STATIC ANALYSIS FOR GPU PROGRAM PERFORMANCE

Nimit Singhania

A DISSERTATION
in
Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in
Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2018

Supervisor of Dissertation

Rajeev Alur
Zisman Family Professor of
Computer and Information Science

Co-Supervisor of Dissertation

Joseph Devietti
Assistant Professor of Computer and
Information Science

Graduate Group Chairperson

Rajeev Alur
Zisman Family Professor of
Computer and Information Science

Dissertation Committee
Stephan Zdancewic, Professor of Computer and Information Science
Mayur Naik, Associate Professor of Computer and Information Science
Sampath Kannan, Henry Salvatori Professor of Computer and Information Science
Vinod Grover, Director of Engineering, NVIDIA

To my parents.

ABSTRACT

STATIC ANALYSIS FOR GPU PROGRAM PERFORMANCE

Nimit Singhania

Rajeev Alur
Joseph Devietti

GPUs have become popular due to their high computational power. Data scientists rely on GPUs to process loads of data being generated by their systems. From a humble beginning as a graphics accelerator for arcade games, they have become essential compute units in many important applications. The programming infrastructure for GPU programs is still rudimentary and the GPU programmer needs to understand the intricacies of GPU architecture, tune various execution parameters and optimize parts of the program using low-level primitives. GPU compilers are still far from the automation provided by CPU compilers where the programmer is often oblivious of the details of the underlying architecture.

In this work, we present *light-weight formal* approaches to improve performance of *general* GPU programs. This enables our tools to be fast, correct and accessible to everyone. We present three works. First, we present a compile-time analysis to identify uncoalesced accesses in GPU programs. Uncoalesced accesses are a well-documented memory access pattern that leads to poor performance. Second, we present an analysis to verify block-size independence of GPU programs. Block-size is an execution parameter that must be tuned to optimally utilize GPU resources. We present a static analysis to verify block-size independence for *synchronization-free* GPU programs and ensure that modifying block-size does not break program functionality. Finally, we present a compile-time optimization to leverage cache reuse in GPU to improve performance of GPU programs. GPUs often abandon cache reuse-based performance improvement in favor of thread-level parallelism, where a large number of threads are executed to hide latency of memory and compute operations.

We define a compile-time analysis to identify programs with significant intra-thread locality and little inter-thread locality, where cache resue is useful, and a transformation to modify block-size which indirectly influences the hardware thread-scheduler to improve cache utilization.

We have implemented the above approaches in LLVM and evaluate them on various benchmarks. The uncoalesced access analysis identifies 111 accesses, the block-size independence analysis verifies 35 block-size independent kernels and the cache reuse optimization improves performance by an average 1.3$\times$ on two Nvidia GPUs. The approaches are fast and finish within few seconds for most programs.

This thesis is based on material drawn from the following publications:

1. Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija and Nimit Singhania. GPUDrano: Detecting Uncoalesced Accesses in GPU Programs. In *Proceedings of the 29th International Conference on Computer-Aided Verification*, CAV 2017, pages 507-525. Springer, 2017.

2. Rajeev Alur, Joseph Devietti and Nimit Singhania. Block-Size Independence for GPU Programs. In *Proceedings of the 25th International Symposium on Static Analysis*, SAS 2018, pages 107-126. Springer, 2018.

3. Rajeev Alur, Joseph Devietti and Nimit Singhania. Static Analysis for Cache Reuse Optimization in GPU Programs. In submission, 2018.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

A Formal Perspective to GPU Computing

Graphics Processing Units, or GPUs, have emerged as a highly-parallel compute platform. Today, they are being used to accelerate numerous data-intensive applications. They have enabled large-scale machine learning and with that the resurgence of Artificial Intelligence. They are powering almost every critical scientific application, and the fastest supercomputers rely on GPUs to accelerate their computation. They have evolved into important compute units sharing the center stage with the CPU (Central Processing Unit).

This evolution has occurred over the course of time. The first GPUs were a set of specialized circuits used to accelerate graphics for arcade games. They had fixed functionality and their only task was to generate display for 2D games. Generating display required computing values for a large number of pixels. The CPU took a long time to perform this computation, which was a hurdle for real-time rendering of display necessary for games. Therefore, GPUs emerged as a solution to this problem. Gradually, they were enriched to support 3D graphics and later various graphics operations like geometry transformations and pixel-shading functions. They still provided a fixed functionality and were not programmable. The first GPU with programmmable shading capability was Nvidia GeForce 3 (NV20) that allowed programmers to write small custom shaders which were run for every pixel in the image or every vertex in the graphics model. Over time, they started supporting more complex shaders with lengthy loops and floating point computation, and eventually evolved into flexible data-parallel graphics engines. Since graphics computation involved a large number of matrix and vector operations, researchers realized their computational abilities and used the shader engine to accelerate operations like matrix multiplication and LU factorization [24]. The computation still had to be mapped on to graphics primitives, and hence, languages like Sh [47], Brook [9] and Accelerator [65] emerged to provide a general-purpose interface to GPUs. The languages still performed redundant graphics computation, and therefore, native support for general-purpose GPU computing developed with languages like CUDA [51] by Nvidia and OpenCL [63] by the Khronos Group. Ever since, the use of GPUs for general-purpose computing has been growing exponentially.

GPUs present a data-parallel compute platform, where a large number of threads exe-

cute the same sequential program. This is useful when the same computation needs to be performed on a large amount of data, as happens to be the case in graphics applications where the same computation is repeated on a large number of pixels and vertices. With frequency scaling no longer possible, CPUs have reached their limits on single-thread performance, and the only way to achieve significant performance is through parallelism. GPUs with their inherently parallel model of computation provide a viable alternative. With burgeoning amounts of data everywhere, GPUs are well-suited to process the data with their high throughput and energy efficiency.

An interesting observation is that, unlike CPUs, which emerged from theoretical models like lambda calculus and Turing machines through decades of research, GPUs started out as an engineering solution to display graphics, and only recently have gained interest from researchers. This means there is a general lack of formal models backing the GPU architecture and the languages supported by GPUs. The model of computation presented by GPUs needs to be studied carefully to understand its capabilities and limitations. The use of GPUs for serious applications also entails the need to ensure that GPU programs produce correct and reliable results. This requires a new set of formal tools and techniques which can ensure correctness of GPU computation. Unlike CPU programs where the programmer can be completely oblivious of the underlying platform due to the sophisticated compiler and hardware technology developed over the years, the GPU compiler and hardware are still catching up and the programmer needs to be aware of the subtleties of the GPU architecture and must carefully tune various execution parameters to achieve significant performance gain. In general, there is need for better language, tool and hardware support to make GPU programming as easy, reliable and performant as programming for CPUs.

A closely related architecture to GPUs is that of vector processors, where a single thread simultaneously operates on *vectors* or arrays of data, also known as the Single Instruction Multiple Data (SIMD) model [19]. This is unlike the GPU model where a large number of threads operate on scalar data. Vector processors have been studied well in the literature, and a large number of automatic optimizations have been developed to optimize sequential code using vector instructions. The vector processors, however, present instruction-level parallelism, where each instruction is executed in parallel for thousands of data elements. In comparison, GPUs present parallelism at the granularity of whole programs, where the complete program is executed in parallel. The distinction between the two architectures hasn't been studied formally, however, and performance and programmability implications of the two architectures need to be explored further.

Significant research has been devoted to advancing tool support for GPUs in recent years. A large number of compiler optimization frameworks have been explored [12, 76, 64, 66, 75, 6, 23, 31]. These frameworks take unoptimized CUDA or OpenCL programs and optimize performance through various techniques like transforming data-layout for arrays and data-structures, explicitly caching data into on-chip fast memory, transforming thread-geometry etc. A large number of auto-tuning techniques have emerged to enable programmers to automatically identify best execution configuration [69, 45, 38, 13, 46, 74]. These techniques empirically explore the space of values for various parameters like the data-layout representation, the thread-coarsening factor and the launch parameters like block-size and grid-size.

Various domain-specific languages have emerged with GPU back-ends that automatically generate code for GPUs, for example Halide [55] for image-processing pipelines and TensorFlow [2] for machine learning. Many generic languages are also being used to generate GPU code, including legacy languages like OpenMP [41] or C [68] and new languages like pragma-based OpenACC [71]. There has been some research to improve the GPU hardware for better utilizing the memory subsystem and for balancing thread-level parallelism with cache and memory utilization [57, 50, 34, 61, 39, 56, 32]. Finally, there has been some research on verifying correctness of GPU programs for issues like data-races and barrier divergence [44, 43, 8, 21]. These techniques employ static and dynamic analyses to detect concurrency issues and in some cases verify absence of bugs. Despite this body of research, a large number of challenges remain, and we have only taken first steps towards building robust tools for efficient GPU programming.

1.1 Challenges with GPU Programming

Almost every field of science today is overwhelmed with data, ranging from medical imaging to cosmological simulations [35]. With increase in sophistication, richer data is produced at a faster pace by the modern systems, and hence, GPU acceleration is necessary to analyze the data. Also, the needs of the computation are specific to each domain, which existing GPU-based frameworks do not cater to. The existing frameworks themselves are often complex and not well-maintained, which leads domain scientists to write their own GPU programs. Efficient GPU programming is an art, however, and non-expert programmers find it difficult to write correct and efficient GPU programs.

There are multiple reasons for this. First, there are a large number of parameters that need to be tuned in order to get good performance. The space of values is large and the the combinations of values that need to be explored is overwhelming. While auto-tuning frameworks have been developed to cater to this problem, not all choices for parameters are correct, and subtle errors can enter the program when the parameters are modified. These errors can be difficult to debug and none of the existing tools ensure correctness for parameter tuning.

Second, tuning parameters may not be sufficient to achieve good performance. GPUs are sensitive to how programs interface with the memory subsystem. If the programmer is not careful and uses poor memory access patterns, the memory subsystem may get congested and take a long time to respond to program's requests, bringing down the overall performance. While this problem has been well-documented, only way to identify such issues is to actually execute the program and observe performance bottlenecks at run-time. There is a need for tools that report performance issues at compile-time or as the programmer writes the GPU program.

Third, while the compiler technology has made significant strides at automatically optimizing CPU programs, this is not true for GPU programs. The GPU programmer has to manually rewrite parts of the program to improve performance which leads to code with poor readability and maintainability. There are two primary problems with automatic optimization for GPUs. First, non-trivial transformations like loop-tiling or loop-reordering are

necessary to improve performance which are difficult to implement correctly. Many existing prototypes have tried implementing these transformations without successful transition to practice. It is also not clear when these techniques are applicable and how the programmer can direct tools to implement necessary optimizations. Second, while a few formal approaches have been developed to systematically optimize GPU programs, they require the programs to be well-structured with well-defined loop bounds and array indices. This prevents their applicability to general GPU programs, which are easier to write for non-expert programmers. Hence, we need tools for robust optimization of general GPU programs.

Finally, there is a need for newer languages with better support from hardware to make GPU programming robust and efficient at a larger scale. We need new languages with semantics that are easy to reason with, both manually and automatically. We need richer controls from the hardware, for the compiler to robustly optimize GPU programs. Lastly, we need program representations which are easy to analyze and manipulate within the compiler.

1.2 Structured Parallelism

A key feature that distinguishes GPU programming model from other concurrency models is that there is *structure* or regularity in the parallelism offered by the model. While there are a large number of threads executing, the threads execute the same sequential program and their executions are often correlated, especially for *regular* programs where the control flow and memory access patterns are independent of input data. For instance, the programs often operate on matrices and vectors, where each thread is assigned a row, column or a cell in the matrix and it performs the same operation. Further, there is structure within the execution of threads, when loops are present in the program and each iteration of the loop performs a similar operation. For example, when a thread is assigned a row in a matrix, it iterates over the cells of the row while repeating the same operation. This repeating pattern within a thread and across threads is very useful for compile-time analysis and optimization.

Our work *formally* identifies this repeating pattern for *general* GPU programs, both across the execution of multiple threads and within the execution of a single thread. While identifying the pattern exactly is difficult, an over-approximation is good enough to establish various properties of the programs, for instance, whether a program uses poorly performing memory access patterns (also known as uncoalesced accesses), or whether modifying an execution parameter is valid and leads to a correct program. We use *abstraction-based* static analyses to identify these properties. We have defined a model for GPU programs in Chapter 2, where we formalize the full-functional behavior of GPU programs while making reasonable simplifications. We use this model to formally describe our analyses. We define abstract domains that identify the information required by the analyses and associate an abstract state and abstract semantics with each analysis to describe how the analyses execute on a GPU program. We rely on the formal model to reason about the correctness of the analyses and to ensure all corner cases are covered. Further, the abstraction-based approach helps filter out the relevant information from program state, which enables the analysis to be light-weight and to scale to large programs. We have also developed a framework to implement such analyses in LLVM, a popular state-of-the-art open-source compiler, which we

describe in Chapter 3.

We use the framework to define three static analyses. The first static analysis determines whether a GPU program consists of poorly performing memory access patterns. In a GPU program, threads execute instructions often in lock-step. When threads execute access to global memory locations in the program, instead of fetching the memory location for each thread individually, the GPU hardware groups together locations accessed by multiple threads into a few memory transactions. If threads access memory locations within the same memory block, a single transaction is sufficient. However, if threads access *distant* locations, multiple transactions are necessary, which leads to poor performance. Such accesses are referred to as *uncoalesced* accesses. Our static analysis identifies uncoalesced accesses by computing the *stride* or the distance between memory locations accessed by consecutive threads. For coalesced accesses, where the accesses lie within a single memory block, the stride is often constant with value one and is computable at compile-time. Therefore, our analysis identifies all such accesses with a stride of value at most one, and reports the remaining accesses as uncoalesced. Our analysis performs fairly well with a false positive rate of 38%, while identifying 111 uncoalesced acccesses in Rodinia, an academic benchmark suite of GPU programs. We describe this analysis further in Chapter 4.

Our second analysis checks if modifying an execution parameter *block-size* is valid. The threads in a GPU program are organized in a two-level hierarchy, where a group of threads forms a *thread-block* and group of thread-blocks forms the overall *thread-grid*. The *block-size* determines the number of threads per thread-block and is often adjusted to improve the overall utilization of cores within the GPU. Modifying block-size is however not valid, if the computation performed by the program depends on the value of block-size and changing the value leads to incorrect results. Our analysis checks if any of the block-size dependent values, including block-size itself, *flow* into the final result computed by each thread. If not, the program is *block-size independent* and modifying block-size is valid. The block-size dependent values are often used in a regular pattern which allows the analysis to prove block-size dependence for a large number of programs. We identify 35 procedures to be block-size independent in Nvidia's SDK samples. We describe this analysis further in Chapter 5.

Our final analysis determines the potential to utilize hardware cache to automatically improve performance of a GPU program. The analysis identifies accesses that benefit from cache reuse across iterations of a loop within the execution of a thread. It determines the distance between memory locations accessed by an access in consecutive iterations of a loop, and if the distance is small, the access is considered *cache reusable*. A program with one or more cache reusable accesses can benefit from cache reuse. On-chip cache reuse to utilize locality within each thread is often relinquished for thread-level parallelism, where the simultaneous execution of a large number of threads leads to cache contention, and therefore, poor utilization. However, if there is significant locality within a thread, identified by the presence of cache reusable accesses in the program, and very little locality across threads, identified by the presence of uncoalesced accesses, then giving up thread-level parallelism for improved cache reuse is beneficial. We have implemented this optimization in LLVM and observe an average $1.3\times$ speedup in benchmark performance against the base compiler without our optimization. We describe the analysis to determine cache reuse and our optimization

to improve cache utilization in Chapter 6.

Finally, this thesis makes the following contributions to the state-of-the-art in analysis and optimization of GPU programs:

(1) We present the first *light-weight formal* analyses and optimization for *general* GPU programs. Light-weight techniques are necessary to scale to large programs and to allow usage at compile-time. A formal approach to define analyses and optimizations helps reason about correctness. Applicability to general GPU programs makes these techniques widely applicable.

(2) We have implemented the approach in LLVM, a popular open-source compiler. The implementation is modular and well-developed and tested on various benchmarks. The build is completely automated and the tool is easy to install on a new system.

1.3 Organization

The thesis is organized as follows. Chapter 2 presents a background on GPU programming and a formal model for functional behavior of GPU programs. Chapter 3 presents a framework to define abstraction-based static analyses. Chapter 4 presents our static analysis to detect uncoalesced accesses. Chapter 5 presents an analysis to verify block-size independence of GPU programs. Chapter 6 presents an analysis to identify intra-thread cache reuse across loop iterations within a GPU program and our optimization to improve cache utilization. Finally, Chapter 7 presents some concluding remarks.

Chapter 2

Fundamentals of GPU Programming

GPUs have emerged as a popular data-parallel architecture and programming languages like CUDA [51] and OpenCL [63] have been developed to support these devices. GPUs present an interesting computation model where a large number of threads execute the same sequential program and the execution for threads is distinguished via unique identifiers assigned to each thread. This leads to a lot of regularity in the parallelism offered by the model, which makes GPU programs amenable to analysis and performance optimization, unlike general concurrent programs.

GPU computing has emerged only recently, and therefore, the programming models have not been studied formally. There is previous work on modeling correctness issues like data-races in GPU programs [8, 43, 44]. There is, however, scope to model other features like performance and correctness of transformations. A formal model forms the basis for correct and efficient analyses and transformations. Therefore, good formal models for GPU programs are required, and in our work we formalize one such model.

This chapter presents the fundamentals of GPU programming. We first present a brief background on the programming model and the architecture for GPUs (Section 2.1). We describe how GPU programs are written in popular languages like CUDA and OpenCL and how the programs are executed on GPUs. We illustrate this further using an example (Section 2.2), and use the example to introduce our formalization. We then describe our formal model that captures the functional and performance behavior of GPU programs (Section 2.3). The model serves as the basis for various analyses which we describe in subsequent chapters. Even otherwise, the model precisely represents various aspects of GPU programming, which is useful in understanding the general programming model. We finally present some limitations of our formalization (Section 2.4).

2.1 Background

We present a brief background on GPUs and their programming model. GPUs present a highly *data-parallel* architecture, where a large number of threads execute the same sequence of instructions. This architecture is useful for applications where the same compu-

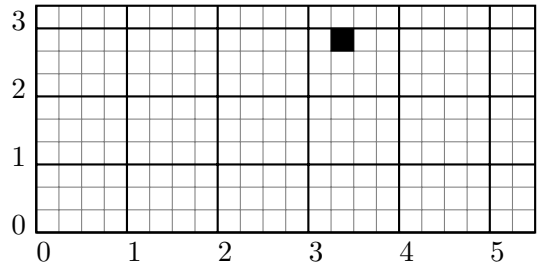


Figure 2.1: An example 2-dimensional thread-grid with 22×10 total threads (grid-size) and 4×3 threads per block (block-size). Each solid block represents a thread-block, while each cell represents a thread. The values represent the block-id along respective dimensions. The darkened cell corresponds to a thread with block-id $(3, 2)$ and thread-id $(1, 2)$.

tation needs to be performed on a large amount of data. The data-parallel nature of the GPU architecture enables numerous optimizations which help GPU applications scale, while traditional multi-threaded applications often do not scale with increase in the amount of resources. Two prominent optimizations include (a) *parallelism-based latency hiding*, where the execution of multiple parallel threads is used to hide the latency of compute and memory operations, and (b) *memory access coalescing*, where memory accesses by multiple threads are coalesced or grouped together into a few memory transactions, significantly reducing the overall access latency. We present the GPU programming model in Section 2.1.1, followed by some insight into the GPU architecture in Section 2.1.2. Note that, we follow the CUDA terminology here. The other popular model, OpenCL, uses a slightly different terminology, though the concepts are similar in both models.

2.1.1 GPU Programming Model

GPUs follow a Single Instruction Multiple Threads (SIMT) programming model, where a large number of threads execute the same sequence of instructions. The threads are organized in a two-level hierarchy: a group of threads form a thread-block and a set of thread-blocks form a thread-grid. Each thread in the grid is assigned a *thread-id* that uniquely identifies the thread within the block, and a *block-id* that uniquely identifies its block within the grid. Further, the grid can be multi-dimensional with up to three dimensions, and each thread is assigned a multi-dimensional thread-id and a block-id. These dimensions map naturally to 2D and 3D images which represent a traditional application domain for GPU computing. The overall size of the grid and blocks is defined by multi-dimensional vectors *grid-size* and *block-size*, respectively, both of which are initialized when the execution of the GPU program is initiated. Figure 2.1 presents an example thread-grid.

The sequence of instructions executed by the threads in a GPU program is often called the *kernel*. The kernel is essentially a sequential method with special read-only variables for thread-id (tid), block-id (bid), grid-size (gdim) and block-size (bdim). The variables tid and bid are instantiated with a distinct value for each thread, which helps distinguish the

```
__global__ void Fan1(float *M, float *A, int N, int i) {
  int x = tid.x + bid.x * bdim.x;
  if(x >= N-i-1) return;
  M[N*(x+i+1)+i] = A[N*(x+i+1)+i] / A[N*i+i];
}
__global__ void Fan2(float *M, float *A, float *B, int N, int i) {
  int x = tid.x + bid.x * bdim.x;
  int y = tid.y + bid.y * bdim.y;
  if(x >= N-i-1) return;
  if(y >= N-i) return;
  A[N*(x+i+1)+(y+i)] -= M[N*(x+i+1)+i] * A[N*i+(y+i)];
  if(y == 0) {
    B[x+i+1] -= M[N*(x+i+1)+(y+i)] * B[i];
  }
}
int ForwardSub(float *m, float *a, float *b, int N) {
  cudaMemcpy(M, m, N*N, 'CPUtoGPU');
  cudaMemcpy(A, a, N*N, 'CPUtoGPU');
  cudaMemcpy(B, b, N, 'CPUtoGPU');
  for (int i=0; i<(N-1); i++) {
    Fan1<<<N, 512>>>(M, A, N, i);
    Fan2<<<(N, N), (512, 512)>>>(M, A, B, N, i);
  }
  cudaMemcpy(m, M, N*N, 'GPUtoCPU');
  cudaMemcpy(a, A, N*N, 'GPUtoCPU');
  cudaMemcpy(b, B, N, 'GPUtoCPU');
}
```

Figure 2.2: Forward Substitution subroutine in a Gaussian elimination application. Methods `Fan1()` and `Fan2()` are GPU kernels and execute on GPU, while method `ForwardSub()` runs on the CPU. Here, tid, bid and bdim refer to thread-id, block-id and block-size, respectively. Method `cudaMemcpy()` copies data between CPU and GPU.

execution of threads. The kernel is embedded as a subroutine within the GPU application and is launched via a function call to the kernel from sequential code that runs on the CPU. Figure 2.2 shows an example GPU application. Note that, methods `Fan1()` and `Fan2()` representing GPU kernels are called from the `ForwardSub()` method which runs on the CPU. Further, the method calls to `Fan1` and `Fan2` are qualified by parameters within `<<<...>>>` enclosure which represent the grid-size and block-size for the grid of threads for which the GPU kernel is executed. We describe more details about the program in Section 2.2. Lastly, the GPU kernels can make calls to other GPU kernels like any other function call. A distinction is often made between *global* kernels, which are called only from CPU code, and the *device* kernels, which are called only from GPU code *i.e.* other global and device kernels.

Each thread in a GPU program has access to different memory spaces. The variables in the kernel are qualified by the memory space they reside in, and the threads access the

corresponding memory space for each variable to fetch/write data to the variable. We briefly describe the different memory spaces here:

- **Local memory:** This memory space is private to each thread and the data stored in this space is not visible to other threads. Each thread has a separate copy of variables in this space, and hence, the same local variable in different threads might store different values. This space is primarily used for performing local computations within each thread.
- **Shared memory:** This memory space is shared across threads within a thread-block and a single copy of variables in this space is visible to all threads within the block. However, each block has a separate copy of variables. This space is used to share data between threads within a block. It provides a fast look-up and is often used as a user-managed cache.
- **Global memory:** This memory space is shared across all threads in the grid and a single copy of variables is visible to all threads. The space acts as an interface between the CPU and GPU, and stores the data on which a GPU kernel performs computation. During a GPU kernel execution, the data is first copied from CPU memory into global memory, then the kernel execution is initiated and operations are performed on the data in global memory, and after the execution is completed, the output data, representing the result of the kernel execution, is copied back from global memory to CPU memory.
- **Constant memory:** This is a special memory space that stores read-only data. The space is similar to global memory and a single copy of variables is visible to all threads. However, since it is read-only, it benefits greatly from caching and provides a fast read-only alternative to global memory.

We next describe some common primitives used to synchronize threads and share data between them. We describe two kinds of primitives: *barriers* that ensure that all threads in a block reach a program point before they continue further execution, and *atomic* operations that ensure exclusive access to a memory location by each thread for the duration of the operation.

- **Synchronization barrier:** The most commonly used synchronization primitive is the `__syncthreads()` barrier. This barrier ensures that all threads within a block synchronize at the barrier, that is threads in the block wait at the barrier until all threads reach the barrier and all shared and global memory accesses by threads in the block, prior to reaching the barrier, are completed. This is useful to split the execution into phases, where threads share data with each other across phases. Further, it is possible to customize the barrier to ensure completion of only shared memory or global memory accesses on certain GPUs.
- **Atomic operations:** GPUs also provide atomics like atomic add, subtract, minimum, maximum, and compare and swap. The atomic operation takes a local varable and a shared or global location as parameters, and atomically performs the operation on

the location, such that no other thread has access to the location while the operation is being performed for one thread. This is useful when multiple threads update the same shared/global variable, say a counter, and it prevents threads from interfering with each other.

When synchronization primitives are not used properly, we end up with synchronization issues like data-races and barrier divergence. A *data-race* occurs when two threads read or write to the same shared or global memory location, with one of the accesses being a write, and the two accesses are not separated via a synchronization primitive like `__syncthreads()` barrier or an atomic operation. A data-race between threads leads to a non-deterministic execution of the program, such that no interleaving of threads defines the state after the execution of the program. Another common synchronization issue is that of barrier divergence. A *barrier divergence* occurs when only few of the threads within a block reach a `__syncthreads()` barrier. This can occur if the barrier is present within a conditional and the conditional evaluates to true for only few of the threads within a block. The threads that reach the barrier are stuck waiting for other threads to reach the barrier, which leads to a deadlock. It is important to ascertain the program does not have data-races or barrier divergence, to ensure correct execution and termination of the program. There is prior work on detecting such problems in GPU programs [8, 43, 44]. Hence, we focus on performance and correctness of transformations and do not address synchronization issues in our model.

2.1.2 GPU Architecture

We now briefly describe the GPU architecture. We first describe the compute infrastructure of a GPU and how it executes the kernel for threads and blocks in the grid. We then describe the memory subsystem of GPUs and how data in different memory spaces is stored and accessed.

Compute infrastructure. The compute infrastructure of a GPU consists of a set of cores, also known as streaming multiprocessors (SMs). Each SM consists of different processing units like integer and floating point adders and multipliers, and often a number of these. The SMs operate independently from each other, and thus each block in the grid is mapped to a single SM since the blocks execute independently. The processing units within an SM operate synchronously, on the other hand. Hence, the operations performed by threads within a thread-block are mapped to these units. The threads within a block are split into groups of threads, called *warps*. A warp is a group of fixed number of threads (usually 32 or 64) with contiguous thread-ids. The threads in the block are scheduled in units of warps and operations for all threads within a warp are issued simultaneously to the processing units. Further, the order in which different warps execute different instructions is determined by the scheduler on the SM. Finally, when a warp encounters a conditional statement in the GPU program and threads take different branches, the execution of the branches is *serialized*, where first few threads in the warp execute the first branch while the remaining threads are idle, and then the remaining threads execute the second branch. If a large number of branches are encountered by a warp, multiple serializations occur which leads to a significant slowdown in performance. This issue is also known as *warp divergence*.

Memory subsystem. We now describe the memory subsystem of a GPU. It first consists

of a large on-chip register file that stores the local variables in a GPU program. The register file provides fast access to data. It next consists of an on-chip shared memory that stores the shared memory variables in the program. Both register file and shared memory are limited resources and determine the number of blocks that can be simultaneously scheduled on an SM. It then consists of an off-chip DRAM that stores data for global memory variables in the program. The DRAM provides high-latency high-bandwidth access. Therefore, it takes a long time to access any data from DRAM, however a large amount of data can be fetched in a single transaction. The memory subsystem utilizes this feature by *coalescing* accesses by threads in a warp and fetching data in a single transaction to global memory. When this is not feasible and more than one transaction is necessary, such global memory accesses are referred to as *uncoalesced accesses*. Finally, both DRAM and shared memory are organized in banks. If simultaneous accesses from different SMs to DRAM and different threads to shared memory lie in the same bank, there is a *bank-conflict* which leads to serialization of accesses.

2.2 Example: Gaussian Elimination

We now briefly illustrate GPU programming via the example program in Figure 2.2. The example shows a Forward Substitution subroutine, `ForwardSub()`, in a Gaussian elimination application. The subroutine takes matrices A and M of size N $\times$ N and a vector B of size N, and performs forward substitution on A and B via M. It runs on CPU and makes calls to GPU kernels `Fan1` and `Fan2` to perform the substitution operation. It runs N iterations where every ith iteration performs substitution for the ith row in A and B. In the ith iteration, the subroutine first launches kernel `Fan1` to compute values in the ith column of matrix M. It launches N threads, where a thread with id x computes the value in cell $(x + i + 1, i)$ in M. Note that A and M are flattened matrices, and hence, a cell (p, q) refers the location $(N \cdot p + q)$ in the flattened array. Next, the subroutine launches kernel `Fan2` with N $\times$ N threads which updates A and B via values in the ith column of M. It uses a two-dimensional thread-grid, where the thread with id (x, y) updates cell $(x + i + 1, y + i)$ in A. Also, threads with id $(x, 0)$ update the $(x + i + 1)$th cell in B. Lastly, the procedure `cudaMemcpy()` copies data between CPU and GPU memory. Initially before the kernels are launched, it copies A, M and B into GPU memory, and subsequently after substitution is completed, it copies data back into CPU memory.

We next focus on the execution of kernel `Fan2`. This is a *global* kernel and can be launched from the CPU code. It is launched with a grid consisting of N $\times$ N threads with each block consisting of 512×512 threads. Matrices A and M and vector B reside in global memory and are shared across all threads in the grid, whereas variables N, i, x and y reside in local memory. The values x and y refer to *global-id*s for each thread which are unique to each thread in the thread-grid and are computed via the thread-id tid, block-id bid, and block-dim bdim along X and Y dimensions, respectively. Further, each thread in the kernel executes the same kernel, and the executions of different threads are distinguished via their global-ids. Each thread updates location $A[x + i + 1, y + i]$ through values in $A[x + i + 1, y + i]$, $M[x + i + 1, i]$ and $A[i, y + i]$. We note that each thread in the grid updates a distinct location in A which is not read by any other thread. This is because, each thread reads and updates

```
void Fan2(int N, int i, float M[N, N], float A[N, N], float B[N]) {
  int x = tid[0] + bid[0] * bdim[0];
  int y = tid[1] + bid[1] * bdim[1];
  if(x < N-i-1 && y < N-i) {
    A[x+i+1, y+i] -= M[x+i+1, i] * A[i, y+i];
    if(y == 0) {
      B[x+i+1] -= M[x+i+1, y+i] * B[i];
    }
  }
}
```

Figure 2.3: Formal model for `Fan2` kernel in Figure 2.2.

location $A[x + i + 1, y + i]$, where $(x + i + 1, y + i)$ is unique to each thread, since x and y have distinct values in each thread. Further, threads read a common location $A[i, y + i]$, however no thread updates this location. Also, none of the threads write to matrix M. We can similarly observe that each thread updates a distinct location in B not read by other threads. Hence, the kernel is *data-race free*.

Formal Model. Given that `Fan2` is data-race free, we now present a formal model for the kernel. The model is shown in Figure 2.3. In the model, we represent one kernel at a time. We assume the kernel consists only of scalars and arrays, and does not have pointer and structure variables. Therefore, we convert pointers A, M and B into array variables. We represent A and M by two-dimensional arrays. We further assume that scalar variables reside in local memory, while array variables in global memory. Hence, N, i, x and y are local variables, while A, M and B are global variables. We replace structure representation for variables tid, bid and bdim with vectors. Hence, tid_0 represents $\mathsf{tid}.x$ while tid_1 represents $\mathsf{tid}.y$. We assume the program does not have `return` or `break` statements, and consists only of local assignments, reads from and writes to global/shared variables, conditionals, loops, or function calls to other kernels. Lastly, the size of each global array is expressed via constants or local variables passed as parameters. Therefore, sizes for A, B and M are defined via the local variable N. We discuss more details about representing kernels in our formal model in Section 2.3.1. We use this representation for all examples in future.

Execution Model. We next present an execution model for the model in Figure 2.3. Given the grid-size $\vec{N} = (N, N)$ and block-size $\vec{B} = (512, 512)$, we maintain a state σ that maps local and global variables to their values. The grid consists of $N \times N$ threads divided into blocks of size 512×512. In our execution model, we execute one block at a time, since the blocks execute independently. Hence, we only maintain state for variables within one block at a time, and σ stores a single copy of shared and global variables and a copy of local variables for each thread in the block. We assume lock-step execution for threads where all threads execute instructions simultaneously. Therefore, in kernel `Fan2`, we first perform the computation of global-id x for all threads in a block, followed by the computation of global-id y for all threads, followed by the execution of the conditional. During the execution of conditionals, we *serialize* the execution of branches, where threads for which the first branch

is true execute the branch while the other threads wait, and then the remaining threads execute the second branch. Hence in Figure 2.3, the threads for which the condition $(x < N-i-1 \bigwedge y < N-i)$ is false, wait for remaining threads to finish executing the conditional. Similarly, among threads that execute the conditional, threads for which $(y == 0)$ is false, wait for remaining threads to complete their execution. Further, since the kernel is data-race free, the order of execution of threads and blocks does not matter and they can execute in arbitrary order. For determinism, we assume the threads execute in lexicographic order of their thread-ids and block-ids. More details about the execution model can be found in Section 2.3.2.

Memory Performance Model. Finally, we look at the performance of accesses to global arrays. Since the global variables reside in an off-chip DRAM, they have a high-latency access. This is however compensated by *coalescing* or grouping together accesses by multiple threads in the grid. In particular, the threads execute in groups of threads called warps, and the accesses by threads in a warp are coalesced together into as few transactions as possible. A warp usually consists of 32 threads with lexicographically consecutive thread-ids. We first consider the access to location $A[x + i + 1, y + i]$. We note that since the size of array is $N \times N$ and the global memory is allocated linearly, the location $(x + i + 1, y + i)$ is flattened into the location $N \cdot (x + i + 1) + (y + i)$. The threads in a warp consist of consecutive thread-ids, where the value tid_0 differs by 1 across consecutive threads, while the remaining ids tid_1, bid_0 and bid_1 are equal. Hence, value of x differs by 1 and value of y is equal across consecutive threads in the warp, and therefore the locations accessed in the array differ by value N (since a thread with $x = k$ and $y = l$ accesses location $N \cdot (k + i + 1) + (l + i)$, while a thread with $x = (k + 1)$ and $y = l$ accesses location $N \cdot (k + i + 2) + (l + i)$). Assuming size of `float` is 4 bytes, the accessed locations differ by $4N$ bytes in memory. A transaction to global memory fetches 128 bytes of contiguous data. Therefore, for a reasonably large value of N, the accessed locations by threads in a warp are spread *far apart* in memory and a separate transaction is necessary for access by each thread. On the other hand, consecutive threads access consecutive locations for the access $B[x + i + 1]$, and hence, all data can be accessed in a single transaction to global memory, which is much more efficient. We model the performance of memory accesses in more detail in Section 2.3.3.

