An Adaptive Performance Modeling Tool for GPU Architectures

Sara S. Baghsorkhi Matthieu Delahaye Sanjay J. Patel William D. Gropp Wen-mei W. Hwu

University of Illinois at Urbana-Champaign
Urbana, IL 61801
{bsadeghi, matthieu, sjp, wgropp, hwu} @illinois.edu

Abstract

This paper presents an analytical model to predict the performance of general-purpose applications on a GPU architecture. The model is designed to provide performance information to an auto-tuning compiler and assist it in narrowing down the search to the more promising implementations. It can also be incorporated into a tool to help programmers better assess the performance bottlenecks in their code. We analyze each GPU kernel and identify how the kernel exercises major GPU microarchitecture features. To identify the performance bottlenecks accurately, we introduce an abstract interpretation of a GPU kernel, *work flow graph*, based on which we estimate the execution time of a GPU kernel. We validated our performance model on the NVIDIA GPUs using CUDA (Compute Unified Device Architecture). For this purpose, we used data parallel benchmarks that stress different GPU microarchitecture events such as uncoalesced memory accesses, scratch-pad memory bank conflicts, and control flow divergence, which must be accurately modeled but represent challenges to the analytical performance models. The proposed model captures full system complexity and shows high accuracy in predicting the performance trends of different optimized kernel implementations. We also describe our approach to extracting the performance model automatically from a kernel code.

Categories and Subject Descriptors C.1.2 [*Processor Architecture*]: Multiple Data Stream Architectures; C.4 [*Performance of Systems*] – Modeling Techniques

General Terms Design, Measurement, Performance

Keywords Analytical model, GPU, Parallel programming, Performance estimation

1. Introduction

Graphics processors traditionally had highly specialized programming models and interfaces that limit the ability of developers to map general-purpose applications to these platforms. With the introduction of CUDA [17] and OpenCL [15], developers now have the programming and architectural features to quickly port programs to a platform with a massively parallel, GPU-based coprocessor [16]. The intent of our work is to model the GPU organization and features for analyzing the performance of general-purpose applications. In this paper we focus on CUDA-enabled NVIDIA GPUs. However, the model is not tightly coupled to any specific GPU architecture or high-level programming interface. In Section 1.2 we lay out the common design features of GPU architectures that are integrated into our model. Our framework can similarly collect information about microarchitectural effects of program statements that are expressed through OpenCL extensions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'10, January 9–14, 2010, Bangalore, India.
Copyright © 2010 ACM 978-1-60558-708-0/10/01...$10.00.

1.1 Motivation

The amount of effort required to maximize the performance of applications on GPU architectures can be relatively high. Due to resource restrictions and the threading model of the GPU, the optimization space can also be discontinuous. A study by Ryoo et al. [19] demonstrated a very large configuration space even for relatively small kernels. Ryoo et al. also concluded that the difference in performance between manually optimized variants of code and the optimal configuration was 17%.

Empirical performance tuning is a well-known technique for solving the above pitfalls. Despite significant research to develop models and frameworks for predicting performance of applications [21, 14, 3, 18], most of it applies to non-GPU architecture.

In this paper we demonstrate a performance model to help prune the search space of GPU kernel optimizations. The model can be used as a supporting module for an automated optimizing compiler for GPU architecture.

1.2 Performance Factors

GPUs support the Single-Program Multiple-Data (SPMD) model. Threads within certain granularities – *thread-blocks* in NVIDIA GPUs, *groups* in ATI GPUs [1], and *work-groups* in OpenCL – share their data and synchronize their actions. During execution, threads within a thread-block are grouped into *warps*, which are the granular multi-threading scheduling units. Threads in a warp are executed in SIMD mode, and warps can be interleaved with hardware multi-threading to tolerate intra-warp stalls, which enables overlap of memory latency with useful computation. A warp is conceptually equivalent to a *wavefront* in ATI GPUs. In this work, the term $\mathrm{SIMD_{work}}$ indicates the size of a warp. A warp is executed on a streaming multiprocessor (SM). We use the term $\mathrm{SIMD_{engine}}$ to refer to the number of streaming processors for each SM. Therefore, to execute an instruction for all the threads in a warp, the SM should be clocked $\frac{\mathrm{SIMD_{work}}}{\mathrm{SIMD_{engine}}}$ times.

We now briefly review major microarchitecture features that are considered in analyzing the performance of a kernel.

1. GPUs generate and maintain thousands of threads to tolerate memory and SIMD pipeline latency. A high compute-to-memory-access ratio is also necessary to avoid saturation of memory channels.
2. To conserve global memory bandwidth, when the neighboring threads that access the memory simultaneously execute a global

load, the loads are consolidated if they meet constraints necessary for the hardware to perform memory coalescing.

3. Working memory within a group of cores consists of software-managed cache memory (shared memory in CUDA or local memory in OpenCL). These high fan-out, low latency, limited-capacity memories are partitioned among thread-blocks.
4. Memories have a limited number of ports, so appropriate thread ordering helps preserve performance by avoiding bank conflicts.
5. The GPU programming model is based on the SPMD model in general and the SIMD mode among subsets of threads (warps in NVIDIA GPUs or wavefronts in ATI GPUs). Although this is a cost-effective hardware model for exploiting data parallelism, it can be ineffective for algorithms that require diverging control flow decisions in data-parallel sections of the code.

1.3 Contributions

Previous studies on performance estimation [12, 8, 6] and tuning [11] for GPUs were constrained by the programming environment and the necessity of mapping algorithms to existing GPU features.

More recently, Ryoo et al. [19] used Pareto-optimal curves to prune the optimization space of general-purpose applications on GPUs. They introduce efficiency (a flat instruction count) and utilization (a measure of how many pure compute cycles are available from other executing warps) as single number metrics. They did not model memory latency and assumed that none of the kernels were memory bound. A more recent work [10] proposes an analytical model to capture a rough estimate of the cost of memory operations. Neither of these approaches [10, 19] take into account performance factors such as diverging control flow, memory bank conflicts and SIMD pipeline delays. Furthermore, in the current work the accuracy has been significantly improved owing to a compiler-based approach to collect information on microarchitectural effects. The work by Schaa et al. [20] focuses on how to predict the execution time for a multi-GPU system, knowing the execution time for a single GPU.

The proposed performance model in this work captures performance effects of all major GPU microarchitecture features. We also explain a systematic approach to analyzing the code and initializing the the performance model with accurate information. For this purpose, we revisited the program dependence graph (PDG) [7], an intermediate program representation, for the purpose of performance evaluation. The PDG provides a coherent framework to explicitly represent control and data dependences for each program operation. Based on the PDG representation, we can identify computationally related operations in the program that exercise key performance factors. We also devise a tractable framework for performing symbolic evaluation of certain fragments of code in order to determine loop bounds, data access patterns, control flow patterns, etc. These program characteristics are helpful for estimating the effects of control flow divergence, memory bank conflicts, and memory coalescing.

Another contribution of this work compared to the previous work [19, 10] is that we measure each performance factor in isolation (through symbolic evaluation or dynamic instrumentation) and later combine them to model the overall performance. Therefore, the interactive effects between different performance factors are modeled correctly. To accomplish this purpose, we introduce the *work flow graph*.

We would like to note that the key factors for the success of the model and tractability of the symbolic evaluation to extract initial information from the source code are the hardware constraints of GPU architecture and the data-parallel programming model.

2. Performance Model

In a GPU architecture, threads within certain granularities (thread-blocks), share their data and synchronize their actions. During execution threads within a block are grouped into warps. A warp is the SIMD work granularity or a batch of threads that are executed in lock-step SIMD fashion by the hardware. Warps can be interleaved with hardware multi-threading to tolerate intra-warp latencies. Interleaving execution of warps at thread-block and multi-thread-block levels is very similar to the notion of thread-level parallelism (TLP), except that warps are special forms of medium-grain threads. In this work we use the term warp-level parallelism (WLP) when we refer to the concurrency at this level.

