A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications

Jaewoong Sim Aniruddha Dasgupta† * Hyesoon Kim Richard Vuduc
Georgia Institute of Technology Power and Performance Optimization Labs, AMD†
{jaewoong.sim, hyesoon.kim, riche}@gatech.edu aniruddha.dasgupta@amd.com

Abstract

Tuning code for GPGPU and other emerging many-core platforms is a challenge because few models or tools can precisely pinpoint the root cause of performance bottlenecks. In this paper, we present a performance analysis framework that can help shed light on such bottlenecks for GPGPU applications. Although a handful of GPGPU profiling tools exist, most of the traditional tools, unfortunately, simply provide programmers with a variety of measurements and metrics obtained by running applications, and it is often difficult to map these metrics to understand the root causes of slowdowns, much less decide what next optimization step to take to alleviate the bottleneck. In our approach, we first develop an analytical performance model that can precisely predict performance and aims to provide programmer-interpretable metrics. Then, we apply static and dynamic profiling to instantiate our performance model for a particular input code and show how the model can predict the potential performance benefits. We demonstrate our framework on a suite of micro-benchmarks as well as a variety of computations extracted from real codes.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architectures; C.4 [*Performance of Systems*]: Modeling techniques; C.5.3 [*Computer System Implementation*]: Microcomputers

General Terms Measurement, Performance

Keywords CUDA, GPGPU architecture, Analytical model, Performance benefit prediction, Performance prediction

1. Introduction

We consider the general problem of how to guide programmers or high-level performance analysis and transformation tools with performance information that is precise enough to identify, understand, and ultimately fix performance bottlenecks. This paper proposes a performance analysis framework that consists of an analytical model, suitable for intra-node analysis of platforms based on graphics co-processors (GPUs), and static and dynamic profiling tools. We argue that our framework is suitable for performance diagnostics through examples and case studies on real codes and systems.

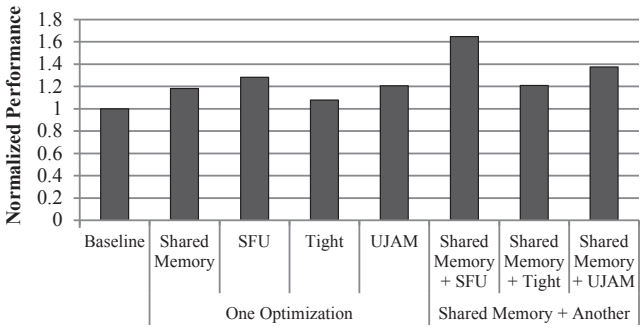

Figure 1. Performance speedup when some combinations of four independent optimization techniques are applied to a baseline kernel.

Consider an experiment in which we have a computational kernel implemented on a GPU and a set of n independent candidate optimizations that we can apply. Figure 1 shows the normalized performance when some combinations of four independent optimization techniques are applied to such a kernel, as detailed in Section 6.2.1. The leftmost bar is the parallelized baseline. The next four bars show the performance of the kernel with exactly one of four optimizations applied. The remaining bars show the speedup when one more optimization is applied on top of $Shared\ Memory$, one of the optimizations.

The figure shows that each of four optimizations improves performance over the baseline, thereby making them worth applying to the baseline kernel. However, most of programmers cannot estimate the degree of benefit of each optimization. Thus, a programmer is generally left to use intuition and heuristics. Here, $Shared\ Memory$ optimization is a reasonable heuristic starting point, as it addresses the memory hierarchy.

Now imagine a programmer who, having applied $Shared\ Memory$, wants to try one more optimization. If each optimization is designed to address a particular bottleneck or resource constraint, then the key to selecting the next optimization is to understand to what extent each bottleneck or resource constraint affects the current code. In our view, few, if any, current metrics and tools for GPUs provide this kind of guidance. For instance, the widely used occupancy metric on GPUs indicates only the degree of thread-level parallelism (TLP), but not the degree of memory-level or instruction-level parallelism (MLP or ILP).

In our example, the kernel would not be improved much if the programmer tries the occupancy-enhancing $Tight$ since $Shared\ Memory$ has already removed the same bottleneck that

* The work was done while the author was at Georgia Institute of Technology.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.

$Tight$ would have. On the other hand, if the programmer decides to apply SFU, which makes use of special function units (SFUs) in the GPU, the kernel would be significantly improved since $Shared\ Memory$ cannot obtain the benefit that can be achieved by SFU.

Our proposed framework, *GPUPerf*, tries to provide such understanding. GPUPerf *quantitatively* estimates potential performance benefits along four dimensions: inter-thread instruction-level parallelism (B_{itilp}), memory-level parallelism (B_{memlp}), computing efficiency (B_{fp}), and serialization effects (B_{serial}). These four metrics suggest what types of optimizations programmers (or even compilers) should try first.

GPUPerf has three components: *a frontend data collector, an analytical model, and a performance advisor*. Figure 2 summarizes the framework. GPUPerf takes a CUDA kernel as an input and passes the input to the frontend data collector. The frontend data collector performs static and dynamic profiling to obtain a variety of information that is fed into our GPGPU analytical model. The analytical model greatly extends an existing model, the MWP-CWP model [9], with support for a new GPGPU architecture ("Fermi") and addresses other limitations. The performance advisor digests the model information and provides *interpretable* metrics to understand potential performance bottlenecks. That is, by inspecting particular terms or factors in the model, a programmer or an automated tool could, at least in principle, use the information directly to diagnose a bottleneck and perhaps prescribe a solution.

Figure 2. An overview of GPUPerf.

For clarity, this paper presents the framework in a "reverse order." It first describes the key potential benefit metrics output by the performance advisor to motivate the model, then explains the GPGPU analytical model, and lastly explains the detailed mechanisms of the frontend data collector. After that, to evaluate the framework, we apply it to six different computational kernels and 44 different optimizations for a particular computation. Furthermore, we carry out these evaluations on actual GPGPU hardware, based on the newest NVIDIA C2050 ("Fermi") system. The results show that our framework successfully differentiates the effects of various optimizations while providing interpretable metrics for potential bottlenecks.

In summary, our key contributions are as follows:

1. We present a comprehensive performance analysis framework, GPUPerf, that can be used to predict performance and understand bottlenecks for GPGPU applications.
2. We propose a simple yet powerful analytical model that is an enhanced version of the MWP-CWP model [9]. Specifically, we focus on improving the *differentiability* across distinct optimization techniques. In addition, by following the work-depth-graph formalism, our model provides a way to interpret model components and relates them directly to performance bottlenecks.
3. We propose several new metrics to *predict* potential performance benefits.

2. MWP-CWP Model

The analytical model developed for GPUPerf is based on the one that uses MWP and CWP [9]. We refer to this model as the MWP-CWP model in this paper. The MWP-CWP model takes the following inputs: the number of instructions, the number of memory requests, and memory access patterns, along with GPGPU architecture parameters such as DRAM latency and bandwidth. The total execution time for a given kernel is predicted based on the inputs. Although the model predicts execution cost fairly well, the understanding of performance bottlenecks from the model is not so straightforward. This is one of the major motivations of GPUPerf.

2.1 Background on the MWP-CWP Model

Figure 3 shows the MWP-CWP model. The detailed descriptions of the model can be found in [9]. Here, we breifly describe the key concepts of the model. The equations of MWP and CWP calculations are also presented in Appendix A.[1]

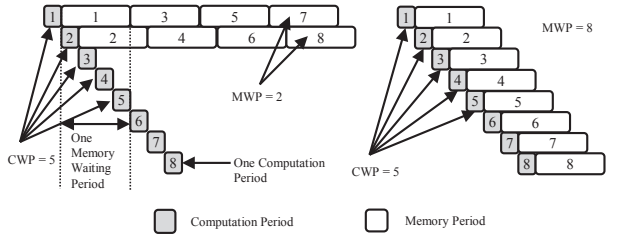

Figure 3. MWP-CWP model. (left: MWP $<$ CWP, right: MWP $>$ CWP)

Memory Warp Parallelism (MWP): MWP represents the maximum number of warps[2] per streaming multiprocessor (SM) [3] that can access the memory simultaneously. MWP is an indicator of memory-level parallelism that can be exploited but augmented to reflect GPGPU SIMD execution characteristics. MWP is a function of the memory bandwidth, certain parameters of memory operations such as latency, and the number of active warps in an SM. Roughly, the cost of memory operations is modeled to be the number of memory requests over MWP. Hence, modeling MWP correctly is very critical.

