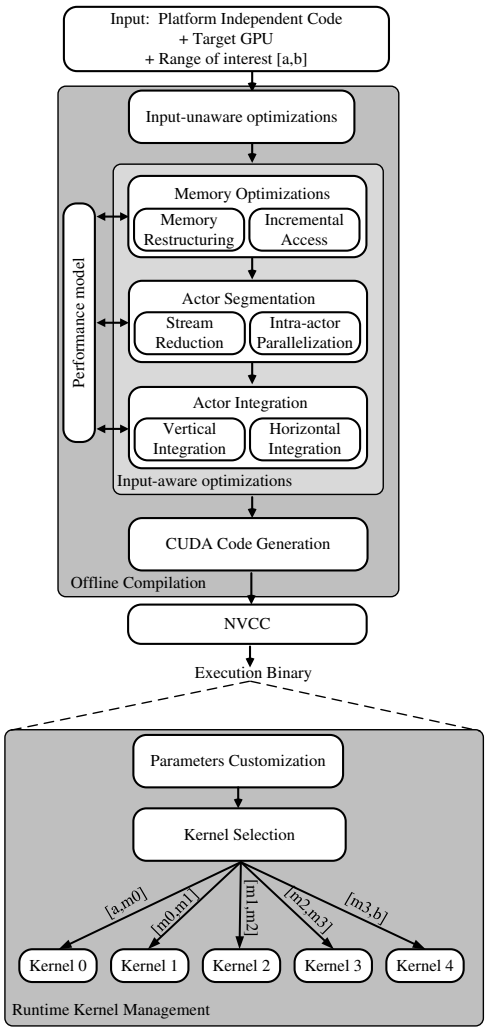


Figure 3.3: *Compilation flow in Adaptic.*

Input-unaware Optimizations: This step performs a set of input-unaware basic optimizations on the program and decides whether each actor should be executed on the CPU or GPU. This decision may be changed later by input-aware optimizations. Input-unaware optimizations are similar to those introduced in [44]. They include optimizations such as loop unrolling, data prefetching, and memory transfer acceleration. They can be used to generate CUDA code that is reasonably optimized and works for all input sizes, but gains its best performance for certain ranges of input and is suboptimal outside those ranges.

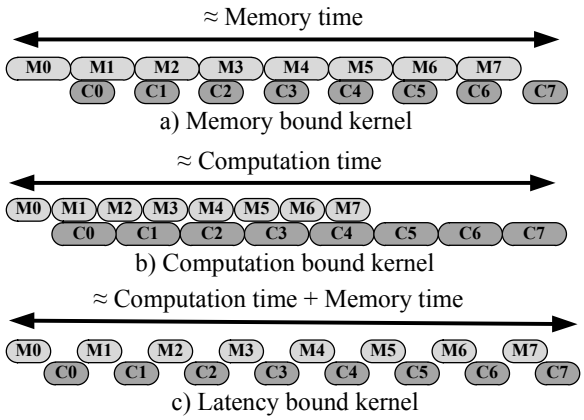


Figure 3.4: *Three different types of kernels in Adaptic's performance model.*

Performance Model: Adaptic relies on a high-level performance model to estimate the execution time of each kernel and to decide on using different optimizations for various problem sizes and GPU targets. This model is similar to the one described in [42], and classifies CUDA kernels into three categories of memory-bound, computation-bound, and latency-bound. Figure 3.4 illustrates a high-level overview of these kernel types.

As shown in the figure, memory-bound kernels have enough warps to efficiently hide the computation latency. Execution time of each warp is dominated by memory accesses, which are overlapped with computation. Therefore, in these kernels Adaptic estimates the execution time of the kernel by the total time spent on memory accesses. This estimation is computed based on the number of coalesced and non-coalesced accesses and the number of synchronization points, all of which are dependent on the input and can be computed at compile time as a function of input size and dimensions. GPU target is also an important factor in computing the performance estimation.

In computation-bound kernels, since most of the time is spent on computation, the execution time can be estimated to be the total computation time. It should be noted that in these kernels, a large number of active warps is also assumed so that the scheduler would be

able to hide memory access latencies with computation. The performance model estimates the computation time using the number of computation instructions and synchronization points which are both input-dependent. Similar to memory-bound kernels, GPU target affects the execution time estimation here as well.

The last category, latency-bound kernels, are those that do not have enough active warps on each SM, and the scheduler cannot hide the latency of the computation or memory by switching between warps. Execution time of these latency-bound kernels is estimated by adding up the computation and memory access times. The GPU determines how many active warps are needed for effectively hiding the latency. There are two situations that lead to a small number of active warps and make the kernels latency-bound. First, if there is not enough data parallelism in the kernel, not many thread blocks can be launched at a time and therefore few active warps are assigned to each SM. In addition, when each thread block uses a large portion of resources such as shared memory or registers, due to the lack of resources, the GPU scheduler can not assign enough thread blocks to each SM.

In order to determine the type of each kernel, Adaptic counts the number of active warps on each SM. Based on this number and the target GPU, it determines whether the kernel is latency-bound or not. If not, Adaptic treats that kernel as both memory-bound and computation-bound and calculates the corresponding execution cycles. The maximum of these two numbers determines the final kernel category. Based on these categories, The performance model estimates the execution time of a kernel both before and after applying each optimization as a function of input dimensions. The performance break-even points determine the dimensions at which the corresponding optimization should be enabled or disabled.

Input-aware Optimizations: At each input-aware optimization phase, its potential performance impact for all input ranges is estimated using the model. These input ranges are provided by previous input-aware phases. If the optimization is beneficial, it is added to the optimization list for the whole range. However, if the optimization is only suitable for a subset of that range, Adaptic divides the range into smaller subranges, and populates optimization lists for each new subrange accordingly. In other words, Adaptic divides up operating input ranges to subranges if necessary, and applies different optimizations to each subrange. Therefore, separate kernels should be later generated for these subranges.

Code Generation: At the end of the compilation flow, the code generation stage generates optimized CUDA kernels for each input range based on optimization lists constructed by the optimization phase and the performance model. Since the performance model uses the target specifications to make optimization decisions, code generation is different for different targets. In addition, necessary code for runtime kernel management is also generated by the code generation unit based on the kernels and their operating input ranges. All these codes are later translated to a binary using the native CUDA compiler.

Runtime Kernel Management: A runtime kernel management unit is developed to dynamically select a properly optimized kernel at runtime based on the program input. This unit also determines the values of parameters that should be passed to each kernel at launch time including the number of blocks, number of threads per block, and the size of allocated shared memory. In order to remove kernel management overhead at runtime, this unit is completely executed on the CPU during the initial data transfer from CPU to GPU.

3.4 Input-aware Optimizations

As mentioned in Section 3.1, several factors such as inefficient use of memory bandwidth, unbalanced workload across processors, and excessive number of threads lead to ineffectiveness of input-unaware optimizations in sustaining performance across different inputs. The goal of input-aware optimizations in this work is to deal with these inefficiencies.

Two *memory optimizations* are introduced in Section 3.4.1 to solve inefficient use of local or off-chip memory bandwidth. In addition, two other sets of optimizations, namely *actor segmentation* and *actor integration* are detailed in Sections 3.4.2 and 3.4.3 respectively to tackle both unbalanced processor workload and excessive number of threads.

3.4.1 Memory Optimizations

In this section, two memory optimizations, *memory restructuring* and *incremental memory access* are explained.

3.4.1.1 Memory Restructuring

One of the most effective ways to increase the performance of GPU applications is coalescing off-chip memory accesses. When all memory accesses of one warp are in a single cache line, the memory controller is able to coalesce all accesses into a single global memory access. Figure 3.5(a) illustrates how an actor with four pops and four pushes accesses global memory. In this example, each actor in each thread accesses four consecutive memory words. The first pop operations in threads 0 to 64 access memory word locations 0, 4,

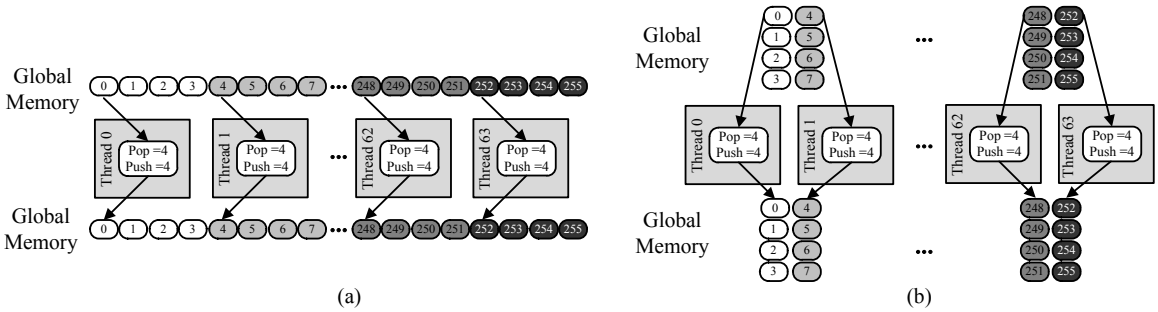


Figure 3.5: *Memory restructuring optimization. (a) Global memory access pattern of an actor with four pops and four pushes. Since accessed addresses are not adjacent, accesses are not coalesced. (b) Access patterns after memory restructuring. Accessed addresses are adjacent at each point in time and accesses are all coalesced.*

8,. . . , 252, second pop operations access locations 1, 5, 9,. . . , 253, etc. Since these locations are not consecutive in memory, non-coalesced global memory accesses occur, leading to poor performance.

There are two ways to coalesce these memory accesses. One way is to transfer all data to the shared memory in a coalesced manner and since shared memory is accessible by all threads in a block, each thread can work on its own data. In this method, each thread fetches other threads' data from global memory as well as part of its own data. The same method can be applied for write backs to global memory as well. All threads write their output to shared memory and then they transfer all data in a coalesced pattern to the global memory. Although using shared memory for coalescing accesses can improve performance, it has two shortcomings: number of threads is limited by the size of shared memory and the total number of instructions is increased due to address computations.

We use another method for coalescing accesses and that is to restructure the input array in a way that each pop access in all threads accesses consecutive elements of one row of the input in global memory. Figure 3.5(b) shows how this restructuring coalesces all memory accesses without using shared memory. This method has the advantage of minimizing the

number of additional instructions and does not limit the number of threads by the size of shared memory. In addition, since this optimization is not using shared memory to coalesce off-chip memory accesses, shared memory can be utilized to store real shared data.

This optimization is not applicable when there are two or more actors with mismatching push and pop rates in the program. In those cases, rate matching buffers between kernels also have to be restructured, which involves extra write and reads from global memory, leading to poor performance.

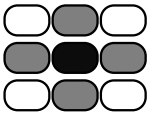
However, as the work in [108] shows, most consecutive actors in streaming benchmarks have matching rates. Therefore, using memory restructuring would be beneficial. The CPU can restructure the data at generation time and transfer it to the global memory of the GPU. The GPU launches kernels and when all of them are finished, the CPU reads back the output data. Due to the dependency of pop and push rates of some of the actors are on input size, this optimization can have different effects for various sizes.

In addition to coalescing global memory accesses, memory restructuring can also be applied to shared memory to remove bank conflicts. After applying this optimization, all threads access consecutive addresses in shared memory. Since adjacent addresses in shared memory belong to different shared memory banks, there would be no bank conflicts.

3.4.1.2 Incremental Memory Access

This optimization can be applied to actors that access multiple neighboring points in their input array. The indexes of these accesses increase linearly in each iteration. In StreamIt, non-destructive read (peek) is used in these actors to read the neighbors' data. These actors are most common in simulation benchmarks, for instance, the temperature of

```
5-Point Stencil(pop,peek:size, push:size)
  for (index=0; index<size; index++)
    if (not on edge)
      Top    = peek(index - width)
      Bottom = peek(index + width)
      Right  = peek(index + 1)
      Left   = peek(index - 1)
      Center = peek(index - 1)
    push(func(Top,Bottom,Right,Left,center))
```



(a) (b)

Figure 3.6: *Incremental-access actors. (a) An example StreamIt code of a five-point stencil actor. (b) Memory access pattern of this actor.*

each point on a surface is computed based on the temperature of its neighbors. Figure 3.6(a) shows an example StreamIt code of a *five-point stencil* actor that has incremental access pattern and Figure 3.6(b) illustrates its corresponding access pattern. In this example, each element is dependent on its top, bottom, right, and left elements. Each thread first reads all top elements, which are consecutive, leading to coalesced memory accesses. The same pattern holds for bottom, right, left and center elements. However, the main problem with this class of actors is excessive global memory accesses. For instance, accessing all top, bottom, left, right and center elements in each thread simply means accessing the whole input five times.

An efficient way to alleviate this problem is to use shared memory such that each block brings in one tile of data to shared memory and works on that. Since the data close to tile edges is needed for both neighboring tiles, tiles should be overlapped. These overlapping regions, called *halo* parts, are brought in for each tile at all four edges. Since branch divergence occurs only within a warp [72], both tile and halo part widths should be multiples of warp size to make all accesses coalesced and prevent control flow divergence.

These halo parts should be as small as possible to minimize extra memory accesses. The ratio of the halo part size to the main tile size is decreased by merging several tiles and

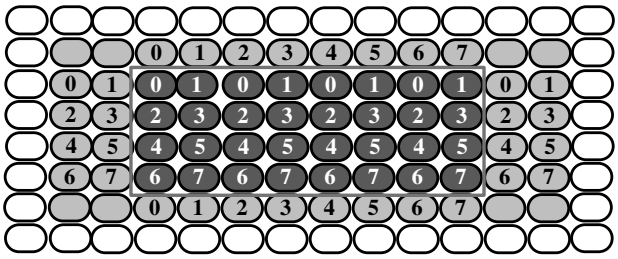


Figure 3.7: *A super tile assigned to one block. Dark gray addresses are main part and light gray parts are halo parts. Numbers in each small box indicates which thread reads this address.*

forming a super tile. Each super tile is assigned to one block and each thread computes several output elements in different tiles. In this case, each block brings in a super tile from global memory to shared memory, performs the computation, and writes back the super tile to global memory.

Figure 3.7 shows a super tile assigned to a block in our example. Dark gray elements construct the main tiles while the light gray elements are halo parts. The number in each element indicates the thread index reading that element's address. In this example, warp size is equal to 2 and there are 8 threads in each block. Each tile is 4x2 and by merging four tiles together, one super tile with 4x8 elements is formed. Since all width values should be multiples of warp size to maintain memory coalescing, the width of right and left halo parts in this example are set to 2.

Increasing the size of super tiles leads to an increase in the allocated shared memory for each block, which in turn, could result in lower number of concurrent blocks executed on each GPU SM. Since this issue may change the type of kernel from computation-bound or memory-bound to latency-bound, the data size processed by each block should be chosen carefully. Adaptic uses the following reuse metric to find the optimal shape and size for each tile:

```
Incremental-access Kernel <<<Blocks, threads>>>
  Top halo      → shared memory
  Bottom halo   → shared memory
  Right halo    → shared memory
  Left halo     → shared memory
  for tile in super tile
    tile → shared memory
  sync();
  Do computations and write results
```

Figure 3.8: *A generic incremental-access CUDA code. First, different halo parts and the super tile are moved from global to shared memory. Subsequently, computations are performed on the shared memory data.*

$$\begin{aligned} Reuse\_Metric &= \frac{Memory\_Accesses\_Served}{Extra\_Parts\_Size} \\ &= \frac{\sum_{Tile} Element\_Accesses}{Halo\_Size} \end{aligned}$$

As can be seen, maximizing the number of served memory accesses while minimizing the size of extra halo parts, maximizes the reuse metric. In this formula, $Element\_Accesses$ is the number of times each element in the shared memory is accessed during the computation of the whole output matrix, and the summation is taken over all elements in the tile. Since the best tile is the one with small halo parts that can compute a large chunk of output, Adaptic uses rectangular tiles with maximum $Reuse_Metric$s if possible. However, the size of each super tile should not be more than the maximum shared memory per block, which is a constant value based on the target GPU. The super tile's size is dependent on the input size. For small input sizes it is beneficial to use smaller super tiles in order to have more blocks. Large super tiles are advantageous for large input sizes to reduce excessive memory accesses.

Once the size of super tiles and halo parts are determined, the output CUDA code will

be similar to the code shown in Figure 3.8. First, the kernel reads in the super tile and all its halos to the shared memory, after which synchronization makes shared memory visible to all threads. Subsequently, each block starts working on its own data residing in the shared memory to perform the computation and output the results.

3.4.2 Actor Segmentation

Optimizations in this category attempt to divide the job of one large actor between several threads/blocks to increase the performance. In order to have balanced workload across processors with efficient number of threads, this segmentation should be done based on the input size.

Reduction is one of the important algorithms used in many GPU applications. The goal of *stream reduction* optimization is to efficiently translate reduction operations to CUDA in streaming programs. *Intra-actor parallelization*'s goal is to break the dependency between iterations of large loops and make them more amenable to execution on GPUs.

3.4.2.1 Stream Reduction

A reduction operation generally takes a large array as input, performs computations on it, and generates a single element as output. This operation is usually parallelized on GPUs using a tree-based approach, such that each level in the computation tree gets its input from the previous level and produces the input for the next level. In uniform reduction, each tree level reduces the number of elements by a fixed factor and the last level outputs one element as the final result. The only condition for using this method is that the reduction operation needs to be associative and commutative.

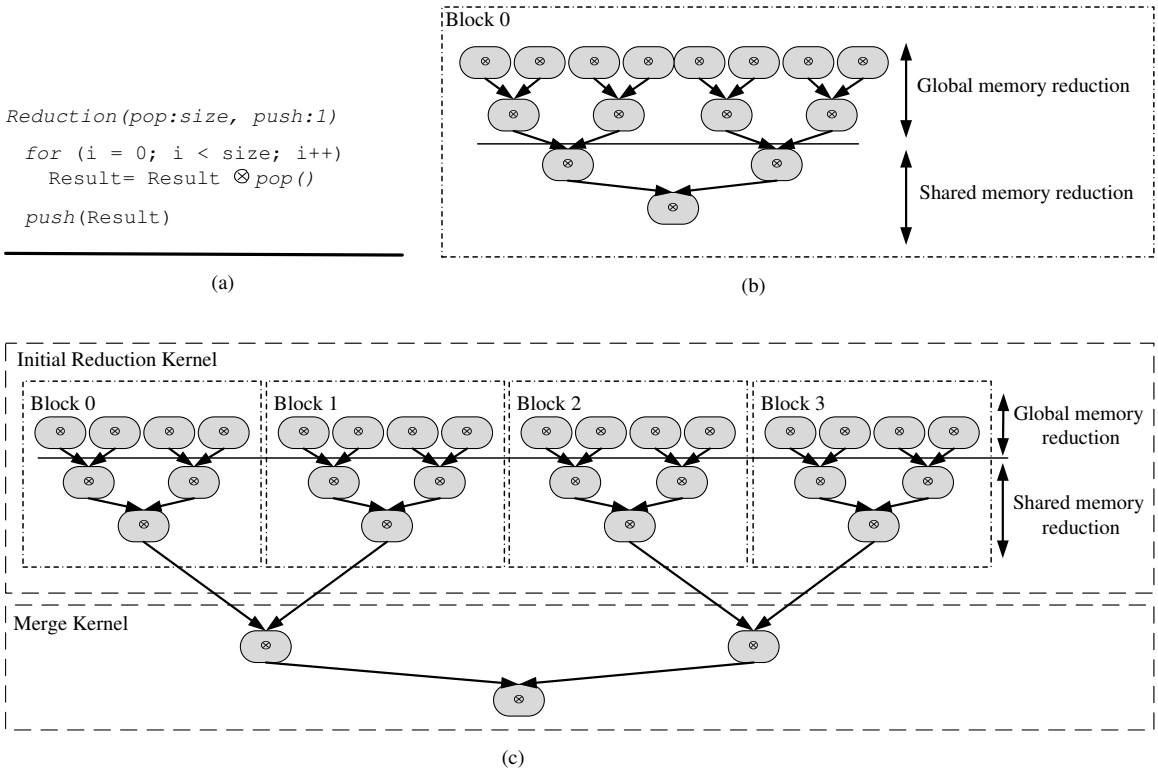


Figure 3.9: *Stream reduction technique. (a) StreamIt code for a reduction actor. (b) Each block is responsible for computing output for one chunk of data in two phases. In the first phase, each thread reads from global memory and writes reduction output to the shared memory and in the second phase, shared memory data is reduced to one output element. (c) In the two kernel approach, different blocks of the first kernel work on different chunks of data and the second kernel reads all reduction kernel's output and compute final result.*

A naive way of implementing the tree-based approach in StreamIt is to represent each tree node as an individual actor with small pop/push rates. Executing one kernel for each small actor would make the kernel launching overhead significant and degrade the performance dramatically. Another method of representing reduction in StreamIt is using one filter that pops in the whole input array and pushes the final result as shown in Figure 3.9(a). Optimizations in [44] can not translate this actor to an efficient kernel due to the limited number of possible in-flight threads.

On the other hand, Adaptic automatically detects reduction operations in its streaming graph input using pattern matching. After this detection phase, it replaces the reduction

actor with a highly optimized kernel in its output CUDA code based on the input size and the target GPU. This reduction kernel receives N_{arrays} different arrays with $N_{elements}$ elements each as input, and produces one element per array as output. Data is initially read from global memory, reduced and written to shared memory, and read again from shared memory and reduced to the final result for each array. In this work, we introduce two approaches for translating reduction actors to CUDA kernels.

When the array input size, $N_{elements}$, is small compared to the total number of input arrays, N_{arrays}, Adaptic produces a single reduction kernel in which each block computes the reduction output for one input array. Thus, this kernel should be launched with N_{arrays} blocks. This approach is beneficial for large array counts so that Adaptic can launch enough blocks to fill up the resources during execution.

However, when the array input size ($N_{elements}$), is large compared to total number of input arrays (N_{arrays}), the reduction output for each array is computed individually by two kernels. The first kernel, called the initial reduction kernel, chunks up the input array and lets each block reduce a different data chunk. The number of these blocks, $N_{initial\_blocks}$ is dependent on the value of $N_{elements}$ and the target GPU. Since there is no global synchronization between threads of different blocks, results of these blocks ($N_{initial\_blocks} * N_{arrays}$ elements) are written back to global memory. Subsequently, another kernel, called the merge kernel, is launched to merge the outputs from different blocks of the initial reduction kernel down to N_{arrays} elements. In the merge kernel, each block is used to compute the reduction output of one input array. Therefore, this kernel should be launched with N_{arrays} blocks.

Figure 3.10 shows Adaptic's resulting CUDA code for the initial reduction kernel. In

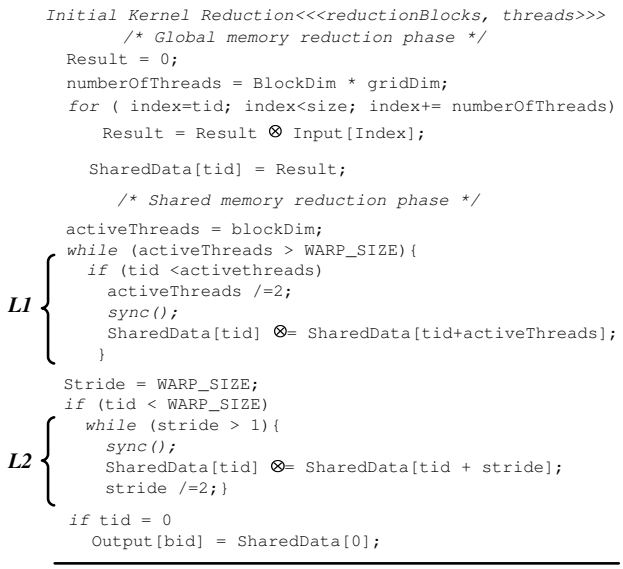


Figure 3.10: *The initial reduction kernel's CUDA code.*

the first phase, the input array in global memory is divided into chunks of data. Each thread computes the output for each chunk, and copies it to shared memory. The amount of shared memory usage in each block is equal to $Threads\_per\_Block * Element\_Size$. As discussed in Section 3.4.1.1, all global memory accesses are coalesced as a result of memory restructuring and there would be no bank conflicts in shared memory in this phase.

In the next phase, the results stored in shared memory are reduced in multiple steps to form the input to the merge kernel. At each step of this phase, the number of active threads performing reduction are reduced by half. Loop L1 in Figure 3.10 represents these steps. They continue until the number of active threads equals the number of threads in a single warp. At this point, reducing the number of threads any further would cause control-flow divergence and inferior performance. Therefore, we keep the number of active threads constant and just have some threads doing unnecessary computation (Loop L2 in Figure 3.10). It should be noted that after each step, synchronization is necessary to make shared memory changes visible to other threads. Finally, the thread with $tid = 0$ computes

the final initial reduction result and writes it back to the global memory.

3.4.2.2 Intra-actor Parallelization

The goal of intra-actor parallelization is to find data parallelism in large actors. As mentioned before, it is difficult to generate optimized CUDA code for actors with large pop or push rates, consisting of loops with high trip counts. This optimization breaks these actors into individual iterations which are later efficiently mapped to the GPU. Using data flow analysis, Adaptic detects cross-iteration dependences. If no dependence is found, Adaptic simply assigns each iteration to one thread and executes all iterations in parallel. It also replaces all induction variable uses with their correct value based on the thread index.

In some cases, Adaptic breaks the dependence between different iterations by eliminating recurrences. Suppose the loop contains an accumulator variable $count$ incremented by a constant C in every iteration ($count = count + C$). This accumulation causes cross-iteration dependences in the loop, making thread assignment as described impossible. However, intra-actor parallelization technique breaks this dependence by changing the original accumulation construct to $count = initial\_value + induction\_variable * C$ and making all iterations independent.

In general, this optimization is able to remove all linear recurrence constructs and replace them by independent induction variable-based counterparts. This is similar to the accumulator expansion optimization that parallelizing compilers perform to break these recurrences and exploit loop level parallelism on CPUs [116].

3.4.3 Actor Integration

This optimization merges several actors or threads together to balance threads' workloads based on the input size in order to get the best performance. Two types of actor integration optimization are introduced in this work. Vertical integration technique reduces off-chip memory traffic by storing intermediate results in the shared rather than global memory. Horizontal integration technique reduces off chip memory accesses and synchronization overhead and also lets the merged actors share instructions.

3.4.3.1 Vertical Integration

During this optimization, Adaptic vertically integrates some actors to improve performance by reducing memory accesses, removing kernel call overhead, and increasing instruction overlap. The reason for its effectiveness is that integrated actors can communicate through shared memory and there is no need to write back to the global off-chip memory. Also, integrating all actors together results in one kernel and global memory accesses of this one kernel are coalesced by the memory restructuring optimization. However, since input and output buffers of the middle actors in the integrated kernel are allocated in the shared memory, the number of active threads executing these actors are limited by the size of shared memory. This limitation often prevents Adaptic from integrating all actors together. Based on the performance model, Adaptic finds the best candidates for this optimizations.

Since push and pop rates of some actors are dependant on the input size, this optimization is beneficial for some ranges of input size. To maintain a steady state schedule, each actor is executed with a different number of threads based on its number of iterations as

```
Integration Kernel <<<Blocks, threads>>>
  if threadId < (# threads for A)
          A;
  sync();
  if threadId < (# threads for B)
          B;
  sync();
  if threadId < (# threads for C)
          C;
  sync();
```

Figure 3.11: *Generated CUDA code after integrating actors A, B, and C.*

shown in Figure 3.11.

Another optimization made possible after actor integration is replacing *transfer actors* with index translation. Transfer actors are the ones that do not have any computation part and their task is only to reorganize data in the input buffer and write it to the output buffer. Since input and output buffers of the middle actors in integrated kernels are both allocated in the shared memory, there is no need to read the data from input buffer, shuffle it, and write it to the output buffer. This task can be done by index translation. Index translation gets thread indexes based on the transfer pattern, generates the new index pattern, and passes it to the next actor.

3.4.3.2 Horizontal Integration

The goal of horizontal integration is removing excessive computations or synchronizations by merging several threads or actors that can run in parallel. There are two kinds of horizontal integration techniques: horizontal actor integration and horizontal thread integration. In streaming languages, we use a duplicate splitter to allow different actors to work on the same data. In this case, instead of launching one kernel for each actor, one kernel is launched to do the job of all the actors working on the same data. Therefore, in

addition to reducing kernel overheads, memory access and synchronization overheads are also reduced. For example, assume there is a program that needs maximum and summation of all elements in an array. Instead of running two kernels to compute these values, Adaptic launches one kernel to compute both. In this case, off-chip memory accesses and synchronizations only happen once instead of twice.

Horizontal thread integration merges several consecutive threads working on consecutive memory locations in one kernel. This method reduces the number of threads and blocks used by the kernel. Merged threads can share part of the computation and decrease the number of issued instructions. When the number of kernel blocks is high, it is beneficial to use horizontal thread integration to reduce the number of threads and blocks and allow them to run in parallel. Otherwise it is better not to integrate threads and have more threads with less work to increase the possibility hiding memory latency by switching between threads.

3.5 Experiments

A set of benchmarks from the NVIDIA CUDA SDK [69] and the CUBLAS library [70] are used for evaluation. We developed StreamIt versions of these benchmarks, compiled them with Adaptic, and compared their performance with the original hand-optimized benchmarks. A case study is performed on a CUBLAS benchmark to investigate the effect of our optimizations over a wide range of inputs. We also present case studies on two real world applications, biconjugate gradient stabilized method [112] and support vector machine [22], executed on two different GPUs, to demonstrate how Adaptic performs

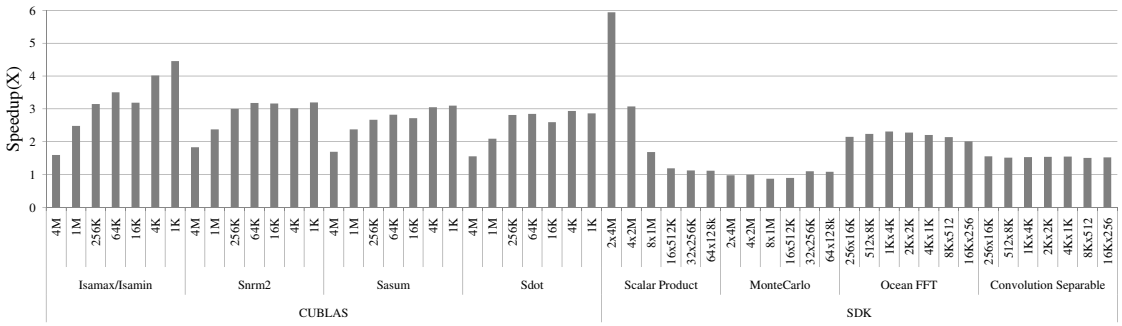


Figure 3.12: *Adaptic-generated code speedups normalized to the hand-optimized CUDA code for 7 different input sizes.*

on larger programs with many actors and on different GPU targets. Adaptic compilation phases are implemented in the backend of the StreamIt compiler [109] and its C code generator is modified to generate CUDA code. Both Adaptic's output codes and the original benchmarks are compiled for execution on the GPU using NVIDIA nvcc 3.2. GCC 4.1 is used to generate the x86 binary for execution on the host processor. The target system has an Intel Xeon X5650 CPU and an NVIDIA Tesla C2050 GPU with 3GB GDDR5 global memory. The other system used for experiments in Sections 3.5.2.2 and 3.5.2.3 has an Intel Core 2 Extreme CPU and an NVIDIA GeForce GTX 285 GPU with 2GB GDDR2 global memory.

3.5.1 Input Portability

In order to show how Adaptic handles portability across different input problem sizes, we set up seven different input sizes for each benchmark and compared their performance with the original CUDA code running with the same input sizes. It should be noted that these seven input sizes are chosen from the working range of the CUDA benchmarks, as there are many sizes for which the SDK benchmarks would not operate correctly.

Figure 3.12 shows the results for eight CUDA benchmarks that were sensitive to changes

in the input size, while results for input-insensitive benchmarks are discussed in Section 3.5.3. As can be seen, Adaptic-generated code is better than the hand-optimized CUDA code for all problem sizes in *Scalar Product*, *MonteCarlo*, *Ocean FFT*, and *Convolution Separable* from the SDK, and *Isamax/Isamin*, *Snrm2*, *Sasum*, and *Sdot* from CUBLAS. A combination of actor segmentation and actor integration were used to optimize all CUBLAS benchmarks. In addition to these optimizations, memory restructuring was applied to *Sdot*.

Sdot is computing the dot product of two vectors. For large vectors, using the two kernel reduction is beneficial, but for small sizes, in order to reduce kernel launch overhead, Adaptic uses the one kernel reduction. Using input-aware optimizations leads to upto 4.5x speedup in this benchmark compared to the original program. *Convolution Separable* has two actors, and processes data row-wise in one and column-wise in the other. Memory optimizations are effective for this benchmark as both of these two actors have incremental memory access pattern. Therefore, as the input becomes smaller, Adaptic reduces the super tile sizes adaptively to retain the high number of blocks and, therefore, achieves better performance than the baseline hand-optimized code. *OceanFFT* also has an incremental access actor and Adaptic uses different tile sizes to improve performance over the hand-optimized code. *Scalar Product* computes scalar products of pairs of vectors. The original benchmark uses the single kernel reduction, and it achieves good performance when there are many pairs of vectors in the input. However, for fewer pairs of vectors, it is better to use the whole GPU to compute the result for each pair. Using the two kernel reduction for those inputs, Adaptic is able to achieve upto 6x speedup compared to the original hand-optimized version.

MonteCarlo performs about the same as the original hand-optimized version. The reason is that the original benchmark already has two kernels performing the same task, but optimized for different ranges of input problem sizes. In other words, *MonteCarlo* has originally been developed in an input portable way. Therefore, the output of Adaptic is similar to the original version and the performance is the same for all sizes.

Since Adaptic generates different kernels for some actors in the input streaming program, the output binary size could be larger than the original binary optimized for one specific range. In our experiments including the case studies, Adaptic's output binaries were on average 1.4x and upto 2.5x larger than their input-unaware counterparts, which is quite reasonable considering the fact that some kernels could have upto five different versions for various input ranges. However, because each program also has kernels with one versions, the combination leads to this moderate code size increase.

These results further show the fact that our approach in Adaptic is able to adaptively generate optimized CUDA code for different problem sizes without any source code modifications.

3.5.2 Case studies

3.5.2.1 Transposed Matrix Vector multiplication

In this section, we look into the effects of our optimizations on the performance of the transposed matrix vector multiplication benchmark from CUBLAS over a wide range of input sizes and dimensions. As was mentioned in Section 3.1, the original benchmark cannot provide sustainable performance gains for different input dimensions. However,

with the aid of input-aware optimizations, Adaptic is able to generate five different kernels with different structures, where each kernel is parameterized to get better performance for a specific range of input dimensions. At runtime the proper kernel is launched based on the program input.

In the first kernel, which is beneficial for matrices with many columns and few rows, Adaptic uses the two kernel version of reduction. For each row, one kernel is launched and the whole GPU is used to compute the dot product of one row with the input vector. The second kernel is a single-kernel reduction function where each block is responsible for one row. This kernel achieves its best performance for square matrices. In the third kernel, in addition to the single-kernel reduction function, by using horizontal thread integration, Adaptic adaptively merges several rows and each block is responsible for computing several dot products instead of one. This kernel is beneficial for matrices with more rows than columns. The fourth kernel is also similar to the single-kernel reduction, except that in its shared memory reduction phase, each thread is responsible for computing one output. The last kernel generated by Adaptic achieves its best performance for matrices with many rows and few columns. In this case, the size of each row is small and the corresponding actor has small pop rates. For this kind of actor, our baseline optimizations are effective in generating efficient code. Therefore, Adaptic does not need to add optimization to that. In this kernel, each thread is responsible for computing the dot product of a single row and the input vector.

