Static Analysis for Improving Performance of GPU Programs

Nimit Singhania
University of Pennsylvania

Abstract. GPUs are high-throughput energy-efficient processors, consisting of a large number of cores. Yet, it remains difficult to program them for performance. A particular challenge is utilizing GPU resources efficiently and accessing the GPU memory subsystem in a structured manner. Automatic compiler and hardware approaches have been explored previously to address these issues. The compiler techniques, however, grapple with the complexity of the program transformations required, while the hardware approaches suffer from a narrow view into the execution of programs.

This thesis presents a three-fold solution to the problems with existing approaches. Each of these solutions tries to utilize some form of regularity in GPU programs. We first present GPUDrano, a scalable and precise static analysis to identify memory performance issues in GPU programs. The analysis relies on tracking the dependence of program variables on thread ids. We next present a static analysis to identify cache-reuse opportunities across iterations of loops in the programs. Such opportunities are easily missed by the hardware due to its limited visibility into the execution. Finally, we present a structured framework to specify transformations on GPU programs, implement the transformations, and then verify their correctness. The framework relies on a small set of generic unit transformations, instances of which can be composed together to generate complex transformations.

1 Introduction

GPUs are emerging as an important computing platform. GPUs consist of large number of cores that operate in an SIMT (Single Instruction Multiple Thread) fashion, and are being used for various applications in image processing, scientific computation, and recently machine learning [10]. With frequency scaling no longer possible, CPU performance has reached a saturation point. Whereas, GPUs have a more scalable architecture and their performance is advancing at an exponential pace. Further, they are ever more relevant in today’s fast-paced communication driven world. A large amount of rich data is being produced and consumed everyday, and GPUs are well positioned to process this data with their high throughput and energy-efficiency.

Despite these benefits, it is an arduous task to program GPUs and extract high performance out of them. High level programming models like CUDA and
OpenCL make it easier to program them. Yet, they do not abstract away details of the hardware and a naively written CUDA/OpenCL code often performs poorly. There are multiple issues. First, utilization of GPU resources is left completely onto the programmers with various control parameters like number of blocks, number of threads per block, number of registers required per thread and amount of shared memory allocated per block. Careful tuning of these parameters is necessary to achieve maximal performance. Further, the parameters need to be re-tuned when the program is ported to a different GPU, since the resources vary across GPUs. Second, the programmer also needs to ensure that their programs access GPU’s memory subsystem in a structured manner in order to achieve maximal memory throughput. The memory subsystem in GPUs is very sensitive to the access pattern and irregular accesses can significantly degrade performance. The GPU programming models also expose an on-chip scratch-pad memory for the programmer to manually optimize memory performance. However, exploiting the on-chip memory remains a challenge for the programmer and resulting programs are often complex and difficult to maintain.

Sophisticated compiler and hardware technology has been developed to automatically optimize programs for CPUs. However, most of this technology is not directly applicable to GPU programs. Unlike CPU programs, a large number of threads are active simultaneously in a GPU program, and they are integral to improving performance. Hence, the analyses and transformations in a compiler need to account for all these threads. The analyses need to reason about program state for all threads simultaneously. The transformations, on the other hand, need to perform complex global changes to the program, like re-structuring loops, thread-geometry or memory-layout of data-structures and arrays, to achieve significant performance improvements. Moreover, the benefits of these transformations are often not immediately obvious and depend on program parameters like number of threads or size of data-structures that are determined at runtime, which further exacerbates the problem. While numerous works in the literature have explored automatic optimization for GPU programs, there is a dearth of principled approaches to do the same. A principled approach is useful, since it simplifies the implementation and correctness checking, which is especially required for the types of program transformations involved here. A formalism “polyhedral model” has been developed to help systematically transform loops in programs. However, it is often difficult to work with and applicable to a restricted class of programs.

An alternate approach to improving GPU performance is to adapt the execution of the program within the hardware, without changing the actual program. This is particularly useful if no simple program transformation can improve the performance of the program. For example, if some loops in the program access a matrix along its rows while the remaining loops along its columns, neither the row-major or column-major layout for the matrix might be better. Further, there is significant flexibility in GPUs on how parallel threads in a program are scheduled. Hence, adapting the execution of a GPU program is quite feasible. While this approach has been explored previously in the hardware, the adap-
tation has been mostly local and within the span of few instructions and few threads, which may not have significant benefits if, for example, the cache-reuse of data is spread across different iterations of a loop or far-apart threads. On the other hand, if we could adapt the execution at a macroscopic level, say for whole loops or all threads, we might be able to utilize cache better and achieve better performance.

In my thesis, I plan to focus on static analysis to detect and repair performance issues in GPU programs, and hence improve their performance. While resource utilization issues can not be detected statically, memory performance bugs exhibit patterns that are amenable to static analysis. We have developed a static analysis, GPUDrano, to identify such bugs (Section 4.1). Further, static analysis can help obtain an abstract view of the program, and this high-level view of the program can be fruitfully analyzed to generate global guidelines for adapting the runtime execution of GPU programs. We plan to build such a static analysis and extend the GPU hardware to accept guidelines from the analysis (Section 4.2). Lastly, there is a need for a structured program transformation framework, where-in the transformations for optimizing GPU programs can be specified and verified for correctness. Polyhedral model is a well-established formal framework for analyzing and transforming programs. We plan to extend this model to build our framework (Section 4.3).

The rest of the sections are organized as follows. Section 2 gives a broad overview of GPUs and the performance issues associated with GPUs. Section 3 explores prior-work on analyzing and optimizing GPU programs, and extending the hardware to improve GPU performance. Section 4 describes the overall thesis proposal. Section 5 describes the overall work plan. Finally, Section 6 concludes.

2 Background

We briefly describe here the execution and the hardware model for GPUs and the various performance issues related to GPUs.