Computation Warp Parallelism (CWP): CWP represents the number of warps that can finish their one computation period during one memory waiting period plus *one*. For example, in Figure 3, CWP is 5 in both cases. One computation period is simply the average computation cycles per one memory instruction. CWP is mainly used to classify three cases, which are explained below.

Three Cases: The key component of the MWP-CWP model is identifying how much (and/or which) cost can be hidden by multi-threading in GPGPUs. Depending on the relationship between MWP and CWP, the MWP-CWP model classifies the three cases described below.

1. *MWP $<$ CWP:* The cost of computation is hidden by memory operations, as shown in the left figure. The total execution cost is determined by memory operations.
2. *MWP $>=$ CWP:* The cost of memory operations is hidden by computation, as shown in the right figure. The total execution cost is the sum of computation cost and one memory period.
3. *Not enough warps:* Due to the lack of parallelism, both computation and memory operation costs are only partially hidden.

2.2 Improvements over the MWP-CWP Model

Although the baseline model provides good prediction results for most GPGPU applications, some limitations make the model oblivious to certain optimization techniques. For instance, it assumes

[1] MWP and CWP calculations are updated for the analytical model in GPUPerf.

[2] Warp, a group of 32 threads, is a unit of execution in a GPGPU.

[3] SM and core are used interchangeably in this paper.

that a memory instruction is always followed by consecutive dependent instructions; hence, MLP is always one. Also, it ideally assumes that there is enough instruction-level parallelism. Thus, it is difficult to predict the effect of prefetching or other optimizations that increase instruction/memory-level parallelism.

Cache Effect: Recent GPGPU architectures such as NVIDIA's Fermi GPGPUs have a hardware-managed cache memory hierarchy. Since the baseline model does not model cache effects, the total memory cycles are determined by multiplying memory requests and the average global memory latency. We simply model the cache effect by calculating average memory access latency (AMAT); the total memory cycles are calculated by multiplying memory requests and AMAT.

SFU Instruction: In GPGPUs, expensive math operations such as transcendentals and square roots can be handled with dedicated execution units called *special function units* (SFUs). Since the execution of SFU instructions can be overlapped with other floating point (FP) instructions, with a good ratio between SFU and FP instructions, the cost of SFU instructions can almost be hidden. Otherwise, SFU contention can hurt performance. We model these characteristics of special function units.

Parallelism: The baseline model assumes that ILP and TLP are enough to hide instruction latency, thereby using the *peak* instruction throughput when calculating computation cycles. However, when ILP and TLP are not high enough to hide the pipeline latency, the effective instruction throughput is less than the peak value. In addition, we incorporate the MLP effect into the new model. MLP can reduce total memory cycles.

Binary-level Analysis: The MWP-CWP model only uses information at the PTX level.[4] Since there are code optimization phases *after* the PTX code generation, using only the PTX-level information prevents precise modeling. We develop static analysis tools to extract the binary-level information and also utilize hardware performance counters to address this issue.

3. Performance Advisor

The goal of the performance advisor is to convey performance bottleneck information and estimate the potential gains from reducing or eliminating these bottlenecks. It does so through four potential benefit metrics, whose impact can be visualized using a chart as illustrated by Figure 4. The x-axis shows the cost of memory operations and the y-axis shows the cost of computation. An application code is a point on this chart (here, point A). The sum of the x-axis and y-axis values is the execution cost, but because computation and memory costs can be overlapped, the final execution cost of T_{exec} (e.g., wallclock time) is a different point, A', shifted relative to A. The shift amount is denoted as T_{overlap}. A diagonal line through $y = x$ divides the chart into *compute bound* and *memory bound* zones, indicating whether an application is limited by computation or memory operations, respectively. From point A', the benefit chart shows how each of the four different potential benefit metrics moves the application execution time in this space.

A given algorithm may be further characterized by two additional values. The first is the ideal computation cost, which is generally the minimum time to execute all of the essential computational work (e.g., floating point operations), denoted T_{fp} in Figure 4. The second is the minimum time to move all data from the DRAM to the cores, denoted by $T_{\text{mem\_min}}$. When memory requests are prefetched or all memory service is hidden by other computation, we might hope to hide or perhaps eliminate all of the memory operation costs. Ideally, an algorithm designer or programmer could provide estimates or bounds on T_{fp} and $T_{\text{mem\_min}}$. However,

[4] PTX is an intermediate representation used before register allocation and instruction scheduling.

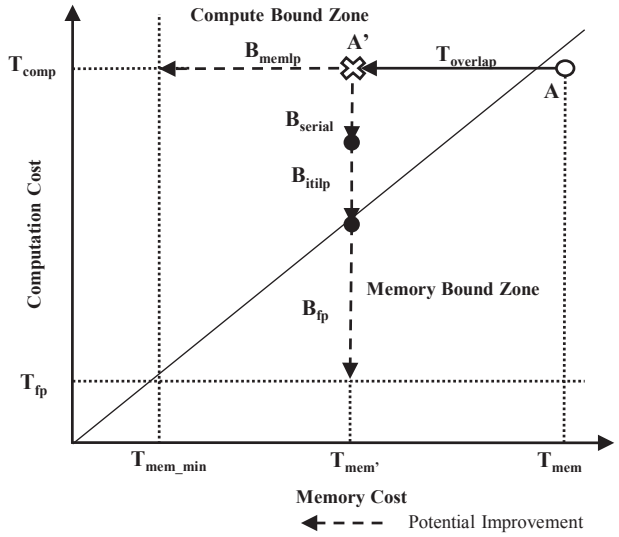

Figure 4. Potential performance benefits, illustrated.

when the information is not available, we could try to estimate T_{fp} from, say, the number of executed FP instructions in the kernel.[5]

Suppose that we have a kernel, at point A', having different kinds of inefficiencies. Our computed benefit factors aim to quantify the degree of improvement possible through the elimination of these inefficiencies. We use four potential benefit metrics, summarized as follows.

- B_{itilp} indicates the potential benefit by increasing inter-thread instruction-level parallelism.
- B_{memlp} indicates the potential benefit by increasing memory-level parallelism.
- B_{fp} represents the potential benefit when we ideally remove the cost of inefficient computation. Unlike other benefits, we cannot achieve the 100% of B_{fp} because a kernel must have some operations such as data movements.
- B_{serial} shows the amount of savings when we get rid of the overhead due to serialization effects such as synchronization and resource contention.

B_{itilp}, B_{fp}, and B_{serial} are related to the computation cost, while B_{memlp} is associated with the memory cost. These metrics are summarized in Table 1.

Name	Description	Unit
T_{exec}	Final predicted execution time	cost
T_{comp}	Computation cost	cost
T_{mem}	Memory cost	cost
T_{overlap}	Overlapped cost due to multi-threading	cost
T'_{mem}	$T_{\text{mem}} - T_{\text{overlap}}$	cost
T_{fp}	Ideal T_{comp}	ideal cost
$T_{\text{mem\_min}}$	Ideal T_{mem}	ideal cost
B_{serial}	Benefits of removing serialization effects	benefit
B_{itilp}	Benefits of increasing inter-thread ILP	benefit
B_{memlp}	Benefits of increasing MLP	benefit
B_{fp}	Benefits of improving computing efficiency	benefit

Table 1. Summary of performance guidance metrics.

[5] In our evaluation we also use the number of FP operations to calculate T_{fp}.

4. GPGPU Analytical Model

In this section we present an analytical model to generate the performance metrics described in Section 3. The input parameters to the analytical model are summarized in Table 2.

4.1 Performance Prediction

First, we define T_{exec} as the overall execution time, which is a function of T_{comp}, T_{mem}, and T_{overlap}, as shown in Equation (1).

$$T_{\text{exec}} = T_{\text{comp}} + T_{\text{mem}} - T_{\text{overlap}} \tag{1}$$

As illustrated in Figure 4, the execution time is calculated by adding computation and memory costs while subtracting the overlapped cost due to the multi-threading feature in GPGPUs. Each of the three inputs of Equation (1) is described in the following.