For each kernel, we calculate the maximum warp-level parallelism, $\mathrm{WLP}_{\mathrm{max}}$, which is the maximum number of warps that can be simultaneously assigned to a GPU streaming multiprocessor without violating local resource usage, i.e., the amount of shared memory and the number of registers used by a kernel along with other hardware limits. We currently rely on NVIDIA's compiler to track kernel resource usage.

As diverging control flow turns off a number of active warps for steps with a sparse computation pattern, WLP can change from one phase of computation to another. We call the WLP that is restrained to a segment of code $\mathrm{WLP}_{\mathrm{local}}$. Similarly we name the average WLP available throughout the whole kernel $\mathrm{WLP}_{\mathrm{avg}}$. Details on computing these parameters are discussed in Section 2.2.

Although warps can be interleaved in any order by the hardware scheduler, we assume that warps within the same thread-block are relatively synchronous, while warps from different thread-blocks run asynchronously. With this assumption, we compute the effective WLP ($\mathrm{WLP}_{\mathrm{effect}}$) according to Equation (1), where $\mathrm{NUM}_{\mathrm{blocks}}$ is the number of active thread-blocks on a streaming multiprocessor.

$$\mathrm{WLP}_{\mathrm{effect}} = \frac{\mathrm{WLP}_{\mathrm{local}} + (\mathrm{NUM}_{\mathrm{blocks}} - 1) \times \mathrm{WLP}_{\mathrm{avg}}}{\mathrm{NUM}_{\mathrm{blocks}}} \quad (1)$$

Equation (1) illustrates that as a warp observes other warps in its own thread-block at the same phase of computation, the amount of WLP contribution from them is equal to $\mathrm{WLP}_{\mathrm{local}}$. Warps from other thread-blocks can be at any random computation point. Accordingly, the amount of WLP delivered by them is approximated by $\mathrm{WLP}_{\mathrm{avg}}$. Notice that in a GPU kernel with no control flow divergence, $\mathrm{WLP}_{\mathrm{local}}$, $\mathrm{WLP}_{\mathrm{avg}}$, and $\mathrm{WLP}_{\mathrm{effect}}$ are all equal to $\mathrm{WLP}_{\mathrm{max}}$.

At the warp level, GPUs attempt to reduce memory latency by exploiting the data-level parallelism (DLP); DLP is achieved by operating on multiple memory banks simultaneously through a single vector instruction and by having SIMD instructions exploit the simultaneously fetched memory words. The key performance indicator at the warp level is the efficient utilization of the SIMD pipeline, and the long and low-latency memory bandwidth. To model these effects we use symbolic evaluation to determine the access patterns of vector memory instructions with respect to memory bank configuration and coalescing rules. Conflicts and uncoalesced accesses lead to a decrease in DLP. In case of a branch divergence, different threads may follow different control flow paths. Through symbolic evaluation we identify the pattern of diverging threads and the corresponding warps that follow each path. A weight is assigned to each diverging control flow path based on the number of warps that execute it. A warp that is counted toward the weight of a control flow path with some of its threads turned off, will exhaust the SIMD pipeline bandwidth.

At the thread level, instruction-level parallelism (ILP) can still improve performance by partially covering intra-warp stalls. In our model intra-warp stalls result from global memory loads (memory latency) and back-to-back register dependencies (pipeline latency).

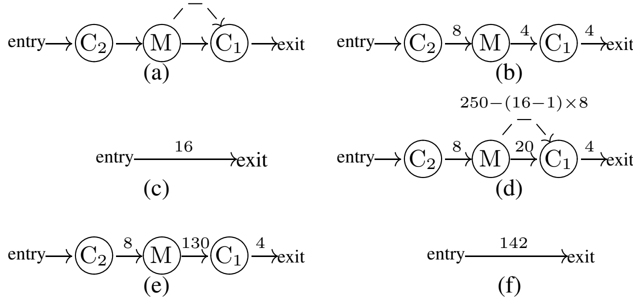


Figure 1. WFG for statement x = A[threadID] + y, which breaks down to two address calculations, a memory load, and an add instruction. (a) The initial WFG. (b) Assuming a pipeline latency of 24 cycles, a warp size of 32, a SIMD engine of 8 streaming processors, and 16 active warps ($\mathrm{WLP_{max}}$) on each streaming multiprocessor, $\mathrm{Latency_{comp}}$ is computed to be 1 cycles [Eq. (3)]. (c) Reduction of the WFG results in a $\mathrm{CYC_{compute}}$ of 16 cycles and a $\mathrm{NBC_{avg}}$ of 8 cycles [Eq. (5)]. (d) If M loads a four-byte word and the allocated bandwidth to each multiprocessor is 4 bytes/cycle, the weight assigned to the transition arc from M to $\mathrm{C_1}$ is computed to be 20 cycles [Eq. (4)]. With an average memory latency of 250 cycles, the weight for the data-dependence arc is set to be 130 cycles [Eq. (6)]. (e) The data-dependence arc is collapsed; the weight on the outgoing transition arc from M is adjusted to reflect the uncovered memory latency. (f) The WFG is reduced once more; the weight on the transition arc represents the average warp latency.

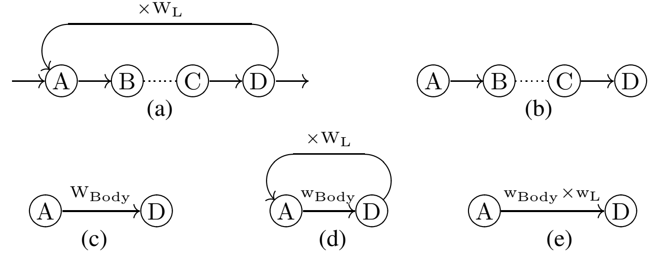


Figure 2. Loop reduction in WFG. (a) A WFG loop with the trip count, $\times\mathrm{W_L}$, labeled on the back-edge arc. (b) Subgraph of the loop body; the back-edge arc is removed. (c) The loop body is reduced; $\mathrm{W_{Body}}$ represents the latency of one iteration. (d) The back-edge arc is added back. (e) Total loop execution latency is equal to $\mathrm{w_{Body}} \times \mathrm{w_L}$.

Available ILP may also vary from one segment of the code to another segment. We approximate the amount of ILP for a region of a kernel, $\mathrm{ILP_{local}}$, by dividing the total number of instructions of that region by the length of the longest def-use chain built from them. In Section 2.2 we show how to calculate $\mathrm{ILP_{avg}}$, the average ILP available in a kernel. Following a similar discussion given earlier for WLP, the amount of ILP observed by a warp at each point of computation is approximated by Equation (2).

$$\mathrm{ILP_{effect}} = \frac{\mathrm{ILP_{local}} + (\mathrm{NUM_{blocks}} - 1) \times \mathrm{ILP_{avg}}}{\mathrm{NUM_{blocks}}} \qquad (2)$$

A performance model's ability to identify bottlenecks and estimate execution cycles accurately depends on examining both the amount of concurrency available in a kernel (WLP, DLP and ILP) and the latencies of the SIMD pipeline and the memory system. To combine these aspects into a coherent framework, we introduce the work flow graph (WFG).

2.1 Work Flow Graph

The work flow graph (WFG) is an extension of the control flow graph of a GPU kernel. Nodes in the WFG are either a long latency (global) memory operation (M), a low latency (scratch-pad) memory operation (S), a barrier synchronization (B), a block of *n* continuous computational instructions ($\mathrm{C_n}$), or synthetic entry and exit nodes. In addition to edges that correspond to the control flow graph (transition arcs), the WFG also contains data-dependence arcs that connect global memory loads to their corresponding uses.