GPUs follow a Single Instruction Multiple Thread (SIMT) execution model, where a large number of threads execute the same program, often called kernel, on different data. This is also known as the data-parallel execution model. The threads are organized in a two-level hierarchy: a bunch of threads together form a thread-block, while a group of blocks forms a grid. Blocks represent coarse-grained units of parallelism and execute independently of each other, while threads exhibit fine-grained parallelism and often collaborate with each other to complete desired tasks. With each thread and block, a thread id and block id is associated to identify the unit, where the id could be a one-to-three dimensional vector. The ids are used to distinguish execution of threads and blocks. The threads can access three types of memory spaces: local memory, private to each thread; shared memory, shared between threads within a block; and global memory, shared between all threads in the grid. Further, the threads within a block can synchronize on a __syncthreads() barrier. Lastly, the threads in a block execute in groups
of threads with consecutive ids, called *warps*, such that all threads within a warp execute in lock-step. A warp usually consists of 32 threads.

GPU hardware consists of multiple streaming multi-processors (SM). Each SM consists of a few streaming-processors (SP) which execute synchronously, a set of registers which are distributed between the threads scheduled on the SM, and some on-chip low-latency scratch-pad memory, which can split between a cache for global memory accesses and shared memory for blocks. A block is scheduled on a single SM. Multiple blocks can be scheduled on an SM, as long as there are enough registers and shared memory to accommodate the blocks. A GPU further consists of an off-chip DRAM which is used for the global memory variables in GPU programs. The DRAM allows a high-latency but high-bandwidth access from SMs. As mentioned in the previous paragraph, threads in a warp execute in lock-step on an SM, and when the threads issue a global memory load/store, the SM coalesces them together into as few DRAM transactions as possible, to utilize the high-bandwidth of DRAM.

There are a few well-known performance bottlenecks in GPUs. First, if the resources utilized by a thread-block are not well balanced with resources on SMs, fewer than maximum possible thread-blocks are allocated on each SM, and the SMs go under-utilized. Next, if a global memory access by a warp requires more than one transaction to the DRAM to complete, such an access is referred to as an *uncoalesced access*. This happens when accessed locations by the warp are far-apart in DRAM. Such accesses can significantly degrade performance. Next, 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, and hence, increased latency of accesses. Finally, when a warp encounters a conditional statement in the GPU program and the threads take different branches, the execution of the two branches is serialized, where first few threads of the warp execute one branch while the remaining threads remain idle, and then the remaining threads execute the other branch. This issue is known as *warp divergence* and if a large number of branches are encountered by a warp, there is significant serialization of threads which leads to a slowdown in performance. This issue is often difficult to fix, and we focus on the remaining performance issues in this proposal.

3 Related Work

In this section, we describe some prior work related to this thesis. We describe work on program transformation approaches to improve GPU performance, hardware approaches for GPU performance, and formalization and verification for GPUs.

3.1 Program transformation for GPU performance

We cover here some prominent approaches on program transformation to improve GPU performance. We first describe some program transformation tech-
niques that modify the memory layout of data-structures to have improved memory accesses from GPU code. Dymaxion [7] provides an API for the programmers to remap memory layout for data-structures from their original row-major layout on CPUs to a new layout on GPUs. The framework supports remapping data while it is transferred from CPU to GPU, and hides the latency of the remap operation by splitting data into small chunks and pipelining the transfer and remap operation (on GPU) for these chunks. It supports four kinds of remapping: row2col, which maps rows to columns of matrices; diagonal, that maps diagonal locations to rows of the matrices, useful in linear algebra and dynamic programming algorithms; indirect, which maps old locations to new ones based on an index array, useful in sparse matrix, sorting and hashing algorithms; and struct-array that transforms arrays of structures to structures of arrays. It also provides index transformation functions to compute new indices after the remap operation. It only supports one remap operation per data-structure. It is evaluated on four case-studies corresponding to each specific remap operation, and shows an overall 20% performance improvement. Dymaxion++ [6] augments Dymaxion with a directive-based API that allows programmers to describe layout changes as pragmas, instead of API function calls. It further supports operations for memory-layout transformations and data-placement on different memory spaces in GPUs.

[33] describes an automatic data-layout transformation approach for dynamic memory access and control flow irregularities. Since, the irregularities are dynamic and not known a-priori, the transformations are determined at runtime. It uses two primary mechanisms: data relocation, which moves data elements to new locations, and reference redirection, which redirects references from threads to new locations. Combinations of these mechanisms are applied to achieve desired data-layout and thread-to-data mapping, though optimal combinations are computationally hard to compute. To further ensure efficiency of optimizations, it first performs the transformations on CPU while a previous kernel is running on GPU, to hide the transformation overhead, and then, it uses a multi-level adaptation scheme to select specific transformations and tune their optimality. Upto 2.5× speedups are obtained on seven benchmarks from different domains with this approach.

[26] presents another automatic data-layout transformation approach, where the transformed data-layout for arrays is determined at compile time, unlike at runtime in [33]. It requires that the array definitions and accesses by the programmers are well-structured. For each array in the program, it defines a flattening function that maps array subscripts to actual memory indices, which determines the corresponding memory layout. It further assumes that subscripts for array accesses can be represented as affine functions of the thread and block ids and the iterators of loops surrounding the accesses. It optimizes memory layout of arrays to ensure that the accesses to the arrays are coalesced and concurrent accesses are distributed uniformly across parallel DRAM banks to avoid bank-conflicts. It does this by mapping the bits corresponding to thread and block ids in array access subscripts to steering bits of accessed memory addresses. Fur-
ther, to ensure consistency of layout during pointer assignments, it tags arrays with flattening functions and statically propagates tags via a data-flow analysis. It is evaluated on three benchmarks with speedups ranging from $1.07 \times$ to $6.60 \times$.