4.1.1 Calculating the Computation Cost, T_{comp}

T_{comp} is the amount of time to execute compute instructions (excluding memory operation waiting time, but including the cost of executing memory instructions) and is evaluated using Equations (2) through (10).

We consider the computation cost as two components, a parallelizable base execution time plus overhead costs due to serialization effects:

$$T_{\text{comp}} = \underbrace{W_{\text{parallel}}}_{\text{Base}} + \underbrace{W_{\text{serial}}}_{\text{Overhead}}. \tag{2}$$

The base time, W_{parallel}, accounts for the number of operations and degree of parallelism, and is computed from basic instruction and hardware values as shown in Equation (3):

$$W_{\text{parallel}} = \underbrace{\frac{\#\text{insts} \times \#\text{total\_warps}}{\#\text{active\_SMs}}}_{\text{Total instructions per SM}} \times \underbrace{\frac{\text{avg\_inst\_lat}}{\text{ITILP}}}_{\text{Effective throughput}}. \tag{3}$$

The first factor in Equation (3) is the total number of instructions that an SM executes, and the second factor indicates the *effective* instruction throughput. Regarding the latter, the average instruction latency, avg_inst_lat, can be approximated by the latency of FP operations in GPGPUs. When necessary, it can also be precisely calculated by taking into account the instruction mix and the latency of individual instructions on the underlying hardware. The value, ITILP, models the possibility of inter-thread instruction-level parallelism in GPGPUs. In particular, instructions may issue from multiple warps on a GPGPU; thus, we consider global ILP (i.e., ILP among warps) rather than warp-local ILP (i.e., ILP of one warp). That is, ITILP represents how much ILP is available among all executing to hide the pipeline latency.

ITILP can be obtained as follows:

$$\text{ITILP} = \min\left(\text{ILP} \times N, \text{ITILP}_{\max}\right) \tag{4}$$

$$\text{ITILP}_{\max} = \frac{\text{avg\_inst\_lat}}{\text{warp\_size}/\text{SIMD\_width}}, \tag{5}$$

where N is the number of active warps on one SM, and SIMD_width and warp_size represent the number of vector units and the number of threads per warp, respectively. On the Fermi architecture, SIMD_width $=$ warp_size $=$ 32. ITILP cannot be greater than $\text{ITILP}_{\max}$, which is the ITILP required to fully hide pipeline latency.

We model serialization overheads, W_{serial} from Equation (2), as

$$W_{\text{serial}} = O_{\text{sync}} + O_{\text{SFU}} + O_{\text{CFdiv}} + O_{\text{bank}}, \tag{6}$$

where each of the four terms represents a source of overhead—synchronization, SFU resource contention, control-flow divergence, and shared memory bank conflicts. We describe each overhead below.

Synchronization Overhead, O_{sync}: When there is a synchronization point, the instructions after the synchronization point cannot be executed until all the threads reach the point. If all threads are making the same progress, there would be little overhead for the waiting time. Unfortunately, each thread (warp) makes its own progress based on the availability of source operands; a range of progress exists and sometimes it could be wide. The causes of this range are mainly different DRAM access latencies (delaying in queues, DRAM row buffer hit/miss etc.) and control-flow divergences. As a result, when a high number of memory instructions and synchronization instructions exist, the overhead increases as shown in Equations (7) and (8):

$$O_{\text{sync}} = \frac{\#\text{sync\_insts} \times \#\text{total\_warps}}{\#\text{active\_SMs}} \times F_{\text{sync}} \tag{7}$$

$$F_{\text{sync}} = \Gamma \times \text{avg\_DRAM\_lat} \times \underbrace{\frac{\#\text{mem\_insts}}{\#\text{insts}}}_{\text{Mem. ratio}}, \tag{8}$$

where Γ is a machine-dependent parameter. We chose 64 for the modeled architecture based on microarchitecture simulations.

SFU Resource Contention Overhead, O_{SFU}: This cost is mainly caused by the characteristics of special function units (SFUs) and is computed using Equations (9) and (10) below. As described in Section 2.2, the visible execution cost of SFU instructions depends on the ratio of SFU instructions to others and the number of execution units for each instruction type. In Equation (9), the *visibility* is modeled by F_{SFU}, which is in $[0, 1]$. A value of $F_{\text{SFU}} = 0$ means none of the SFU execution costs is added to the total execution time. This occurs when the SFU instruction ratio is less than the ratio of special function to SIMD units as shown in Equation (10).

$$O_{\text{SFU}} = \frac{\#\text{SFU\_insts} \times \#\text{total\_warps}}{\#\text{active\_SMs}} \times \underbrace{\frac{\text{warp\_size}}{\text{SFU\_width}}}_{\text{SFU throughput}} \times F_{\text{SFU}} \tag{9}$$

$$F_{\text{SFU}} = \min\left\{\max\left\{\underbrace{\frac{\#\text{SFU\_insts}}{\#\text{insts}}}_{\text{SFU inst. ratio}} - \underbrace{\frac{\text{SFU\_width}}{\text{SIMD\_width}}}_{\text{SFU exec. unit ratio}}, 0\right\}, 1\right\}. \tag{10}$$

Control-Flow Divergence and Bank Conflict Overheads, O_{CFdiv} and O_{bank}: The overhead of control-flow divergence (O_{CFdiv}) is the cost of executing additional instructions due, for instance, to divergent branches [9]. This cost is modeled by counting all the instructions in both paths. The cost of bank conflicts (O_{bank}) can be calculated by measuring the number of shared memory bank conflicts. Both O_{CFdiv} and O_{bank} can be measured using hardware counters. However, for control-flow divergence, we use our instruction analyzer (Section 5.2), which provides more detailed statistics.

4.1.2 Calculating the Memory Access Cost, T_{mem}

T_{mem} represents the amount of time spent on memory requests and transfers. This cost is a function of the number of memory requests, memory latency per each request, and the degree of memory-level parallelism. We model T_{mem} using Equation (11),

$$T_{\text{mem}} = \underbrace{\frac{\#\text{mem\_insts} \times \#\text{total\_warps}}{\#\text{active\_SMs} \times \text{ITMLP}}}_{\text{Effective memory requests per SM}} \times \text{AMAT}, \tag{11}$$

where AMAT models the average memory access time, accounting for cache effects. We compute AMAT using Equations (12) and (13):

$$\text{AMAT} = \text{avg\_DRAM\_lat} \times \text{miss\_ratio} + \text{hit\_lat} \quad (12)$$

$$\text{avg\_DRAM\_lat} = \text{DRAM\_lat} + (\text{avg\_trans\_warp} - 1) \times \Delta. \quad (13)$$

avg_DRAM_lat represents the average DRAM access latency and is a function of the baseline DRAM access latency, DRAM_lat, and transaction departure delay, Δ. In GPGPUs, memory requests can split into multiple transactions. In our model, avg_trans_warp represents the average number of transactions per memory request in a warp. Note that it is possible to expand Equation (12) for multiple levels of cache, which we omit for brevity.

We model the degree of memory-level parallelism through a notion of inter-thread MLP, denoted ITMLP, which we define as the number of memory requests per SM that is concurrently serviced. Similar to ITILP, memory requests from different warps can be overlapped. Since MLP is an indicator of intra-warp memory-level parallelism, we need to consider the overlap factor of multiple warps. ITMLP can be calculated using Equations (14) and (15).

$$\text{ITMLP} = \min\left(\text{MLP} \times \text{MWP}_{\text{cp}}, \text{MWP}_{\text{peak\_bw}}\right) \quad (14)$$

$$\text{MWP}_{\text{cp}} = \min\left(\max\left(1, \text{CWP} - 1\right), \text{MWP}\right) \quad (15)$$

In Equation (14), MWP_{cp} represents the number of warps whose memory requests are overlapped during one computation period. As described in Section 2.1, MWP represents the *maximum* number of warps that can simultaneously access memory. However, depending on CWP, the number of warps that can concurrently issue memory requests is limited.