The initial WFG is an abstraction of the computation of a kernel independent of the underlying hardware. No weight is assigned to transition arcs initially. A transition arc only indicates that the destination node can be executed immediately after the source node. Figure 1(a) shows the initial WFG for a simple statement.

To estimate the performance on a particular GPU, we specialize the arcs in the WFG with information that is calculated by symbolic evaluation based on hardware parameters such as memory bandwidth, read latency, memory coalescing rules, memory bank configuration, SIMD work granularity, SIMD engine width,

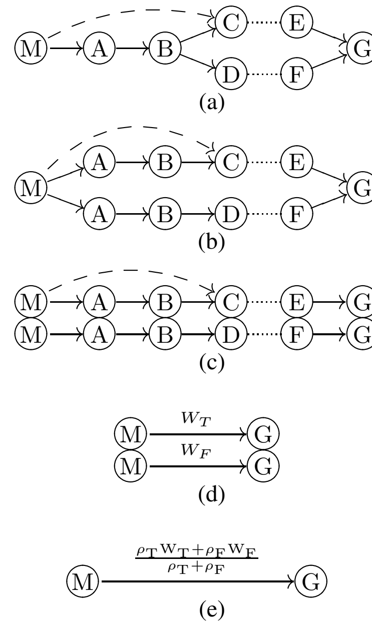


Figure 3. Branch reduction in WFG. (a) Control flow diverges at node B and converges back at node G. The dashed arc indicates that node C depends on the data loaded by node M. (b) Control flow paths are required to be disconnected from the rest of the WFG except through the nodes at which divergence starts or ends. To restore this property, some of the nodes before or after the diverging control flow are duplicated in both paths. (c) Each path is reduced independently. (d) $\mathrm{W_T}$ and $\mathrm{W_F}$ are computed latencies for each path. (e) With ρ_T and ρ_F being the execution weights of each path, the average latency from M to G is computed to be $\frac{\rho_\mathrm{T}\mathrm{W_T}+\rho_\mathrm{F}\mathrm{W_F}}{\rho_\mathrm{T}+\rho_\mathrm{F}}$.

pipeline latency, etc. These parameters are either provided to the model through a spec-file or collected by running a set of micro-benchmarks. The resulting WFG fuses the GPU kernel code signature to the hardware configuration. After specializing the WFG for the hardware, the weight assigned to each transition arc indicates the number of cycles that are required on average to execute the instruction(s) at the source node.

The WFG represents an average warp in a GPU kernel. In case of a branch divergence, different threads may follow different control flow paths. Therefore, proper weight should be assigned to the diverging control flow paths to indicate the fraction of warps that execute each path. Similarly, if there are long-latency or low-

latency memory accesses along these diverging paths, the number of memory transactions or bank conflict serialization delays should be adjusted according to the portion of warps and pattern of threads that issue the memory instructions. In Section 3 we discuss how symbolic evaluation can be used to augment the WFG with adjusted weights for each diverging path, the adjusted number of memory transactions required to be issued for global memory accesses, and shared memory access delays, based on hardware specifications such as SIMD work granularity, memory coalescing rules and memory bank configuration.

We mentioned earlier that the execution latency exposed by each instruction determines the total number of execution cycles. The amount of exposed latency is itself a function of hardware parameters and the level of parallelism available in the code. To estimate the execution latency of a kernel, we investigate the exposed SIMD pipeline latency first and later account for the untolerated global memory latency.

2.1.1 SIMD Pipeline Latency

The degree of resilience of a kernel at each phase of computation to the pipeline latency is determined by the observed amount of WLP, intra-thread ILP, and the ratio of SIMD work granularity (size of a warp) to the SIMD engine width; special function instructions are modeled based on the pipeline latency of their functional unit and the corresponding SIMD width. Based on the latency of the SIMD pipeline, the $\mathrm{WLP_{effect}}$ and the $\mathrm{ILP_{effect}}$, the latency of a simple computational instruction is computed by Eq. (3). Notice that $\mathrm{Latency_{comp}}$ is at least equal to one clock cycle (due to the issue latency) and may change from one segment of code to another.

$$\mathrm{Latency_{comp}} = \max\left(1, \frac{\mathrm{Latency_{pipeline}}}{\frac{\mathrm{SIMD_{work}}}{\mathrm{SIMD_{engine}}} \times \mathrm{ILP_{effect}} \times \mathrm{WLP_{effect}}}\right) \quad (3)$$

Accordingly, the execution latency of a computational instruction in the granularity of a warp is equal to $\mathrm{Latency_{comp}} \times \frac{\mathrm{SIMD_{work}}}{\mathrm{SIMD_{engine}}}$. Consequently, the weight of an outgoing transition arc from a WFG computation node with n instructions, $\mathrm{C_n}$, is set to be $\mathrm{n} \times \mathrm{Latency_{comp}} \times \frac{\mathrm{SIMD_{work}}}{\mathrm{SIMD_{engine}}}$. Weights on transition arcs from a low-latency memory operation node is computed through symbolic evaluation. Barrier synchronization nodes have fixed execution issue latency of $\frac{\mathrm{SIMD_{work}}}{\mathrm{SIMD_{engine}}}$. We also temporarily set the weights on outgoing arcs from global memory operations equal to the instruction issue latency of $\frac{\mathrm{SIMD_{work}}}{\mathrm{SIMD_{engine}}}$.

Figures 2 and 3 show a high-level overview of a system of transformation rules on the WFG that recursively collapses loops and branches into a pair of nodes with a single arc in between. Having the WFG populated with proper weights on transition arcs, we use these transformation rules to collapse the WFG into a pair of entry and exit nodes and a transition arc from entry to exit. The weight associated with this transition arc reflects the average number of compute cycles (latency) of a warp ($\mathrm{CYC_{compute}}$). When estimating $\mathrm{CYC_{compute}}$, the data-dependence arcs in the WFG are ignored during the reduction process. Parts (b) and (c) in Fig. 1 show the the initial weight assignments and the reduced WFG for the single statement example that we discussed earlier.

2.1.2 Global Memory Latency

We classify memory stalls into those due to limited memory bandwidth and those purely due to latency. We first measure the stalls due to lack of available bandwidth ($\mathrm{Latency_{BW}}$) and later show how an abundance of WLP and ILP helps tolerate the latter stalls.

In the process of reducing the WFG to compute $\mathrm{CYC_{compute}}$, we also collect the average number of global memory operations ($\mathrm{NUM_{mem}}$), the total amount of data that are required to be transfered to or from global memory ($\mathrm{NUM_{bytes}}$), and the number of barrier synchronization points ($\mathrm{NUM_{sync}}$) in a warp. Part of this information is calculated through symbolic evaluation (Sections 3.1 and 3.3) and is added to the initial WFG. Based on the memory bandwidth allocated for each streaming multiprocessor and $\mathrm{NUM_{bytes}}$, we compute the average number of cycles that are required to transfer all the data to or from global memory ($\mathrm{CYC_{mem}}$). The latency for a kernel's global memory operations is calculated based on the difference between $\mathrm{CYC_{mem}}$ and $\mathrm{CYC_{compute}}$ as shown in Eq. (4). The ratio $\frac{\mathrm{CYC_{compute}}}{\mathrm{CYC_{mem}}}$ can be interpreted as the compute intensity of an average warp. When $\mathrm{CYC_{mem}}$ is less than $\mathrm{CYC_{compute}}$, memory bandwidth is not the critical limiting factor if execution of different warps is interleaved. Otherwise, the latency is adjusted accordingly to compensate for the number of memory cycles that are not covered by the compute cycles of the kernel. The term $\frac{\mathrm{SIMD_{work}}}{\mathrm{SIMD_{engine}}}$ in Eq. (4) is associated with the cycles required to issue the memory instructions of a warp.

$$\mathrm{Latency_{BW}} = \max\left(0, \frac{\mathrm{CYC_{mem}} - \mathrm{CYC_{compute}}}{\mathrm{Num_{mem}}}\right) + \frac{\mathrm{SIMD_{work}}}{\mathrm{SIMD_{engine}}} \quad (4)$$

We combine the bandwidth-related stalls into the WFG by setting the weight of all outgoing transition arcs from global memory operations equal to $\mathrm{Latency_{BW}}$.