We next cover techniques where the performance of array accesses is improved by restructuring loops and thread-geometry and efficiently caching arrays or parts of arrays on the on-chip scratch-pad in GPUs. These techniques often require detailed analysis and transformation of loops and array accesses, which necessitates the loop nests and array accesses to be affine. Further, the techniques can be automatic or require some user-guidance as annotations. CUDA-lite [27] presents an approach to optimize uncoalesced accesses in CUDA program. It requires significant annotation from the user for arrays and functions of interest, dimensions of arrays, and loop iterators and their bounds. It uses a basic data-flow analysis to identify the uncoalesced accesses in the program. For each uncoalesced load/store within a loop, it transforms code to efficiently cache data on the scratch-pad or shared memory by introducing shared memory variables, tiling the loop, using coalesced loads/stores to transfer data to shared memory variables for each tile, and finally replacing uncoalesced loads/stores with shared memory loads and stores. It is evaluated on three benchmarks with speedups ranging from $2 \times$ to $17 \times$.

[32] combines multiple transformations to optimize naive CUDA functions into optimized versions. First, it applies an analysis to detect uncoalesced accesses and transforms code to cache corresponding data in shared memory, similar to CUDA-lite. Next, it vectorizes memory accesses and replaces float variables with float2 (two floats) variables. Then, it applies thread-block merging and thread merging to share data on scratch-pad and registers across blocks and threads, respectively, and hence reduce scratch-pad and register usage. The actual merging parameters are determined via empirical tuning. Next, it prefetches data to hide memory access latency with computation. Finally, similar to [26], it distributes concurrent accesses across parallel DRAM banks/partitions by shuffling thread-block ids. It is evaluated on 10 naive kernels with 3-27 lines of code and speed-ups upto $128 \times$ are obtained with an average of $15.1 \times$. Also, the optimized kernels perform on an average 33% better than CUDA CUBLAS library (version 2.2) implementations. Most of the speed-up in evaluated kernels is observed via coalescing and thread/thread-block merging.

[3] presents a technique to automatically transform simple sequential kernels to efficient parallel CUDA functions. It uses polyhedral model to analyze and transform the kernels. It relies on existing tool-chain to extract the polyhedral model, parallelize the model, and generate GPU code from the final transformed model. To transform the model, it first tries coalescing array accesses in the program by reordering loops and the thread-geometry. It sets up execution-order and spacial-locality constraints, and searches a valid reordering transformation satisfying the constraints. For the accesses not coalesced by this transformation, it marks them for efficient caching in shared memory, similar to CUDA-lite ([2] presents details on code-generation for efficient caching in shared memory).
Next, it algorithmically finds an optimal padding for arrays in shared memory, to minimize shared memory bank-conflicts. Finally, it tiles and unrolls loops to control the amount of parallelism and the use of shared memory and registers, just like thread/thread-block merging in [32], and uses empirical tuning to find optimum tile sizes and unroll factors. It is evaluated on four kernels, where the optimized kernels show performance slightly better than CUBLAS (version 1.0) implementations.

[11] presents a modeling of array accesses, similar to polyhedral model, and uses this model to apply data-transformations and vectorization for AMD GPUs and select appropriate memory spaces in NVIDIA GPUs. The work however does not provide a compiler implementation, and experiments are conducted in a semi-automated fashion. It is evaluated on five benchmarks for AMD GPUs and eight benchmarks on NVIDIA GPUs and shows speedups upto $11 \times$ on AMD GPUs and upto $13.5 \times$ on NVIDIA GPUs.

Table 1 summarizes the above approaches. For the correctness guarantees column, we assume data-layout transformations are correct in principle, since a data-layout transformation preserves correctness implicitly and only affects the performance of memory accesses. Further, [11] maps data to different memory spaces which preserves correctness. [3,2] rely on formal analysis using polyhedral model, and transform code only if it is allowed by the model. Note that the implementation of transformations by these systems is not verified, and subtle bugs might be introduced into the program while the transformation is carried out.

### 3.2 Hardware approaches to GPU performance

We next cover some approaches that rely on hardware extensions to improve GPU memory performance. There are two primary problems that these techniques try to address. First, to exploit the thread-level parallelism, the GPU warp scheduler tends to issue a memory instruction simultaneously for all warps
in a block, which leads to bursts of requests to the memory subsystem. This often leads to memory congestion and an increase in latency for memory requests. Second, GPUs have a provision for on-chip data cache, which can be used to cache data for threads. The amount of cache can be configured manually by the programmer. Due to a large number of threads in GPU programs, cache capacity per thread is quite small and often not enough to fit working memory sets of all threads. The traditional warp scheduler, however, follows a round-robin scheduling, due to which the working set of one warp is evicted by another before it can be reused, and this leads to cache thrashing.

We first describe approaches that tweak the warp scheduler to address these issues. CCWS [23] improves the scheduler to exploit intra-warp temporal locality. It dynamically adjusts the number of warps that are actively scheduled by the scheduler, so that working sets of these warps can fit the cache. It uses a lost locality detector to identify warps that are losing locality on memory instructions and upgrades their scheduling priority to prevent them from losing locality in future. Further, it actively schedules only the warps that have a priority above a certain threshold, and thus, controls the number of actively scheduled warps. The approach provides a harmonic mean 63\% improvement on four highly cache sensitive benchmarks and harmonic mean 24\% overall improvement against alternate scheduling approaches on 12 different benchmarks, including some from Rodinia [5]. It further makes a provision for the programmer to statically specify the limit on number of actively scheduled warps. It is observed that a carefully chosen limit leads to better performance than the above approach, for most benchmarks.