$\text{MWP}_{\text{peak\_bw}}$ represents the number of memory warps per SM under peak memory bandwidth. Since the value is equivalent to the maximum number of memory requests attainable per SM, ITMLP cannot be greater than $\text{MWP}_{\text{peak\_bw}}$.

4.1.3 Calculating the Overlapped Cost, T_{overlap}

T_{overlap} represents how much the memory access cost can be hidden by multi-threading. In the GPGPU execution, when a warp issues a memory request and waits for the requested data, the execution is switched to another warp. Hence, T_{comp} and T_{mem} can be overlapped to some extent. For instance, if multi-threading hides all memory access costs, T_{overlap} will equal T_{mem}. That is, in this case the overall execution time, T_{exec}, is solely determined by the computation cost, T_{comp}. By contrast, if none of the memory accesses can be hidden in the worst case, then T_{overlap} is 0.

We compute T_{overlap} using Equations (16) and (17). In these equations, F_{overlap} approximates how much T_{comp} and T_{mem} overlap and N represents the number of active warps per SM as in Equation (4). Note that F_{overlap} varies with both MWP and CWP. When CWP is greater than MWP (e.g., an application limited by memory operations), then F_{overlap} becomes 1, which means all of T_{comp} can be overlapped with T_{mem}. On the other hand, when MWP is greater than CWP (e.g., an application limited by computation), only part of computation costs can be overlapped.

$$T_{\text{overlap}} = \min(T_{\text{comp}} \times F_{\text{overlap}}, T_{\text{mem}}) \quad (16)$$

$$F_{\text{overlap}} = \frac{N - \zeta}{N}, \quad \zeta = \begin{cases} 1 \ (\text{CWP} \leq \text{MWP}) \\ 0 \ (\text{CWP} > \text{MWP}) \end{cases} \quad (17)$$

4.2 Potential Benefit Prediction

As discussed in Section 3, the potential benefit metrics indicate performance improvements when it is possible to eliminate the delta between the ideal performance and the current performance. Equations (18) and (19) are used to estimate the ideal compute and memory performance (time). Alternatively, an algorithm developer might provide these estimates.

Model Parameter	Definition	Source
#insts	# of total insts. per warp (excluding SFU insts.)	Sec. 5.1
#mem_insts	# of memory insts. per warp	Sec. 5.1
#sync_insts	# of synchronization insts. per warp	Sec. 5.2
#SFU_insts	# of SFU insts. per warp	Sec. 5.2
#FP_insts	# of floating point insts. per warp	Sec. 5.2
#total_warps	Total number warps in a kernel	Sec. 5.1
#active_SMs	# of active SMs	Sec. 5.1
N	# of concurrently running warps on one SM	Sec. 5.1
AMAT	Average memory access latency	Sec. 5.1
avg_trans_warp	Average memory transactions per memory request	Sec. 5.2
avg_inst_lat	Average instruction latency	Sec. 5.2
miss_ratio	Cache miss ratio	Sec. 5.1
size_of_data	The size of input data	source code
ILP	Inst.-level parallelism in one warp	Sec. 5.3
MLP	Memory-level parallelism in one warp	Sec. 5.3
MWP (Per SM)	Max #warps that can concurrently access memory	Appx.A
CWP (Per SM)	# of warps executed during one mem. period plus one	Appx.A
$\text{MWP}_{\text{peak\_bw}}$ (Per SM)	MWP under peak memory BW	Appx.A
warp_size	# of threads per warp	32
Γ	Machine parameter for sync cost	64
Δ	Transaction departure delay	Table 3
DRAM_lat	Baseline DRAM access latency	Table 3
FP_lat	FP instruction latency	Table 3
hit_lat	Cache hit latency	Table 3
SIMD_width	# of scalar processors (SPs) per SM	Table 3
SFU_width	# of special function units (SFUs) per SM	Table 3

Table 2. Summary of input parameters used in equations.

$$T_{\text{fp}} = \frac{\text{\#FP\_insts} \times \text{\#total\_warps} \times \text{FP\_lat}}{\text{\#active\_SMs} \times \text{ITILP}} \quad (18)$$

$$T_{\text{mem\_min}} = \frac{\text{size\_of\_data} \times \text{avg\_DRAM\_lat}}{\text{MWP}_{\text{peak\_bw}}} \quad (19)$$

Then, the benefit metrics are obtained using Equations (20)-(23), where $\text{ITILP}_{\text{max}}$ is defined in Equation (5):

$$B_{\text{itilp}} = W_{\text{parallel}} - \frac{\text{\#insts} \times \text{\#total\_warps} \times \text{avg\_inst\_lat}}{\text{\#active\_SMs} \times \text{ITILP}_{\text{max}}} \quad (20)$$

$$B_{\text{serial}} = W_{\text{serial}} \quad (21)$$

$$B_{\text{fp}} = T_{\text{comp}} - T_{\text{fp}} - B_{\text{itilp}} - B_{\text{serial}} \quad (22)$$

$$B_{\text{memlp}} = \max\left(T'_{\text{mem}} - T_{\text{mem\_min}}, 0\right). \quad (23)$$

5. Frontend Data Collector

As described in Section 4, the GPGPU analytical model requires a variety of information on the actual binary execution. In our framework, the frontend data collector does the best in accurately obtaining various types of information that instantiates the analytical model. For this purpose, the frontend data collector uses three different tools/ways to extract the information: compute visual profiler, instruction analyzer (IA), and static analysis tools, as shown in Figure 5.

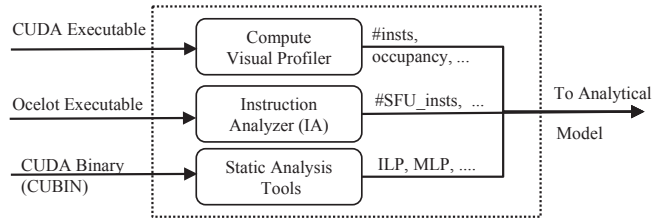

Figure 5. Frontend data collector.

5.1 Compute Visual Profiler

We use Compute Visual Profiler [14] to access GPU hardware performance counters. It provides accurate architecture-related information: occupancy, total number of global load/store requests, the number of registers used in a thread, the number of DRAM reads/writes and cache hits/misses.

5.2 Instruction Analyzer

Although the hardware performance counters provide accurate runtime information, we still cannot obtain some crucial information. For example, instruction category information, which is essential for considering the effects of synchronization and the overhead of SFU utilization, is not available.

Our instruction analyzer module is based on Ocelot [6], a dynamic compilation framework that emulates PTX execution. The instruction analyzer collects instruction mixture (SFU, Sync, and FP instructions) and loop information such as loop trip counts. The loop information is used to combine static analysis from CUDA binary (CUBIN) files and run-time execution information. Although there is code optimization from PTX to CUBIN, we observe that most loop information still remains the same.

5.3 Static Analysis Tools

Our static analysis tools work on PTX, CUBIN and the information from IA. The main motivation for using static analysis is to obtain ILP and MLP information in binaries rather than in PTX code. Due to instruction scheduling and register allocation, which are performed during target code generation, the degree of parallelism between PTX and CUBIN can be significantly different. Hence, it is crucial to calculate ILP/MLP on binaries.

First, we disassemble a target CUBIN file with cuobjdump [13]. We then build a control flow graph (CFG) and def-use chains with the disassembled instructions. The number of memory requests between a load request (def) and the first instruction that sources the memory request (use) is a local MLP. The average of local MLPs in a basic block is the basic block MLP. For ILP, we group the instructions that can be scheduled together within a basic block. (If an instruction has true dependencies to any of other instructions, they cannot be issued at the same cycle.) Then, the basic block ILP is the number of instructions in the basic block over the number of groups in the block.

Second, we combine this static ILP/MLP information with the dynamic information from IA.[6] Basically, we give high weights to ILP/MLP based on basic block execution frequency. The following equation shows the exact formula. In the equation, ILP/MLP for basic block (BB) K is denoted as $\text{ILP(MLP)}_{\text{K}}$. $\text{ILP(MLP)}_{\text{AVG}}$ is the same as ILP/MLP in Equations (4) and (14).

$$\text{ILP(MLP)}_{\text{AVG}} = \sum_{\text{K}=1}^{\#\text{BBs}} \frac{\text{ILP(MLP)}_{\text{K}} \times \#\text{accesses\_to\_BB}_{\text{K}}}{\#\text{basic\_blocks}}$$

6. Results

Evaluation: Our evaluation consists of two major parts.