Figure 3.13 compares the performance of this benchmark with Adaptic-generated code for three different matrix sizes over a range of matrix dimensions. As it can be seen, although for some input dimensions Adaptic's performance is really close to CUBLAS, for

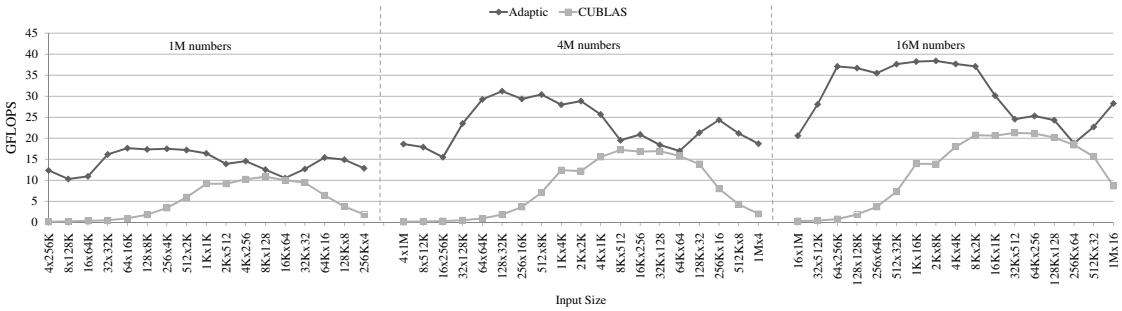


Figure 3.13: *Transposed matrix vector multiplication performance comparison of CUBLAS and Adaptic.*

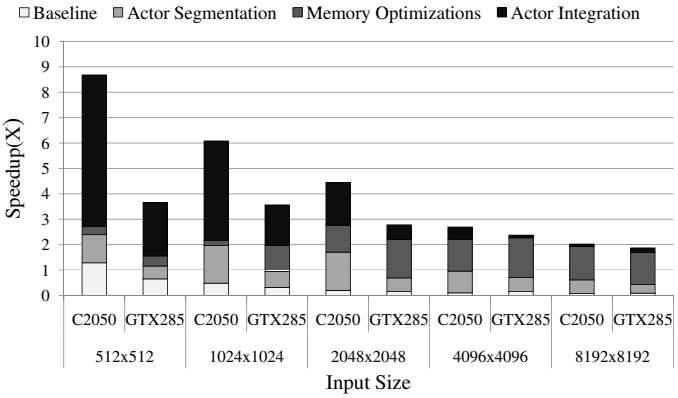


Figure 3.14: *Performance of the Adaptic-generated Biconjugate gradient stabilized method benchmark normalized to the CUBLAS implementation on two different GPU targets.*

most of them Adaptic outperforms CUBLAS by a large margin. This figure shows how Adaptic can adaptively maintain its performance across various input ranges.

3.5.2.2 Biconjugate gradient stabilized method

The biconjugate gradient stabilized method (BiCGSTAB) is an iterative method used for finding the numeral solution of nonsymmetric linear systems such as $Ax=B$ for x where A is a square matrix. This method has 11 linear steps that can be written easily with the CUBLAS library for GPUs. We wrote this program both in StreamIt and CUDA with CUBLAS functions and measured the performance of the two for different sizes of A. Figure 3.14 shows an in-depth comparison and breakdown of the effects of Adaptic's indi-

vidual optimizations on this benchmark for different input sizes across two GPU targets - NVIDIA Tesla C2050 and GTX285. The baseline in this figure is the generated code after only applying size-unaware optimizations. The Sgemv, Sdot, Sscal and Saxpy CUBLAS functions were used to implement the CUDA version of this benchmark. The problem of using the CUBLAS library is that the programmer should split each step into several sub-steps to be able to use CUBLAS functions. Execution of these sub-steps leads to more memory accesses and kernel launch overhead.

On the other hand, Adaptic merges all these sub-steps together and launches a single kernel for one step. As shown in Figure 3.14, most of the speedup for small sizes comes from the integration optimization. Since most of the execution time is spent in matrix vector multiplication for large sizes such as 8192x8192, the effect of integration is not as high for these sizes. However, actor segmentation that generates smaller actors and increases parallelism, and memory restructuring play more important roles in achieving better performance for larger sizes.

3.5.2.3 Nonlinear Support Vector Machine Training

Support Vector Machines (SVMs) are used for analyzing and recognizing patterns in the input data. The standard two class SVM takes a set of input data and for each input predicts which class it belongs to among the two possible classes. This classification is based on a model, which is generated after training with a set of example inputs. Support vector machine training and classification are both very computationally intensive.

We implemented a StreamIt version of this algorithm based on the implementation in [22]. The kernel function used for training is the Gaussian Radial Basis Function (RBF).

```
Distance( pop:2×size, push:1)
   for ( index = 0 ; index < size; index ++)
        diff = pop() - pop()
        result = result + (diff × diff)
   push(result )

Final( pop:1, push:1)
   push(exp(-gamma × pop()))
```

Figure 3.15: *StreamIt implementation of the RBF kernel.*

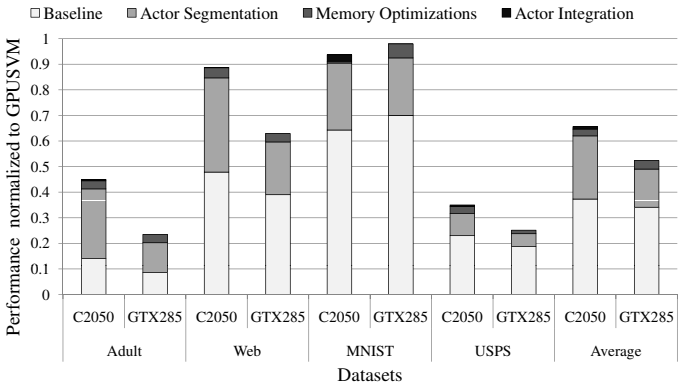


Figure 3.16: *Performance of the Adaptic-generated SVM training benchmark compared to the hand-optimized CUDA code in the GPUSVM implementation on two different GPU targets.*

The StreamIt implementation of this function is shown in Figure 3.15. It has two actors that compute the Gaussian RBF of the input using equation: $\Phi(\overrightarrow{x_i}, \overrightarrow{x_j}) = exp(-\gamma \parallel \overrightarrow{x_i} - \overrightarrow{x_j} \parallel^2)$.

Figure 3.16 shows the performance of the Adaptic-generated code compared to the GPUSVM [22] hand-optimized CUDA code in this benchmark for four different input datasets. On average, Adaptic achieves 65% of the performance of the GPUSVM implementation. The reason for the large performance gap in Adult and USPS datasets is that GPUSVM performs an application-specific optimization where it utilizes unused regions of the GPU memory to cache the results of some heavy computations' results. In case those computations have to be performed again, it simply reads the results in from the memory. Therefore, for input sets which cause a lot of duplicate computations, including *Adult* and *USPS*, GPUSVM performs better than Adaptic-generated code.

In this program, unlike the previous example, actor integration is not very effective

and most of the performance improvement comes from actor segmentation. On average, actor segmentation, memory restructuring, and actor integration improve the performance by 37%, 4%, and 1%, respectively.

3.5.3 Performance of Input Insensitive Applications

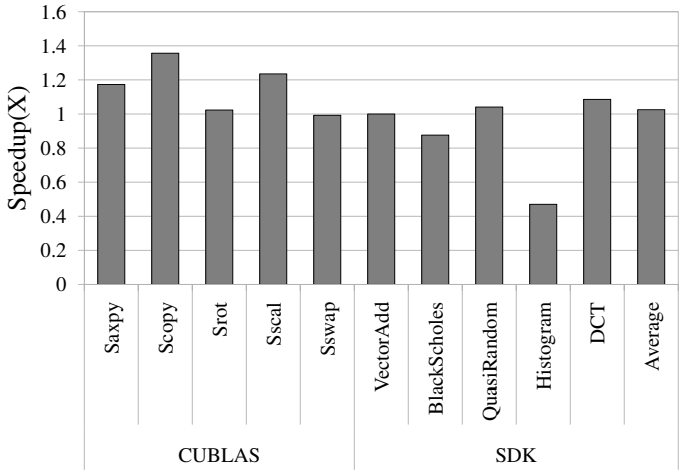


Figure 3.17: *Adaptic-optimized code speedups normalized to the hand-optimized CUDA code, both running on the NVIDIA Tesla C2050.*

Although the main goal of Adaptic compiler is to maintain good performance across a wide range of inputs, it also performs well on the benchmarks that are not sensitive to input. Figure 3.17 shows the performance of Adaptic-optimized codes normalized to the original hand-optimized CUDA codes in these input insensitive benchmarks. All results are gathered for problem sizes that the CUDA codes are written for. As can be seen, a combination of Adaptic optimizations makes the average performance of our compiler generated code on par with the hand-optimized benchmarks, while writing StreamIt applications as the input to Adaptic involves much less effort by the programmer compared to the hand-optimized programs.

In *BlackScholes*, *VectorAdd*, *Saxpy*, *Scopy*, *Sscal*, *Sswap*, and *Srot*, due to their low rate

pop/push actors, memory restructuring proves quite effective in improving performance. Using this optimization, all accesses are coalesced without using shared memory. In these benchmarks, Adaptic allocates each input vector in one row of the two dimensional input array to make all accesses coalesced. In *BlackScholes*, due to the low number of instructions executed in each thread, adding one or two extra instructions by the compiler degrades performance. This sensitivity to the number of instructions is the reason behind the 20% performance degradation between Adaptic and the hand-optimized code in this benchmark. *VectorAdd*'s performance, on the other hand, is almost equal to the hand-optimized version.

In *DCT*, actor integration speeds up the program by 9% compared to the hand-optimized code. Finally, intra-actor parallelization makes *QuasiRandomGenerator* runs 4% faster than the baseline hand-optimized code. In the *Histogram* benchmark, the hand-optimized version is about 2x faster compared to the Adaptic-generated code. This performance degradation by Adaptic is mainly due to the fact that the granularity of computations in this benchmark are one byte and Adaptic automatically changes this granularity to 4 bytes. This leads to heavier threads compared to the hand-optimized version and causes slowdown. Working at the byte granularity in Adaptic requires a complex process to remove shared memory bank conflicts which is quite difficult in the automatic compiler generated code.

3.6 Related Work

The most common languages GPU programmers use to write GPU code are CUDA and OpenCL. Although these new languages partially alleviate the complexity of GPU pro-

gramming, they do not provide an architecture independent solution. There is an extensive literature investigating many alternative methods to support device portability.

Works in [20, 44, 111, 21, 45, 54, 55, 79, 38] focus on generating optimized CUDA code from higher levels of abstraction. The Sponge compiler [44] compiles StreamIt programs and generates optimized CUDA to provide portability between different GPU devices. The work in [111] compiles stream programs for GPUs using software pipelining techniques. Copperhead [21] provides a nested set of parallel abstractions expressed in the Python programming language. Their compiler gets Python code as input and generates optimized CUDA code. It uses built-in functions of Python such as sort, scan, and reduce to abstract common CUDA program constructs. The work in [55] automatically generates optimized CUDA programs from OpenMP programs. Works in [45] and [54] choose Haskell and BSGP as their input languages and compile them to CUDA. BSGP is bulk synchronous GPU programming language which is similar to sequential C with parallel primitives. Brook for GPUs [20] is one of the first papers about compilation for GPUs, which extends the C language to include simple data-parallel constructs. Compiling Matlab file to CUDA is also investigated in [79]. CnC_CUDA [38] use Intel's Concurrent Collections programming model to generate optimized CUDA code. All these works look into improving the programmability of GPUs, and in some cases, provide target device. However, Adaptic provides portability across different inputs as well as GPU targets. In addition, Adaptic employs various input-aware optimizations and its output performance is comparable to hand written CUDA code.

Several other works have focused on automatically optimizing CUDA kernels [122, 121, 42]. The work in [121] performs GPU code compilation with a focus on memory

optimizations and parallelism management. The input to this compiler is a naive GPU kernel function and their compiler analyzes the code and generates optimized CUDA code for various GPU targets. CUDA-Lite [122] is another compilation framework that takes naive GPU kernel functions as input and tries to coalesce all memory accesses by using shared memory. Hong et al. [42] propose an analytical performance model for GPUs that compilers can use to predict the behavior of their generated code. None of these works provide means to address the input portability problem.

There are other works that have focused on generating CUDA code from sequential input [39, 13, 117, 106]. hiCUDA [39] is a high level directive based compiler framework for CUDA programming where programmers need to insert directives into sequential C code to define the boundaries of kernel functions. The work in [13] is an automatic code transformation system that generates CUDA code from input sequential C code without annotations for affine programs. In [117], by using C pragma preprocessor directives, programmers help compiler to generate efficient CUDA code. In [106], programmers use C# language and a library to write their programs and let the compiler generate efficient GPU code.

Gordon et al.([37] and [36]) perform stream graph refinements to statically determine the best mapping of a StreamIt program to a multi-core CPU. Researchers have also proposed ways to map and optimize synchronous data-flow languages to SIMD engines [43], distributed shared memory systems [49]. In a recent work [108], the authors talk about the usefulness of different features of StreamIt to a wide range of streaming applications.

It should be noted that StreamIt language has several limitations that make writing general StreamIt programs difficult. First of all, StreamIt is limited to the well structured

programs with regular memory accesses. For applications with irregular memory accesses, it is hard to program in StreamIt. Hence, compiling this kind of program from StreamIt to CUDA is not practical. Similar to other streaming languages, only streaming benchmarks can be written by StreamIt. Writing codes that reuse one data often may lead to many duplication of input data and makes analyzing these actors very hard. In addition, removing all these duplications from the program is not straight forward and they results in more memory accesses and poor performance.

The work in [87] is one of the early studies about increasing performance of reduction on GPU and [119] tries to extract parallelism from complex reduction codes by using general reduction function. Ravi et al. [82] map reduction to heterogeneous system and divide computation between CPU and GPU. Mapping stencil loops to GPUs and tiling size trade-off are also studied by [13] and [64]. However, Adaptic applies input-aware optimizations adaptively and more generally on streaming applications to provide input portability.

3.7 Conclusion

GPUs provide an attractive platform for accelerating parallel workloads. However, their programming complexity poses a significant challenge to application developers. In addition, they have to deal with portability problems across both different targets and various inputs. While device portability has received a great deal of attention in the research community, the input portability problem has not been investigated before. This problem arises when a program optimized for a certain range of inputs, shows poor performance along different input ranges.

In this work, we proposed Adaptic, an adaptive input-aware compiler for GPUs. Using this compiler, programmers can implement their algorithms once using the high-level constructs of a streaming language and compile them to CUDA code for all possible input sizes and various GPUs targets. Adaptic, with the help of its input-aware optimizations, can generate highly-optimized GPU kernels to maintain high performance across different problem sizes. At runtime, Adaptic's runtime kernel management chooses the best performing kernel based on the input. Our results show that Adaptic's generated code has similar performance to the hand-optimized CUDA code over the original program's input comfort zone, while achieving upto 6x speedup when the input falls out of this range.

CHAPTER IV

Cooperative Loop Speculation

4.1 Introduction

In recent years, multicore CPUs have become commonplace, as they are widely used not only for high-performance computing in servers but also in consumer devices such as laptops and mobile devices. Besides CPUs, GPUs have presented programmers with a different approach to parallel execution. Researchers have shown that for applications that fit the execution model of GPUs, in the optimistic case, speedups of 100-300x [71], and in the pessimistic case, speedups of 2.5x [57] can be achieved between the most recent versions of GPUs compared to the latest multicore CPUs.

The main languages for developing applications for GPUs are CUDA and OpenCL. While they try to offer a more general purpose way of programming GPUs, extracting high performance from GPUs is still a daunting challenge. Difficulty in extracting massive data-level parallelism, utilizing the non-traditional memory hierarchy, complicated thread scheduling and synchronization semantics, and lack of efficient handling of control instructions are the main complications that arise while porting applications to GPUs [90].

As a result of this complexity, the computational power of graphics engines is often under-utilized or not used at all.

Although many researchers have proposed new ways to solve these problems [19, 28, 123, 120], there is still no solution for an average programmer to target GPUs. To efficiently run a sequential or parallel (for small number of cores) C/C++ application on a GPU, there are two primary methods used by developers: manually re-designing the underlying algorithm of an application for GPUs to get rid of the memory and control bottlenecks, or using a compiler to perform automatic parallelization. In most cases, it is difficult to manually identify the bottlenecks and redesign an application for the massively data-parallel execution engines of GPUs. This solution is clearly not suitable for average programmers and often expensive to apply due to the cost of re-implementing and redesigning large chunks of legacy applications. The second solution is to use compiler analysis to automatically extract enough data-parallelism from an application to gain some performance benefit from the resulting code on the target GPU. In many cases, ambiguous memory dependencies or control flow divergences in a small number of threads can negatively affect thousands of other threads on a GPU. The main problem with this approach is that the compiler analyses used for automatic parallelization are usually too conservative and fragile resulting in small or no performance gains on most commodity computer systems.

In this work, we take a different approach to this problem. Considering the amount of parallelism exposed by GPUs and their ubiquity in consumer devices, we propose co-operative speculative loop execution on GPUs and CPUs using *Paragon* for implicitly data-parallel programs written in C/C++. Paragon, using data-parallel speculation and distributed conflict detection engines carefully designed for cores in GPUs, enables pro-

grammers to transparently take advantage of GPUs for pieces of their applications that are possibly-data-parallel without manually changing the application or relying on complex compiler analyses, thus reducing the cost of migrating to GPUs. Further, the set of applications that can be mapped onto a GPU is broadened beyond loops that exclusively use arrays with affine indices. Paragon's use of cooperative execution between the GPU and CPU increases the performance of the overall system in the presence of conflicts since the CPU is not left idle while the GPU is speculatively running an application.

The idea of speculative loop execution is not a new one. Speculative parallelization has been extensively investigated in both hardware and software (see Section 4.7) in the context of multicore CPUs [102, 47, 114, 40, 63, 75, 110]. However, speculation techniques for multicore CPUs are not designed to scale to thousands of active threads and deal with the complex memory hierarchy available on GPUs. Paragon's compilation and runtime system is the first system, that we are aware of, that explores the idea of cooperative speculation by leveraging GPUs and CPUs simultaneously while using lightweight and scalable conflict detection and recovery for large numbers of data-parallel threads. In Paragon, the CPU is used to execute parts of an application that are sequential, and both the GPU and CPU are utilized for execution of possibly-parallel for-loops. The GPU and CPU both start executing their version of a possibly-parallel for-loop (sequential on the CPU, data-parallel on the GPU). The GPU executes the for-loop assuming there is no data-dependency between the iterations, but monitors all the active threads for possible dependency violations. If a dependence violation is detected, the GPU waits for the execution of the dependency on the CPU, and then resumes the remaining iterations. This approach puts otherwise idle GPUs to productive use albeit at the cost of energy efficiency.

The Paragon compilation system is divided into two parts: *static compilation for speculation* and *cooperative execution management*. The static part mainly performs loop classification and generates CUDA code for the runtime system which monitors the loops on the GPU for dependency violations. The execution management also performs light-weight one-time loop monitoring and decides which loops are more likely to benefit from executing on the GPU. These two phases together enable the execution of C/C++ loops with statistically improbable cross-iteration data dependencies on the GPU.

In summary, the main contributions of this work are:

- Static compilation and runtime systems for cooperative speculative execution on GPU/CPUs
- Lightweight runtime conflict detection on GPUs
- Low overhead rollback mechanism by using the concurrency between GPUs and CPUs

4.2 Motivation

Parallelizing an existing single-threaded application for a multi-core system is often more challenging as it may not have been developed to be easily parallelized in the first place. It will be even harder to extract the fine-grained parallelism necessary for efficient use of many core systems like GPUs with thousands of threads. Therefore, several automatic static parallelization techniques for GPUs have been proposed to exploit more parallelism [39, 13, 117, 58, 106].

However, even the best static parallelization techniques cannot parallelize programs that contain irregular dependencies that manifest infrequently, or statically-unresolvable dependencies that may not manifest during runtime at all. Removing these dependencies speculatively will dramatically improve the parallelization possibilities. This work optimistically assumes that these programs can be executed in parallel on the GPU, and relies on a runtime monitor to ensure that no dependency violation is produced.

Applications that are implicitly data-parallel but at the same time difficult to parallelize often contain array index expressions that cannot be statically analyzed. We have identified three common types of loops that demonstrate this property: *non-linear array access*, *indirect array access*, and *array access through pointers*.

Non-linear array access: If a loop accesses an array with a nonlinear function of loop's induction variables, it is hard to statically disambiguate the loop-carried dependencies. To illustrate, Figure 4.1(a) shows the `make_lattice()` function in the *milc* benchmark from SPEC2006. This function accesses the `lattice` array with the index `i`, which depends on the induction variables (`x, y, z, and t`) and the loop-independent variable `squaresize`. As shown in lines 4 to 8 of Figure 4.1(a), the index is calculated through modulo operation with loop-independent variables, which makes it difficult to disambiguate cross-iteration dependencies at the compile time. In fact, this loop may or may not have dependencies between iterations depending on `squaresize`.

Indirect array access: This type of access occurs when an array index is produced in runtime. For example, Figure 4.1(b) shows the code for forward elimination of a matrix in

```
1   for(t=0; t<nt; t++) for(z=0; z<nz; z++)
2     for(y=0; y<ny; y++) for(x=0; x<nx; x++)
3       if(node_number(x,y,z,t)==mynode()){
4         xr=x%squaresize[XUP];
5         yr=y%squaresize[YUP];
6         zr=z%squaresize[ZUP];
7         tr=t%squaresize[TUP];
8         i=xr+squaresize[XUP] *(yr+squaresize[YUP] *(zr+squaresize[ZUP
              ]*tr));
9         lattice[i].x = x;
10        lattice[i].y = y;
11        lattice[i].z = z;
12        lattice[i].t = t;
13        lattice[i].index=x+nx*(y+ny*(z+nz*t));}
```

(a)

```
1   for(i=1; i<n; i++)
2     for(j=iaL[i]; j<iaL[i+1]-1; j++)
3       x[i] = x[i] - aL[j] * x[jaL[j]];
```

(b)

```
1   void VectorAdd(int n, float *c, float *a, float *b)
2     for(int i=0; i<n; i++)
3       *(c + i) = *(a + i) + *(b + i);
```

(c)

Figure 4.1: Code examples for (a) non-linear array access, (b) indirect array access, (c) array access through pointer

compressed sparse row (CSR) format where suffix `L` denotes the array for lower triangular matrix. Forward elimination is generally used as a part of Gaussian elimination algorithm, which changes the matrix to a triangular form to solve the linear equations. CSR uses three arrays to store a sparse matrix, (1) a real array `a[1:nnz]` contains the nonzero elements of the matrix row by row, (2) an integer array `ja[1:nnz]` stores the column indices of the nonzero elements stored in `a`, and (3) an integer array `ia[1:n+1]` contains the indices to the beginning of each row in the arrays `a` and `ja`.

Like the previous example, a static compiler cannot determine whether these loops are parallelizable since the inner loop in Figure 4.1(b) accesses arrays using another array value as an index, which can be identified only at runtime. Since the inner loop is a sparse dot product of the `i`-th row of array `a` and the dense vector `x`, runtime-profiling will categorize this loop as a parallel loop.

Array access through pointers: This type of access also makes it difficult for static compilers to parallelize a loop. Figure 4.1(c) shows a function that simply adds two vectors taking pointers as parameters. If there is a possibility that the pointer `c` overlaps with either `a` or `b`, the loop cannot be parallelized. Conservative static compiler will give up parallelizing the loop if there is any chance of pointer aliasing. If the runtime behavior shows that the probability of pointer aliasing is low, it is beneficial to speculatively parallelize the loop at the runtime.

As described in these examples, loops that are not possible to parallelize at compile time must be re-investigated at runtime. For loops that have cross-iteration dependencies with low probabilities, speculatively parallelizing loops on the GPU will yield a great performance speed up.

4.3 Paragon Overview

The main goal of Paragon's execution system is to automatically extract fine-grain data parallelism from its sequential input code and generate efficient C/CUDA code to run on a heterogeneous system consisting of a CPU and GPU. However, applications with irregular or complex data-dependencies are hard or even impossible to parallelize at compile time.

To overcome this problem, Paragon detects possibly-parallel loops and runs them speculatively on the GPU. As with any speculation system, two mechanisms are required: check-pointing state to enable execution rollback and runtime dependence checking to identify miss-speculations.

Paragon utilizes a check-pointing mechanism that is tailored for GPU-enabled systems. Traditionally, at each checkpoint, before starting speculative kernel execution, the speculative execution system takes a snapshot of the architectural state. Storing copies of a few registers at transaction threads in a CPU core is relatively cheap. For GPUs, however, with thousands of threads running, naively check-pointing large register files would incur significant overhead [35]. Therefore, it is not practical to use traditional CPU check-pointing mechanisms on the GPU.

Since GPUs and CPUs have separate memory systems, there is no need for special check-pointing before launching a speculative kernel on the GPU. Paragon always keeps one version of the correct data in the CPU's memory and in case of conflict, it uses the CPU's data to recover. To reduce the overhead of recovery, Paragon uses cooperative execution. Instead of waiting for a speculative kernel to finish and run the recovery process if it is needed, Paragon runs the safe sequential version of the kernel on the CPU in parallel to the GPU version. If there was a conflict in the speculative execution on the GPU, Paragon ignores the GPU's results and waits for the safe execution to finish and uses its result to run the next kernel. On the other hand, if there was not any conflict, Paragon terminates the CPU execution after GPU kernel is finished successfully. Cooperative execution is key to achieving good performance in Paragon.

The second speculation mechanism is runtime dependence checking to identify miss-

speculations. Bulk tracking of memory dependences using signatures along with dedicated structures works well for CPUs with limited numbers of threads. However, for tracking memory accesses of thousands threads, large signatures per thread are needed. Maintaining and accessing these large signatures dramatically degrades the performance on the GPU. Also, many of these traditional conflict detection approaches need fast communication mechanism between the cores, which is not available in GPUs. Therefore, Paragon uses a distributed conflict detection mechanism that can check memory accesses of many threads in parallel. This conflict detection mechanism is done in two phases. In the first phase, Paragon updates the write-log and read-log for each memory access. Then, Paragon checks the write-log and read-log to detect any conflicts. Both of these phases are specifically designed to utilize the data-parallel power of the GPU to reduce the overhead of conflict detection.

Figure 4.2 shows an example of Paragon's execution for a program with five different code segments. Like most programs, this program starts with a sequential code. There are four loops with different characteristics in this example. *Loop1* and *Loop3* are parallel. *Loop2* is a possibly-parallel loop that has complex or data-dependent cross-iteration dependency so the compiler is unable to guarantee the safe parallel execution of this loop. Finally, *Loop4* has cross-iteration dependencies and it is statically classified as a sequential loop. Paragon launches a conflict management thread (CMT) on the CPU. The CMT is responsible for orchestrating GPU-CPU transfers, running kernels on the GPU or the CPU and managing the cooperative execution between CPU and GPU for speculative kernels. In order to run a kernel on the CPU, the CMT launches another thread called *working thread* on the CPU.

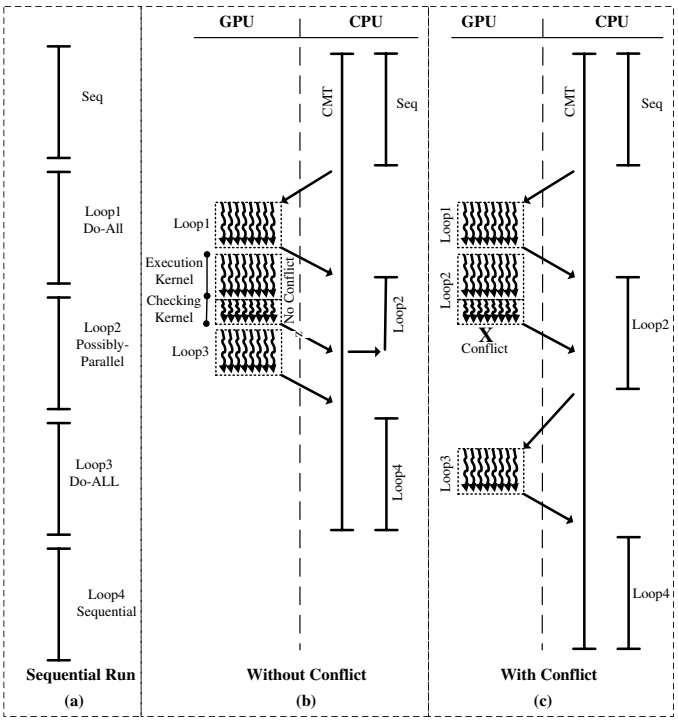


Figure 4.2: An example of running a program with Paragon. (a) sequential run (b) execution without any conflict (c) execution with conflict.

In this example, Paragon starts the execution by running the sequential part on the CPU. After running the sequential code, Paragon transfers the data needed for the execution of *Loop1* to the GPU and starts the parallel version of *Loop1*. Since *Loop2* is possibly-parallel, it should be speculatively executed on the GPU. In order to keep the correct data at this checkpoint, Paragon transfers data to the CPU. For *reentrant* loops that do not update their input arrays, using asynchronous concurrent execution, Paragon launches the CUDA kernel for *Loop2* at the same time. If *Loop2* reads and writes to the same array (i.e. non-reentrant), Paragon should wait for the data to be completely transferred to the CPU, and then launch the GPU kernel. The CPU executes the safe and sequential version of *Loop2* after it receives the data needed for execution of *Loop2* from the GPU. Paragon checks for conflicts in the speculative execution of possibly-parallel loops such as *Loop2*. The conflict

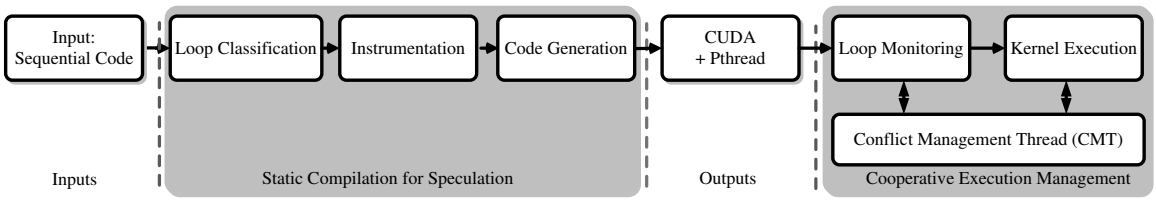


Figure 4.3: Compilation flow in Paragon.

detection process is done in parallel on the GPU with two kernels: the *execution kernel* and *checking kernel*. The execution kernel executes the loop and also marks addresses accessed by this loop. The checking kernel investigates all these addresses in parallel to detect conflicts and will set a conflict flag if it detects any dependency violation. After *Loop2* is finished, the GPU transfers the conflict flag to the CPU. Based on the conflict flag, there are two possibilities: first, if there was no conflict (Figure 4.2(b)), the CMT stops the working thread which is executing *Loop2* on the CPU and uses the GPU data to start *Loop3*. The second case is when a conflict is found in parallel execution of *Loop2* as shown in Figure 4.2(c). In this case, Paragon waits for the CPU execution to finish, then transfers data needed for the *Loop3* to the GPU. Since *Loop3* is a do-all loop, this loop will be executed only on the GPU without speculation. In order to run the sequential *Loop4*, Paragon copies the output of *Loop3* to the CPU.

Figure 4.3 shows the overall flow of Paragon's compilation and runtime system.

4.4 Compiling for Data-Parallel Speculation

One of the main challenges in Paragon is how to perform light-weight speculation and conflict detection on a massively data-parallel engine similar to a GPU. Traditional approaches for performing speculation on a multi-core system fall short in this context due

to the vast number of active threads, complex memory architecture, and communication and synchronization overheads in GPUs. Therefore, Paragon is equipped with light-weight data-parallel speculation and distributed conflict detection engines to address these issues.

Paragon focuses on loops in sequential C/C++ applications. As shown in Figure 4.3, Paragon first performs loop classification to determine which code segments are safe to parallelize. Based on this information, loop classification categorizes each loop into one of the following three categories: parallel (do-all), sequential and *possibly-parallel*. Parallel loops do not have any cross-iteration dependency and can be run in parallel on the GPU. Sequential parts, which will be run on the CPU, are parts that do not have enough parallelism to run on the GPU or have system function calls. Loops that static analysis cannot determine if they are parallel or sequential, will be in the last group called possibly-parallel loops.

Loop classification passes all this information to the code generation and instrumentation units. Since the sequential loops will be run on the CPU, Paragon generates only C code for such loops. For parallel loops, CUDA kernels will be generated. Code generation generates the CPU and GPU code with instrumentation for possibly-parallel loops. The purpose of the instrumentation is to detect any possible conflict in the execution of unsafe kernels. This distributed conflict detection mechanism has two kernels: the execution kernel and the checking kernel. These two kernels and instrumentations that need to be added will be discussed in Section 4.4.3. Before that, in the next two parts, Paragon's loop classification and code generation are explained.

4.4.1 Loop Classification

Loop classification categorizes each loop into one of the following three categories: parallel (do-all), sequential and possibly-parallel. Paragon is using static analyses and transformations such as scalar and array privatization, symbolic data dependence testing, reduction recognition and induction variable substitution to detect parallel loops [116].

Besides detection of parallel loops using static analyses, Paragon also searches for sequential loops with indirect, nonlinear or pointer accesses which may be parallel and marks them as possibly-parallel loops. The rest of the loops will be marked as sequential loops. Loop classification sends these information to the next stages which are kernel generation and instrumentation for conflict detection.

Distributing the workload evenly among thousands of threads is the main key to gaining good performance on a GPU. How to assign loop iterations to threads running on the GPU is a significant challenge for the compiler. This section illustrates how Paragon distributes iterations of the loop among GPU threads.

For single do-all loops, Paragon assigns the loop's iterations to the GPU's threads based on the trip count. If the trip count is fixed and it is smaller than the maximum number of possible threads, Paragon assigns one iteration per thread. Since our experiments show that the best number of threads per block is constant (for our GPUs, it is equal to 256), the number of threads per block (*TpB*) is always equal to 256. Therefore, the number of blocks will be equal to the trip count divided by 256. This number can be easily changed based on the GPU for which Paragon is compiling. If the trip count is more than the maximum possible number of threads, Paragon assigns more than one iteration per thread.

```
1  #pragma unroll
2  for (i=0; i<iterationsPerThread ; i++)
3   perform iteration #(i * blockDim + threadId)
```

(a)

```
1  for (i=threadId ; i<tripCount ; i+=blockDim)
2   perform iteration #(i)
```

(b)

Figure 4.4: Generated CUDA code for parallel loops with (a) Fixed trip count, (b) Variable trip count.

The number of iterations per thread (*IpT*) is always a power of two to make it easier to handle on the GPU. In this case, number of blocks (*B*) will be:

$$B = \frac{Trip\_Count}{TpB * IpT}$$

4.4.2 Kernel Generation

On the other hand, if the trip count is not known during the compile time, the compiler cannot assign a specific number of iterations to each thread. In this case, Paragon sets the number of blocks to a predefined value but this number will be tuned based on the previous runs of this kernel. As shown in Figure 4.4(b), each thread will run iterations until no iterations are left. We could use this method for loops with fixed trip counts, but our experiments show that assigning the exact iterations per thread increases the performance for these loops. If the number of threads launched is less than the number of iterations, some

threads will be idle during the kernel execution and that may degrade the performance. Another advantage is that for loops similar to the loop in Figure 4.4(a) which has a fixed trip count, the compiler can unroll the loop efficiently.