2.3 Formal Model

This section describes a formal model for GPU programs. The model concretizes different aspects of a GPU program. First, it defines the programming model which describes how the behavior of a large number of threads is expressed in the model. Next, it defines an execution model wherein each instruction in the program is assigned a precise semantics and the semantics for individual instructions are composed together to obtain the semantics of executing the program for the grid of threads. Finally, it presents a modelling for the performance of different types of memory accesses. Each of these components are associated with mathematical definitions that help avoid ambiguity in the description. The formal model however comes with the cost of abstracting away many of the underlying hardware/software features, which makes the model an inaccurate description of the underlying system.

"All models are wrong, but some are useful". As this quote by George Box suggests, despite being inaccurate, the formal model is useful for reasoning about the program. More importantly, it is useful in defining automated *program analyses*, which can scan the program and check specific properties of the program like whether the program runs correctly or finishes in reasonable time. The formal model provides the basis for reasoning about the correctness of an analysis and ensuring that the property checked by the analysis is accurate with respect to the model.

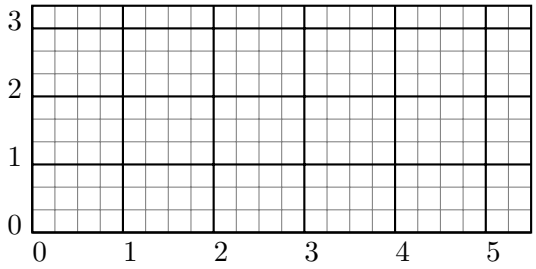
We have designed the model keeping three analyses in mind: The first analysis identifies all global memory accesses that perform poorly; the second analysis checks if changing the block-size preserves the functionality of the program; the final analysis identifies global memory accesses within loops that benefit from cache reuse. We describe these analyses in subsequent chapters. To capture these analyses, however, we model the functional and performance behavior of GPU programs. We eschew any correctness concerns like data races and barrier divergence, and the model assumes that the program, in the first place, is devoid of any such issues. This allows the model to be simple and easy to reason with, yet captures necessary behavior required by the analyses.

The model consists of the following components: the programming model which defines the core constructs of a GPU program including the thread hierarchy, different types of variables and the syntax for the GPU program (Section 2.3.1); the execution model that defines how each instruction is executed and the order in which different instructions are executed (Section 2.3.2); and finally, the memory performance model that defines how memory accesses are performed in the model and the cost for different types of accesses (Section 2.3.3).

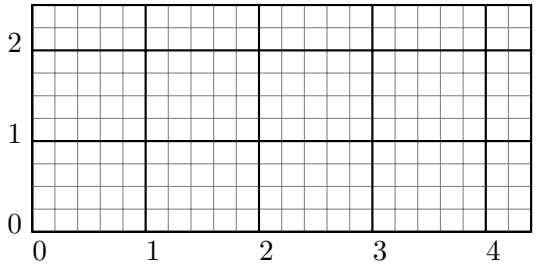
2.3.1 Programming Model

We first define a simple programming model that captures the key constructs of GPU programming. A GPU program $\mathcal{P}$ is essentially a sequence of instructions K executed by a grid of threads $\mathcal{G}$. The grid is defined via a *grid-size* $\vec{\mathsf{N}}$ which defines the overall size of the grid, and a *block-size* $\vec{\mathsf{B}}$ which defines the size of each block in the grid. Each thread τ in the grid is identified by a unique *thread-id*, tid, and a *block-id*, bid, which are used to distinguish the thread's execution from that of other threads. Further, the thread is provided access to the grid-size $\vec{\mathsf{N}}$ and block-size $\vec{\mathsf{B}}$ via variables gdim and bdim, respectively. We define these quantities further while defining the grid. The grid can be multi-dimensional, and hence, the ids are multi-dimensional vectors of size d, which is useful for applications with multi-dimensional arrays like matrices and images. Each thread has access to local variables (represented by set V_{L}), shared variables ($V_{\mathcal{S}}$), constant variables (V_{C}) and the global variables (V_{G}), representing variables in corresponding address-spaces.

Formally, the program $\mathcal{P}$ is represented by the tuple $\langle \mathsf{d}, V_{\mathrm{L}}, V_{\mathcal{S}}, V_{\mathrm{G}}, V_{\mathrm{C}}, \mathsf{K} \rangle$. The set V_{C} further consists of sizes gdim and bdim and the ids tid and bid. Let l be a local variable in V_{L}, c be a constant variable in V_{C} and ν be an $(\mathfrak{n}+1)$-dimension shared or global array variable in $V_{\mathcal{S}} \cup V_{\mathrm{G}}$. For simplicity, we assume that the local and constant variables are scalars, while the shared and global variables are multi-dimensional arrays. Let E be a computable expression on local variables. The sequence of instructions K, also known as the *kernel*, is



(a) $\vec{N} = 22 \times 10$, $\vec{B} = 4 \times 3$.



(b) $\vec{N} = 22 \times 10$, $\vec{B} = 5 \times 4$.

Figure 2.4: The example illustrates the independence of block-size $\vec{B}$ and grid-size $\vec{N}$. For a fixed value of $\vec{N}$, block-size $\vec{B}$ can take arbitrary values, where the left-over threads are accommodated in the last few blocks.

defined by the following grammar:

$$S := AS \mid \textbf{if } \langle \texttt{test} \rangle \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } \langle \texttt{test} \rangle \textbf{ do } S \mid \texttt{\_\_syncthreads()} \mid f(l_0, l_1, \ldots, l_m, v_0, v_1, \ldots, v_n, \langle \texttt{isAtomic} \rangle) \mid S_1; S_2.$$

$$AS := l \leftarrow c \mid l \leftarrow E(l_0, l_1, \ldots, l_n) \mid l \leftarrow v[l_0, l_1, \ldots, l_n] \mid v[l_0, l_1, \ldots, l_n] \leftarrow l.$$

The statement S is either an assignment statement, a conditional where statement S_1 is executed if the boolean condition $\langle \texttt{test} \rangle$ is true and otherwise S_2 is executed, a loop, a synchronization primitive for threads, a call to a function f with local variables, shared and global variables as parameters and $\langle \texttt{isAtomic} \rangle$ set to true if f is atomic, or a composition of individual statements. An assignment statement AS is either a constant assignment, a local assignment, a shared/global array read, or a shared/global array write.

Thread Grid. We next define the grid of threads $\mathcal{G}$ for which the program is executed. The grid $\mathcal{G}$ consists of a two-level hierarchy of threads: the grid is sub-divided into a set of blocks, and each block is then sub-divided into a group of threads. To define the grid, we first associate a *grid-size* with the grid which defines the overall size of the grid. Since the grid can be multi-dimensional, it is represented by a d-dimensional vector $\vec{N}$, where $\vec{N}_i$

represents the number of threads along the ith dimension of the grid. Next, we associate a *block-size* which defines the size of blocks in the grid. It is represented by a d-dimensional vector $\vec{B}$, where $\vec{B}_i$ represents the number of threads in a block along the ith grid-dimension. Given $\vec{N}$ and $\vec{B}$, the grid $\mathcal{G}(\vec{N}, \vec{B})$ is first split into a set of blocks $\mathcal{B} = \{b_{\vec{0}}, \ldots b_{\vec{k}}\}$, where the index for each block represents its position within the grid. Next, each block $b_{\vec{i}}$ is split into a set of threads $\mathcal{T}(b_{\vec{i}}) = \{\tau_{\vec{0}}, \ldots, \tau_{\vec{m}}\}$, where the index for each thread gives its local position within the block. There might be a mismatch between the grid-size $\vec{N}$ and the block-size $\vec{B}$, such that if N_i is not a perfect multiple of B_i for some dimension i, then the last block along the ith dimension consists of fewer than B_i threads, to constrain the total number of threads to N_i. For example, Figure 2.1 in the beginning of the chapter presents a 2-dimensional grid. Here, $\vec{N} = (22, 10)$ and $\vec{B} = (4, 3)$. Each solid block in the figure represents a block of threads, while each cell represents a thread. The last block along each dimension has fewer threads than other blocks to overcome the mismatch between block-size and the grid-size, since neither dimension of $\vec{N}$ is a perfect multiple of the corresponding dimension in $\vec{B}$. It is useful to note that block-size $\vec{B}$ is independent of grid-size $\vec{N}$, and hence, $\vec{B}$ can be set to arbitrary value for the same value of $\vec{N}$, as shown in Figure 2.4.

We next define the thread-id tid and the block-id bid for each thread τ in the grid. The thread-id refers to the position of the thread within the block, whereas the block-id refers to the position of thread's block within the grid. Both tid and bid are d-dimensional vectors with the initial position being the $\vec{0}$ vector. In the example grid in Figure 2.1, the block-ids range between $(0, 0)$ to $(5, 3)$, while the thread-ids range between $(0, 0)$ and $(3, 2)$. The darkened cell in the grid corresponds to a thread with bid $(3, 2)$ and tid $(1, 2)$.

2.3.2 Execution Model

The execution model describes how each statement in the GPU program is executed, and how the execution of statements for threads and blocks is composed together to get the overall execution of a GPU program $\mathcal{P}$ for a grid of threads $\mathcal{G}(\vec{N}, \vec{B})$. We define the execution model using a top-down approach, where the execution of the program for the overall grid is defined via the execution for individual blocks and the execution for blocks via the execution for threads.

To define the execution for the grid, we first define a *global state* σ^G, that maps global variables to their values. Let $\mathcal{V}$ be the set of values. Then, the global state is a function that maps global variables in V_G to type-consistent values in $\mathcal{V}$, *i.e.* $V_G \to \mathcal{V}$. Let $[\![K]\!](\sigma^G, \vec{N}, \vec{B})$ represent the execution of kernel K for the grid of threads $\mathcal{G}(\vec{N}, \vec{B})$, represented by sizes $\vec{N}$ and $\vec{B}$. The execution takes in an initial global state σ^G and returns the updated global state. We define the execution for the grid, $[\![K]\!](\sigma^G, \vec{N}, \vec{B})$, via the execution for individual blocks in the grid. The model assumes that the blocks execute independently of each other. This is also true in practice, since the blocks are often allocated on independent GPU cores. Further, it assumes that the program has no data-races. Hence, the order of execution for the blocks does not matter and the execution for the grid is defined by a sequential composition of the execution for individual blocks. Let $[\![S]\!](\sigma^G, b)$ represent the execution of statement S for a block b in the grid. Let $\mathcal{B}$ be the list of blocks in the grid. The execution for the grid is

defined by the following equations:

$$[\![K]\!](\sigma^G, \vec{N}, \vec{B}) = [\![K]\!](\sigma^G, \mathcal{B}).$$

$$\text{for all } S, \Gamma, [\![S]\!](\sigma^G, \Gamma) = [\![S]\!]([\![S]\!](\sigma^G, b), \Gamma'), \text{ where } \Gamma \equiv \{b, \Gamma'\}.$$

Note that Γ is a list, where b is the first element while Γ' represents the remaining list. To keep things simple, we further assume that the composition is performed in the order of block-ids, *i.e.* the block with block-id $\vec{0}$ is executed first, followed by the remaining blocks. This cannonical order helps achieve a deterministic execution for the program.

We next define the execution of a statement S for a block b, $[\![S]\!](\sigma_0^G, b)$. We assume that all threads execute in *lock-step*. Note the lock-step assumption is only valid if the original program does not have data-races. Otherwise, the execution in this model may not correspond to the actual execution in practice. However, this assumption greatly simplifies the semantics. The set of threads in the block is given by $\mathcal{T}(b)$. We define the state σ for the execution of threads in the block as follows. The state consists of a *local state* σ^L, that maps local variables in each thread to their respective values, and hence is a function $V_L \times \mathcal{T}(b) \to \mathcal{V}$. It also consists of the *shared state* σ^S, that maps the shared variables to their values and represents the function $V_S \to \mathcal{V}$. Finally, the state consists of the global state σ^G mapping global variables to their values. Let $[\![S]\!](\sigma, \Pi)$ represent the execution of statement S for the set of threads Π. The execution starts in state σ and returns the updated state. During the execution for the block, the local and shared variables are initialized to undefined values ($\perp$), while the global state is set to σ_0^G. Note that the updated state consists of all variables, whereas the execution for the block returns only the global state. Hence, after the execution for the threads complete, we discard the shared and local state and return the updated global state. Let p^G be a function that projects out a state on to the set of global variables. Let $\sigma_\perp^L$ and $\sigma_\perp^S$ represent local and shared state with variables initialized to unknown value $\perp$. The execution for the block is defined as:

$$[\![S]\!](\sigma_0^G, b) = p^G([\![S]\!](\sigma, \mathcal{T}(b))), \text{ where } \sigma = \sigma_0^G \cup \sigma_\perp^S \cup \sigma_\perp^L.$$

We next define the execution of a statement for a set of threads, $[\![S]\!](\sigma, \Pi)$. We define this by structural induction on S:

- **Assignments:** We first define the execution of an assignment statement for a single thread τ, $[\![AS]\!](\sigma, \tau)$. Let σ' be the resulting state. We consider different types of assignments.
  - $l \leftarrow c$: The value of the local variable l gets updated to that of c. Hence, $\sigma'(l, \tau) = c(\tau)$, where $c(\tau)$ is the value of constant c in thread τ. Note that the values of remaining variables remains unchanged. Hence, $\sigma'(l', \tau) = \sigma(l', \tau)$ for all $l' \neq l$.
  - $l \leftarrow E(l_0, l_1, \ldots, l_n)$: The value of local variable l is updated to the computation of E on values of variables $l_0, \ldots, l_n$. Hence, $\sigma'(l, \tau) = E(\sigma(l_0, \tau), \ldots, \sigma(l_n, \tau))$.
  - $l \leftarrow v[l_0, l_1, \ldots, l_n]$: The value of local variable l is updated to the value of array v at location $\vec{p} = (\sigma(l_0, \tau), \ldots, \sigma(l_n, \tau))$. Hence, $\sigma'(l, \tau) = \sigma(v)(\vec{p})$.

– $\nu[l_0, l_1, \ldots, l_n] \leftarrow l$: The value of array ν at location $\vec{p} = (\sigma(l_0, \tau), \ldots, \sigma(l_n, \tau))$ is updated to the value $\sigma(l, \tau)$. Hence, $\sigma'(\nu)(\vec{p}) = \sigma(l, \tau)$. Further, the values at remaining locations remains unchanged. Hence, for all $\vec{q} \neq \vec{p}, \sigma'(\nu)(\vec{q}) = \sigma(\nu)(\vec{q})$.

Given the semantics of executing an assignment for a single thread, we define the execution of the assignment for a set of threads Π, $[\![AS]\!](\sigma, \Pi)$, by composing the execution for individual threads. As described earlier, the model assumes the program does not have data races, and therefore, the order of execution does not matter. We assume the threads execute in the order of their thread-ids to have deterministic execution. The execution is defined as:

$$[\![AS]\!](\sigma, \Pi) = [\![AS]\!]([\![AS]\!](\sigma, \tau), \Pi'), \text{ where } \Pi \equiv \{\tau, \Pi'\}.$$

- **Sequences:** We next define the execution for a sequence of statements $[S_1; S_2]$ for a set of threads Π. The semantics first execute S_1 for all threads in Π, followed by the execution of S_2 for all threads. This ensures the lock-step execution of threads. Formally, the execution is defined as:

$$[\![S_1; S_2]\!](\sigma, \Pi) = [\![S_2]\!]([\![S_1]\!](\sigma, \Pi), \Pi).$$

- **Conditionals:** We next define the execution for conditionals [**if** $\langle$`test`$\rangle$ **then** S_1 **else** S_2]. We use the value of a local boolean variable l to represent the $\langle$`test`$\rangle$ condition. The semantics consist of executing the first branch S_1 for all threads where the condition is true *i.e.* $\sigma(l, \tau) =$ true, followed by the execution of branch S_2 for the remaining threads. Note that we *serialize* the execution of branches, where a few threads first execute the first branch while the remaining threads are idle and then the remaining threads execute the second branch. Again due to the assumption about data-race freedom, the order of execution for branches does not matter. Formally, the execution is defined as:

$$[\![\textbf{if } l \textbf{ then } S_1 \textbf{ else } S_2]\!](\sigma, \Pi) = [\![S_2]\!]([\![S_1]\!](\sigma, \Pi_1), \Pi \backslash \Pi_1), \\ \text{where } \Pi_1 = \{\tau \in \Pi : \sigma(l, \tau) = \text{true}\}.$$

- **Loops:** We next define the execution for loops [**while** $\langle$`test`$\rangle$ **do** S]. The semantics consists of repeating the loop body S, until the $\langle$`test`$\rangle$ condition (represented by a local boolean variable l) becomes false for all threads in Π. Similar to conditionals, we serialize the execution of different iterations of the loop. Let there exist a sequence of states $\sigma_0, \sigma_1, \ldots, \sigma_n$ and a sequence of sets of threads $\Pi_0, \Pi_1, \ldots, \Pi_n$, such that:

$$\sigma_0 = \sigma,\ \Pi_0 = \{\tau \in \Pi : \sigma(l, \tau) = \text{true}\}.$$

$$\sigma_{i+1} = [\![S]\!](\sigma_i, \Pi_i),\ \Pi_{i+1} = \{\tau \in \Pi_i : \sigma_{i+1}(l, \tau) = \text{true}\}, \text{ for all } i \geqslant 0.$$

$$\Pi_n = \{\}.$$

Then the state after the execution of the loop is given by σ_n:

$$[\![\textbf{while } l \textbf{ do } S]\!](\sigma, \Pi) = \sigma_n.$$

Note that we assume the loop is terminating, and the non-termination of loops is not taken into consideration. There is a rich body of work on proving termination of loops for sequential programs and proving termination of loops for GPU programs is beyond the scope of this thesis.

- **Function Calls:** We next consider function calls $f(l_0, \ldots, l_n, v_0, \ldots, v_n, \langle \texttt{isAtomic} \rangle)$. To address function calls, we define a map ψ, that given a function f, a state σ, and a set of threads Π returns the updated state after lock-step execution of the function for all threads. If f is a non-atomic function, then the semantics are:

$$[\![f(l_0, \ldots, l_n, v_0, \ldots, v_n, \text{false})]\!](\sigma, \Pi) = \psi(f, \sigma, \Pi).$$

If f is atomic, then each thread executes the function separately and the semantics are:

$$[\![f(l_0, \ldots, l_n, v_0, \ldots, v_n, \text{true})]\!](\sigma, \Pi) = \\ [\![f(l_0, \ldots, l_n, v_0, \ldots, v_n, \text{true})]\!](\psi(f, \sigma, \{\tau\}), \Pi'), \text{ where } \Pi \equiv \{\tau, \Pi'\}.$$

- **Synchronization:** The statement `__syncthreads()` synchronizes threads within a block and allows sharing of data (in shared or global variables or both) between threads without leading to a data-race. Due to the lock-step execution followed in our model, we don't need special semantics for this statement. Hence, the execution of the statement returns the same state:

$$[\![\texttt{\_\_syncthreads()}]\!](\sigma, \Pi) = \sigma.$$

This completes our description of the execution model. We next define a memory performance model to describe how memory accesses are executed in our model.

2.3.3 Memory Performance Model

We describe how accesses to different memory spaces are executed in our model. We model these accesses primarily to understand their performance. Hence, we do not model correctness aspects and assume the accesses are *sequentially consistent*, which ensures the execution order of the memory operations is preserved and is the same across all threads. To describe the memory model, we define an address map ϕ that stores information about the size of each shared/global array, and given an array g and a location $\vec{p}$, returns the flattened absolute location $\phi(g, \vec{p})$ for the array. We further define an element-size map ξ that maps each array to the size of elements in the array.

Moreover, the accesses are carried out in groups of threads called warps. A *warp* w is group of a fixed number of threads, consisting of threads from a block b with lexicographically contiguous thread-ids. Let the number of threads in a warp be n_w. Formally, we define

a warp as:

$$w(b,k) = \{\tau \in \mathcal{T}(b) : kn_w \leqslant \phi(\tau) < (k+1)n_w\},$$
$$\text{where } \phi(\tau) = \sum_i (\mathsf{bid}_i(\tau).\mathsf{bdim}_i(\tau) + \mathsf{tid}_i(\tau)) \prod_{j=0}^{i} \mathsf{gdim}_j(\tau).$$

We now describe how different types of memory accesses are executed in our model, and their associated cost of execution.

2.3.3.1 Global Memory Accesses

We describe the execution of global memory accesses in our memory model. Consider a global memory read $[l \leftarrow g[l_0, l_1, \ldots, l_n]]$ or write $[g[l_0, l_1, \ldots, l_n] \leftarrow l]$ in state σ by a set of threads Π. The accessed location is $(l_0, l_1, \ldots, l_n)$ in a global array g. The access is performed in units of warps. During the execution of an access AS for a warp w, we first collect the set of addresses accessed by the warp. For each thread τ in $(\Pi \cap w)$, we first flatten the accessed location $\vec{p}(\tau) = (\sigma(l_0,\tau), \sigma(l_1,\tau), \ldots, \sigma(l_n,\tau))$ via the address map ϕ into an absolute location $\phi(g, \vec{p}(\tau))$. Then we fetch the element at this location consisting of $\xi(g)$ bytes. Overall, the set of accessed addresses for the warp is:

$$\mathcal{A}(AS, \sigma, \Pi, w) = \bigcup_{\tau \in \Pi \cap w} [\phi(g, \vec{p}(\tau)), (\phi(g, \vec{p}(\tau)) + \xi(g) - 1)],$$
$$\text{where } \vec{p}(\tau) = (\sigma(l_0,\tau), \ldots, \sigma(l_n,\tau)).$$

GPUs have a large bandwidth and can access multiple contiguous addresses in one transaction to global memory. Let the global memory bandwidth be η bytes. A single transaction can fetch addresses $[k\eta, k\eta + \eta - 1]$ for all $k \in \mathbb{Z}^+$. Therefore, all addresses a, a' in $\mathcal{A}(AS, \sigma, \Pi, w)$ such that $\lfloor a/\eta \rfloor = \lfloor a'/\eta \rfloor$ are fetched in the same transaction. Hence, the *cost* of executing the access for a warp, $N_{\mathcal{A}}(AS, \sigma, \Pi, w)$, equals the number of transactions required to fetch addresses $\mathcal{A}(AS, \sigma, \Pi, w)$, which is equal to the number of unique elements in the set $\{\lfloor a/\eta \rfloor : a \in \mathcal{A}(AS, \sigma, \Pi, w)\}$.

2.3.3.2 Shared Memory Accesses

We next consider shared memory accesses. Similar to global memory accesses, the shared memory accesses are executed in units of warps. The organization of data in shared memory is different however. The data in shared memory is organized in banks, where the addresses are distributed into banks in a round-robin fashion. Therefore, consecutive addresses lie in consecutive banks; however, the addresses at distance equal to the number of banks lie in the same bank. Access to different banks can be executed in parallel, whereas the accesses to the same bank are serialized. Let the number of banks be κ. For an access AS by a warp w, the set of accesses mapped to the ith bank is given by the set $\mathcal{A}_i$:

$$\mathcal{A}_i = \{a \in \mathcal{A}(AS, \sigma, \Pi, w) : a \bmod \kappa = i\}.$$

The number of transactions required is given by the maximum number of accesses mapped to the same bank *i.e.* $\mathrm{N}_{\mathcal{A}}(AS, \sigma, \Pi, w) = \max_{0 \leqslant \mathtt{i} < \kappa} |\mathcal{A}_{\mathtt{i}}|$, which is the required cost of execution.

2.3.3.3 Local and Constant Memory Accesses

The local memory accesses are accessed via on-chip registers which take constant time to access, for all threads and all addresses. Constant memory accesses, on the other hand, benefit significantly from caching, and therefore, are almost constant time. Hence, we do not model the performance of these accesses, and assume the cost of these accesses to be constant.

2.4 Limitations of Formalization

This section discusses various limitations of our formalization. We describe various features of popular GPU programming models that we do not capture in our formal model. We also compare our execution model against the execution on real GPUs. We finally identify features of GPU memory model that we do not address in our formal model. We note that these simplifications not just make it easy to understand and reason about the model, they also help simplify the analyses and optimizations that build on the semantics of the model, which in-turn improves the scalability and precision of the analyses and the performance of the optimizations.

Our programming model captures a small subset of the features supported by real GPU programming models, which is sufficient to describe most GPU programs in practice. The small subset of features allows focus on semantic properties of GPU programs, and prevents distraction from various syntactic features available for the ease of programming. We describe the limitations of our model here:

- *Grid Size.* In our formalization, we represent grid-size as the total number of threads $\vec{\mathrm{N}}$ along each grid dimension. The OpenCL programming model follows this convention. However, CUDA uses the *number of blocks* $\vec{\mathrm{N}}_{\mathtt{b}}$ along each grid dimension to represent grid-size. Using number of blocks for grid-size restricts the possible choices for number of threads to the multiples of block-size. Our representation, on the other hand, provides greater flexibility in deciding the number of threads in the grid. Hence, we represent grid-size by the total number of threads $\vec{\mathrm{N}}$ along each grid-dimension.

- *Atomics and Synchronization Primitives.* Our formalization has two primitives for synchronization. First, we support atomic functions via the $\langle$`isAtomic`$\rangle$ bit in function calls. Second, we provide a `__syncthreads()` barrier to synchronize threads at the barrier. These are also the two primary forms of synchronization supported by CUDA and OpenCL. However, they allow customizing these primitives with fine-grained features like the set of variables (shared or global) and the set of threads (not limited to all threads within a block) for which the synchronization is performed. The primitives can be further customized with consistency requirements that enforce the order in

which memory operations are performed. These customizations help improve the fine-grained performance of GPU programs. In our work, we focus on orthogonal features like memory coalescing and cache reuse to improve performance, and simplifying the model reduces the complexity of proving correctness of the analyses. Therefore, we do not model the fine-grained customizations.

- *Structures and Pointers.* Our formal model only considers scalars and arrays, while the general models CUDA and OpenCL also support structures and pointers. GPU programs often operate on simple data structures and rarely involve aliasing of pointers. The key insights for arrays carry over to structures and pointers as well, and for simplicity, we omit them from our model. We do address these in our implementation for the analyses and discuss relevant issues in subsequent chapters.

- *Nested Parallelism.* GPU programming models have recently added support for nested parallelism, where a kernel can launch another kernel with its own set of threads. Most GPU programs do not require this feature, and given its recent emergence, it is not in common use yet. Hence, we do not support this feature in our model.

We next describe our simplifications to the execution model. We make two main simplifications. First, we assume the threads execute in lock-step within a block. Second, we assume that the programs are devoid of data-races and barrier divergence. These assumptions do not always hold. However, the assumptions greatly simplify semantics and also the analyses and optimizations that build on these semantics.

- *Lock-step Execution.* In our execution model, we assume threads within a block execute in lock-step. However, in practice, the threads often execute in groups of warps. As long as the programs are data-race free, the semantics of lock-step execution are equivalent to that in practice. As a consequence, our analyses and optimizations for GPU programs are oblivious of the order in which different threads execute instructions, which helps them scale to large programs.

- *Data-races and Barrier Divergence.* Our execution model assumes GPU programs to be free of data races and barrier divergence. In our work, we focus on analyzing and improving performance of GPU programs, and concurrency bugs like data-races and barrier divergence are not a primary concern. Furthermore, these issues have been tackled in previous work [8, 43, 44]. Hence, we do not model these issues in our model. Again, this is useful in scaling analyses, since they no longer need to address concurrency issues.

Finally, we assume memory operations to be sequentially consistent, where the order of memory operations is preserved and is same across threads. Most GPUs, on the other hand, support relaxed consistency and enforce memory ordering via *thread-fences*. First, this has correctness implications since a lack/improper-use of fences leads to data races. We however assume the programs to be devoid of data races and rely on existing tools to prove data-race freedom. Hence, this is not a concern in our model. Second, we currently do not use low-level transformations like replacing a costly fence with a cheaper fence to improve performance.

Therefore, modeling semantics of different types of fences is not required. Such a modeling would however help enable the low-level transformations and we leave this as future work.

2.5 Conclusion

This chapter presents a formal model for GPU programs. The model makes strategic choices like assuming the programs are data-race free and the threads execute in lock-step. While this restricts the set of programs covered by the model, it helps simplify the model which makes it easy to understand and reason about. Further, the analyses and the optimizations build on the formal model, since their correctness is reasoned via the model. Hence, this helps simplify the analyses and the optimizations themselves, which enables analyses to give precise results while being scalable and optimizations to achieve significant speedups. That said, there is significant scope to refine the model to support larger class of GPU programs and more fine-grained optimizations, which we will address in future work.

Chapter 3

Abstract Execution-based Static Analyses

Abstraction-based analysis is a popular way to define compile-time analyses. In an abstraction-based approach, an abstract domain is defined where each variable in the program is associated with an abstract value. Further, a new set of abstract semantics are assigned to the statements in the program. The analysis execution consists of initializing an abstract state where each variable is assigned an initial abstract value. The program is then executed using the abstract semantics defined for the analysis. Finally, the abstract trace is used to collect the desired information required by the analysis. There are multiple ways of defining an abstraction-based analysis. The first approach is popularly known as *Abstract Interpretation*, where the abstract values and the abstract semantics are an over-approximation or an under-approximation of the underlying concrete values and the concrete semantics. This approach ensures one-sided error with either soundness or completeness of the analysis. The second approach is *Symbolic Execution*, where the abstract value consists of either an actual concrete value or a symbolic value which represents all possible concrete values. The abstract semantics correspond to the concrete semantics. This approach is precise (sound and complete) but takes a long time to finish. Finally, the last approach is that *Abstract Execution*, where we do not impose any constraints on the abstract values and the abstract semantics. This approach does not immediately ensure soundness or completeness, however some additional reasoning can establish the guarantees provided by the analysis. We use this approach to define abstraction-based analyses in our work. The idea was first introduced by Sarbó [60] to speedup symbolic execution-based analyses. Recently, it has been popularly and widely used in computation of worst-case execution time (WCET) bounds for verification of real-time systems [29, 22, 28].

This chapter presents a generic abstract execution-based framework. We present an informal description of the framework where each idea is presented from the perspective of implementing it in practice. Broadly, the abstract execution framework consists of three components: (a) the *abstract domain*, which determines the abstract values tracked for each variable in the program and the overall information tracked in the abstract state (Section 3.1); (b) the *abstract semantics*, which assigns abstract semantics to each statement in the pro-

gram and determines how the abstract state is updated during the execution of a statement (Section 3.2); and (c) the *abstract execution engine*, that describes how the program is "abstractly" executed, in particular, the order in which different statements are executed and the merging of abstract states at join points (Section 3.3). Usually, the abstract execution engine or the execution strategy remains constant across analyses, though minor changes can help improve the analysis performance and precision. The analysis designer primarily needs to define the abstract values for variables, the abstract state, and the abstract semantics for each statement in the program. We have implemented the framework in LLVM compiler framework. We present an LLVM-based implementation of the framework in Section 3.4. We present some other abstraction-based approaches in Section 3.5 and conclude in Section 3.6.

3.1 Abstract Domain

An abstract domain is the core of any abstraction-based static analysis. It defines the information necessary for an analysis that must be filtered out from the program state. A good abstract domain is necessary for a precise and scalable analysis. Hence, it is important to carefully design the abstract domain while building a static analysis. The abstract domain also establishes the correspondence between concrete and abstract states, which is necessary to ascertain the correctness of the analysis.

3.1.1 Example: Divide-by-zero Error

To understand an abstract domain, let's consider the following example. We would like to design an analysis that detects divide-by-zero errors in a program at compile-time. Note that we want to ensure that any divide-by-zero error reported must correspond to an actual error, and thus, the analysis needs to be complete. A divide-by-zero error occurs when the divisor in a division operation has value 0. Hence, the analysis needs to check if any of the divisors in the program has a value 0. We assume for simplicity that all variables in the input state are initialized to non-zero values. Therefore, a variable gets the value 0 only if it is assigned value 0 in some program statement.

A simple abstract domain for the analysis tracks for each variable whether the variable has value 0 at a program location. The abstract domain tracks a bit of information for each variable at each program location, which is set to true if the variable has value 0, and otherwise it is set to false. Initially, all variables are assigned false, since they are non-zero. A variable is set to true if it is assigned the value 0 in some statement. Further, it is set to true if it is assigned an expression where all variables are set to true and hence have a value 0. We can similarly compute the abstract values in other scenarios. Given this information, the analysis checks if a divisor is assigned the abstract value true, and if so, there is a divide-by-zero error.

Now, consider the following program:

```
y = -x;
p = x + y;
r = x / p;
```

Assuming all variables are non-zero initially, our simple abstract domain identifies p to be non-zero. Both x and y are assigned false, and hence, p is also assigned false. However, y is the negation of x, and summing the two variables returns a value 0. Hence, p has a value 0. Therefore, the analysis misses these types of errors, and we define a refined abstract domain. We refine our abstract domain to track three values for each variable:

0 : the variable has value 0.

$-v$: the variable is negation of another variable v.

$\top$: otherwise.

The computation of abstract values for variables is performed similar to the simple abstract domain, except when a variable is assigned the negation of a non-zero variable v, we set its abstract value to $-v$. Note that both $-v$ and $\top$ correspond to non-zero values. Using this abstract domain, we can prove p to have a value 0 in the above program. Since x is assigned $\top$, y is assigned $-x$ and p is the sum of values x and $-x$, p is assigned the abstract value 0, and the analysis now identifies the divide-by-zero error in the next statement.

3.1.2 Abstract Values and Abstract State

Abstract values. An abstract domain defines a set of abstract values for each variable in the program. Each abstract value can be a constant or a function of values of other variables. For instance, in the simple abstract domain in Section 3.1.1, the set of abstract values corresponds to the set $\{\text{true}, \text{false}\}$, while for the refined abstract domain, the set is $\{0, -v, \top\}$. Further, different sets of abstract values can be defined for different types of variables.

We must be careful while using values of variables to define abstract values. If the value of a variable is used as an abstract value (like $-v$ in the refined abstract domain), then we must ensure that the variable is not modified after it is assigned as the abstract value. Otherwise, if it is modified after it is assigned as the abstract value, the abstract value is no longer valid and this leads to erroneous results. For instance, consider the following example:

```
y = -x;
x = 0;
p = x + y;
r = x / p;
```

Here, the abstract value $-x$ for variable y is invalid after x is assigned value 0, since y is non-zero whereas x is 0, and hence, adding the two values returns a non-zero value. This is often taken care of via a program representation where each variable has a *unique definition*, and it is never modified after it is first assigned a value. Such a representation is called a Single Static Assignment (SSA) [20] and is widely used in popular compilers like LLVM.

The abstract domain also defines a $join$ function on abstract values. Given two abstract values v_1 and v_2, the $join$ function returns an abstract value v that supersedes the individual abstract values, such that the properties which hold for both values also hold for the returned value. For instance, the $join$ function for the simple abstract domain returns true if both v_1 and v_2 are true, and false otherwise. This is because if one of the values is non-zero, the

$join$ of the two values is not guaranteed to be 0, and hence, it is conservatively assigned the abstract value false. Similarly, for the refined abstract domain, the $join$ of values is set to 0 if both values are 0, $-v$ if both values are $-v$ for some variable v, and $\top$ otherwise. An important point to note is that the abstract values have an implicit ordering with a greatest element $\top$ that supersedes the remaining elements (often called a upper semi-lattice). For example, in the refined domain $\top$ supersedes the other two elements. The $join$ function must ensure that the resulting abstract value is always *greater* than the individual values.