To accommodate the stalls purely due to memory latency, we introduce the average number of non-blocking cycles for a warp, $\mathrm{NBC_{avg}}$, which is computed according to Eq. (5). In Eq. (5), $\mathrm{NUM_{mem}} + \mathrm{NUM_{sync}}$ is an approximation of the number of context switches that occur during a warp execution time.

$$\mathrm{NBC_{avg}} = \frac{\mathrm{CYC_{compute}}}{\mathrm{NUM_{mem}} + \mathrm{NUM_{sync}} + 1} \quad (5)$$

On average each of the $\mathrm{WLP_{effect}}$ active warps can use $\mathrm{NBC_{avg}}$ cycles before execution returns to a warp that has issued a long latency memory load. Now, let $\mathrm{Latency_{mem}}$ be the global memory load latency for the GPU. We set the weight on each data-dependence arc in the WFG equal to $\mathrm{Latency_{exposed}}$, which is the portion of the GPU load latency that is not covered by interleaving execution of different warps, and is computed as shown in Eq. (6).

$$\mathrm{Latency_{exposed}} = \mathrm{Latency_{mem}} - (\mathrm{WLP_{effect}} - 1) \times \mathrm{NBC_{avg}} \quad (6)$$

Latency-based memory stalls can also be implicitly covered by ILP through reduction of data-dependence arcs. Notice that there is also a control flow path from the source of a data-dependence arc (a memory load) to its destination (the use of the memory load). Cumulative weights of all transition arcs that are on this path equals the number of intra-warp cycles that can be interleaved with the exposed memory latency. If the weight on a data-dependence arc is greater than the cumulative latencies of transition arcs from the memory node to its use, the difference is added to the weight on the last transition arc of the path. Figures 4(a) and (b) illustrate the process through which the data-dependence arc is collapsed. Parts (c), (d), and (e) of Fig. 4 show a similar reduction process in the case of interleaving memory loads. Notice that at the time of reducing a data-dependence arc, all nodes between the memory load and its use are laid on a sequential path; the subgraph has already been flattened through application of the transformation rules shown in Figs. 2 and 3.

As weights on transition arcs are adjusted to accommodate long memory latencies, we reduce the WFG once again. This time the weight on the transition arc from the entry to the exit node reflects the total latency of an average warp. Parts (d), (e) and (f) of Fig. 1 describe the process of incorporating the global memory latency into the WFG for the single statement example. Given the average warp latency, estimating the total execution time of a kernel is straightforward.

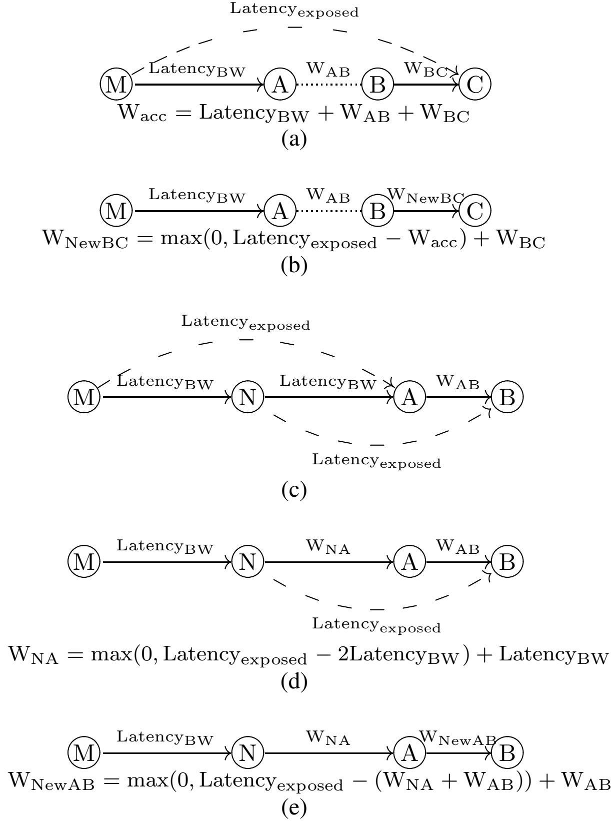


Figure 4. Reducing data-dependence arcs. (a) A sequential WFG subgraph with the long-latency data-dependence arc dashed. (b) Data-dependence arc removed after the untolerated memory latency is incorporated into $\rm W_{NewBC}$. (c) Two interleaved long latency memory operations. (d) Data-dependence arcs are removed in the order that their corresponding uses appear in WFG. (e) $\rm W_{NA}$ that includes the untolerated memory latency of M is incorporated in covering the memory latency for N; the overlapped latency is counted once toward the warp execution latency.

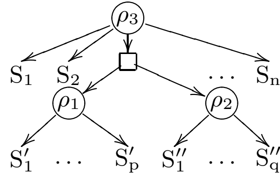


Figure 5. Program dependence graph

2.2 Extracting the Model

In the previous section we explained that through the WFG we combine effects of different performance factors. In this section we present our approach to approximate some of the parameters that were discussed in Section 2 such as $\rm WLP_{avg}$, $\rm WLP_{local}$, $\rm ILP_{local}$, and $\rm ILP_{avg}$. We also describe how the initial WFG is constructed.

Our compiler front-end analyzes the kernel source code and translates it into a program dependence graph (PDG) representation. The PDG provides a coherent framework to explicitly represent control and data dependences in the program [7]. We perform traditional scalar analysis such as induction variable detection and forward substitution based on an SSA[4] form on the PDG. As a result, program expressions are represented by symbolic expressions in terms of thread coordinates (thread ID and block ID), induction variables, and symbolic constants (input parameters). Our framework currently provides closed form expressions for linear and geometric induction variables. We also ignore the redundant computation that is eventually eliminated by the back-end optimizing compiler through value numbering [13].

Figure 5 shows an example of a PDG. Nodes in the PDG represent either a statement in the program ($\rm S_1, S'_1, S''_1$, etc), a predicate node that controls execution of a set of instructions (the square node in Fig. 5), or a region node that summarizes the set of control conditions for a node or a set of nodes. Region nodes, shown with circles in Fig. 5, also summarize information such as the average execution weight of their descendent nodes. For example, instructions in a loop should be weighted proportional to the trip count of the loop. Therefore, the region that contains the body of a loop is augmented with the trip count of the loop. If the trip count or the execution weight are not known at compile time, we generate a parametric performance number, which can be specialized later based on the input parameters. In case of a branch divergence, proper weight should be assigned to the two descendent region nodes of the predicate that controls the divergence; the weight assigned to the *true* descendent region of a predicate node indicates what fraction of the warps that reach the condition evaluate it to true and therefore execute the instructions under the true region.

Through a preorder walk on the PDG, we symbolically evaluate conditions and memory access expressions through efficient techniques discussed in Section 3. Knowing which threads are active during a computation step, we determine the number of active warps and the pattern of memory accesses for each step. Based on this information we compute the average execution weight for a region, the average bank conflict penalty for shared memory accesses, and the average number of bytes transferred to or from global memory for each long latency memory operation.