Nested do-all loops will be easy to compile if Paragon can merge those loops and generate one do-all loop. However, it is not always possible. For imperfectly-nested loops, which all assignment statements are not contained in the innermost loop, it is hard to merge nested loops. In these cases, Paragon merges nested loops as far as it is possible. Finally, two loops will be mapped to the GPU. The outer loop will be mapped to the blocks, and the inner loop will be mapped to threads of blocks. Therefore, number of blocks will be equal to the trip count of the outer loop and the number of threads per block is still equal to 256.

Reduction loop: A reduction operation generally takes a large array as its input, performs computations on it, and generates a single element as its output. This operation is usually parallelized on GPUs using a tree-based approach, such that each level in the computation tree gets its input from the previous level and produces the input for the next level. In a uniform reduction, each tree level reduces the number of elements by a fixed factor and the last level outputs one element as the final result. The only condition for using this method is that the reduction operation needs to be associative and commutative.

Paragon automatically detects reduction operations in its input using reduction variable analysis [116]. After this detection phase, the compiler replaces the reduction loop with a highly optimized kernel in its output CUDA code. Paragon uses the optimized CUDA version of the reduction kernel as described in different studies such as the work proposed by Roger et al. [87].

If there are multiple do-all loops and the innermost loop is a reduction loop, Paragon compiles them based on the trip count of the outer loops. If the trip counts of the outer loops are low, Paragon maps the outer loops to the blocks and each block executes the reduction loop. On the other hand, if the outer loops have a high number of iterations, Paragon may assign each reduction process to one thread. Therefore, iterations of the outer loops will be distributed among threads and each thread executes one instance of the innermost loop.

After generating CUDA codes for parallel and possibly-parallel loops, Paragon inserts copying instructions between the kernels. All live-in and live-out variables for all kernels are determined by Paragon at compile time. After each kernel, Paragon inserts copy instructions based on previous and next kernel's types. If both consecutive kernels are parallel or sequential there is no need to transfer data. If one of them is parallel and the other one is sequential, transferring data is needed. In cases where at least one of the kernels is possibly-parallel, Paragon adds copy instructions in both directions: from the CPU to the GPU and from the GPU to the CPU. Cooperative execution management will decide how to move the data at runtime based on the place of correct data.

4.4.3 Instrumenting for Conflict Detection

One of the main challenges for speculative execution on the GPU is designing a conflict detection mechanism that works effectively for thousands of threads. Traditional techniques used for multi-core CPUs are not well-suited for GPUs because of the non-traditional memory hierarchy , different synchronization tradeoffs on GPUs, and also the vast number of active threads available at runtime.

To deal with these constraints, we designed a distributed conflict detection mechanism

in our system. Paragon detects the dependencies between different iterations of possibly-parallel loops with two kernels: the execution kernel and the checking kernel. The first kernel executes the computations and also tags load and store addresses, and the checking kernel inspects these addresses to find a conflict. In this case, a conflict means writing to the same address by multiple threads (WAW dependency) or writing to an address by one thread and reading the same address by other threads (RAW dependency). The rest of this section describes the implementation execution and checking kernels for indirect and pointer memory accesses.

4.4.3.1 Execution Kernel Instrumentation

Indirect Memory Accesses: Execution kernel is instrumented to mark the elements that are accessed during runtime. Traditionally, Bloom filters have been used to track the dependencies between threads with very low overhead. However, using a Bloom filter for keeping track of thousands of threads at the same time requires large signatures [16]. Furthermore, accessing these signatures on the GPU needs uncoalesced accesses which leads to the performance degradation on the GPU. Therefore, instead of using a Bloom filter, Paragon stores all memory accesses in read-log and write-log arrays separately. During execution, each store to a conflict-candidate array will be marked in the corresponding write-log array and each load from that array will be marked in the corresponding read-log array.

For indirect and nonlinear array accesses, Paragon detects the arrays that can cause conflicts (*conflict-candidate arrays*), and for each of those arrays, it allocates write-log and read-log arrays. Write-log array is used to mark elements in the conflict-candidate array

that are modified during execution. Similarly, read-log array is used to mark elements in the conflict-candidate array that are read during execution.

Since the order of execution of threads on the GPU is not known a priori, any two threads which write to the same address can potentially cause a WAW conflict. This conflict may result in a wrong output. Therefore, if the number of writes to one address is more than one, there is considered a WAW dependency violation and that loop is not parallelizable. To detect WAW dependencies, Paragon utilized two approaches:

- **Atomic Method**: In this approach, Paragon uses CUDA atomic increment instructions to increment the number of writes for each store in a kernel. This method used GPU-specific atomic instruction. Based on the values stored in the read and write logs, the checking kernel can detect dependency violations. Figure 4.5 shows an example of using atomic approach. Figure 4.5(a) shows the original code and Figure 4.5(b) shows the execution kernel using atomic operation. Since each iteration modified x[i] and also reads x[jal[j]], these accesses to array x can cause conflicts.Therefore, Paragon instruments all accesses to array x. In order to prevent false positive conflict detection, Paragon just set the write_log for x[i] not read_log. As shown in this example, if Paragon statically detects that one thread accesses the same element several times, it just keep track of only one of those accesses.

- **Reduction Method**: In this approach, each thread sets the addresses of writes in the write-log array and also each thread counts the number of addresses that it modifies and stores this number in the total-writes array (*tw*) as shown in Figure 4.6(a). The checking kernel then compares these numbers to detect any possible dependency

```
1   for(i=1; i<n; i++)
2     for(j=iaL[i]; j<iaL[i+1]-1; j++)
3       x[i] = x[i] - aL[j] * x[jaL[j]];
```

(a)

```
1  Execution_Kernel()
2    initialize sharedSum to zero
3    for (i=blockIdx.x; i<n ; i+=gridDim.x){
4     sum = 0;
5     for (j=jaL[i]+threadIdx.x;j<iaL[i+1]-1;j+=blockDim.x){
6       sharedSum[j] += aL[j] * x[jaL[j]];
7       rd_log_x[jaL[j]] = 1;  /* Marking elements that are read */
8     }
9     sum = compute_sum(sharedSum);
10    if (threadIdx.x == 0){
11      x[i] -= sum;
12      AtomicInc(wr_log_x[i]);} /* Marking elements that are written */
13   }
```

(b)

```
1  Checking_Kernel()
2    tid = blockIdx.x * blockDim.x + threadIdx.x;
3    wr = wr_log_x[tid];
4    rd = rd_log_x[tid];
5    conflict = wr >> 1 /* WAW */ | (rd & wr) /* RAW */;
6    if (conflict)  conflictFlag = 1;
```

(c)

Figure 4.5: Generated CUDA code for example code in (a) with atomic approach. (b) the execution kernel code with instrumentation, (c) the checking kernel.

violations. This approach is a variation of LRPD [81] which is employed in multi-core CPUs.

In addition to the output dependency (WAW conflicts), writes to and reads from the same address by two different threads may violate the dependency constraints (RAW conflict), and the GPU's result may not be valid anymore. In this case, one read is sufficient to cause a conflict and invalidate the results. Therefore, for decreasing the overhead of main-

```
1  Execution_Kernel()
2    initialize sharedSum to zero
3    write_count = 0;
4    for (i=blockIdx.x; i<n ; i+=gridDim.x){
5     sum = 0;
6     for (j=jaL[i]+threadIdx.x;j<iaL[i+1]-1;j+=blockDim.x){
7       sharedSum[j] += aL[j] * x[jaL[j]];
8       rd_log_x[jaL[j]] = 1;  /* Marking elements that are read */
9     }
10    sum = compute_sum(sharedSum);
11    if (threadIdx.x == 0){
12      x[i] -= sum;
13      wr_log_x[i] = 1;  /* Marking elements that are written */
14      write_count_x ++;} /* Counting the number of writes */
15   }
16   tw_x[thread_id] = write_count_x;
```

(a)

```
1  Distinct_Writes_x = compute_sum(wr_log_x);
2  Total_Writes_x = compute_sum(tw_x);
3  Checking_Kernel()
4    tid = blockIdx.x * blockDim.x + threadIdx.x;
5    if (tid == 0)
6      if (Total_Writes_x != Distinct_Writes_x)
7        conflictFlag = 1;   /* WAW */
8    wr = wr_log_x[tid];
9    rd = rd_log_x[tid];
10   conflict = (rd & wr);  /* RAW */
11   if (conflict) conflictFlag = 1;
```

(b)

Figure 4.6: Generated CUDA code for example code in Figure 4.1(b) with reduction approach. (a) the execution kernel code with instrumentation, (b) the checking kernel.

taining read-log array, Paragon does not increment read-log elements atomically. Instead, it just sets the corresponding bit in the read-log for each read without using any atomic instruction as shown in Figures 4.5(b) and 4.6(a).

Pointer Memory Accesses: The execution kernel is different for loops with pointer accesses, because these loops access memory through pointers and, statically, it is not clear which array they access. In this case, for loops with pointer accesses, it is hard to detect

```
1  int Find_Array(Pointer p)
2    for (index = 0: number_of_Arrays)
3       diff =  p - GPU_Table[index].begin;
4       if (diff >=0 & diff<GPU_Table[index].size)
5         return index;     /* begin <= p < begin + size */
6    return -1;
```

(a)

```
1  bool Range_Check(Pointer Max, Pointer Min, int p_Array)
2   begin = GPU_Table[p_Array].begin;
3   size =  GPU_Table[p_Array].size;
4   if (Min >= begin & Max < size + begin)
5     return True; /* all the accesse were to the same array */
6   else
7     return False;
```

(b)

Figure 4.7: CUDA functions that Paragon uses to check pointer memory accesses. (a) Finding array that each pointer accesses, this function is called outside the main loop, (b) For each pointer, Paragon computes the minimum and maximum addresses that are accessed through that pointer. The range check function checks these maximums and minimums at the end of the execution kernel to see if all accesses were to the corresponding array or not.

which arrays may cause conflicts. Therefore, Paragon allocates one write-log array and one read-log array whose size is equal to the sum of the sizes of arrays that this loop accesses.

Each array has its own range in the write-log and read-log arrays. At the beginning of the kernel, Paragon again detects which array each pointer accesses and determines its corresponding address in the write-log and read-log arrays as shown in Figure 4.7(a). Each pointer's address is compared to the beginning and finish addresses of all arrays, which are stored in *GPU_Table*, to find the corresponding array. By doing this, Paragon is able to detect conflicts when two or more pointers access the same array. Since this process is done at the beginning of the kernel and outside of the main loop, its overhead is small for the kernels that have high trip count loops.

In order to keep track of the arrays that each kernel accesses, Paragon stores the start

address and size of arrays that are statically allocated on the CPU in a global table. A similar table is also loaded into GPU's memory. In order to find the arrays corresponding to each of the input pointers, Paragon compares the address of each of the kernel's input pointers with the start addresses and sizes of all the allocated arrays before launching the kernel. Afterwards, Paragon transfers these arrays to the GPU memory before launching the kernel. If the array that the pointer accesses is not found in the address table, the pointer is accessing dynamically allocated arrays. In this case, Paragon will run the loop sequentially on the CPU. Moreover, Paragon assumes that each pointer accesses only one array during the kernel execution. Therefore, if a pointer accesses more than one array, Paragon will detect that and raise the conflict flag.

Also, Paragon translates the pointers from the CPU's memory address space to the GPU's memory address space. In order to do this translation, Paragon subtracts the start address of the CPU array from the pointer address and adds it to the start address of the corresponding GPU array. Similarly, Paragon translates these pointers from the GPU to the CPU after the execution of the kernel.

4.4.3.2 Checking kernel implementation:

After running the execution kernel, the checking kernel will be launched. This kernel investigates the read-log and write-log arrays to find conflicts.

RAW Conflicts: To find RAW conflicts, the easiest implementation of the checking kernel is to check all addresses to detect memory addresses that are read and modified during execution as shown in Figures 4.5(c) and 4.6(b). If there is at least one write and one read

(*rd & wr*), the checking kernel will set the conflict flag.

WAW Conflicts: Detecting WAW conflicts is different for reduction and atomic methods.

- **Reduction Method**: For the reduction approach, Paragon calculates the sum of writes performed by all threads and number of distinct writes performed as follows:

$$Total\_Writes = \sum_{i=0}^{Threads} tw[i]$$
$$Distinct\_Writes = \sum_{i=0}^{Addresses} write\text{-}log[i]$$

  These two sums are computed using reduction kernels as shown in Figure 4.6(b). If *Total_Writes* is more than *Distinct_Writes*, it means that two or more threads write to the same address which is an output conflict.

- **Atomic Method**: Since the exact number of writes to each address is known in the atomic approach, there is no need to launch reduction kernels. Line 5 of Figure 4.5(c) checks the number of writes and reads of the corresponding element. If the number of writes is more than one ($wr \gg 1$) the checking kernel will set the conflict flag.

For loops with pointer accesses, Paragon runs the same checking kernel as Figure 4.5(c), but it also takes an additional step to make sure no cross-array dependency violation is happening. Paragon assumes that each pointer accesses only one array during the execution of a kernel. Therefore, for kernels with pointers that may access multiple arrays, Paragon raises the conflict flag. In order to detect these pointers, Paragon keeps track of the maximum and the minimum addresses that each pointer accesses. By computing maximum and

minimum with the max and min intrinsic functions available in CUDA, this range check process is done without any data-flow divergences. At the end of the kernel, All these maximums and minimums will be checked to see if each pointer accesses only one array or not as shown in Figure 4.7(b). If Paragon detects that a pointer accesses different arrays during the kernel execution, it stops the GPU execution and transfers the execution to the CPU.

Whenever Paragon finds a conflict, it will set the conflict flag. This flag will be sent to the CPU and, based on that, the CMT makes further decisions. These decisions will be discussed in Section 4.5.

Checking Kernel Optimizations : To optimize the checking kernel execution, Paragon applies two optimizations when it is possible:

- **Selective Checking:** Checking all addresses to find conflicts is not always necessary and may degrade the performance. Instead, it will be advantageous to just check those addresses that at least one of the execution kernel's threads writes to them. In order to check these addresses, the checking kernel should regenerate addresses that threads of the execution kernel wrote to them. For each store, Paragon starts from the index of the store instruction and traverses the data flow graph in the reverse order, to build up a slice of instructions on which the store depends, either directly or indirectly. This process stops when it reaches the input variables or the loop indices. The checking kernel executes these instructions to regenerate the store indices and investigates them to find a conflict.

- **Removing Checking Kernel:** Paragon uses an optimization for loops in which WAW dependencies are the only possible source of conflicts. In these types of loops, there is no need to launch the checking kernel because the *atomicInc* function returns the old value of the write-log element. For each write that may cause conflict, the execution kernel increments the corresponding element in the write-log array and it also checks the old value. If the old value is more than zero, it shows that another thread already wrote to the same element. In such a case, this access is marked as a conflict.

4.5 Cooperative Execution Management

The cooperative execution management unit in Paragon is a runtime component that is in charge of deciding where a loop should execute, coordinating execution of a possibly-parallel loop between the CPU and GPU, and orchestrating data transfers between the host and GPU memories. Paragon tries to increase the efficiency of speculation by utilizing both GPU and CPU at the same time. This cooperation between CPU and GPU can reduce the overhead of speculation in case of miss-speculations.

During runtime, the first invocation of possibly-parallel loops will be monitored to find any dependency between different iterations. After loop monitoring, possibly-parallel loops will be categorized as a parallel or sequential loop based on the number of dependencies found in the monitoring result. Sequential loops will be run on the CPU, and parallel loops will be run speculatively on the GPU. For speculative execution on the GPU, Paragon requires the original code to be augmented with the instructions that drive the runtime de-

pendence analysis.

The main unit of cooperative execution management is the CMT which uses monitoring information to decide which kernels should be executed on the GPU and which of them should be run on the CPU. The CMT also takes care of data movement between CPU and GPU especially in miss-speculation cases.

4.5.1 Loop Monitoring

This section describes how Paragon monitors possibly-parallel loops on the CPU to find the dependency between iterations and uses this information to improve the performance of the generated code. Paragon executes the first invocation of possibly-parallel loops on the CPU with two threads: *working thread* and *monitoring thread*. The working thread executes the loop sequentially and the monitoring thread monitors the loop in parallel to decrease the overhead of monitoring. The monitoring thread keeps track of all memory accesses. This one-time monitoring has a negligible overhead because Paragon only monitors possibly-parallel loops in parallel with the real execution.

The monitoring thread executes every instruction from the loop except stores and keeps track of the number of conflicts. After monitoring each possibly-parallel loop, if there was no conflict (Read-After-Write, Write-After-Read, and Write-After-Write), the monitoring thread marks the kernel as a parallel kernel for the CMT. If there were conflicts, the monitoring thread marks the loop as sequential. After all possibly-parallel kernels are categorized based on the monitoring results, Paragon enters the kernel execution phase. In this phase, Paragon keeps track of the number of iterations that each loop has. Based on these numbers, it will tune the number of blocks for the next execution of each kernel on

the GPU to get the best performance.

4.5.2 Conflict Management Thread (CMT)

Conflict management thread is a thread running on the CPU and its responsibility is to manage GPU-CPU transfers and run kernels speculatively on the GPU. The CMT decides which kernel should be executed on the CPU or GPU. In case of conflicts, it uses the correct data on the CPU to run the next kernel. If there was a dependency violation, the CMT does not launch the next kernel on the GPU and waits for the working thread on the CPU to finish. Based on the next kernel type, the CMT makes different decisions. If the next kernel should be run on the GPU, the CMT transfers all live-out variables to the GPU and launches the next kernel. If the next kernel is possibly-parallel, in addition to the GPU version, one version will also be run on the CPU. The last case is that the next kernel is sequential, so the CMT runs the sequential code on the CPU.

If there was no conflict in the GPU execution, the CMT sets a global variable to inform the working thread on the CPU to stop. To decrease the overhead, the working thread checks that global variable once every several iterations (10 in our experiments). This global variable works as a memory barrier to manage the data transferring between CPU and GPU. If the next kernel is parallel, the CMT will launch the next kernel. Otherwise, it transfers live-out variables and runs the next kernel on the CPU.

4.5.3 Execution Scenarios

This section explains the advantages and disadvantages of using cooperative loop execution for different possible scenarios. Figure 4.8 shows four different possibilities for an

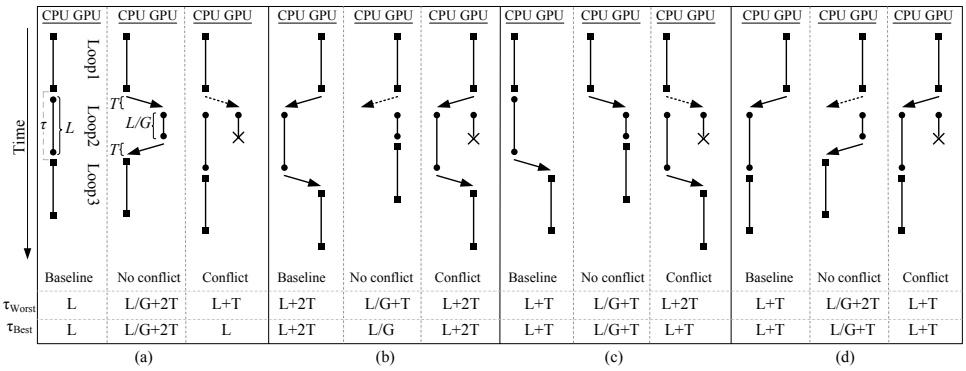


Figure 4.8: Different scenarios for Paragon execution. This figure compares the execution time (τ) of *Loop2* for different scenarios. τ is equal to the time between termination of the first loop and the start of the last loop. L is the execution time of the *Loop2* on the CPU and G is the speedup of the GPU execution of the *Loop2* with instrumentation compared to the sequential CPU execution. T is the transfer time between the CPU and the GPU.

example with three loops. In this example, the *Loop2* is a possibly-parallel loop and based on the characteristics of the *Loop1* and the *Loop3*, different scenarios may take place. In the first scenario shown in Figure 4.8(a), *Loop1* and *Loop3* are sequential loops and they will be executed on the CPU. Figure 4.8(b) shows a case when both *Loop1* and *Loop3* are do-all loops and will be run on the GPU. In Figures 4.8(c) and 4.8(d), one of these two loops is do-all and the other one is sequential.

For each of these scenarios, there are three cases: the first case is the baseline when Paragon does not run the possibly-parallel (*Loop2*) speculatively on the GPU. In all baseline cases, *Loop2* will be executed on the CPU. In the next two cases, Paragon runs the loop speculatively on the GPU. If there was no conflict in the GPU execution, Paragon uses the GPU's results and launches the next kernel. In the last case, there is a conflict in the GPU execution. Therefore, Paragon continues the CPU version of *Loop2* and uses its results to execute the last loop.

Figure 4.8 compares the execution time (τ) of *Loop2* which includes the transfer times

needed for executing this loop. In other words, τ is equal to the time between the termination of the first loop and the start of the last loop. L is the execution time of *Loop2* on the CPU and G is the speedup of the GPU execution of *Loop2* compared to the sequential run of *Loop2*. For the sake of simplicity, it is assumed that transfer time from the GPU to the CPU is equal to the transfer time from the CPU to the GPU, which is equal to T.

If the *Loop2* does not modify the inputs of the loop, there is no need to wait for the transfer operation to be over, and Paragon can perform the transfer and launch the kernel at the same time. For example in Figure 4.8(b) with no conflicts, if *Loop2* is reentrant, the GPU version can start right after the *Loop1*. However, if *Loop2* is not reentrant, Paragon transfers the data to the CPU before starting *Loop2* on the GPU to make sure that there is a correct version of the data in the CPU's memory. Dashed arrows in Figure 4.8 represent these kind of transfers, which based on the characteristics of the possibly-parallel loop, may or may not affect the execution time.

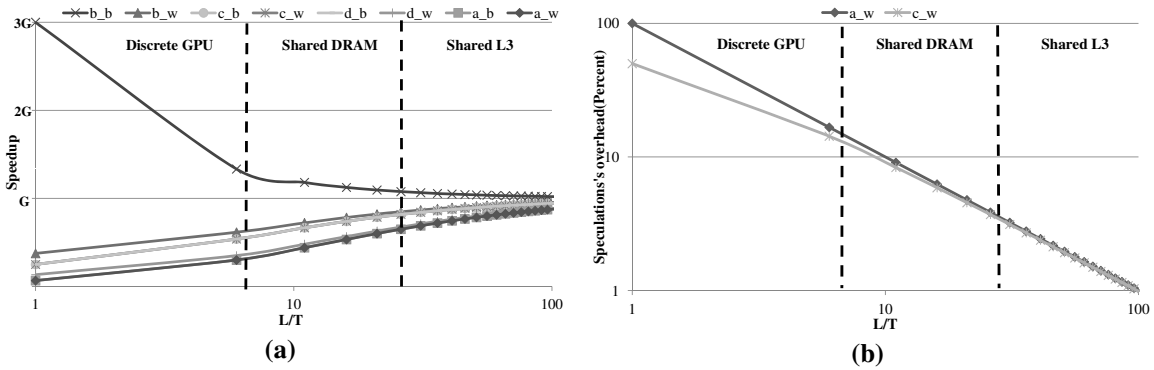


Figure 4.9: This figure shows performance and speculative overhead for different execution scenarios in Figure 4.8 . Part (a) illustrates speedup of different scenarios compared to the baseline when there is no conflict. Scenario b in the best case (b_b) has the highest speedup and scenario a in the worst case (a_w) has the lowest speedup. All the legends are sorted based on the speedup on top of the figure. Part (b) illustrates the overhead of theses scenarios compared to the baseline in case of miss-speculation.

	Input Size	Output Size	Number of loops
FDTD	4096 x 4096 matrix	4096 x 4096 matrix	6
Seidel	4096 x 4096 matrix	4096 x 4096 matrix	2
Jacobi1d	16M array	16M array	1
Jacobi2d	4096 x 4096 matrix	4096 x 4096 matrix	2
Gemm	two 4096 x 4096 matrix	4096 x 4096 matrix	3
Tmv	4096 x 4096 matrix + 4096 array	4096 array	2
Saxpy	two 32M array	32M array	1
House	two 32M array	32M array	2
Ipvec	32M array	32M array	1
Ger	two 64K array + sparse 64k x 64k matrix	sparse 64k x 64k matrix	2
Gemver	two 64K array + sparse 64k x 64k matrix	64k array	6
FWD	64K array + sparse 64k x 64k matrix	64k array	2
SOR	64K array + sparse 64k x 64k matrix	64k array	2

Table 4.1: Application specifications for Paragon evaluation

Figure 4.9a shows the speedup that Paragon can gain with speculation for different L/Ts. This Speedup is equal to $\tau_{baseline}/\tau_{noconflict}$. In fused architectures where the CPU and the GPU are integrated on the same die and share DRAM, like in AMD Fusion, or L3 cache, like in Intel Sandy Bridge, transfer time is low compared to the discrete GPUs [1]. As it can be seen in the figure, speedup for these systems will be close to GPU's gain (G) in all scenarios. The interesting point in this figure is that speedup is increasing by decreasing the transfer time except in scenario (b) with reentrant loop (the best case). The reason for that is *Loop1* and *Loop3* are both executed on the GPU. In the baseline case, Paragon should transfer the input data of *Loop2* to the CPU, execute that loop, and transfer the data back to the GPU. For speculative execution, there is no need to transfer the data. Therefore, reducing the transfer time will reduce the advantage of speculation over baseline for this scenario.

Figure 4.9b shows the overhead of miss-speculation for scenarios 4.8(a) and 4.8(c) in the worst case (non- reentrant loop) for different L/Ts. All other scenarios do not have any performance overhead in case a conflict happens. With decreasing transfer cost, this overhead decreases rapidly as shown in this figure.

[1] A typical discrete GPU has a separate memory from system memory and data transfer is done through PCIExpress

4.6 Experiments

Paragon compilation phases are implemented in the backend of the Cetus compiler [56]. We modified the C code generator in Cetus to generate CUDA code. Paragon's output codes are compiled for execution on the GPU using NVIDIA nvcc 4.0. GCC 4.4.6 is used to generate the x86 binary for execution on the host processor. The target system has an Intel i7 CPU and an NVIDIA GTX 560 GPU with 2GB GDDR5 global memory.

In order to evaluate Paragon, we compiled benchmarks with pointer and indirect memory accesses, and compared their performance with hand-optimized unsafe parallelized C code. [2] We implemented unsafe parallel versions of these benchmarks for the CPU with 2 and 4 threads and for the GPU too. Although there are many works on speculation for CPUs like CorD [110], their performance cannot be better than unsafe parallel versions. For example, CorD has 7% overhead. That's why we use unsafe code as an upper bound in our performance measurements for comparison purposes. A summary of the benchmarks characteristics is shown in Table 4.1. Also, we present a case study of accelerating a real-world application, Rayleigh quotient iteration, which will be discussed in Section 4.6.4.

Benchmarks with pointer memory accesses: We re-implemented six benchmarks from the Polybench benchmark suite [77] in C with pointers to show Paragon's performance for loops with pointers.

FDTD, Finite Difference Time Domain method, is a powerful computational technique for modeling electromagnetic space. This benchmark has three pair of different stencil

[2] Unsafe means sequential code that is optimistically parallelized and does not perform any dynamic dependence checking or synchronization/locking to ensure correct results.

loops and all these loops are highly memory intensive. The *Seidel* benchmark uses the Gauss-Seidel method which is an iterative method used to solve a linear systems of equations. Seidel is a stencil benchmark with more computation than FDTD. *Jacobi* is another stencil method to solve linear systems; We used one dimensional and two dimensional versions of this benchmark.

Gemm is a general matrix multiplication benchmark that has three nested loops. The innermost loop is a reduction loop and two outer loops are parallel. As mentioned before, Paragon decides which loops should be parallelized based on the number of iterations. Since both outer loops have high trip counts, Paragon parallelizes these loops and executes reduction sequentially inside each thread. It should be noted that this code is automatically generated for matrix multiply with pointers, so most compilers cannot detect that these loops are parallel. For the CPU version, we parallelized the outermost loop.

Tmv is a transposed matrix vector multiplication benchmark that has two nested loops. The outer one is a do-all loop and the inner one is a reduction loop. The outer loop will be mapped to thread blocks and each thread block performs the reduction in parallel.

Benchmarks with indirect memory accesses: Seven benchmarks from the sparse matrix library are used to show Paragon's performance for loops with indirect array accesses. We selected them because they have loops that cannot be analyzed by traditional compilers. For each sparse matrix benchmark, we generated matrices randomly with one percent nonzero elements.

Saxpy adds a sparse array with a dense array and writes the result in the dense array. The householder reflection benchmark, *House*, computes the reflection of a plane or hyper-

plane containing the origin. This method is widely used in linear algebra to compute QR decompositions. This benchmark consists of two parts. The first part is a reduction loop that cannot cause conflict and this loop will be compiled to CUDA without any instrumentation. The second part has a loop which is similar to *Saxpy* and it may have cross-iteration dependencies.

Ipvec is a dense matrix benchmark that shuffles all elements of the input array based on another array and puts the results in the output array. Sparse BLAS functions *Ger* and *Gemver* also have loops that can cause conflicts. Dependencies between different iterations of these loops cannot be analyzed statically so we need to use Paragon to run these loops on the GPU speculatively.

Forward Elimination with Level Scheduling, *FWD*, is another method which is used in solving linear systems. FWD's code is shown in Figure 4.1 and it has both reads and writes to the conflicted array. The next benchmark is *SOR*, a Multicolor SOR sweep in the EllPack format, and its code is similar to FWD. This benchmark has two loops: the outer loop is do-across and the inner loop is parallel, but traditional static compilers cannot easily detect that.

4.6.1 Performance

Figures 4.10b and 4.10a compare the performance of the benchmarks with pointer and indirect memory accesses to the unsafe parallel execution. The Paragon-Reduction version is the performance of the Paragon's generated code with instrumentation using reduction to check the dependencies. Paragon-Atomic uses the CUDA atomic instruction to find the conflicts on the fly. CPU 4 and 2 are unsafe parallel CPU versions without any checks

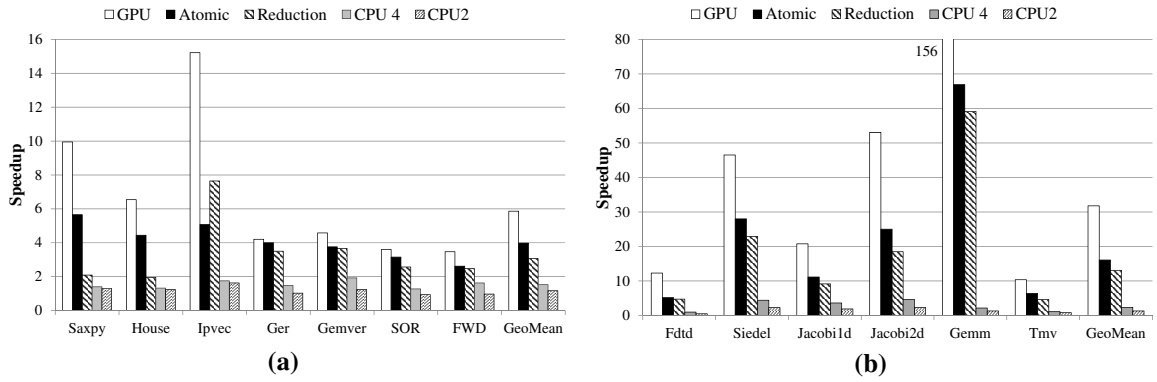


Figure 4.10: This figure shows performance of Paragon approaches compared to unsafe parallelized versions. Baseline is running the code sequentially on the CPU. Part (a) illustrates performance comparison of Paragon with unsafe parallel versions on the GPU and CPU with 4 and 2 threads for loops with pointers. Part (b) shows performance for loops with indirect accesses.

for conflicts. GPU is unsafe parallel version of applications without any instrumentations. All these different versions are compared with the sequential runs on the CPU without any threading.

Since memory accesses in benchmarks with indirect accesses are irregular, the GPU's performance is lower for these benchmarks than regular access benchmarks. In these loops, unlike the pointer loops, Paragon marks arrays that can cause conflicts. Since Paragon only checks memory accesses for these arrays, the overhead of conflict checking is lower compared to the pointer loops.

As shown in Figures 4.10b and 4.10a, for all benchmarks except *Ipvec*, Paragon-Atomic performs better than Paragon-Reduction. Since atomic instructions are slower than non-atomic memory accesses, maintaining write history of different iterations in Paragon-Reduction has less overhead than Paragon-Atomic. However, in order to find conflicts, Paragon-Reduction needs to calculate sum of two arrays: write-log and total-writes. Since these two reduction operations have a large overhead for large array sizes, the performance

of the Paragon-Atomic approach is better than Paragon-Reduction. The performance gap between these two approaches is higher for benchmarks with higher checking kernel overhead. Paragon-Reduction performs better than Paragon-Atomic for *Ipvec* because atomic memory-accesses performs poorly for the many uncoalesced memory accesses found in *Ipvec*.

For benchmarks with pointer accesses, Paragon-Atomic is 6.8x faster than CPU execution with 4 threads. For these benchmarks, Paragon is 12x faster than 2 thread execution. Also, Paragon-Atomic is 1.3x faster than Paragon-Reduction approach on average. As can be seen in Figure 4.10a, the performance of the Paragon-Atomic is 2.5x better than the unsafe parallel version of the code running on the CPU with 4 threads for benchmarks with indirect accesses. Paragon-Atomic is 3.4x faster than 2 thread execution. Also, Paragon-Atomic is 1.3x faster than Paragon-Reduction approach on average. It should be noted that in the CPU version, we assumed that there is no conflict between different iterations and, therefore, our results are pessimistic. Figure 4.10a shows that running safely on the GPU is better than running unsafely on the CPU for these data parallel loops.

On average, for benchmarks with pointer accesses and indirect array accesses, the unsafe parallel GPU versions are respectively 1.9x and 1.5x faster than Paragon-Atomic's performance. The reason is that in loops with pointers, all arrays can cause conflicts, and Paragon's approach can lead to 2x more memory accesses. These extra memory accesses can degrade the performance of memory-intensive loops. The unsafe GPU code's performance is not realistically achievable and we report this number to show the potential of our system if further optimizations and smarter runtime systems are deployed in Paragon.

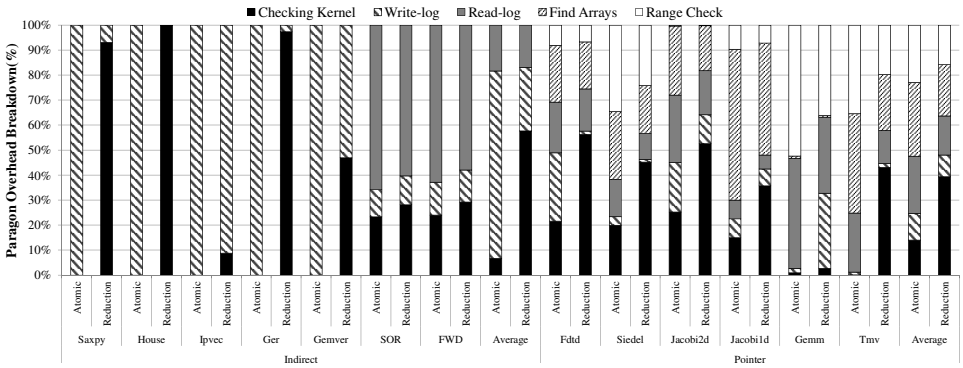


Figure 4.11: Breakdown of Paragon's overhead compared to unsafe parallel version on the GPU for loops with pointers.

4.6.2 Overhead breakdown

Figure 4.11 shows the overhead of Paragon execution compared to the unsafe GPU execution without any instrumentation. This figure also breaks down the overhead into five groups: write-log maintenance, read-log maintenance, checking kernel execution, detecting which arrays each pointer accesses, and range check of indices that each pointer accesses. Note that only benchmarks with pointer memory accesses have find-arrays or range-check overhead.