- We show that our GPGPU analytical model improves over the prior state-of-the-art for current generation Fermi-based GPUs (Section 6.1). Since this model is the basis for our benefit analysis, validating is critical.
- We show how the benefit metrics can be applied to a variety of GPGPU codes (Sections 6.2–6.4). Our goal is to see to what extent our benefit metrics can help assess potential performance bottlenecks and candidate optimizations.

Processor Model: For all of our experiments, we use NVIDIA's Tesla C2050, whose hardware specifications appear in Table 3. Memory model parameters used in this study (DRAM_lat and Δ) are measured using known techniques [9], while cache latencies and FP latency are obtained using micro-benchmarks which we design for this study. We use the default L1/shared configuration where the cache sizes are 16KB and 48KB, respectively. We use the CUDA 3.2 Toolkit.

Workloads: For Section 6.1, we use micro-benchmarks that are designed to have different ILP values and FMA (fused multiply-add) instructions. For real code, we select the workloads using the following criteria: kernels from a full application (Section 6.2), CUDA SDK (Section 6.3), and a public benchmark suite (Section 6.4).

Model	Tesla C2050
Processor clock	1.15 GHz
Memory bandwidth	144.0 GB/s
Compute capability	2.0
The number of SMs	14
The number of SPs (per SM)	32
The number of SFUs (per SM)	4
Shared memory (per SM)	48 KB
L1 cache (per SM)	16 KB
L2 cache	768 KB
FP latency, FP_lat	18
DRAM latency, DRAM_lat	440
Departure delay, Δ	20
L1 cache latency	18
L2 cache latency	130

Table 3. The specifications of the GPGPU and model parameters used in this study.

6.1 Improvements of the Analytical Model

Inter-Thread ILP, ITILP in Eq. (4): Figure 6 shows the performance measured on Tesla C2050 when we have different ILP and TLP for the same kernel. For instance, ILP=2 indicates that all warps have the same ILP value of two. The x-axis represents TLP as the number of active warps per SM, and the y-axis is normalized to the performance when both ILP and TLP are one. The result shows that increasing ITILP leads to a performance improvement up to some points by providing more parallelism, but after the points, it does not help improve performance by merely increasing ILP or TLP. In the graph, the performance becomes stable at the points where ITILP is around 18-24. (The number of warps is 18 for ILP=1, 10 for ILP=2, and eight for ILP=3.) As shown in Figure 7, our analytical model captures this effect and provides better analysis by modeling ITILP, which is not considered in the MWP-CWP model. Figure 7 also shows that B_{itilp} adequately estimates the possible performance gains by providing inter-thread ILP.

Awareness of SFU Contentions, O_{SFU} in Eq. (9): Figure 8 shows the execution time and B_{serial} when we vary the number of SFU instructions per eight FMA instructions. The line graphs represent the actual execution and the predicted time of MWP-CWP and new models, while the bar graph shows B_{serial} normalized to the predicted time of each variation. The result shows that the cost of SFU instructions is almost hidden when there is one SFU instruction per eight FMAs. As the number of SFU instructions increases, however, the execution of SFU instructions significantly contributes to the total execution time, and our model adequately reflects the overhead cost while the MWP-CWP model fails to predict the overhead. In addition, B_{serial} indicates that the serialization

[6] We match loop information from IA and basic block information from static analysis to estimate the basic block execution counts.

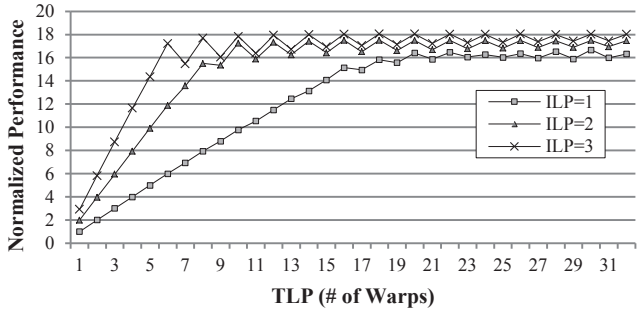

Figure 6. ITILP micro-benchmark on Tesla C2050.

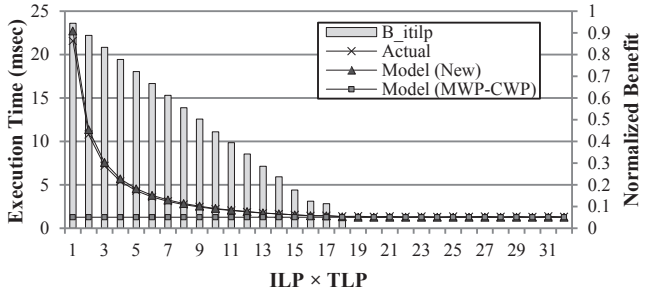

Figure 7. ITILP: Comparison between MWP-CWP and new models.

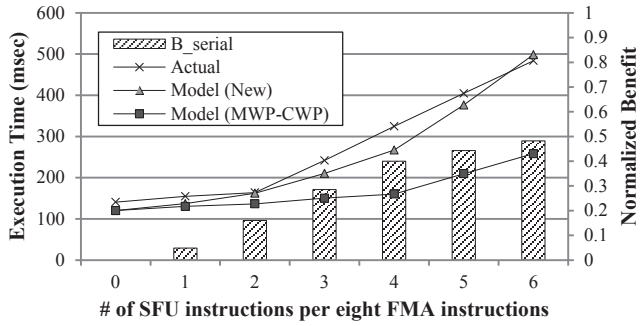

Figure 8. The execution time and B_{serial} when increasing the number of SFU instructions per eight FMA instructions.

effects might be the performance bottleneck, which implies that the ratio of SFU and FMA instructions is not optimal.

Code optimization effects: As we have discussed in Section 2.2, our model improves on the previously proposed the MWP-CWP model in a way that can differentiate distinct optimizations. Figure 9 shows the comparisons between the two models for real code of FMM_U (Section 6.2). The y-axis shows the performance improvements when different optimization techniques are applied to the baseline in FMM_U. The result shows that our model successfully predicts the performance impact resulting from code optimizations, but the MWP-CWP model often estimates the benefit as less than the actual one, or even the opposite. For instance, although prefetching improves performance in real hardware, the MWP-CWP model predicts that the optimization leads to a performance degradation. In general, as shown in Figure 9, our model can estimate the performance delta resulting from several code optimizations more accurately than the MWP-CWP model.

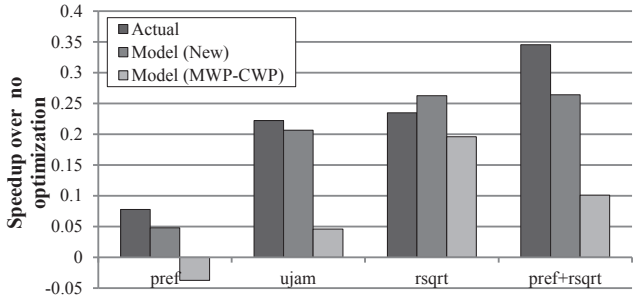

Figure 9. The performance improvement prediction over the baseline FMM_U of the MWP-CWP and our models.

6.2 Case Study of Using the Model: FMM_U

We apply our model to an implementation of the *fast multipole method* (FMM), an $O(n)$ approximation method for n-body computations, which normally scale as $O(n^2)$ [8]. We specifically consider the most expensive phase of the FMM, called the *U-list* phase (FMM_U), which is a good target for GPU acceleration. The FMM_U phase appears as pseudocode in Algorithm 1.