As we walk down the PDG, we also compute $\rm WLP_{local}$ and $\rm ILP_{local}$ for each region. Local WLP for the three regions in Fig. 5 is computed as $\rho_3 \rm WLP_{max}$, $\rho_3\rho_1 \rm WLP_{max}$, and $\rho_3\rho_2 \rm WLP_{max}$. To compute the local ILP for a regions with no predicate child, we divide the total number of instructions of that region by the length of the longest def-use chain built from them. For a region with diverging control flow, to compute $\rm ILP_{local}$ we only consider the immediate instruction children.

Next, we approximate the values for $\rm WLP_{avg}$ and $\rm ILP_{avg}$ by performing a postorder walk on the PDG. For each region, $\rm WLP_{avg}$ is approximated by getting a weighted average of the local WLP of all the regions it contains, including itself; regions that contain more instructions contribute more to the average. For example, $\rm WLP_{avg}$ for the top PDG region in Fig. 5 is computed by Eq. (7).

$$\rm WLP_{avg} = \frac{p\rho_3\rho_1 WLP_{max} + q\rho_3\rho_2 WLP_{max} + n\rho_3 WLP_{max}}{p + q + n} \quad (7)$$

The process of approximating $\rm ILP_{avg}$ is similar to that of $\rm WLP_{avg}$. For example, the $\rm ILP_{avg}$ for the top region of Fig. 5 is approximated according to Eq. (8), where $\rm ILP_p$, $\rm ILP_q$, and $\rm ILP_n$ are local ILP values for each of the three PDG regions in Fig. 5.

$$\rm ILP_{avg} = \frac{p\rho_1 ILP_p + q\rho_2 ILP_q + nILP_n}{p\rho_1 + q\rho_2 + n} \quad (8)$$

We construct the initial WFG recursively through another preorder walk of the PDG that is augmented with results of symbolic evaluation and other dataflow analyses.

3. Symbolic Evaluation

In this section, we propose a tractable approach to symbolic evaluation of conditions and array access expressions. Our symbolic evaluation system has two components:

1. A symbolic execution engine for evaluating effects of expressions that are simple enough.
2. A set of simplification rules for simplifying more complex expressions.

If the expression to be evaluated is too complicated for the symbolic evaluation engine, the expression is conditionally simplified such that the evaluation of the simplified expression implies results equal to that of the original expression. The simplification process may be applied recursively until tractable expressions are obtained or the simplification engine decides that no further simplification is applicable. In the latter case, the expression is interpreted conservatively, which results in a lower bound for the performance.

During the simplification process we may impose certain constraints on free variables, e.g., input parameters. The validity of the results is contingent on the truth of the conjectures made about free variables by the simplification system. Therefore, the results of each evaluation step are accompanied by a set of constraints that specify the domain of applicability of each evaluation step.

The average evaluation time for a kernel is in the order of a few milliseconds and below the average kernel execution time, even when we ignore the data transfer time from CPU to GPU.

3.1 Structural Conditions

A predicate denotes either a structural condition or a data-dependent condition. A structural conditional expression is a function of thread coordinates, enclosing loop induction variables and symbolic constants of the kernel. In this section we show how to analyze a large class of structural conditions that are defined by Definition (9), through symbolic evaluation. Analyzing data-dependent conditions requires statistical knowledge of the data and is discussed in Section 4.3.

$$\begin{aligned}&\mathcal{O}(\mathrm{At+B},\mathcal{I}(\mathrm{i})),\ \ \mathcal{I}(\mathrm{i})=\alpha\mathrm{i}+\beta\ \text{ or }\ \mathcal{I}(\mathrm{i})=2^{\alpha\mathrm{i}+\beta}\\&\mathrm{i}\in\{0,1,...,\mathrm{I}\},\ \ \mathrm{t}\in\{0,1,...,\mathrm{T}-1\}\ \text{ and }\ \mathrm{A,B},\alpha,\beta\in\mathbb{Z}\\&\mathcal{O}(\mathrm{x,y})\in\{\mathrm{x=y,x\neq y,x\leq y,x\geq y,x}\pmod{\mathrm{y}}\equiv 0\}\end{aligned}\tag{9}$$

For brevity, we consider one-dimensional thread-blocks of the size T, where T is well-bounded due to hardware limitations. Variable t stands for the thread ID in the affine expression $\mathrm{At+B}$. In Definition (9), i is the loop induction variable with the upper bound of I, which may not be known at compile time. Operator $\mathcal{O}$ can be either a comparison or a modulo operator. Without loss of generality, we restricted our discussions to $\leq$ for comparison operators, $\mathrm{A},\ \mathrm{B},\ \alpha,\ \beta\in\mathbb{N}$, and $\mathcal{I}(\mathrm{i})=\alpha\mathrm{i}+\beta$. Conclusions on other cases can be derived similarly. The following two cases apply, based on the type of operator $\mathcal{O}$.

1. $\mathcal{O}$ is a comparison operator: $\mathrm{At+B}\leq\alpha\mathrm{i}+\beta$
   We solve $\mathrm{At+B}$ for each $\mathrm{t}\in\{0,1,...,\mathrm{T}-1\}$. Let $\mathrm{At'+B}\leq\alpha\mathrm{i}+\beta$ for $0\leq\mathrm{t'}<\mathrm{T}$; we have $\frac{\mathrm{At'+B}-\beta}{\alpha}\leq\mathrm{i}\leq\mathrm{I}$. Consequently, the total number of steps for symbolic evaluation is given by:

   $$\sum_{\mathrm{t}=0}^{\mathrm{T}-1}\mathrm{I}-\left\lceil\frac{\mathrm{At+B}-\beta}{\alpha}\right\rceil+1\approx\frac{2\mathrm{T}(\alpha\mathrm{I}-\mathrm{B}+\beta+\alpha)-\mathrm{AT}(\mathrm{T}-1)}{2\alpha}\tag{10}$$

   Notice that for $\mathrm{I_0}\leq\mathrm{i}$, where $\mathrm{I_0}=\frac{\mathrm{A(T-1)+B}-\beta}{\alpha}$, all threads satisfy the inequality. In other words, control flow divergence does not occur for computation steps greater than $\mathrm{I_0}$. If we replace I with $\mathrm{I_0}$ in Eq. (10) the upper-bound for the total number of evaluation steps becomes:

   $$\frac{\mathrm{AT}^2-\mathrm{AT}+2\alpha\mathrm{T}}{2\alpha}\tag{11}$$

   Notice that we have iterated over the thread IDs. To measure the thread ID dependent effects correctly, we replay the active threads for each computation step i and measure the control flow divergence, coalescing and bank conflict effects. As a result, the total cost is twice the value of Expression (11). Notice that if $\mathrm{I_0}\leq\mathrm{T}$, we directly iterate over i. The cost in Expression (11) is formulated in terms of program constants or known hardware limits and is proportional to cost of symbolically evaluating the condition. We compute the cost in advance. If the computed cost is less than a predefined threshold, we symbolically evaluate the condition. Otherwise, the condition is assumed to be true for all threads.

   Now, let $\mathrm{w_0}$ be the computed execution weight for the region under the control of the above predicate. To make up for the divergence-free part (the $\mathrm{I}-\mathrm{I_0}$ iterations), the overall weight is computed as:

   $$\frac{\max(\mathrm{I}-\mathrm{I_0},0)+\mathrm{w_0I_0}}{\max(\mathrm{I_0},\mathrm{I})}\tag{12}$$

   If I is known at compile time, Expression (12) is reduced to a single value, otherwise it remains parametric on I and consequently the total execution cycles become a function of I. Note that in cases where I is not known statically and its actual value is less than $\mathrm{I_0}$, we have approximated the corresponding execution weight by $\mathrm{w_0}$.