Abstract state. We next define an abstract state. An abstract state primarily consists of a map from variables in the program to their abstract values. It might further consist of information that abstracts other properties of the concrete state like the number of enabled threads. An abstract state is the primary object that flows through the program and is modified via the abstract semantics for statements.

We further define the $merge$ function for abstract states. The $merge$ of two abstract states returns a state where the value of each variable is the $join$ of the values in the individual states. Other abstract information in the states can be joined similarly into the merged state. The merging of states is essential to scale analyses. It helps replace execution of statements for multiple states with the execution for a single merged state. This is useful in removing redundant computation and helps scale the static analysis to large programs.

Merging of states, however, comes at the cost of losing precision, since the information contained in the merged state is often not as precise as that in individual states. This is because, we might lose information while performing the $join$ operation on the abstract values of a variable. For instance, in the refined abstract domain in Section 3.1.1, the $join$ of values 0 and $-v$ is set to $\top$, since the merged value is neither 0 nor the negation of a variable. Now before the join, we know that the variable is either 0 or the negation of some variable v, whereas the joined value only informs us that the variable is non-zero. This loss in information can affect the overall abstract execution of the program and lead to loss in precision in the analysis results.

3.2 Abstract Semantics

The user needs to define an abstract semantics for the analysis to execute a program in the abstract domain. The abstract semantics are defined for each statement in the program. The abstract semantics for a statement takes in an abstract state σ and returns the updated abstract state σ'. Given the abstract domain, the abstract semantics is often intuitive and easy to define. However, the user needs to be careful about the correctness of the semantics, and must ensure that the resulting abstract state after each statement is correct with respect to the abstract domain. We next describe how abstract semantics for different statements is defined.

Assignments. We first describe the abstract semantics for assignments. We consider three types of assignments. First, an assignment where a local variable is assigned an expression of other local variables. Here the user needs to define how the abstract values for

variables in the expression are combined together to produce the abstract value for the resultant variable. The user must ensure that the resulting abstract value is correct and covers all possible concrete values that could be assigned to the variable. For example, consider the addition operation $[\nu_0 \leftarrow \nu_1 + \nu_2]$ for the simple abstract domain in Section 3.1.1. If the abstract values for both ν_1 and ν_2 are set to true, and hence, the variables have a value 0, then ν_0 is assigned 0 and an abstract value true. However, if either one of ν_1 and ν_2 has an abstract value false, then the only information available is that the variable is non-zero. With this information, we can not conclude with certainity that ν_0 gets a value 0. Hence, we conclude that ν_0 is potentially non-zero, and assign it the abstract value false. In the refined abstract domain, however, the abstract values carry more information, which helps us handle the scenario where ν_1 is negation of ν_2.

We next consider reads to array variables $[\nu \leftarrow A[\nu_0, \ldots, \nu_n]]$. Here, the analysis needs to model the array A and how each *abstract index*, consisting of abstract values for variables $\nu_0, \ldots, \nu_n$, maps to a value in the array. One approach is to not model the array, and hence, always return the abstract value representing any value at the location, for example, $\top$ in the refined abstract domain in Section 3.1.1. An alternate approach is to map each abstract index to a $merge$ of all values at the abstract location and return this value during the read. Also, we might need to take aliasing into consideration, if the array variables can alias with each other. The appropriate modelling depends on the requirements of the analysis. We can similarly define writes to array variables depending on how arrays are modelled.

Conditionals. We next consider conditionals [**if** ν **then** S_1 **else** S_2]. The semantics for the conditional are defined using the semantics for S_1 and S_2. Unlike concrete execution, where the value of conditional ν is known, the abstract execution needs to consider both executions where ν is true and false respectively, and merge the results of the two executions to get the updated state. Given the initial abstract state σ, we propagate the abstract states σ_1 and σ_2 to S_1 and S_2, where σ_1 represents σ along with the condition that the value ν is true, while σ_2 represents σ when condition ν is false. One approach is to propagate the same initial state σ to both branches, though there is scope to improve the precison of the analysis by taking the condition into account while propagating states. Next, the updated states σ_1' and σ_2', after the execution of S_1 and S_2 respectively, are merged to produce the updated state after the conditional. We can use the $merge$ function defined in Section 3.1.2 to merge states or define a custom merge function, depending on the requirements of the analysis.

Loops. We next consider loops [**while** ν **do** S]. The semantics for loops are similar to conditionals, except the abstract execution for S is repeated until the updated state stabilizes. Similar to conditionals, the updated state in each iteration might execute the loop body S or might completely exit the loop. Hence, the user needs to define how the state before the loop is propagated to S and how the resulting state after S is merged with the initial state to produce the initial state for the next iteration of the loop. Note that the semantics must ensure that the initial state before the next iteration is same or *greater* than that for previous iteration and the state stabilizes after a finite number of iterations. Otherwise, since there is no bound on the number of loop iterations, the analysis may not terminate.

Sequences. We next consider sequences $[S_1; S_2]$. The semantics involve propagating the state obtained after the execution of S_1 to the execution of S_2.

Function calls. We next consider function calls. Different approaches are taken to handle function calls. The first approach is to analyze each function in isolation, and assume a fixed behavior for function calls irrespective of which function is called. This is also referred to as an *intraprocedural* analysis approach. Here the function calls return a conservative abstract state which consists of arbitrary values for some or all of the variables. This approach is useful if the program does not contain function calls, or the result of function calls is not required for the analysis.

We next consider *interprocedural* analysis approaches where function calls are handled with more precision. There are two common approaches here. The first approach is the *bottom-up* approach where the *callees* or the functions called by other functions are analyzed before the functions calling them or the *callers*, In this approach, each function is analyzed for a few or all calling contexts. A *calling context* assigns specific abstract values to each input parameter of the function. The analysis results are stored in a *callee-summary* for each function analyzed. When a function call is encountered, the callee-summary of the callee is used to get the desired result of the function call. This approach is useful, when the analysis of a function depends on the analysis of its callees.

The other approach is the *top-bottom* approach where the callers are analyzed before the callees. Here, a fixed behavior is assumed for function calls similar to the intra-procedural approach. However, the calling context or the initial values for the function parameters are constructed based on the callers of the function. Hence, the calling context for the function is constructed by *merging* all possible contexts in which the function is called. This is also referred to as the *caller-summary*. This approach is useful when the calling context greatly influences the analysis result and each function is called for a few calling contexts only. We look at examples of both bottom-up and top-bottom analyses in the subsequent chapters.

This concludes our discussion for different types of statements in the program. Note that the program representation used can constrain the allowed abstract semantics. For instance, the most commonly used representation is the *control flow graph* [3], where a program is represented as a graph of nodes. Each node is a *basic block* which is a sequence of assignments ending in a conditional or unconditional jump to one or more basic blocks. Conditionals and loops are broken down into groups of connected basic blocks. For this representation, therefore, it may not feasible to write a custom merge function for loops and conditionals, since the original structure is no longer available. The user needs to default to the $merge$ function, defined for merging abstract states, to merge states from alternate iterations or branches.

3.3 Abstract Execution Engine

The abstract execution engine determines how a program is executed in the abstract domain. As the underlying compiler and hardware platform determines how fast a program executes and how precise the results are, the execution strategy for abstract execution determines the precision of the analysis results and the running time of the analysis. Often there is a trade-off between the running time and the precision of the analysis, and one needs to balance the

Algorithm 3.1: Abstract Execution Engine

```
input  : Program S_0, Initial abstract state σ_0.
output: State before statement map, preStateMap,
        State after statement map, postStateMap.
worklist.push(S_0, σ_0);
while ¬worklist.empty() do
    (S, σ) ← GetNextExecutionUnit(worklist);
    if preStateMap[S] exists then σ ← MergeState(σ, preStateMap[S]);
    if σ = preStateMap[S] then break;
    preStateMap[S] ← σ;
    /* nextUnitList collects next units to be executed during the
       ExecuteStatement call.                                      */
    nextUnitList.clear();
    σ' ← ExecuteStatement(S, σ, nextUnitList);
    postStateMap[S] ← σ';
    for (S_i, σ_i) in nextUnitList do
        if worklist.hasStmt(S_i) then
            (S_i, σ'_i) ← worklist.findStmt(S_i);
            worklist.updateState(S_i, MergeState(σ_i, σ'_i));
        else
            worklist.push(S_i, σ_i);
        end
    end
end
```

two quantities while designing the analysis.

We present a specific execution strategy for the abstract execution engine in Algorithm 3.1. The algorithm takes as input, the program S_0 and an initial abstract state σ_0 in which S_0 is executed. It returns two statement-to-state maps: preStateMap and postStateMap. The map preStateMap maps each statement S to the state in which S was executed. Note that if S is executed multiple times, this corresponds to a $merge$ of all states in which S was executed. A $merge$ of two states σ and σ' returns a state that supersedes both individual states. This is useful when we want to process a single merged state instead of multiple individual states, and the execution of statement S for the merged state, therefore, encompasses the execution for each of the individual states. The other output, postStateMap, returns a map from each statement S to the state after the execution of S.

The algorithm takes three user-defined methods: `MergeState()`, that given two abstract states σ and σ', returns the $merge$ of the two states; `ExecuteStatement()`, that executes a statement S in an abstract state σ; and `GetNextExecutionUnit()`, that returns next execution unit, which is a pair of a statement S and the state σ in which S is executed. Given these user-defined methods, the algorithm works as follows. The algorithm keeps a list of execu-

tion units, worklist, with each unit consisting of a statement to be executed and the state in which it is executed. It initializes the list with statement S_0 and initial state σ_0. Next, it processes units in worklist until no units remain. It obtains the next unit to be processed from worklist, (S, σ), via the `GetNextExecutionUnit()` method call. This is a user-defined method, and the user can use different heuristics to define the function. For example, the method could return the first unit in worklist, or the next unit with most recently executed statement (which is beneficial for faster execution of loops).

On obtaining the next unit (S, σ), it first merges σ with the pre-execution state before S, preStateMap[S], if it exists. If the merged state σ is same as the previous state preStateMap[S], the state in which the execution begins remains unchanged, and hence, the execution itself is unchanged. Therefore, we skip processing the unit. Otherwise, we first update the map preStateMap[S] with the updated state before S. Then, we begin execution of S for the merged state σ via the method call `ExecuteStatement()`. The method is defined by the user and captures the abstract semantics necessary for abstract execution. The method call returns the state after the execution of the statement. During the execution, we maintain a buffer of units, nextUnitList, that gets populated with execution units which need to be processed next after the execution of S. The method call `ExecuteStatement()` is responsible for populating nextUnitList. Note we assume that for sequence of statements $[S_1; S_2]$, statement S_1 has access to the next statement S_2 in the sequence. Hence, after S_1 finishes its execution, it can populate nextUnitList with the execution unit (S_2, σ_1), where σ_1 is the state obtained after the execution of statement S_1. This is a reasonable assumption and most program representations provide access to the next statement to be executed.

After the execution for a statement completes, we add the execution units in nextUnitList to our worklist. We use an optimization to prevent the explosion of units, especially when the same statement needs to be processed for multiple states. This can happen in the control flow graph representation of programs, where the statement after a loop or a conditional is visited multiple times based on the path taken (consisting of different branches for conditionals and different number of iterations for loops). Hence, we *compress* the processing of the statement for multiple states into that for a single merged state. We do this compression in a demand-driven fashion. While inserting a unit (S_i, σ_i) from nextUnitList into worklist, we check if there already exists an execution unit for statement S_i in worklist. If there exists such a unit, say (S_i, σ_i'), then we merge the new unit with the existing unit by updating it to $(S_i, \texttt{MergeState}(\sigma_i, \sigma_i'))$. Otherwise, if no existing unit is found, we push the new unit (S_i, σ_i) into worklist.

Finally, the maps preStateMap and postStateMap provide an execution trace for the overall abstract execution. Note the method `MergeState()` is used to merge abstract states for multiple visits to a statement in the program. This merging of states is essential to scale the execution. Without merging, we may never terminate the execution and we might indefinitely cycle through a loop with a different initial state during each visit. Merging states is also necessary to prevent explosion of states due to a large number of feasible execution paths in a program. It is important to carefully implement this method and ensure that the merged state always supersedes the individual states being merged. Otherwise, the execution might not terminate or take a really long time to complete.

Complexity: We now briefly discuss the time complexity for Algorithm 3.1. We assume `ExecuteStatement()` takes constant time for each statement in the program. Then, the running time of the algorithm is linear in the number of units processed by the algorithm. We observe that an execution unit (S, σ), for each statement S and each state σ, will be processed at most once. This is because any subsequent copy of the unit will either get merged with the first unit while being entered into the worklist, if the unit has not been processed, or will be subsumed by the pre-state mapping at S, if the unit has already been processed, since the pre-state $\mathsf{preStateMap}[S]$ is a merge of all previously processed states. Suppose the size of the program being analyzed is N_S and the number of feasible abstract states is N_σ. The maximum number of units processed is $N_S N_\sigma$, and therefore, the running time for the algorithm is $O(N_S N_\sigma)$.

3.4 Implementation

We have implemented the *abstract execution framework* in the LLVM compiler framework [40]. LLVM presents a modular framework to implement compiler analyses and optimizations. It exposes an intermediate representation, called LLVM IR, for programs. LLVM IR is both easy to read in textual form and easy to analyze and manipulate in the data-structure representation exposed by the framework. It is well-documented and supported by a huge community of compiler developers. An analysis or a transformation is represented as a *pass* in LLVM. A large number of passes have been implemented in LLVM, that scan through the IR to analyze it for specific properties or to generate an optimized IR. Further, the passes can be stacked together in a sequence with one pass running after the other. The passes can specify dependencies on other passes, and the dependency chain can be used to appropriately schedule the passes. Note, the dependency chain must a DAG (directed acyclic graph), so that a total order on the sequence in which the passes must execute can be defined.

We have implemented a core framework which can be extended by an LLVM pass to implement the required analysis. The framework consists of a generic implementation of the abstract execution engine, described in Section 3.3, an abstract C++ class for *abstract value*, useful to define the abstract domain, and an abstract C++ class for *abstract state*. An analysis in this framework must implement the abstract classes for the abstract value and abstract state to define the abstract domain. Further, it must implement the `ExecuteStatement()` method in the abstract execution engine to define the abstract semantics. This division of the implementation between the core abstract execution framework and the analysis-specific abstract domain and abstract semantics greatly reduces the burden of an analysis developer and makes it easy to implement abstraction-based analyses. Further, it helps keep the analysis modular where the *core design* of the analysis, represented by its abstract domain and abstract semantics, is separated from the *implementation* of the analysis, represented by the abstract execution framework. There exist similar frameworks that separate the implementation from the definition of the analyses (for example, using Datalog to define static analyses [70]). However, these frameworks are often very formal and restricted in the types of static analyses defined in the framework. Abstract execution framework allows rich analyses to be defined while keeping the design and implementation separate from each other.

3.5 Other Approaches

We describe two alternate approaches to define abstraction-based static analysis: data-flow analysis and abstract interpretation. Both approaches have been widely used to define static analyses. Data-flow analysis [36, 52] represents the analysis as a flow of facts across the edges of a control flow graph. The program is represented as a control flow graph, where each node represents a basic block or a sequence of statements, and each edge represents the flow of control from one node to another under some guard condition. The analyses specify how incoming facts are processed by each node to generate the outgoing facts, and how facts from multiple incoming edges are merged together before they are processed. This approach has been used widely to define efficient analyses and transformations for sequential programs, many of which have been implemented in popular compilers. A key drawback of this approach is that it does not allow control over the order in which the facts are processed and what facts are forwarded to different outgoing edges. The order in which facts are processed and merged can affect the scalability of the analysis. Hence, fine-tuning the processing order is important.

Abstract interpretation [18, 52] represents the analysis as an interpretation of concrete executions for an abstract domain. The analyses define an abstraction function, that converts a concrete state into an abstract state, and abstract transfer functions, that propagate abstract state across statements in the program. The approach is similar to abstract execution, except there is a strong emphasis on ensuring that the abstraction is valid, and each abstract state and abstract transfer function is a sound approximation of the concrete state and concrete semantics for the program. This is useful for ensuring soundness/completeness of an analysis. However, for many of the analyses proving soundness or completeness is challenging. Hence, a large number of useful and light-weight analyses can not be defined in this framework.

3.6 Conclusion

This chapter presents the abstract execution framework to define static analyses. It requires the analysis designer to specify an abstract domain, which consists of abstract values for variables and the abstract state, and the abstract semantics for various program statements. Given these, the analysis can use the abstract execution engine to execute the program in the abstract domain. The generated abstract trace can then be utilized to generate the required analysis results. We rely on this common framework to define our static analyses in subsequent chapters.

Chapter 4

Static Detection of Uncoalesced Accesses

GPUs provide tremendous computational power, which when tapped can reduce the running time of an application by orders of magnitude. Yet, tapping this potential in GPUs is difficult, and a programmer needs to be aware of various subtleties of GPU architecture in order to achieve significant performance gain. One such area of subtlety is accessing global memory or the GPU DRAM. Global memory provides a high-latency access and any access to global memory takes hundreds of cycles to complete. To compensate for that, the global memory provides high-bandwidth and a large amount of data can be fetched in a single transaction. The GPU hardware utilizes this feature by *coalescing* or grouping together accesses to global memory by multiple threads into a few transactions. The key constraint, however, is that only contiguous bytes from a single memory block can be fetched in a single transaction. Hence, it needs to be ensured that the accesses by threads lie in a few memory blocks. Otherwise, multiple transactions are necessary and the accesses take much longer to complete, affecting the overall program performance.

The GPU hardware uses a simple approach to coalesce accesses for multiple threads. It groups together threads into units called *warps*. Each warp consists of a fixed number of threads (usually 32 or 64), with contiguous thread-ids. For instance, in a 1-dimensional thread-grid, threads with thread-ids 0 to 31 are placed in the first warp, threads with thread-ids 32 to 63 are placed in the second warp, and so on. Having contiguous thread-ids helps because threads with consecutive ids often access adjacent memory locations. A global memory access in the GPU program is executed simultaneously for all threads within a warp. When a warp executes a global memory access, the locations accessed by the individual threads are collected, and the GPU hardware coalesces them together into as few transactions as possible. If the accesses lie within a memory block, then the accesses require a single memory transaction. If the accesses are spread-out in the memory, multiple transactions are necessary, and such accesses are referred to as *uncoalesced accesses*. The problem of uncoalesced accesses has been well-documented in literature [53, §5.3.2].

Uncoalesced accesses pose a severe performance issue, and the programmers should ensure their programs do not contain uncoalesced accesses, unless they are unavoidable. Due to

the subtlety of the access patterns involved, it is often difficult for programmers to determine whether an access is coalesced or not. Therefore, automatic tools to report all uncoalesced accesses in a GPU program are desirable. A common approach used to automatically detect such accesses is to execute the program for some input data and then trace the execution to track number of transactions required by each global memory access. The accesses which require a large number of transactions are then reported back to the user as uncoalesced. This approach is also referred to as *dynamic analysis*, where the program is analyzed during an actual execution. There are multiple drawbacks of this approach. First, it requires an *actual* execution of the program. This implies that the program must be fully implemented before the dynamic analysis can run, and appropriate inputs are necessary for the uncoalesced accesses to present themselves during the execution. Second, the dynamic analysis is often slow and the GPU program takes orders of magnitude longer to complete with the dynamic analysis enabled. Hence, it is desirable to have a fast compile-time analysis to identify uncoalesced accesses in GPU programs.

We present here a light-weight compile-time analysis to detect uncoalesced accesses. The analysis relies on the idea that during the execution of a global memory access, the location accessed by a thread is a function of its thread-id. The function is often known at compile-time, specially when the access is coalesced. There are two primary forms of coalesced accesses: first where each thread accesses the same location in memory, in which case the function is an expression independent of thread-id; second where consecutive threads access consecutive locations, in which case the function is an expression of form $(\pm \mathsf{tid} + \mathrm{const})$, where tid represents thread-id and *const* represents an expression independent of thread-id. These are the most common patterns observed and can often be computed statically. The computation however may not be obvious if the value of thread-id flows through a chain of variable assignments into the final access index. We would need to track the flow through expressions to compute the access pattern. We rely on the abstract execution framework (defined in Chapter 3) to define an analysis where we track the dependence of each variable on tid. We further refine the analysis to track the number of threads that execute a statement. If a single thread in a warp executes a global memory access, then a single transaction to global memory is sufficient and the access is therefore coalesced. This is similar to an information flow analysis or slicing [1] where the flow of *dependence on* tid is tracked across program statements. However, our approach involves non-trivial computation of dependence values during assignments involving arithmetic and boolean operations and the merge operation after conditionals. This makes our approach unique as compared to the existing approaches.

The chapter is organized as follows. We first describe an example to explain the problem of uncoalesced accesses and how to detect such accesses using our static analysis in Section 4.1. Then we formalize the problem in Section 4.2. We describe the details of the design of our analysis in Section 4.3, and its implementation in Section 4.4. We describe some evaluation in Section 4.5. We present some related work in Section 4.6. We finally conclude in Section 4.7.

```
// i, N ↦ c0
if(tid+i+1 < N) {
  x = tid+i+1; // x ↦ c1
  for(y=i; y<N; y++){ // y ↦ c0
    xi = N*x + i; // xi ↦ ⊤
    xy = N*x + y; // xy ↦ ⊤
    iy = N*i + y; // iy ↦ c0
    A[xy] -= M[xi]*A[iy];
    if(y == i)
      B[x] -= M[xi]*B[i];
  }
}
```

(a) Original Fan2 snippet

```
// i, N ↦ c0
if(tid+i < N) {
  y = tid+i; // y ↦ c1
  for(x=i+1; x<N; x++){ // x ↦ c0
    xi = N*x + i; // xi ↦ c0
    xy = N*x + y; // xy ↦ c1
    iy = N*i + y; // iy ↦ c1
    A[xy] -= M[xi]*A[iy];
    if(y == i)
      B[x] -= M[xi]*B[i];
  }
}
```

(b) Repaired Fan2 snippet

Figure 4.1: `Fan2` Kernel snippets from Gaussian Elimination program to illustrate uncoalesced accesses.

4.1 An Uncoalesced Access: Gaussian Elimination

We use an example GPU program to illustrate the problem of uncoalesced accesses and our static analysis to detect such accesses. Figure 4.1a shows a 1-dimensional variant of `Fan2` kernel from Gaussian Elimination program in Figure 2.3 in Chapter 2. The kernel performs row operations on a matrix A and a vector B using the i^{th} column of a multiplier matrix M. The kernel uses a 1-dimensional thread-id, tid to distinguish executions of different threads. The kernel is executed for threads with ids in range $[0, N-i-2]$. Each thread is assigned a distinct row and updates the $(tid+i+1)^{th}$ row of matrix A and vector B. Note that A, B and M reside in global memory and are shared across threads, while the remaining variables are private to each thread.

The GPU executes threads in units of *warps*, where threads in each warp consist of consecutive ids and execute instructions in lock-step. The above kernel, for example, might be executed for warps: w_0 with ids $[0, 31]$, w_1 with ids $[32, 63]$, and so on ... When a warp, say w_0, accesses $A[xy]$ for some iteration y_0, the elements $A[N(i+1)+y_0], A[N(i+2)+y_0], \ldots, A[N(i+32)+y_0]$ are fetched simultaneously. The elements are at least N locations apart from each other, and thus, separate transactions are necessary to access each element which takes significant time and energy. This is an *uncoalesced* access. Access to $M[xi]$ is similarly uncoalesced. Figure 4.1b shows a repaired version of the kernel where each thread is mapped to a column in matrices A and M. The access to $A[xy]$ by the warp w_0 results in elements $A[Nx+i], A[Nx+i+1], \ldots, A[Nx+i+31]$ to be accessed. These are consecutive elements, and therefore, are fetched in a single global memory transaction. Access to $M[xi]$ is similarly coalesced. Our experiments show a 25% reduction in run-time for the repaired kernel when run for an input where $N = 1024$.

Analysis. We next describe how our static analysis detects uncoalesced accesses in the

kernel in Figure 4.1a. Our analysis defines an abstract domain that tracks local variables as a function of tid. All variables that are independent of tid are assigned value c_0 in the abstract domain. Thus, variables i and N are assigned value c_0 initially (shown in comments). Further, variables y and iy are constructed from tid-independent variables, and therefore, assigned c_0. All variables that are linear function of tid with a unit coefficient (*i.e.* of the form tid + const), are assigned value c_1. Hence, variable x is assigned value c_1. Lastly, all variables that are either non-linear function of tid or linear function with possibly greater than one coefficient are assigned $\top$. For example, variable xi is assigned expression $N.(tid + i + 1) + i$, where the coefficient for tid is N. Since the value of N is not known at compile-time and can be greater than one, xi is assigned $\top$. Similarly, variable xy is assigned $\top$. The analysis flags all global array accesses where the index variable has value $\top$ as uncoalesced. Hence, accesses A[xy] and M[xi] are marked uncoalesced. Note in the repaired kernel in Figure 4.1b, none of the index variables are $\top$, and hence, none of the accesses are marked as uncoalesced.

4.2 Formalization

This section presents a formal definition for uncoalesced accesses. We rely on the formal model described in Section 2.3 to define uncoalesced accesses. To define them, we first define the set of *reachable configurations* $\mathcal{R}(b, \sigma^G)$ reachable during the execution of kernel K for threads in block b starting in global state σ^G. A configuration is a tuple (σ, Π, S), where σ is the current state, Π is the set of threads which are currently enabled, and S is the next statement to be executed. We give an inductive definition for $\mathcal{R}$. The initial configuration $(\sigma, \mathcal{T}(b), K)$ belongs to $\mathcal{R}$, where $\sigma = (\sigma^G \cup \sigma^S_\perp \cup \sigma^L_\perp)$ is the initial state and $\mathcal{T}(b)$ is the set of threads in the block. For the recursive case, suppose (σ, Π, S) belongs to $\mathcal{R}$. When S is the sequence $[S_1; S_2]$, the configuration (σ, Π, S_1) belongs to $\mathcal{R}$, since S_1 is the next statement to be executed. Further, the configuration $(\llbracket S_1 \rrbracket(\sigma, \Pi), \Pi, S_2)$ belongs to $\mathcal{R}$ if the state $\llbracket S_1 \rrbracket(\sigma, \Pi)$ is not undefined. When S is a conditional [**if** l **then** S_1 **else** S_2], both branches are reachable, and hence, configurations (σ, Π_1, S_1) and (σ, Π_2, S_2) belong to $\mathcal{R}$, where $\Pi_1 = \{\tau \in \Pi : \sigma(l, \tau) = \text{true}\}$ and $\Pi_2 = \Pi \backslash \Pi_1$. Lastly, when S is a loop [**while** l **do** S′], the configuration (σ, Π', S'), where $\Pi' = \{\tau \in \Pi : \sigma(l, \tau) = \text{true}\}$, belongs to $\mathcal{R}$. Further, the configuration $(\llbracket S' \rrbracket(\sigma, \Pi'), \Pi', S)$ belongs to $\mathcal{R}$, if the state after executing S′ is not undefined.

We now define uncoalesced global memory accesses. Consider the configuration (σ, Π, AS), where AS is a read or write on a global array g at location $(l_0, l_1, \ldots, l_n)$. Consider the number of transactions $N_{\mathcal{A}}(\sigma, \Pi, AS, w)$ required for the access by a warp, as defined in Section 2.3.3. If it is greater than a threshold $t_{\mathcal{A}}$ for some warp w in the set of warps, we define the configuration (σ, Π, AS) as an *uncoalesced configuration*. We define a global memory access AS as *uncoalesced*, if an uncoalesced configuration involving the access belongs to $\mathcal{R}(b, \sigma^G)$ for some block b in the grid and some initial state σ^G. We state it formally in the following definition.

Definition 4.1. An access AS in a GPU program $\mathcal{P}$ is uncoalesced, if for some block b, initial

(a) Range-based uncoalescing **(b)** Alignment-based uncoalescing

Figure 4.2: An example illustrating range-based and alignment-based uncoalesced access. In the first case, the locations are strided, whereas in the second case, the locations are contiguous but mis-aligned with memory block boundary.

state σ^G and a warp w, there is a reachable configuration (σ, Π, AS) in $\mathcal{R}(b, \sigma^G)$ such that the number of transactions required, $N_{\mathcal{A}}(\sigma, \Pi, AS, w)$, is greater than threshold $t_{\mathcal{A}}$. Formally, an access AS is *uncoalesced* if:

$$\text{exists } b, \sigma^G, w, (\sigma, \Pi, AS) \in \mathcal{R}(b, \sigma^G), \text{ s.t. } N_{\mathcal{A}}(\sigma, \Pi, AS, w) > t_{\mathcal{A}}.$$

For many current GPUs, the bandwidth $\eta = 128$ bytes and warp size $n_w = 32$. We use threshold $t_{\mathcal{A}} = 1$, so that accesses requiring more than one transaction are marked uncoalesced. We observe that most global memory accesses either require a single transaction to finish or many more than one transaction. Therefore, $t_{\mathcal{A}} = 1$ serves an appropriate threshold to distinguish efficient coalesced accesses from poorly performing "uncoalesced" accesses.

Range-based Uncoalescing. One of the primary reasons for uncoalesced accesses is that the accesses by consecutive threads in a warp are strided. For such accesses, a large number of memory transactions are necessary to fetch all addresses. Consider an access $g[l_0, \ldots, l_n]$ to a global array g. Suppose index variable l_n is a linear function of tid_0, i.e. $l_n \equiv c.tid_0 + c_0$, while the remaining indices are independent of tid_0. The locations accessed by consecutive threads in a warp differ by c, and the range of addresses, $\mathcal{A}$, accessed by a warp w with all threads enabled is $32|kc|$ bytes, where $k = \xi(g)$ is the size of each element in g. Hence, the number of transactions $N_{\mathcal{A}}$ required is at least $|\mathcal{A}|/\eta = |kc|/4$. If $k \geqslant 4$ bytes and $|c| \geqslant 1$ (with one of the inequalities being strict), $N_{\mathcal{A}}$ is greater than 1 and the access is uncoalesced. We define such accesses where the range of addresses accessed by a warp is large as *range-based* uncoalesced accesses. Figure 4.2a illustrates such an uncoalesced acccess.

Alignment-based Uncoalescing. Alternately, an uncoalesced access can occur due to alignment issues when the range of accessed locations is small but mis-aligned with the memory block boundaries. Suppose $k = 4$ and $c = 1$ but $c_0 = 8$ for the access above. Since, $kc = 4$, the range of addresses is small and sufficient to be fetched in a single memory transaction. Yet, the access requires two global memory transactions. We refer to such accesses as *alignment-based* uncoalesced accesses. Figure 4.2b illustrates such an uncoalesced acccess.

4.3 Detecting Uncoalesced Accesses

We present a static analysis to detect range-based uncoalesced accesses in GPU programs. To define the analysis, We rely on the abstract execution framework presented in Chapter 3. The core idea behind the static analysis is to track the flow of thread-id tid_0 into the global memory accesses. For each global memory access, $g[\mathtt{l}_0, \ldots, \mathtt{l}_n]$, if the indices $\mathtt{l}_0$ to $\mathtt{l}_{n-1}$ are independent of tid_0 and the index $\mathtt{l}_n$ is either independent of tid_0 or a linear function of tid_0 with unit coefficient *i.e.* $\mathtt{l}_n = \mathsf{tid}_0 + \mathsf{c}$ where c is a constant, then the access is coalesced. However, if the above can not shown to be true for an access, then the access is potentially uncoalesced. The analysis tracks the dependence of index variables on thread-id tid_0 to check if an access is coalesced. To track the dependence, it defines a novel abstract domain that tracks dependence of all local integer and real variables on tid_0. We describe this domain in Section 4.3.1.

The analysis further tracks the number of threads enabled during the execution of an access. If an access is enabled for a single thread in a warp, a single location is accessed by the warp containing the enabled thread. Thus, a single global memory transaction is sufficient to complete the access, irrespective of the dependence of index variables on tid_0. The analysis hence also tracks the set of enabled threads during the execution of a statement. It tracks the set of enabled threads by tracking the dependence of *path-predicate*, or the condition under which the access is executed by a thread, on tid_0. The path-predicate is a conjunction of $\langle \mathtt{test} \rangle$ conditions for all conditionals surrounding the access. Each $\langle \mathtt{test} \rangle$ condition is defined by a local boolean variable. The analysis, therefore, also tracks the dependence of local boolean variables on tid_0 in its abstract domain.

We further present an abstract semantics in Section 4.3.2, to support the abstract domain. We present semantics for assignments with arithmetic and boolean operations on local variables, global array reads and writes, conditionals, and loops. The semantics are easy to define in most cases. However, an intricate definiton of the $merge$ operation is necessary to handle condtionals and loops. We finally present the overall analysis to detect uncoalesced accesses in Section 4.3.3, which essentially is an abstract execution of the program for the abstract domain and the abstract semantics. We further present some reasoning for the correctness of the analysis in this section.

4.3.1 Abstract Domain

We now present the abstract domain for our analysis. The abstract domain tracks the dependence of local variables in a GPU program on tid_0, or the thread-id along the first grid dimension. We track two kinds of dependences. First for integer and real variables, we track whether the value of the variable is independent of tid_0, it has a unit linear dependence on tid_0 whereby consecutive threads have consecutive values, or it has a non-unit linear or non-linear dependence on tid_0. The first two scenarios lead to coalesced accesses, while the last scenario is potentially uncoalesced. Next, we define the dependence of boolean variables on tid_0, which is whether the variable is independent of tid_0 whereby the variable has the same value in all threads, the variable is true for at most one thread, the variable is false for at most

one thread, or the variable is true for arbitrary number of threads. When the test condtion for a conditional is true for at most one thread, the true branch has a single thread enabled, and hence, all accesses within the branch are coalesced. Similarly, when the test condition is false, all accesses within the false branch are coalesced irrespective of their access patterns.

Integer/real abstact domain. We now define the abstract domain for local integer and real variables. We first give an intuitive definition for the abstract values and then a formal definition via the corresponding concrete values. We assign a single abstract value to each local variable that represents its dependence on tid_0. Each integer and real variable gets a value from the set $\widehat{\mathcal{V}}_{\mathsf{int}} = \{\bot, \mathsf{c}_0, \mathsf{c}_1, \mathsf{c}_{-1}, \top\}$. Let $\hat{v}$ represent an abstract value. The abstract values are defined as:

$\bot$: the value is not defined.
c_0 : the value is independent of thread-id tid_0 *i.e.* $\hat{v} \equiv [\text{const}]$.
c_1 : the value has unit dependence on tid_0 *i.e.* $\hat{v} \equiv [\mathsf{tid}_0 + \text{const}]$.
c_{-1} : the value has negative unit dependence on tid_0 *i.e.* $\hat{v} \equiv [-\mathsf{tid}_0 + \text{const}]$.
$\top$: otherwise.