Saxpy, *House*, *Ipvec*, *Ger* and *Gemver* only write to the conflict-candidate arrays. Since the *atomicInc* function used in Paragon-Atomic approach returns the old value, there is no need to launch the checking kernel. For each write in the execution kernel, each thread atomically increments the corresponding element in the write-log and it checks the old value. If the old value is more than zero, the execution kernel sets the conflict flag. Therefore, for these benchmarks there is no checking-kernel overhead.

For all benchmarks, the write-log overhead is higher for Paragon-Atomic than Paragon-Reduction. The reason is that Paragon-Atomic uses atomic instructions to update the write-log which are not as fast as just writing to the global memory. Also, since Paragon-

Reduction needs to count the number of writes with executing two reduction kernels, the overhead of the checking kernel is higher for Paragon-Atomic approach.

SOR and *FWD* benchmarks read from an array and write to the same array with different address. Consequently, both Paragon approaches need to launch the checking kernel. Therefore, the overhead breakdown is similar for both approaches.

Benchmarks with pointer memory accesses have range-check and find-arrays overhead, too. Find-arrays overhead is negligible for benchmarks with high computation such as *Gemm* because finding arrays is done only once for each kernel. Range-check overhead is high for benchmarks with a large number of memory accesses such as *Gemm* and *Tmv* because for each memory access, Paragon needs to compare the accessed address with maximum and minimum addresses that are accessed by that pointer. Since the *Gemm* and *Tmv* have more reads than writes, the overhead of maintaining the read-log is higher than write-log's maintenance overhead. The effect of finding arrays is smaller on *Jacobi2d* compared to the same value in *Jacobi1d* because the two dimensional version has more computation and memory accesses and finding array process is done completely outside of the loop at the beginning of the kernel.

On average, for Paragon-Atomic scheme, the overhead introduced by checking kernel is 6% for indirect access and 14% for pointer access benchmarks. As mentioned before, this overhead is higher for Paragon-Reduction scheme. The checking kernel overhead for Paragon-Reduction is 57% and 39% for benchmarks with indirect and pointer memory accesses, respectively. For benchmarks with indirect memory accesses, maintaining write-log overhead is 74% for Paragon-Atomic but it is 25% for Paragon-Reduction. Since the only difference between two schemes is how they detect write-write conflicts, the effects

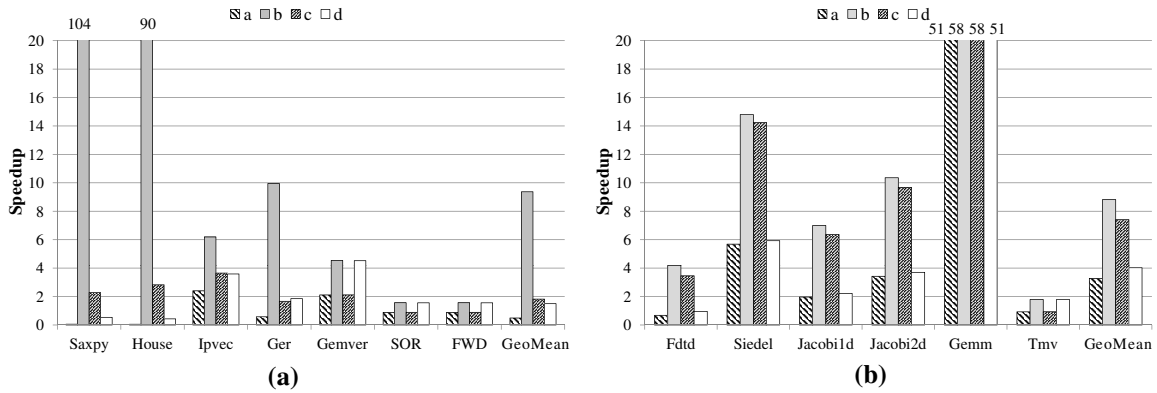


Figure 4.12: This figure shows the performance of Paragon for all four different scenarios to the sequential C code. Part (a) illustrates performance for loops with pointers. Part (b) shows performance for loops with indirect accesses.

of read-log, range-check and find-arrays are similar for both approaches.

4.6.3 Execution Scenarios Performance

This section describes the impact of transferring data between CPU and GPU for different scenarios discussed in Section 4.5.3. Figures 4.12b and 4.12a compare the performance of the benchmarks with pointer and indirect memory accesses to the sequential C code on the CPU for all four different scenarios. As discussed in Section 4.5.3, transferring overhead is high for scenario (a) because the previous and next kernel are executed on the CPU. In this case, Paragon transfers the input data to the CPU and transfers the result back. That's why performance improvement for scenario (a) is smaller than the gain reported in Figures 4.10a and 4.10a which do not consider the transferring time.

On the other hand, transferring time helps the Paragon to get better speedups for scenario (b). In this scenario, baseline transfers the data from the GPU to the CPU, run the possibly-parallel kernel, and transfer the data back to the GPU. Instead, Paragon executes

the possibly parallel kernel on the GPU and if the loop is re-entrant, there is no need to wait for transferring data. For this scenario, Paragon gets more than 8x speedup for both types of loops on average.

For scenarios (c) and (d), final performance gain is dependant on transferring time for input or output data, and whether the loop is re-entrant or not. For loops with pointer accesses, Paragon cannot decide whether the loop is reentrant. Therefore, it waits for the transfer. That's the reason that transferring overhead for scenarios (c) and (d) is higher for loops with pointer accesses than loops with indirect accesses.

4.6.4 Case study

In this section, we look into the effects of our compiler on the performance of the Rayleigh quotient benchmark. We used this benchmark to demonstrate Paragon's performance for applications with several loops where a large amount of data has to be shipped back and forth between the GPU and CPU. We also investigate the overhead of Paragon execution in the presence of conflicts in this section. A Rayleigh quotient iteration is an eigenvalue algorithm which extends the idea of the inverse iteration by using the Rayleigh quotient to obtain increasingly accurate eigenvalue estimates.

Rayleigh quotient iteration is an iterative method, that is, it must be repeated until it converges to an answer. Fortunately, very rapid convergence is guaranteed and no more than a few iterations are needed in practice. The Rayleigh quotient iteration algorithm converges cubically for symmetric matrices, given an initial vector that is sufficiently close to an eigenvector of the matrix that is being analyzed.

To solve the linear systems in lines 3 and 9 in Figure 4.13, we used biconjugate gradient

```
1 rayleigh(A,epsilon,mu,x)
2   x = x / norm(x);
3   y = (A-mu*eye(rows(A))) \ x;
4   lambda = transpose(y)*x;
5   mu = mu + 1 / lambda
6   error = norm(y-lambda*x) / norm(y)
7   while (error > epsilon){
8     x = y / norm(y);
9     y = (A-mu*eye(rows(A))) \ x;
10    lambda = transpose(y)*x;
11    mu = mu + 1 / lambda
12    error = norm(y-lambda*x) / norm(y)
13  }
```

Figure 4.13: Rayleigh quotient code

stabilized method (BiCGSTAB) which is an iterative method used for finding the numeral solution of linear systems such as $Ax=B$ for x where A is a square matrix. The whole BiCGSTAB process can be executed on the GPU.

If matrix A is a sparse matrix, computing *A-mu*eye(rows(A))* will be a possibly-parallel code. In this case, a conservative compiler will run this part on the CPU and transfer the result from the CPU to the GPU. However, Paragon speculatively runs this loop on the GPU and removes the transfer overhead. We observed that this loop is executed 5.3x faster on the GPU and if we consider the transfer time, this speedup will be increased to 7.8x. The effect of this speculation on the whole benchmark is dependant on how accurate linear systems in lines 3 and 9 should be solved.

To show the overhead of Paragon's execution in case of conflict, we added one write-write dependency to every twenty iterations of the speculative kernel. As expected, the performance impact of detecting conflict and using the CPU's data to continue the execution is negligible. Our experiments show that the overhead is less than one percent for this benchmark.

4.7 Related Work

As many-core architectures have become mainstream, there has been a large body of work, such as SUIF [115] and Polaris [17], on static compiler techniques to automatically parallelize applications to utilize thread-level-parallelism. These compilers automatically detect loops that can be parallelized using static analyses, and transform the loops for parallel execution. However, it is hard to statically decompose the application to take advantage of the growing number of processor cores [52, 51]. One of the most challenging issues in automatic parallelization is to discover loop-carried dependencies. Although various research projects on loop-dependence analysis [80] and pointer analysis [74] have tried to disambiguate dependencies between iterations, parallelism in most real applications cannot be uncovered at compile time due to irregular access patterns, complex use of pointers, and input-dependent variables.

For those applications that are hard to parallelize at compile time, thread-level speculation (TLS) is used to resolve loop-carried dependencies at *runtime*. In order to implement TLS, several extra compiler and runtime steps such as buffering memory access addresses for each thread, checking violations, and recovery procedures in case of conflicts between threads, are necessary. Software-only approaches [102, 18, 114, 27, 47, 48, 40, 63, 75, 110] implement all these steps in software. However, most existing proposals for software-only speculative runtimes target tens of cores at most [63, 75, 110]. Kim et. al. [47] targets 100 cores but even their method is not applicable to the GPU because they validate the correctness of all speculative memory accesses on a core in parallel to the loop execution on other cores. However, this parallel check is not possible on the GPU due to communication and

synchronization overheads on the GPU.

There are previous works that have focused on generating CUDA code from sequential input [39, 13, 117, 58, 106]. HiCUDA [39] is a high level directive based compiler framework for CUDA programming where programmers need to insert directives into sequential C code to define the boundaries of kernel functions. The work proposed by Baskaran et al. [13] is an automatic code transformation system that generates CUDA code from input sequential C code without annotations for affine programs. In the system developed by Wolfe [117], by using C pragma preprocessor directives, programmers help the compiler to generate efficient CUDA code. Tarditi et al. [106] proposed accelerator, in which programmers use C# and a library to write their programs and let the compiler generate efficient GPU code. The work by Leung et al. [58] proposes an extension to a Java JIT compiler that executes program on the GPU. Delite [24] is another approach which aims at simplifying the creation of performance oriented DSLs and compiling them for heterogeneous systems, including systems with GPUs. Our approach is orthogonal to these systems and can be integrated in such compilation frameworks to increase the efficiency of these systems by enabling them to run more applications on the GPU. In order to improve the performance of automatic parallelization, Paragon can take advantage of Polyhedral models [15, 78, 12] which can perform more powerful automatic parallelization.

While none of the previous works on automatic compilation for current GPUs considered speculation, there are other works [65, 30, 61] which studied the possibility of speculative execution on the GPU. Menon et al. [65] modified the GPU hardware to support voltage speculation. Gregory et al. [30] described speculative execution on multi-GPU systems exploiting multiple GPUs, but they explored the use of traditional techniques to

extract parallelism from a sequential loop in which each iteration launches a GPU kernel. This approach leveraged the possibility of speculatively partitioning several kernels on multiple GPUs. Liu et al. [61] showed the possibility of using GPUs for speculative execution using a GPU-like architecture on FPGAs. They implemented software value prediction techniques to accelerate programs with limited parallelism, and software speculation techniques which re-executes the whole loop in case of a dependency violation.

Recent works [23, 35] proposed software and hardware transactional memory systems for graphic engines. In these works each thread is a transaction and if a transaction aborts, it needs to re-execute. This re-execution of several threads among thousands of threads may lead to control divergence on the GPU, and will degrade the performance. For Paragon, each kernel is a transaction and if it aborts, Paragon uses the CPU's results instead of re-executing the kernels. There are many other works that try to improve the performance of GPUs by different approaches such as reducing the overhead of divergence [19, 28, 123], coalescing more memory accesses [123], improving inter-block communication [120], generating different kernels for different input sizes [93].

4.8 Conclusion

GPUs provide an attractive platform for accelerating parallel workloads. Due to their non-traditional execution model, developing applications for GPUs is usually very challenging. As a result, these devices are left under-utilized in many commodity systems. Several languages have emerged to solve this challenge, but past research has shown that developing applications in these languages is a difficult task because of the tedious performance

optimization cycle or inherent algorithmic characteristics of an application. Also, previous approaches of automatically generating optimized parallel code in CUDA for GPUs using complex compiler infrastructures have failed to utilize GPUs that are present in everyday computing devices.

In this work, we proposed *Paragon*: a static/dynamic compiler platform to speculatively and cooperatively run possibly-data-parallel pieces of sequential applications on GPUs and CPUs. Paragon monitors the dependencies for possibly-data-parallel loops running speculatively on the GPU and non-speculatively on the CPU using a light-weight distributed conflict detection designed specifically for GPUs, and transfers the execution to the CPU in case a conflict is detected. Paragon resumes the execution on the GPU after the CPU resolves the dependency. We looked at two classes of implicitly data-parallel applications: applications with indirect and pointer memory accesses. Our experiment show that, for applications with indirect memory accesses, Paragon achieves 2.5x on average and up to 4x speedup compared to unsafe CPU execution with 4 threads. Also, for applications with pointer memory accesses, Paragon achieves 6.8x on average and up to 30x speedup compared to unsafe CPU execution with 4 threads.

CHAPTER V

Self-Tuning Approximation

5.1 Introduction

To keep up with information growth, companies such as Microsoft, Google and Amazon are investing in larger data centers with thousands of machines equipped with multi-core processors to provide the necessary processing capability on a yearly basis. The latest industry reports show that in the next decade the amount of information will expand by a factor of 50 while the number of servers will only grow by a factor of 10 [31]. At this rate, it will become more expensive for companies to provide the compute and storage capacity required to keep pace with the growth of information. To address this issue, one promising solution is to perform approximate computations on massively data-parallel architectures, such as GPUs, and trade the accuracy of the results for computation throughput.

There are many domains where it is acceptable to use approximation techniques. In such cases some variation in the output is acceptable, and some degree of quality degradation is tolerable. Many image, audio, and video processing algorithms use approximation techniques to compress and encode multimedia data to various degrees that provide trade-

Figure 5.1: Application of image blurring filter with varying degrees of output quality. Four levels of output quality are shown: 100%, 95%, 90%, and 86%.

offs between size and correctness such as lossy compression techniques. For example, while trying to smooth an image, the exact output value of a pixel can vary. If the output quality is acceptable for the user or the quality degradation is not perceivable, approximation can be employed to improve the performance. In the machine learning domain, exact learning and inference is often computationally intractable due to the large size of input data. To mitigate this, approximate methods are widely used to learn realistic models from large data sets by trading off computation time for accuracy [50, 100]. We believe that as the amount of information continues to grow, approximation techniques will become ubiquitous to make processing such information feasible.

To illustrate this behavior more concretely, consider the two examples shown in Figures 5.1 and 5.2. Figure 5.1 shows the output of a blurring filter applied to an image with varying degrees of quality loss. The leftmost image shows the correct output, and the subsequent images to the right show the results with 5%, 10%, and 14% quality loss, respectively. For most people, it is difficult to observe any significant differences between the first three images. Therefore, a range of outputs are acceptable. However, the fourth image is noticeably distorted and would be unacceptable to many users. This illustrates that limited losses in output quality may be unnoticeable, but approximation must be controlled

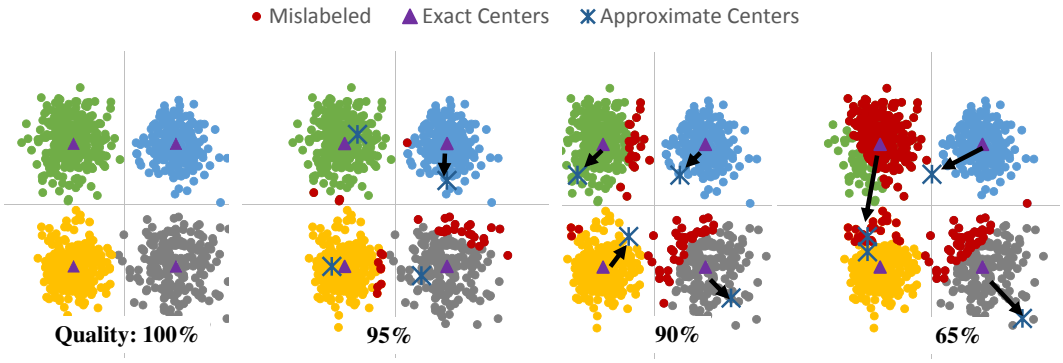


Figure 5.2: Clustering of a sample data set into four clusters using the K-means algorithm. Exact and approximate clusters' centers are also shown four levels of output quality: 100%, 95%, 90% and 65%.

and closely monitored to ensure acceptable quality of results.

Figure 5.2 provides a similar set of results but for the K-means data clustering algorithm. The leftmost image shows the correct output: each input data (dot) is clustered into one of four groups as indicated by the color of the dot with the centroid of each cluster marked by the triangle. The subsequent images show the results with 5%, 10% and 35% quality loss, respectively. In each image, the dots colored red represent the misclassified data points and the 'X's show the approximate centroids. Again, for small amounts of quality loss (3 leftmost images), the application output largely matches the correct output due to the inherent error-tolerance of data clustering. However, the rightmost image shows poor results particularly for one of the clusters (green cluster) with more than half the input data misclassified and the centroid substantially out of position.

The idea of approximate computing is not a new one and previous works have studied this topic in the context of more traditional CPUs and proposed new programming models, compiler systems, and runtime systems to manage approximation [84, 86, 2, 10, 96, 9, 32]. In this work, we instead focus on approximation for GPUs. GPUs represent affordable but

powerful compute engines that can be used for many of the domains that are amenable to approximation. However, in the context of GPUs, previous approximation techniques have two limitations: (1) the programmer is responsible for implementing and tuning most aspects of the approximation, and (2) approximation is generally not cognizant of the hardware upon which it is run. There are several common bottlenecks on GPUs that can be alleviated with approximation. These include the high cost of serialization, memory bandwidth limitations, and diminishing returns in performance as the degree of multithreading increases. Because many variables affect each of these characteristics, it is very difficult and time consuming for a programmer to manually implement and tune a kernel.

Our proposed framework for performing systematic runtime approximation on GPUs, SAGE, enables the programmer to implement a program once in CUDA, and depending on the *target output quality (TOQ)* specified for the program, trade the accuracy for performance based on the evaluation metric provided by the user. SAGE has two phases: offline compilation and runtime kernel management. During offline compilation, SAGE performs approximation optimizations on each kernel to create multiple versions with varying degrees of accuracy. At runtime, SAGE uses a greedy algorithm to tune the parameters of the approximate kernels to identify configurations with high performance and a quality that satisfies the TOQ. This approach reduces the overhead of tuning as measuring the quality and performance for all possible configurations can be expensive. Since the behavior of approximate kernels may change during runtime, SAGE periodically performs a calibration to check the output quality and performance and updates the kernel configuration accordingly.

To automatically create approximate CUDA kernels, SAGE utilizes three optimization

techniques. The first optimization targets atomic operations, which are frequently used in kernels where threads must sequentialize writes to a common variable (e.g., a histogram bucket). The atomic operation optimization selectively skips atomic operations that cause frequent collisions and thus cause poor performance as threads are sequentialized. The next optimization, data packing, reduces the number of bits needed to represent input arrays, thereby sacrificing precision to reduce the number of high-latency memory operations. The third optimization, thread fusion, eliminates some thread computations by combining adjacent threads into one and replicating the output of one of the original threads. A common theme in these optimizations is to exploit the specific microarchitectural characteristics of the GPU to achieve higher performance gains than general methods, such as ignoring a random subset of the input data or loop iterations [2], which are unaware of the underlying hardware.

In summary, the main contributions of this work are:

- The first static compilation and runtime system for automatic approximate execution on GPUs.
- Three GPU-specific approximation optimizations that are utilized to automatically generate kernels with variable accuracy.
- A greedy parameter tuning approach that is utilized to determine the tuning parameters for approximate versions.
- A dynamic calibration system that monitors the output quality during execution to maintain quality with a high degree of confidence, and takes corrective actions to stay within the bounds of target quality for each kernel.

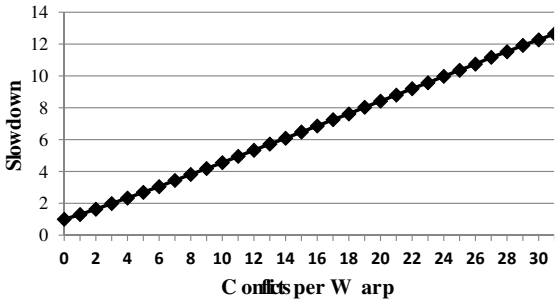
The rest of the chapter is organized as follows. Section 5.2 discusses why SAGE chooses these three approximation optimizations. Section 5.3 explains how the SAGE framework operates. Approximation optimizations used by SAGE are discussed in Section 5.4. The results of using SAGE for various benchmarks are presented in Section 5.5. Section 5.6 proposes a new way to monitor quality during runtime. Section 5.7 discusses the related work in this area and how SAGE is different from previous works. The summary and conclusion of this work is outlined in Section 5.8.

5.2 Approximation Opportunities

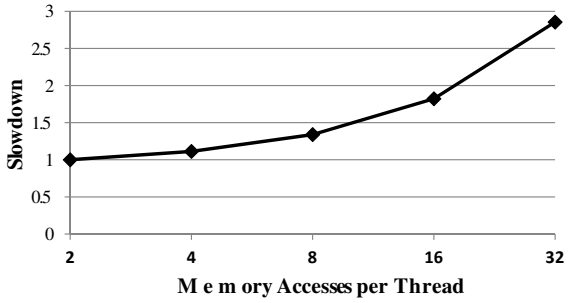
The central idea behind SAGE is to automatically detect and systematically skip or simplify processing of the operations that are particularly expensive to perform on GPUs. In order to do this, SAGE exploits three specific characteristics of GPUs.

Contention caused by atomic operations has a significant impact on performance. Atomic operations are widely used in parallel sorting and reduction operations [73] so that many different threads can update the same memory address in parallel code, as seen in the NVIDIA SDK Histogram application. As the GPU serializes accesses to the same element, performance of atomic instructions is inversely proportional to the number of threads per warp that access the same address. Figure 5.3(a) shows how the performance of *atomicAdd* decreases rapidly as the number of conflicts per warp increases for the Histogram benchmark. SAGE's first optimization improves performance by skipping atomic instructions with high contention.

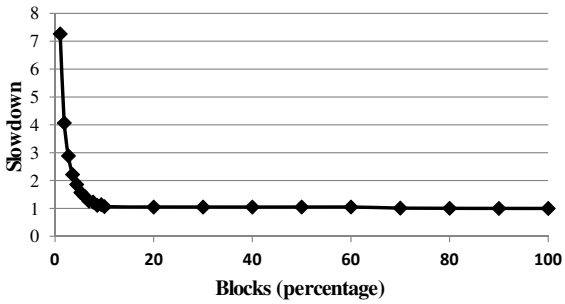
Efficiently utilizing memory bandwidth is essential to improving performance.



(a) High cost of serialization



(b) Memory bandwidth limitation



(c) Diminishing return of multithreading

Figure 5.3: Three GPU characteristics that SAGE's optimizations exploit. These experiments are performed on a NVIDIA GTX 560 GPU. (a) shows how accessing the same element by atomic instructions affects the performance for the Histogram kernel. (b) illustrates how the number of memory accesses impacts performance while the number of computational instructions per thread remains the same for a synthetic benchmark. (c) shows how the number of thread blocks impacts the performance of the Blackscholes kernel.

Considering the large number of cores on a GPU, achieving high throughput often depends on how quickly these cores can access data. Optimizing global memory bandwidth utilization is therefore an important factor in improving performance on a GPU. Figure 5.3(b) shows the impact of the number of memory accesses per thread on the total performance of a synthetic benchmark. In this example, the number of computational instructions per thread is constant and only the number of memory accesses per thread is varied. As the relative number of memory accesses increases, performance deteriorates as the memory bandwidth limitations of the GPU are exposed. The second optimization improves the memory bandwidth utilization by packing the input elements to reduce the number of memory ac-

cesses.

As long as there are enough threads, the number of threads does not significantly affect the performance. Since the number of threads running on the GPU is usually more than 10x the number of cores, fewer threads can finish the same job with similar performance. Figure 5.3(c) illustrates how changing the number of thread blocks in a kernel can affect its performance for the Blackscholes benchmark with 4M options. The baseline is the same kernel using 480 blocks. This figure shows that even 48 blocks (10% of the baseline) can utilize most of the GPU resources and achieve comparable performance. Based on these findings, SAGE's third optimization performs a low overhead thread fusion that joins together adjacent threads. After fusing threads, SAGE computes the output for one of the original, or *active*, threads and broadcasts it to the other neighboring *inactive* threads. By skipping the computation of the inactive threads, SAGE can achieve considerable performance gain.

5.3 SAGE Overview

The main goal of the SAGE framework is to trade accuracy for performance on GPUs. To achieve this goal, SAGE accepts CUDA code and a user-defined evaluation metric as inputs and automatically generates approximate kernels with varying degrees of accuracy using optimizations designed for GPUs. The SAGE framework consists of two main steps: *offline compilation* and *runtime kernel management*. Figure 5.4 shows the overall operation of the SAGE compiler framework and runtime.

The *offline compilation* phase investigates the input code and finds opportunities for

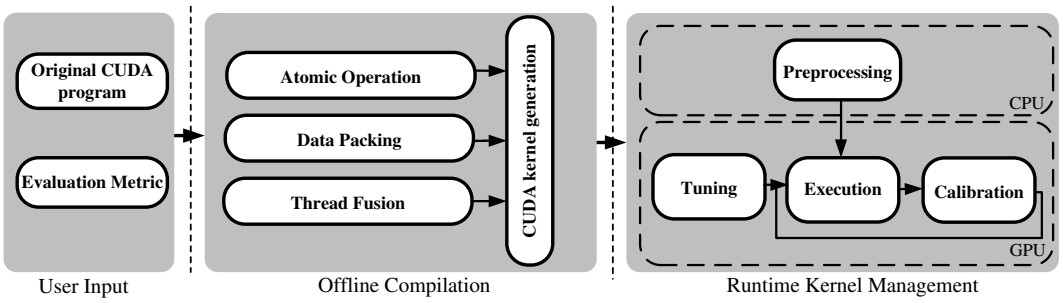


Figure 5.4: *An overview of the SAGE framework.*

trading accuracy for performance. This phase automatically generates approximate versions of CUDA kernels using three optimizations which are tailored for GPU-enabled systems. These optimizations systematically detect and skip expensive GPU operations. Each optimization has its own tuning parameters that SAGE uses to manage the performance-accuracy tradeoff. These optimizations are discussed in Section 5.4.

The *runtime management* phase dynamically selects the best approximate kernel whose output quality is better than the user-defined target output quality (TOQ). The runtime management phase consists of three parts: tuning, preprocessing and optimization calibration. Using a greedy algorithm, tuning finds the fastest kernel with better quality than the TOQ. The main goal of preprocessing is to make sure that the data needed by these approximate kernels is ready before execution. As the program behavior can change during runtime, SAGE monitors the accuracy and performance dynamically in the calibration phase. If the output quality does not meet the TOQ, calibration chooses a less aggressive approximate kernel to improve the output quality.

5.3.1 Tuning

The goal of the tuning phase is to find the fastest approximate kernel whose output quality satisfies the TOQ. Instead of searching all possible configurations, SAGE uses an online greedy tree algorithm to find reasonable approximation parameters as fast as possible to reduce the tuning overhead. Each node in the tree corresponds to an approximate kernel with specific parameters as shown in Figure 5.5. All nodes have the same number of children as the number of optimizations used by SAGE, which is two in this example. Each child node is more aggressive than its parent for that specific optimization which means that a child node has lower output quality than its parent. At the root of the tree is the unmodified, accurate version of the program. SAGE starts from the exact version and uses a *steepest-ascent hill climbing* algorithm [88] to reach the best speedup while not violating the TOQ. SAGE checks all children of each node and chooses the one with the highest speedup that satisfies the TOQ. If the two nodes have similar speedups, the node with better output quality will be chosen. This process will continue until tuning finds one of three types of nodes:

1. A node that outperforms its siblings with an output quality close to the TOQ. Tuning stops when a node has an output quality within an adjustable margin above the TOQ. This margin can be used to control the speed of tuning, and how close the output quality is to the TOQ.

2. A node whose children's output quality does not satisfy the TOQ.

3. A node whose children have less speedup.

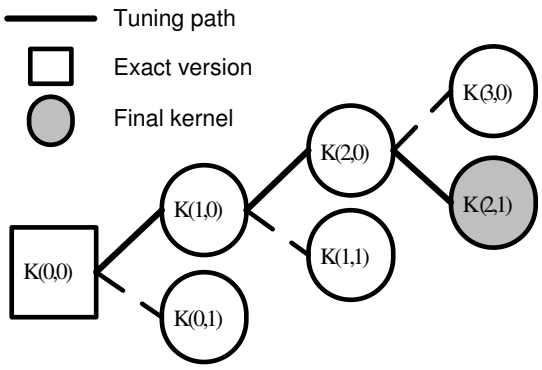


Figure 5.5: *An example of the tuning process. A node, $K(X, Y)$, is a kernel optimized using two approximation methods. X and Y are the aggressiveness of the first and second optimizations, respectively.*

In the example shown in Figure 5.5, it takes six invocations (nodes) for the tuner to find the final kernel. Once tuning completes, SAGE continues the execution by launching the kernel that the tuner found. SAGE also stores the *tuning path*, or the path from the root to the final node, and uses it in the calibration phase to choose a less aggressive node in case the output quality drops below the TOQ. If this occurs, the calibration phase traverses back along the tuning path until the output quality again satisfies the TOQ.

For all applications that we tried, the depth of the search tree for tuning is small (less than 4). Therefore, hill climbing algorithms can find the optimal solution for all of these applications and the final result is independent of the hill climbing strategy.

5.3.2 Preprocessing

Two of SAGE's optimizations need preprocessing to prepare the data necessary for the generated kernel. For the data packing optimization, preprocessing packs the input data for the next kernel. For the atomic operation optimization, the preprocessor checks input data to predict how much contention occurs during execution of atomic instructions. Details of preprocessing for these approximation optimizations are described in Section 5.4.

SAGE runs the preprocessor on the CPU in parallel to GPU execution using synchronous execution. At each time quantum, the GPU runs the selected kernel on a chunk of data while the CPU preprocesses the next chunk before transferring it to GPU memory. This way preprocessing is completely overlapped by kernel execution and its overhead is negligible.

5.3.3 Optimization Calibration

As the program behavior can change at runtime, SAGE monitors the accuracy and performance dynamically. After every *N* invocations of the kernel, the calibration unit runs both the exact and approximate kernels on the GPU to check the output quality and performance. We call *N* the calibration interval. Computing the output quality is also executed on the GPU in parallel to reduce the overhead of calibration. If the measured quality is lower than the TOQ, SAGE switches to a slower but more precise version of the program. These decisions are based on the tuning path previously described in Section 5.3.1. By backtracking along the tuning path, SAGE identifies more accurate kernels and calibrates their quality. This process will continue until the output quality satisfies the TOQ. Section 5.6 discusses different variations of the quality monitoring techniques and their impact on the overall output quality and performance.

Although checking every N^{th} invocation does not guarantee that all invocations satisfy the TOQ, checking more samples will increase our confidence that the quality of the output is acceptable. In order to compute the confidence, we assume that the prior distribution is uniform. Therefore, the posterior distribution will be $BETA(k+1, n+1-k)$, where n is the number of observed samples and k is the number of samples that satisfies the

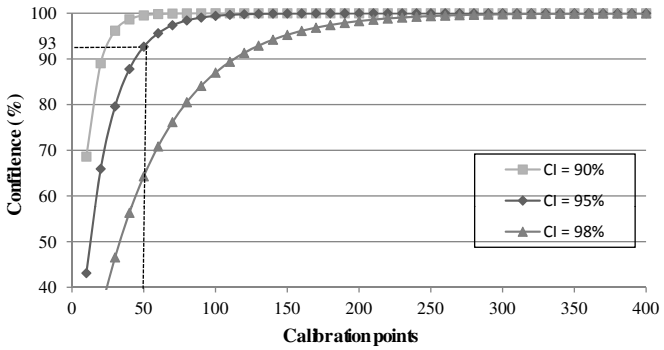


Figure 5.6: *SAGE's confidence in output quality versus the number of calibrations points for three different confidence intervals (CI).*

hypothesis [105]. In this case, the hypothesis is that the output quality is better than the TOQ. Figure 5.6 shows how confidence increases as more samples are checked for three different confidence intervals. For example, for a confidence interval equal to 95% and 50 calibration points, confidence is 93%. In other words, after checking 50 invocations, we are 93% confident that more than 95% of the invocations have better quality than the TOQ. If there is an application working on frames of a video at a rate of 33 frames per second and our calibration occurs every 10 kernel invocations, the runtime will be 99.99% confident that more than 95% of output frames will meet the TOQ in under a minute.

At the beginning of execution, there is low confidence and the runtime management system performs calibration more frequently to converge to a stable solution faster. As confidence improves, the interval between two calibration points is gradually increased so that the overhead of calibration is reduced. Every time the runtime management needs to change the selected kernel, the interval between calibrations is reset to a minimum width and the confidence is reset to zero.

5.4 Approximation Optimizations

This section details three GPU optimizations that SAGE applies to improve performance by sacrificing some accuracy: atomic operation optimization, data packing, and thread fusion.

5.4.1 Atomic Operation Optimization

Idea: An atomic operation is capable of reading, modifying, and writing a value back to memory without interference from any other thread. All threads that try to access the same location are sequentialized to assure atomicity. Clearly, as more threads access the same location, performance suffers due to serialization. However, if all threads access different locations, there is no conflict and the overhead of the atomic instruction is minimal. This optimization discards instances of atomic instructions with the highest degree of conflicts to eliminate execution segments that are predominantly serial, while keeping those with little or no conflicts. As a result of reducing serialization, SAGE can improve performance.
Detection: SAGE first finds all the atomic operations inside loops used in the input CUDA kernel and categorizes them based on their loop. For each category, SAGE generates two approximate kernels which will be discussed later.

To make sure that dropping atomic instructions does not affect the control flow of the program, SAGE checks the usage of the output array of atomic operations. It traces the control and data dependence graph to identify the branches which depend on the value of the array. If it finds any, SAGE does not apply this optimization. To detect failed convergence due to dropped atomic operations, a watchdog timer can be instrumented around the

kernel launch to prevent infinite loops.

Implementation: The atomic operation optimization performs preprocessing to predict the most popular address for the next invocation of the kernel while the GPU continues execution of the current invocation. To accomplish this, SAGE uses an approach introduced in MCUDA [103] to translate the kernel's CUDA code to C code, and then profiles this code on the CPU. To expedite preprocessing, SAGE marks the addresses as live-variables in the translated version and performs dead code elimination to remove instructions that are not used to generate addresses. In cases where the GPU modifies addresses during execution, the CPU prediction may be inaccurate. SAGE addresses this by launching a GPU kernel to find the most popular address. The overhead of preprocessing will be discussed in Section 5.5.

This optimization uses preprocessing results to find the number of conflicts per warp during runtime as follows. First, it uses the CUDA $\_\_ballot$ function[1] to determine which threads access the popular address. Next, it performs a population count on the $\_\_ballot$'s result using the $\_\_popc$ function[2] in order to find the number of threads within a warp which access the popular address.

By using runtime conflict detection, the atomic operation optimization generates two types of approximate kernels: ker_M and ker_L. ker_M skips *one* iteration that contains the *most* conflicts. ker_L skips *all* iterations except the one containing the *least* number of conflicts. Both types of kernels contain the code necessary to detect conflicts for each loop iteration at runtime. As all threads within a warp continue to execute the same iterations,

[1] $\_\_ballot()$ takes a predicate as input, and evaluates the predicate for all threads of the warp. It returns an integer whose N^{th} bit is set if and only if the predicate is non-zero for the N^{th} thread of the warp [72].