2. $\mathcal{O}$ is the modulo operator: $(\mathrm{At+B})\equiv 0\pmod{\alpha\mathrm{i}+\beta}$
   We solve $\mathrm{At+B}$ for each $\mathrm{t}\in\{0,1,...,\mathrm{T}-1\}$. Let $(\mathrm{At'+B})\equiv 0\pmod{\alpha\mathrm{i}+\beta}$ which implies that $\alpha\mathrm{i}+\beta$ is some combination of prime factors of $\mathrm{At'+B}$. If $\mathrm{AT+B}$ is not a very large number we can store combinations of prime factors of all numbers less that $\mathrm{AT+B}$ in a lookup table in order to retrieve them efficiently. Let P be a combination of prime factors of $\mathrm{At'+B}$. Factor P is equal to $\alpha\mathrm{i}+\beta$ if $\frac{\mathrm{P}-\beta}{\alpha}$ is an integer and $0\leq\frac{\mathrm{P}-\beta}{\alpha}\leq\mathrm{I}$; the converse is also true. Therefore, $\mathrm{t'}$ is a candidate thread that satisfies the condition. The total number of steps to test all values of t is bounded by:

   $$\sum_{\mathrm{t}=0}^{\mathrm{T}-1}\left\lfloor\frac{\mathrm{At+B}}{2}\right\rfloor\approx\frac{\mathrm{AT}(\mathrm{T}-1)}{4}+\frac{\mathrm{BT}}{2}$$

   Similar to case 1, this method requires two passes to collect the performance information.

   Since the largest prime factor that is examined is equal to $\mathrm{A}(\mathrm{T}-1)+\mathrm{B}$, the condition evaluates to true for all threads if $\mathrm{I_0}\leq\mathrm{i}$, where $\mathrm{I_0}=\frac{\mathrm{A(T-1)+B}-\beta}{\alpha}$. Similar to the discussion given for the previous case, the overall execution weight is computed as stated in Expression (12).

3.2 Structural Memory Accesses

In this section we first describe the class of memory access expressions that is accepted by our expression simplification engine. Later we explain how a candidate expression is simplified through an example. Finally, we discuss our approach to efficiently capture qualitative properties of these expressions from the point of view of GPU architecture performance.

The subscripting function of an array reference, $\mathcal{F}$, is a multivariate expression. Function $\mathcal{F}$ can be defined recursively as the sum of possibly complex terms according to Definitions (13) and (14), where $\mathbf{v}$ is a vector of integer variables of the program (thread

coordinates, input parameters, enclosing loops induction variables, and an element representing integer number 1) all of which are involved in the evaluation of the subscripting expression, and $\mathbf{v}[\ell]$ is the ℓth element of the variable vector $\mathbf{v}$.

$$\mathcal{F}^{(0)}(\mathbf{v}) = \mathbf{v}[\ell] \;\text{ for some } \ell \in \{\,1, 2, \cdots, |\mathbf{v}|\} \tag{13}$$
$$\mathcal{F}^{(n)}(\mathbf{v}) = \sum_{i=1}^{N^{(n)}} \prod_{j=1}^{M_i^{(n)}} P_{i,j}$$
$$P_{i,j} \in \{\mathcal{O}_k \cdot (\mathcal{F}_{i,j}^{(n-1)})(\mathbf{v})\} \;\text{ for }\; k = 1,\ 2,\ 3,\ 4$$

For some $\alpha, \beta \in \mathbb{Z}$ and some $\ell \in \{\,1, 2, \cdots, |\mathbf{v}|\}$, we define the following four operators, namely $\mathcal{O}_1, \cdots, \mathcal{O}_4$, on the subscripting function $\mathcal{F}$. Note that negative exponent expressions are excluded by definition.

$$\mathcal{O}_1 \cdot (\mathcal{F}_{i,j}^{(n-1)})(\mathbf{v}) = \mathcal{F}_{i,j}^{(n-1)}(\mathbf{v}) \bmod 2^{\alpha \mathbf{v}[\ell]+\beta} \tag{14}$$
$$\mathcal{O}_2 \cdot (\mathcal{F}_{i,j}^{(n-1)})(\mathbf{v}) = \lfloor \frac{\mathcal{F}_{i,j}^{(n-1)}(\mathbf{v})}{2^{\alpha \mathbf{v}[\ell]+\beta}} \rfloor$$
$$\mathcal{O}_3 \cdot (\mathcal{F}_{i,j}^{(n-1)})(\mathbf{v}) = 2^{\alpha \mathbf{v}[\ell]+\beta} \times \mathcal{F}_{i,j}^{(n-1)}(\mathbf{v})$$
$$\mathcal{O}_4 \cdot (\mathcal{F}_{i,j}^{(n-1)})(\mathbf{v}) = \alpha \times \mathcal{F}_{i,j}^{(n-1)}(\mathbf{v})$$

The operators defined above are commonly used to express complex access patterns, e.g., wrap-around and strided accesses.

Example 1. Expression (15) shows a relatively complex index expression from the FFT kernel. Variables involved in this expression include induction variables of two enclosing loops (i and 2^{j}), input parameter (N), and thread coordinates (b as thread-block ID and t as thread ID).

$$\lfloor \frac{b \times N + t}{2^j} \rfloor \times 2^{j+1} + (b \times N + t) \bmod 2^j + i \times 2^j \tag{15}$$

The following derivation shows how the first term in Expression (15) satisfies the definition of $\mathcal{F}$, where the vector of involved variables is initialized as $\mathbf{v} = [b,\ t,\ i,\ j,\ N,\ 1]$:

$$\mathcal{F}^{(2)}(\mathbf{v}) = \lfloor \frac{\mathcal{F}_{1,1}^{(1)}(\mathbf{v})}{2^j} \rfloor \times 2^{j+1} \times \mathcal{F}_{1,2}^{(1)}(\mathbf{v}) \tag{16}$$
$$= \lfloor \frac{(\mathcal{F}_{1,1})_{1,1}^{(0)}(\mathbf{v}) \times (\mathcal{F}_{1,1})_{1,2}^{(0)}(\mathbf{v}) \; + \; (\mathcal{F}_{1,1})_{2,1}^{(0)}(\mathbf{v})}{2^j} \rfloor$$
$$\times\, 2^{j+1} \times (\mathcal{F}_{1,2})_{1,1}^{(0)}(\mathbf{v})$$
$$= \lfloor \frac{b \times N \; + \; t}{2^j} \rfloor \times 2^{j+1} \times 1$$

In step two of the derivation above, it is understood that for example, the term $(\mathcal{F}_{1,1})_{1,2}$ refers to the second factor of the first term in $\mathcal{F}_{1,1}$.

Expression (15) is examined to determine whether the qualitative behavior of residing threads in a memory vector instruction (a half-warp in the case of NVIDIA GPUs) can be confined to their local thread coordinates. Figure 6(a) shows the expression tree for Expression (15). We refer to the multivariate expressions that are composed of thread ID and a combination of other thread coordinates (thread-block ID) with free variables as *complex* expressions. Through a bottom-up process we label the complex subexpressions. Labels are propagated upward in the expression tree until a non-distributive operator such as modulo or integer division is reached (referred to as a *tainted* operator). At this point, the simplification engine attempts to recursively disentangle the local