Algorithm 1 FMM_U algorithm

```
1: for each target leaf node, B do
2:   for each target point t ∈ B do
3:     for each neighboring source node, S ∈ U(B) do
4:       for each source point s ∈ S do
5:         φ_t += F(s,t) /* E.g., force evaluation */
```

The parallelism and dependency structure are straightforward. The loop iterations at line 1 are completely independent, as are those at line 2. Using an owner-computes assignment of threads to targets is natural. The loop in lines 4–5 implements a reduction. There is considerable data reuse; the sizes of B and S in Algorithm 1 are always bounded by some constant q, and there are $O(q^2)$ operations on $O(q)$ data, where q is typically $O(1,000)$. The FMM_U is typically compute bound.

6.2.1 FMM Optimizations

Prefetching (pref): Prefetching generates memory requests in advance of their use to avoid stalls. For FMM_U, the data access patterns are fairly predictable. We consider prefetching $s \in S$ into registers. On current GPUs, explicit software prefetching can help since there are no hardware prefetchers. Prefetching can increase memory- and instruction-level parallelism.

Use Shared Memory (shmem): To limit register pressure, we can use shared memory (scratchpad space) instead. In FMM_U, it is natural to use this space to store large blocks of S. This approach yields two benefits. First, we increase memory-level parallelism. Second, we increase the reuse of source points for all targets.

Unroll-and-Jam (ujam): To increase register-level reuse of the target points, we can unroll the loop at line 2 and then fuse (or "jam") the replicated loop bodies. The net effects are to reduce branches, thereby increasing ILP, as well as to increase reuse of each source point s. The trade-off is that this technique can also increase register live time, thereby increasing the register pressure.

Data Layout (trans): When we load points data into GPU shared memory, we can store these points using an array-of-structures (AoS) or structure-of-arrays (SoA) format. By default, we use SoA; the "trans" optimization uses AoS.

Vector Packing (vecpack): Vector packing "packs" multiple memory requests with short data size into a larger request, such as

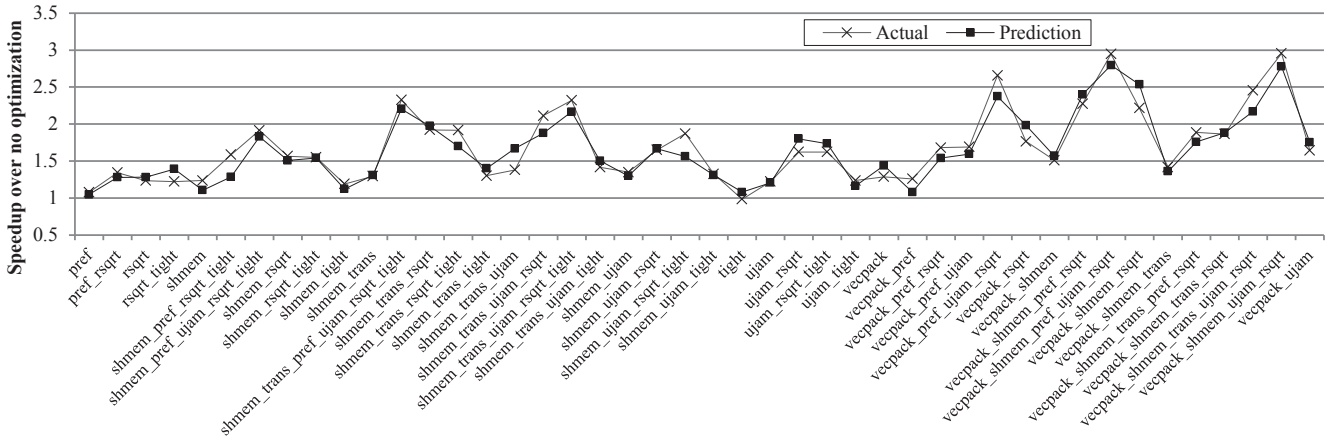

Figure 10. Speedup over the baseline of actual execution and model prediction on 44 different optimizations.

replacing four separate 32-bit requests into a single 128-bit request, which can be cheaper. This technique increases MLP as well as reduces the number of instructions. Vector packing is also essentially a structure-of-arrays to array-of-structures transformation.

Tight (tight): The "tight" optimization utilizes the 'float4' datatype to pack data elements. However, this technique is different from vecpack above since it still issues four of the 32-bit memory requests.

Reciprocal Square Root - SFU (rsqrt): Our FMM_U can replace separate divide and square root operations with a single reciprocal square-root. This exploits the dedicated special function unit (SFU) hardware available for single-precision transcendental and multiplication instructions on NVIDIA GPUs.

6.2.2 Performance Prediction

Figure 10 shows the speedup over the baseline kernel of actual execution and its prediction using the proposed model. The x-axis shows the code optimization space, where 44 optimization combinations are presented. The results show that, overall, the model closely estimates the speedup of different optimizations. More importantly, the proposed model follows the trend and finds the best optimization combinations, which makes our model suitable for identifying potential performance benefits.

6.2.3 Potential Benefit Prediction

To understand the performance optimization guide metrics, we first compute potential benefit metrics for the baseline, which are as follows: $B_{\text{serial}} = 0$, $B_{\text{memlp}} = 0$, $B_{\text{itilp}} = 6068$, and $B_{\text{fp}} = 9691$. Even from the baseline, it is already limited by computation. Hence, techniques to reduce the cost of computation are critical.

Figure 11 shows actual performance benefits and B_{itilp} when the shared memory optimization ($shmem$) is applied on top of different combinations of optimizations. For example, $ujam$ indicates the performance delta between $ujam$ and $ujam + shmem$ optimizations. In the graph, both $Actual$ and B_{itilp} are normalized to the execution time before $shmem$ is applied. Using the shared memory improves both MLP and ILP. It increases the reuse of source points, which also increases ILP. Hence, the benefit of using the shared memory can be predicted using B_{itilp}, because $B_{\text{memlp}} = 0$. As shown in the figure, B_{itilp} predicts the actual performance benefit closely for most of the optimization combinations except $tight$ optimizations. The interaction between $shmem$ and $tight$ optimizations should be analyzed further.

Figure 12 represents the performance speedup and potential benefits normalized to the baseline execution time when optimizations are applied one by one. For the baseline, a high B_{fp} value indicates that the kernel might have the potential to be improved. Hence, it is a reasonable decision to try to optimize the baseline kernel. As we apply adequate optimization techniques that improve the kernel, the potential benefit decreases. When $vecpack$, $rsqrt$, $shmem$ and $ujam$ have been applied, the potential benefit metric indicates that the kernel may have very small amount of inefficient

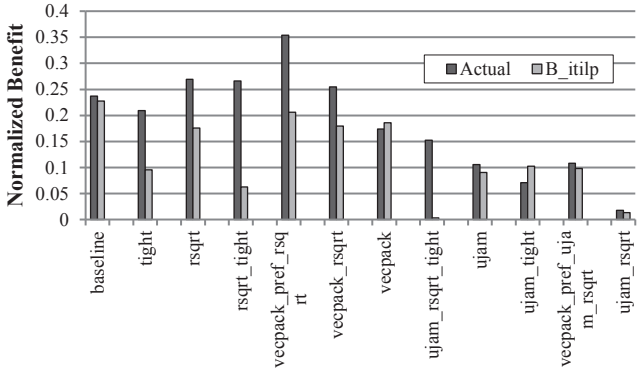

Figure 11. Actual performance benefit and B_{itilp} when $shmem$ is applied to each optimization in the x-axis.

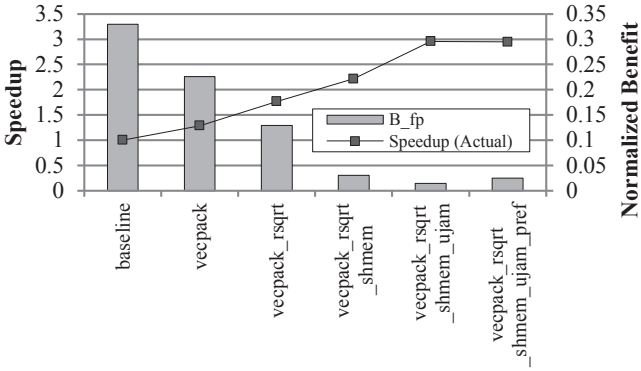

Figure 12. Performance speedup and potential benefits when applying optimizations to the baseline one by one.

computation and the potential performance gain through further optimizations might be limited. As shown in the last bar, $pref$ does not lead to a performance improvement. Rather, B_{fp} slightly increases due to the inefficient computation for prefetching code.