We now formally define the values via the formal model in Section 2.3. Let $\hat{\sigma}$ represent the abstract state corresponding to a concrete state σ for a block b. The abstract state $\hat{\sigma}$ tracks the abstract values for variables in the program for all threads within the block. Since the abstract values track the dependence of variables on tid_0, we assign a single value for *all threads* in the block. We now define the abstract values for integer and real variables. Let l be a local variable. The abstract values are:

$$\hat{\sigma}(l) = \begin{cases} \bot, & \text{exists } \tau \in \mathcal{T}(b) \text{ s.t. } \sigma(l,\tau) = \bot. \\ \mathsf{c}_0, & \text{exists } c_0 \text{ s.t. for all } \tau \in \mathcal{T}(b), \sigma(l,\tau) = \mathsf{c}_0. \\ \mathsf{c}_1, & \text{exists } c_0 \text{ s.t. for all } \tau \in \mathcal{T}(b), \sigma(l,\tau) = \mathsf{tid}_0(\tau) + \mathsf{c}_0. \\ \mathsf{c}_{-1}, & \text{exists } c_0 \text{ s.t. for all } \tau \in \mathcal{T}(b), \sigma(l,\tau) = -\mathsf{tid}_0(\tau) + \mathsf{c}_0. \\ \top, & \text{otherwise.} \end{cases}$$

The two definitions are exactly alike, except in the second definition we associate the abstract values in $\hat{\sigma}$ to the corresponding concrete values in σ. This is useful to precisely define the abstract domain and also to reason about its correctness. For instance, from the second definition, it is obvious that each variable is mapped to a single abstract value for all threads in the block.

Boolean abstact domain. We next define the abstract domain for boolean variables which is used to track path-predicates in the analysis. We assign each boolean variable an abstract value from the set $\widehat{\mathcal{V}}_{\mathsf{bool}} = \{\bot, \mathsf{b}_{\mathsf{TF}}, \mathsf{b}_{\mathsf{Tt}}, \mathsf{b}_{\mathsf{Ff}}, \top\}$. We define the values as:

$\bot$: the value is not defined.
b_{TF} : the value is either true for all threads or false for all threads.
b_{Tt} : the value is false for at most one thread.
b_{Ff} : the value is true for at most one thread.
$\top$: otherwise.

We next define the values formally. The choice of names for the values will become clear once we see the formal definition. We first define a primitive set of values:

$$\hat{\sigma}(l) = \begin{cases} \bot, & \text{exists } \tau \in \mathcal{T}(b), \sigma(l,\tau) = \bot. \\ b_T, & \text{for all } \tau \in \mathcal{T}(b), \sigma(l,\tau) = \text{true}. \\ b_t, & \text{exists } \tau \in \mathcal{T}(b), \sigma(l,\tau) = \text{false, and for all } \tau' \in \mathcal{T}(b) \backslash \{\tau\}, \sigma(l,\tau') = \text{true}. \\ b_F, & \text{for all } \tau \in \mathcal{T}(b), \sigma(l,\tau) = \text{false}. \\ b_f, & \text{exists } \tau \in \mathcal{T}(b), \sigma(l,\tau) = \text{true, and for all } \tau' \in \mathcal{T}(b) \backslash \{\tau\}, \sigma(l,\tau') = \text{false}. \\ \top, & \text{otherwise.} \end{cases}$$

Here, b_T represents an abstract value that is true for all threads, b_t is true for all but one thread, b_F is false for all threads, while b_f is false for all but one thread. Now, the values b_{TF}, b_{Tt} and b_{Ff} are represented as a combination of above values. The value b_{TF} represents the pair $\{b_T, b_F\}$, the value b_{Tt} represents $\{b_T, b_t\}$, and the value b_{Ff} represents $\{b_F, b_f\}$, respectively.

We next define the *join* operation on the abstract values. The operation is quite straightforward. If any one of the values $\hat{v}_1$ and $\hat{v}_2$ being joined is undefined (*i.e.* $\bot$), then the joined value is also undefined. Otherwise, if $\hat{v}_1$ and $\hat{v}_2$ are equal, then *join* returns the same value. Otherwise, *join* returns the value $\top$, since the dependence of the joined value on tid for all abstract values $\hat{v}_1$ and $\hat{v}_2$ can not be defined by any other abstract value in the abstract domain.

Finally, we do not track shared or global arrays in our abstract domain. This is because, the index variables for array accesses are often a function of the local variables only. Also, when an index variable is computed by indexing into a shared or global array, it often represents an index non-linear in tid_0 (an exception is indexing for neighbour vertices in stencil computations). Hence, we do not model shared/global arrays in our abstract domain. We assume that arrays can hold arbitrary values, and indexing into an array returns $\top$, unless all threads access the same location in the array via a tid_0-independent index. To conclude the abstract state $\hat{\sigma}$ is a function from local variables to integer and boolean abstract values *i.e.* $V_L \rightarrow \widehat{\mathcal{V}}_{int} \cup \widehat{\mathcal{V}}_{bool}$.

We next define an abstract entity that tracks the number of threads enabled during the execution of a statement. We represent it via the symbol $\hat{\pi}$ and assign it an abstract boolean value representing the number of enabled threads. We assign it two values: b_{Ff} if at most one thread is enabled and $\top$ if an arbitrary number of threads is enabled. Note that we can visualize the set of threads Π as a predicate on the set of enabled threads in the block. Hence, $\hat{\pi}$ is an abstract boolean value representing Π.

Design Justification. We designed our abstraction by manually analyzing the set of benchmark programs in Rodinia [10], a popular GPU benchmark suite. We observed that most global memory access indices had a unit dependence or a large linear or non-linear dependence on tid_0. We did not observe small linear coefficients for the access indices. This motivated us to only consider unit dependence $\{c_1, c_{-1}\}$ on tid_0 for coalesced accesses and arbitrary dependence $\top$ for uncoalesced access. Further, we distinguish positive unit dependence c_1 from negative unit dependence c_{-1}. We observed a few scenarios where the values with positive and negative unit dependence were added together to produce a

tid_0-independent value and the distinction between the two dependences was useful. We identified abstract boolean values $\mathsf{b_{Ff}}$ and $\mathsf{b_{Tt}}$ to capture conditionals where either the true branch or the false branch had at most one thread enabled. The value $\mathsf{b_{TF}}$ was only a side-effect of our choice for boolean abstract domain. However, it turned out to be useful for the $merge$ operation in tid_0-independent conditionals, as explained in abstract semantics for conditionals in Section 4.3.2. Finally, we used an undefined value $\perp$ to initialize variables. This helps track the flow of only the defined variables, and also identify errors when the flow of values is broken due to the presence of undefined values.

4.3.2 Abstract Semantics

We now define the abstract semantics for the abstract domain defined in Section 4.3.1. We define the semantics for different types of statements. Given an abstract state $\hat{\sigma}$ and the path-predicate for the set of enabled threads $\hat{\pi}$, let $\hat{\sigma}'$ be the resulting state. We first define the semantics for local assignments. The semantics for most prominent scenarios is shown in Figure 4.3. Index tid_0 evaluates to c_1, while for all $\mathsf{i} \neq 0$, tid_i evaluates to c_0. Arithmetic operations on abstract values are defined just as regular arithmetic, except all values that do not have linear dependency on tid with coefficient c_0, c_1 or c_{-1}, are assigned $\top$. For example, $[\mathsf{c}_1 + \mathsf{c}_1] = \top$ since the resultant value has a dependency of c_2 on tid. Boolean values are constructed from comparison between arithmetic values. Equalites $[\hat{v}_1 = \hat{v}_2]$ are assigned a boolean value $\mathsf{b_{Ff}}$ (rule EQ) and inequalities $[\hat{v}_1 \neq \hat{v}_2]$ a boolean value $\mathsf{b_{Tt}}$ (rule NEQ), where one of $\hat{v}_1$ and $\hat{v}_2$ equals c_1 or c_{-1}, and the other is c_0. Note this is consistent with our abstraction. The equalities are of the form $[\mathsf{tid}_0 = \mathsf{c}]$ for some constant c and are true for at most one thread. The inequalities are of the form $[\mathsf{tid}_0 \neq \mathsf{c}]$ and are true for all except one thread. For boolean operations, we observe that $\neg \mathsf{b_{Tt}} = \mathsf{b_{Ff}}$ (rules BOOLNEG1 and BOOLNEG2), $[\mathsf{b_{Ff}} \bigwedge \mathsf{b}] = \mathsf{b_{Ff}}$ (rule AND), and $[\mathsf{b_{Tt}} \bigvee \mathsf{b}] = \mathsf{b_{Tt}}$ (rule OR), for all $\mathsf{b} \in \{\mathsf{b_{TF}}, \mathsf{b_{Ff}}, \mathsf{b_{Tt}}, \top\}$. Other comparison and boolean operations are defined similarly.

We show the semantics for the remaining statements in Figure 4.4. We first consider reads from shared and global arrays $[\mathsf{l} \leftarrow \mathsf{v}[\mathsf{l}_0, \ldots, \mathsf{l}_\mathsf{n}]]$. We do not model arrays in our abstract domain. It is easy to observe, however, if all index variables are tid_0-independent, then all threads in a warp access the same location and get the same value. Hence, the resulting value is tid_0-independent (rule READ1). However, if any of the indices depend on tid_0, then the threads access different locations in the array. Since we do not model arrays in our abstract domain, the threads can get arbitrary values from the array. Hence, the resulting value can have an arbitrary dependence on tid_0 and it is assigned the abstract value $\top$ (rule READ2). We do not define semantics for writes since we do not model arrays in our domain. We next define abstract semantics for a sequence of statements $[S = S_1; S_2]$. Let $\widehat{[\![S]\!]}(\hat{\sigma}, \hat{\pi})$ represent the abstract semantics of executing statement S in abstract state $\hat{\sigma}$ for the path-predicate $\hat{\pi}$ on enabled threads. Now, the semantics for sequence consists of composing executions of individual statements as shown in rule SEQ in Figure 4.4.

We next consider conditionals [**if** l **then** S_1 **else** S_2] shown in rules ITEIND and ITEDEP. The semantics takes the initial abstract state $\hat{\sigma}$ and the path-predicate $\hat{\pi}$. It first computes the path-predicates $\hat{\pi}_1$ and $\hat{\pi}_2$ for the abstract execution of statements S_1 and S_2. The path-

$$\textsc{Tid}[0]\frac{}{\hat{\sigma}(\mathsf{tid}_0) = \mathsf{c}_1} \qquad \textsc{Tid}[\mathrm{I}]\frac{i \neq 0}{\hat{\sigma}(\mathsf{tid}_i) = \mathsf{c}_0} \qquad \textsc{Bid}\frac{}{\hat{\sigma}(\mathsf{bid}[i]) = \mathsf{c}_0}$$

$$\textsc{Bdim}\frac{}{\hat{\sigma}(\mathsf{bdim}[i]) = \mathsf{c}_0} \qquad \textsc{Sum1}\frac{l \leftarrow l_0 + l_1 \quad \hat{\sigma}(l_0) = \mathsf{c}_0 \quad \hat{\sigma}(l_1) \in \widehat{\mathcal{V}}_{\mathsf{int}}}{\hat{\sigma}'(l) \leftarrow \hat{\sigma}(l_1)}$$

$$\textsc{Sum2}\frac{l \leftarrow l_0 + l_1 \quad \hat{\sigma}(l_0) = \mathsf{c}_1 \quad \hat{\sigma}(l_1) \in \mathsf{c}_{-1}}{\hat{\sigma}'(l) \leftarrow \mathsf{c}_0} \qquad \textsc{Sum3}\frac{l \leftarrow l_0 + l_1 \quad \hat{\sigma}(l_0) = \hat{\sigma}(l_1) \quad \hat{\sigma}(l_0) \in \{\mathsf{c}_1, \mathsf{c}_{-1}\}}{\hat{\sigma}'(l) \leftarrow \top}$$

$$\textsc{Neg1}\frac{l \leftarrow -l_0 \quad \hat{\sigma}(l_0) = \mathsf{c}_1}{\hat{\sigma}'(l) \leftarrow \mathsf{c}_{-1}} \qquad \textsc{Neg2}\frac{l \leftarrow -l_0 \quad \hat{\sigma}(l_0) = \mathsf{c}_{-1}}{\hat{\sigma}'(l) \leftarrow \mathsf{c}_1} \qquad \textsc{Neg3}\frac{l \leftarrow -l_0 \quad \hat{\sigma}(l_0) = \mathsf{c}_0}{\hat{\sigma}'(l) \leftarrow \mathsf{c}_0}$$

$$\textsc{Eq}\frac{l \leftarrow (l_0 = l_1) \quad \hat{\sigma}(l_0) = \mathsf{c}_0 \quad \hat{\sigma}(l_1) \in \{\mathsf{c}_1, \mathsf{c}_{-1}\}}{\hat{\sigma}'(l) \leftarrow \mathsf{b}_{\mathsf{Ff}}} \qquad \textsc{Neq}\frac{l \leftarrow (l_0 \neq l_1) \quad \hat{\sigma}(l_0) = \mathsf{c}_0 \quad \hat{\sigma}(l_1) \in \{\mathsf{c}_1, \mathsf{c}_{-1}\}}{\hat{\sigma}'(l) \leftarrow \mathsf{b}_{\mathsf{Tt}}}$$

$$\textsc{BoolNeg1}\frac{l \leftarrow \neg l_0 \quad \hat{\sigma}(l_0) = \mathsf{b}_{\mathsf{Ff}}}{\hat{\sigma}'(l) \leftarrow \mathsf{b}_{\mathsf{Tt}}} \qquad \textsc{BoolNeg2}\frac{l \leftarrow \neg l_0 \quad \hat{\sigma}(l_0) = \mathsf{b}_{\mathsf{Tt}}}{\hat{\sigma}'(l) \leftarrow \mathsf{b}_{\mathsf{Ff}}}$$

$$\textsc{And}\frac{l \leftarrow l_0 \bigwedge l_1 \quad \hat{\sigma}(l_0) = \mathsf{b}_{\mathsf{Ff}} \quad \hat{\sigma}(l_1) \in \widehat{\mathcal{V}}_{\mathsf{bool}} \backslash \{\bot\}}{\hat{\sigma}'(l) \leftarrow \mathsf{b}_{\mathsf{Ff}}} \qquad \textsc{Or}\frac{l \leftarrow l_0 \bigvee l_1 \quad \hat{\sigma}(l_0) = \mathsf{b}_{\mathsf{Tt}} \quad \hat{\sigma}(l_1) \in \widehat{\mathcal{V}}_{\mathsf{bool}} \backslash \{\bot\}}{\hat{\sigma}'(l) \leftarrow \mathsf{b}_{\mathsf{Tt}}}$$

$$\textsc{ArithZero}\frac{l \leftarrow l_0 \text{ op } l_1 \quad \hat{\sigma}(l_0) = \mathsf{c}_0 \quad \hat{\sigma}(l_1) \in \mathsf{c}_0}{\hat{\sigma}'(l) \leftarrow \mathsf{c}_0} \qquad \textsc{RelZero}\frac{l \leftarrow l_0 \text{ rel } l_1 \quad \hat{\sigma}(l_0) = \mathsf{c}_0 \quad \hat{\sigma}(l_1) \in \mathsf{c}_0}{\hat{\sigma}'(l) \leftarrow \mathsf{b}_{\mathsf{TF}}}$$

$$\textsc{BoolZero}\frac{l \leftarrow l_0 \text{ bop } l_1 \quad \hat{\sigma}(l_0) = \mathsf{b}_{\mathsf{TF}} \quad \hat{\sigma}(l_1) \in \mathsf{b}_{\mathsf{TF}}}{\hat{\sigma}'(l) \leftarrow \mathsf{b}_{\mathsf{TF}}}$$

Figure 4.3: Abstract semantics for different local assignments and initial abstract states. State $\hat{\sigma}$ is the incoming abstract state, while $\hat{\sigma}'$ is the updated state after the assignment. Operators *op*, *rel* and *bop* are arithmetic, relational and boolean operators, respectively. For the scenarios not shown above, an operation involving a variable with value $\bot$, returns the value $\bot$, whereas an operation involving value $\top$, returns value $\top$ for the operation.

$$\textsc{Read1}\ \frac{l \leftarrow v[l_0,\ldots,l_n] \quad v \in V_S \cup V_G \quad \text{for all } i \in \{0,\ldots,n\}, \hat{\sigma}(l_i) = c_0}{\hat{\sigma}'(l) \leftarrow c_0} \qquad \textsc{Read2}\ \frac{l \leftarrow v[l_0,\ldots,l_n] \quad v \in V_S \cup V_G \quad \text{exists } i \in \{0,\ldots,n\}, \hat{\sigma}(l_i) \neq c_0}{\hat{\sigma}'(l) \leftarrow \top}$$

$$\textsc{Seq}\ \frac{\hat{\sigma}_1 = \widehat{[\![S_1]\!]}(\hat{\sigma},\hat{\pi}) \quad \hat{\sigma}_2 = \widehat{[\![S_2]\!]}(\hat{\sigma}_1,\hat{\pi})}{\widehat{[\![S_1;S_2]\!]}(\hat{\sigma},\hat{\pi}) = \hat{\sigma}_2} \qquad \textsc{IteInd}\ \frac{\hat{\sigma}(l) = b_{TF} \quad \hat{\pi}_1 = \hat{\pi} \bigwedge \hat{\sigma}(l) \quad \hat{\pi}_2 = \hat{\pi} \bigwedge \neg\hat{\sigma}(l) \quad \hat{\sigma}_1 = \widehat{[\![S_1]\!]}(\hat{\sigma},\hat{\pi}_1) \quad \hat{\sigma}_2 = \widehat{[\![S_2]\!]}(\hat{\sigma},\hat{\pi}_2) \quad \hat{\sigma}' = merge(\hat{\sigma}_1,\hat{\sigma}_2)}{\widehat{[\![\textbf{if } l \textbf{ then } S_1 \textbf{ else } S_2]\!]}(\hat{\sigma},\hat{\pi}) = \hat{\sigma}'}$$

$$\textsc{IteDep}\ \frac{\hat{\sigma}(l) \neq b_{TF} \quad \hat{\pi}_1 = \hat{\pi} \bigwedge \hat{\sigma}(l) \quad \hat{\pi}_2 = \hat{\pi} \bigwedge \neg\hat{\sigma}(l) \quad \hat{\sigma}_1 = \widehat{[\![S_1]\!]}(\hat{\sigma},\hat{\pi}_1) \quad \hat{\sigma}_2 = \widehat{[\![S_2]\!]}(\hat{\sigma},\hat{\pi}_2) \quad \hat{\sigma}' = merge(\hat{\sigma}_1,\hat{\sigma}_2,S_1,S_2)}{\widehat{[\![\textbf{if } l \textbf{ then } S_1 \textbf{ else } S_2]\!]}(\hat{\sigma},\hat{\pi}) = \hat{\sigma}'} \qquad \textsc{Merge1}\ \frac{\hat{\sigma} = merge(\hat{\sigma}_1,\hat{\sigma}_2) \quad \hat{\sigma}_1(l) = \hat{v}_1 \quad \hat{\sigma}_2(l) = \hat{v}_2}{\hat{\sigma}(l) = join(\hat{v}_1,\hat{v}_2)}$$

$$\textsc{Merge2}\ \frac{\hat{\sigma} = merge(\hat{\sigma}_1,\hat{\sigma}_2,S_1,S_2) \quad AS \equiv [l \leftarrow \ldots] \text{ in } S_1 \text{ or } S_2}{\hat{\sigma}(l) = \top} \qquad \textsc{Merge3}\ \frac{\hat{\sigma} = merge(\hat{\sigma}_1,\hat{\sigma}_2,S_1,S_2) \quad AS \equiv [l \leftarrow \ldots] \text{ not in } S_1 \text{ and } S_2}{\hat{\sigma}(l) = \hat{\sigma}_1(l)}$$

$$\textsc{WhileInd}\ \frac{\hat{\sigma}(l) = b_{TF} \quad \hat{\pi}_1 = \hat{\pi} \bigwedge \hat{\sigma}(l) \quad \hat{\sigma}_1 = \widehat{[\![S]\!]}(\hat{\sigma},\hat{\pi}_1) \quad \hat{\sigma}_2 = merge(\hat{\sigma},\hat{\sigma}_1) \quad \hat{\sigma}_3 = \widehat{[\![\textbf{while } l \textbf{ do } S]\!]}(\hat{\sigma}_2,\hat{\pi}_1)}{\widehat{[\![\textbf{while } l \textbf{ do } S]\!]}(\hat{\sigma},\hat{\pi}) = \hat{\sigma}_3}$$

$$\textsc{WhileDep}\ \frac{\hat{\sigma}(l) \neq b_{TF} \quad \hat{\pi}_1 = \hat{\pi} \bigwedge \hat{\sigma}(l) \quad \hat{\sigma}_1 = \widehat{[\![S]\!]}(\hat{\sigma},\hat{\pi}_1) \quad \hat{\sigma}_2 = merge(\hat{\sigma},\hat{\sigma}_1,S) \quad \hat{\sigma}_3 = \widehat{[\![\textbf{while } l \textbf{ do } S]\!]}(\hat{\sigma}_2,\hat{\pi}_1)}{\widehat{[\![\textbf{while } l \textbf{ do } S]\!]}(\hat{\sigma},\hat{\pi}) = \hat{\sigma}_3}$$

Figure 4.4: Abstract semantics for shared/global reads, conditionals and loops for different initial abstract states. State $\hat{\sigma}$ is the incoming abstract state while $\hat{\sigma}'$ is the updated state after the assignment.

predicate $\hat{\pi}_1$ is computed by conjuncting abstract values $\hat{\sigma}(l)$ and $\hat{\pi}$. This is because if either $\hat{\sigma}(l)$ has value b_{Ff} *i.e.* l is true for at most one thread, or $\hat{\pi}$ has value b_{Ff} *i.e.* at most one thread is enabled for the conditional, then S_1 is executed for at most one thread, and hence $\hat{\pi}_1$ is set to b_{Ff}. Otherwise, S_1 is executed for arbitrary number of threads, and $\hat{\pi}_1$ is set to the abstract value $\top$. The abstract semantics for $\bigwedge$ operation capture this accurately. The

path-predicate $\hat{\pi}_2$ is computed similarly.

Next, we abstractly execute S_1 and S_2 in the abstract state $\hat{\sigma}$ for path-predicates $\hat{\pi}_1$ and $\hat{\pi}_2$. Since we do not know the concrete values for boolean variable l during the abstract execution, we execute both branches for each thread and then conservatively *merge* the resulting states $\hat{\sigma}_1$ and $\hat{\sigma}_2$ to get the final updated state $\hat{\sigma}'$ after the conditional. We consider two scenarios for merging the resulting states. First, when $\hat{\sigma}(l)$ is tid_0-independent (rule ITEIND), all threads in the block execute either the **if** branch S_1 or all threads execute the **else** branch S_2. Hence, the final value for any variable l_i is one of the values $\hat{\sigma}_1(l_i)$ or $\hat{\sigma}_2(l_i)$, and therefore, l_i is assigned the $join$ of the two values in $merge(\hat{\sigma}_1, \hat{\sigma}_2)$ (rule MERGE1). Consider the abstract execution for the example shown here:

```
if (x = 5){
  y := 10;
} else {
  y := tid[0] + 10;
}
```

Suppose $\hat{\sigma}(x) = c_0$ and $\hat{\sigma}(y) = \bot$ initially. In the program, the predicate $[x = 5]$ is independent of tid_0, and hence, all threads in a warp execute the same branch. If they execute the **if** branch, y is assigned a constant and y gets the abstract value c_0, and otherwise, y is assigned $tid_0 + 10$ and gets the value c_1. Since either branch can execute, the dependence of y on tid_0 is not known, and therefore, it is assigned $join(c_0, c_1) = \top$.

Next, we consider the case when $\hat{\sigma}(l)$ is dependent on tid_0 (shown in rule ITEDEP). Here, a special $merge$ operation is required to get the merged state $\hat{\sigma}'$. To understand this better, consider the abstract execution for the following example:

```
if (tid[0] < x){
  y := 10;
} else {
  y := 20;
}
```

In this program, the predicate $[tid_0 < x]$ is dependent on tid_0. Therefore, all threads with tid_0 less than x execute the **if** branch, while the remaining threads execute the **else** branch. Along both branches y is assigned a constant, and hence, gets an abstract value c_0. However, the final value for y is dependent on tid_0 and is a non-linear function of tid_0, since depending on which branch is executed it gets the value 10 or 20. Hence, y must be assigned the abstract value $\top$. But the usual $merge$ operation on the abstract states returns $join$ of the two values which is $join(c_0, c_0) = c_0$. To address this inconsistency, we define a new $merge$ operation that takes S_1 and S_2 as additional parameters. The new $merge$ operation checks if a variable l_i is assigned a value in S_1 or S_2. If so, it sets the value for l_i in the merged state to $\top$ (rule MERGE2), since it is potentially a non-linear function of tid_0. The values for remaining variables remain unchanged, and hence, it returns their original value in $\hat{\sigma}$ (rule MERGE3).

The abstract semantics for loops are defined similar to conditionals in rules WHILEIND and WHILEDEP. There are two primary differences here. First, the $merge$ operation for tid_0-dependent test condition (rule WHILEDEP) only takes S as the additional operator, since the

$merge$ operation is performed between the original state $\hat{\sigma}$ and the state after executing S, $\hat{\sigma}_1$, depending on whether the statement S was executed or not. Second, the semantics for loop involve repeating the loop until the resultant state $\hat{\sigma}_2$ after the execution saturates and is same as that in the previous iterations. Unlike concrete execution where the bounds on the number of iterations of the loop is exactly defined for all threads, the bounds are not known in abstract execution. Hence, we repeat the loop indefinitely, until repeating the loop does not change the resultant state $\hat{\sigma}_2$. We can stop at this point, because executing a new iteration for the loop results in the same state. Such a state is also known as a *fixpoint*, and is uniquely defined for most abstract domains with finite set of values and a monotonic $join$ operation.

We next define abstract semantics for function calls. We assume a function call always returns an arbitrary return value, and hence, set the return value to $\top$. We further assume that none of the parameters sent to the function call are modified by the function. Local variables are passed by value. Shared/global arrays are passed by reference and can be modified, however we do not track values stored in arrays. Therefore, no abstract semantics are necessary to update these after the function call.

4.3.3 Overall Analysis

We now define the overall analysis. The analysis initializes an abstract state for the kernel. It initializes all local variables to $\bot$ and all constants to c_0 in the initial abstract state $\hat{\sigma}_0$. Further, a kernel usually consists of a set of local parameters passed to the kernel from the calling method. These parameters are initialized appropriately. We address how to initialize these parameters in Section 4.4.2. The analysis similarly initializes the path-predicate $\hat{\pi}_0$ based on the calling method. Once the initial abstract state $\hat{\sigma}_0$ and path-predicate $\hat{\pi}_0$ is defined, the analysis begins abstract execution using the abstract semantics defined in Section 4.3.2. During the execution, when it reaches a global memory read or write AS, it checks if the access is "uncoalesced". Suppose the reaching configuation consists of an abstract state $\hat{\sigma}$ and path-predicate $\hat{\pi}$. Let the statement AS access a global array g at location $[\mathsf{l}_0, \ldots, \mathsf{l}_n]$. The analysis reports the access as *uncoalesced* if the following conditions hold:

- $[\hat{\pi} = \top]$: This implies that multiple threads are potentially enabled at this statement during the execution.
- One of following three cases must be true:
  - [exists $\mathsf{i} \neq \mathsf{n}, \hat{\sigma}(\mathsf{l}_\mathsf{i}) \neq \mathsf{c}_0$]: This implies one of the non-final indices is dependent on tid_0. On flattening, this index is multiplied by a large value, which leads to a large dependence on tid_0, and hence, a potentially uncoalesced access.
  - $[\hat{\sigma}(\mathsf{l}_\mathsf{n}) = \top]$: This directly implies that the index is a non-linear function of tid_0, and hence, it leads to an uncoalesced access.
  - $[\hat{\sigma}(\mathsf{l}_\mathsf{n}) \in \{\mathsf{c}_1, \mathsf{c}_{-1}\} \bigwedge \xi(\mathsf{g}) > 4]$: Here, the index is a linear function of tid_0 with unit coefficient. However, the size of each element in the array is large, and hence, addresses accessed by consecutive threads are at least $\xi(\mathsf{g})$ addresses

apart. Therefore, consecutive threads access distant locations and this potentially leads to an uncoalesced access.

The analysis continues abstract execution until the abstract state saturates for all loops and all statements are executed and no statements are left to be executed. It collects all accesses that were potentially uncoalesced during the execution and returns these as the output of the analysis. We now show that the reported accesses cover all range-based uncoalesced accesses present in the GPU program.

4.3.3.1 Correctness

We briefly discuss the correctness of the analysis. We first define an abstract reachable configuration. An abstract configuration $(\hat{\sigma}, \hat{\pi}, S)$ is reachable if S is visited in state $\hat{\sigma}$ with predicate $\hat{\pi}$ during the abstract execution for the analysis. We can define the set of abstract reachable configurations similar to reachable configurations set $\mathcal{R}$, defined in Section 4.2. An abstract configuration is uncoalesced, if the analysis identifies it as uncoalesced based on the conditions defined above.

We show that for all global memory accesses AS if a range-based uncoalesced configuration can reach AS for some initial state σ_0 and set of threads Π_0, then an abstract uncoalesced configuration is also reachable and the analysis identifies AS as uncoalesced. We first prove that for any reachable configuration in $\mathcal{R}$, there is an abstract configuration that subsumes the concrete configuration and is reachable during the abstract execution. The initial abstract configuration subsumes all initial configurations (assuming it is initialized correctly), *i.e.* $\alpha(\sigma_0) \sqsubseteq \hat{\sigma}_0$ and $\alpha(\Pi) \sqsubseteq \hat{\pi}_0$, where $\alpha()$ returns the smallest abstract entity that represents the concrete entity in the abstract domain, and $\hat{v}_1 \sqsubseteq \hat{v}_2$ implies $\hat{v}_2$ supersedes or is larger than $\hat{v}_1$. Further, since the abstract semantics preserves the abstraction, we can show by induction that at each step of the abstract execution, the concrete configuration is subsumed by the resulting abstract configuration. Hence, any reachable configuration (σ, Π, S) is subsumed by the corresponding abstract configuration $(\hat{\sigma}, \hat{\pi}, S)$ obtained by replaying the execution in the abstract domain.

Now for a range-based uncoalesced configuration (σ, Π, AS), where AS is an access to a global array g at location $[l_0, \ldots, l_n]$, first the access must be executed by more than one thread. Hence, $|\Pi| > 0$ and the abstraction of Π, $\alpha(\Pi) = \top$. Further, the flattened access index must be a non-linear or large linear function of tid_0, so that accesses by threads are spread far-apart in memory. This can occur, first if one of l_0 to l_{n-1} is dependent on tid_0 and on flattening leads to a large access index, which implies $\alpha(l_i) \neq c_0$; second, if l_n is has a large dependence on tid_0, in which case $\alpha(l_n) = \top$; third, if l_n has a unit dependence on tid_0, but the size of each element in the array is greater than 4 bytes. If none of the above conditions hold, then the access is coleasced, because the flattened index is either a constant or has at most unit dependence on tid_0 but the size of each element is less than 4 bytes. Both of these scenarios lead to a single transaction to global memory. Therefore, the access must be uncoalesced if one of the above conditions is true. Also, there must exist an abstract reachable configuration that subsumes the configuration (σ, Π, AS), and such a configuration would be reported as uncoalesced by the analysis.

Theorem 4.2. *For all global memory accesses* AS*, if a range-based uncoalesced configuration* (σ, Π, AS) *is reachable, then an abstract uncoalesced configuration* $(\hat{\sigma}', \hat{\pi}', AS)$ *is also reachable, and the analysis identifies* AS *as* uncoalesced.

4.4 Implementation

We have implemented the static analysis to detect uncoalesced accesses in our abstract execution framework for LLVM (Section 3.4). Recently, a front-end for CUDA was added to LLVM [72], that converts programs written in CUDA into LLVM IR. Since CUDA programs consist of both the CPU and GPU code, the frontend makes two passes through the program. In the first pass, it parses all GPU kernels and generates a single file containing LLVM IR for the GPU code, also known as the device code. In the second pass, it parses CPU methods into LLVM IR and generates a file with LLVM IR for the CPU code, also called the host code. The individual IRs can be subsequently analyzed and transformed to obtain the optimized IRs which can then be converted into a GPU binary.

In Section 4.3, we present the overall design and the core idea behind the analysis. To do this however, we gloss over many features of mordern programming langauges like pointers, structures and inter-procedural nature of programs. Most of these are present in current GPU programming languages. Therefore, it is necessary to support them in a static analysis. We address this here and describe how our static analysis for uncoalesced accesses handles these practical challenges for current programming languages.

4.4.1 Handling Pointers and Structures

Pointers. Like any other programming language, current GPU programming languages support pointers and structures. We first describe how pointers are handled in our analysis. We extend the abstract values in our analysis with a bit of information `isPointerType` that distinguishes a regular abstract value from a pointer abstract value. A *pointer abstract value* tracks the dependence of the *address* of a variable on tid_0 unlike regular abstract values which track the dependence of the *value* of a variable. This is useful to track the dependence for variables whose address is initialized dynamically, for instance, by dereferencing a pointer variable. We observe that for each variable in the program, storing either one of the regular abstract value or the pointer abstract value suffices. If the address of a variable is initialized dynamically at run-time, then storing the abstract value for the address of the variable gives enough information about the value of the variable. Consider the following example to understand this further. Suppose $\mathtt{a}$ is a pointer variable independent of tid_0.

```
int *p = a + tid[0];
int q = *p;
int r = *a;
```

Now p is assigned a pointer abstract value with value c_1. Since the address of p is dependent on tid_0, the instance of p in each thread corresponds to a distinct location. Each location can hold an arbitrary value, and therefore, dereferencing the pointer would return a value

that has arbitrary dependence on tid_0. Hence, dereferencing $\mathtt{p}$ assigns an abstract value $\top$ to $\mathtt{q}$. However, if the pointer variable is independent of tid_0, which happens to be the case for variable $\mathtt{a}$, then each thread accesses the same location in $\mathtt{a}$. We assume that distinct threads do not write to the same location in the array, otherwise we would have a data-race. Therefore, the value read by each thread is identical, and hence, the dereferenced value is also independent of tid_0. Therefore, we assign the abstract value c_0 to variable $\mathtt{r}$.

The semantics for pointer abstract values can be defined similar to regular abstract values, except the statements considered involve pointer arithmetic and pointer assignments. We assume that the pointers do not alias with each other, and therefore, we do not implement any alias analysis. This assumption is generally true for GPU programs. The GPU programs are much simpler than regular programs with very few complex data-structures. Hence, aliasing of pointers is not a problem in general.

Structures. Handling structures does not pose any major challenges. LLVM IR implements field accesses in structures by indexing the structure at a constant location. Hence, array accesses and field accesses appear alike and no special semantics are necessary. One small issue, however, is that the size of element accessed by a thread may not correspond to the stride for the array. Consider the following example. Suppose $\mathtt{a}$ is a global array of type `int4`.

```
// struct int4 {int p; int q; int r; int s; } a[1024];

int x = a[tid[0]].p;
```

Here, each thread accesses the field $\mathtt{p}$ in consecutive locations of array $\mathtt{a}$. While the size of element accessed is 4 bytes, the size of each element in the array is 16 bytes. Hence, a total of 4 global memory transactions are necessary to fetch data for a fully-enabled warp, even though the size of element accessed is only 4 bytes. We handle this discrepency by back-tracking from the element access to the access where the array was accessed with a tid_0-dependent index and using the size of element in the array being indexed as the desired element size.