[19] uses two-level warp scheduling where warps with nearby IDs are grouped together into fetch-groups. The scheduler uses round-robin scheduling both for the fetch-groups and the warps within the fetch groups. It prioritizes warps within a fetch group until all warps in the group reach a long-latency operation, and then moves on to another fetch-group. This ensures that not all warps reach a long-latency operation simultaneously and thus, the memory subsystem is not saturated. This also allows warps to exploit spatial locality between warps within the fetch group. The approach provides a geometric mean speedup of 9.9\% over round-robin scheduling on Rodinia and NVIDIA CUDA SDK benchmarks.

OWL [14] uses multiple techniques to improve performance by an average 33\% (geometric mean 28\%) over round-robin scheduling, on benchmarks from Rodinia, NVIDIA CUDA SDK and Parboil [25]. It uses two-level scheduling similar to [19] but for thread-blocks instead of warps, where first a group of thread-blocks is scheduled on an SM via round-robin scheduling, and then blocks within the group are scheduled. Instead of a round-robin scheduling within the group, it gives priority to one block over the others, to exploit cache locality within the block and not lose all locality due to round-robin scheduling. It prioritizes non-consecutive blocks on different SMs to exploit bank-level parallelism in GPU DRAM. It also uses open-row prefetching from DRAM to further improve performance.
Mascar [24] presents two ideas to improve performance. First, it uses two warp-scheduling modes. In the usual case it uses round-robin scheduling. But when the memory subsystem gets saturated, it gives priority to all memory requests from a single warp, so that data for the warp is available sooner and the warp’s computation can start earlier. Next when the memory is saturated, it offloads stalled memory requests to a re-execution queue, so that new memory requests can be issued and the chances of reusing the cache increase. It is evaluated on benchmarks from Rodinia and Parboil, and gives a geometric mean 34% speed-up over baseline and 10% speed-up over OWL and CCWS.

We next describe some compile-time approaches supported by hardware extensions to improve GPU performance. [9] presents a compiler-centric approach where two versions of the same kernel are executed simultaneously by different warps on GPU. The versions are produced using different instruction schedulers in the compiler. The key insight of the work is that warp schedulers execute instructions in-order, and hence, the different order of instructions leads to different schedules. Specifically, if the memory instructions are located at different positions in these schedules, warps execute the same memory instruction at different times. This leads to spreading of requests to the memory subsystem, and this prevents memory congestion. The approach gives speed-up of average 12% on 11 benchmarks from AMD APP SDK v2.9.

[13] presents a compile-time analysis to predict if a memory access by a warp has within-warp spatial locality. If so, the access is cached in a 128B cache-line. Otherwise, the access bypasses cache and uses a 32B memory request, which saves time and energy. Note that the analysis is manually applied on to Rodinia benchmarks, and improves performance by about 16.8%. [31] builds on [13] to further exploit cross-instruction temporal locality within warps. It presents a compiler framework that uses light-weight profiling to capture this locality between instruction in a traffic reduction graph. It judiciously selects a set of instructions that exhibit good locality and potential for cache reuse. It bypasses cache for the remaining instructions and hence, prevents these instructions from polluting the cache. It shows on average 12.9% speed-up over baseline on various benchmarks from Rodinia and Parboil, as compared to an average 4.4% speed-up for a cache-all strategy.

MRPB [12] is a hardware approach that builds on previously described techniques. It proposes a memory request prioritization buffer (MRPB), to reorder memory requests and prioritize requests from a select group of threads to efficiently reuse cache, and to bypass cache for requests that potentially lead to stalling or cache-thrashing. It gives a speed-up of geometric mean $2.65\times$ on entire Polybench [21] and $1.27\times$ on entire Rodinia benchmark suites. It is also observed that the approach benefits regular benchmarks like SRAD in Rodinia, where a shared memory caching is possible but not applied for the ease of programming. [22] presents an augmented memory hierarchy in GPUs with course-grained and fine-grained accesses, where the access granularity of memory requests is dynam-
ically adjusted by a low-cost hardware predictor. It provides an average 12-14% improvement over traditional approaches on different benchmarks.

3.3 Formalization and verification for GPUs

We cover some existing literature on formalization and verification of GPU programs. These works describe the foundations on which we can build our tools and ensure their correctness.

GKLEE [17] presents a symbolic-execution based approach to identify correctness issues like data-races and barrier-divergence, and performance issues like uncoalesced accesses and shared memory bank conflicts in GPU programs. It initializes part of the state with symbolic values and the remaining state with concrete values, and initiates GPU program execution for a set of threads on this state. It uses a canonical schedule for threads to minimize the number of thread inter-leavings explored. It uses an SMT solver to explore different paths in the GPU program. It focuses on code coverage and uses heuristics to minimize the sets of paths explored. It uncovers data-race bugs in NVIDIA CUDA SDK.

[15] presents a hybrid static + dynamic analysis to verify functional correctness and identify data-races and barrier-divergence issues in GPU programs. It relies on the access-invariance of GPU programs, where the control flow and the memory accesses depend only on a few configuration variables and are independent of the given input data. During the analysis, it first logs memory accesses made during a program’s execution for a specific input, and then checks the logs to ensure the accesses are free of data-races and barrier-divergence. Then, it checks access-invariance of the program via taint-flow analysis. If the program is access-invariant, the memory accesses and control-flow are input-data independent and the properties extend to all inputs with the same configuration. Further, for such programs, only one input per configuration needs to be analyzed, which helps scale the analysis. It observes that access-invariance holds for large number of programs in NVIDIA CUDA SDK, and hence, the analysis is applicable to a large number of programs.