[2] $\_\_popc()$ sums the number of set bits in an integer input [72].

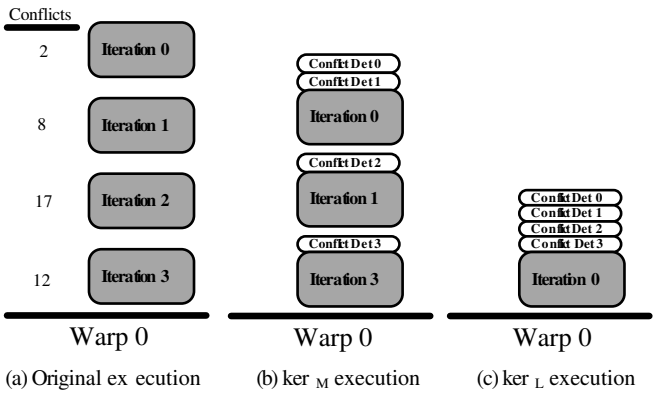


Figure 5.7: *An illustration of how atomic operation optimization reduces the number of iterations in each thread.*

no control divergence overhead is added by this optimization. Since ker_L skips more loop iterations than ker_M, ker_L is more aggressive than ker_M.

Figure 5.7 illustrates how ker_M and ker_L use conflict detection to discard atomic instructions with a large number of conflicts for sample code with four iterations per thread. In this example, each iteration contains an atomic operation. The number of conflicts in each iteration is shown on the left in Figure 5.7(a).

As shown in Figure 5.7(b), ker_M computes the number of conflicts for the first two iterations and executes the one with fewer conflicts (*Iteration 0*). ker_M continues execution by computing the number of conflicts for *Iteration 2*. Since *Iteration 2* has more conflicts than the previously skipped *Iteration 1*, ker_M executes *Iteration 1* and skips *Iteration 2*. Finally, SAGE executes *Iteration 3* because it has fewer conflicts than *Iteration 2*, which was most recently skipped. At the end of the kernel's execution, ker_M detected and skipped the loop iteration which had the most conflicts (*Iteration 2*) using SAGE's online conflict detection. It accomplished this without needing to run the loop once to gather conflict data, and a second time to apply approximation using this data.

On the other hand, ker_L performs conflict detection by running the original loop with-

out any atomic instructions. This finds the iteration with the minimum number of conflicts per warp. After conflict detection, ker_L executes the found iteration, this time running the atomic instruction. In the example in Figure 5.7(c), ker_L selected *Iteration 0* after it found that this iteration had the minimum number of conflicts (two).

Parameter Tuning: In order to tune how many atomic instructions ker_M or ker_L skip, SAGE modifies the number of blocks of the CUDA kernel. Equations 5.1, 5.2, and 5.3 show the relationship between the number of blocks and the percentage of skipped instructions for both ker_M and ker_L. Since the number of threads per block (TPB) is usually constant, if the total number of iterations (the trip count of the loop) is constant, more blocks will increase the number of threads and reduce the number of iterations per thread which can be derived from Equation 5.1. A lower number of iterations per thread (IPT) results in more dropped iterations which can be computed by Equation 5.2. However, even with the highest possible number of blocks (two iterations per thread), the dropped percentage of iterations is at most 50% for ker_M. In order to discard more than 50% of iterations, tuning switches to ker_L. With this kernel, the dropped iteration percentage can go from 50% to near 100% which can be computed using Equation 5.3. Figure 5.8 shows how this optimization affects the percentage of dropped iterations by varying the number of blocks per kernel for two different input sizes.

$$total = IPT \times TPB \times Blocks \tag{5.1}$$

$$skipped\_ker_M = \frac{1}{IPT} \times TotalIts \tag{5.2}$$

$$skipped\_ker_L = \frac{IPT - 1}{IPT} \times TotalIts \tag{5.3}$$

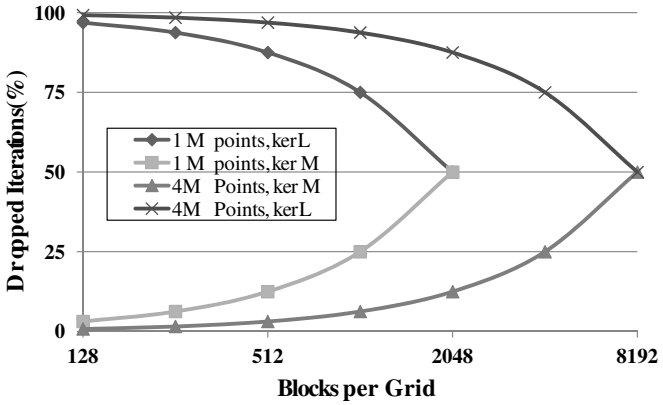


Figure 5.8: *SAGE controls the number of dropped iterations using Equations 5.1-5.3 by changing the number of blocks for one and four million data points.* ker_M *drops only one iteration per thread and* ker_L *executes only one iteration per thread. In this case, the threads per block* (TPB) *is set to 256.*

5.4.2 Data Packing Optimization

Idea: In GPUs, memory bandwidth is a critical shared resource that often throttles performance as the combined data required by all the threads often exceeds the memory system's capabilities. To overcome this limitation, the data packing optimization uses a lossy compression approach to sacrifice the accuracy of input data to lower the memory bandwidth requirements of a kernel. SAGE accomplishes this by reducing the number of memory accesses by packing the input data, thereby accessing more data with fewer requests but at the cost of more computation. This optimization packs the read-only input data in the preprocessing phase and stores it in the global memory. Each thread that accesses the global memory is required to unpack the data first. This approach is more beneficial for iterative applications which read the same array in every iteration. Most iterative machine learning applications perform the same computation on the same data repeatedly until convergence is achieved.

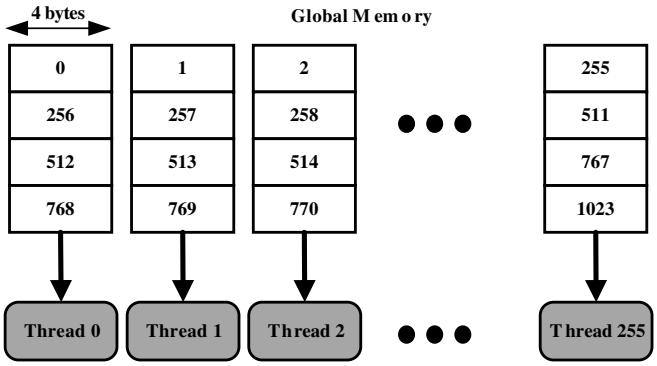
Unlike other approximate data type techniques used for CPUs which are implemented in hardware and target computations [96], this software optimization's goal is to reduce the number of memory requests with an overhead of a few additional computation instructions. All computations are done with full precision after unpacking. The added computation overhead is justifiable because, for most GPU kernels which are memory bound, it is more beneficial to optimize memory accesses at the cost of a few extra computation instructions than to optimize the computation of the kernel.

Detection: To apply this optimization, SAGE finds the read-only input arrays of kernels. As unpacking occurs in each thread, the memory access pattern must be known statically so that SAGE is able to pack the data before the kernel executes. In many applications which operate on a matrix, each thread is working on the columns/rows of the input matrix. Therefore, SAGE packs the columns/rows of the input matrix and each thread must unpack a column/row before performing the computation. For each candidate input array, SAGE generates an approximate kernel.

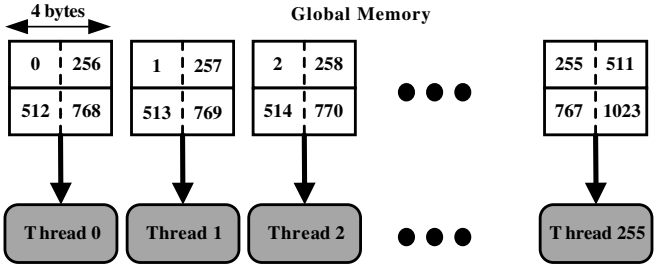
It is possible that this optimization causes a divide by zero situation as the least significant bits are truncated. However, the GPU does not throw divide by zero exceptions. Rather, it produces a large number as the result. Therefore, the program will continue without stopping and only the output quality may be affected.

Implementation: This optimization performs a preprocessing step which normalizes the data in the input matrix to the range [0,1). The scaling coefficients used are stored in the constant memory of the GPU.

After deciding the number of quantization bits (q_bits), the preprocessor packs $ratio(= \frac{number\ of\ bits\ per\ int}{q\_bits})$ number of floats into one unsigned integer. The packing process is



(a) Original memory accesses



(b) After applying data packing optimization

Figure 5.9: An example of how the data packing optimization reduces the number of global memory accesses.

done by keeping the most significant q_bits of each float and truncating the rest of the bits. Figure 5.9(a) shows the original memory accesses before applying the data packing optimization and Figure 5.9(b) illustrates an example of packing two floats in the place of one integer. When a GPU thread accesses the packed data, it reads an unsigned integer. Each thread unpacks that integer and rescales the data by using coefficients residing in the constant memory and uses the results for the computation.

Parameter Tuning: To control the accuracy of this optimization, SAGE sweeps the number of quantization bits per float from 16 to 2 to change the memory access ratio from 2 to 16.

5.4.3 Thread Fusion Optimization

Idea: The underlying idea of the thread fusion optimization is based on the assumption that outputs of adjacent threads are similar to each other. For domains such as image or video processing where neighboring pixels tend to have similar values, this assumption is often true. In this approach, SAGE executes a single thread out of every group of consecutive threads and copies its output to the other inactive threads. By doing this, most of the computations of the inactive threads are eliminated.

Detection: The thread fusion approach works for kernels with threads that do not share data. In kernels which use shared memory, all threads both read and write data to the shared memory. Therefore, by deactivating some of the threads, the output of active threads might be changed too, and this will result in an unacceptable output quality. Therefore, SAGE uses this optimization for kernels which do not use shared memory.

Implementation: In this approach, one thread computes its result and copies its output to adjacent threads. This data movement can be done through shared memory, but the overhead of sharing data is quite high due to synchronizations and shared memory latency. It also introduces control divergence overhead, and the resulting execution is too slow to make the optimization worthwhile. Instead, SAGE reduces the number of threads through fusion. Fused threads compute the output data for one of the original threads which are called *active threads*. Fused threads copy the results of the active threads to the *inactive threads*. Since this data movement occurs inside the fused thread, the transferring overhead is much less than that of using shared memory. However, in order to copy the data, fused threads should compute output addresses for inactive threads.

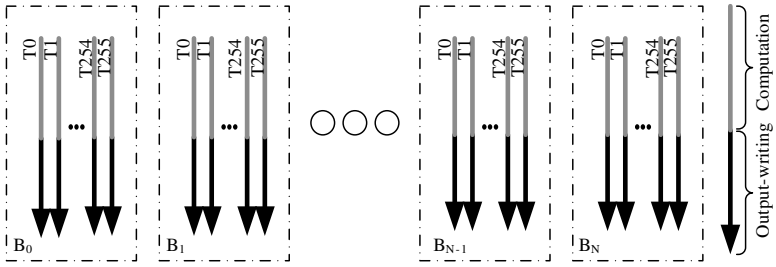
To fuse threads, SAGE translates the block ID and thread ID of the fused threads to use in the active threads. For inactive threads, SAGE walks back up the use-def chain to mark instructions that are necessary to compute the output index. Fused threads compute these instructions for all inactive threads to find which addresses they write to, and copy the active thread output values to those addresses.

There are two ways to fuse the threads: One involves reducing the number of threads per block and the other one involves fusing blocks in addition to threads and reducing the number of blocks of the kernel. Since reducing the number of threads per block results in poor resource utilization, SAGE additionally fuses blocks of each kernel. Figure 5.10(a) shows the original thread configuration before applying the optimization, and Figure 5.10(b) shows the thread configuration after fusing two adjacent threads and thread blocks. In the new configuration, each thread computes one output element and writes it to two memory locations. Although the thread fusion optimization reduces the overall computations performed by the kernel, reducing more blocks may result in poor GPU utilization as shown in Figure 5.3(c). Therefore, at some point, SAGE stops the fusion process as it eventually leads to slowdown.

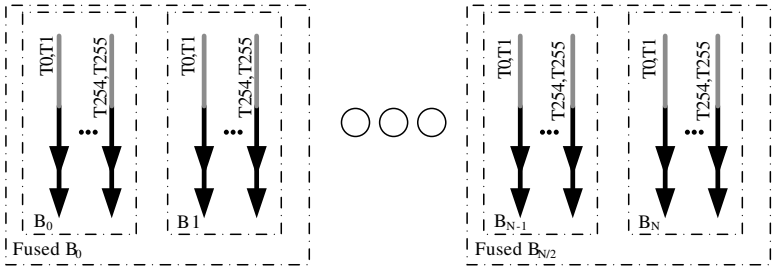
Parameter Tuning: SAGE changes the number of threads that are fused together to control performance and output accuracy.

5.5 Experimental Evaluation

In this section, we show how the optimizations in SAGE affect the execution time and accuracy of different applications. Ten applications from two domains are used: machine



(a) Original thread configuration



(b) After applying thread fusion optimization

Figure 5.10: The thread fusion optimization reduces the computation executed by this kernel by fusing two adjacent threads together and broadcasting the single output for both threads.

learning and image processing. A summary of the application characteristics is shown in Table 5.1. As each optimization targets specific, common performance bottlenecks of GPU applications, each application has usually one or two bottlenecks that SAGE optimizes as described in Table 5.1.

	Domain	Input Data	Approximation Opportunity	Evaluation Metric
K-Means	ML	1M random points, 32 features	Atomic, Packing	Mean relative difference
Naive Bayes	ML	KDD Cup [34]	Atomic	Mean relative difference
Histogram	IP	2048 x 2048 images	Atomic	Mean relative difference
SVM	ML	USPS [34]	Fusion, Packing	Mean relative difference
Fuzzy K-Means	ML	KDD Cup [34]	Packing	Mean relative difference
Means Shift	ML	KDD Cup [34]	Packing	Mean relative difference
Image Binarization	IP	2048 x 2048 images	Fusion	1 if incorrect, 0 if correct
Dynamic Range Compression	IP	2048 x 2048 images	Fusion	Mean pixel difference
Mean Filter	IP	2048 x 2048 images	Fusion	Mean pixel difference
Gaussian Smoothing	IP	2048 x 2048 images	Fusion	Mean pixel difference

Table 5.1: Application specifications (ML = Machine Learning, IP = Image Processing)

5.5.1 Applications

The *Naive Bayes Classifier* is based on the Bayesian theorem. The training process is done by counting the number of points in each cluster and the number of different feature values in each cluster. To implement Naive Bayes Classifier training, we divide the data points between the threads and each thread uses an *atomicInc* operation to compute number of points in each cluster. This implementation is based on OptiML's implementation [104].

K-Means is a commonly used clustering algorithm used for data mining. This algorithm has two steps which are iteratively executed. The first step computes the centroids of all clusters. The second step finds the nearest centroid to each point. We launch one kernel to compute all centroids. Each thread processes a chunk of data points and each block has an intermediate sum and the number of points for all clusters. Atomic instructions are used in this kernel because different threads may update the same cluster's sum or number. After launching this kernel, a reduction operation adds these intermediate sums and counters to compute centroids. As each iteration reads the same input data and changes the centroids based on that, SAGE applies both atomic operation and data packing optimizations to this benchmark.

Support Vector Machines (SVMs) are used for analyzing and recognizing patterns in the input data. The standard two class SVM takes a set of input data and for each input, predicts which class it belongs to from the two possible classes. We used Catanzaro's [22] implementation for this application. *Fuzzy K-Means* is similar to K-Means clustering except that in fuzzy clustering, each point has a degree of belonging to clusters rather than belonging completely to just one cluster. Unlike K-Means, cluster centroids are a weighted

average of all data points. Therefore, there is no need to use atomic operations.

Mean Shift Clustering is a non-parametric clustering that does not need to know the number of clusters apriori. The main idea behind this application is to shift the points toward local density points at each iteration. Points that end up in approximately the same place belong to the same cluster. *Histogram* is one of the most common kernels used in image processing applications such as histogram equalization for contrast enhancement or image segmentation. Histogram plots the number of pixels for each tonal value.

Image Binarization converts an image to a black and white image. This application is used before optical character recognition. *Dynamic Range Compression* increases the dynamic range of the image. *Mean Filter* is a smoothing filter which is used to reduce noise in images. It smoothes an image by replacing each pixel with the average intensity of its neighbors. *Gaussian Smoothing* is another smoothing filter which is used to blur images. We used the texture memory to store the input image to improve the performance of these two applications.

5.5.2 Methodology

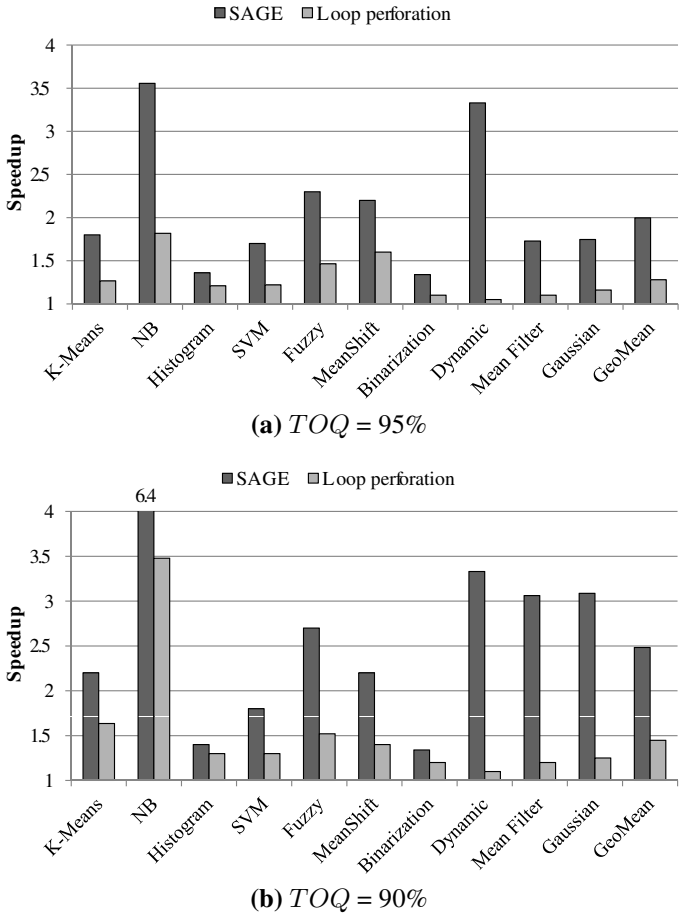
The SAGE compilation phases are implemented in the backend of the Cetus compiler [56]. We modified the C code generator in Cetus to read and generate CUDA code. SAGE's output codes are compiled for execution on the GPU using NVIDIA nvcc 4.0. GCC 4.4.6 is used to generate the x86 binary for execution on the host processor. The target system has an Intel Core i7 CPU and an NVIDIA GTX 560 GPU with 2GB GDDR5 global memory.

Output Quality: To assess the quality of each application's output, we used an application-

specific evaluation metric as shown in Table 1. Since SAGE uses an online calibration, it is limited to a computationally simple metric that minimizes the overhead. Also, the chosen metric is normalized (0%-100%) so that we can present results that can easily be compared to one another. It is very easy to modify SAGE so that different metrics, like PSNR or MSE, can be used. In all cases, we compare the output of the original application to the output of the approximate kernel.

Based on a case study by Misailovic et al. [68], the preferred quality loss range is between 0-10% for applications such as video decoding. Other works [96, 10, 33] have benchmarks that have quality loss around 10%. We also used the LIVE image quality assessment database [99, 124] to verify this threshold. Images in this database have different levels of distortion by white noise and were evaluated by 24 human subjects. The quality scale is divided into five equal portions: "Bad","Poor","Fair","Good", and "Excellent". We measured output quality of images used in the LIVE study with our evaluation metric. The results show that more than 86% of images with quality loss less than 10% were evaluated as "Good" or "Excellent" by human subjects in the LIVE study. Therefore, we used 90% as the target output quality in our experiments. We also perform our experiments with 95% target quality to show how SAGE trades off accuracy for performance per application.

Loop Perforation: SAGE optimizations are compared to another well-known and general approximation approach, loop perforation [2], which drops a set of iterations of a loop. For atomic operation and data packing optimizations, we drop every N^{th} iteration of the loops. For thread fusion optimization, dropping the N^{th} thread results in poor performance due to thread divergence. Instead, we dropped the last N iterations to avoid such divergence. We changed N to control the speedup and output quality generated by loop perforation. The



(a) TOQ = 95%

(b) TOQ = 90%

Figure 5.11: Performance for all applications approximated using SAGE compared to the loop perforation technique for two different TOQs. The results are relative to the accurate execution of each application on the GPU.

loop perforation technique is only applied to loops that are modified by SAGE to evaluate the efficiency of SAGE's optimizations. Also, it should be noted that loop perforation and data packing are orthogonal approaches and can be used together.

5.5.3 Performance Improvement

Figures 5.11(a) and 5.11(b) show the results for all applications with a TOQ of 95% and 90%, respectively. Speedup is compared to the exact execution of each program. As

computing centroids in K-Means is done by averaging over points in that cluster, ignoring some percentage of data points does not dramatically change the final result. However, since computing the centroids is not the dominant part of K-Means, the K-Means application achieves better performance by using the data packing optimization rather than the atomic operation optimization. In Section 5.5.4, we show how SAGE gets better speedup by combining these optimizations.

For the Naive Bayes classifier, computing probabilities is similar to averaging in K-Means. Since the atomic instructions take most of the execution time in this application, SAGE gets a large speedup by using this approximation optimization. On the other hand, loop perforation proportionally decreases the output quality. By decreasing the TOQ from 95% to 90%, the speedup increased from 3.6x to 6.4x.

For the Histogram application, although most of the execution time is dedicated to atomic operations, SAGE gets a smaller speedup than K-Means. The reason is that unlike K-Means, Histogram does not have similar averaging to compute the results and dropping data points directly affects the output quality. Therefore, quality loss is increased rapidly by dropping more data, and as a result, the speedup is only 1.45x for 90% TOQ.

The data packing optimization shows strong performance for memory bound applications such as fuzzy K-Means. Fuzzy K-Means is one of the more error-tolerant applications and ignoring half of the input bits does not affect the output quality significantly. SVM also shows good speedup when using 16 bits and 8 bits per float, but the quality drops below the TOQ at 4 bits per float. For MeanShift, the speedup does not increase with more packing. Therefore, the speedup of SAGE is similar for both 90% and 95% TOQ.

SAGE uses the thread fusion optimization for four applications: Dynamic range com-

pression, Image Binarization, Mean Filter, and Gaussian Smoothing. We used 2048 x 2048 pixel images to compute the quality for these applications. Dynamic Range compression and Image Binarization performances are reduced after fusing more than four threads. This is mainly because of the memory accesses and fewer numbers of blocks needed to fully utilize the GPU. Therefore, tuning stops increasing the aggressiveness of the optimization because it does not provide any further speedup. As seen in the figures, these two applications show the same speedup for both quality targets. However, for Mean Filter and Gaussian Smoothing, increasing the number of fused threads results in better performance. By decreasing the TOQ, the speedup of Mean Filter goes from 1.7x to 3.1x.

Performance-Accuracy Curve: Figure 5.12 shows a performance-accuracy curve for three sample applications to show how SAGE manages the speedup-accuracy tradeoff. Here, the atomic operation optimization is used for Naive Bayes. As the percentage of dropped iterations is increased, both quality loss and speedup are increased. Since SAGE changes the approximate kernel from $kernel_{max}$ to $kernel_{min}$, there is a performance jump between 96% and 95% output qualities. SAGE uses the data packing optimization for Fuzzy K-Means and controls the speedup by changing the number of floats that are packed. The performance-accuracy curve for Mean Filter is also shown in Figure 5.12 to show how the thread fusion optimization impacts speedup and quality. By fusing more threads, both speedup and quality loss are increased. After fusing more than eight threads, speedup decreases because of poor GPU utilization.

Error Distribution: To study the application level quality loss in more detail, Figure 5.13 shows the cumulative distribution function (CDF) of final error for each element of the application's output with a TOQ of 90%. The CDF shows the distribution of output errors

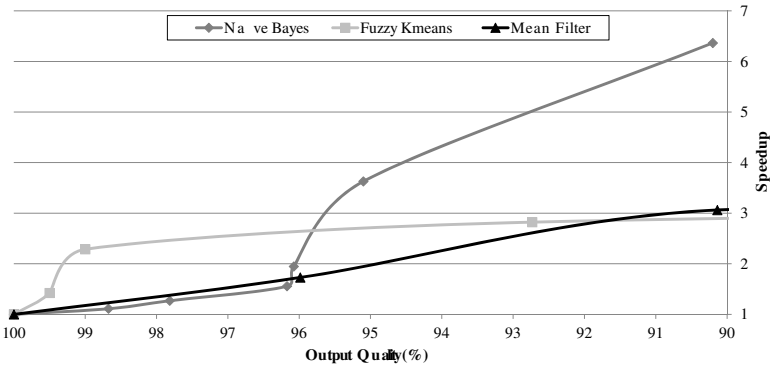


Figure 5.12: *Performance-accuracy curves for three sample applications. The atomic operation optimization is used for Naive Bayes classifier. The data packing is used for Fuzzy K-Means application and the thread Fusion is applied to Mean Filter.*

among an application's output elements and shows that only a modest number of output elements see large error. The majority (78% to 100%) of each transformed application's output elements have an error of less than 10%. As can be seen in this figure, for Image Binarization, most of the pixels have zero percent error but others have 100 percent. These pixels correspond to the edges of objects in the picture where adjacent threads outputs are dissimilar.

5.5.4 Case Studies

This section describes how SAGE uses the tuning and calibration phase to control the accuracy and performance for two example applications. In both cases, the tuning margin is set to 1% which means that tuning stops if it finds a kernel with output quality one percent better than the TOQ. In these examples, we assumed that we have enough confidence about the output quality to show how the calibration interval changes during runtime.

In the first example, we run the Gaussian Smoothing application on 100 consecutive frames of a video. Figure 5.14(a) shows the accumulative speedup and accuracy for this example. For this experiment, we assume that the TOQ is 90%. SAGE starts with the

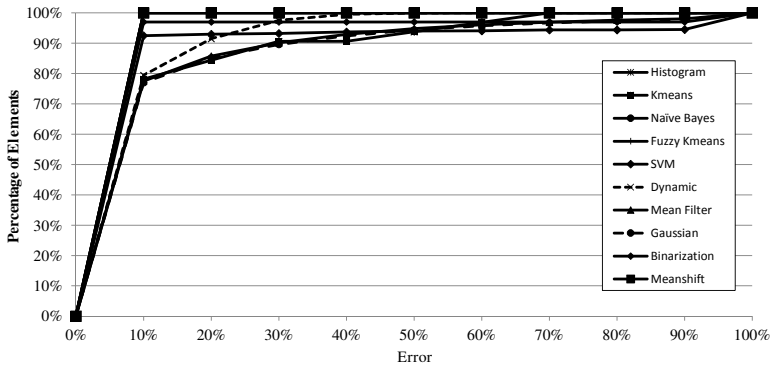


Figure 5.13: *Cumulative distribution function (CDF) of final error for each element of all application's output with the TOQ equal to 90%. The majority of output elements (more than 78%) have less than 10% error.*

exact kernel and increases the aggressiveness of the optimization until the output quality is near 90%. In order to compute the output quality for tuning, SAGE runs the approximate and exact versions one after the other. Therefore, during tuning, the output quality that the user observes is 100% and there is a temporary slowdown. As seen in Figure 5.14(a), it takes three different invocations to tune this application. After tuning, SAGE uses the final kernel that is found by tuning to continue the execution and the speedup starts to increase. The initial interval between two calibrations is set to 10 invocations. SAGE runs both versions (exact and approximate) to compute the output quality for calibration. The first calibration happens at the 10^{th} invocation after tuning and the output quality is now better than the TOQ. Therefore, there is no need to change the kernel. As it is shown in the figure, each calibration has a negative impact on the overall performance. Therefore, after each calibration where output quality is better than TOQ, SAGE increases the interval between two consecutive calibrations to reduce the calibration overhead.

The second example is K-Means. We run K-Means with 100 random input data sets of 1M points each and 32 dimensions. Each data set has 32 clusters with a random radius

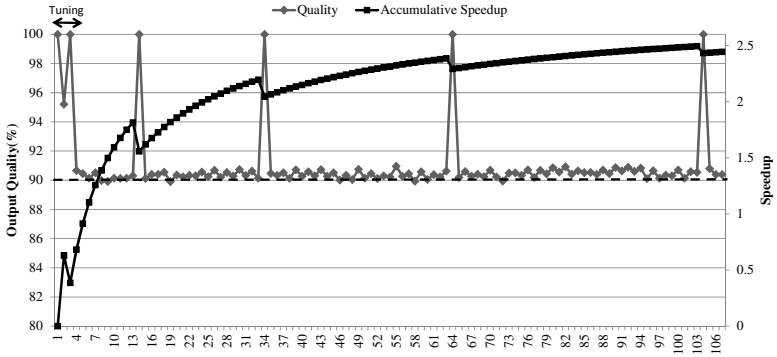
and one of the clusters is dominant. Figure 5.14(b) shows the accuracy and speedup for all invocations. K-Means starts with the exact kernel, after which SAGE increases the aggressiveness of both optimizations: atomic operation and data packing. Since data packing is more effective than atomic operation, SAGE continues tuning the kernel by packing two floats. Again, SAGE checks both child nodes and packing still provides the best speedup for the next tuning level. At the end of tuning, packing more data does not further improve performance. Therefore, SAGE increases the dropped iterations by using the atomic operation optimization. Tuning is stopped because the output quality is between 91% and 90%. As seen in the figure, it takes six different invocations to tune. The first calibration happens 10 invocations after tuning is finished. In subsequent calibrations, accuracy is in the margin of the TOQ and SAGE begins to gradually increase the interval between two calibrations.

As mentioned in Section 5.3.3, SAGE does not guarantee that output quality is always better than the TOQ. As seen in Figure 5.14, at some point, quality drops below 90%. This is because we sample invocations for calibration.

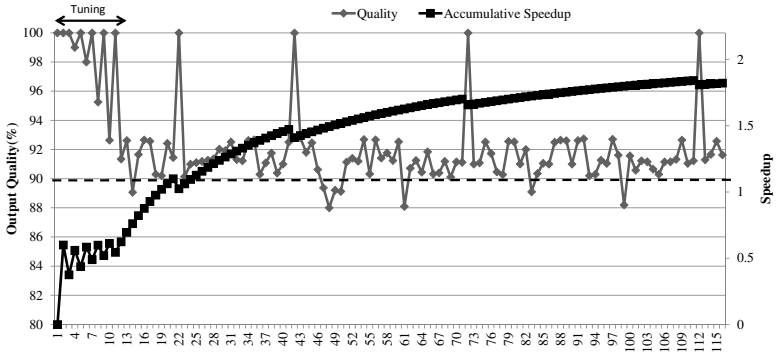
5.5.5 Runtime Overhead

Preprocessing Overhead: For the data packing optimization, SAGE transfers the packed data instead of the actual data. For the atomic optimization, preprocessing sends the most popular address as a new argument to the approximate kernel. Therefore, there is no additional transferring overhead for these two optimizations.

Since preprocessing is done on the CPU in parallel to GPU execution, the overhead is negligible for all benchmarks except K-Means. In K-Means, addresses used by atomic operations are changed dynamically during GPU execution, and SAGE finds the most pop-



(a) Gaussian Smoothing



(b) K-Means

Figure 5.14: Performance and output quality for two applications for 100 invocations with different input-sets. The horizontal dashed line represents the TOQ.

ular address on the GPU. However, the preprocessing overhead is less than 1% to find the cluster with maximum number of points.

Tuning overhead: Tuning overhead is dependent on how many invocations are needed to tune the application for the specified TOQ. When it is 90%, all our applications take three to six invocations to tune. These results are considerably better than checking all configurations. For example, searching the whole configuration for K-Means needs 20x more invocations. This gain will be larger for benchmarks with more kernels and approximation opportunities.

Calibration overhead: Calibration overhead is a function of how much speedup SAGE

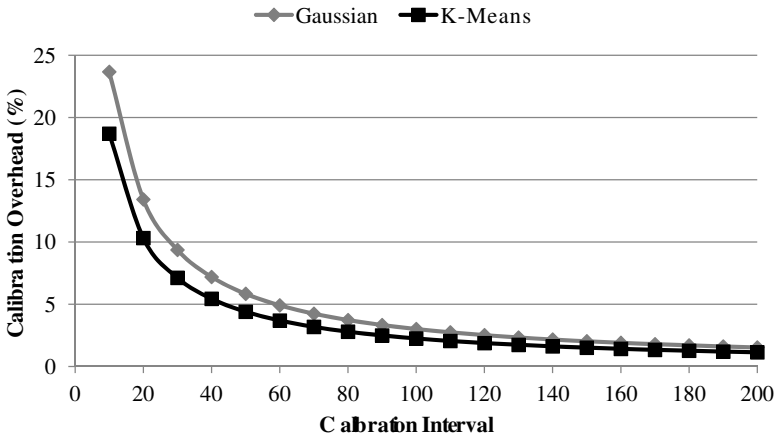


Figure 5.15: *Calibration overhead for two benchmarks for different calibration intervals.*

can achieve by using approximation and the interval between two consecutive calibration phases. Equation 5.4 shows the calibration overhead for calibration interval N. t_e is the execution time of one exact invocation of the application, G is the gain achieved by SAGE using approximation, and t_c is the time that SAGE spends to compute the output quality. Since this quality checking phase is done in parallel on the GPU, it is negligible compared to the actual execution of the application.

$$overhead_{calibration} = \frac{t_{calibration}}{t_{total}} = \frac{t_e + t_c}{N \times t_e/G + t_e + t_c} \tag{5.4}$$

Figure 5.15 illustrates the calibration overhead for the two case studies (TOQ is 90%) for different calibration intervals. This overhead is about 1% for calibration intervals more than 100. Gaussian Smoothing has a higher overhead compared to K-Means because SAGE enables a better speedup for Gaussian Smoothing. Therefore, for Gaussian Smoothing, the difference between execution time of the exact and approximate versions is larger.

5.6 CPU-GPU Collaborative Quality Monitoring

This section describes different variations of the quality monitoring and calibration technique proposed in this paper. As mentioned in Section 5.3, in order to reduce the calibration overhead, SAGE only checks every N^{th} invocation of the program. This approach works efficiently for applications that have *temporal similarity*, where two consecutive input sets shows similar output quality using the same configuration of approximation methods. One example of this type of application is applying the Gaussian filter on different frames of a video as we showed in Section 5.5.4. Since this application has temporal similarity, there is a small difference between the maximum sampled error (MSE) and the maximum real error (MRE). Figure 5.16 illustrates the difference between the MSE and the MRE for one calibration interval of Gaussian Smoothing from Figure 5.14(a). To illustrate this difference for various input sets, we applied Gaussian Smoothing to all frames of 10 different videos. Figure 5.17 shows the percent difference between MSE and MRE. As the interval between two calibrations increases, this difference increases. However, even with a calibration interval of 100 invocations, the difference between these errors is less than 10% for most of the videos.

However, this quality monitoring approach does not work efficiently for applications that do not have temporal similarity. Because in these applications, calibration samples are not representative of all invocations. Also, it is not possible to check the output quality for all invocations due to its high overhead. To solve this problem, we propose a new collaborative CPU-GPU quality monitoring technique (CCG) [94] which runs the quality monitoring code on the CPU while the GPU executes approximate data-parallel kernels.