4.4.2 Handling Multiple Procedures

The analysis in Section 4.3 glosses over how the parameters for which the analysis is run are initialized. We now describe how the calling context is computed and how the parameters for a kernel are initialized. We do a multi-procedure analysis to compute these for each kernel in the program. We use a top-bottom analysis approach where we analyze callers before callees. Note that, we analyze each kernel only *once*, but in a sequence where a caller for a kernel is analyzed before the kernel itself. We start the analysis from global kernels which are kernels called from the CPU code. For these kernels, we assume that the parameters passed to the kernel are independent of tid_0, and hence, we initialize them to c_0 (b_{TF} for boolean variables). We next define how the calling context for a GPU function $\mathtt{f}$ is constructed. When a function call to $\mathtt{f}$ is encountered during the analysis of $\mathtt{g}$, a caller for

f, we record the abstract values for parameters, $\hat{v}_0^g, \ldots, \hat{v}_n^g$, passed to f during the function call. Let the existing calling context for f consist of abstract values $\hat{v}_0(f), \ldots, \hat{v}_n(f)$, where $\hat{v}_i(f)$ refers to the current value for ith parameter in function f. We update the calling context by merging the new parameter values with existing ones to obtain values $\hat{v}'_0(f) = join(\hat{v}_0(f), \hat{v}_0^g), \ldots, \hat{v}'_n(f) = join(\hat{v}_n(f), \hat{v}_n^g)$ for the updated calling context.

We observe that most kernels are called with the same call context and the parameters get the same values across function calls. Therefore, analyzing kernels under each individual call context is not required, and analyzing them under a single *merged* context suffices. Further, a top-bottom analysis suffices, since the result of a kernel call rarely flows into an access index, and when that is the case, such calls are often inlined for efficiency. This greatly improves the scalability of the analysis, and allows the analysis to finish at compile-time. We note, however, this approach does not support recursive kernels, but then GPU programs often do not consist of recursive kernels in practice. Hence, this approach provides a simple and efficient way to analyze GPU programs.

4.4.3 Handling Control Flow Graph Representation

LLVM IR exposes a control flow graph representation of the program, which consists of a directed graph of nodes, each node representing a statement or a sequence of statements, also known as a *basic block*, and each edge representing the flow of control from one statement to the next. Each edge might be guarded by a condition under which the transition can be taken. The conditionals and loops are lowered into this representation and their explicit structure is lost. This poses problems specially for the special $merge$ operation after the execution of branches and the loop body. Further, the IR is often converted into an SSA (Single Static Assignment) form where each variable v is assigned a value once in the program. If the same variable is assigned a value in both branches of a conditional, then during the SSA conversion, the variable v is replaced by new temporaries v_1 and v_2, assigned values within branches, and the variables v_1 and v_2 are then merged in a 'PHI statement' to produce a new variable v_3 after the conditional. The PHI statement records the variables being merged and the corresponding nodes in which each variable is assigned. It produces the merged value and assigns it to a new variable. The following program shows the usage of PHI statements.

```
int v = 0;
if (v > 5) {
  v1 = v + 5;
} else {
  v2 = v + 10;
}
v3 = PHI(v1, v2);
```

Since the explicit structure of conditionals and loops is lost, we use an alternate approach to implement the $merge$ operation for conditionals and loops with a tid_0-dependent condition. First, we identify the variables assigned a value on either branch via the PHI statements present after the conditional or loop. For each PHI statement S that merges variables v_1 and v_2 into v_3, we compute the abstract value for v_3 as follows. We first identify the *immediate*

dominator for the PHI statement S. A node u is a dominator for node v in a graph G if every path to v from the root node goes through u. An immediate dominator is a dominator that has the least distance to v. For loops and conditionals, the branches and the loop body are dominated by a node with a *branch* statement where the test condition for the loop/conditional is checked. Further, the node after the branches and the loop body where the PHI statement S is present, is immediately dominated by the node with the branch statement. Therefore, we identify the dominating node for PHI statement S and check if it consists of branch statement with a tid_0-dependent condition. If so, we assign v_3 the value $\top$, since v_3 potentially has a non-linear dependence on tid_0. Otherwise, we assign it the join of the incoming value and its previous value, so that it is assigned the value $join(v_1, v_2)$. Remaining variables are not updated inside the branches/loop-body, and hence, remain unchanged after the loop/conditional.

We have a problem when the values are stored within branches/loop-body via a store statement, and retrieved after the conditional/loop via a load statement. Consider the following example (based on a benchmark program ParticleFilter in Rodinia benchmark suite). Suppose p is a pointer variable independent of tid_0 and points to a local variable x in each thread.

```
int *p = &x;
if (tid[0] > N) {
  *p = 10;
} else {
  *p = 20;
}
int y = *p;
```

As can be observed, the value of y has non-linear dependence on tid_0. Our approach can not handle this scenario and incorrectly assigns the value c_0 to y. This is because there is no PHI statement after the conditional to merge assignments within the branches. One approach to address this is to use *memory dependence* information that maps each load/store to the immediate store/load statement prior to the load/store instruction. We observed that memory dependence analysis is too conservative, and hence, the generated information is too imprecise which leads to a large number of false positives. Also, we observed this issue in only one benchmark ParticleFilter in the Rodinia benchmark. Therefore, we decided not to handle this scenario.

4.5 Evaluation

This section describes the evaluation of our static analysis on the Rodinia benchmarks (version 3.1) [10]. Rodinia consists of GPU programs from various scientific domains. We compare static analysis against a dynamic analysis implementation that executes benchmarks for the inputs provided with Rodinia and collects uncoalesced accesses at run-time. The dynamic analysis is precise, and hence, each reported access is an actual access. However, the analysis might miss uncoalesced accesses in uncalled kernels and along unexecuted paths in

Benchmark	LOC	Real UA	Static Analysis		Dynamic Analysis	
			UA (real)	Runtime(s)	UA	Runtime(s)
backprop	110	7	0 (0)	0.14	7	5.23
bfs	35	7	7 (7)	0.07	0-7	3.89
b+tree	115	19	19 (19)	0.35	7	16.71
CFD	550	0	22 (0)	12.41	-	-
dwt2D	1380	0	16 (0)	5.99	n/a	3.72
gaussian	30	6	6 (6)	0.07	5-6	6.82
heartwall	1310	8	25 (8)	39.87	-	-
hotspot	115	3	2 (0)	0.75	3	0.89
hotspot3D	50	2	12 (2)	0.21	2	327.00
huffmann	395	21	26 (21)	0.68	3	2.42
lavaMD	180	9	9 (9)	0.73	5	511.60
lud	160	3	0 (0)	0.34	3	0.83
myocyte	3240	19	19 (19)	1,813.72	0	134.13
nn	10	4	4 (4)	0.06	2	0.13
nw	170	7	2 (2)	0.41	6	4.17
particle filter	70	4	3 (2)	0.58	4	11.62
pathfinder	80	3	0 (0)	0.22	3	4.25
srad_v1	275	2	14 (2)	0.33	2	185.00
srad_v2	250	9	0 (0)	1.38	9	53.94
streamcluster	45	10	10 (10)	0.11	-	-
		143	180 (111)		69	

Table 4.1: Evaluation results for static analysis and dynamic analysis on Rodinia benchmark programs. We use UA to refer to uncoalesced accesses. "-" indicates the DA hit the 2-hour timeout.

kernels. More details about the dynamic analysis can be found in [4]. We run our static and dynamic analyses to identify existing uncoalesced accesses in these programs. We run an *intra-procedural* version of the static analysis, where the initial parameters for each kernel are initialized to tid_0-independent values. We observe that most kernels were called with tid_0-independent values, and therefore, the intra-procedural analysis is sufficient to identify uncoalesced accesses present in the benchmark suite. We have implemented our analyses in LLVM version 3.9.0, and compile programs with `--cuda-gpu-arch=sm_30`. We use CUDA SDK version 7.5. We run our experiments on an Amazon EC2 instance with Amazon Linux 2016.03 (OS), an 8-core Intel Xeon E5-2670 CPU running at 2.60GHz, and an Nvidia GRID K520 GPU (Kepler architecture).

Table 4.1 shows results of our experiments. It shows the benchmark name, the lines of GPU source code analyzed, the manually-validated real uncoalesced accesses, and the number of uncoalesced accesses found and running time for each analysis. The Rodinia suite consists of 22 programs. We exclude 4 (hybridsort, kmeans, leukocyte, mummergpu) as

they could not be compiled due to lack of support for texture functions in LLVM. A subtle point to note here is that since the static analysis was designed by analyzing the benchmarks themselves, the static analysis results might be biased. Yet, the analysis itself is simple and well-designed without specific adaptation for any of the benchmarks. Also, the results span a large number of benchmarks from varied domains, and are reasonable for any static analysis. Hence, given that the static analysis finishes within a few minutes for most benchmarks and can execute at compile-time, the merits of static analysis stand even when they overfit the provided benchmarks. Though, a further analysis on other benchmarks to validate generalization of the analysis would be useful. We next address different questions related to the evaluation.

Do uncoalesced accesses occur in real programs? We found 143 manually-validated uncoalesced accesses in Rodinia benchmarks, with uncoalesced accesses in almost every program (Column "Real uncoalesced accesses" in Table 4.1). A few uncoalesced accesses involved random or irregular access to global arrays (bfs, particle filter). Such accesses are dynamic and data-dependent, and hence, difficult to fix. Next, we found uncoalesced accesses where consecutive threads access consecutive rows of global matrices, instead of columns (gaussian). Such accesses could be fixed by assigning consecutive threads to consecutive columns or changing the layout of matrices, but this is possible only when consecutive columns can be accessed in parallel. Another common uncoalesced access occurred when data was allocated as an array of structures instead of a structure of arrays (nn, streamcluster). A closely related issue was one where the array was divided into contiguous chunks and each chunk was assigned to a thread, instead of allocating elements in a round-robin fashion (myocyte, streamcluster). There were some uncoalesced accesses which involved reduction operation (for example, sum) on arrays (heartwall, huffmann). These accesses do not have a standard solution, and few of the above solutions could be applicable. A few uncoalesced accesses were caused by alignment issues where accesses by a warp did not align with cache-block boundaries, and hence, got spilled over to multiple blocks. These were caused, first, when the input matrix dimensions were not a multiple of the warp size which led consecutive rows to be mis-aligned (backprop, hotspot3D), or when the array itself was misaligned due to incorrect padding (b+tree). These could be fixed by proper padding.

Which real uncoalesced accesses does static analysis miss? While the static analysis identifies a significant number of uncoalesced accesses (111 out of 143), it does miss a few in practice. We found two primary reasons. 22 of the missed uncoalesced accesses depend on the second dimension of thread-id, tid_1, while we only considered the smallest dimension in our analysis. Uncoalesced accesses typically do not depend on higher dimensions unless the block dimensions are small or not a multiple of the warp size. We modified our analysis to track the second dimension and observed that all such uncoalesced accesses were caught by our static analysis at the cost of 20 new false positives. Next, eight of the missed uncoalesced accesses were alignment-based which were caused by an unaligned offset added to tid_0. The actual offsets are challenging to track via a static analysis. Finally, two missed uncoalesced accesses (particle filter) were due to an issue with conditionals described in Section 4.4.3.

What false positives does static analysis report? For most programs, the static analysis reports few or no false positives. The primary exceptions are CFD, dwt2D and heartwall,

which account for the bulk of our false positives. A common case occurred when tid_0 was divided by a constant, and multiplied back by the same constant to generate the access index (heartwall, huffman, srad_v1). Such an index should not lead to uncoalesced accesses. The static analysis, however, cannot assert that the two constants are equal, since we do not track exact values, and hence, sets the access index to $\top$, and reports any accesses involving the index as uncoalesced. Another type of false positive occurred when access indices were non-linear function of tid_0, but consecutive indices differed by at most one, and led to coalesced accesses. Such indices were often either generated by indirect indexing (CFD, srad_v1) or by assigning values in conditionals (heartwall, hotspot, hotspot3D). In both cases, our static analysis conservatively assumed them to be uncoalesced. Lastly, a few false positives happened because the access index was computed via a function call (__mul24) which returned a coalesced index (huffmann), though we conservatively set the index to $\top$.

How scalable is static analysis? As can be noted, the static analysis is quite fast, and finishes within seconds for most benchmarks. The largest benchmark, myocyte, is 3240 lines of GPU code, with the largest kernel containing 930 lines. The static analysis takes significant time for this benchmark, however for the remaining benchmarks, it finishes within a few minutes.

How does static analysis compare with dynamic analysis? The dynamic analysis misses about half of the uncoalesced accesses in our benchmarks, but the results reported by the analysis correspond to actual uncoalesced accesses. We found several benchmarks where different inputs varied the number of uncoalesced accesses reported by dynamic analysis (bfs, gaussian, lud). Similarly, the analysis finds uncoalesced accesses along a single execution path, so all uncoalesced accesses in unexecuted branches or uncalled kernels were not found. Due to compiler optimizations it can be difficult to map the results of dynamic analysis back to source code. In dwt2D, we were unable to do so due to multiple uses of C++ templates. Moreover, the dynamic analysis does not scale to long-running programs, as it incurs slowdown of orders of magnitude.

4.6 Related Work

There are a few existing static analysis tools to identify uncoalesced accesses. First, many existing compilers for optimizing GPU programs use implicit analysis to detect uncoalesced accesses [75, 64, 66, 6]. However, these analyses are stated informally, often hidden as part of the optimization, and not evaluated for precision. Further, they present additional constraints which restricts their applicability. CUDA-lite [66] relies on programmer annotations. Sung et al [64] and Baskaran et al [6] require the programs to have affine acccess patterns. Yang et al [75] present an incomplete analysis covering a restricted set of coalesced accesses and conservatively reporting others as uncoalesced. CUPL [5] presents a preliminary static analysis to detect uncoalesced accesses. However, unlike our work, it does not present a formalization and a detailed evaluation for its analysis.

Several tools have been developed to verify GPU programs, which also identify uncoalesced accesses in programs along with other performance and correctness issues. GKLEE [44] presents a symbolic execution engine for CUDA programs, where the programs

are executed for symbolic inputs to identify various issues including uncoalesced accesses. PUG [43] and GPUVerify [8] present static analysis tools to identify data-races in GPU programs where threads are also treated symbolically apart from symbolic inputs, and two symbolic threads are sufficient to identify all possible data-races in the programs. The primary drawback of these approaches is that they do not scale to large programs. They rely on SMT solvers to check the validity of symbolic expressions computed during the symbolic execution, which do not scale to large formulae. On the other hand, our work uses abstract values and abstract semantics to compute values during the analysis, which avoids the need for SMT solvers and scales the analysis to large programs.

There are a few dynamic analysis approaches to identify uncoalesced accesses. Fauzia et al [23] use dynamic analysis to identify uncoalesced accesses, and then use this information to drive polyhedral model based code transformations to generate colaesced accesses. CuMAPz [37] relies on runtime traces to present a comprehensive analysis of GPU program performance for different types of memory access patterns. Nvidia presents various performance analysis tools [54] based on profiling and tracing via hardware performance counters, where uncoalesced accesses is one of the few analysis results. All these approaches are, however, computationally expensive and require an actual execution of GPU programs. Our work can complement these by providing a fast compile-time alternative to compute uncoalesced accesses.

4.7 Conclusion

In this chapter, we present the problem of uncoalesced accesses, a precise definition of the problem, and a light-weight compile-time analysis to identify such accesses in GPU programs. The analysis relies on the regularity of global memory access patterns which are easy to identify at compile-time. The analysis uses on a non-trivial abstract domain that tracks the *dependence* of variables on thread-id tid_0. This is similar to taint-tracking in information flow analyses, where the flow of taint is tracked from sensitive variables to public variables. However, our analysis tracks the dependence on tid_0 rather than the value itself, which makes it unique. Tracking the dependence on thread-id, or in general, the relationship between the execution of a thread and its thread-id is essential for scalable analysis of GPU programs. This idea can be further extended to other kinds of analyses like detecting shared memory bank-conflicts, which we will address in future work.

Chapter 5

Block-Size Independence for GPU Programs

Tuning GPU applications is important to achieve significant speedups and to have performance portability across GPUs. It can, however, introduce subtle errors into the application which can be difficult to debug and resolve. We need tools that can automatically detect these errors and ensure any transformations performed while tuning an application are correct. Existing tools for GPU verification help identify correctness issues like data-races and barrier-divergence [44, 43, 8], but none verify correctness of transformations. Further, automatically synthesizing optimal execution configuration is difficult since the optimization space is large and non-convex [58]. Therefore, tuning applications by trying out different values for parameters is unavoidable. This further necessitates automatic tools that can verify correctness of transformations. This chapter focuses on the correctness of tuning an execution parameter *block-size*. The parameter block-size represents the number of threads in each thread-block and is specified during kernel invocation, along with the total number of threads. The block-size determines how resources required by the program are allocated on GPU cores, and is often tuned to maximally utilize each core for performance, while balancing performance across cores in a GPU.

We present an analysis to verify *block-size independence* of GPU programs which ensures changing block-size is a valid transformation and does not introduce errors into the program. In the GPU execution model, threads in a thread-block can share data via *shared memory*, and changing the block-size alters the sets of threads allowed to share data, making program equivalence hard to reason about. Therefore, we only consider *synchronization-free* programs, where each thread executes independently of other threads and any sharing of data between threads is prohibited and leads to a data-race. For synchronization-free programs, the analysis needs to ensure that the execution of each individual thread is independent of block-size. The analysis is a taint-tracking approach that checks if any block-size dependent values flow into the final result for a thread. Each thread in a GPU program is provided with a block-id, `bid`, a thread-id within the block, `tid`, and the block-size, `bdim`. These values get modified when the block-size is modified, and hence, the analysis tracks the flow of these values through program variables to verify that they do not influence the final result.

```
__global__ void cudaProcess(int imgw, unsigned g_odata[][imgw]) {
    int tx = tid[0]; int ty = tid[1];
    int bw = bdim[0]; int bh = bdim[1];
    int x = bid[0]*bw + tx;
    int y = bid[1]*bh + ty;
    uchar4 c4 = make_uchar4((x & 0x20)?100:0, 0,
                            (y & 0x20)?100:0, 0);
    g_odata[y, x] = rgbToInt(c4.z, c4.y, c4.x);
}
```

Figure 5.1: Example illustrating block-size independence.

Interestingly, the expression $(\mathsf{bid}_i \cdot \mathsf{bdim}_i + \mathsf{tid}_i)$ identifies a globally unique id for each thread for all dimensions i, and remains unchanged when the block-size is modified. This is because, the set of global-ids for a thread-grid defines a continuous space of values where every id within the space is assigned to some thread. Changing the block-size does not change this space of values, and hence, the global-id for each thread remains unchanged. The analysis tracks the sub-expressions of the global-id expression and when a variable is observed to be a function of this expression, it is marked as independent of block-size by the analysis. To gain further precision, the analysis also tracks block-size independent *multipliers*, so the expressions of the form $(k \cdot \mathsf{bid}_i \cdot \mathsf{bdim}_i + k \cdot \mathsf{tid}_i)$, where k is a block-size independent value, can be proven block-size independent. Finally, if none of the block-size dependent values flow into the final state for any thread, the program is determined as block-size independent. The analysis uses a novel abstraction to track these values, where *symbolic* constants track block-size independent multipliers while *abstract* constants track sub-expressions of global-id. This combination of abstract values with symbolic values helps scale the analysis while retaining good precision.

To understand this further, consider the function `cudaProcess()` in Figure 5.1 from a GPU program "simpleCUDA2GL" in Nvidia CUDA SDK. The function initializes pixels in an image represented by array `g_odata`. Each thread initializes a globally unique location (y, x) with a value that is a function of these coordinates. The coordinates x and y are independent of block-size. Also, the function is synchronization-free and each thread executes independently. Therefore, the function must be block-size independent. To prove this, the analysis tracks the flow of block-size dependent values $\mathsf{bid}, \mathsf{bdim}$, and tid through program variables. Note that, to mirror the 2-dimensional nature of the image, the threads are organized in a 2-dimensional grid, where the first and second dimensions identify the x and y coordinates, respectively. Initially, $imgw$ is block-size independent. Next, tx is assigned tid_0, ty is assigned tid_1 and so on. Importantly, variables x and y are assigned $(\mathsf{bid}_0 \cdot \mathsf{bdim}_0 + \mathsf{tid}_0)$ and $(\mathsf{bid}_1 \cdot \mathsf{bdim}_1 + \mathsf{tid}_1)$, respectively, both of which are block-size independent. Further, calls to functions `make_uchar4()` and `rgbToInt()` return block-size independent values. Therefore, writes into array `g_odata` by threads and the resultant final array are block-size independent. Hence, the analysis verifies the program to be block-size independent. Now, suppose

the program instead initializes variable y as:

```
int y = bid[1]*bw + ty;
```

Notice we use variable bw which corresonds to $bdim_0$ instead to bh, and therefore, value of y is block-size dependent, and the program is block-size dependent for this initialization of y. If the block-size along both first and second dimensions is equal initially (which is often the case), then this would not be noticable during the execution of the program. However, if block-size is updated so that it has different values along the two dimensions, then we might get inconsistent results.

The chapter is organized as follows. Section 5.1 identifies and formalizes the problem of *block-size independence* for GPU programs. Section 5.2 presents a *scalable* inter-procedural analysis to verify block-size independence for the class of *synchronization-free* GPU programs. Section 5.3 demonstrates the relevance of the problem for real-world GPU programs through an extensive evaluation on Nvidia CUDA SDK samples. Section 5.4 presents some related work and Section 5.5 concludes.

5.1 Formalization

In this section, we present a formalization for the problem of block-size independence in Section 5.1.1. We rely on the formal-model described in Section 2.3 to define the property. We next prove some interesting properties about synchronization-free GPU programs in Section 5.1.2, which are useful to prove the correctness of the analysis.

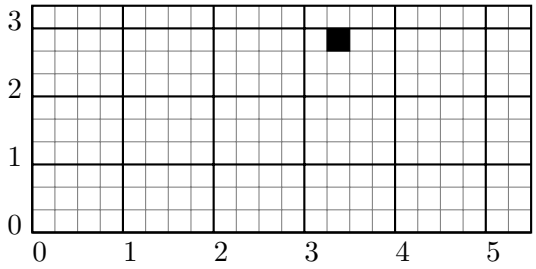
5.1.1 Block Size Independence

We formally define block-size independence for a GPU program. Let two states σ and σ' be equivalent (*i.e.* $\sigma \equiv \sigma'$) if they consist of the same set of variables and each variable has the same valuation in both states. Also, recall from Section 2.3, $[\![K]\!](\sigma^G, \vec{N}, \vec{B})$ represents the execution of a kernel K in initial global state σ^G for a grid with grid-size $\vec{N}$ and block-size $\vec{B}$. We state block-size independence in the following definition.

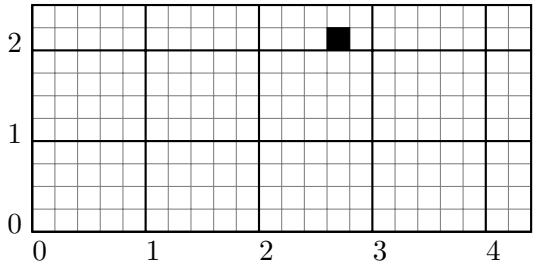
Definition 5.1. A GPU program $\mathcal{P}$ is *block-size independent*, if and only if for all initial global states σ^G and grid-sizes $\vec{N}$, the execution of the program is independent of the block-size $\vec{B}$, that is:

$$\text{for all } \sigma^G, \vec{N}, \vec{B}, \vec{B}', \ [\![K]\!](\sigma^G, \vec{N}, \vec{B}) \equiv [\![K]\!](\sigma^G, \vec{N}, \vec{B}').$$

Note that as described in Section 2.4, CUDA specifies the number of blocks $\vec{N}_b$ instead of the total number of threads as grid-size. However, if the updates to block-size are restricted to divisors of total number of threads *i.e.* $\vec{B}'_i$ is a divisor for $(\vec{N}_b)_i \vec{B}_i$ for all grid dimesions i, then the new grid consists of the same number of total threads as the original grid and the above definition is also applicable to CUDA programs.



(a) $\vec{N} = 22 \times 10$, $\vec{B} = 4 \times 3$.



(b) $\vec{N} = 22 \times 10$, $\vec{B} = 5 \times 4$.

Figure 5.2: Example illustrating block-size independence of global-id expression. Note the global position of the thread for darkened cell remains $(13, 8)$, also referring to the global-id. Further, the thread-grid defines a continuous space where each thread is assigned a global-id.

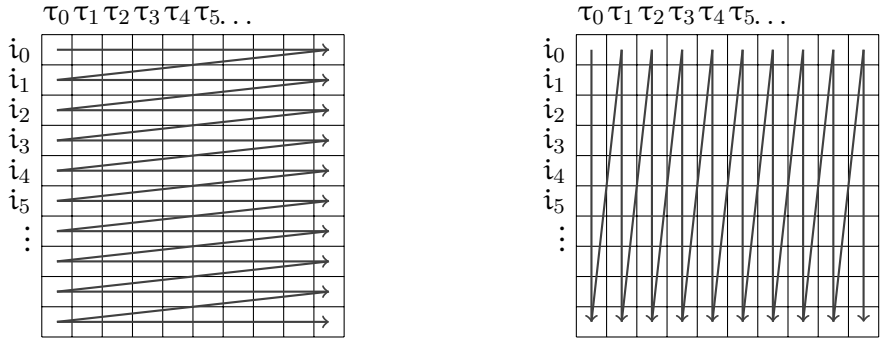
5.1.2 Reduction to Thread-local Block Size Independence

In a synchronization-free GPU program, each thread must execute independently of the other threads (since any dependence on updates from other threads would lead to a data-race). Therefore, the *global* problem of verifying block-size independence of the program can be reduced to the *local* problem of verifying block-size independence for the execution of each thread in the program. We show that if the program is synchronization-free, then verifying thread-local block-size independence for the program is sufficient to verify block-size independence for the program.

We first define *thread-local block-size independence* for GPU programs. A GPU program is thread-local block-size independent if the execution of each thread in the thread-grid is independent of block-size. Given block-sizes $\vec{B}$ and $\vec{B}'$, let a thread τ in grid $\mathcal{G}(\vec{N}, \vec{B})$ be equivalent to another thread τ' in grid $\mathcal{G}(\vec{N}, \vec{B}')$, *i.e.* $\tau \equiv \tau'$, if they have the same unique global location in the thread-grid, namely:

$$\text{for all } 0 \leqslant i < d,\ \mathsf{bid}_i(\tau).\mathsf{bdim}_i(\tau) + \mathsf{tid}_i(\tau) = \mathsf{bid}_i(\tau').\mathsf{bdim}_i(\tau') + \mathsf{tid}_i(\tau')$$

where $\mathsf{bid}_i(\tau)$ refers to the value bid_i in thread τ. For instance, the global-id for thread corresponding to the darkened cell in Figure 5.2 remains unchanged as $(13, 8)$, even though the individual thread-ids and block-ids are modified when block-size is modified. We can



(a) Fine-grained interleaving **(b)** Coarse-grained interleaving

Figure 5.3: The figure shows fine-grained vs coarse-grained interleaving of threads in a block. The rows represent sequence of instructions to be executed, while the columns represent the threads in a block. The arrows signify the order in which the threads and the instructions are executed.

observe this equivalence to be a one-to-one relation, where each thread τ in the first grid corresponds to a *unique* global thread τ' in the second grid. Now, the program is thread-local block-size independent, if each pair of equivalent global threads has equivalent executions. Recall $[\![S]\!](\sigma, \Pi)$ denotes the execution of statement S for a set of threads Π starting in initial state σ. Also remember $p^G(\sigma)$ represents the projection of state σ on to the global variables V_G.

Definition 5.2. A GPU program $\mathcal{P}$ is *thread-local block-size independent*, if and only if for all initial states σ and grid-sizes $\vec{N}$, the global state after the execution of each thread in the thread-grid is independent of block-size. Formally, the program is thread-local block-size independent iff:

$$\text{for all } \sigma, \vec{N}, \vec{B}, \vec{B}', \tau \in \mathcal{G}(\vec{N}, \vec{B}), \tau' \in \mathcal{G}(\vec{N}, \vec{B}'),$$
$$\tau \equiv \tau' \implies p^G([\![K]\!](\sigma, \{\tau\})) \equiv p^G([\![K]\!](\sigma, \{\tau'\})).$$

We now observe that for a synchronization-free program, the lock-step execution of threads in a block is equivalent to executing threads one after another. This is because, to avoid data-races, each thread must operate independently and not see updates from other threads. Therefore, the order of execution between threads does not matter and a fine-grained interleaving (Figure 5.3a) produces the same execution as a coarse-grained interleaving (Figure 5.3b).

Lemma 5.3. *Given a synchronization-free GPU program $\mathcal{P}$ and a set of threads $\Pi = \{\tau_0, \ldots, \tau_k\}$,*

the lock-step execution of threads is equivalent to executing threads sequentially:

$$\textit{for all } \sigma, \Pi,\ [\![K]\!](\sigma, \Pi) \equiv \sigma_{k+1}, \\ \textit{where } \sigma_0 = \sigma \textit{ and for all } 0 \leqslant i \leqslant k, \sigma_{i+1} = [\![K]\!](\sigma_i, \{\tau_i\}).$$

Proof sketch. Suppose the lock-step execution of threads is not equivalent to sequential execution of threads. Then, there must exist a variable v that differs in its final value after the two executions. This implies the variable v must have seen two distinct writes in the two executions. Let the two writes be S_1 and S_2 respectively. We show that the two writes must define a data-race. If the writes lie within distinct threads, then clearly there is data-race between the two threads. Alternately, if the writes lie within the same thread, there must be some variable v' in the thread that is assigned distinct values in the two executions. By backward reasoining, we can show that the distinct values for the variable must originate in distinct writes to a variable in distinct threads, which represents a data-race. □

By Lemma 5.3, the lock-step execution of threads in a block can be substituted with sequential execution of threads. Next, we observe that we can execute each thread in a state where the local and shared variables are undefined initially. This is because the thread must not observe any updates to these variables from the previously executed threads, or we would have a data-race. Also, these variables are discarded at the end of the execution of the block and we need not retain their values. Remember $\mathcal{B}$ represents the set of all blocks in thread-grid and $[\![K]\!](\sigma^G, \Gamma)$ represents execution for a set of blocks Γ, where the local and shared variables are undefined initially and the result of the execution consists of the global state only.

Lemma 5.4. *Given a synchronization-free GPU program $\mathcal{P}$ and a block b, the lock-step execution for block b is equivalent to executing threads sequentially with local and shared variables initialized to undefined values:*

$$\textit{for all } \sigma^G, b \in \mathcal{B}, \\ [\![K]\!](\sigma^G, \{b\}) \equiv \sigma^G_{k+1}, \textit{ where } \sigma^G_0 = \sigma^G \textit{ and } \sigma^G_{i+1} = p^G([\![K]\!](\sigma^L_\perp \cup \sigma^S_\perp \cup \sigma^G_i, \{\tau_i\})), \\ \textit{for all } \tau_i \textit{ in } \mathcal{T}(b) = \{\tau_0, \ldots, \tau_k\}.$$

Finally from Lemma 5.4 and the definition of thread-local block-size independence, the execution of each thread in the first grid can be substituted with the execution of equivalent thread in the second grid, and therefore, thread-local block-size independence of a synchronization-free program implies block-size independence for the program. We conclude the following theorem.

Theorem 5.5. *If a synchronization-free GPU program $\mathcal{P}$ is* thread-local block-size independent*, then it is also* block-size independent.

5.2 Analysis for Synchronization-free GPU Programs

This section presents an analysis to verify block-size independence for *synchronization-free* GPU programs, where the kernel does not contain `__syncthreads()` statements. In a synchronization-free GPU program, the *global* problem of verifying block-size independence of the program can be reduced to the *local* problem of verifying block-size independence for the execution of each thread in the program, as shown in Section 5.1.2. Further, the execution of a thread is independent of block-size if the writes by the thread to the shared and global variables do not depend on block-size[1]. A write can depend on block-size if either the location accessed, the value written or the condition under which the write is executed is dependent on block-size. The only sources of block-size dependence in a thread are the thread's block-id, $\mathsf{bid}(\tau)$, the thread-id, $\mathsf{tid}(\tau)$, and the block-size itself, $\mathsf{bdim}(\tau) = \vec{B}$. Further, as we observed earlier, the expression $\mathsf{gid}_i(\tau) = (\mathsf{bid}_i \cdot \mathsf{bdim}_i + \mathsf{tid}_i)(\tau)$ is independent of block-size for all thread dimensions i.

We present our analysis to check thread-local block-size independence of GPU programs and to ensure that the execution of each thread is block-size independent. Initially when a thread's execution starts, only constants $\mathsf{bid}, \mathsf{bdim}$ and tid are block-size dependent and the remaining variables are block-size independent. While bdim is equal to block-size, the thread-id tid and block-id bid of a thread also depend on the block-size and get updated when the block-size is modified. Hence, if any of these values potentially flows into a global variable update, then the final global state after the thread's execution depends on block-size and the program is block-size dependent. The analysis defines an abstraction of state and abstract semantics for kernel instructions to track the flow of block-size dependent values during a thread's execution. Note we run the analysis separately for each dimension of the thread-grid. So for the subsequent discussion, consider bid, bdim and tid to be one-dimensional values.

Abstract Domain. The analysis defines an abstract domain to track the dependence of each local scalar variable on block-size. The abstract domain assigns each local integer/real variable an abstract value from the set $\widehat{\mathcal{V}}_{\mathsf{int}} = \{\mathsf{c}_{\mathsf{ind}}, \mathsf{kc}_{\mathsf{tid}}, \mathsf{kc}_{\mathsf{bid}}, \mathsf{kc}_{\mathsf{bdim}}, \mathsf{kc}_{\mathsf{bid}}\mathsf{c}_{\mathsf{bdim}}, \mathsf{c}_{\mathsf{bsize}}\}$, where k is a block-size independent variable. The values are:

$\mathsf{c}_{\mathsf{ind}}$:	the value is independent of block-size.
$\mathsf{kc}_{\mathsf{bid}}$:	the value is of the form $\mathsf{k} \cdot \mathsf{bid}$.
$\mathsf{kc}_{\mathsf{bdim}}$:	the value is of the form $\mathsf{k} \cdot \mathsf{bdim}$.
$\mathsf{kc}_{\mathsf{tid}}$:	the value is of the form $(\mathsf{k} \cdot \mathsf{tid} + \text{const})$.
$\mathsf{kc}_{\mathsf{bid}}\mathsf{c}_{\mathsf{bdim}}$:	the value is of the form $(\mathsf{k} \cdot \mathsf{bid} \cdot \mathsf{bdim} + \text{const})$.
$\mathsf{c}_{\mathsf{bsize}}$:	otherwise.

The value $\mathsf{c}_{\mathsf{ind}}$ represents all block-size independent values. The abstract value $\mathsf{c}_{\mathsf{bsize}}$ represents values with arbitrary dependence on block-size. We observe the expression $(\mathsf{k} \cdot \mathsf{bid} \cdot \mathsf{bdim} + \mathsf{k} \cdot \mathsf{tid})$, where k is a block-size independent variable, is independent of block-size. To

[1]Reads can be ignored because our `__syncthreads()`-free and race-free assumptions permit a thread to only read values it has written itself or are part of the initial state.

take this account, the analysis tracks abstract values for sub-expressions of this expression, $k c_{bid}$, $k c_{bdim}$, $k c_{tid}$ and $k c_{bid} c_{bdim}$, where k is the *multiplier* or a symbolic constant representing a block-size independent local variable. We assume each local variable has a unique definition (*e.g.* SSA form), and the variables are not updated after they are first defined. Hence, the symbolic constant represents the correct abstract value for each variable k.