PUG [16] and GPUVerify [4] present a static analysis based on SMT solving. These techniques generate a verification condition for the desired correctness properties and use an SMT solver to check the condition. To simplify the verification condition, they use a canonical schedule for threads, similar to GKLEE. They also show that it is sufficient to use two symbolic threads to check data-races and barrier-divergence, which further simplifies the condition. These techniques, however, scale poorly as compared to [17] and [15], and rely on specification or automatic generation of loop-invariants which may be difficult for complex loops. It should be noted that all the above approaches employ different formalizations of GPU programs, to verify their correctness.

4 Thesis Proposal

In this section, we describe the overall thesis proposal. The proposal is split in three parts. In the first part, we describe GPUDrano, a static analysis to detect
performance bugs in GPU programs. In the second part, we describe a static analysis based warp-scheduling for GPUs. Finally in the third part, we describe a structured framework for program transformations in GPU programs.

4.1 Static analysis to detect performance bugs

Analysis to detect performance issues is integral to developing a framework that fixes them in GPU programs. Static analysis is particularly lucrative: bugs can be identified at compile-time, no costly run-time instrumentation is necessary, and the analysis can be exhaustive and identify corner-cases exhibited for rare inputs. Static analysis, however, often tends to be costly and imprecise, which prohibits its extensive use in practice. The performance bugs in GPUs, especially the bugs related to memory performance, usually have a regular pattern. This makes them amenable to fast and precise static analysis.

Static analysis to detect performance issues has been previously explored in the literature. Techniques like GKLEE [17] and PUG [16] use heavy-weight analysis which prevents them from scaling to real-world GPU programs. Other techniques like CUDA-lite [27], [32], and [3] have explored analysis to detect bugs in the context of fixing uncoalesced accesses. These techniques, however, either explore limited patterns or are applicable to a restricted set of programs. Further, they focus on automatic fixing of these bugs, and do not evaluate the effectiveness of their analysis for detecting bugs.

Recently, we have developed GPUDrano, a light-weight intra-procedural static analysis to detect uncoalesced accesses. The analysis is based on abstract interpretation [8,20], wherein we first define an abstraction of the program state. The abstraction captures specific features of the program execution essential to identify uncoalesced accesses, which helps the analysis scale to large programs. It tracks values, particularly load/store access indices, as a function of the thread id, tid. The analysis flags global array accesses as uncoalesced only when the access indices have potentially a large linear or non-linear dependency on tid. This is because, access indices with potentially large linear or non-linear dependency access far-apart locations in memory which requires multiple DRAM transactions. Further, if a segment of code is executed by a single thread (which is often the case, when some sequential work needs to be done in the program), a single transaction is required to complete an access, and thus, the access cannot be uncoalesced. Hence, our abstraction also tracks whether single or multiple threads are active during the execution of a statement.

To understand this further consider the example in Figure 1, a snippet from kernel Fan2 in Gaussian Elimination program in Rodinia benchmark suite [5]. The kernel performs row operations on matrix $A$ (size $N \times N$) and vector $B$ (size $N \times 1$) using the $t^{th}$ column of a multiplier matrix $M$ (size $N \times N$) and the $t^{th}$ row of $A$ and $B$. The kernel is a sequential procedure that takes in a thread id, tid, to distinguish executions of different threads. The kernel is executed for threads with ids in range $[0, N - t - 2]$. Each thread is assigned a distinct row and updates row $(tid + t + 1)$ of matrix $A$ and vector $B$. Note that $A$, $B$ and
// $t, N \mapsto 0$
if ($t + t + 1 >= N$) return;
x = $t + t + 1$; // $x \mapsto 1$
for ($y = t; y < N; y++$) { // $y \mapsto 0$
x = $N \times x + t$; // $xt \mapsto \top$
y = $N \times x + y$; // $xy \mapsto \top$
t = $N \times t + y$; // $ty \mapsto 0$
if ($y == t$)
$B[x] -= M[xt] \times B[t]$;
}
Fig. 1: Original Fan2 snippet

$M$ reside in global memory and are shared across threads, while the remaining variables are private to each thread.

Our abstraction for the example tracks local variables as a function of $tid$. All variables that are independent of $tid$ are assigned value 0 in the abstraction. Thus, variables $t$ and $N$ are assigned value 0 initially (shown in comments). Further, variables $y$ and $ty$ are constructed from $tid$-independent variables, and hence, assigned value 0. Next, all variables that are linear function of $tid$ with coefficient 1 (i.e., of the form $tid + c$), are assigned value 1. The variable $x$ is therefore assigned 1. Lastly, all variables that are either non-linear function of $tid$ or linear function with possibly greater than one coefficient are assigned $\top$. Variable $xt$, for example, is assigned the expression $N \times (tid + t + 1) + t$, where the coefficient for $tid$ is $N$. Since $N$ can be greater than one, $xt$ is assigned $\top$. Similarly, variable $xy$ is assigned $\top$. Now, global array accesses where the index variable has value $\top$, potentially have a large dependence on $tid$ and thus access far-apart memory locations in memory. Hence, accesses $A[xy]$ and $M[xt]$ are flagged as uncoalesced.

We now briefly discuss our abstraction. Let $\sigma$ be the program state and $\alpha()$ be the abstraction function. The abstraction mainly tracks the dependence of local scalar variables on $tid$ and assigns a single value to each local variable for all threads. It assigns conservative values to other variables in the program. Further, we use a different abstraction for integer and boolean variables. For the integer variables, we use the set $\mathbb{V}_{int} = \{\bot, 0, 1, -1, \top\}$ to abstract values. The value $\bot$ represents undefined values, while $\top$ represents all values. Let $l$ be a local variable, $t$ be a thread, $T$ be the set of all threads, and $c_0$ be a constant. Let $\sigma(l, t)$ define the value of $l$ in thread $t$, and $\alpha(\sigma)(l)$ the abstract value for $l$.