6.3 Reduction

6.3.1 Optimization Techniques

Reduction has seven different kernels (K0−K6) to which different optimization techniques are applied. Table 4 shows the optimization techniques applied to each kernel.

Eliminating Divergent Branches: Branch divergence occurs if threads in a warp take different paths due to conditional branches. When a warp executes divergent branches, it serially executes both branch paths while disabling threads that are not on the current path. By eliminating divergent branches, we can increase the utilization of the execution units.

Eliminating Shared Memory Bank Conflicts: Eliminating bank conflicts causes shared memory banks to be serviced simultaneously to any shared memory read or write requests.

Reducing Idle Threads: As reduction proceeds, the number of threads that work on data reduces. Thus, this technique packs more computations into threads, thereby reducing the number of idle threads.

Loop Unrolling: As described in Section 6.2, loop unrolling alleviates the loop overhead while eliminating branch-related instructions. Thus, reducing the number of instructions will improve performance for compute-bound kernels.

	K0	K1	K2	K3	K4	K5	K6
Eliminating Divergent Branches		O	O	O	O	O	O
Eliminating Bank Conflicts			O	O	O	O	O
Reducing Idle Threads (Single)				O	O	O	
Loop Unrolling (Last Warp)					O		
Loop Unrolling (Completed)						O	O
Reducing Idle Threads (Multiple)							O

Table 4. Optimization techniques on reduction kernels.

6.3.2 Results

Figure 13 shows the actual execution time and the outcome of the analytical model. The model closely estimates the execution time over optimizations. Figure 14 shows the benefit X-Y chart of the reduction kernels. From K0 to K5, the kernels are in the compute-bounded zone. All the memory operation cost is hidden because the cost of computation is higher than that of memory operations; thus, B_{memlp} is zero because of $T'_{\text{mem}} = 0$. In addition, even from the K0 kernel, the ITILP value is already the peak value, so B_{itilp} is also zero.

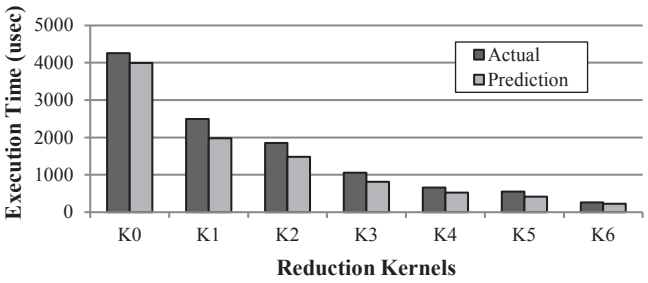

Figure 13. Actual and model execution time of the reduction kernels.

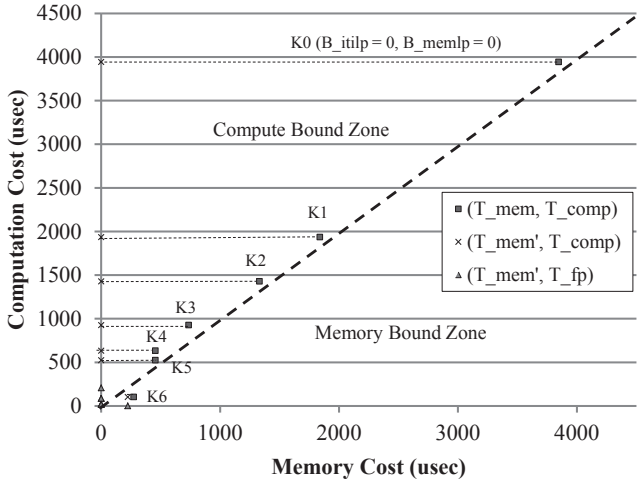

Figure 14. Benefit X-Y chart of the reduction kernels.

From the chart, we can observe that there are huge gaps between the (T'_{mem}, T_{comp}) and (T'_{mem}, T_{fp}) of each kernel, which implies that the kernels suffer from computing inefficiency. Once we eliminate most of the inefficiency of computation, kernel (K6) becomes memory bounded.

The benefit X-Y chart also indicates that, although some optimization techniques attempt to reduce the overhead of serialization, the actual benefits mainly come from improving the efficiency of computation. In particular, the number of instructions was significantly reduced by each optimization, which proves our findings.

6.4 Parboil Benchmark

6.4.1 Optimization Techniques

Parboil [17] has algorithm-specific optimization techniques. In this section we explain each of them briefly. Table 5 shows the techniques applied to the evaluated applications.

Regularization: This optimization is a preprocessing step that converts irregular workloads into regular ones. In order to do so, the workloads can be spread over multiple kernels or even to CPUs.

Binning: This optimization pre-sorts input elements into multiple bins based on location in space. For each grid point, only bins within a certain distance are used for computation, thereby preventing each thread from reading the entire input.

Coarsening: Thread coarsening is a technique in which each thread works on several elements instead of one to amortize redundant computation overhead.

Tiling/Data Reuse: This optimization moves data, which can be shared by multiple threads, into the shared memory to reduce global memory accesses. To maximize the advantage of the shared memory, computation is distributed among threads.

Privatization: Privatization allows each thread to have its own copy of output data. The results from all threads are later combined when all the writings from all threads are finished. This technique alleviates the contention of write accesses to the same output data by multiple threads.

6.4.2 Results

Figure 15 shows the actual and predicted time for the applications in the parboil benchmarks. The results show that the optimization techniques for `cutcp` are effective, while the techniques for `tpacf` fail to effectively optimize the kernel. Our model also predicts the same.

	cutcp	tpacf	Potential Benefit Metrics
Regularization	O		B_{fp}
Binning	O		B_{fp}
Coarsening	O		B_{fp}
Tiling/Data Reuse		O	B_{memlp}, B_{fp}
Privatization		O	B_{memlp}, B_{serial}, B_{itilp}

Table 5. Optimization techniques on the evaluated applications and relevant potential benefit metrics.

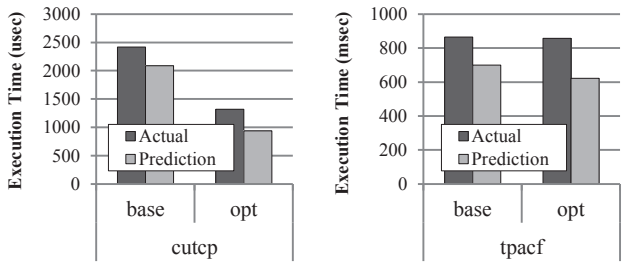

Figure 15. Actual and predicted time of cutcp and tpacf in the parboil benchmarks.

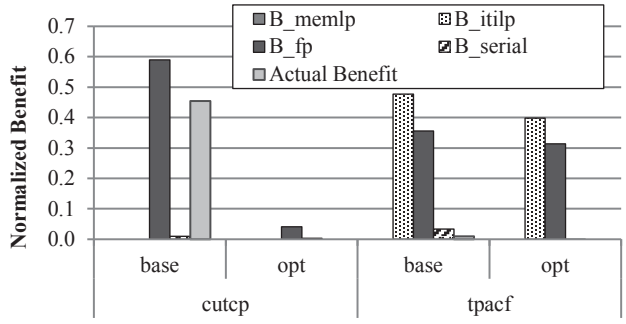

Figure 16. Benefit metrics in the parboil benchmarks.

The results can be further analyzed with the potential benefit metrics in Figure 16, where benefit metrics are normalized to the baseline execution time. The benefit metrics lead to two conclusions. First, `cutcp` achieves most of the potential performance benefits in the optimized version. Hence, trying to further optimize the kernel might not be a good idea. Second, the baseline of `tpacf` also had high performance benefit potential, but the applied optimizations failed to achieve some of the benefits.

Surprisingly, although `tpacf` has the high potential benefits of B_{itilp}, the optimization techniques are not designed for that. This explains why there is almost no performance improvement from the baseline. Tiling, data reuse, and privatization mostly target improving memory operations. In fact, the optimized kernel significantly reduces the number of memory requests, which can lead to a performance improvement in GPGPUs where there are no caches. In the Fermi GPGPUs, unfortunately, most of the memory requests hit the caches, thereby resulting in no B_{memlp}. Hence, programmers or compilers should focus more on other optimizations to increase instruction-level or thread-level parallelism.