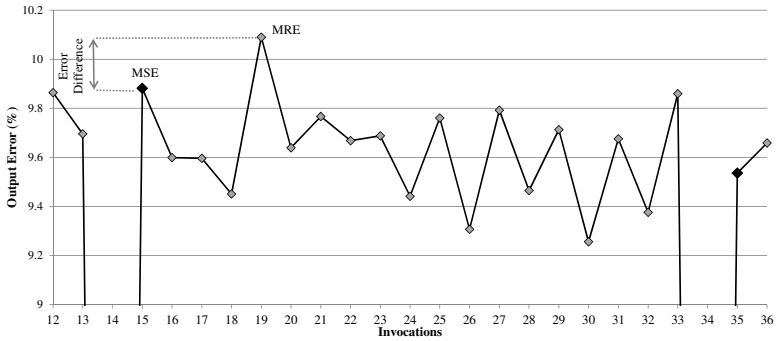


Figure 5.16: *The difference between maximum real error (MRE) and maximum sampled error (MSE) for one calibration interval of Gaussian Smoothing from Figure 5.14(a). Since SAGE runs the exact version to compare output quality, the output error is zero for invocations 14 and 34.*

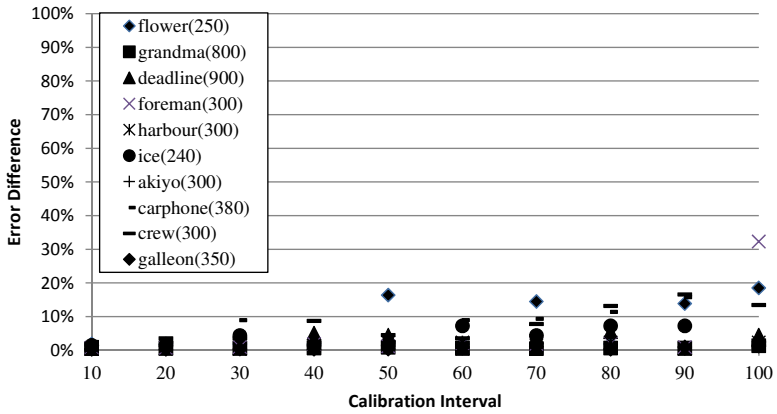


Figure 5.17: *The percent difference between maximum real error (MRE) and maximum sampled error (MSE) during calibrations for 10 videos. The number of frames is displayed next to the video's name in parentheses.*

Instead of checking every N^{th} invocation, this technique checks the quality of all invocations and runs the quality checking on the CPU in parallel to the GPU execution using synchronous execution. At each time quantum, the GPU runs the selected kernel for an invocation while the CPU computes the output quality for the next invocation. To make the overhead of quality checking almost zero, it should be completely overlapped by the kernel execution as shown in Figure 5.18.

However, since GPU's the performance is higher than the CPU's for data parallel kernels, the CPU cannot keep up with the GPU while computing the output quality for the

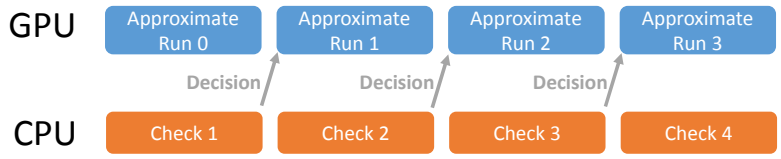


Figure 5.18: *An example of collaborative CPU-GPU quality monitoring (CCG)*

whole input data set. We solved this problem by two means: First, we parallelized the output quality computing on the CPU with four threads. Second, instead of performing full quality checking, this technique runs partial quality checking, which applies the exact and approximate kernels to a subset of input data and compares the results to estimate the overall output quality. To perform partial quality monitoring, we identified three central challenges that must be solved.

First, it is not straight-forward how to generate partial quality checking code for general applications automatically. In this paper, we wrote these codes manually. However, it is possible to use the same pattern-based compilation method as used in Paraprox [92]. Paraprox creates approximate kernels by recognizing common computation idioms found in data-parallel programs (e.g., Map, Scatter/Gather, Reduction, Scan, Stencil, and Partition) and substituting approximate implementations in their place. It is possible to generate pattern-based partial checking codes too.

Second, the subset of the input data that is used for partial quality monitoring should be chosen carefully to be a representative for the whole input data set. For now, we chose a uniformly distributed data from the input array and applied partial quality monitoring to that.

Third, the method of choosing the aggressiveness of approximation for the next kernel based on the partial output quality is important to get the best accuracy. In this work,

we checked the output for three levels of approximation for each invocation (the current level, one level more aggressive, and one level less aggressive). After computing the partial output quality for three approximate versions, the CPU will decide which one to use for the next kernel.

We illustrate by applying two image processing applications, Mosaic and Mean filter, to 1600 flower images. Since these images have different characteristics and their order is random, there is no similarity between two consecutive images. To approximate these benchmarks, we used the approximation methods described in Paraprox [92]. For the Mosaic application, loop perforation [2] was used. For Mean filter, we assumed that neighbor pixels have similar values. Based on this assumption, rather than accessing all neighbors within a tile, we access only a subset of them and assume the rest of the neighbors have the same value.

To show the calibration efficiency, we considered four alternatives to SAGE's calibration technique:

Conservative Fixed Interval (CFI): This technique checks the output quality every N^{th} invocation. The output quality is computed by running the approximate and exact versions sequentially. After that, the runtime system computes the output quality by comparing the exact and approximate outputs. Since this process has a high overhead, quality checking has a high impact on the overall performance of this technique.

At the point of quality checking, if the measured quality is lower than $TOQ - delta$, the runtime system switches to a slower but more precise version of the program. Since this technique only reduces the aggressiveness of the approximate versions, we call it conservative.

Conservative Adaptive Interval (CAI): This approach reduces the monitoring overhead of CFI by performing quality checking more frequently to converge to a stable solution faster at the beginning of kernel execution. If the output quality is higher than $TOQ - delta$, the interval between two checking points is gradually increased so that the overhead of quality checking is reduced. Every time the runtime management needs to change the selected kernel (output quality is lower than $TOQ - delta$), the interval between checking points is reset to a minimum width. Like CFI, this technique is conservative, so the overall performance might be less than ideal. This approach is the same as the calibration technique used in SAGE if $delta = 0$.

Aggressive Fixed Interval (AFI): To improve performance, unlike the aforementioned techniques, AFI looks for opportunities to reduce the output quality. At the checking point, if the output quality is higher than $TOQ + delta$, the runtime system increases the aggressiveness of the approximate versions. On the other hand, if the output quality is lower than $TOQ - delta$, the runtime system decreases the aggressiveness of the approximate versions.

Aggressive Adaptive Interval (AAI): This technique is similar to AFI but with adaptive intervals. Since this technique is both adaptive and non-conservative, its overall performance should be higher than the last three mentioned techniques.

Figure 5.19a shows the percent of images for which their output quality is not acceptable (lower than $TOQ - delta$) for all four quality monitoring techniques mentioned above. Figure 5.19b shows the overall speedup of different techniques for applying the Mosaic and Mean filter applications on all 1600 images. In these experiments, we set the TOQ to 95% and delta is 1%. As expected, the conservative techniques' (CFI and CAI) output qualities

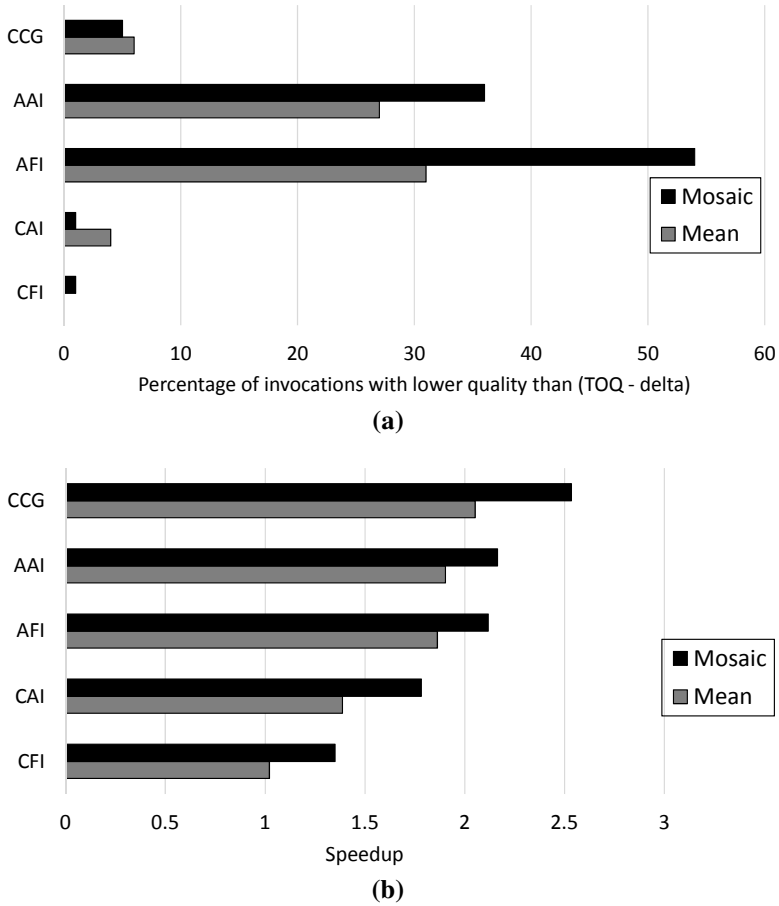


Figure 5.19: (a) Percent of images with unacceptable quality (lower than $TOQ-delta$) for different quality monitoring techniques. (b) Overall speedup of applying two programs on all 1600 images considering the calibration overhead.

are always better than those of the aggressive techniques. However, they do not show great performance. Aggressive techniques provide better speedups but the quality of more than 25% of images is not acceptable using such techniques. The reason for this that aggressive techniques are based on the assumption that it is possible to predict the quality of images by computing the quality of every N^{th} invocation. However, this assumption is not true for applications that do not have temporal similarity. As seen in these figures, the CCG outperforms other techniques mostly because its monitoring overhead is negligible and it checks

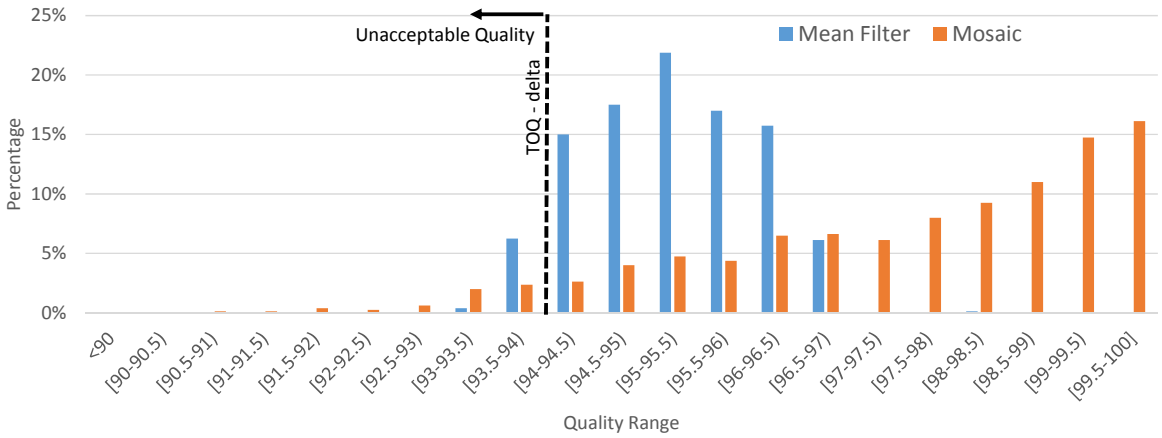
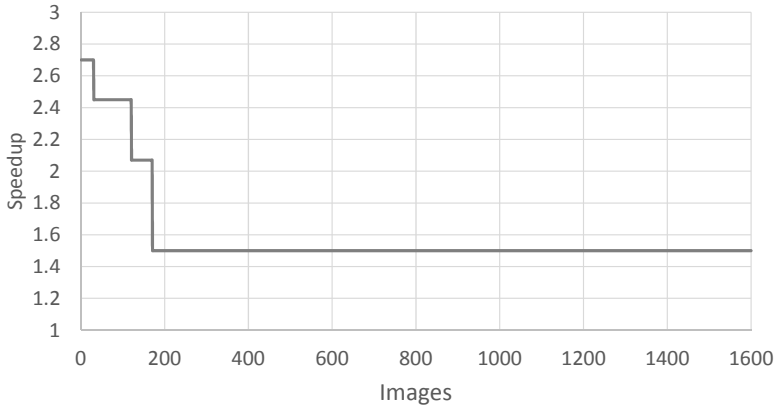


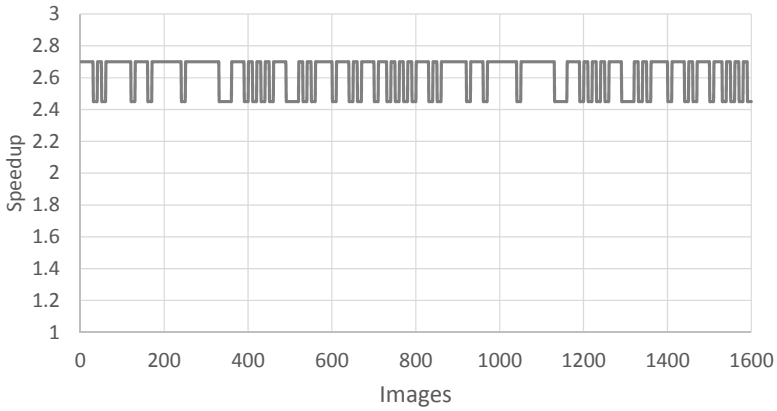
Figure 5.20: *Quality distribution of 1600 images using collaborative CPU-GPU quality monitoring (CCG)*

the quality for all different invocations. To study the output quality of unacceptable images processed by CCG, Figure 5.20 shows the quality distribution of all images. This figure shows that although quality of about 5% of images is not acceptable, even those images have quality really close to the TOQ.

To study the impact of conservative/aggressive methods on performance, Figure 5.21 shows the approximation speedup when applying the Mosaic application to 1600 images. This speedup is representative of the aggressiveness of the approximation method used for each image. Figure 5.21a shows the aggressiveness of approximation when calibrating using the conservative fixed interval (CFI) technique. Since this approach is conservative, the speedup just decreases over time and it will choose a pessimistic approximation method to make sure that fewer images have unacceptable quality. On the other hand, Figure 5.21b shows the aggressiveness of approximation for the aggressive adaptive interval (AAI). This technique switches between two most aggressive approximation methods for all of the images. Therefore, AAI provides a better performance than conservative techniques. However, using this technique, many images have unacceptable output quality as



(a) Conservative Fixed Interval



(b) Aggressive Adaptive Interval

Figure 5.21: Instantaneous speedup of applying the Mosaic application to all 1600 images using two calibration intervals (CFI and AAI). This speedup is representative of the aggressiveness of the approximation method used for each image.

shown in Figure 5.19a.

5.7 Related Work

Trading accuracy for other benefits such as improved performance or energy consumption is a well-known technique [86, 2, 101, 41, 10, 96, 9, 32, 92, 94]. Some of these techniques are software-based and can be utilized without any hardware modifications [86, 101, 41, 2, 98, 92, 94]. Agarwal et al. [2] used code perforation to improve

performance and reduce energy consumption. They perform code perforation by discarding loop iterations. Instead of skipping every N^{th} iteration or random iterations, SAGE skips iterations with the highest performance overhead which results in the same accuracy loss but better performance gain. Rinard et al. [86] terminate parallel phases as soon as there are too few remaining tasks to keep all of the processors busy. Since there are usually enough threads to utilize the GPU resources, this approach is not beneficial for GPUs. Sartori et al. [98] also use a software approach which targets control divergence that can be added to the SAGE framework. Samadi et al. [92] introduced pattern-based approximation framework, Paraprox, which detects common patterns in the input data-parallel program and applies pattern-specific approximation methods.

Green [10] is another flexible framework that developers can use to take advantages of approximation opportunities to achieve better performance or energy consumption. The Green framework requires the programmer to provide approximate kernels or to annotate their code using extensions to C and C++. In contrast to these techniques, this paper automatically generates different approximate kernels for each application. Another difference is that SAGE's framework is specially designed for GPUs with thousands of threads and its approximation techniques are specially tailored for GPU-enabled devices. Also, the process of finding the candidate kernel to execute is different from that of the Green framework. Instead of offline training, SAGE uses an online greedy tree algorithm to find the final kernel more quickly. Ansel et. al. [9] also propose language extensions to allow the programmer to mark parts of code as approximate. They use a genetic algorithm to select the best approximate version to run. Unlike these approaches, Paraprox chooses the approximation optimization based on the patterns detected in the input code and generates

approximate versions automatically for each pattern without programmer annotation. Paraprox, however, can be utilized by the runtime systems introduced in these works to optimize performance. Samadi and Mahlke [94] proposed CPU-GPU collaborative monitoring technique which predicts the quality for all kernel invocations instead of time sampling method used in SAGE.

EnerJ [96] uses type qualifiers to mark approximate variables. Using this type system, EnerJ automatically maps approximate variables to low power storage and uses low power operations to save energy. Esmaeilzadeh et al. [32] used the same approach to map approximate variables to approximate storage and operations. While these approximate data type optimizations need hardware support, our data packing optimization is applicable to current GPU architectures and does not require any hardware modification. Another work by Esmaeilzadeh [33] designs neural processing unit (NPU) accelerators to accelerate approximate programs. Li and Yeung [59] used approximate computing concept to design a light weight recovery mechanism. Relax [29] is a framework that discards the faulty computations in fault-tolerant applications. Sampson et. al. [97] show how to improve memory array lifetime using approximation.

Finally, there exists a large variety of work which maps machine learning and image processing applications to the GPU such as Support Vector Machine [22] and K-Means [60]. OptiML [104] is another approach which proposes a new domain specific language (DSL) for machine learning applications that target GPUs.

Besides approximate systems, there are a number of systems [8, 93] that support adaptive algorithm selection to evaluate and guide performance tuning. PetaBricks [8] introduces a new language and provides compiler support to select amongst multiple imple-

mentations of algorithms in order to solve a problem. Adaptic [93] is a compiler which automatically generates different kernels based on the input size for GPUs.

5.8 Conclusion

Approximate computing, where computation accuracy is traded for better performance or higher data throughput, provides an efficient mechanism for computation to keep up with exponential information growth. For several domains such as multimedia and learning algorithms, approximation is commonly used today. In this work, we proposed the SAGE framework for performing systematic runtime approximation on GPUs.

Our results demonstrate that SAGE enables the programmer to implement a program once in CUDA and, depending on the target output quality (TOQ) specified for the program, automatically trade accuracy for performance. Across ten machine learning and image processing applications, SAGE yields an average of 2.5x speedup with less than 10% quality loss compared to the accurate execution on a NVIDIA GTX 560 GPU. This paper also shows that there are GPU-specific characteristics that can be exploited to gain significant speedups compared to hardware-incognizant approximation approaches. We also discussed how SAGE controls the accuracy and performance at runtime using optimization calibration in two case studies.

CHAPTER VI

Pattern-Based Approximation

6.1 Introduction

Over the past few years, the information technology industry has experienced a massive growth in the amount of data that it collects from consumers. Analysts reported that in 2011 alone the industry gathered a staggering 1.8 zettabytes of information, and they estimate that by 2020, consumers will generate 50 times this figure [31]. Most major businesses that host such large-scale data-intensive applications, including Google, Amazon, and Microsoft, frequently invest in new, larger data centers containing thousands of multi-core servers. However, it seems that such investments in new hardware alone may not translate to the computation capability required to keep up with the deluge of data. Rather, it may be necessary to consider using alternative programming models that exploit the data parallel computing abilities of existing servers in order to address this problem. This work focuses on applying one such model, approximate computing, where the accuracy of results is traded off for computation speed, to solve the problem of processing big data.

Approximation is applicable in domains where some degree of variation or error can

be tolerated in the result of computation. For domains where some loss of accuracy during computation may cause catastrophic failure, e.g. cryptography, approximation should not be applied. However, there are many important domains where approximation can greatly improve application performance, including multimedia processing, machine learning, data analysis, and gaming. Video processing algorithms are prime candidates for approximation as occasional variation in results do not cause the failure of their overall operation. For example, a consumer using a mobile device can tolerate occasional dropped frames or a small loss in resolution during video playback, especially when this allows video playback to occur seamlessly. Machine learning and data analysis applications also provide opportunities to exploit approximation to improve performance, particularly when such programs are operating on massive data sets. In this situation, processing the entire dataset may be infeasible, but by sampling the input data, programs in these domains can produce representative results in a reasonable amount of time.

Improving performance by applying approximation has been identified as an important goal by prior works [84, 86, 2, 10, 96, 9, 32, 33]. These works have studied this topic and proposed new programming models, compiler systems, and runtime systems to systematically manage approximation. However, these approaches have three critical limitations. We categorize the prior works based on these limitations:

- **Programmer-based [10, 9]:** In these systems, the programmer must write different approximate versions of a program and a runtime system decides which version to run. Although the programmer may best understand how his code works, writing different versions of the same program with varying levels of approximation is neither

easy nor practical to be applied generally.

- **Hardware-based [96, 32, 33]:** These approaches introduce hardware modifications such as imprecise arithmetic units, register files, or accelerators. Although these systems work for general algorithms, they cannot be readily utilized without manufacturing new hardware. Furthermore, having both exact and approximate versions of the same hardware increases the hardware design complexity and the difficulty of validating and verifying such hardware.

- **Software-based [84, 86, 2, 95]:** Previous software-based approximation techniques do not face the problems of the other two categories as they (a) remove the burden of writing several versions of the program from the programmer, and (b) can be used with existing, commodity systems. However, with past approaches, *one solution does not fit all applications*. Each of these solutions works only for a small set of applications. They either cannot achieve a desired amount of performance improvement or generate unacceptable computation errors for applications that they were not explicitly built to handle.

To address these issues, this work proposes a software framework called *Paraprox*. Paraprox identifies common patterns found in data-parallel programs and uses a custom-designed approximation technique for each detected pattern. Paraprox enables the programmer to write software once and run it on a variety of modern processors, without manually tuning code for different hardware targets. It is applicable to a wide range of applications as it determines the proper approximation optimizations that can be applied to each input program. Because Paraprox does not apply a single solution to all programs, it

overcomes the aforementioned limitation of prior software-based approaches.

In this work, we identify different patterns commonly found in data parallel workloads and we propose a specialized approximation optimization for each pattern. We closely study data parallel programs because they are well-fitted for execution on prevalent multi-core architectures such as CPUs and GPUs. Paraprox is capable of targeting any data parallel architecture, provided that the underlying runtime supports such hardware.

Overall, Paraprox enables the programmer to implement a kernel once using the OpenCL or CUDA data parallel languages and, depending on the *target output quality (TOQ)* specified for the kernel, tradeoff accuracy for performance. To control the efficiency, accuracy, and performance of the system, each optimization allows some variables to be dynamically varied. After Paraprox generates the approximate kernels, a runtime system tunes the aforementioned variables to get the best performance possible while meeting the constraints of the TOQ.

To automatically create approximate kernels, Paraprox utilizes four optimization techniques which target six data parallel patterns: Map, Scatter/Gather, Reduction, Scan, Stencil, and Partition. Paraprox applies approximate memoization to map and scatter/gather patterns where computations are replaced by memory accesses. For reduction patterns, Paraprox uses sampling plus adjustment to compute the output by only computing the reduction of a subset of the data. The stencil & partition approximation algorithm is based on the assumption that adjacent locations in an input array are typically similar in value for such patterns. Therefore, Paraprox accesses a subset of values in the input array and replicates that subset to construct an approximate version of the array. For scan patterns, Paraprox only performs the scan operation on a subset of the input array and uses the results

to predict the results for the rest of the array.

The specific contributions of this work are as follows:

- Pattern based compilation system for approximate execution.
- Automatic detection of data parallel patterns in OpenCL and CUDA kernels.
- Four pattern-specific approximation optimizations which are specifically designed for six common data parallel computation patterns.
- The ability to control performance and accuracy tradeoffs for each optimization at runtime using dynamic tuning parameters.

The rest of the chapter is organized as follows. Section 6.2 explains how the Paraprox framework operates. Approximate optimizations used by Paraprox are discussed in Section 6.3. The results of using Paraprox for various benchmarks and architectures are presented in Section 6.4. Limitations of Paraprox's framework are discussed in Section 6.5. Section 6.6 discusses the related work in this area and how Paraprox is different from previous work. Section 6.7 concludes this chapter and summarizes its contributions and findings.

6.2 Paraprox Overview

In order to generate approximate programs, Paraprox must detect data parallel patterns for optimization. As shown in Figure 6.1, these patterns have distinct characteristics that require specialized optimizations in order to create fast, approximate versions. In the following list, we describe the characteristics of the six patterns that Paraprox targets. These

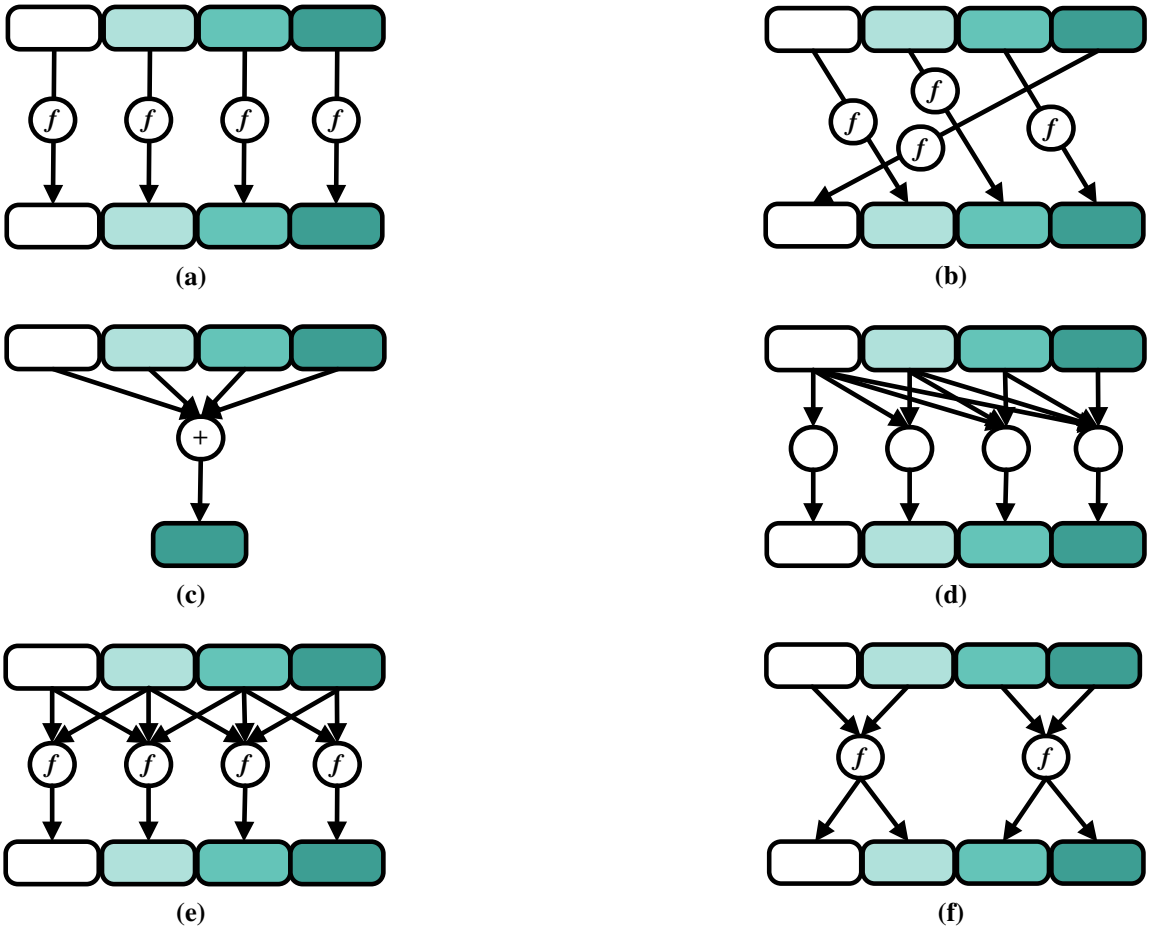


Figure 6.1: The data parallel patterns that Paraprox targets: (a) Map (b) Scatter/Gather (c) Reduction (d) Scan (e) Stencil (f) Partition.

patterns are chosen from all patterns described in ”structured parallel programming” book by McCool [62] based on their popularity in data parallel applications that are tolerant to some degree of approximation.

- ***Map:*** In the map pattern, a function operates on every element of an input array and produces one result per element as shown in Figure 6.1a. To process all the input elements in parallel, a map function should be *pure*. A pure function always generates the same result for the same input, and its execution does not have any side-effects,

e.g., it cannot read or write mutable state. Since there is no need to synchronize between two threads and no sharing of data is necessary, the map pattern is perfectly matched to data parallel, many-core architectures. In parallel implementations of map patterns, each thread executes one instance of a map function and generates its corresponding result. This pattern is used in many domains, including image processing and financial simulations.

- ***Scatter/Gather:*** Scatter and gather patterns are similar to map patterns but their memory accesses are random as illustrated in Figure 6.1b. Based on McCool's definition [62], scatter is a map function that writes to random locations, and gather is the combination of a map function with memory accesses that read from random input elements. The parallel implementations of scatter/gather patterns are similar to map implementations. This pattern is commonly found in statistics applications.

- ***Reduction:*** When a function combines all the elements of an input array to generate a single output, it is said to be performing a reduction (Figure 6.1c). If the function used by the reduction pattern is both associative and commutative, e.g., XOR, the order in which the reduction operation is applied to its inputs is unimportant. In this case, tree-based implementations can be used to parallelize such a reduction. Reductions can be found in many domains, such as machine learning, physics, and statistics.

- ***Scan:*** The all-prefix-sums operation, more commonly termed *scan*, applies an associative function to an input array and generates another array. Every N^{th} element of the output array is the result of applying the scan function on the first N (inclu-

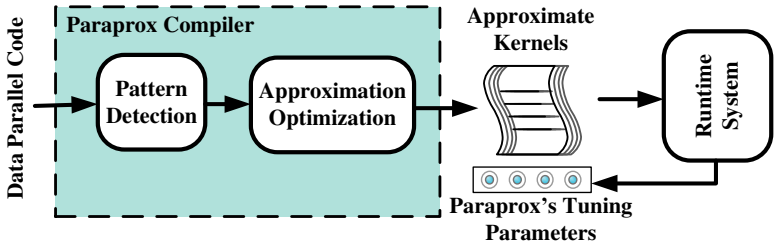


Figure 6.2: Approximation system framework.

sive scan) or $N-1$ (exclusive scan) input elements. An inclusive scan example is shown in Figure 6.1d. The scan pattern is common in the signal processing, machine learning, and search domains.

- ***Stencil:*** In a stencil pattern, each output element is computed by applying a function on its corresponding input array element and its neighbors as shown in Figure 6.1e. This pattern is common in image processing and physics applications.

- ***Partition:*** The partition (or tile) pattern is similar to the stencil pattern. The input array is divided into partitions and each partition is processed separately. Each partition is wholly independent of the others as shown in Figure 6.1f. Partitioning is commonly used in data parallel applications to efficiently utilize the underlying architecture's memory hierarchy to improve performance. This pattern is common in domains such as image processing, signal processing, and physics modeling.

In order to manage the output quality during execution, the Paraprox compilation framework should be used in tandem with a runtime system like Green [10] or SAGE [95] as shown in Figure 6.2. After Paraprox detects the patterns in the program and generates approximate kernels with different tuning parameters, the runtime profiles the kernels and tunes the parameters so that it provides the best performance. If the user-defined *target*

output quality (TOQ) is violated, the runtime system will adjust by retuning the parameters and/or selecting a less aggressive approximate kernel for the next execution.

6.3 Approximation Optimizations

We will now discuss the approximation optimizations that are applied to each data parallel pattern. For each pattern, we describe the intuition behind the optimization, the algorithm used to detect such a pattern, the implementation of the optimization, and tuning parameters that are used by a runtime to control the performance and accuracy of an approximate kernel during execution.

6.3.1 Map & Scatter/Gather

6.3.1.1 Idea:

Paraprox applies *approximate memoization* to optimize map and scatter/gather patterns. This technique replaces a function call with a query into a lookup table which returns a precomputed result. Since the size of this lookup table is limited by the size of memory and by performance considerations, there are situations in which the exact result is not stored in the table. In such cases, Paraprox finds the element nearest to the input present in the lookup table and returns that element instead. Consequently, the quality of the output is inversely proportional to the size of the lookup table (a.k.a. the number of quantization levels). As this optimization replaces the computations done by map and scatter/gather functions with a memory access, the unoptimized code should have more latency due to computation than that of one memory operation in order to achieve speedup.

To fill the lookup table with precomputed data, Paraprox computes the output of the map or scatter/gather function for a number of representative input sets (quantization levels) offline. During runtime, the launched kernel's threads use this lookup table to find the output for all input values.

6.3.1.2 Detection:

To detect map or scatter/gather patterns, Paraprox checks all functions in the input program to look for functions that can be replaced by a lookup table. There are two requirements for such functions. First, these functions should be pure. Pure functions do not have side effects and their output is only dependent on their inputs. To meet these constraints, pure functions should not:

- read or write any global or static mutable state.
- call an impure function.
- perform I/O.

In addition to being pure, these functions should not access global memory during execution, and their outputs should not be dependent on the thread ID. Therefore, Paraprox looks for functions which do not contain global/shared memory accesses, atomic operations, computations involving thread or block IDs, or calls to impure functions. If a function meets all these conditions, Paraprox marks it as a candidate for approximate memoization.

It should be noted that although Paraprox works at a function granularity, it is possible to find pure sections of code within a function. Detection of such map or scatter/gather sections within a function is left for future research.

As Paraprox will replace computation with memory accesses, this optimization should only be applied to computationally intensive map and scatter/gather patterns in order to achieve high performance improvements. To determine which functions to optimize, Paraprox computes the sum of the latencies of each instruction in the function as a metric to estimate the function's computation cycles as follows:

$$cycles\_needed \quad = \quad \sum_{inst \in f} latency(inst) \tag{6.1}$$

Instruction latency values are passed to Paraprox in a table based on the target architecture. Paraprox uses this latency table to compute the $cycles\_needed$ for all map and scatter/gather functions found in the program. For GPUs, we used microbenchmarks from Wong et al. [118] to measure the $latency$ of all instructions. We found that if a function's $cycles\_needed$ is at least one order of magnitude greater than the L1 read latency, it can benefit from this approximation. Therefore, Paraprox only applies the approximation on such functions.