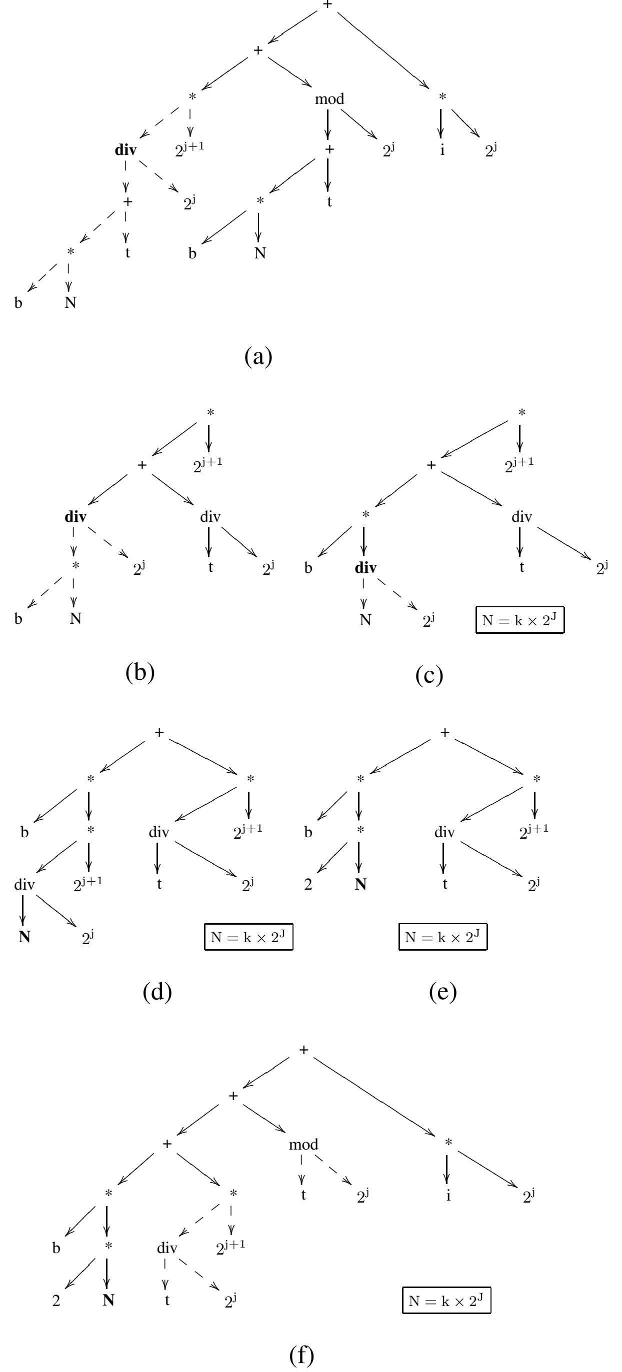


Figure 6. Expression simplification for Example 1. (a) The initial expression tree. (b) Distributing integer division over the add operator. (c) Distribution is carried on along the path that contains free variables; constraint $\mathrm{N} = \mathrm{k} \times 2^{\mathrm{J}}$ is bound to free variable N. (d) Multiplication is distributed over the add operator. (e) Multiplication and integer division cancel each other out. (f) The simplified expression tree.

thread ID part from the rest of the terms by imposing constraints on free variables.

We explain this simplification process for the first term of Expression (15) which is highlighted in the expression tree of Fig. 6(a) by dashed arcs. The first non-distributive tainted operator node that

is visited during the bottom-up process for this sub-tree is the div operator ($\lfloor \frac{}{2^j} \rfloor$). Operator div can be distributed if at least one of the terms in its enumerator is a multiple of the denominator 2^j. Tainted operator div is distributed speculatively over $+$ operator. From the two resulting sub-expression trees in Fig. 6(b), only the one on the left contains a free variable. Therefore, speculation continues down the left sub-tree and along the path that leads to the free variable N, as shown in Fig. 6(c). At this point, the constraint $\mathrm{N} = \mathrm{k} \times 2^{\mathrm{J}}$ (for some $\mathrm{k} \in \mathbb{Z}$) is added to the simplified expression tree of Fig. 6(c), where J is the upper bound for the induction variable j. The simplification steps are valid only if the constraint on N is proved to be true. The proof can be either derived at compile time or checked at runtime. Figure 6(d) shows the routine distribution of multiplication over addition. In Fig. 6(e), based on the constraint that we put on N earlier, multiplication and integer division have equaled out each other. As the result of these simplifications, the original expression tree is transformed to the one shown in Fig. 6(f). The local thread coordinate t is now separated from other thread coordinates such as b. The two highlighted sub-trees in Fig. 6(f) are now simple enough to be symbolically evaluated.

3.3 Memory Bank Conflict and Coalescing

In general, deriving the exact memory access pattern is infeasible, especially with the presence of relatively complex subscripting functions (Section 3.2) and unknown upper bounds for the induction variables. However, we are interested in certain qualitative properties of the array subscript expressions. For example, for efficient utilization of memory bandwidth, successive words are assigned to successive memory banks. Let $\mathrm{NUM_{bank}}$ be the number of memory banks, which is also equal to the number of threads that can simultaneously access memory. The problem of determining the pattern of memory bank conflicts can be transformed to the problem of computing $\mathrm{x} \bmod \mathrm{NUM_{bank}}$, where x belongs to a finite set of integers. Size of this set is proportional to the number of variables present in the subscripting expression. Let $\mathbf{v} = [\mathrm{v_1}, \mathrm{v_2}, \cdots, \mathrm{v_n}]$ be all n participating variables in evaluation of expression $\mathcal{E}(\mathbf{v})$. Each $\mathrm{v_i}$ can be formulated as $\alpha_\mathrm{i}\mathrm{NUM_{bank}} + \rho_\mathrm{i}$, where $0 \leq \rho_\mathrm{i} < \mathrm{NUM_{bank}}$. Evaluating $\mathcal{E}(\mathbf{v})$ to capture memory bank conflict patterns is equivalent to evaluation of $\mathcal{E}(\rho)$, where $\rho = [\rho_1, \rho_2, \cdots, \rho_\mathrm{n}]$. The difference is that members of variable vector ρ are well-bounded.

The simplification engine is more stringent on subscripting expressions that are evaluated for memory coalescing. Not only should the terms involving local coordinates (thread ID) be separable from the terms that involve other thread coordinates and free variables, but each thread ID dependent term should be expressable as a sum of terms such as $\mathrm{At} + \mathrm{B}$, $\lfloor \frac{\mathrm{At}+\mathrm{B}}{2^{\alpha\mathrm{i}+\beta}} \rfloor$ or $(\mathrm{At} + \mathrm{B}) \bmod 2^{\alpha\mathrm{i}+\beta}$, where t is the local thread coordinate, $\mathrm{A}, \mathrm{B} \in \mathbb{Z}$ and $\alpha, \beta \in \mathbb{I}$. As a result, the symbolic evaluation engine can identify through a limited number of steps, how many different memory segments are accessed by threads within a single memory vector instruction.

4. Evaluation

In this section we show how well the execution times predicted by the proposed performance model comply with the actual measured times. In cases where the estimated execution cycles were a function of the input parameters, we evaluated the function for a given set of input parameters for clarity of the graphs. We use the NVIDIA GeForce 8800 GPU for our experiments.

4.1 Memory Bandwidth and Latency

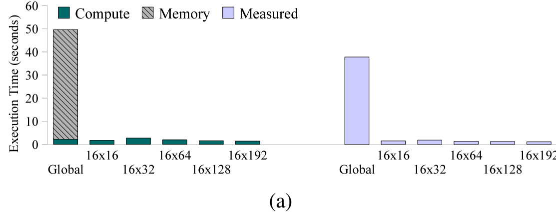
To evaluate the effectiveness of our model in capturing the effects of GPU global memory bandwidth and latency on performance, we apply it to dense matrix multiplication and FFT kernels.