We now define the abstract values formally. Let $\hat{\sigma}$ be the abstraction of program state σ, which maps each local variable to an abstract value, *i.e.* $V_L \to \widehat{\mathcal{V}}$. Let l, k be local variables. Let f_0 be a function that maps each thread to a block-size independent value. For integer and real variables, the abstraction is defined as:

$$\hat{\sigma}(l) = \begin{cases} c_{ind}, & \text{for all } \tau,\ \sigma(l,\tau) = f_0(\tau). \\ k c_{bid}, & \hat{\sigma}(k) = c_{ind}; \text{for all } \tau,\ \sigma(l,\tau) = \sigma(k,\tau).\mathsf{bid}(\tau). \\ k c_{bdim}, & \hat{\sigma}(k) = c_{ind}; \text{for all } \tau,\ \sigma(l,\tau) = \sigma(k,\tau).\mathsf{bdim}(\tau). \\ k c_{tid}, & \hat{\sigma}(k) = c_{ind}; \text{for all } \tau,\ \sigma(l,\tau) = \sigma(k,\tau).\mathsf{tid}(\tau) + f_0(\tau). \\ k c_{bid} c_{bdim}, & \hat{\sigma}(k) = c_{ind}; \text{for all } \tau,\ \sigma(l,\tau) = \sigma(k,\tau).\mathsf{bid}(\tau).\mathsf{bdim}(\tau) + f_0(\tau). \\ c_{bsize}, & \text{otherwise.} \end{cases}$$

We similarly define an abstraction for local boolean variables, which tracks whether the boolean variable depends on block-size or not. Let b_0 be a block-size independent boolean function. The abstraction for boolean variables is:

$$\hat{\sigma}(l) = \begin{cases} b_{ind}, & \text{for all } \tau,\ \sigma(l,\tau) = b_0(\tau). \\ b_{bsize}, & \text{otherwise.} \end{cases}$$

Finally, we do not track shared and global variables or arrays in our abstraction. We compensate by tracking each write to these variables and ensuring that the writes are independent of block-size. We further define a *path-predicate*, $\hat{\pi}$, which is the condition under which a statement is executed. The value of $\hat{\pi}$ is an abstract boolean value, representing whether the condition is dependent on block-size or not.

Abstract Semantics. We now define some abstract semantics for propagating the abstract state $\hat{\sigma}$ and the path-predicate $\hat{\pi}$ through statements in the kernel. Figure 5.4 defines updates to the abstract states for different assignment statements and initial states. Note the rules in Figure 5.4 are only valid if the path-predicate $\hat{\pi}$ is b_{ind}. Further, we show rules for the scenarios where the result is non-trivial and not equal to c_{bsize}/b_{bsize}. For the remaining scenarios, the updated value for arithmetic/boolean variables is set to c_{bsize}/b_{bsize}. Lastly, the path-predicate remains unchanged after each assignment statement.

We now briefly describe the rules shown in Figure 5.4. Note that when the multiplier k_0 for an abstract value is constant 1, we drop the multiplier, *e.g.* c_{bid} in rule PROD1. The rules ensure that the abstraction is preserved. For example, in rule SUM2, abstract values $k_0 c_{tid}$ and $k_1 c_{bid} c_{bdim}$ are added together, where k_0 equals k_1. This is equivalent to the expression $(k_0 \cdot \mathsf{tid} + k_0 \cdot \mathsf{bid} \cdot \mathsf{bdim})$, which we know is block-size independent. Hence, the final result is assigned the value c_{ind}. Similarly, the other rules update the abstract state while preserving the abstraction. An important point to note here is that during the product operation (rules PROD1, PROD2, PROD3), the multiplier for at least one of the operands must be constant 1,

$$\textsc{tid}\ \frac{}{\hat{\sigma}(\mathsf{tid}) = \mathsf{c_{tid}}} \qquad \textsc{bid}\ \frac{}{\hat{\sigma}(\mathsf{bid}) = \mathsf{c_{bid}}} \qquad \textsc{bdim}\ \frac{}{\hat{\sigma}(\mathsf{bdim}) = \mathsf{c_{bdim}}}$$

$$\textsc{gdim}\ \frac{}{\hat{\sigma}(\mathsf{gdim}) = \mathsf{c_{ind}}} \qquad \textsc{Sum1}\ \frac{\begin{array}{c} l \leftarrow l_0 + l_1 \qquad \hat{\sigma}(l_0) = \mathsf{c_{ind}} \\ \hat{\sigma}(l_1) \in \{\mathsf{k_0 c_{tid}}, \mathsf{k_1 c_{bid} c_{bdim}}\} \end{array}}{\hat{\sigma}'(l) \leftarrow \hat{\sigma}(l_1)}$$

$$\textsc{Sum2}\ \frac{\begin{array}{c} l \leftarrow l_0 + l_1 \qquad \hat{\sigma}(l_0) = \mathsf{k_0 c_{tid}} \\ \hat{\sigma}(l_1) = \mathsf{k_1 c_{bid} c_{bdim}} \qquad \mathsf{k_0} \equiv \mathsf{k_1} \end{array}}{\hat{\sigma}'(l) \leftarrow \mathsf{c_{ind}}} \qquad \textsc{Prod1}\ \frac{\begin{array}{c} l \leftarrow l_0.l_1 \qquad \hat{\sigma}(l_0) = \mathsf{c_{ind}} \\ \hat{\sigma}(l_1) \in \{\mathsf{c_{bid}}, \mathsf{c_{bdim}}, \mathsf{c_{tid}}, \mathsf{c_{bid} c_{bdim}}\} \end{array}}{\hat{\sigma}'(l) \leftarrow l_0 \hat{\sigma}(l_1)}$$

$$\textsc{Prod2}\ \frac{\begin{array}{c} l \leftarrow l_0.l_1 \qquad \hat{\sigma}(l_0) = \mathsf{c_{bid}} \\ \hat{\sigma}(l_1) = \mathsf{k_0 c_{bdim}} \end{array}}{\hat{\sigma}'(l) \leftarrow \mathsf{k_0 c_{bid} c_{bdim}}} \qquad \textsc{Prod3}\ \frac{\begin{array}{c} l \leftarrow l_0.l_1 \qquad \hat{\sigma}(l_0) = \mathsf{k_0 c_{bid}} \\ \hat{\sigma}(l_1) = \mathsf{c_{bdim}} \end{array}}{\hat{\sigma}'(l) \leftarrow \mathsf{k_0 c_{bid} c_{bdim}}}$$

$$\textsc{Arith}\ \frac{\begin{array}{c} l \leftarrow l_0\ \mathrm{op}\ l_1 \\ \hat{\sigma}(l_0) = \mathsf{c_{ind}} \\ \hat{\sigma}(l_1) = \mathsf{c_{ind}} \end{array}}{\hat{\sigma}'(l) \leftarrow \mathsf{c_{ind}}} \qquad \textsc{Rel}\ \frac{\begin{array}{c} l \leftarrow l_0\ \mathrm{rel}\ l_1 \\ \hat{\sigma}(l_0) = \mathsf{c_{ind}} \\ \hat{\sigma}(l_1) = \mathsf{c_{ind}} \end{array}}{\hat{\sigma}'(l) \leftarrow \mathsf{b_{ind}}} \qquad \textsc{Bool}\ \frac{\begin{array}{c} l \leftarrow l_0\ \mathrm{bop}\ l_1 \\ \hat{\sigma}(l_0) = \mathsf{b_{ind}} \\ \hat{\sigma}(l_1) = \mathsf{b_{ind}} \end{array}}{\hat{\sigma}'(l) \leftarrow \mathsf{b_{ind}}}$$

$$\textsc{Read}\ \frac{\begin{array}{c} l \leftarrow v[l_0, \ldots, l_n] \\ \hat{\sigma}(l_0) = \mathsf{c_{ind}} \ldots \hat{\sigma}(l_n) = \mathsf{c_{ind}} \end{array}}{\hat{\sigma}'(l) \leftarrow \mathsf{c_{ind}}} \qquad \textsc{Merge}\ \frac{\begin{array}{c} \hat{\sigma}_0(l) = \hat{v}_0 \qquad \hat{\sigma}_1(l) = \hat{v}_1 \\ \hat{\sigma} = \mathit{merge}(\hat{\sigma}_0, \hat{\sigma}_1) \qquad \hat{v}_0 = \hat{v}_1 \end{array}}{\hat{\sigma}(l) \leftarrow \hat{v}_0}$$

Figure 5.4: Abstract semantics for different assignment statements and initial abstract states. State $\hat{\sigma}$ is the incoming abstract state while $\hat{\sigma}'$ is the updated state after the assignment. The path-predicate $\hat{\pi}$ for the rules is $\mathsf{b_{ind}}$. Lastly, *op*, *rel* and *bop* are arithmetic, relational and boolean operators, respectively.

so that the multiplier for the other operand is set as the final multiplier. Otherwise, the result is set to $\mathsf{c_{bsize}}$. This ensures that the set of symbolic values for the multiplier is limited to the set of variables in the program and we do not consider complex expressions on variables for the multiplier. While this is imprecise, it is necessary to scale the analysis.

A special scenario is that of writes to shared/global arrays $[v[l_0, \ldots, l_n] \leftarrow l]$, where the analysis checks if the accessed location, the value written and the path-predicate are independent of block-size, *i.e.* the values $\hat{\sigma}(l_0), \ldots, \hat{\sigma}(l_n)$ and $\hat{\sigma}(l)$ must be $\mathsf{c_{ind}}$ and the path-predicate $\hat{\pi}$ must be $\mathsf{b_{ind}}$. If this is not the case, the write is potentially a function of block-size and the analysis reports the write, and the kernel itself, to be *block-size dependent*. Also, this ensures the values in shared/global arrays are always block-size independent, and thus, the array reads return a consistent value in rule READ in Figure 5.4.

For conditionals [**if** l **then** S_1 **else** S_2], the analysis sets the path-predicates for S_1 and S_2 to $(\hat{\pi} \wedge \hat{\sigma}(l))$ and $(\hat{\pi} \wedge \neg\hat{\sigma}(l))$, respectively, while propagating the same initial abstract

state $\hat{\sigma}$ to both statements. Further, the final state after the conditional is a $merge$ of the states after S_1 and S_2. If the values for a variable are identical in both states (*i.e.* the type and the multiplier are equal), then this is set as the merged value for the variable (rule MERGE). Otherwise, the merged value is set to c_{bsize}/b_{bsize}. Also, the path-predicate after the conditional is set to the initial predicate $\hat{\pi}$.

The semantics for loops are defined similarly to conditionals, but we must additionally ensure that the analysis terminates. We observe that the set of abstract values is finite, with a small number of different value types where the multiplier for each type ranges over the finite set of local variables. Further, the $merge$ operation ensures that the merged state supersedes the input states, and is strictly "greater" than the input states (the set of abstract values and the $merge$ operation form a finite *upper semi-lattice*). Further, the abstract semantics are monotonic over the semi-lattice. Therefore, the fixed point computation on loops must terminate, and the analysis will reach the fixed point for the loop in a finite number of steps.

Algorithm. The overall algorithm is as follows. We initialize local variables to c_{ind}/b_{ind} in the initial abstract state $\hat{\sigma}$, while the path-predicate $\hat{\pi}$ is initialized to b_{ind}. The constants bid, bdim and tid are assigned values c_{bid}, c_{bdim} and c_{tid}, respectively, while gdim is independent of block-size and is assigned c_{ind}. The analysis executes the kernel for the abstract state $\hat{\sigma}$ and the path-predicate $\hat{\pi}$ with the abstract semantics defined above. If it encounters a potentially block-size dependent shared or global write, it terminates with block-size dependence. Otherwise, it reports the kernel to be block-size independent.

Inter-procedural analysis. Our analysis also supports inter-procedural analysis, where a kernel can call other kernels. We use a bottom-up analysis approach where the callees are analyzed before the callers. We analyze each kernel assuming the parameters are set to c_{ind}/b_{ind} initially and reuse this analysis result for all calls to the kernel with call arguments c_{ind}/b_{ind}. For calls with block-size dependent arguments, we conservatively report the call to be block-size dependent and return c_{bsize}/b_{bsize} as the return value. For library calls (where the library code is not linked) and inline assembly instructions, we conservatively assume the function to be block-size dependent and return value c_{bsize}/b_{bsize}. However, for specific cases, like library calls to Math functions `__sinf`, `__cosf`, `__sqrtf` etc., where we know the result is a trivial function of inputs, we assume the call to be block-size independent, and return c_{ind}/b_{ind} if the call-arguments are c_{ind}/b_{ind}.

Implementation. The implementation for this analysis is very similar to that for detecting uncoalesced accesses (Section 4.4). The primary difference lies in the representation of multipliers in the abstract domain. LLVM exposes each variable in the program as a unique `Value*` pointer. We use this pointer to represent the multiplier and to compare it against other pointers. Since LLVM uses the SSA form, the pointer corresponds to a unique definition and the value for the variable is not updated after it is first defined. Note the program variables that are accessed via `load/store` instructions do not appear as operands in regular arithmetic or boolean operations, and vice-versa. Hence, such variables are never used as multipliers in the abstract domain and the value for the multipliers is never updated through indirect `store` operations.

```
__global__ void cudaProcess(int imgw, unsigned g_odata[][imgw]) {
    // imgw ↦ c_ind
    int tx = tid[0]; int ty = tid[1]; // tx ↦ c_tid, ty ↦ c_ind
    int bw = bdim[0]; int bh = bdim[1]; // bw ↦ c_bdim, bh ↦ c_ind
    int x = bid[0]*bw + tx; // x ↦ (c_bid.c_bdim + c_tid) = c_ind
    int y = bid[1]*bh + ty; // y ↦ c_ind
    uchar4 c4 = make_uchar4((x & 0x20)?100:0, 0,
                            (y & 0x20)?100:0, 0); // c4 ↦ c_ind
    g_odata[y, x] = rgbToInt(c4.z, c4.y, c4.x);
}
```

Figure 5.5: Analysis for first grid-dimension on the program in Figure 5.1.

Example. We illustrate our analysis using the example in Figure 5.1. The resulting analysis run is shown in comments in Figure 5.5. We run the analysis separately for the two thread-grid dimensions. For the first thread-grid dimension, the analysis initializes variables as $\hat{\sigma}(\mathsf{bid}_0) = \mathsf{c}_{\mathsf{bid}}, \hat{\sigma}(\mathsf{bdim}_0) = \mathsf{c}_{\mathsf{bdim}}, \hat{\sigma}(\mathsf{tid}_0) = \mathsf{c}_{\mathsf{tid}}, \hat{\sigma}(\mathsf{bid}_1) = \hat{\sigma}(\mathsf{bdim}_1) = \hat{\sigma}(\mathsf{tid}_1) = \hat{\sigma}(\mathit{imgw}) = \mathsf{c}_{\mathsf{ind}}$. Also, it initializes the path-condition to $\mathsf{b}_{\mathsf{ind}}$, which is never modified. Next, it executes the statement $[\mathrm{tx} \leftarrow \mathsf{tid}_0]$, and sets $\hat{\sigma}(\mathrm{tx})$ to $\mathsf{c}_{\mathsf{tid}}$. It assigns values to variables $\mathrm{ty}, \mathrm{bw}, \mathrm{bh}$ similarly. Now when computing x, it first computes the product $\mathsf{bid}_0.\mathrm{bw}$ which is equal to $\mathsf{c}_{\mathsf{bid}}\mathsf{c}_{\mathsf{bdim}}$, and then computes x as the sum of values $\mathsf{c}_{\mathsf{bid}}\mathsf{c}_{\mathsf{bdim}}$ and $\mathsf{c}_{\mathsf{tid}}$, which we know is $\mathsf{c}_{\mathsf{ind}}$. The execution for the remaining statements continues similarly. Finally, the global write to image `g_odata` is executed with block-size independent abstract values and path-condition, and hence, the write is block-size independent. Therefore, the analysis declares the program block-size independent along this thread-grid dimension. The analysis repeats a similar process for other thread-grid dimensions and concludes the program to be block-size independent.

Correctness. We now show the correctness of our analysis. The analysis preserves the abstraction and ensures that each variable gets an abstract value $\mathsf{c}_{\mathsf{ind}}/\mathsf{b}_{\mathsf{ind}}$ only if the value is truly block-size independent *i.e.* the assigned value and the path-predicate are block-size independent. Further, each write to global variables is guarded by a check for block-size independence. Therefore, if the analysis does not report any block-size dependent writes, the updates to the global memory are always block-size independent, and the global state at the end of each thread's execution must also be block-size independent. This implies the program is thread-local block-size independent, and hence, we conclude the following theorem.

Theorem 5.6. *If the block-size independence analysis reports a synchronization-free GPU program $\mathcal{P}$ to be block-size independent, then $\mathcal{P}$ is* block-size independent.

Benchmark	# Kernels	# BSI	Benchmark	# Kernels	# BSI
Mandelbrot	6	0	concurrentKernels	2	0
simpleGL	1	0	eigenValues	4	0
convolutionSeparable	2	0	fastWalshTransform	3	2
cudaDecodeGL	2	2	FDTD3dGPU	1	0
dwtHaar1D	2	0	interval	1	0
histogram	4	0	mergeSort	7	3
recursiveGaussian	3	2	newDelete	16	8
simpleCUDA2GL	2	2	reduction	132	0
binomialOptions	1	0	scalarProd	1	0
BlackScholes	1	0	scan	3	0
MonteCarloMultiGPU	2	0	shfl_scan	4	0
quasiRandomGenerator	2	2	SimpleHyperQ	3	0
SobolQRNG	1	1	sortingNetworks	6	0
nbody	2	0	StreamPriorities	1	1
oceanFFT	3	2	threadFenceReduction	40	0
alignedTypes	12	12	threadMigration	1	0
cdpLUDecomposition	2	0	transpose	8	0

Table 5.1: Results of BSI analysis for Nvidia CUDA SDK 8.0 samples.

5.3 Evaluation

We have implemented the block-size independence analysis in LLVM 7.0, a popular open-source compiler framework, and evaluate it on the Nvidia CUDA SDK 8.0 sample programs. The SDK consists of 62 applications, out of which 28 benchmarks rely on texture memory fetches and the Thrust library and could not be compiled with LLVM. We therefore analyze the remaining benchmarks. For each benchmark, we analyze *global* kernels which are invoked directly from CPU code. For each global kernel, the analysis reports whether the kernel is block-size independent (BSI), and if not, the potential block-size dependent accesses in the kernel. Note that the kernels may not be synchronization-free, and hence, our analysis also checks this. We run the analysis on an Amazon EC2 machine with 4-core Intel Xeon 2.3GHz CPU and 16GB memory running Ubuntu 16.04 LTS (OS).

How many BSI kernels are found by the analysis? Table 5.1 shows the results for the analysis. The table shows the the total number of global kernels and the kernels found to be BSI. Note in a few benchmarks, the global kernels are instantiations of templated kernels, and hence, the numbers are slightly bloated. For example, in benchmarks “reduction”, “threadFenceReduction”, and “alignedTypes”, the total number of kernels is 132, 40 and 16, though these are instantiations of 7, 2 and 1 templated kernels, respectively. Yet, the analysis is able to verify a large number of kernels as BSI. It finds 35 BSI kernels in 11 benchmarks, and runs in a few seconds for most benchmarks (with a maximum of 100 seconds).

Are there truly non-BSI kernels? We manually investigated the benchmarks and found a few non-BSI kernels. These kernels asymmetrically distribute computation between blocks and threads, and hence, are block-size dependent. For example, benchmarks “binomialOp-

tions" and "MonteCarloMultiGPU" allocate an 'option' per block while the threads collaborate to compute the value for the option. Similarly, "scalarProd" allocates a vector-pair per block while the threads multiply and add individual elements to get the scalar product.

What class of kernels could not be verified? We could not verify block-size independence for kernels where shared memory and thread-synchronization were used to intricately share data between threads within a block. A common scenario was a parallel reduction operation such as summing elements. The block-size was hard-coded via `#define` constants for few of the kernels, which prevented verification. We observed an interesting pattern in benchmarks "dwtHaar1D" and "reduction" where each thread operated on two locations in a global array: $(2\mathsf{bid} \cdot \mathsf{bdim} + \mathsf{tid})$ and $(2\mathsf{bid} \cdot \mathsf{bdim} + \mathsf{bdim} + \mathsf{tid})$. The locations individually are block-size dependent. However, cumulatively, the threads operate on all elements, which makes the operation block-size independent. Finally, we could not verify kernels in "simpleGL", "oceanFFT" and "interval", because library calls containing assembly calls and addition between integers and booleans were inlined into kernels, which prevented the analysis from proving block-size independence.

Does tuning block-size for BSI kernels improve performance? We experimented with benchmark "SobolQRNG" to gauge performance improvement via block-size tuning. The benchmark originally used shared memory to cache some global constants and was reported non-BSI by our analysis. The block-size was set to 64 threads/block and produced 18.8 Gsamples/s (baseline) on an Nvidia GTX Titan X GPU. We removed caching to obtain a BSI version. Here for 64 threads/block, we lost performance by 40% (11.6 Gsamples/s), but then for 256 threads/block, we regained performance with an improvement of 9% over the baseline (20.5 Gsamples/s). Hence, our analysis helped tune block-size to gain performance while ensuring correctness, unlike the other optimization.

How many kernels could be easily fixed to become BSI? We fixed 7 kernels to be proven BSI by our analysis (included in the 35 BSI kernels found by the analysis). In "quasiRandomGenerator" and "fastWalshTransform", the number of blocks for the second grid dimension was set to 1, and thus bid_1 was always set to 0 and dropped from the computation for gid_1. In "cudaDecodeGL", gid_0 was computed as $(\mathsf{bdim}_0).(\mathsf{bid}_0 << 1) + (\mathsf{tid}_0 << 1)$, where the '$<<$' operator was not supported by our analysis. Finally, in "quasiRandomGenerator", gid_0 was computed as $(\mathsf{mul}(\mathsf{bid}_0, \mathsf{bdim}_0) + \mathsf{tid}_0)$, where the 'mul' method was not supported.

5.4 Related Work

Auto-tuning: A rich body of work exists on automatically tuning GPU applications for specific hardware configurations. Broadly, there are three types of auto-tuning: *empirical tuning* [77, 45, 55, 49, 62, 74], where different program variants are executed and the best variant is identified via exhaustive search or a hill-climbing approach; *model-based tuning* [13, 14], where a hand-crafted model is used to select the best program variant; and *predictive model-based tuning* [69, 45, 38, 46, 7], where a predictive model trained via machine learning techniques like decision trees is used to select the best program variant. All these approaches either automatically generate the final GPU program, or transform an existing program to generate the tuned program. A few of these works tune block-size di-

rectly [45, 46, 7, 74], but do not verify the correctness of the transformation. A few are domain-specific [77, 55, 14, 49, 62], often using programs written in a domain-specific languages instead of CUDA and OpenCL. Finally, many recent works focus on data-layout optimization [69, 38] and data placement [13]. These works segregate specification of data-layout and data-placement from the actual program by hiding it under a data-abstraction layer. Hence, only the spec for data-layout and placement is modified during auto-tuning and the program remains unchanged. This localizes any errors to the implementation of data-layout specifications, which ensures greater correctness. Tuning block-size is, however, essential to utilize resources on GPUs effectively, and our work on validating block-size independence can enable robust auto-tuning of block-size.

GPU Verification: Several systems exist for verification of GPU programs. GKLEE [44] and KLEE-CL [16] extend KLEE, a popular symbolic execution engine, to verify GPU programs against data-races and barrier divergence. Due to the presence of a large number of threads, these tools do not scale to large programs. GPUVerify [8] and PUG [43] improve upon GKLEE and KLEE-CL, by using *symbolic threads* and SMT-based verification to identify data-races. The underlying SMT solvers have trouble scaling to very large formulae as well. Finally, Leung et al. [42] present an approach where they analyze programs for *input-independence*, verify safety properties of input-independent programs for a small set of inputs and then generalize results to all other inputs. The analysis to verify input-independence is similar to ours, except that it tracks the flow of input variables rather than the block-size dependent constants.

Abstract Interpretation + Symbolic Execution: A few works, similar to our work, use symbolic constants to improve precision of an abstract domain, while retaining the scalability of the analysis. Sankaranarayanan et al. [59] and Venet [67] extend the Interval domain with symbolic ranges, where the upper and lower bounds of an interval are a linear combination of symbolic constants representing program variables. Miné [48] presents two generic techniques: *linearization*, which instantiates symbolic variables with abstract constants to obtain a linear expression in symbolic variables, and *symbolic constant propagation*, which propagates symbolic constants across expressions to gain precision.

5.5 Conclusion

This chapter presents the notion of block-size independence for GPU programs and an analysis to verify block-size independence for synchronization-free programs. The analysis uses a novel abstract domain that combines symbolic multipliers with abstract constants for different dependencies on block-size. The analysis is practical and finishes within a few seconds for most programs, while finding a large number of BSI kernels in Nvidia CUDA SDK 8.0 samples.

This work is a first step towards verifying block-size independence for GPU programs. In future, we would like to extend the analysis to more programs, by either transforming these into synchronization-free programs or ensuring the execution of each thread is independent of the set of threads it synchronizes with. Then, the present analysis for synchronization-free programs would be sufficient to verify block-size independence for the programs.

Chapter 6

Static Analysis for Improving Cache Reuse

Traditional compiler technology, developed through decades of research, is useful for improving performance of individual threads in a GPU program. The performance of a GPU program, however, is greatly dependent on cross-thread behavior, and if the threads don't operate in synchrony with each other, we see significant performance degradation, as in the case of uncoalesced accesses. The compiler technology to improve cross-thread performance is yet to catch-up. The challenges involve both reasoning about cross-thread behavior and transforming programs to improve cross-thread performance. Recent compiler approaches rely on the *polyhedral model* [27], where a mathematical representation is used to reason about and transform programs. These approaches, however, require programs to be well-structured with well-defined loops and array accesses. Various compiler optimizations for general GPU programs have been explored. However, they are often informal and flaky with few correctness guarantees owing to the complexity of the transformations involved. The programmer, therefore, often resorts to manual re-writing to improve program performance.

GPU hardware relies primarily on parallel threads to hide latency of compute operations and memory accesses, which results in a round-robin scheduling of threads. Round-robin scheduling, however, is detrimental for cache reuse since the cache is split across a large number of threads, due to which the data required by one thread is evicted by another thread before it is reused, leading to poor cache hit-rates. Recently, there have been efforts to balance cache reuse with thread-level parallelism in hardware by careful scheduling of threads [57, 39]. While it is possible by tracking reuse information within the hardware, this requires intricate hardware extensions which might be difficult to implement correctly. Further, the hardware scheduler needs to simultaneously observe the program execution and adjust the thread-schedule for cache reuse. Therefore, it may take some time to reach an optimal schedule.

A software analysis to detect cache reuse within GPU programs can help overcome drawbacks of hardware approaches. In particular, detecting reuse of data across iterations of a loop can be useful to improve performance for many "regular" programs. Two prominent examples are (1) matrix and vector manipulating programs, where each thread processes

elements in a row, and (2) data chunking programs, where each thread is assigned a small contiguous chunk of data from a large pool of data to be processed. In both these cases, each thread iterates over consecutive elements in a row/chunk and there is significant data locality between loop iterations. Further, consecutive threads access elements in distinct rows or chunks, and hence, the accesses are spread apart in memory leading to uncoalesced accesses. While data-layout transformation or redistribution of work can help, these approaches are not always applicable and also the code becomes difficult to maintain. In such scenarios, improving cache reuse within threads leveraging data locality across loop iterations can help improve program performance.

In this work, we present such a compile-time approach to optimize general GPU programs by improving cache reuse within threads. We present a *cache reuse analysis* (Section 6.2) that detects global memory accesses which can reuse cache data across loop iterations. The analysis relies on computing the maximum increment to the value of each variable across a loop iteration. If the increment to the accessed location during a global memory access is small, then the access is considered *loop-reusable*. Next, if there are sufficient number of loop-reusable uncoalesced global memory accesses (since uncoalesced accesses benefit significantly from cache reuse) within a kernel, we mark the kernel to be optimized for cache reuse. We further compute the required working set per thread for each such kernel and other necessary information within a *cache reuse predictor* (Section 6.3.1).

Updating the thread-schedule to improve cache reuse is not directly feasible within a compiler, since the GPU does not provide external knobs to control how threads are scheduled on the GPU. However, we observe that threads within a thread-block are often scheduled together and threads from the same block have priority over threads from other blocks. Hence, the set of *active* threads that execute on a GPU core at a time consist of threads from a *single* thread-block. Therefore, we resize (often reduce) the number of threads in each block, so that all threads in a block can effectively utilize cache for reusing data across loop iterations. This is however contingent on first proving that the kernel is also block-size independent, otherwise the block resizing is not allowed. To change block-size, we have implemented a *block-resize transformation* (Section 6.3.2) that inserts code to convert old block-size and grid-size into updated sizes. The transformation is easy to implement and involves changing block-size and grid-size only, which ensures robustness.

Both existing and new GPU programs can benefit from our optimization, where each thread is assigned a *coarse-grained* unit of work like a complete row in a matrix or a chunk of data, instead of a fine-grained unit with significant inter-thread locality. While it might be possible to transform the program to allot a fine-grained unit of work to each thread, this requires a lot of manual rewriting leading to programs which are not maintainable. Our approach presents a light-weight alternative to improve performance with minimal change to the existing program.

We have evaluated our approach on six benchmarks, three matrix manipulating and three data chunking-based benchmarks (Section 6.4). We have implemented our tool in LLVM 7.0 and we compare our optimization, clang-opt, with Nvidia's native compiler, nvcc, and the base LLVM-based CUDA compiler, clang-base. We evaluate on two Nvidia GPUs. We observe an average speedup of 1.3$\times$ across compilers and GPUs. We compare the perfor-

```
void Fan2(int N, int i, float M[N, N], float A[N, N], float B[N]) {
  int x = tid[0] + bid[0] * bdim[0];
  for(y = 0; y < N; y++) {
    if(x < N-i-1 && y < N-i) {
      A[x+i+1, y+i] -= M[x+i+1, i] * A[i, y+i];
      if(y == 0) {
        B[x+i+1] -= M[x+i+1, y+i] * B[i];
      }
    }
  }
}
```

Figure 6.1: 1D version of `Fan2` kernel in Figure 2.3.

mance improvements with cache-utilization performance counters and observe some interesting patterns. First, the change in cache utilization is observed to be directly proportional to the speedups obtained. Second, our approach performs particularly well when the kernel involves non-trivial code, for example, non-trivial matrix manipulation or imperfectly nested loops. We finally present some related work on existing program transformation and hardware-based approaches to improve performance (Section 6.5) and then conclude the chapter (Section 6.6).

6.1 Example: Revisiting Gaussian Elimination

We now illustrate the overall approach using an example kernel in Figure 6.1. This is a one-dimensional implementation of the `Fan2` kernel from Gaussian elimination program in Rodinia benchmark suite [10], shown in Figure 2.3. Note that we have substituted the y dimension of the thread-grid with a **for** loop. We observe that each thread accesses a distinct row in each of the accesses $A[x + i + 1, y + i]$, $M[x + i + 1, i]$ and $M[x + i + 1, y + i]$. For instance, thread with $tid_0 = 0$ and $bid_0 = 0$ accesses the $(i + 1)^{th}$ row, while the thread with $tid_0 = 1$ and $bid_0 = 0$ accesses the $(i + 2)^{th}$ row, for each of the accesses. We assume the matrices are laid out in a row major order. Hence, there is little locality across threads and simultaneous access by a warp of threads leads to an uncoalesced access. Next, for each of these accesses, a thread accesses consecutive elements in consecutive iterations of the y-loop. For instance, in the access $A[x + i + 1, y + i]$, the first iteration ($y = 0$) of a thread with $tid_0 = 0$ and $bid_0 = 0$ accesses location $(i + 1, i)$ and the second iteration ($y = 1$) accesses location $(i + 1, i + 1)$. Therefore, when data is fetched for the first iteration of the loop, we fetch not just the location required by the iteration but also the next few locations into cache, which can be reused by subsequent iterations. Thus, there is potential for cache reuse. Finally, since there is little locality across threads but significant locality within each thread, optimizing the kernel for cache-reuse against thread-level parallelism should improve performance.

We next show how our approach optimizes the kernel for cache reuse. It is easy to see that accesses $A[x + i + 1, y + i]$, $M[x + i + 1, i]$ and $M[x + i + 1, y + i]$ are uncoalesced, since x has unit linear dependence on tid_0, and the flattened access location for each of these accesses is of the form $(x + i + 1).N + c_0$, which leads to a non-unit dependence on tid_0. We next show that consecutive iterations of the y-loop access consecutive locations in each of these accesses. In the program, we observe that y is directly incremented by 1 across each iteration of the y-loop. Also, variables N, x and i are not modified inside the loop, and hence, are loop-invariant and have a increment of 0 across consecutive iterations. Next, the expression $(x + i + 1)$ has an increment 0, since each of the variables has an increment 0, and the expression $(y + i)$ has an increment 1, since y has an increment 1 and variable i is loop-invariant. Thus, the access $A[x + i + 1, y + i]$ has an overall increment of 1. The flattened location for the access is given by $(x + i + 1).N + (y + i)$ where N is loop-invariant. Therefore, the product $(x + i + 1).N$ is also loop-invariant since both individual values are loop-invariant. The sum of increments for $(x + i + 1).N$ and $(y + i)$ results in an overall increment 1. We similarly conclude the accesses $M[x + i + 1, i]$ and $M[x + i + 1, y + i]$ have an increment of 1 across each loop iteration. Consecutive iterations access consecutive locations for each of these accesses, and hence, the accesses can benefit from cache reuse. We call these accesses *loop-reusable*.

Each of the three accesses are both loop-reusable and uncoalesced. In a kernel, we are primarily interested in loop-reusable uncoalesced accesses, since such accesses take a long time to complete but benefit from cache reuse. Each access fetches a complete cache-line into the memory for each thread. Assuming the cache-line size of 128B and cache-size of 48KB per core, we need to retain cache-lines for each of the three accesses. Hence, we need at least $3 \times 128 = 384$B of cache memory per thread, which restricts the total number of active threads per core to cache-size divided by the cache-memory required per thread, which is $48 \times 1024/384 = 128$ threads. Hence, we must allow a maximum of 128 active threads per GPU core for each thread to benefit from cache-reuse. While this is difficult to enforce directly, we update the block-size for the kernel to 128 threads. Assuming the GPU thread-scheduler prioritizes threads within the block, there would be at most 128 threads actively executing on a GPU core which would allow effective cache reuse.

6.2 Cache Reuse Analysis

We first present an analysis to identify global memory accesses that benefit from cache reuse across iterations of a loop during the execution of a thread. An access benefits from cache reuse if the data fetched by a thread is reused by the same thread (intra-thread) or another thread (inter-thread) in a subsequent access. We focus on *intra-thread* reuse for instances of the *same* access across loop iterations. Inter-thread reuse can be useful to identify, especially if the reuse occurs across the second or third dimension of the thread-grid. Identifying inter-thread reuse can be difficult at compile-time, however, and poses problems similar to analyses for loop transformations to improve cache-locality in sequential programs. Also, updating the thread schedule for inter-thread reuse can be difficult in practice. Similarly, identifying reuse between two *distinct* accesses in a loop is difficult at compile-time and

would require tracking relationships between different index variables. This in-turn requires the loop indices and the access indices to be well-defined, which restricts the set of applicable programs. Hence, we focus on intra-thread cache reuse for the same global memory access across loop iterations.