6.3.1.3 Implementation:

Approximate memoization is accomplished in three steps: quantizing the inputs, joining these bit representations of inputs together to create an address, and accessing a lookup table using that address to get the final result. Figure 6.3(a) shows the dataflow in the BlackScholesBody function of the *BlackScholes* benchmark. This function meets all the

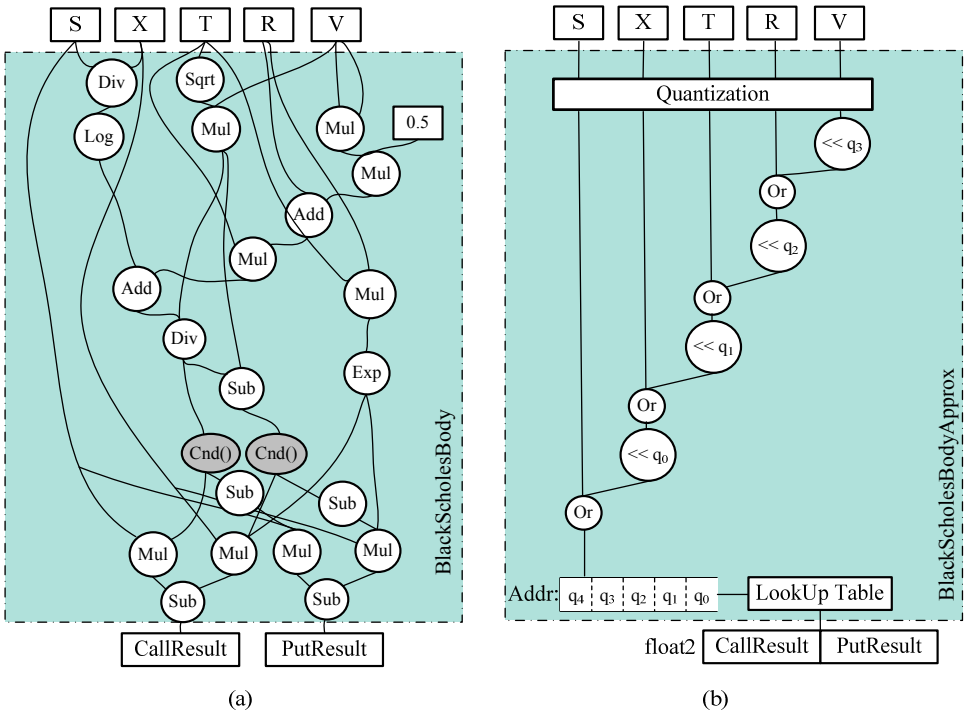


Figure 6.3: (a) illustrates the dataflow graph of the main function of the *BlackScholes* benchmark. The function $Cnd()$ is a pure function. (b) shows the approximate kernel created using the map and scatter/gather technique described in 6.3.1.

candidacy conditions described in Section 6.3.1.2. Figure 6.3(b) shows the approximate version of the same function.

Paraprox quantizes the function's inputs to generate an address into the lookup table. For a quantized input i, Paraprox can control the output quality of the approximate function by altering the number of bits (q_i) used to represent that input. If a pattern has multiple input variables, e.g. i and $i+1$, each input has its own quantization bits (q_i and q_{i+1}). When concatenated together, these quantization bits form the address into the lookup table. The table's size is thus equal to 2^Q, where $Q = \sum_{i=0}^{n} q_i$ for all n inputs.

Using fewer bits reduces the number of quantization levels (2^{q_i}) that represent an input value, thus limiting the input's accuracy. Conversely, increasing the number of bits will

permit more quantization levels, which will increase the accuracy of the input representation. If the pattern's output is very sensitive to small changes in the input and there are not enough bits allocated to adequately represent this, Paraprox detects that the output quality is deteriorating and increases q_i. On the other hand, if the output is not very sensitive to changes in the input or the input's dynamic range is very small, Paraprox can reduce q_i.

Bit tuning: The process of determining q_i for inputs is called *bit tuning* and is performed offline. For each input argument to the function, Paraprox computes the range of the function's output by applying training data to the function and storing the results in memory. If an input at runtime is not within this precomputed range, it will map to the nearest value present in the lookup table.

If a function has multiple inputs, naively dividing the quantization bits equally amongst all inputs does not necessarily yield ideal results, so Paraprox can unevenly divide the bits of the quantized input to favor some inputs over others. For example, in Figure 6.3, the BlackScholesBody function has five inputs, two of which (R and V) are always constant during profiling. When Paraprox detects this, it chooses to allot all quantized bits to represent the other three variable inputs.

Our experiments show that the overall speedup of this optimization is dependent on the size of the lookup table but not the number of bits in q_i assigned to each input. However, the quantization bits still need to be distributed carefully amongst inputs to guarantee satisfactory output quality.

To reduce output quality loss for a given lookup table size, bit tuning uses a tree algorithm. Each node in the tree corresponds to an approximate kernel with a specific q_i bits

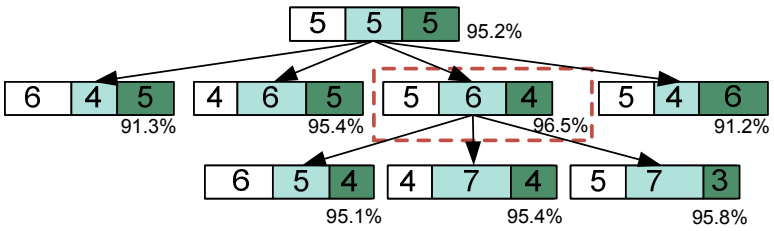


Figure 6.4: An example of how Paraprox's bit tuning finds the number of bits assigned to each input for the BlackScholesBody function. The lookup table has 32768 entries and its address is 15 bits wide. The output quality is printed beside each node. Bit tuning's final selection is outlined with a dotted box.

per input. The root node divides bits equally between the inputs. Figure 6.4 shows the tree for the example shown in Figure 6.3(b). In this example, the lookup table size is 32768, which implies that the address into the table is 15 bits wide. The root of the tree shows that this address initially is evenly split into five bits each for the three variable inputs. Each child node is different from its parent such that one bit is reassigned from one input to an adjacent input.

The bit tuning process starts from the root and uses a steepest ascent hill climbing algorithm to reach a node with the highest output quality. Paraprox checks all the children of each node and selects the one with the best output quality. This process will continue until it finds a node for which all of its children have lower output quality than itself. In the example shown in Figure 6.4, Paraprox starts from node ($q_1 = 5$, $q_2 = 5$, $q_3 = 5$) and checks all its children. Among them, node (5,6,4) has the best output quality. Since all children of node (5,6,4) have a lower output quality than itself, node (5,6,4) is selected, and Paraprox assigns 5, 6, and 4 quantization bits to the first, second, and third inputs, respectively. Paraprox uses this process to find a configuration that returns the highest output quality for the specified lookup table size.

As bit tuning aims to control quality loss, it needs to determine how much error is introduced for each bit configuration it considers. To do so, bit tuning first quantizes the inputs using the division of bits specified by the current tree node under inspection. It then calculates the results of the exact and approximate functions and compares the two to compute a percent difference. Figure 6.4 shows these quality metrics for the BlackScholesBody example. It should be noted that bit tuning does not need to use an actual lookup table as it computes the approximate result that it is currently investigating.

To determine the size of the lookup table, Paraprox starts with a default size of 2048. For each lookup table size, Paraprox performs bit tuning to find the output quality. If the quality is better than the TOQ, Paraprox decreases the size of lookup table to see if it can further improve performance. If the quality is worse than the TOQ, Paraprox doubles the lookup table's size, as larger tables improve accuracy. This process stops when Paraprox finds the smallest table size that has an output quality that satisfies the TOQ.

After computing the size of the lookup table and assigning quantization bits for each input, Paraprox populates the lookup table. For each quantization level of each input, Paraprox computes the output and stores it in the lookup table. After filling the lookup table, Paraprox passes the approximate kernel a pointer to the lookup table. The lookup table can be allocated in the global memory, or if a target has fast access memories, like the constant cache or shared memory in GPUs, those can be utilized instead of the global memory. Section 6.4.4.2 investigates these different options and compares their impacts on performance. Should the output quality change during runtime, Paraprox can accelerate the process of switching between different sized lookup tables by storing multiple tables in memory and changing the pointer passed to the kernel at runtime to reflect this decision.

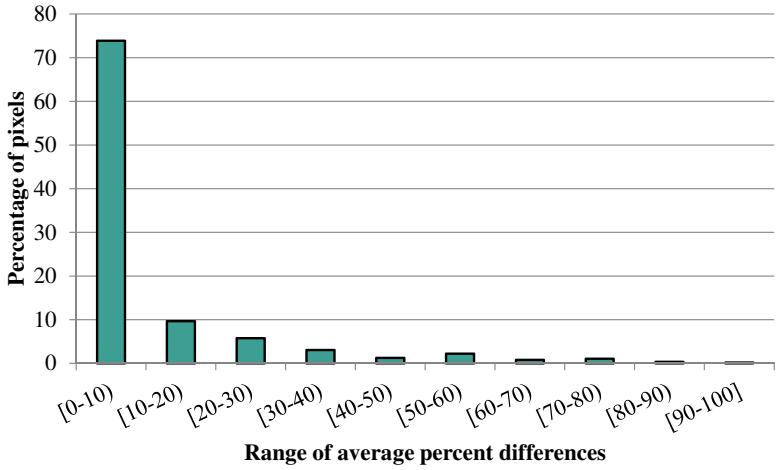


Figure 6.5: The average percent differences between adjacent pixels in ten images. More than 75% of pixels are less than 10% different from their neighbors.

Paraprox can generate as many tables as it can fit in memory. However, in our experiments we found that no more than three tables are needed for our benchmarks.

6.3.1.4 Tuning Parameter:

To tune the output quality and performance, Paraprox allows the runtime to select amongst lookup tables of different sizes.

6.3.2 Stencil & Partition

6.3.2.1 Idea:

The stencil and partition approximation algorithm is based on the assumption that adjacent elements in the input array usually are similar in value. This is often the case for domains such as image and video processing, where neighboring pixels tend to be similar if not the same. To evaluate this assumption, Figure 6.5 shows the average percent difference of each pixel with its eight neighbors, which constitute a tile, for all pixels in 10

different images. As the figure shows, on average, more than 70% of the each image's pixels have less than 10% difference from their neighbors. Therefore, most of the neighbors of each pixel have similar values.

Under this assumption, rather than access all neighbors within a tile, Paraprox accesses only a subset of them and assumes the rest of the neighbors have the same value. This is similar to changing the resolution of the input data. However, the advantage of this optimization over changing the resolution is that it is possible to control the aggressiveness of approximation during runtime with really low overhead. However, changing the resolution should be done offline because there is a high overhead to change the resolution during runtime.

6.3.2.2 Detection:

To detect stencil/partition patterns, Paraprox checks the load accesses to the arrays and looks for a constant number of affine accesses to the same array, indicating a tile size. These accesses can be found in loops with a constant loop trip or in manually unrolled loops. After finding these accesses, Paraprox computes the tile's size and dimensionality. Paraprox detects stencil/partition patterns based on the array access indices ($(f + i) * w + g + j$). Parameters f, g, and w are the same (loop invariant) for all accesses that are examined. Parameters i and j can be hand-coded constants or loop induction variables. The size of a tile can be determined by looking at the dynamic range of i and j.

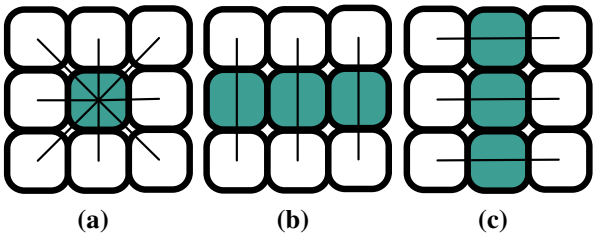


Figure 6.6: The three different schemes Paraprox uses to approximate the stencil pattern. (a) illustrates how the value at the center of the tile approximates all neighboring values. (b) and (c) depict how one row/column's values approximate the other rows/columns in the tile.

6.3.2.3 Implementation:

To approximate stencil/partition patterns, Paraprox uses three different approximation schemes: center, row, and column based. For each approximation, a *reaching distance* parameter controls the number of memory elements that Paraprox accesses. In the center based approach, the element at the center of a tile is accessed and Paraprox assumes that all its neighbors have the same value. When Paraprox accesses an element, its neighbors, whose distances from the accessed element are less than the reaching distance, will not be accessed as shown in Figure 6.6a.

Figures 6.6b and 6.6c illustrate the row and column based approximation schemes. In these schemes, one row/column within a tile is accessed, and all other rows/columns within a reaching distance from it are assumed to be the same and are left unaccessed.

6.3.2.4 Tuning Parameter:

To control performance and output quality, Paraprox allows a runtime to select from various approximate kernels and tune each kernel's reaching distance.

6.3.3 Reduction

6.3.3.1 Idea:

To approximate reduction patterns, Paraprox aims to predict the final result by computing the reduction of a subset of the input data in a way similar to loop perforation [2]. Figure 6.7 illustrates how this concept is applied. The assumption here is that the data is distributed uniformly, so a subset of the data can provide a good representation of the entire array. For example, instead of finding the minimum of the original array, Paraprox finds the minimum within one half of the array and returns it as the approximate result. If the data in both subarrays have similar distributions, the minimum of these subarrays will be close to each other and approximation error will be negligible.

Some reduction operations like addition need some adjustment to produce more accurate results. For example, after computing the sum of half of an array, if the result is doubled it more closely resembles the results of summing the entire array, thus the output quality is improved. In this case of addition, Paraprox assumes that the other half of the array has the exact same sum as the first half, so it doubles the approximated reduction result.

6.3.3.2 Detection:

Reduction recognition has been studied extensively by previous works [119, 87]. To detect reduction patterns, Paraprox searches for accumulative instructions that perform an operation like $a = a + b$, where a is called the reduction variable and addition is the reduction operation. Reduction loops have the following two characteristics: *a)* they contain an

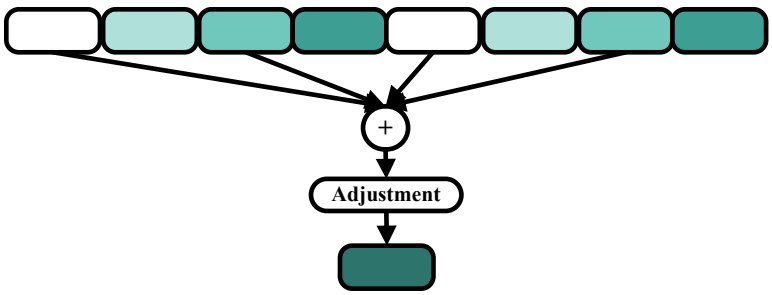


Figure 6.7: An illustration of how Paraprox approximates the reduction pattern. Instead of accessing all input elements, Paraprox accesses a subset of the input array and adds adjustment code to improve the accuracy.

accumulative instruction; and *b)* the reduction variable is neither read nor modified by any other instruction inside the loop.

In order to parallelize a reduction loop for a data parallel architecture, tree-based reduction implementations are often used. These reductions have three phases. In the first phase (Phase I), each thread performs a reduction on a chunk of input data. In the next phase (Phase II), each block accumulates the data generated by its threads and writes this result to the global memory. The final phase (Phase III) then accumulates the results of all the blocks to produce the final results. All of the phases contain a reduction loop that Paraprox optimizes, creating approximate kernels for each loop. The runtime determines which approximate version to execute.

Atomic operations can also be used to write data parallel reductions. An atomic function performs a read-modify-write atomic operation on one element residing in global or shared memory. For example, CUDA's *atomicInc()* and OpenCL's *atomic_inc()* both read a 32-bit word at some address in the global or shared memory, increment it, and write the result back to the same address [72, 46]. Among atomic operations, the atomic add, min, max, inc, and, or, and xor operations can be used in a reduction loop. Paraprox searches

for and marks loops containing these operations as reduction loops.

6.3.3.3 Implementation:

After detecting a reduction loop, Paraprox modifies the loop step size to skip iterations of the loop. In order to execute every N^{th} iteration and skip the other $N - 1$ iterations, Paraprox multiplies the loop step by N. We call N the *skipping rate*. For example, if Paraprox multiplies the loop step size by four, only a quarter of the original iterations are executed and the rest are skipped.

If the reduction operation is addition, Paraprox inserts adjustment code after the loop. This code multiplies the result by the skipping rate. To make the adjustment more accurate, the reduction variable's initial value should be equal to zero before the reduction loop. Otherwise, by multiplying the result, the initial value is multiplied as well which produces an unacceptable output quality. In order to address this, Paraprox replaces the loop's reduction variable with a temporary variable set to zero just before the loop's entrance. After adjustment, Paraprox then adds the scaled temporary variable back to the original reduction variable to produce the final result.

6.3.3.4 Tuning Parameter:

Paraprox allows a runtime to change the skipping rate in order to tune the speedup and accuracy of the kernels.

6.3.4 Scan

6.3.4.1 Idea:

To approximate scan patterns, Paraprox assumes that differences between elements in the input array are similar to those in other partitions of the same input array. Parallel implementations of scan patterns break the input array into subarrays and computes the scan result for each of them. In order to approximate, Paraprox only applies the scan to a subset of these subarrays and uses its results for the rest of the subarrays.

As the N^{th} element of the scan result is the sum of the first N elements of its input array, any change to the N^{th} element modifies the N^{th} output element and all elements afterwards. Therefore, if Paraprox applies approximation to one of the early input elements, any approximation error will propagate to the results for all the following elements, resulting in an unacceptable output quality. This effect is studied in Section 6.4.4.3.

In order to avoid this cascading error, rather than uniformly skipping loop iterations, Paraprox predicts the last elements of the scan results by examining the first output elements. Figure 6.8 presents an example of how Paraprox copies the first elements of the result to the end of the array to approximate the last elements.

6.3.4.2 Detection:

The data parallel implementation of the scan pattern is traditionally composed of three phases as illustrated in Figure 6.9. As an example, this figure shows how these phases compute the scan results for an input array containing all ones. In the first phase, the input is divided into many subarrays and each block of threads performs a scan on one

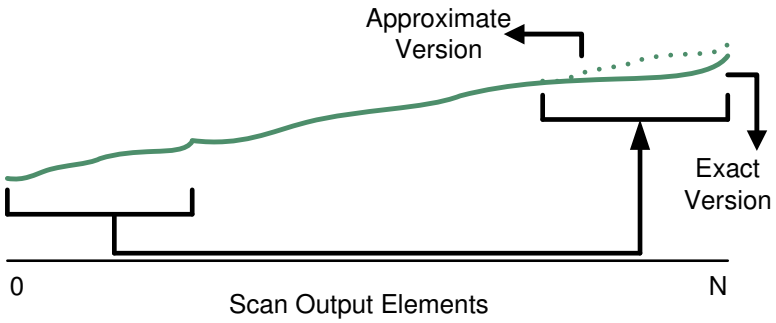


Figure 6.8: An example of how Paraprox uses the first elements of the scan results to approximate the end of the output array.

subarray and stores results in a partial scan array. The sum of each subarray is also written to another array called *sumSub*. The second phase then runs a scan on the *sumSub* array. The i^{th} element of *sumSub*'s scan result is equal to the sum of elements in subarrays 0 to i. In the third phase, every i^{th} element of *sumSub*'s scan result is added to the scan results of the $i+1$ partial scan subarray to produce the final scan results.

Because of its complicated implementation, detecting a scan pattern is generally difficult. A programmer can mark scan patterns for the compiler using pragmas, or the compiler can use template matching to find scan kernels used in benchmarks [73]. Paraprox uses the second approach by performing a recursive post order traversal of the abstract syntax tree of the kernel and comparing it with the template. If they match, Paraprox assumes that the kernel contains a scan pattern.

6.3.4.3 Implementation:

The first phase of the scan pattern takes the longest time to execute, so approximation techniques should target this phase. As mentioned before, Paraprox approximates the re-

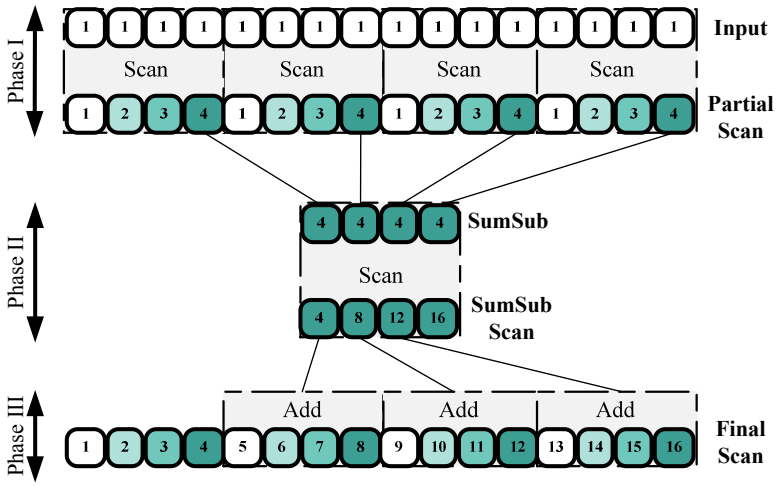


Figure 6.9: A data parallel implementation of the scan pattern has three phases. Phase I scans each subarray. Phase II scans the sum of all subarrays. Phase III then adds the result of Phase II to each corresponding subarray in the partial scan to generate the final result. This figure depicts how the scan is computed for an input array of all ones.

sults for the last subarrays to prevent the propagation of error through all of the results. In this approximation, Paraprox assumes that last subarrays have similar scan results to the first subarrays. Therefore, instead of computing scan results for all subarrays, Paraprox *skips* some and uses the first multiple subarrays' scan results in place of the scan results for the skipped subarrays.

In order to skip the last N subarrays, Paraprox skips some of the computations in Phases I and II. In Phase I, Paraprox launches fewer blocks to skip the last N subarrays. In Phase II, Paraprox changes the argument containing the number of subarrays that is passed to the kernel.

In Phase III, threads that are responsible for adding to generate the first N subarrays add their scan results to the last element of Phase II's results (the *sumSub* scan array) and write these results as the scan's output for the last skipped subarrays. Figure 6.8 shows how

Applications	Domain	Input Size	Patterns	Error Metric
BlackScholes [73]	Financial	4M elements	Map	L1-norm
Quasirandom Generator [73]	Statistics	1M elements	Map	L1-norm
Gamma Correction	Image Processing	2048x2048 image	Map	Mean relative error
BoxMuller [73]	Statistics	24M elements	Scatter/Gather	L1-norm
HotSpot [26]	Physics	1024x1024 matrix	Stencil-Partition	Mean relative error
Convolution Separable [73]	Image Processing	2048x2048 image	Stencil-Reduction	L2-norm
Gaussian Filter	Image Processing	512x512 image	Stencil	Mean relative error
Mean Filter	Image Processing	512x512 image	Stencil	Mean relative error
Matrix Multiply [73]	Signal Processing	2560x2560 matrix	Reduction-Partition	Mean relative error
Image Denoising [73]	Image Processing	2048x2048 image	Reduction	Mean relative error
Naive Bayes [104]	Machine Learning	256K elements with 32 features	Reduction	Mean relative error
Kernel Density Estimation [66]	Machine Learning	256K elements with 32 features	Reduction	Mean relative error
Cumulative Frequency Histograms	Signal Processing	1M elements	Scan	Mean relative error

Table 6.1: Applications specifications for Paraprox evaluation

these threads copy an early portion of the results to generate the result's last elements.

6.3.4.4 Tuning Parameter:

A runtime can control the number of subarrays Paraprox skips in order to tune output quality and performance.

6.4 Experimental Evaluation

6.4.1 Methodology

The Paraprox compilation phases are implemented in the Clang compiler version 3.3. Paraprox's output codes are then compiled into GPU binaries using the NVIDIA nvcc compiler release 5.0. GCC 4.6.3 is used to generate the x86 and OpenCL binaries for execution on the host processor. To run OpenCL code on the CPU, we used the Intel OpenCL driver. We evaluated Paraprox using a system with an Intel Core i7 965 CPU and a NVIDIA GTX 560 GPU with 2GB GDDR5 global memory. We selected 13 applications from various domains and different patterns as benchmarks. We ran each application 110 times with different input sets. We ran the first 10 executions to train and detect the best kernel, and

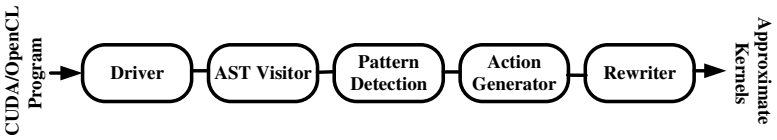


Figure 6.10: *Paraprox's compilation flow.*

then we measured and averaged the runtimes of the next 100 executions. A summary of each application's characteristics is shown in Table 6.1.

Compilation flow in Paraprox: This section describes Paraprox's compilation flow as illustrated in Figure 6.10. First, Clang's *driver* generates the abstract syntax tree (AST) of the input code and sends it to the AST visitor. The *AST visitor* traverses the AST and runs the *pattern detector* on each kernel. The *pattern detector* identifies the parallel patterns within each kernel, and informs the action generator which kernels contain what patterns. The *action generator* then creates a list of actions for each approximate kernel, where an action represents a modification to the output CUDA code for each optimization applied. These actions include: adding, deleting, and substituting an expression in the final code. For each list of actions, the *rewriter* copies the input kernel and applies all actions on the copied version and generates the approximate kernel. To evaluate the impact of Paraprox's optimizations on the CPU, we created a CUDA-to-OpenCL script that converts Paraprox's generated CUDA code to an equivalent OpenCL version.

6.4.2 Results

In this section, we analyze how Paraprox's optimizations affect the execution time and accuracy of different applications. Figure 6.11 presents the results for the benchmarks run separately on a CPU and a GPU. The speedup is relative the exact execution of each

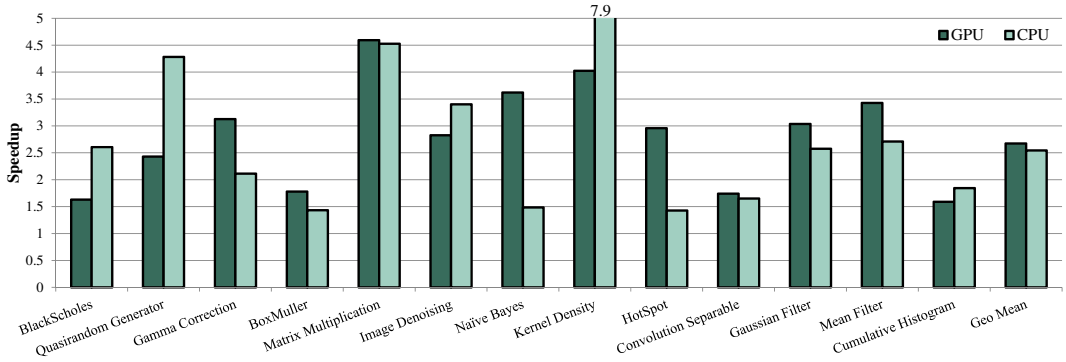


Figure 6.11: The performance of all applications approximated by Paraprox for both CPU and GPU code. The baseline is the exact execution of each application on the same architecture. In these experiments, the target output quality (TOQ) is 90%.

program on the same architectures. As seen in the figure, Paraprox achieves an average speedup of ~2.5x for approximated code run on either the CPU or GPU with a target output quality of 90%.

Output Quality: To assess the quality of each application's output, we used application-specific evaluation metrics as listed in Table 1. For benchmarks that already contained a specific evaluation metric, the included metric was used. Otherwise, we used the mean relative error as an evaluation metric. For all benchmarks, we compare the output of the unmodified, exact application to the output of the approximate kernel created by Paraprox.

A case study by Misailovic et al. [68] shows that users will tolerate quality loss in applications such as video decoding provided it does not exceed ~10%. Similar works [96, 10, 33, 95] cap quality losses for their benchmarks at around 10%. SAGE [95] verified this threshold using the experiments in the LIVE image quality assessment study [99]. Images in LIVE's database have different levels of distortion and were evaluated by 24 human subjects, who classified the quality of the images using a scale equally divided

amongst the following ratings: ”Bad,” ”Poor,” ”Fair,” ”Good,” and ”Excellent.” SAGE [95] showed that more than 86% of images with quality loss less than 10% were evaluated as ”Good” or ”Excellent” by human subjects in the LIVE study. Therefore, we used 90% as the minimum target output quality (TOQ) in our experiments.

6.4.3 Performance Improvement

Paraprox applied map approximation to the *BlackScholes*, *Quasirandom Generator*, *Gamma Correction*, and *BoxMuller* benchmarks. For *BlackScholes*, Paraprox detects two map functions: *Cnd()* and *BlackScholesBody()*. Since the estimated cycle count for Cnd() is low, Paraprox only applies the optimization on BlackScholesBody() which has a high estimated cycle count. As a result, *BlackScholes* achieves ~60% improvement in performance with <10% loss in output quality. *BoxMuller* has a scatter/gather functionwith two inputs and two outputs. *Gamma Correction* is very resilient to quality losses caused by approximation, as its output quality remains at 99% while it achieves >3x speedup on the GPU. When reducing the lookup table size, however, its output quality drops suddenly to <90%. *BlackScholes* and *Quasirandom Generator* get better results on the CPU but *Gamma Correction* and *BoxMuller* perform better on the GPU. The reason is that for benchmarks that can retain good output quality with smaller lookup tables, the GPU achieves better performance. However, as the size of lookup table increases, the number of cache misses increases. In such cases, execution on a CPU is preferable to that on a GPU as cache misses have a lower impact on the performance for CPUs.

The reduction approximation is applied to the *Matrix Multiplication*, *Naive Bayes trainer*, *Image Denoising*, and *Kernel Density Estimation* applications. *Matrix Multiplication* and

Image Denoising show similar performance on both the CPU and GPU. On the other hand, *Naive Bayes* achieves better speedup on the GPU. The approximated *Naive Bayes* performs very well on a GPU (>3.5x vs ~1.5x on a CPU) since this benchmark uses atomic operations, which are more expensive for GPU architectures with many threads running concurrently. By skipping a subset of atomic operations, great speedups in execution time are achieved on a GPU. Since the main component of *Kernel Density Estimation* is an exponential instruction and there is hardware support for such transcendental operations on a GPU (i.e. the special function unit on a CUDA device), skipping these operations provides better performance improvements for CPUs than it does for GPUs.

The *HotSpot*, *Convolution Separable*, *Gaussian filter*, and *Mean Filter* applications contain stencil patterns. *HotSpot*, *Gaussian filter*, and *Mean filter* use 3x3 tiles and *Convolution Separable* has two stencil loops with 1x17 tiles. Since the loop in *Mean Filter* is unrolled manually by the programmer and memory accesses are kept outside the function while computations are inside, there is no reduction loop and the reduction optimization is not applied. Paraprox just applies the stencil optimization on this application. On the other hand, *Convolution Separable* has both stencil and reduction patterns. Paraprox applies both optimizations on this application. The stencil optimization returns a 1.7x performance speedup while maintaining 90% output quality while the reduction optimization results in a 1.6x speedup. On the other hand, the reduction optimization's performance is better than stencil optimization for CPU. Therefore, when targeting a GPU Paraprox only used the results of the stencil optimization in its final kernel, and when targeting a CPU it used the reduction optimization. Because the partition and stencil optimizations primarily optimize memory accesses, speedups are greater for GPU approximated code as memory accesses

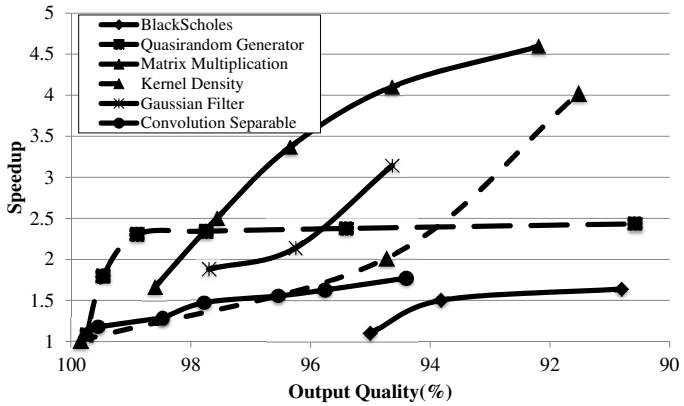


Figure 6.12: *Controlling the speedup and output quality by varying an optimization's tuning parameters for six benchmarks.*

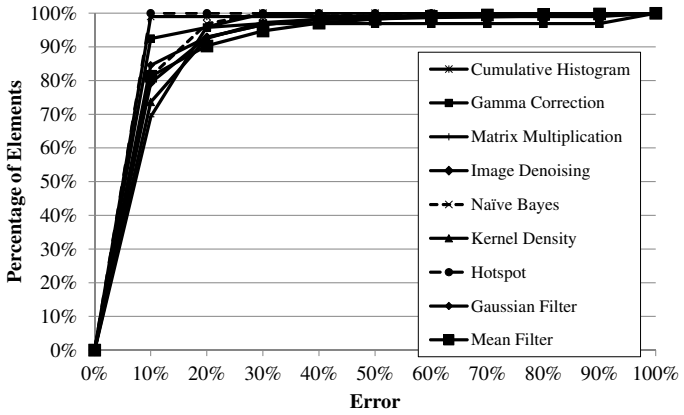


Figure 6.13: *The CDF of final error for each element of an application's output with the $TOQ = 90\%$. The majority of output elements ($>70\%$) have $<10\%$ error.*

are more costly on this platform.

The *Cumulative Histogram* benchmark contains a scan pattern. This application is another resilient application — even when skipping half of the subarrays, the output quality stays at ~99%. For this pattern, the speedup is similar for both CPU and GPU approximated kernels.

Performance-Quality Tradeoffs: Figure 6.12 illustrates how Paraprox manages the performance-accuracy tradeoff for six benchmarks. The map approximated *BlackScholes* starts with 95% output quality and performance similar to the exact version, but as the size of the lookup

table decreases, the speedup increases to 1.6x speedup with only ~4% more loss in quality. Similar behavior is observed for the *Quasirandom Generator*. When the table size is small enough to fit in the cache, the speedup gains begin to saturate for these map optimized kernels. Both *Matrix Multiplication* and *Kernel Density Estimation* contain reduction patterns. As Paraprox doubles the skipping rate for these kernels, the difference between two consecutive nodes grows, thus causing both the speedup and quality loss to grow. The performance of *Gaussian Filter* and *Convolution Separable* rises as output quality degrades. For *Convolution Separable*, Paraprox changes the reaching distances of both loops in the kernel to control the output quality. Since *Gaussian Filter* applies a 2D filter to an image, Paraprox uses row, column, and center stencil patterns to control the output quality. For this benchmark, Paraprox gets >2x speedup with <4% quality loss.

Error Distribution: To study each application's quality losses in more detail, Figure 6.13 shows the cumulative distribution function (CDF) of the error for each element of the application's output with the $TOQ = 90\%$. The CDF illustrates the distribution of output errors amongst an application's output elements. The figure shows that only a modest number of output elements see large output error. The majority (70%-100%) of each approximated application's output elements have an error of $<10\%$.

6.4.4 Case Studies

6.4.4.1 Specialized Optimizations Achieve Better Results:

To show that one optimization does not work well when generally applied, we apply only the reduction optimization to benchmarks that do not contain such a pattern. We

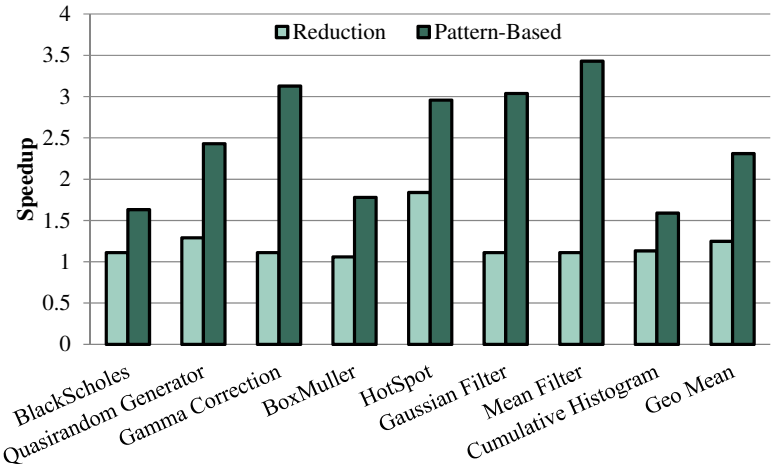


Figure 6.14: *A performance comparison of the reduction optimization vs. specific pattern-based optimizations on benchmarks that do not contain a reduction pattern. In these experiments, the code targets a GPU, and the $TOQ = 90\%$.*

chose this optimization as it is most similar to a well-known approximation technique, loop perforation [2], where loop iterations are skipped to accelerate execution. Figure 6.14 compares the reduction optimization's performance with Paraprox's results on a GPU with the $TOQ = 90\%$. For benchmarks containing map and stencil patterns, skipping iterations results in unmodified output elements. Therefore, the output quality rapidly decreases, severely limiting the speedup. For benchmarks with scan patterns, the cascading error will reduce the output quality and speedup is similarly limited. On average, the reduction optimization alone achieves only ~25% speedup, compared to the 2.3x speedup that Paraprox achieves by matching patterns to specialized optimizations.