The remaining values in the abstraction are defined here:

$$
\alpha(\sigma)(l) = \begin{cases} 
0, & \text{ exists } c_0 \text{ s.t. for all } t \in T, \sigma(l, t) = c_0 \\
1, & \text{ exists } c_0 \text{ s.t. for all } t \in T, \sigma(l, t) = tid(t) + c_0 \\
-1, & \text{ exists } c_0 \text{ s.t. for all } t \in T, \sigma(l, t) = -tid(t) + c_0 
\end{cases}
$$

i.e., the abstract value 0 represents values constant across threads; 1 represents values that are a linear function of $tid$ with coefficient 1; and, -1 represents
values that are a linear function with coefficient -1. This abstraction is necessary to track dependency of access indices on tid.

We use the set $\mathcal{V}_{bool} = \{\bot, T, T^-, T\neg, F, F^-, T\neg F, T F\neg, F F\neg, \top\}$ to abstract boolean variables. Again $\bot$ and $\top$ represent the undefined value and all values, respectively. The remaining values are defined here.

$$\alpha(\sigma)(l) \equiv \begin{cases} T, & \text{for all } t \in T, \sigma(l, t) = true \\ T^-, & \text{exists } t \in T \text{ s.t. } \sigma(l, t) = false \\ F, & \text{for all } t \in T, \sigma(l, t) = false \\ F^-, & \text{exists } t \in T \text{ s.t. } \sigma(l, t) = true \\ \text{and for all } t' \in T \setminus t, \sigma(l, t') = true \end{cases}$$

i.e. the abstract value $T$ represents values true for all threads; $T^-$ represents values true for all but one thread; $F$ represents values false for all threads; $F^-$ represents values false for all but one thread. Further, we construct three additional boolean values: $T F = \{T, F\}$ representing values true or false for all threads, $T T^- = \{T, T^-\}$ representing values false for at most one thread, and $F F^- = \{F, F^-\}$ representing values true for at most one thread. We only use these compound values in our analysis, along with $\bot$ and $\top$. We use them to abstract branch predicates in kernels. This completes the abstraction for state.

We have further defined abstract semantics of GPU programs that preserves the above abstraction. The analysis uses these abstract semantics to execute the program abstractly and identify uncoalesced accesses. We have implemented the analysis in gpucc CUDA compiler [30], an open-source compiler based on LLVM. We have evaluated the analysis on Rodinia benchmarks (version 3.1) [5] and compared it against a dynamic analysis implementation. We observe that the static analysis identifies 111 out of 143 real bugs, whereas the dynamic analysis identifies 73 of these. The static analysis further reports 180 bugs in total, which means about two out of three reported bugs are real bugs. Also, for most programs it finishes in less than a second and scales to thousands-of-lines of GPU code.

In future, we plan to further refine the analysis, extend it to an interprocedural analysis, and cover other memory performance bugs like shared memory bank-conflicts.

### 4.2 Static analysis based warp-scheduling in GPUs

Program transformation approaches can significantly improve GPU program performance. However, they rely heavily on programs being well-structured. On the contrary, there is a lot of scope to improve performance of other programs, by adapting their execution on GPUs. We discussed different hardware/software based warp-scheduling approaches in Section 3.2. These approaches relied on ensuring that the working sets of threads lie within the cache, to prevent cache-thrashing, and the memory subsystem is not inundated with requests, to prevent memory saturation. They proposed different ideas to ensure this. However, almost all the works focused on hardware-based solutions.
A key benefit of a hardware-based solution is that complete information about the execution is available to the hardware, and the hardware can make a better informed decision. This allows the hardware to handle dynamic data-dependent performance issues. However, there are a couple of drawbacks. First, the hardware has a narrow view into the execution, due to which the decisions made by the hardware only have a local impact on the execution. Second, there is a cost associated with the analysis done in the hardware in terms of time, energy and on-chip resources. This cost hinders complex solutions and also adds an overhead to the performance.

As we observed in the case of GPUDrano, the regular nature of the GPU programs makes static analysis quite effective for such programs. Hence, static analysis can be used not just to detect performance issues, but also to guide the hardware in tackling these issues. Static analysis can overcome both the drawbacks of hardware-based solutions. It can use abstract-interpretation to filter out irrelevant information, and use the filtered information to get a wider view into the execution. Further, there is no runtime cost associated with static analysis. Hence, it can consume more resources to come up with better solutions.

Here we focus on statically throttling the number of actively scheduled warps by the scheduler. To understand this further, let’s revisit the example in Figure 1. We can see that each iteration of \( y \)-loop accesses the same value in the access \( M[xt] \). Further, consecutive iterations of \( y \)-loop access consecutive locations in the access \( A[xy] \). Since a cache line stores consecutive locations of an array, there is potential for cache reuse in consecutive iterations of \( y \)-loop, though there is very little reuse within each iteration. Further, since the reuse is distant in time, a hardware-based solution may not be able to detect it. However, a static analysis can detect such a reuse and instruct the hardware to throttle the number of actively scheduled warps during the execution of the loop, to ensure the cached data is not evicted before it is reused.

Such reuse of cached data on the accesses at the same instruction across consecutive iterations of a loop is quite common in GPU programs. We propose a static analysis to detect this. Let \( A[x] \) and \( A[x'] \) be two accesses at the same instruction in consecutive iterations of a loop. There is reuse between the two accesses if the difference \( (x - x') \) is small. Hence, the analysis tracks difference between values of expressions in consecutive iterations of the loop, and if the difference between indices of an access is small, there is potential for cache-reuse at the instruction. For example, in Figure 1, the analysis would initialize variables \( t, N, x \) and \( y \) to a base value 0 at the beginning of the loop. Now, variables \( xt, xy \) and \( ty \) are set to 0 in the first iteration of the loop. In the next iteration, \( y \) is incremented to 1, while the remaining variables remain the same. Hence, in the next iteration, \( xt \) stays 0 while \( xy \) and \( ty \) are updated to 1. Now, since the difference between all indices in first and second iteration is at most 1, there is potential to reuse cache for every access in the loop.