A global memory access in a loop can benefit from cache reuse if the data fetched in an iteration of a loop can be reused in subsequent iterations of the loop. An access to global memory fetches a complete cache-line or memory block from memory. Hence, if the subsequent iteration accesses a neighboring element in the cache-line (*i.e.* exhibits spatial locality), the data fetched previously can be reused, and the access is cache reusable if the increment in the accessed location across consecutive loop iterations is less than the cache-line size. To identify such accesses, therefore, we need to track the increment in access index across consecutive loop iterations.

Our static analysis identifies cache reusable accesses one loop at a time. Given a loop, the analysis computes the maximum increment in values of variables, and index variables in particular, across a loop iteration. For a global memory access, if the increment in the accessed location is less than the cache-line size, then the access is considered cache reusable. The analysis runs in two passes through the loop. In the first pass, it scans instructions in the loop to identify instructions of the form $v \leftarrow v + c_0$ where v is a local variable and c_0 is a constant. Such variables are incremented by a value at least c_0, and the maximum of the sum of all such constants along a path is assigned as the *simple* increment for the variable. Note that we only track constants smaller than cache-line size, since larger increments are not useful for cache reuse. Also, if the variable is loop-invariant and not assigned any value inside the loop, we set the simple increment to 0.

In the second pass, the analysis uses the simple increments for variables to compute *derived* increments. Consider the statement $x = a + b$, where the simple increments for a and b are $\Delta a = 3$ and $\Delta b = 4$. It is easy to see that the increment for variable x is $\Delta x = \Delta a + \Delta b = 7$. We compute these increments via an abstract execution where the variables are initialized with simple increments or undefined values, and are updated with derived increment values during the execution of the loop. We note that this is similar to computing *induction* variables [15, 25, 17], useful in operator strength reduction. The techniques, however, compute precise increments, whereas we are interested only in increments smaller than the cache-line size which helps simplify the analysis and improve its efficiency.

Finally, if the access index increment for a global memory access is less than the cache-line size, the access benefits from cache reuse within the loop, and we refer to such accesses as *loop reusable accesses*. We formally define loop reusable accesses in Section 6.2.1. We then describe the analysis to compute simple increments in Section 6.2.2 and the analysis to compute derived increments in Section 6.2.3.

6.2.1 Loop Reusable Accesses

A global memory access AS to an array g is considered *loop-reusable* in a loop L, if for all threads τ, the execution of AS in an iteration of loop L can reuse cached data from the previous iteration of L. A GPU accesses data in global memory in units of η bytes, where

η is the bandwidth for global memory. Hence, when a memory location p is read from the memory, the whole cache-line consisting of η bytes along with the data at location p is fetched into an on-chip cache. If the location accessed in the subsequent iteration of the loop lies within the memory block fetched into the cache and the cached data is not over-written, the cached data can be reused to access the required data, which significantly improves the performance of the access. This is greatly useful for uncoalesced accesses where a large amount of unnecessary data is fetched during the access. Caching and reusing this data in subsequent iterations can help utilize this data and amortize the overall cost of executing the uncoalesced access across loop iterations.

We present here a formalization for loop-reusable accesses. Given an initial state σ_0 and a set of threads Π_0, we first define the set of configurations (σ, Π, S) reaching the i^{th} iteration of a loop L, $\mathcal{R}(i, \sigma_0, \Pi_0, L)$. We define this inductively. First, the initial configuration (σ_0, Π_0, L) is in $\mathcal{R}(0)$. For the recursive case, suppose a configuration (σ, Π, S) is in $\mathcal{R}(i)$. When S is a sequence $[S_1; S_2]$, then (σ, Π, S_1) is in $\mathcal{R}(i)$. Further, $([\![S_1]\!](\sigma, \Pi), \Pi, S_2)$ is in $\mathcal{R}(i)$ if the state after the execution of S_1 is not undefined. The reach sets for conditionals are defined similarly. When S is a loop, [**while** l **do** S'], first the configuration (σ, Π', S'), where $\Pi' = \{\tau \in \Pi : \sigma(l, \tau) = \text{true}\}$, is in $\mathcal{R}(i)$ if Π' is not empty. Next, if S is not the loop L, then the next iteration of S still belongs to the ith iteration of the loop L, and hence, the configuration $([\![S']\!](\sigma, \Pi'), \Pi', S)$ is in $\mathcal{R}(i)$ if not undefined. However, if S is the desired loop L, the next iteration of S corresponds to the next iteration of the loop, and therefore, $([\![S']\!](\sigma, \Pi'), \Pi', S)$ is in $\mathcal{R}(i + 1)$.

We now formally define a loop-reusable access. Given a loop L and a global memory access AS to array g at location $(l_0, l_1, \ldots, l_n)$ within the loop, consider two configurations $(\sigma, \Pi, AS) \in \mathcal{R}(i)$ and $(\sigma', \Pi', AS) \in \mathcal{R}(i{+}1)$ for an initial state σ_0, initial set of threads Π_0 and an iteration i. Consider a thread τ in $\Pi \cap \Pi'$. Consider the global memory locations accessed by τ in the two configurations, $q = \phi(g, \vec{p}(\tau))$ and $q' = \phi(g, \vec{p'}(\tau))$, where $\phi(v, \vec{q})$ returns the absolute location for the index $\vec{q}$ in array v, and indices $\vec{p}(\tau) = (\sigma(l_0, \tau), \ldots, \sigma(l_n, \tau))$ and $\vec{p'}(\tau) = (\sigma'(l_0, \tau), \ldots, \sigma'(l_n, \tau))$, respectively. If the distance between locations, $|q - q'|$, is less than cache-line size or the bandwidth for global memory η, then the thread τ can reuse cached data from i^{th} iteration in the $(i + 1)^{th}$ iteration. If this is true for all threads $\tau \in \Pi \cap \Pi'$ and all configurations (σ, Π, AS) and (σ', Π', AS), then the access AS is considered *loop-reusable* in L. Note that thresholding is sufficient to identify loop-reusable accesses, since most reusable accesses have a small increment when compared to the non-reusable acceses (similar to uncoalesced accesses).

Definition 6.1. Given a loop L and a global memory access AS in the loop to array g at location $(l_0, l_1, \ldots, l_n)$, consider two configurations $(\sigma, \Pi, AS) \in \mathcal{R}(i, \sigma_0, \Pi_0, L)$ and $(\sigma', \Pi', AS) \in \mathcal{R}(i + 1, \sigma_0, \Pi_0, L)$ for all initial states σ_0, initial sets of threads Π_0 and iterations i. Consider the locations accessed for the two configurations by a thread $\tau \in \Pi \cap \Pi'$, $q = \phi(g, \vec{p}(\tau))$ and $q' = \phi(g, \vec{p'}(\tau))$. If the distance between the locations $|q - q'|$ is less than η for all threads τ and all configurations (σ, Π, AS) and (σ', Π', AS), then the access AS is loop-reusable in

L. Formally, an access AS in *loop-reusable* in a loop L if:

$$\text{for all } \sigma_0, \Pi_0, i, (\sigma, \Pi, AS) \in \mathcal{R}(i, \sigma_0, \Pi_0, L), (\sigma', \Pi', AS) \in \mathcal{R}(i+1, \sigma_0, \Pi_0, L), \tau \in \Pi \cap \Pi',$$
$$|\phi(g, \vec{p}(\tau)) - \phi(g, \vec{p'}(\tau))| < \eta,$$
$$\text{where } \vec{p}(\tau) = (\sigma(l_0, \tau), \ldots, \sigma(l_n, \tau)) \text{ and } \vec{p'}(\tau) = (\sigma'(l_0, \tau), \ldots, \sigma'(l_n, \tau)).$$

6.2.2 Simple Increment Analysis

We describe our analysis to compute simple increments for variables. We first describe our abstract domain for tracking increments. Let the maximum value tracked by the analysis be the cache-line size or global memory line size η. We consider the following abstract values:

- $\bot$: the value is not defined.
- c_i : the value is a loop-invariant with constant value i.
- δ_i : the value has an increment with value at most i across one loop iteration.
- $\top$: the value corresponds to an arbitrary increment.

Note that, we track loop-invariant constants c_i along with increments δ_i. Loop-invariant constants are useful specially during a multiplication operation where a variable is multiplied by a constant k *i.e.* $v_1 = v_0 * k$. In such a scenario, the increment for v_1 is set to k times the increment for v_0. Finally, we only track the increment values for local variables in the program and the abstract state consists of the abstract valuation of local variables. Also, we do not track path-predicate or the condition under which a statement is executed during the analysis.

The analysis for simple increments starts in an abstract state with all variables initialized to undefined values $\bot$. It starts from the loop header, which is the entry block into the loop, and makes a single pass through the loop body. To ensure a single pass, we stop the execution whenever we reach the loop header again and the values in the state at this point represent the simple increments for variables. We define abstract semantics for only specific statements. For remaining statements, the same abstract state is passed to the next statement. We define the semantics in Figure 6.2.

We assign updated values such that the abstraction is preserved, *i.e.* variables with constant values c_i retain the constant value i across iterations, while the variables with increment values δ_j have an increment at most j across an iteration. The semantics mainly considers four scenarios. First, if a variable is assigned a constant k, its abstract value is updated to c_k (rule CONST). Next, if a variable is initially undefined and is incremented by value k (rule INCRUNDEF), we update the value to $\delta_{|k|}$, since the absolute value of the variable may be incremented by $|k|$ within each loop iteration during this assignment. If the variable is a constant c_i before the increment operation (rule INCRCONST), we update the value to c_{i+k}, since the variable is a constant c_i before the increment and remains a constant c_{i+k} after the increment. We track values only up to cache-line size η, and any value above η results in an unknown value $\top$. Finally, if the variable already has an increment δ_j, the new increment

$$\textsc{Const}\ \frac{v \leftarrow k}{\hat{\sigma}'(v) = c_k} \qquad \textsc{IncrUndef}\ \frac{v \leftarrow v + k \quad \hat{\sigma}(v) = \bot}{\hat{\sigma}'(v) = \delta_{|k|}} \qquad \textsc{IncrConst}\ \frac{v \leftarrow v + k \quad \hat{\sigma}(v) = c_i}{\hat{\sigma}'(v) = c_{i+k}}$$

$$\textsc{Incr}\ \frac{v \leftarrow v + k \quad \hat{\sigma}(v) = \delta_j \quad j + |k| < \eta}{\hat{\sigma}'(v) = \delta_{j+|k|}} \qquad \textsc{MergeUndef}\ \frac{v \leftarrow join(v_1, v_2) \quad \hat{\sigma}(v_1) = \bot \quad \hat{\sigma}(v_2) \neq c_i}{\hat{\sigma}'(v) = \hat{\sigma}(v_2)}$$

$$\textsc{MergeConstEq}\ \frac{v \leftarrow join(v_1, v_2) \quad \hat{\sigma}(v_1) = c_i \quad \hat{\sigma}(v_2) = c_i}{\hat{\sigma}'(v) = c_i} \qquad \textsc{MergeConstNeq}\ \frac{v \leftarrow join(v_1, v_2) \quad \hat{\sigma}(v_1) = c_i \quad \hat{\sigma}(v_2) = c_j \quad i \neq j}{\hat{\sigma}'(v) = \top}$$

$$\textsc{MergeIncr}\ \frac{v \leftarrow join(v_1, v_2) \quad \hat{\sigma}(v_1) = \delta_i \quad \hat{\sigma}(v_2) = \delta_j}{\hat{\sigma}'(v) = \delta_{max(i,j)}} \qquad \textsc{MergeIncrConst}\ \frac{v \leftarrow join(v_1, v_2) \quad \hat{\sigma}(v_1) = \delta_i \quad \hat{\sigma}(v_2) = c_j}{\hat{\sigma}'(v) = \top}$$

Figure 6.2: Abstract semantics for computing simple increments. Note k is a constant less than the maximum value. State $\hat{\sigma}$ is the abstract state before execution of the statement, while $\hat{\sigma}'$ is the state after the execution.

is at most the sum $(j + |k|)$, and hence, the variable is assigned an increment value $\delta_{j+|k|}$ (rule INCR).

Next, we consider the join or merge of values. This occurs when states flowing through different paths get merged. First, the undefined value $\bot$ is superseded by increment values and the unknown value (rule MERGEUNDEF). This is not true for constant values, however, because if the merged value is assumed to be a constant and is incremented subsequently, it is not guaranteed to remain constant in rule INCRCONST. Hence, we set merged value to value $\top$. Next, when two constants are merged, we return the same constant if they are equal (rule MERGECONSTEQ). Otherwise, since the variable might be assigned different constants in different iterations depending on the path taken, we return the unknown value $\top$ (rule MERGECONSTNEQ). Merge of two increment values results in assigning the maximum of the two values, since only one of the two increments flows into the merged value (rule MERGEINCR). Finally, the merge of an increment value and a constant value result in an unknown value $\top$, because depending on which path is taken in different iterations, the value might be incremented or reset to the constant value, leading to an unknown change to the value (rule MERGEINCRCONST). Finally, for the remaining cases, the resultant variable is assigned the value $\top$, especially if a variable is assigned a constant or increment value greater than η.

The above semantics assigns simple increment values to variables, where the variables are either directly assigned a constant value or they are directly incremented by a constant

$$\textsc{SumConst}\ \frac{v \leftarrow v_1 + v_2 \quad \hat{\sigma}(v_1) = c_i \quad \hat{\sigma}(v_2) = c_j \quad i + j < \eta}{\hat{\sigma}'(v) = c_{i+j}} \qquad \textsc{SumIncr}\ \frac{v \leftarrow v_1 + v_2 \quad \hat{\sigma}(v_1) = \delta_i \quad \hat{\sigma}(v_2) = \delta_j \quad i + j < \eta}{\hat{\sigma}'(v) = \delta_{i+j}}$$

$$\textsc{SumIncrConst}\ \frac{v \leftarrow v_1 + v_2 \quad \hat{\sigma}(v_1) = \delta_i \quad \hat{\sigma}(v_2) = c_j \quad i + j < \eta}{\hat{\sigma}'(v) = \delta_i} \qquad \textsc{ProdConst}\ \frac{v \leftarrow v_1 \cdot v_2 \quad \hat{\sigma}(v_1) = c_i \quad \hat{\sigma}(v_2) = c_j \quad i \cdot j < \eta}{\hat{\sigma}'(v) = c_{i \cdot j}}$$

$$\textsc{ProdIncr}\ \frac{v \leftarrow v_1 \cdot v_2 \quad \hat{\sigma}(v_1) = \delta_0 \quad \hat{\sigma}(v_2) = \delta_0}{\hat{\sigma}'(v) = \delta_0} \qquad \textsc{ProdIncrConst}\ \frac{v \leftarrow v_1 \cdot v_2 \quad \hat{\sigma}(v_1) = \delta_i \quad \hat{\sigma}(v_2) = c_j \quad i \cdot |j| < \eta}{\hat{\sigma}'(v) = \delta_{i \cdot |j|}}$$

$$\textsc{Undef}\ \frac{v \leftarrow v_1\ \mathrm{op}\ v_2 \quad \hat{\sigma}(v_1) = \bot \quad \hat{\sigma}(v_2) \neq \top}{\hat{\sigma}'(v) = \bot} \qquad \textsc{Unknown}\ \frac{v \leftarrow v_1\ \mathrm{op}\ v_2 \quad \hat{\sigma}(v_1) = \top}{\hat{\sigma}'(v) = \top} \qquad \textsc{FuncCall}\ \frac{v \leftarrow f(\dots)}{\hat{\sigma}'(v) = \top}$$

Figure 6.3: Abstract semantics for computing derived increments. State $\hat{\sigma}$ is the abstract state before execution of the statement, while $\hat{\sigma}'$ is the state after the execution. Semantics for $join$ operation are same as in Figure 6.2.

value. We also make an initial pass through all statements in the loop to identify variables that are read during the loop but not written to, and assign an increment value δ_0 since they are loop-invariant. We note for arrays, if any location in the array is written to, we conservately assume all locations might be written to, and hence, none of the reads to the array return a loop-invariant value. This completes our discussion of the computation of simple invariants. We next consider the computation of derived increments and a soundness argument for the computation of correct increment values for all variables in the loop.

6.2.3 Derived Increment Analysis

After computing the simple increments for variables, we run another analysis pass to compute the derived increments and the final increment values for all variables. The analysis pass initializes variables with their simple increment values (or undefined value $\bot$ if the simple increment was not computed). Next, it uses the semantics shown in Figure 6.3 to execute statements in the loop repeatedly, until the final state saturates for all statements and remains unchanged on executing another iteration. The final abstract state before each statement stores the increments for variables.

We now describe the semantics shown in Figure 6.3. We primarily consider two kinds of operations: sum $(+)$ and product $(\cdot)$. For the sum operation, if both variables have constant

values or both have increment values, we sum their values together to get the desired result (rules SUMCONST and SUMINCR). Note that if the resulting value is greater than η, we replace it with the unknown value $\top$. If one of the variables has a constant value while the other has an increment value, we set the result to increment value (rule SUMINCRCONST). This is because the constant has a zero increment across iterations, and thus, the overall increment for the result is given by the increment value for the other variable. Note that, this also ensures that simple increments for variables remain unchanged during an increment operation $v \leftarrow v + k$, since the constant k has a zero increment across iterations. Hence, no special semantics are necessary to retain simple increments of variables.

We next consider the product operation. If both operands have constant values, their values are multiplied together (rule PRODCONST). If both operands are increment values and the operands have zero increments δ_0, we set the result to δ_0 (rule PRODINCR). Otherwise, we set the result to value $\top$. This is because the increment for the result is given by $(v_1\Delta v_2 + v_2\Delta v_1 + \Delta v_1 \cdot \Delta v_2)$, and thus if either of the operands have a non-zero increment, the increment for the result depends on the absolute value for variables which is not known during the analysis. Finally, if one operand is a constant and other an increment value, the increment for result is amplified by the constant, and the result is assigned an increment value which is the product of the constant value and the increment value (rule PRODINCRCONST).

For all remaining operations and function calls, we set the result to the unknown value $\top$. We also note that if any operand is not defined, then the result is also not defined. Similarly, if one of the variables has an unknown value, then the result is unknown value. Therefore, for the $join$ operation, we use the same semantics as in the case of simple increment analysis.

Finally, after the analysis completes and the abstract state saturates, we check for each global memory access if the access is reusable across iterations. An access is reusable, if the increment in absolute location across a loop iteration is less than the cache-line size η. Given a global access to a location $(l_0, \ldots, l_n)$ in an array g, if local variables $l_0, \ldots, l_{n-1}$ have a zero increment value (or a constant value) and the product of increment value for variable l_n and the size of each element in g, $\xi(g)$, is less than η, then we consider the access to be loop reusable.

Correctness. We briefly discuss the correctness of our analysis. We claim the final increment values for variables overapproximate the actual increments to variables across any loop iteration. Our semantics for $join$ operation ensures that each variable is assigned the maximum increment along any path, and hence, we only need to ensure that the increment computed along a path is correct. The increment stored in each variable at the end of the derived increment analysis is a fixed point for the loop, and consists of the flow of increments from other variables. The source of these increments consists of the simple increments assigned to variables during the simple increment analysis and unknown values $\top$ assigned during arbitrary operations and function calls. We assert that if a variable is ever assigned an unknown increment value, the final increment value for the variable is set to unknown, since no operation (other than assignment to constants) can replace the unknown value with an alternate value (rule UNKNOWN in Figure 6.3). Therefore, a variable gets an increment value δ_i only if either it is directly incremented along the path or it is derived from other

simple increments. The simple increments assigned to variables are conservative, assuming the variables are not set to unknown value $\top$ during the derived increment analysis. Hence, the increments assigned to variables at the end of the derived increment analysis are sound and overapproximate the actual increments. This provides some evidence for the correctness of the analysis.

6.3 Overall Approach

We now describe the overall approach to determine cache reuse in a kernel K and to improve cache reuse via block-size transformation. First, we run the cache reuse analysis for each loop in K and collect all accesses that are loop reusable for some loop in the kernel. This set of accesses LR represents the potential for cache reuse within the kernel. If there are no loop-reusable accesses, then optimizing the kernel for cache-reuse will not be useful. Next, we compute the set of uncoalesced accesses UC in the kernel using the uncoalesced access analysis described in Chapter 4. The set UC on the other hand represents the set of accesses that have poor performance and can benefit from cache-reuse. Hence, the intersection of the two sets $\mathsf{UC} \cap \mathsf{LR}$ gives the desired set of accesses that need optimization and can benefit from cache reuse.

We further check if the kernel is block-size independent using the block-size independence analysis in Chapter 5. Given that the kernel is block-size independent and the set $\mathsf{UC} \cap \mathsf{LR}$ is non-empty, we compute the *working set* or the amount of data that must be cached by each thread in order to benefit from cache reuse. This is difficult to determine accurately at compile-time. However, we use heuristics to obtain an overapproximation. We present further details in Section 6.3.1.

We observe that threads within a block are likely to be scheduled together. Hence, the set of threads actively executing on a core consists of threads from a *single* thread-block. Therefore, we update block-size for each call to the kernel, such that the cache on GPU is sufficient to support all threads within a thread-block. We rely on a block-size transformation pass that inserts dynamic code before each call to the kernel to take in the previous grid configuration and generate the updated configuration. We describe this in Section 6.3.2.

The overall approach to optimize a *global* kernel K, which is a kernel directly called from CPU code, is shown in Figure 6.4. The approach first checks if the kernel is block-size independent. If so, it runs the cache reuse analysis and uncoalesced access analysis to generate sets LR and UC, respectively. Next, it invokes the cache reuse predictor (Section 6.3.1). The predictor checks if the intersection of the two sets is non-empty. If so, it first computes the working set for the kernel, and then based on the working set, computes the new block-size $\vec{B}'$ for the kernel. Finally, it invokes the block-size transformation pass (Section 6.3.2) to insert code before every call to the kernel K from CPU code, to update grid configuration with the new block-size. This completes the required transformation and the transformed IR is compiled down to generate an executable binary.

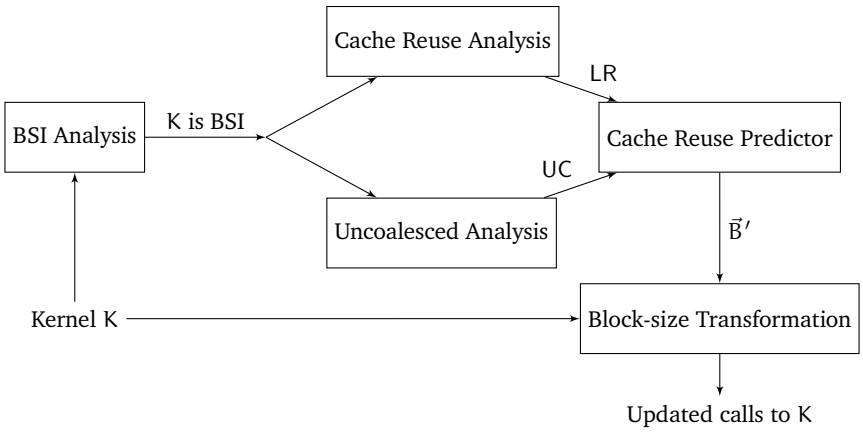


Figure 6.4: Overall approach to optimize a GPU program for a global kernel K for cache reuse. BSI Analysis checks if the kernel is block-size independent. Cache Reuse Analysis outputs loop reusable accesses in the kernel, LR. Uncoalesced Analysis outputs the set of uncoalesced accesses, UC. Cache Reuse Predictor checks if the intersection $\mathsf{LR} \cap \mathsf{UC}$ is non-empty. If so, it computes the working set per thread and the new block-size $\vec{B}'$. Finally, Block-size Transformation updates calls to K with the new block-size.

6.3.1 Cache Reuse Predictor for GPU Kernels

Before diving into the details of the cache reuse predictor, which predicts whether a GPU kernel has the potential for cache reuse, we briefly describe the caching sub-system inside GPUs. A significantly smaller portion of the chip is devoted to cache in GPUs as compared to CPUs. There are two primary forms of on-chip cache provided: L1 cache, which is local to each GPU core, ranging from 16kB to 48kB in size, and L2 cache, which is a system-wide cache common to all cores, ranging from 512kB to 2MB in size. L1 cache is often disabled since the cache hit-rates are low. We, on the other hand, rely on L1 cache to improve performance of global memory accesses, and hence, enable its use during compilation. L1 cache provides a fast local access and significantly improves performance of a program when used efficiently. Depending on GPU architecture, the caches are used both for reads and writes, or only the reads from global memory.

The predictor determines if a given kernel would benefit from optimizing for cache reuse. The optimization for cache reuse involves restricting the number of threads active at a time on a GPU core, so that sufficient portions of cache are available to all active threads. This is also popularly known as *warp throttling* [57]. Warp throttling directly impacts thread-level parallelism within a kernel. Therefore, we selectively choose kernels that benefit from cache reuse rather than thread-level parallelism. We target matrix manipulating and data chunking-based programs, where a thread accesses consecutive elements in consecutive iterations of a loop, but each thread operates on a distinct chunk of data and simultaneous accesses by threads are spread apart by the size of row/data-chunk. This often exhibits in programs with the presence of *loop reusable* accesses for access to consecutive elements in

consecutive iterations, and *uncoalesced accesses* for the simultaneous accesses to distant elements by threads. These programs have significant locality within a thread and almost no locality across threads, and benefit significantly from optimizing for cache-reuse. Hence, if the set $\mathsf{LR} \cap \mathsf{UC}$ or the set of uncoalesced loop reusable accesses within a kernel is non-empty, we optimize the kernel for cache reuse.

Working Set Computation. We next describe our heuristics to compute the working set or the amount of data that needs to be retained in cache per thread during the execution of a kernel, for effective cache reuse. We compute the working set for each loop L within the kernel and use the maximum of working sets for loops as the working set for the kernel, since cache reuse occurs for accesses within a single loop nest at a time. We observe that cache reuse often occurs in the innermost loop of a loop nest. Hence, we only consider the accesses in the current loop L while computing the working set for L and do not take into account accesses within sub-loops of L. We compute working set for a loop by considering *distinct* uncoalesced loop-reusable accesses in the loop. We observe that for each distinct access, we must retain a complete cache-line in memory. Hence, the overall working set per thread for a loop is defined by the product of the number of distinct accesses and the cache-line size η.

Computing the exact set of distinct uncoalesced loop-reusable accesses is infeasible at compile-time, because it is difficult to determine at compile-time whether two accesses point to the same location in memory. Hence, we rely on a heuristic where two accesses $g[l_0, \ldots, l_n]$ and $g'[l'_0, \ldots, l'_n]$ access the "same" location if they have the same *access signature*. We define the access signature as the set of unique symbolic variables used in the access expression. This consists of both the root array pointer g and the set of variables in expressions assigned to index variables l_0 to l_n. For instance the access signature for the access $g[a + b, c + 1]$ is given by the set $\{g, a, b, c\}$. We observe that distinct accesses often differ in the root pointer or at least one variable used in expressions for index variables. Hence, considering the set of symbolic values suffices.

A kernel may call other kernels, and hence, we also need to account for working sets of the *called kernels* while computing the working set for a kernel. We again observe that cache reuse occurs either in the current kernel or one of its callees, and not simultaneously in both a caller and the callee. Therefore, taking the maximum of the working sets of the current kernel and its callees suffices. We do a bottom-up computation of working set, so that the working sets for callees is computed before that for callers. Finally, the maximum of the working sets of all loops and callees within a kernel gives the desired working set per thread for the kernel.

Block Size Computation. Once the working set per thread is computed for a kernel, we next compute the new block-size for calls to the kernel. The essential idea is that threads from a single thread-block are often active on a GPU core, and hence, the cache is divided between the threads of a single thread-block. The cache space available for each thread should be larger than the computed working set for the thread. Hence, the maximum number of allowed threads in the block is equal to cache-size divided by the working set per thread. We assume all of cache is available to threads in a block (true for associative caches). However,

if that is not true, we can update the cache-size to reflect the total available cache. We currently work with programs with one-dimensional thread-blocks (hence higher dimensions are set to value 1) and update the smallest grid-dimension only. The block-size is often set to a power of 2 in GPU programs, and therefore, we set the new block-size to the largest power of 2 that is smaller than the maximum number of allowed threads.

6.3.2 Block Size Transformation for Cache Reuse Optimization

Given a global kernel K and a new block-size $\vec{B}'$, we describe the transformation to update calls to kernel K with the new block-size. The transformation inserts dynamic code before each call to K, to take the existing block-size $\vec{B}$ and grid-size $\vec{N}$ and return the updated block-size $\vec{B}'$ and grid-size $\vec{N}'$. When the grid-size $\vec{N}$ represents the total number of threads in the grid (as is the case for OpenCL), the new grid-size $\vec{N}'$ is same as the old grid-size. However in CUDA, the grid-size is represented by the number of blocks along each dimension which also needs to be updated when the block-size is modified. Hence, we update the grid-size such that the total number of threads is preserved.

LLVM Transformations. We have implemented the transformation in the LLVM compiler framework. LLVM supports compiling CUDA programs into executable binaries. The compiler can be configured with custom passes to be run during compilation. Command-line options can be specified which are forwarded to these custom passes. There is also provision to insert custom code (in separate files) during compilation. We present here a sample command-line to invoke a pass `xyz` in pass library `abc.so` with arguments $p = 10$, $q = 20$ and $r = 25$ and to insert functions in `custom.c` during the compilation of a CUDA program `gpu-prog.cu`:

```
clang++ -include custom.c
        -Xclang -load -Xclang abc.so
          -mllvm "-p=10" -mllvm "-q=20" -mllvm "-r=25"
        gpu-prog.cu
        --cuda-gpu-arch=<arch>
        -I/usr/local/cuda/samples/common/inc -L/usr/local/cuda/lib64
        -lcudart_static -lcuda -ldl -lrt -pthread
```

Note that we configure pass `xyz` such that it is automatically invoked by LLVM's pass manager when the pass library abc.so is loaded. The command-line arguments are automatically forwarded to the pass by the compiler.

Implementation. We now describe the the implementation for our block-size transformation pass. We have defined custom functions that take the old block-size and grid-size and the new block-size for a kernel call, and return the updated grid-size and block-size. For each kernel K that needs to be optimized for cache reuse, our transformation pass inserts calls to these custom functions to take the old grid configuration and return the updated grid configuration before each call to the kernel, based on the new block-size $\vec{B}'$. The set of kernels in a GPU program that must be transformed and the corresponding block-sizes

are passed to the transformation pass as command-line arguments. This information in-turn is scraped from a temporary text-file that is generated by a prior analysis where the cache reuse predictor is run to identify kernels that must be transformed by the transformation pass.

CUDA Programs. We have currently implemented the transformation for CUDA programs, where the grid-size corresponds to the number of blocks $\vec{N}_b$ instead of the total number of threads $\vec{N}$. We note that during the transformation, we need to ensure that the total number of threads remains unchanged. Thus, our custom functions check if the total number of threads along each dimension $(\vec{N}_b)_i.\vec{B}_i$ is a multiple of new block-size $\vec{B}'_i$. If so, the new grid-size is set to $(\vec{N}_b)_i.\vec{B}_i/\vec{B}'_i$. Otherwise, it returns the original grid configuration. Note that, since the grid configuration is only available at runtime, the check cannot be performed at compile-time and must be inserted in the dynamic code to transform the grid configuration.

6.4 Evaluation

Experimental Setup. We next describe some evaluation for our approach. We have implemented it in LLVM. We evaluate it on 6 benchmarks, three matrix manipulating and three data chunking-based programs. Matrix manipulating programs perform non-trivial operations on vectors and matrices, while the data chunking-based programs split the data into chunks and each thread is assigned one chunk. In both types of programs, there is little locality across threads and significant loop-level locality within the execution of each thread. We evaluate our approach on two real GPUs: Nvidia Tesla M60 (Maxwell) and Quadro P4000 (Pascal). We compare our approach against the native Nvidia compiler `nvcc`, the base LLVM compiler `clang-base`, and LLVM with our optimization `clang-opt`. We use CUDA version 9.2 and compile all benchmarks with the `-O2` flag. For each kernel optimized with our approach, we collect the running time (average of five runs), the average cache hit rate and the global load throughput. We use a Linux system function `gettimeofday()` to collect the running time, while rely on Nvidia's profiler *nvprof* to collect data using metrics `global_hit_rate` and `gld_throughput`.

We briefly decribe the benchmarks. Kernel `Gaussian/fan2` is a 1D kernel from a "Gaussian elimination" program in Rodinia benchmark suite [10], shown in Figure 6.1. The remaining kernels are from the Mars benchmark suite [30]. Mars is a map-reduce framework implemented in CUDA, where the overall work is divided into tasks and each thread is assigned a task to perform. We primarily look at the *map* and *mapCount* operations and the corresponding kernels in the program. The task itself consists of a contiguous chunk of data which is processed by each thread iterating over each data element in a loop. Two matrix-manipulating programs "matrix multiplication" (`mm`) and "similarity score" (`ss`) and three data chunking-based string manipulating programs "inverted index" (`ii`), "word count" (`wc`) and "page view count" (`pvc`) are implemented in this framework.

Evaluation process. In the evaluation, we are concerned with the following questions: (1) Does our optimization improve performance for GPU kernels? (2) What types of benchmarks benefit from our optimization? (3) How are the speed-ups correlated with improvement in cache reuse and effective memory throughput? Note that we launch each benchmark with a large input size, so that there are enough thread-blocks to fill all GPU cores, even before our optimization is applied. Otherwise, the speedup might be obtained by filling up of cores due to increase in the number of thread-blocks after the optimization. For most programs, the block-size is shrunk to increase the number of blocks. However, in some rare cases the block-size might be increased, though we assume that the programs have been optimized for thread-level parallelism with large block-sizes. We will address the scenario of optimizing for thread-level parallelism in future.

Also, we made slight modifications to the Mars framework to ensure kernels are block-size independent, and therefore, our optimization is applicable. The framework previously allocated tasks to threads in a round-robin fashion. We modified it to instead allocate contiguous sets of tasks to each thread. Second, the framework relied on shared memory buffers to store intermediate data. We substituted this with storing data in registers and local memory. These changes were necessary to prove block-size independence of programs in the framework.

Runtime improvements. Figure 6.5 shows the speedups obtained for different kernels on the two GPUs. Overall, the kernels compiled with clang-opt have an average speedup of $1.317\times$ and $1.34\times$ vs nvcc, and $1.16\times$ and $1.347\times$ vs clang-base, on Tesla M60 and Quadro P4000 GPU, respectively. On both GPUs, we obtain significant speedups for `pvc/map`. The speedup against nvcc provides an estimate of the speedup against a production compiler used by a large fraction of GPU developers, while the speedup against clang-base provides an estimate of how well our optimization performs compared to the base compiler. The fact that the original kernels were not modified and the speedups are obtained on *real* GPUs makes these results quite significant.