7. Related Work

7.1 GPU Performance Modeling

In the past few years, many studies on GPU performance modeling have been proposed. Hong and Kim [9] proposed the MWP-CWP based GPU analytical model. Concurrently, Baghsorkhi et al. [2] proposed a work flow graph (WFG)-based analytical model to predict the performance of GPU applications. The WFG is an extension of a control flow graph (CFG), where nodes represent instructions and arcs represent latencies. Zhang and Owens [20] proposed a performance model where they measured the execution time spent on the instruction pipeline, shared memory, and global memory to find the bottlenecks. Although Zhang and Owens also target identifying the bottlenecks, their method does not provide estimated performance benefits. Furthermore, our analytical model addresses more detailed performance bottlenecks, such as SFU resource contention.

Williams et al. [18] proposed the Roofline model, which is useful for visualizing compute-bounded or memory-bounded multicore architectures. Our X-Y benefit chart not only shows these limitations but also estimates ideal performance benefits.

Our work is also related to a rich body of work on optimizations and tuning of GPGPU applications [4, 7, 12, 15, 16, 19]. Ryoo et al. [16] introduced two metrics to prune optimization space by calculating the utilization and efficiency of GPU applications. Choi et al. [4] proposed a performance model for a sparse matrix-vector multiply (SpMV) kernel for the purpose of autotuning. A GPGPU compiler framework proposed by Yang et al. [19] performs GPGPU-specific optimizations, which can improve naive GPGPU kernels.

7.2 Performance Analysis Tools for CUDA

A handful of tools are available for the performance analysis of CUDA applications. However, most of the tools simply provide the performance metrics of the current running application. On the contrary, GPUPerf estimates potential performance benefit, thereby providing guidelines for how to optimize applications. As we discussed in Section 5, GPUPerf utilizes several tools such as visual profiler [14], Ocelot [6], and cuobjdump [13] to obtain accurate and detailed basic program analysis information.

Kim and Shrivastava [10] presented a tool that can be used for analyzing memory access patterns of a CUDA program. They model major memory effects such as memory coalescing and bank conflict. However, they do not deal with run-time information as their approach is a compile-time analysis, which often leads to inaccurate results for cases such as insufficient parallelism.

Meng et al. [11] proposed a GPU performance projection framework. Given CPU code skeletons, the framework predicts the cost and benefit of GPU acceleration. Their predictions are also built on the MWP-CWP model, but our work greatly improves on the MWP-CWP model.

There are also GPU simulators that can be used for program analysis. A G80 functional simulator called *Barra* by Collange et al. [5] can execute NVIDIA CUBIN files while collecting statistics. Bakhoda et al. [3] analyzed CUDA applications by implementing a GPU simulator that runs PTX instruction set. A heterogeneous simulator called MacSim [1] can also be used for obtaining detailed statistics on CUDA workloads.

8. Conclusions

The GPUPerf framework determines what bottlenecks are present in code and also estimates potential performance benefits from removing these bottlenecks. The framework combines an accurate analytical model for modern GPGPUs (e.g., Fermi-class) as well as a set of interpretable metrics that directly estimate potential im-

provements in performance for different classes of performance optimization techniques. We demonstrate these performance benefit predictions on FMM_U with 44 optimization combinations and several other benchmarks. The results show that the predicted potential benefit estimates are both informative and attainable.

Based on our case studies, we found that among four metrics, B_{itilp} and B_{fp} are easy to exploit through a variety of optimizations. The B_{memlp} metric is often zero indicating that most evaluated CUDA applications are in fact limited only by computation in Fermi-class GPGPUs. The B_{fp} metric generally has a relatively high value, which implies that removing compute inefficiencies is the key to achieving ideal performance. For example, coarsening and binning reduce such compute inefficiencies greatly in the parboil benchmarks.

Our future work will integrate this framework into compiler tools that can be directly used to improve parallel program efficiency on GPGPUs and other many-core platforms.

Acknowledgments

Many thanks to Sunpyo Hong, Jiayuan Meng, HPArch members, and the anonymous reviewers for their suggestions and feedback on improving the paper. We gratefully acknowledge the support of the National Science Foundation (NSF) CCF-0903447, NSF/SRC task 1981, NSF CAREER award 0953100, NSF CAREER award 1139083; the U.S. Department of Energy grant DE-SC0004915; Sandia National Laboratories; the Defense Advanced Research Projects Agency; Intel Corporation; Microsoft Research; and NVIDIA. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF, SRC, DOE, DARPA, Microsoft, Intel, or NVIDIA.

References

[1] MacSim. http://code.google.com/p/macsim/.

[2] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. W. Hwu. An adaptive performance modeling tool for gpu architectures. In *PPoPP*, 2010.

[3] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed GPU simulator. In *IEEE ISPASS*, April 2009.

[4] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *PPoPP*, 2010.

[5] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A parallel functional simulator for gpgpu. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0:351–360, 2010.

[6] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *PACT-19*, 2010.

[7] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju. Auto-tuning of fast fourier transform on graphics processors. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, 2011.

[8] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, Dec. 1987.

[9] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ISCA*, 2009.

[10] Y. Kim and A. Shrivastava. Cumapz: A tool to analyze memory access patterns in cuda. In *DAC '11: Proc. of the 48th conference on Design automation*, June 2011.

[11] J. Meng, V. Morozov, K. Kumaran, V. Vishwanath, and T. Uram. Grophecy: Gpu performance projection from cpu code skeletons. In *SC'11*, November 2011.

[12] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *ICS*, 2009.

[13] NVIDIA. CUDA OBJDUMP. http://developer.nvidia.com.

[14] NVIDIA Corporation. NVIDIA Visual Profiler. http://developer.nvidia.com/content/nvidia-visual-profiler.

[15] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *SC '10*, 2010.

[16] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W. mei W. Hwu. Program optimization space pruning for a multithreaded gpu. In *CGO-6*, pages 195–204, 2008.

[17] The IMPACT Research Group, UIUC. Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php.

[18] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[19] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proc. of the ACM SIGPLAN 2010 Conf. on Programming Language Design and Implementation*, 2010.

[20] Y. Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *HPCA*, 2011.

A. Calculating CWP and MWP

A.1 CWP

$$\text{CWP} = \min\left(\text{CWP\_full}, N\right) \tag{A.1}$$

$$\text{CWP\_full} = \frac{\text{mem\_cycles} + \text{comp\_cycles}}{\text{comp\_cycles}} \tag{A.2}$$

$$\text{comp\_cycles} = \frac{\#\text{insts} \times \text{avg\_inst\_lat}}{\text{ITILP}} \tag{A.3}$$

$$\text{mem\_cycles} = \frac{\#\text{mem\_insts} \times \text{AMAT}}{\text{MLP}} \tag{A.4}$$

mem_cycles: memory waiting cycles per warp
comp_cycles: computation cycles per warp
#insts: number of instructions per warp (excluding SFU insts.)
#mem_insts: number of memory instructions per warp

A.2 MWP

$$\text{MWP} = \min\left(\frac{\text{avg\_DRAM\_lat}}{\Delta}, \text{MWP}_{\text{peak\_bw}}, N\right) \tag{A.5}$$

$$\text{MWP}_{\text{peak\_bw}} = \frac{\text{mem\_peak\_bandwidth}}{\text{BW\_per\_warp} \times \#\text{active\_SMs}} \tag{A.6}$$

$$\text{BW\_per\_warp} = \frac{\text{freq} \times \text{transaction\_size}}{\text{avg\_DRAM\_lat}} \tag{A.7}$$

mem_peak_bandwidth: bandwidth between the DRAM and GPU cores (e.g., 144.0 GB/s in Tesla C2050)
freq: clock frequency of the SM processor (e.g., 1.15 GHZ in Tesla C2050)
transaction_size: transaction size for a DRAM request (e.g., 128B in Tesla C2050)
BW_per_warp: bandwidth requirement per warp