We can further use static analysis to compute working set for each thread in the loop (which is essentially the number of distinct global array accesses minus the accesses that are reused within a loop iteration). Next, for loops
with potential reuse, we can generate hardware directives, inserted before and after the loop, to request the hardware to start warp-throttling before the loop and stop after the loop. On the hardware-side, we propose to augment the warp-scheduler to start throttling when the first warp enters the hazardous region, and continue throttling until the last warp exits the region. We plan to implement the static analysis in gpucc, and the extension to the warp-scheduler in GPGPU-sim [1].

4.3 Structured framework for program transformations in GPU programs

Complex program transformations are often necessary to improve performance of GPU programs. As we discussed in the Section 3.1, broadly three classes of transformations are useful: data-layout transformation, where layout of arrays and data-structures is transformed to reduce latency of memory accesses from GPU code; loop and thread-geometry restructuring, where tiling or interchanging iterators/thread-dimensions is used to improve accesses and better utilize GPU resources; and explicitly caching data in shared memory, where data from DRAM is brought into low latency on-chip scratch-pad, thus reducing the latency of accesses. Each of these transformations involves complex changes to the GPU program.

While existing approaches explore benefits of these transformations to GPU programs, none of these approaches ensure that the transformed program is correct and functionally equivalent to the original program. Given the complexity of the transformations, it is necessary to verify them. However, this complexity itself becomes a hurdle in the process of verification. The transformations involve global changes to the code such that individual steps do not preserve correctness. Further, transformations like loop interchange, loop tiling and data chunking require loops and data-accesses to be structured, in order to be amenable to verification. Even if this is ensured, there are few formal tools to allow such analysis. These issues complicate verification of GPU transformations.

An algebraic framework namely “polyhedral model” has been developed over the past thirty years in the compiler community. It represents a program as a set of statements parameterized by the loop iterators and the parameters of the program. Each statement is associated with an access-relation for the accesses made within the statement and a schedule that determines the “time” at which the statement is executed. The model allows piece-wise affine transformations to the schedule and the access-relations of the statements, which leads to transformation of the program. It also consists of a dependence relation that stores execution-order dependencies between statements and prevents incorrect transformations from being applied to the model.

The model however has multiple drawbacks. First, the model is ill-suited for GPU programs. Parallelism is captured in the model by mapping multiple statements to the same time-stamp. This however is insufficient to distinguish block-level and thread-level parallelism. Also, there is no provision to represent
different memory spaces of GPUs, and it is not clear how to create new variables/statements in the model which is required for the above transformations. Next, the model allows transformations at the granularity of statements. The transformations, however, are often carried out at the level of loops, and the model does not provide a direct way to identify loops or groups of statements. Next, the space of transformations allowed by the model is quite huge, while the transformations applied in practice are a composition of few unit transformations like loop-tiling, loop-interchange etc. Lastly, transformations in the model are often carried out by creating an Integer Linear Program that encodes the correctness constraints and optimizes a performance related cost function. The space of transformations is however non-convex and the cost model varies across GPUs, and therefore, this approach often leads to a sub-optimal performance optimization.

To overcome these drawbacks, we propose an extension to the polyhedral model. In our extended model, we first propose to have “named loop components” where a name tag is associated with a group of statements representing loop nests/parts of loop nests in the GPU program. The name tags could be generated automatically by the compiler or specified by the programmer. Further, we propose to represent threads and blocks in the program via special loops in the model, associated with tags “block-parallel” and “thread-parallel”. Each iteration of a “block-parallel” loop would represent an independent block with id corresponding to the iterator value, while each iteration of “thread-parallel” loop would represent an independent thread. Note that block-parallel and thread-parallel loops must correspond to the outermost loops in the model.

Next, we propose a few block transformations that define the common unit transformations for GPU code, along with the verification procedures to check the correctness of the transformations. The block transformations are:

- $L = \text{split-merge}(L_1, L_2, k)$: This transformation splits or merges adjacent loop nests with tags $L_1$ and $L_2$ so that the two loop nests share $k$ common loops after the transformation. It further generates a tag $L$ for new component (useful if merging was applied and the merged component is used for further transformations). The verification procedure needs to check that $L_1$ and $L_2$ are adjacent before the transformation. Note that the transformation may require extending one of the loop nests with dummy loops, so that the merging can be applied.

- $\text{affine}(L, f)$: This transformation applies an affine transformation $f$ on the loop iterators of the loop nest $L$. The transformations here include loop interchange, loop skewing, loop tiling etc. The verification check involves ensuring that $f$ is a valid transformation on the iterators.

- $\text{parallelize}(L, k, b/t)$: This transformation tags the $k^{th}$ loop in the loop nest $L$ as “block-parallel”($b$) or “thread-parallel”($t$). Before applying the tags, it needs to be ensured that the iterations of the $k^{th}$ loop are independent. Also, the tag “block-parallel” must be applied to the outermost loop of $L$, while “thread-parallel” must be applied to the loop next to a “block-parallel” loop.
- **data-affine**\((A, f)\): This transformation applies an affine transformation \(f\) on the layout of array \(A\). Note that, to encompass array-of-structures to structure-of-arrays transformation, the fields of a structure must be exposed as iterations of a dimension of \(A\). Now, the transformation corresponds to reordering the field-dimension with other dimensions of the array.

  The verification check needs to ensure that \(f\) is a valid transformation on \(A\). Further, it must check that the array is local to the function and not read or written from external functions. This transformation is already implemented partly within the existing model.