We now discuss the benchmarks and the corresponding speedups in more detail. We compare speedups over the base LLVM compiler, since they present a better estimation of the speedup for our optimization. We first discuss kernels from the three matrix manipulating programs. Kernel `Gaussian/fan2` kernel performs row operations on a matrix where each thread is assigned a row of the matrix and iterates over the elements in the row. Kernels `mm/map` and `ss/map` perform standard matrix operations similar to matrix multiplication. We observe that the performance improvement for `Gaussian/fan2` kernel is slightly higher than that for the other two kernels. This is probably because the non-trivial row operations in this kernel prevent automatic optimization by the compiler/hardware.

Next, we consider kernels from the string manipulating programs. In each of these kernels, threads are assigned a segment of a string file and threads iterates over the characters in the segment to extract information like HTML links, word count and number of page views. Again, we observe that the speedup for `pvc/map` is significantly higher than the other two kernels. We investigated code for this kernel and observed that the loop iteration over characters occurs inside a branch of a conditional, whereas for the other benchmarks, the

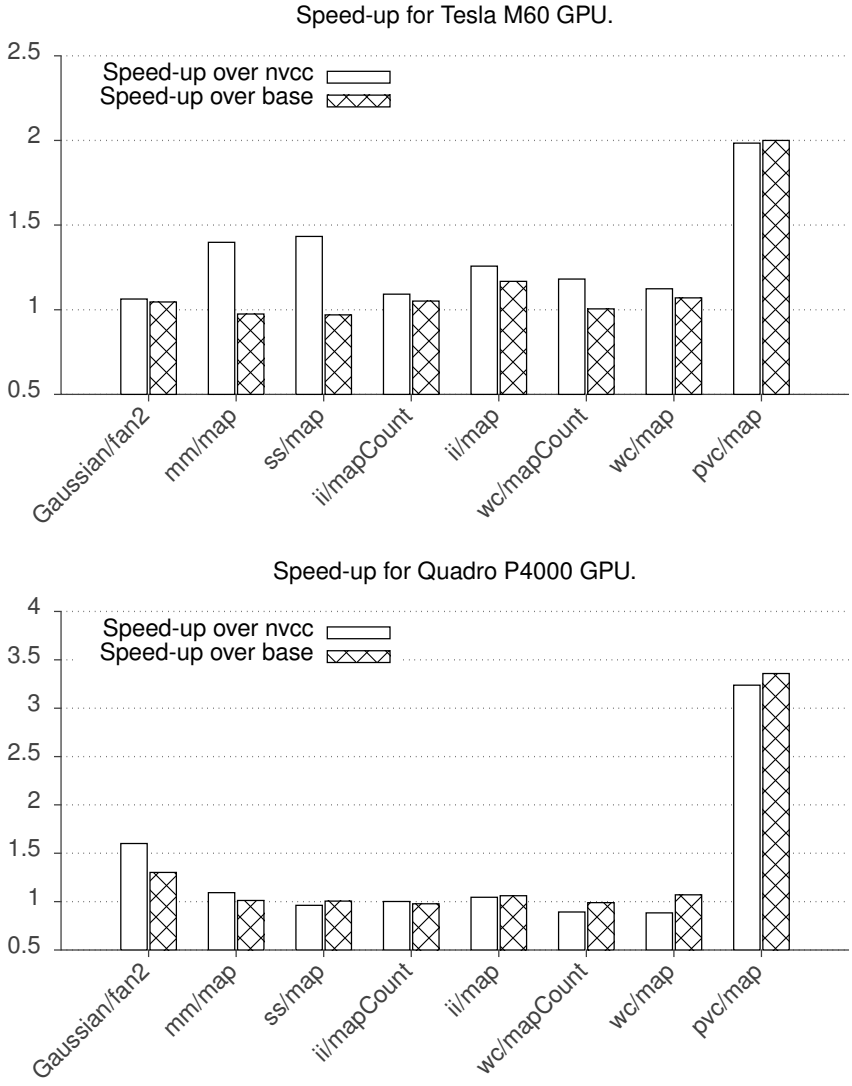


Figure 6.5: Speedup for our optimization clang-opt over Nvidia's compiler nvcc and base LLVM compiler clang-base for different kernels on the Tesla M60 and Quadro P4000 GPUs.

code consists of perfectly nested loops. It appears that perfectly nested loops are easier to optimize automatically. Further, kernels `ii/map` and `ii/mapCount` consist of non-trivial case analysis within a loop, whereas `wc/map` and `wc/mapCount` consist of a simple iteration over the characters in a loop, which is better optimized by existing compiler/hardware. Hence, lower speedups are observed for `wc/map` and `wc/mapCount`. Overall, it seems this class of benchmarks benefits more from our optimization.

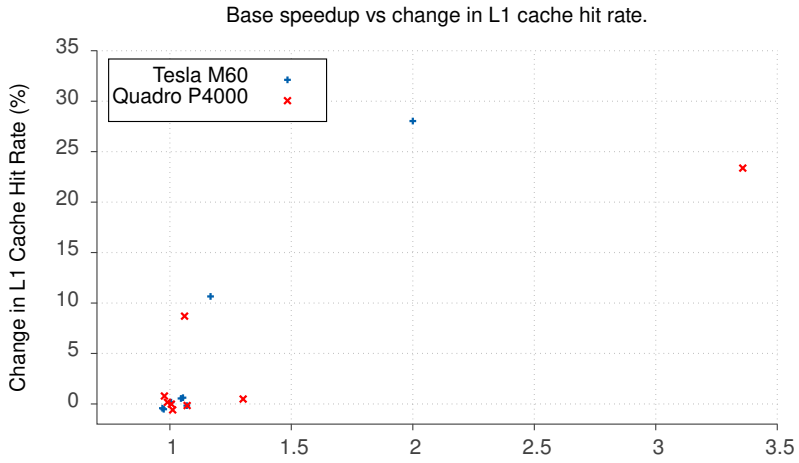


Figure 6.6: Change in L1 cache hit rate against the speedup for our optimization clang-opt over the base LLVM compiler clang-base. The '+' points represent the results for Tesla M60 GPU, while the '×' points represent results for Quadro P4000 GPU.

L1 cache hit rate. Figure 6.6 compares the change in cache hit rate with the speedup over base LLVM compiler. Clearly, for large speedups, the change in cache hit rate is also high for both GPUs. The correlation seems to be stronger for Tesla M60 as compared to Quadro P4000. Overall, there is a positive correlation between change in cache hit rate with the speedup obtained which indicates our optimization seems to improve the cache hit rate while also improving the performance.

Global load throughput. Figure 6.7 compares change in global load throughput against the speed of our optimization over clang-base. Again, similar to L1 cache hit rate, there is a positive correlation between global load throughput and the speedup. Global load throughput measures the effective throughput for global memory load accesses, which includes hits in L1 and L2 cache. Clearly, a positive correlation indicates that our optimization improves throughput, and therefore, leading to speedups.

6.5 Related Work

We describe some related work, covering various existing program transformation and hardware approaches to improve performance. We describe our work in context of these works. We first describe progam transformation approaches, where the compiler transforms the program into an optimized version, without relying on hardware support for performance. We next discuss approaches to improve performance by augmenting hardware.

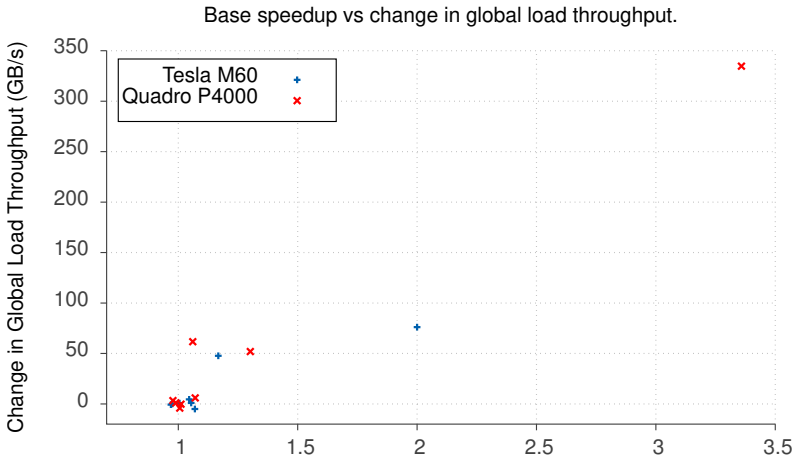


Figure 6.7: Change in global load throughput against the speedup for our optimization clang-opt over the base LLVM compiler clang-base. The '+' points represent the results for Tesla M60 GPU, while the '×' points represent results for Quadro P4000 GPU.

Program transformation-based approaches. There are broadly two types. We first consider approaches that transform the data-layout of arrays and structures to gain performance. The data-layout defines how elements are laid out in memory and determines how the memory accesses are carried out and whether they are coalesced or not. Dymaxion [12, 11] presents a user-directed approach, where the user specifies the data-layout tranformation that must be applied to a data-structure, as API calls or pragmas. The transformations include rows to columns or diagonal elements into rows for matrices, indirect mapping to map old locations in arrays to new ones through an index array, and array-of-structures to structure-of-arrays for data structures. Zhang et al [76] describe a similar approach, except the transformations are automatically determined at run-time, which is useful for irregular programs where the irregularities in data-layout are not known at compile-time. The transformations dynamically move data elements or redirect references from threads to data-elements. Sung et al [64] present another automatic approach which operates at compile-time and is useful for regular programs. They however rely on array definitions and accesses to be well-structured. The approach automatically computes a transformation to ensure all accesses to a data structure are coalesced by remapping thread-ids to *steering* bits in array accesses. In all of the approaches, the transformations are performed at run-time when data is transferred from CPU to GPU, and the latency is hidden by pipelining tranformations with data-transfer. Note that a single transformation is implemented per data-structure for one kernel call. These transformations require complex run-time analysis for irregularities or user-guidance in terms of pragma's and well-defined array accesses. Our approach, on the other hand, requires minimal runtime analysis and modifications with no user-guidance.

The other set of program transformation-based approaches modify the GPU program itself. They rely on simple techniques like transforming program to cache data in shared

memory and reordering loops and thread-geometry. The techniques are also accompanied with analyses to identify performance issues and to search optimal transformations. CUDA-lite [66] presents an approach to optimize uncoalesced accesses in programs by transforming program to cache the accesses in shared memory. This involves loop tiling, transferring data between global memory and shared memory, and substituting global memory accesses with shared memory accesses. The technique relies on significant user annotation for arrays which are analyzed, array bounds and dimensions, and loop iterators and their bounds. Yang et al [75] present a comprehensive compiler that performs many transformations to optimize CUDA code. Similar to CUDA-lite, they optimize uncoalesced accesses by caching data in shared memory. They further substitute `float` variables with `float2` variables. They apply thread-block merging and thread merging to reduce local and shared memory usage. Both CUDA-lite and this work rely on simple compiler analyses and transformations to gain performance. On the other hand, Bhaskaran et al [6] use sophisticated polyhedral model-based analysis to automatically identify loop and thread-geometry transformations. They set up execution-order and spatial-locality constraints and search for a valid reordering transformation that optimizes uncoalesced accesses. They however rely on loop-bounds and array accesses in programs to be well-structured (*affine* functions of thread-ids and loop-iterators). Jang et al [31] present a similar approach, however they use their own representation, rather than the polyhdedral model, to analyze and transform programs. All of the above approaches require extensive transformation of programs, which can be both difficult and error-prone. Our approach in comparison requires minimal transformation which makes it robust and easy to implement.

Hardware-based approaches. The hardware-specific approaches improve performance via two techniques. First, they control the global memory traffic to avoid congestion due to a large number of simultaneous requests from multiple threads. Second, they improve cache utilization, similar to our approach. These approaches rely on additional hardware support, not available in current GPUs. Some of these are purely hardware-based techniques, while others also involve a compiler component, similar to our approach.

We first describe some approaches that tweak hardware scheduler to improve performance. CCWS [57] extends the GPU warp-scheduler to utilize intra-warp spatial locality for effective cache reuse. It dynamically controls the set of warps that are active on the core, so that the cache is effectively reused by warps. It maintains a lost-locality score for each warp, and prioritizes warps that are losing locality and encountering cache misses, to prevent them from losing locality in future. OWL [34] and Narasiman et al [50] use multi-level warp scheduling to prevent bursts of requests to global memory and thus avoid congestion. The scheduler first selects a group of warps to be scheduled and then a warp within the group. It prioritizes warps within the group for scheduling, and moves on to another group only when all warps in the previous group are stalled. This ensures that the memory requests sent to the global memory are restricted to one group at a time which prevents congestion. Also, the cache is divided between warps of only one group which leads to better utilization. Mascar [61] presents a similar approach, however it prioritizes requests from a single warp when it detects congestion, and uses round-robin scheduling otherwise when there is no

congestion. This helps maximize thread-level parallelism in the general case, and prioritizing a single warp helps faster recovery from memory congestion. MRPB [32] and Rhu et al [56] augment the memory subsystem to prioritize requests from certain threads and bypass cache for memory requests that don't have enough reuse. Finally, APCM [39] presents a warp scheduling strategy that identifies cache reusable memory accesses by dynamically observing the behavior of a single warp in hardware, and then using this information to schedule remaining warps such that the cache is effectively reused. Our approach similarly tries to control the scheduling of threads and memory requests to global memory by updating the number of threads in a block. However, since the analysis happens at compile-time, we don't need hardware extensions to predict cache reuse and to determine the scheduling strategy. Also, our analysis predicts cache reuse at the level of loops, which might be difficult in hardware if the loop body is large. Yet, our approach can effectively complement the above approaches to improve overall performance.

We next consider some compile-time approaches, similar to our approach, that utilize hardware features or extensions to improve performance. Jia et al [33] present an algorithm to determine if an access is coalesced or not. If it is colaesced, they use 128B cache-line to fetch data, and otherwise, they bypass cache and use a 32B memory request. The algorithm is manually applied and not embedded into a compiler. Xie et al [73] present a compiler framework with light-weight profiling to identify locality between instructions, and hence, their potential for cache reuse. They select a set of instructions with good locality, and use cache to store data for these instructions and bypass cache for remaining instructions. Gong et al [26] present a compiler framework to compile each kernel into two different binary versions using different instruction schedulers. Each warp in the thread-grid is assigned one of these versions, and the hardware is tweaked to support simultaneous execution of the two versions. The primary insight is that different instruction schedules lead to spreading of memory requests in time which prevents memory congestion, and hence, results in better performance. Our work is complementary to these approaches and can benefit from ideas presented in these works.

6.6 Conclusion

This chapter presents a compile-time framework to improve performance of GPU programs through effective cache reuse. It presents an analysis to identify accesses that benefit from caching, a predictor to compute appropriate block-size for kernels that must be optimized, and a transformation pass to update the block-size. The framework leads to minimal code transformations at compile-time, which makes it robust and easy to implement. Further, it requires no hardware changes and produces an average speedup of $1.3\times$ on two real GPUs.

In future, we would like to extend this work to multi-dimensional thread-grids, especially when consecutive threads along a grid dimension access consecutive locations in memory. A similar analysis as the cache-reuse analysis would be useful, however non-trivial changes to thread-grid configuration might be neeeded. We would also like to extend the idea of utilizing existing hardware features to improve performance, while making minimal source-code changes, to other goals like reducing energy usage in GPU programs.

Chapter 7

Concluding Remarks

This thesis explores a new approach to analysis and optimization of GPU programs. The analyses are light-weight and execute efficiently at compile-time. Yet, they are robust and based on formal abstraction-based techniques. Similarly, the optimization leverages hardware cache to improve performance, while ensuring minimal transformation which again gives greater guarantees of correctness. Further, the analyses and the optimizations work with general GPU programs and do not impose constraints on how the programs must be written.

This is unlike previous work in analysis and optimization of GPU programs. The existing analyses for GPU programs are either described informally as part of compile-time optimization frameworks which diminishes their reliability (for example in CUDA-Lite [66]), or they rely on complex techniques like symbolic execution which take a long time to run (for example in GPUVerify [8] and GKLEE [44]). We present light-weight analyses based on abstraction and reason about their correctness via a formal model for GPU programs. Abstraction filters necessary information from program state which helps scale the analysis, yet formally captures the execution of the analysis via an abstract state and abstract semantics which allows reasoning about correctness. For GPU programs, a novel abstraction that tracks the *dependence* of program variables on thread descriptors like thread-id, block-id and block-size is required, which has not been used previously and distinguishes our analyses from existing work.

Similarly, compiler optimization to improve scheduling of threads within GPUs has not been explored previously. While this has been explored in hardware [57, 39], there are limited opportunities to identify cache reuse due to the narrow visibility into program execution, especially for regular programs where there is significant cache reuse across loop iterations. Also, simultaneous detection of cache reuse and optimization of thread-schedule can be difficult in hardware. Hence, the hardware presents limited opportunities to improve thread schedule to leverage such cache reuse. Further, existing compiler optimization frameworks transform the data-layout or the program itself to improve performance, which requires non-trivial changes like reordering loops and access indices, tiling loops, pre-fetching data in user-managed cache etc. These changes are difficult to implement robustly for general GPU programs where the loops and memory accesses are not well-formed. We, on the other hand,

leverage the fact that the order in which different threads execute is flexible, and hence, can be reconfigured for better cache reuse without implications to correctness. Therefore, our optimization overcomes limitations of existing work, via compile-time analyses that predict loop-level cache reuse which is difficult in hardware, and minimal program transformation which gives greater guarantees of correctness than existing compilers.

We have implemented the analyses and optimization in LLVM. The implementation is modular and easy to work with. The core abstract execution engine has been extracted into a generic framework, which is then extended by each our analyses to implement the required functionality. This simplifies the implementation and also makes it easy to define new analyses. Next, the build is completely automated and the tool is easy to install on a new system. We have also written automated scripts to invoke our tool on benchmarks. The tool has been artifact evaluated for the uncoalesced access analysis and the block-size independence analysis, which further demonstrates its robustness.

Furthermore, we have results demonstrating usefulness of our work in practice. The analysis to detect uncoalesced accesses found 111 uncoalesced accesses in Rodinia benchmark suite with a false positive rate of 38%. The analysis is fast, finishing in few seconds for most programs. Similarly, the analysis to prove block-size independence identifies 35 block-size independent global kernels in Nvidia CUDA SDK samples and again finishes in a few seconds for most programs. Finally, our compiler optimization to reuse cache in hardware shows an average speedup of $1.3\times$ on six matrix manipulating and data chunking benchmarks on two real Nvidia GPUs. Overall, our analyses and optimization are fast, robust and applicable to real world GPU programs.

Our work also comes with certain limitations. First, we assume the programs being analyzed or optimized are *correct* and devoid of concurrency issues. Any guarantees fail if the original program is not data-race-free or barrier-divergence-free. Second, the block-size independence analysis and cache reuse-based optimization work only for a small class of programs currently. Finally, our cache reuse optimization has been evaluated on a small set of benchmarks. We will address these limitations in future work. Despite that, each of these works break ground into a new and practical dimension of GPU compiler research with significant scope for improvement.

7.1 Future Directions

Each work in this thesis, on finding uncoalesced accesses, proving block-size independence and leveraging cache reuse for compile-time optimization, only scratches the surface and there is scope for improvement, which we would like to address in future work. There are also other possible directions that we discuss shortly. Broadly, we would like to improve the state-of-the-art in programming tools for *general* GPU programs, written in programming languages like CUDA and OpenCL. These languages have gained wide spread adoption and cater to general audiences. We are also open to extending other existing languages (especially functional languages) to support GPU programming. We describe here various directions we would like to pursue in future.

- **Identifying properties of GPU programs.** We would like to build on our experience in analysis for uncoalesced accesses to identify other properties of GPU programs, particularly performance and correctness issues like shared memory bank-conflicts and data-races. The essential idea is to track relationship between the executions of threads via an appropriate abstraction. We believe this approach could be useful for identifying other properties. An important application would be detection of data races where most of the existing static approaches rely on heavy-weight SMT solving and do not scale to large programs. A light-weight approach based on abstraction to detect data-races would be desirable.

- **Block-size independence.** This is an important property necessary to ensure robust tuning of block-size in GPU programs. Currently, we handle a small subset of synchronization-free programs. We would like to extend the analysis to other GPU programs with restricted synchronization.

- **Cache reuse optimization.** We propose a novel compile-time approach to balance cache reuse with thread-level parallelism via improved thread-scheduling, and hence improve performance. The cache reuse analysis and the cache reuse predictor can be improved to predict other types of cache reuse like inter-thread reuse and reuse across different global memory accesses.

- **Traditional compile-time optimizations.** There is scope to improve the robustness and applicablity of classic optimizations like loop-reordering, loop-fusion and loop-fission to general GPU programs. There is also scope for new formal methods that allow such optimizations to be implemented on generic programs while ensuring correctness. There is a rich body of work done for sequential programs. GPU programs, however, exhibit greater regularity, and therefore, many existing techniques can be adapted for better performance on GPU programs.

- **New languages for GPU programming.** The existing languages for GPU programming present many challenges to automatic analysis and optimization. The challenges arise due to the absence of high-level code structure (like loops and conditionals), presence of aliases and side-effects, and non-trivial transformations necessary for improving cross-thread performance. A few of these challenges can be overcome with better support from hardware, like compiler-level control for tuning thread-scheduling etc. A fresh look at both hardware and software for GPU programming could be useful in making it robust and accessible to masses.

With that, we would like to conclude this thesis. We hope it is an informative and enjoyable read. We also hope it opens up news ideas for exploration and enables new research on tools for robust and efficient GPU programming.

Bibliography

[1] Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 147–160. POPL '99, ACM, New York, NY, USA (1999), http://doi.acm.org/10.1145/292540.292555

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. pp. 265–283. OSDI'16, USENIX Association, Berkeley, CA, USA (2016), http://dl.acm.org/citation.cfm?id=3026877.3026899

[3] Allen, F.E.: Control flow analysis. SIGPLAN Not. 5(7), 1–19 (Jul 1970), http://doi.acm.org/10.1145/390013.808479

[4] Alur, R., Devietti, J., Leija, O.S.N., Singhania, N.: GPUDrano: Detecting uncoalesced accesses in GPU programs. In: CAV (2017)

[5] Amilkanthwar, M., Balachandran, S.: CUPL: A compile-time uncoalesced memory access pattern locator for CUDA. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. pp. 459–460. ICS '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2464996.2467288

[6] Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: A compiler framework for optimization of affine loop nests for GPGPUs. In: Proceedings of the 22Nd Annual International Conference on Supercomputing. pp. 225–234. ICS '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1375527.1375562

[7] Bergstra, J., Pinto, N., Cox, D.: Machine learning for predictive auto-tuning with boosted regression trees. In: 2012 Innovative Parallel Computing (InPar). pp. 1–9 (May 2012)

[8] Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: A verifier for GPU kernels. SIGPLAN Not. 47(10), 113–132 (Oct 2012), http://doi.acm.org/10.1145/2398857.2384625

[9] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: Stream computing on graphics hardware. In: ACM SIGGRAPH 2004 Papers. pp. 777–786. SIGGRAPH '04, ACM, New York, NY, USA (2004), http://doi.acm.org/10.1145/1186562.1015800

[10] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC). pp. 44–54 (Oct 2009)

[11] Che, S., Meng, J., Skadron, K.: Dymaxion++: A directive-based api to optimize data layout and memory mapping for heterogeneous systems. In: Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops. pp. 916–924. IPDPSW '14, IEEE Computer Society, Washington, DC, USA (2014), http://dx.doi.org/10.1109/IPDPSW.2014.104

[12] Che, S., Sheaffer, J.W., Skadron, K.: Dymaxion: Optimizing memory access patterns for heterogeneous systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 13:1–13:11. SC '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2063384.2063401

[13] Chen, G., Wu, B., Li, D., Shen, X.: PORPLE: An extensible optimizer for portable data placement on GPU. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 88–100. MICRO-47, IEEE Computer Society, Washington, DC, USA (2014), http://dx.doi.org/10.1109/MICRO.2014.20

[14] Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 115–126. PPoPP '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1693453.1693471

[15] Cocke, J., Kennedy, K.: An algorithm for reduction of operator strength. Commun. ACM 20(11), 850–856 (Nov 1977), http://doi.acm.org/10.1145/359863.359888

[16] Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic testing of OpenCL code. In: Haifa Verification Conference (HVC 2011) (1 2011)

[17] Cooper, K.D., Simpson, L.T., Vick, C.A.: Operator strength reduction. ACM Trans. Program. Lang. Syst. 23(5), 603–625 (Sep 2001), http://doi.acm.org/10.1145/504709.504710

[18] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. POPL '77, ACM, New York, NY, USA (1977), http://doi.acm.org/10.1145/512950.512973

[19] Cypher, R., Sanz, J.L.C.: The SIMD Model of Parallel Computation. Springer Publishing Company, Incorporated, 1st edn. (2011)

[20] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (Oct 1991), http://doi.acm.org/10.1145/115372.115320

[21] Eizenberg, A., Peng, Y., Pigli, T., Mansky, W., Devietti, J.: BARRACUDA: Binary-level analysis of runtime races in CUDA programs. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 126–140. PLDI 2017, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3062341.3062342

[22] Ermedahl, A., Gustafsson, J., Lisper, B.: Deriving WCET bounds by abstract execution. Proc. 11th International Workshop on Worst-Case Execution Time (WCET) Analysis (WCET 2011:) s. 72-82 (2011)

[23] Fauzia, N., Pouchet, L.N., Sadayappan, P.: Characterizing and enhancing global memory data coalescing on GPUs. In: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 12–22. CGO '15, IEEE Computer Society, Washington, DC, USA (2015), http://dl.acm.org/citation.cfm?id=2738600.2738603

[24] Galoppo, N., Govindaraju, N.K., Henson, M., Manocha, D.: LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing. pp. 3–. SC '05, IEEE Computer Society, Washington, DC, USA (2005), https://doi.org/10.1109/SC.2005.42

[25] Gerlek, M.P., Stoltz, E., Wolfe, M.: Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. ACM Trans. Program. Lang. Syst. 17(1), 85–122 (Jan 1995), http://doi.acm.org/10.1145/200994.201003

[26] Gong, X., Chen, Z., Ziabari, A.K., Ubal, R., Kaeli, D.: TwinKernels: An execution model to improve GPU hardware scheduling at compile time. In: Proceedings of the 2017 International Symposium on Code Generation and Optimization. pp. 39–49. CGO '17, IEEE Press, Piscataway, NJ, USA (2017), http://dl.acm.org/citation.cfm?id=3049832.3049838

[27] Grosser, T.: Polyhedral Compilation, https://polyhedral.info

[28] Gustafsson, J.: Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation. Ph.D. thesis, Department of Computer Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden, and Department of Computer Systems, Information Technology, Uppsala University, Box 325, S-751 05 Uppsala, Sweden (May 2000), http://www.es.mdh.se/publications/231-

[29] Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In: Proceedings of the 27th IEEE International Real-Time Systems Symposium. pp. 57–66. RTSS '06, IEEE Computer Society, Washington, DC, USA (2006), https://doi.org/10.1109/RTSS.2006.12

[30] He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: A MapReduce framework on graphics processors. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. pp. 260–269. PACT '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1454115.1454152

[31] Jang, B., Schaa, D., Mistry, P., Kaeli, D.: Exploiting memory access patterns to improve memory performance in data-parallel architectures. IEEE Trans. Parallel Distrib. Syst. 22(1), 105–118 (Jan 2011), http://dx.doi.org/10.1109/TPDS.2010.107

[32] Jia, W., Shaw, K.A., Martonosi, M.: MRPB: Memory request prioritization for massively parallel processors. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). pp. 272–283 (Feb 2014)

[33] Jia, W., Shaw, K.A., Martonosi, M.: Characterizing and improving the use of demand-fetched caches in GPUs. In: Proceedings of the 26th ACM International Conference on Supercomputing. pp. 15–24. ICS '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2304576.2304582

[34] Jog, A., Kayiran, O., Chidambaram Nachiappan, N., Mishra, A.K., Kandemir, M.T., Mutlu, O., Iyer, R., Das, C.R.: Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 395–406. ASPLOS '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2451116.2451158

[35] Jones, K.E.: Exploring the dark universe with supercomputers, https://www.symmetrymagazine.org/article/exploring-the-dark-universe-with-supercomputers

[36] Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 194–206. POPL '73, ACM, New York, NY, USA (1973), http://doi.acm.org/10.1145/512927.512945

[37] Kim, Y., Shrivastava, A.: CuMAPz: A tool to analyze memory access patterns in CUDA. In: Proceedings of the 48th Design Automation Conference. pp. 128–133. DAC '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2024724.2024754

[38] Kofler, K., Cosenza, B., Fahringer, T.: Automatic data layout optimizations for GPUs. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015: Parallel Processing. pp. 263–274. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)

[39] Koo, G., Oh, Y., Ro, W.W., Annavaram, M.: Access pattern-aware cache management for improving data utilization in gpu. In: Proceedings of the 44th Annual International Symposium on Computer Architecture. pp. 307–319. ISCA '17, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3079856.3080239

[40] Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004), http://dl.acm.org/citation.cfm?id=977395.977673

[41] Lee, S., Min, S.J., Eigenmann, R.: OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 101–110. PPoPP '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1504176.1504194

[42] Leung, A., Gupta, M., Agarwal, Y., Gupta, R., Jhala, R., Lerner, S.: Verifying GPU kernels by test amplification. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 383–394. PLDI '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2254064.2254110

[43] Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 187–196. FSE '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1882291.1882320

[44] Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: Concolic verification and test generation for GPUs. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 215–224. PPoPP '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2145816.2145844

[45] Liu, Y., Zhang, E.Z., Shen, X.: A cross-input adaptive framework for GPU program optimizations. In: 2009 IEEE International Symposium on Parallel Distributed Processing. pp. 1–10 (May 2009)

[46] Magni, A., Dubach, C., O'Boyle, M.: Automatic optimization of thread-coarsening for graphics processors. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation. pp. 455–466. PACT '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2628071.2628087

[47] McCool, M., Toit, S.D.: Metaprogramming GPUs with Sh. AK Peters Ltd (2004)

[48] Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 348–363. VMCAI'06, Springer-Verlag, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/11609773_23

[49] Monakov, A., Lokhmotov, A., Avetisyan, A.: Automatically tuning sparse matrix-vector multiplication for GPU architectures. In: Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers. pp. 111–125. HiPEAC'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-11515-8_10

[50] Narasiman, V., Shebanow, M., Lee, C.J., Miftakhutdinov, R., Mutlu, O., Patt, Y.N.: Improving GPU performance via large warps and two-level warp scheduling. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 308–317. MICRO-44, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2155620.2155656

[51] Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue 6(2), 40–53 (Mar 2008), http://doi.acm.org/10.1145/1365490.1365500

[52] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Publishing Company, Incorporated (2010)

[53] Nvidia: CUDA C Programming Guide v9.0, http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[54] Nvidia: Nvidia Performance Analysis Tools, http://developer.nvidia.com/performance-analysis-tools/

[55] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 519–530. PLDI '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2491956.2462176

[56] Rhu, M., Sullivan, M., Leng, J., Erez, M.: A locality-aware memory hierarchy for energy-efficient GPU architectures. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 86–98. MICRO-46, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2540708.2540717

[57] Rogers, T.G., O'Connor, M., Aamodt, T.M.: Cache-conscious wavefront scheduling. In: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 72–83. MICRO-45, IEEE Computer Society, Washington, DC, USA (2012), http://dx.doi.org/10.1109/MICRO.2012.16

[58] Ryoo, S., Rodrigues, C.I., Stone, S.S., Baghsorkhi, S.S., Ueng, S.Z., Stratton, J.A., Hwu, W.m.W.: Program optimization space pruning for a multithreaded GPU. In: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 195–204. CGO '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1356058.1356084

[59] Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program analysis using symbolic ranges. In: Proceedings of the 14th International Conference on Static Analysis. pp. 366–383. SAS'07, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=2391451.2391476

[60] Sarbó, J.: Abstract execution of programs. Periodica Polytechnica Electrical Engineering (Archives) 30(1), 37–47, https://pp.bme.hu/ee/article/view/4640

[61] Sethia, A., Jamshidi, D.A., Mahlke, S.: Mascar: Speeding up GPU warps by reducing memory pitstops. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). pp. 174–185 (Feb 2015)

[62] Sørensen, H.H.B.: Auto-tuning dense vector and matrix-vector operations for Fermi GPUs. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) Parallel Processing and Applied Mathematics. pp. 619–629. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[63] Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. IEEE Des. Test 12(3), 66–73 (May 2010), http://dx.doi.org/10.1109/MCSE.2010.69

[64] Sung, I.J., Stratton, J.A., Hwu, W.M.W.: Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. pp. 513–522. PACT '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1854273.1854336

[65] Tarditi, D., Puri, S., Oglesby, J.: Accelerator: Using data parallelism to program GPUs for general-purpose uses. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 325–335. ASPLOS XII, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1168857.1168898

[66] Ueng, S.Z., Lathara, M., Baghsorkhi, S.S., Hwu, W.M.W.: Languages and compilers for parallel computing. chap. CUDA-Lite: Reducing GPU Programming Complexity, pp. 1–15. Springer-Verlag, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-89740-8_1

[67] Venet, A.J.: The gauge domain: Scalable analysis of linear inequality invariants. In: Proceedings of the 24th International Conference on Computer Aided Verification. pp. 139–154. CAV'12, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-31424-7_15

[68] Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., Catthoor, F.: Polyhedral parallel code generation for CUDA. ACM Trans. Archit. Code Optim. 9(4), 54:1–54:23 (Jan 2013), http://doi.acm.org/10.1145/2400682.2400713

[69] Weber, N., Goesele, M.: MATOG: Array layout auto-tuning for CUDA. ACM Trans. Archit. Code Optim. 14(3), 28:1–28:26 (Aug 2017), http://doi.acm.org/10.1145/3106341

[70] Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog with binary decision diagrams for program analysis. In: Proceedings of the Third Asian Conference on Programming Languages and Systems. pp. 97–118. APLAS'05, Springer-Verlag, Berlin, Heidelberg (2005), http://dx.doi.org/10.1007/11575467_8

[71] Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC: First experiences with real-world applications. In: Proceedings of the 18th International Conference on Parallel Processing. pp. 859–870. Euro-Par'12, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-32820-6_85

[72] Wu, J., Belevich, A., Bendersky, E., Heffernan, M., Leary, C., Pienaar, J., Roune, B., Springer, R., Weng, X., Hundt, R.: GPUCC: An open-source GPGPU compiler. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization. pp. 105–116. CGO '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2854038.2854041

[73] Xie, X., Liang, Y., Sun, G., Chen, D.: An efficient compiler framework for cache bypassing on GPUs. In: Proceedings of the International Conference on Computer-Aided Design. pp. 516–523. ICCAD '13, IEEE Press, Piscataway, NJ, USA (2013), http://dl.acm.org/citation.cfm?id=2561828.2561929

[74] Yang, Y., Xiang, P., Kong, J., Mantor, M., Zhou, H.: A unified optimizing compiler framework for different GPGPU architectures. ACM Trans. Archit. Code Optim. 9(2), 9:1–9:33 (Jun 2012), http://doi.acm.org/10.1145/2207222.2207225

[75] Yang, Y., Xiang, P., Kong, J., Zhou, H.: A GPGPU compiler for memory optimization and parallelism management. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 86–97. PLDI '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1806596.1806606

[76] Zhang, E.Z., Jiang, Y., Guo, Z., Tian, K., Shen, X.: On-the-fly elimination of dynamic irregularities for GPU computing. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 369–380. ASPLOS XVI, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1950365.1950408

[77] Zhang, Y., Mueller, F.: Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization. pp. 155–164. CGO '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2259016.2259037