6.4.4.2 Design Considerations for the Map Optimization:

To fully investigate the impact of map approximation on both accuracy and performance, we used four common computationally intensive map functions from different domains:

- **Credit card balance equation [1]:** This equation finds the number of months it will take to pay off credit card debt.

$$N(i) = \frac{-1}{30}\frac{ln(1+\frac{b_0}{p}(1-(1+i)^{30}))}{ln(1+i)} \tag{6.2}$$

- **Shifted Gompertz distribution [76]:** This equation gives the distribution of the largest of two random variables.

$$F(x) = (1-e^{-bx})e^{-\eta e^{-bx}} \tag{6.3}$$

- **Log gamma [11]:** This equation calculates the logarithm of the gamma function. To implement this equation, we used the CUDA *lgammaf* [72] function.

$$LG(z) = log(\Gamma(z)) \tag{6.4}$$

- **Bass diffusion model [14]:** This equation describes how new products get adopted as an interaction between users and potential users.

$$S(t) = m\frac{(p+q)^2}{p}\frac{e^{-(p+q)t}}{(1+\frac{p}{q}e^{-(p+q)t})^2} \tag{6.5}$$

For all of these equations, all parameters other than the input variable are constant.

Selecting an Output for an Unrepresented Input: As discussed in Section 6.3.1.3, there are a limited number of quantization levels based on the size of the lookup table. It is possible that there are inputs that do not directly map to a precomputed output. In such cases, Paraprox can either select the *nearest* precomputed output, or it can apply *linear* approximation to the two nearest values in the table to generate a result in between these values. Figure 6.15 shows the performance-quality curve for all four equations using the *nearest* and *linear* methods on the GPU. For all four equations, *nearest* gives better performance compared to *linear* but with lower output quality. Even though the same lookup table size is used, *linear* generates more accurate output, but the overhead of adding another memory access and more computation is overwhelming. On the other hand, *linear* is better at achieving higher output quality (~99%). In this experiment, the lookup table is allocated in the GPU's global memory.

As seen in Figure 6.15, the shifted Gompertz distribution achieves a lower speedup than the other functions. This is due to it having many low latency instructions. Both the Bass and Credit equations execute floating point divisions, which translate to subroutine calls to code with high latency and low throughput for GPUs [118]. On the other hand, the Gompertz equation uses several exponential instructions, which are not high latency as they are handled by a special functional unit on a GPU [72].

Location of the Lookup Table: To find which memory location is best for storing lookup tables, we created approximate versions of the Bass function that used the constant, shared,

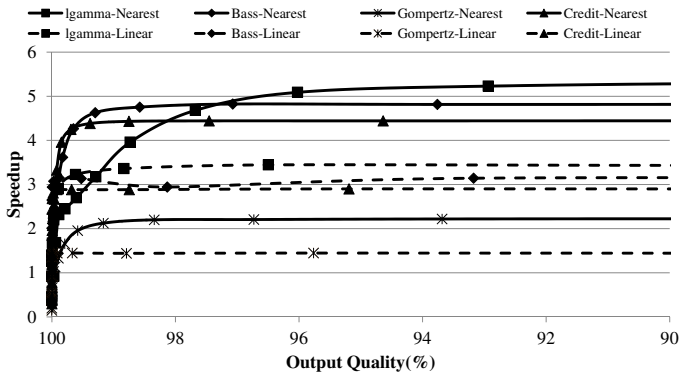


Figure 6.15: *The impact of approximate memoization on four functions on a GPU. Two schemes are used to handle inputs that do not map to precomputed outputs: nearest and linear. Nearest chooses the nearest value in the lookup table to approximate the output. Linear uses linear approximation between the two nearest values in the table. For all four functions, nearest provides better speedups than linear at the cost of greater quality loss.*

and global memories of the GPU to store the lookup table. Figure 6.16 shows the performance versus the table size for these three versions of Bass on the GPU. For lookup tables stored in global and constant memory, we set the L1 cache size to 32KB and size of the shared memory to 16KB. When the lookup table is stored in shared memory, we set the size of the shared memory to 32KB and the L1 cache to 16KB.

Using constant memory never gives optimal results regardless of the cache size. The reason is that for larger table sizes, using shared memory or the global L1 cache will have a lower read latency [118].

To compare global and shared memory, we divided the figure into three regions. When the cache size is small, both global and shared memory show similar speedups. Since it takes time to warm up the L1 cache for global memory, shared memory outperforms the global memory in the second region. In the third region, however, by increasing the size of the lookup table, the overhead of transferring data from global to shared memory is increased and the global memory outperforms the shared memory.

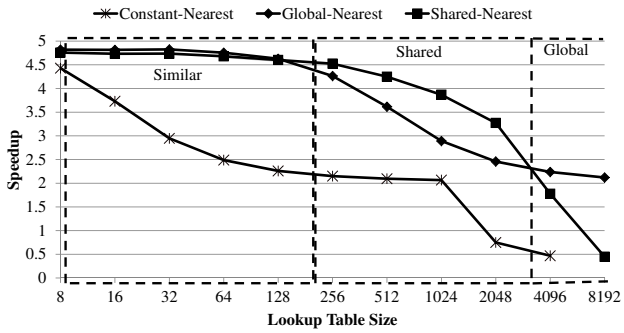


Figure 6.16: *A comparison of the performance of approximate memoization when the lookup table is allocated in the constant, shared, and global memories on a GPU.*

Based on these results, Paraprox generates both shared and global approximate kernels, and the runtime system will choose one based on the performance, output quality, and the lookup table size. If the lookup table is larger than the size of the shared memory, the lookup table must be stored in global memory.

Lookup Table Size vs. Performance: Figure 6.16 shows that speedup drops when increasing the size of the lookup table. Using the CUDA profiler, we found that the number of uncoalesced memory accesses is primarily responsible for this. As the lookup table's size increases, the number of uncoalesced accesses also increases, thus resulting in lower speedups as shown in Figure 6.17. This figure shows that the number of instructions that get serialized increases as the size of the lookup table grows. This serialization is caused by more uncoalesced accesses.

6.4.4.3 Cascading Error in Scan Patterns:

Paraprox approximates the last subarrays of the results for scan patterns. To illustrate why this is done, we use the *Cumulative frequency histogram* benchmark with one million random input data points. For our first run, we "corrupt" the first input subarray (10% of

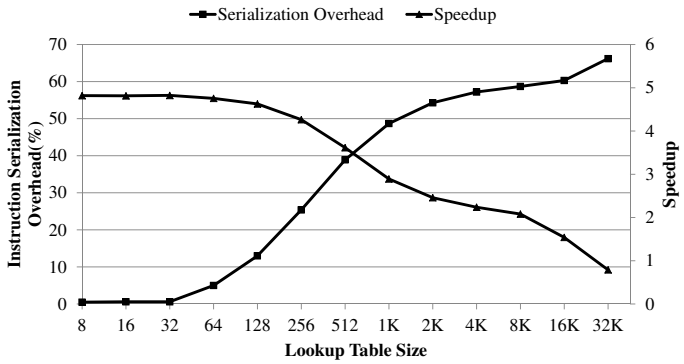


Figure 6.17: *The impact of the lookup table size on the percentage of uncoalesced accesses on the GPU.*

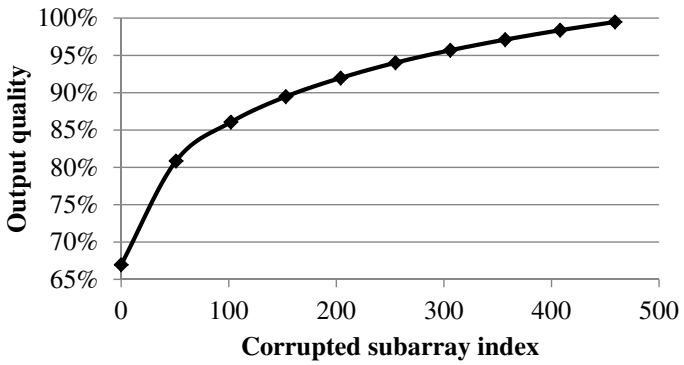


Figure 6.18: *The impact of the starting point of the data corruption on an approximated scan pattern's final output.*

the input elements) by setting its elements to zero. We then move this section of all zeroed data to the next subarray of elements, and rerun the scan. For each test modifying the first to the last input subarray, we record the output quality. Figure 6.18 shows the impact of the starting point of the data corruption on the final output result. When the first subarray of the input is zeroed, the overall output quality will be ~67%. This is caused by the error propagating through the rest of the results. However, if the error happens at the end of the input array, the output quality will be ~99%. Therefore, Paraprox only approximates the last elements of the final scan results to ensure a high output quality.

6.5 Limitations

Paraprox is a research prototype and it has some limitations. Below, we discuss some of these limitations which will be addressed in future work.

Runtime System: In this work, our main focus is automatically generating approximate kernels and providing tuning knobs for a runtime system. Paraprox generates approximate kernels, and a separate runtime system will decide which one to use and how tune the selected kernel's parameters. In our results, we did not consider runtime overhead. However, as shown in SAGE [95] and Green [10], it is not necessary to constantly monitor the quality, so checks are performed every N^{th} invocation. Based on the experiments done in [95], checking the output quality every 40-50 invocations during runtime has less than 5% overhead. This would reduce our reported performance but only by a modest level.

Pattern Recognition: Since we used the AST to detect patterns, variations in code can make the pattern recognition process difficult, especially when detecting scan patterns. However, pattern recognition for other patterns like reduction or detecting pure functions for map and scatter-gather patterns are stable techniques which can detect patterns across a wide variety of implementations. It is also possible to enhance pattern detection by getting hints from the programmer or using higher level languages.

Compiler Optimizations: It is possible that approximation eliminates some other compiler optimization opportunities such as auto-vectorization. In these cases, an approximate kernel might not perform as well as expected. Fortunately, the runtime system chooses which approximate kernel to run based on their speedup and quality. Therefore, if the approximate kernel does show great performance improvement, the runtime system will

choose the original kernel which is highly optimized.

Safety of Optimizations: It is possible that execution of approximate code causes raising exceptions or segmentation faults. There are compiler analyses that detect the possibility of crashing to prevent the compiler from applying the optimizations. For example, for a division that uses an approximated output and may raise a divide by zero exception, it is possible to instrument the code to skip this calculation where the approximated divisor is zero. However, improving the safety of approximation techniques is beyond the scope of this work and it is left for future study.

6.6 Related Work

Pattern-based programming is well-explained by McCool [62]. This book introduces various parallel patterns. Our focus is on the detection and approximation of data parallel patterns.

The concept of trading accuracy for improved performance or energy consumption is well-studied [84, 86, 2, 10, 96, 9, 32, 67, 33, 95, 7, 6]. Previous approximation techniques can be categorized in three categories:

Software-based: Using software approximation, SAGE's [95] framework accelerates programs on GPUs. SAGE's goal is to exploit the specific microarchitectural characteristics of the GPU to achieve higher performance. Although these optimization performs better than general methods, their applicability is limited compared to Paraprox's approximation methods. SAGE also has a runtime system which Paraprox can use to tune and calibrate the output quality during runtime.

Rinard et al. [84, 86] present a technique for automatically deriving probabilistic distortion and timing models that can be used to manage the performance-accuracy tradeoff space of a given application. Given a program that executes a set of tasks, these models characterize the effect of skipping task executions on the performance and accuracy. Agarwal et al. [2] use code perforation to improve performance and reduce energy consumption. They perform code perforation by discarding loop iterations. Paraprox uses a similar method for reduction patterns, but while loop perforation is applied only to sequential loops, Paraprox applies it to *all* loops in such patterns. Skipping iterations, however is not suitable for all data parallel patterns, so Paraprox only applies it to loops with reduction patterns. For example, by skipping iterations of a map loop, a subset of the output array will be left unmodified which results in an unacceptable output quality. A variation of approximate memoization is utilized in a work by Chadhuri [25] for sequential loops. Our approach is different in that it is designed for data parallel applications and it detects when to apply memoization to achieve performance improvement. Previous work by Sartori et. al. [98] targets control divergence on the GPU. Rinard et. al. [84] also proposes an optimization for parallel benchmarks that do not have balanced workloads. Misailovic et. al. [67] propose probabilistic guarantees for approximate applications using loop perforation. Relaxed synchronization is also used as an approximation method to improve performance [83, 85]. Although these approaches perform well for their target applications, their applicability is far more limited than tools that can identify and finely optimize kernels based on the varied data parallel patterns they may contain, which is one of Paraprox's key contributions.

Programmer-based: Green [10] is a flexible framework that developers can use to take advantage of approximation opportunities to improve performance or energy efficiency. This framework requires the programmer to provide approximate kernels or annotate their code using C/C++ extensions. In contrast to these techniques, Paraprox automatically generates different approximate kernels for each application. Ansel et. al. [9] also propose language extensions to allow the programmer to mark parts of code as approximate. They use a genetic algorithm to select the best approximate version to run. Unlike these approaches, Paraprox chooses the approximation optimization based on the patterns detected in the input code and generates approximate versions automatically for each pattern without programmer annotation. Paraprox, however, can be utilized by the runtime systems introduced in these works to optimize performance.

Hardware-based: EnerJ [96] uses type qualifiers to mark approximate variables. Using this type system, EnerJ automatically maps approximate variables to low power storage and uses low power operations to save energy. EnerJ also guarantees that the approximate part of a program cannot affect the precise portion of the program. Esmaeilzadeh et al. [32] demonstrated dual-voltage operation, with a high voltage for precise operations and a low voltage for approximate operations. Another work by Esmaeilzadeh [33] designs a neural processing unit (NPU) accelerator to accelerate approximate programs. Alvarez et al. [7, 6] introduced hardware-based fuzzy memoization and tolerant region reuse techniques for multimedia applications. Other works [107, 113] also designed different approximate accelerators. Sampson et. al. [97] show how to improve memory array lifetime using approximation. These approximate data-type optimizations and special accelerators

require hardware support. Our approach, however, can be used by current architectures without hardware modification.

Unconventional computing techniques can also be used to exploit accuracy vs. area/power/delay trade-offs. One example is stochastic computing (SC), which performs computation on (pseudo-) random bit-streams that are interpreted as probabilities [4]. Stochastic circuits consist of simple logic gates that perform complex arithmetic operations [3], but they tend to be inaccurate due to their probabilistic nature. SC has a natural accuracy vs. run-time trade-off, that is referred to as progressive precision [4]. This property allows the stochastic circuits to stop computation as soon as a valid output generated, and thus saving time and energy. Alaghi et al. [5] have shown how progressive precision can be exploited in several image-processing applications, and have shown that stochastic circuits can outperform conventional binary circuits.

6.7 Conclusion

Approximate computing, where computation accuracy is traded for better performance or higher data throughput, provides an efficient mechanism for computation to keep up with the exponential growth of information. However, approximation can often be time consuming and tedious for programmers to implement, debug, and tune to achieve the desired results. This work proposes a software-only framework called *Paraprox* that identifies common computation patterns found in data-parallel programs and uses a custom-designed approximation template to replace each pattern. Paraprox enables the programmer to write software once and run it on a variety of commodity processors, without manual tuning for

different hardware targets, input sets, or desired levels of accuracy.

For 13 data-parallel applications, Paraprox yields an average of 2.7x and 2.5x speedup with less than 10% quality degradation compared to an accurate execution on a NVIDIA GTX 560 GPU and Intel Core i7 965 CPU, respectively. We also show that Paraprox is able to control the accuracy and performance by varying template configuration parameters at runtime. Our results show that pattern-specific optimizations yield nearly twice the performance improvement compared to naively applying a single, well-known approximation technique to all benchmarks.

CHAPTER VII

Summary and Conclusion

Heterogeneous systems, where sequential work is done on traditional processors and parallelizable work is offloaded to a specialized computing engine, are mainstream these days. Among the different solutions that can take advantage of this parallelism, GPUs are the most popular solution and have been shown to provide significant performance for general purpose computing. While GPUs provide low-cost and efficient platforms for accelerating massively parallel applications, tedious performance tuning, managing the amount of on-chip memory used per thread, the total number of threads per multiprocessor, and the pattern of off-chip memory accesses, are required to maximize application execution efficiency.

In addition to complex programming model, a lack of performance portability across various systems with different runtime properties is another major challenge. Programmers usually make assumptions about runtime properties when they write a code and optimize it based on those assumptions. However, if any of these properties changes during execution, the optimized code performs poorly. We showed how these runtime properties such as underlying architecture, input size and dimensions, data dependencies between threads,

and data values impact the performance of data parallel applications. One way to solve this problem is to ask the programmer to write different versions of the same code optimized for various runtime properties. However, it is not easy and in many cases it is not a practical solution. Therefore, in order to provide portability, a compilation framework is required to generate multiple versions of the same code. This thesis introduced several solutions such as dynamic compilation, speculation, and approximation to generate data-parallel applications which are optimized for different runtime properties.

To target input portability problem, in Chapter III, we proposed *Adaptic*, an adaptive input-aware compiler for GPUs. Using this compiler, programmers can implement their algorithms once using the high-level constructs of a streaming language and compile them to CUDA code for all possible input sizes and various GPUs targets. Adaptic, with the help of its input-aware optimizations, can generate highly-optimized GPU kernels to maintain high performance across different problem sizes. At runtime, Adaptic's runtime kernel management chooses the best performing kernel based on the input. Our results show that Adaptic's generated code has similar performance to the hand-optimized CUDA code over the original programs input comfort zone, while achieving upto 6x speedup when the input falls out of this range.

In Chapter IV, we proposed *Paragon*: a static/dynamic compiler platform to speculatively and cooperatively run possibly-data-parallel pieces of sequential applications on GPUs and CPUs. Paragon monitors the dependencies for possibly-data-parallel loops running speculatively on the GPU and non-speculatively on the CPU using a light-weight distributed conflict detection system designed specifically for GPUs, and transfers the execution to the CPU in case a conflict is detected. Paragon resumes the execution on the

GPU after the CPU resolves the dependency. We looked at two classes of implicitly data-parallel applications: applications with indirect and pointer memory accesses. My experiment showed that, for applications with indirect memory accesses, Paragon achieves a speedup of 2.5x on average and up to 4x speedup compared to unsafe CPU execution with 4 threads. Also, for applications with pointer memory accesses, Paragon achieves a speedup of 6.8x on average and up to 30x compared to unsafe CPU execution with 4 threads.

In Chapters V and VI, we targeted the problem of value portability for data-parallel execution. *Sage* enables the programmer to implement a program once in CUDA and, depending on the target output quality (TOQ) specified for the program, automatically trade accuracy for performance. This work shows that there are GPU-specific characteristics that can be exploited to gain significant speedups compared to hardware-incognizant approximation approaches. We also discussed how Sage controls the accuracy and performance at runtime using optimization calibration in two case studies. *Paraprox* is also a software-only framework that identifies common computation patterns found in data-parallel programs and uses a custom-designed approximation template to replace each pattern. Paraprox enables the programmer to write software once and run it on a variety of commodity processors, without manual tuning for different hardware targets, input sets, or desired levels of accuracy. We also showed that Paraprox is able to control the accuracy and performance by varying template configuration parameters at runtime. My results showed that pattern-specific optimizations yield nearly twice the performance improvement compared to naively applying a single, well-known approximation technique to applications.

This dissertation has introduced novel techniques for static compilation and runtime systems for data-parallel execution. These techniques provide portability for data-parallel

applications for different devices, input-sets, dependencies, and data values.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] The credit card equation, 2013. http://faculty.bennington.edu/ jzimba/CreditCardEquationDerivation.pdf. 174

[2] A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2009-042, MIT, Mar. 2009. 99, 101, 122, 134, 138, 143, 144, 160, 173, 180, 181

[3] A. Alaghi and J. Hayes. A spectral transform approach to stochastic circuits. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 315–321, Sept 2012. 183

[4] A. Alaghi and J. P. Hayes. Survey of stochastic computing. *ACM Trans. Embed. Comput. Syst.*, 12(2s):92:1–92:19, May 2013. 183

[5] A. Alaghi, C. Li, and J. P. Hayes. Stochastic circuits for real-time image-processing applications. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 136:1–136:6, New York, NY, USA, 2013. ACM. 183

[6] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multi-

media applications. *IEEE Transactions on Computers*, 54(7):922–927, 2005. 180, 182

[7] C. Alvarez, J. Corbal, and M. Valero. Dynamic tolerance region computing for multimedia. *IEEE Transactions on Computers*, 61(5):650–665, 2012. 180, 182

[8] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. In *Proc. of the '09 Conference on Programming Language Design and Implementation*, pages 38–49, June 2009. 140

[9] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proc. of the 2011 International Symposium on Code Generation and Optimization*, pages 85 –96, 2011. 99, 138, 139, 143, 180, 182

[10] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proc. of the '10 Conference on Programming Language Design and Implementation*, pages 198–209, 2010. 99, 122, 138, 139, 143, 149, 168, 179, 180, 182

[11] D. H. Bailey. *Experimental mathematics in action*. A.K. Peters, 2007. 174

[12] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, R. Atanas, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proc. of the 2008 International Conference on Supercomputing*, pages 225–234, 2008. 94

[13] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Proc. of the 19th International Conference on Compiler Construction*, pages 244–263, 2010. 15, 50, 51, 56, 94

[14] F. Bass. A new product growth for model consumer durables. *Management Science*, 50(12):215–227, 1969. 174

[15] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, 2004. 94

[16] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. 69

[17] W. Blume et al. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996. 93

[18] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 247–258, 2008. 93

[19] N. Brunie, S. Collange, and G. Diamos. Simultaneous branch and warp interweaving for sustained gpu performance. In *Proc. of the 39th Annual International Symposium on Computer Architecture*, pages 49–60, 2012. 54, 95

[20] I. Buck et al. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, Aug. 2004. 15, 49

[21] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proc. of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 47–56, 2011. 15, 49

[22] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *Proc. of the 25th International Conference on Machine learning*, pages 104–111, 2008. 39, 45, 46, 120, 140

[23] D. Cederman, P. Tsigas, and M. T. Chaudhry. Towards a software transactional memory for graphics processors. In *Proc. of the 12th Eurographics Symposium on Parallel Graphics and Visualization*, pages 121–129, 2010. 95

[24] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proc. of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 35–46, 2011. 94

[25] S. Chaudhuri, S. Gulwani, R. L. Roberto, and S. Navidpour. Proving programs robust. In *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 102–112, 2011. 181

[26] S. Che, M. Boyer, J. Meng, D. Tarjan, , J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IEEE Symposium on Workload Characterization*, pages 44–54, 2009. 166

[27] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: dependable

distributed software transactional memory. In *Proc. of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 307–313, 2009. 93

[28] B. Coutinho, D. Sampaio, F. Pereira, and W. Meira. Divergence analysis and optimizations. In *Proc. of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pages 320–329, 2011. 54, 95

[29] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 497–508, June 2010. 140

[30] G. Diamos and S. Yalamanchili. Speculative execution on Multi-GPU systems. In *2010 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2010. 94

[31] EMC Corporation. Extracting value from chaos, 2011. www.emc.com /collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf. 97, 142

[32] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–312, 2012. 99, 138, 140, 143, 144, 180, 182

[33] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proc. of the 45th Annual International Symposium on Microarchitecture*, pages 449–460, 2012. 122, 140, 143, 144, 168, 180, 182

[34] A. Frank and A. Asuncion. UCI machine learning repository, 2010. 119

[35] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. Hardware transactional memory for GPU architectures. In *Proc. of the 44th Annual International Symposium on Microarchitecture*, 2011. 60, 95

[36] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, 2006. 50

[37] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, Oct. 2002. 50

[38] M. Grossman, A. Simion, Z. Budimli, and V. Sarkar. CnC-CUDA: Declarative Programming for GPUs. In *Proc. of the 23rd Workshop on Languages and Compilers for Parallel Computing*, pages 230–245, 2010. 49

[39] T. Han and T. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):52–61, 2010. 15, 50, 56, 94

[40] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions.

Proc. of the '06 Conference on Programming Language Design and Implementation, 41(6):14–25, 2006. 55, 93

[41] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–212, 2011. 138

[42] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 152–163, 2009. 19, 23, 49, 50

[43] A. Hormati, Y. Choi, M. Woh, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Macross: Macro-simdization of streaming applications. In *15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 285–296, 2010. 50

[44] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 381–392, 2011. 6, 15, 22, 33, 49

[45] Q. Hou, K. Zhou, and B. Guo. BSGP: bulksynchronous GPU programming. *ACM Transactions on Graphics*, 27(3):1–12, 2008. 15, 49

[46] KHRONOS Group. OpenCL - the open standard for parallel programming of heterogeneous systems, 2013. 1, 15, 161

[47] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August. Automatic speculative doall for clusters. In *Proc. of the 2012 International Symposium on Code Generation and Optimization*, pages 94–103, 2012. 55, 93

[48] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *Proc. of the 2008 International Conference on Parallel Processing*, pages 51–58, 2008. 93

[49] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. of the '08 Conference on Programming Language Design and Implementation*, pages 114–124, June 2008. 50

[50] A. Kulesza and F. Pereira. Structured learning with approximate inference. In *Advances in Neural Information Processing Systems20*, pages 785–792, 2008. 98

[51] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Cascaval. How much parallelism is there in irregular applications? In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–14, 2009. 93

[52] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proc. of the '07 Conference on Programming Language Design and Implementation*, pages 211–222, 2007. 93

[53] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. 20

[54] S. Lee, V. Grover, M. M. T. Chakravarty, and G. Keller. GPU kernels as data parallel array computations in Haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, pages 1–9, 2009. 15, 49

[55] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 101–110, 2009. 15, 49

[56] S. I. Lee, T. Johnson, and R. Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *Proc. of the 16th Workshop on Languages and Compilers for Parallel Computing*, 2003. 83, 121

[57] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 451–460, 2010. 1, 14, 53

[58] A. Leung, O. Lhoták, and G. Lashari. Automatic parallelization for graphics processing units. In *Proc. of the 7th International Conference on Principles and Practice of Programming in Java*, pages 91–100, 2009. 56, 94

[59] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 181–192, Feb. 2007. 140

[60] Y. Li, K. Zhao, X. Chu, and J. Liu. Speeding up K-Means algorithm by GPUs. In *Proc. of the 2010 10th International Conference on Computers and Information Technology*, pages 115 –122, 2010. 140

[61] S. Liu, C. Eisenbeis, and J.-L. Gaudiot. Value prediction and speculative execution on GPU. In *International Journal of Parallel Programming*, volume 39, pages 533–552, 2011. 94, 95

[62] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012. 147, 148, 180

[63] M. Mehrara, J. Hao, P. chun Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proc. of the '09 Conference on Programming Language Design and Implementation*, pages 166–176, June 2009. 55, 93

[64] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proc. of the 2009 International Conference on Supercomputing*, pages 256–265, 2009. 51

[65] J. Menon, M. de Kruijf, and K. Sankaralingam. iGPU: Exception support and speculative execution on GPUs. In *Proc. of the 39th Annual International Symposium on Computer Architecture*, pages 72–83, June 2012. 94

[66] P. D. Michailidis and K. G. Margaritis. Accelerating kerne stimation on the GPU using the CUDA framework. *Journal of Applied Mathematical Science*, 7(30):1447–1476, 2013. 166

[67] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *Proc. of the 18th Static Analysis Symposium*, pages 316–333, 2011. 180, 181

[68] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *Proc. of the 32nd ACM/IEEE conference on Software Engineering*, pages 25 –34, 2010. 122, 168

[69] NVidia. Ptx: Parallel thread execution isa. http://docs.nvidia.com/cuda/parallel-thread-execution/. 39

[70] NVIDIA. CUBLAS Library, 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs /CUBLAS_Library.pdf. 16, 39

[71] NVIDIA. GPUs Are Only Up To 14 Times Faster than CPUs says Intel, 2010. http://blogs.nvidia.com/ntersect/2010/06/gpus-are-only-up-to-14-times-faster-than-cpus-says-intel.html. 1, 14, 53

[72] NVIDIA. NVIDIA CUDA C Programming Guide, version 4.0, 2011. 1, 11, 15, 29, 111, 161, 174, 175

[73] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110, 2012. www.nvidia.com/content/PDF/NVIDIA_Kepler _GK110_Architecture_Whitepaper.pdf. 3, 102, 164, 166

[74] E. Nystrom, H.-S. Kim, and W. Hwu. Bottom-up and top-down context-sensitive

summary-based pointer analysis. In *Proc. of the 11th Static Analysis Symposium*, pages 165–180, Aug. 2004. 93

[75] C. E. Oancea and A. Mycroft. Software thread-level speculation: an optimistic library implementation. In *Proc. of the 1st International Workshop on Multicore Software Engineering*, pages 23–32, 2008. 55, 93

[76] K. Ohishi, H. Okamura, and T. Dohi. Gompertz software reliability model: Estimation algorithm and empirical validation. *Journal of Systems and Software*, 82(3):535 – 543, 2009. 174

[77] Polybench. the polyhedral benchmark suite, 2011. http://www.cse.ohio-state.edu/ pouchet/software/polybench. 83

[78] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: part ii, multidimensional time. In *Proc. of the '08 Conference on Programming Language Design and Implementation*, pages 90–100, 2008. 94

[79] A. Prasad, J. Anantpur, and R. Govindarajan. Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors. In *Proc. of the '11 Conference on Programming Language Design and Implementation*, pages 152–163, 2011. 49

[80] K. Psarris, X. Kong, and D. Klappholz. The direction vector I test. *IEEE Journal of Parallel Distributed Systems*, 4(11):1280–1290, 1993. 93

[81] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time paralleliza-

tion of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160, 1999. 71

[82] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proc. of the 2010 International Conference on Supercomputing*, pages 137–146, 2010. 51

[83] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener. Programming with relaxed synchronization. In *Proc. of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, pages 41–50, 2012. 181

[84] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proc. of the 2006 International Conference on Supercomputing*, pages 324–334, 2006. 99, 143, 144, 180, 181

[85] M. Rinard. Parallel synchronization-free approximate data structure construction. In *Proc. of the 5th USENIX Workshop on Hot Topics in Parallelism*, pages 1–8, 2012. 181

[86] M. C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Proc. of the 22nd annual ACM SIGPLAN conference on Object-Oriented Systems and applications*, pages 369–386, 2007. 99, 138, 139, 143, 144, 180, 181

[87] D. Roger, U. Assarsson, and N. Holzschuch. Efficient stream reduction on the GPU.

In *Proc. of the 1st Workshop on General Purpose Processing on Graphics Processing Units*, pages 1–4, 2007. 51, 67, 160

[88] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009. 106

[89] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 2008. 1, 15

[90] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W. mei W. Hwu. Program optimization space pruning for a multithreaded GPU. In *Proc. of the 2008 International Symposium on Code Generation and Optimization*, pages 195–204, 2008. 53

[91] M. Samadi, A. Hormati, J. Lee, and S. Mahlke. Paragon: collaborative speculative loop execution on gpu and cpu. In *Proc. of the 5th Workshop on General Purpose Processing on Graphics Processing Units*, pages 64–73, 2012. 7

[92] M. Samadi, A. Hormati, J. Lee, and S. Mahlke. Leveraging GPUs using cooperative loop speculation. *ACM Transactions on Architecture and Code Optimization*, page To Appear, 2014. 7, 133, 134, 138, 139

[93] M. Samadi, A. Hormati, M. Mehrara, J. Lee, and S. Mahlke. Adaptive input-aware compilation for graphics engines. In *Proc. of the '12 Conference on Programming Language Design and Implementation*, pages 13–22, 2012. 6, 95, 140, 141

[94] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–50, 2014. 9, 131, 138, 140

[95] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. SAGE: Self-tuning approximation for graphics engines. In *Proc. of the 46th Annual International Symposium on Microarchitecture*, pages 13–24, 2013. 8, 144, 149, 168, 169, 179, 180

[96] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: approximate data types for safe and general low-power computation. *Proc. of the '11 Conference on Programming Language Design and Implementation*, 46(6):164–174, June 2011. 99, 115, 122, 138, 140, 143, 144, 168, 180, 182

[97] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. pages 25–36, 2013. 140, 182

[98] J. Sartori and R. Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. In *IEEE Transactions on on Multimedia*, pages 427–428, 2012. 138, 139, 181

[99] H. Sheikh, M. Sabir, and A. Bovik. A statistical evaluation of recent full reference image quality assessment algorithms. *IEEE Transactions on Image Processing*, 15(11):3440–3451, 2006. 122, 168

[100] M. Shindler, A. Wong, and A. W. Meyerson. Fast and accurate k-means for large

datasets. In *Advances in Neural Information Processing Systems24*, pages 2375–2383, 2011. 98

[101] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 124–134, 2011. 138

[102] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005. 55, 93

[103] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 16–30, 2008. 111

[104] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky. OptiML: an implicitly parallel domain specific language for machine learning. In *Proc. of the 28th International Conference on Machine learning*, pages 609–616, 2011. 120, 140, 166

[105] A. C. Tamhane and D. D. Dunlop. *Statistics and Data Analysis*. Prentice-Hall, 2000. 109

[106] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *12th International Conference on Architectural*

Support for Programming Languages and Operating Systems, pages 325–335, 2006. 15, 50, 56, 94

[107] O. Temam. A defect-tolerant accelerator for emerging high-performance applications. In *Proc. of the 39th Annual International Symposium on Computer Architecture*, pages 356–367, 2012. 182

[108] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proc. of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 365–376, 2010. 28, 50

[109] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 2002 International Conference on Compiler Construction*, pages 179–196, 2002. 20, 40

[110] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 330–341, 2008. 55, 83, 93

[111] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *Proc. of the 2009 International Symposium on Code Generation and Optimization*, pages 200–209, 2009. 49

[112] H. A. van der Vorst. BI-CGSTAB: a fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing*, 13:631–644, Mar. 1992. 39

[113] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. In *Proc. of the 46th Annual International Symposium on Microarchitecture*, pages 1–12, 2013. 182

[114] H. Volos, A. Welc, A.-R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. Nepaltm: design and implementation of nested parallelism for transactional memory systems. In *Proc. of the 23nd European conference on Object-Oriented Programming*, pages 123–147, 2009. 55, 93

[115] R. P. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994. 93

[116] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995. 36, 65, 67

[117] M. Wolfe. Implementing the PGI accelerator model. In *Proc. of the 3rd Workshop on General Purpose Processing on Graphics Processing Units*, pages 43–50, 2010. 15, 50, 56, 94

[118] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Proc. of the 2010 IEEE Symposium on Performance Analysis of Systems and Software*, pages 235–246, 2010. 152, 175, 176

[119] X.-L. Wu, N. Obeid, and W.-M. Hwu. Exploiting more parallelism from applications having generalized reductions on GPU architectures. In *Proc. of the 2010 10th*

International Conference on Computers and Information Technology, pages 1175–1180, 2010. 51, 160

[120] S. Xiao and W. chun Feng. Inter-block gpu communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2010. 54, 95

[121] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *Proc. of the '10 Conference on Programming Language Design and Implementation*, pages 86–97, 2010. 15, 49

[122] S. zee Ueng, M. Lathara, S. S. Baghsorkhi, and W. mei W. Hwu. CUDA-Lite: Reducing GPU programming complexity. In *Proc. of the 21st Workshop on Languages and Compilers for Parallel Computing*, pages 1–15, 2008. 15, 49, 50

[123] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 369–380, 2011. 54, 95

[124] H. S. Z.Wang, L. Cormack, and A. Bovik. Live image quality assessment database release 2, 2006. http://live.ece.utexas.edu/research/quality. 122