- **data-shared**\((L, k, A, lb, ub)\): This transformation explicitly caches data in shared memory in the \(k^{th}\) loop of \(L\) for array \(A\). The bounds \(lb\) and \(ub\) define lower and upper bounds for the part of the array that is cached in shared memory. The transformation involves caching data in shared memory at the beginning of the loop and writing it back at the end of the loop. The verification check needs to ensure that, for each access to \(A\) inside the loop, the access is redirected to the cached data, and that the accessed data lies within the bounds \(lb\) and \(ub\).

A sequence of above transformations can be applied on the original model to get the final transformed model. We further need to ensure that the overall transformation is correct. In particular, we need to ensure that the control and data dependencies of the program are preserved during the overall transformation. This can be done by initially computing the control and data-dependencies, transforming them as the unit transformations are applied to the model and finally after all transformations are applied, checking that none of the dependencies are violated. We also need to check that no dependencies are carried across the block-parallel and thread-parallel loops, so that the loops can be run in parallel.

Let’s consider this with the example kernel Fan2 in Figure 1. As discussed in Section 4.1, the kernel consists of two uncoalesced accesses, \(A[xy]\) and \(M[xt]\). Each thread in the kernel is assigned a row of the matrices \(A\) and \(M\), which are both laid-out in a row-major order. Hence, during these accesses, consecutive threads access elements along the column which are far apart in memory, and this makes the accesses uncoalesced. These accesses can be coalesced in two ways. First, we can restructure the program so that the threads are assigned columns of the matrices, instead of rows. Second, we can change the layout of \(A\) and \(M\) to be column-major, rather than row-major.

Figure 2(a) shows the view of the program in our extended model. Note that we have added a loop for threads in the kernel and marked the loop as thread-parallel. Further, we have labeled two groups of statements enclosed within curly braces, \(L_1\) representing initialization of \(x\), and \(L_2\) representing the statements inside the \(y\)-loop. Figure 2(b) shows the restructuring transformation, where first a split-merge operation is applied on \(L_1\) and \(L_2\). During the operation, first a dummy loop is introduced around \(L_1\) with a single iteration in it. Now, both \(L_1\) and \(L_2\) are surrounded by two loops. Next, the two loop nests are merged into a single loop nest to complete the operation. Next, the affine
/!

```c
// thread-parallel
for (tid = 0; tid < T; tid++)
{
    L1: if (tid + t + 1 >= N) return;
        x = tid + t + 1;
    }
    for (y = t; y < N; y++)
    L2: xt = N * x + t;
        xy = N * x + y;
        ty = N * t + y;
            if (y == t)
                B[x] -= M[xt] * B[t];
    }
```

(a) View of the kernel in our model
(b) Loop restructuring
(c) Data layout transformation

Fig. 2: Program transformations for Fan2 kernel in Figure 1.

We have partly addressed the above extension to polyhedral model in Loopy [18] for sequential programs. Loopy allows tagging loops in C programs, and specifying a sequence of transformations consisting of split-merge and affine operations on loops. Loopy takes a C program with tagged loops and a script consisting of operations to be applied on the loops. It applies the operations in the script on the model extracted from the program, checks the correctness of the transformed model, and if correct, generates the final program. We observe up to 22× speed-ups on Polybench [21], a suite of C kernels used widely to evaluate polyhedral model based tools.

We further need procedures to extract the model from a GPU program and to generate GPU code from the transformed model. PET [28] is an existing tool to extract polyhedral model from C programs, while PPCG [29] is a tool to generate CUDA code from polyhedral model. We plan to extend these to get the required extraction and code generation.

5 Work Plan

The thesis consists of three parts: a static analysis to detect performance bugs, static analysis based warp-scheduling in GPUs, and a structured framework for program transformations in GPUs.
The first part on detecting performance bugs is completed. We have submitted a paper to CAV 2017 on this work. We, however, plan some enhancements to the analysis to make it viable in practice. This includes refining the analysis, extending it to an inter-procedural analysis, and covering other memory performance bugs.

The second part on static analysis based warp-scheduling is yet to be done. The static analysis should however be easy to implement, given its simplicity and my experience with the infrastructure. The extension to the simulator would be the most challenging component here. Overall, this part involves the following tasks:

- (1 week) Manual inspection of a Rodinia benchmark suite to estimate potential benefits of the analysis.
- (2 weeks) Implementation of the analysis.
- (1 week) Testing the implementation on few benchmarks from the suite.
- (3 weeks) Implementing the extension for the warp-scheduler in the simulator.
- (1 week) Evaluation of the approach on Rodinia.
- (3 weeks) Writing a paper on the work.

The third part on structured framework for program transformations is also yet to be done. However, I have explored this direction previously with Loopy and my experience with Loopy would be useful here. Extracting the model from CUDA code, and generating CUDA code back from the model would be the challenging component here. This part involves the following tasks:

- (3 weeks) Formalizing the model.
- (4 weeks) Implementing the model in Polly, a polyhedral-model framework in LLVM.
- (5 weeks) Extraction of the model from CUDA programs, and CUDA code generation from the model.
- (3 weeks) Evaluation of the model on benchmark suites.
- (5 weeks) Writing a paper on the work.

The overall anticipated timeline for my thesis is as follows:

- (Jun - Aug 2017) Static analysis internship at Google.
- (Sep 2017) Write up the second part (potential target - ISCA).
- (Mar 2018) Write up the third part (potential target - OOPSLA).

6 Conclusion

GPUs are high-throughput energy-efficient processors. While traditionally used for gaming and graphics, they are making their way into serious applications
like scientific computing and machine learning. Programming them for performance is still a huge hurdle though, which makes them inaccessible to a regular programmer. This is a significant challenge in their adoption into mainstream computing. The work will hopefully bridge this gap and help regular programmers write efficient GPU programs.

References