We first discuss several versions of dense matrix multiplication which represents many tiled algorithms. A simple version of matrix multiplication that loads the same array input elements multiple times by different threads (`Global`), will saturate the global memory bandwidth of the GPU. Another pitfall with this version is that global memory accesses to one of the arrays are not coalesced. In the GeForce 8800, global memory delivers the 86.4 GB/s memory bandwidth only when the global memory accesses are coalesced within a half-warp (16 threads). The GeForce 8800 can fetch data in a single 64-byte or 128-byte transaction [17]. If the memory accesses cannot be coalesced, then a separate memory transaction will be issued for each thread in the half-warp. As a result, for accessing one word of the array with non-coalesced accesses the GeForce 8800 issues a 16-word global memory transaction. This results in an increase in the average number of memory cycles, $\mathrm{CYC_{mem}}$, which consequently increases the total warp latency.

A tiled version of the the kernel takes advantage of shared memory to enhance data sharing between threads computing nearby results. We chose tile sizes of 16×16, 16×32, 16×64, 16×128, and 16×192 elements to be executed by a thread-block. During execution, threads work within two input tiles that stride across 16 contiguous rows and 16, 32, 64, 128, or 192 columns of input matrices. For the tiled kernels, global memory loads are reduced by a factor of 16, 32, 64, 128, or 192 respectively. Consequently, the measured performance improves as the tile size increases. Figures 7(a) and 7(b) show the predicted and measured performance numbers for matrix multiply kernels next to each other. Based on these results, the model perfectly captures the effect of data reuse and coalescing on the overall performance. NVIDIA's compiler unrolls the loop automatically for the tile size of 16×16; when estimating the performance for this version, we modeled the unrolled code.

The second benchmark we use is a power-of-two batched Fast Fourier Transform (FFT), which is based on the Stockham formulation. We used the pseudo-codes provided by Govindaraju et al. [9] to implement the FFT kernels. The first set of kernels load the data from global memory, compute an R-point FFT, and write the results back to global memory. The kernels are invoked several times and during each iteration of the outer loop, values from R different FFTs are combined together to generate a larger size FFT. For the first few invocations of these kernels, global memory writes cannot be coalesced, which decreases the performance. This is the same subscripting expression that we discussed in Section 3. Through symbolic evaluation we can determine the fraction of noncoalesced accesses and adjust $\mathrm{CYC_{mem}}$ accordingly.

The second set of the FFT kernels improves the data reuse by keeping intermediate results in shared memory. They also write the results in proper order to global memory to avoid poor global memory coalescing. For this experiment, each set of kernels includes different radix sizes of 2, 4, and 16. Larger radices reduce the total number of iterations required to combine the results of smaller size FFTs. Meanwhile, larger radix sizes also consume more GPU resources. For example, both global and shared memory versions for radix-16 increase the use of registers substantially. Consequently local arrays are spilled to global memory, resulting in an increase in global memory traffic. Spilling local arrays into global memory also increases the number of stall points in the kernel and the average non-blocking cycles, $\mathrm{NBC_{avg}}$, is reduced accordingly. The number of active warps, $\mathrm{WLP_{max}}$, is also reduced as each thread uses more registers. Reduction in both $\mathrm{NBC_{avg}}$ and $\mathrm{WLP_{max}}$ makes global memory latency blatant for radix-16 kernels based on Eq. (6). These effects are reflected in both predicted and measured performance numbers in Fig. 8.



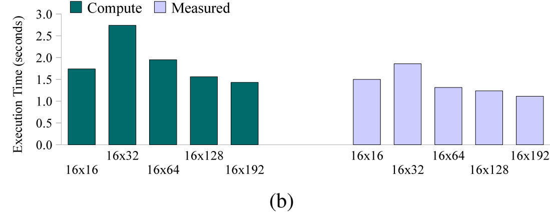


Figure 7. Matrix multiply kernels. (a) Initial and tiled kernels: a breakdown of the predicted time for the global version shows the portion of memory stalls versus compute time. (b) Zoomed for tiled kernels.

4.2 SIMD Divergence and Bank Conflicts

The third benchmark that we use is the prefix sum scan kernel, which computes partial sums of all prefixes of an array. Unlike the other two benchmarks, computation is not distributed uniformly to threads. In mapping the algorithm to parallel threads, the tree-like structure of the computation has been divided into discrete steps containing different amounts of computation separated by barrier synchronizations. Threads determine what to do by computing branch conditions and array indices from their thread ID. How computation is assigned to threads and the resulting branch behavior affects performance.

We start with a simple version of the scan kernel (`Init`), where thread t is responsible for the computation that produces array element $2t + 1$. For a thread-block size of T, the kernel loads data from global memory, computes partial sums in $\log T + 1$ steps, propagates the partial sums to all array elements in another round of $\log T + 1$ steps, and saves the results to global memory. Subsets of the threads in a block are turned off for each step of execution, but in this organization the active threads are distributed throughout all warps for most execution steps, leading to an increase in compute latency (toward useless work).

By reassigning computation to different threads, we can group threads that take the same control flow path into the same warp, thus eliminating branch divergence except during the steps where fewer than one warp of threads runs. The new kernel (`Div`) has a lower compute latency with regard to pure computational cycles, but as the regrouped threads issue more simultaneous accesses to the same shared memory bank, increased latency for shared memory accesses results in a small overall latency increase from the initial version for some thread-block sizes.

Next, the shared memory bank conflicts are removed (by padding shared memory arrays) for both of the above versions. The performance rises for the kernel with reduced branch divergence (`Div_B`). For the initial version performance degrades after the array layout is changed (`Init_B`) as the latency of conflicting shared memory bank accesses is less than the compute latency of extra address calculation instructions.

Figure 9 summarizes the predicted and measured performance numbers for different versions that we discussed above. For each kernel configuration we tried thread-block sizes of 64, 128 and 256. Half the threads become superfluous after each step of computation in the first half of the scan kernel, yet they still must consume execution cycles to participate in barrier synchronization. As a result, thread-blocks of larger sizes are expected to pay a slightly larger penalty for synchronization in these kernels. In addition, with larger thread-blocks fewer independent thread-blocks are active simultaneously during the last stages of tree-like computation; $\text{WLP}_{\text{effect}}$ drops below the level that is required to hide the pipeline latency. Consistently, Fig. 9 shows the rise in pipeline latency for the thread-block size of 256. The results in Fig. 9 verify the accuracy of the proposed performance model in capturing the effect of SIMD divergence and shared memory bank conflicts.

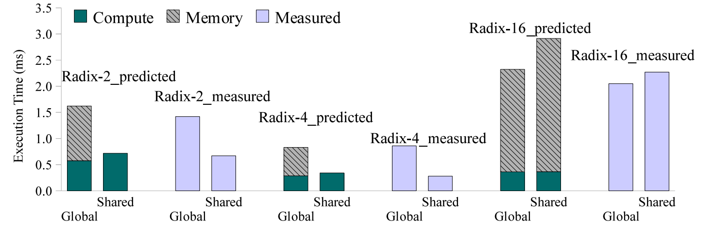


Figure 8. FFT kernels: predicted versus measured execution times. Predicted time is composed of memory stalls and compute time.

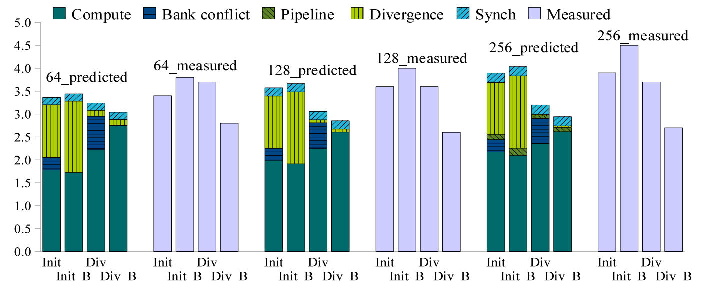


Figure 9. Prefix sum scan kernels: predicted versus measured execution times. A more detailed break down of predicted execution time is presented to show where the cycles are used in each implementation.

4.3 Data-Dependent Conditions and Memory Accesses

Finally, we applied our performance model to sparse matrix-vector multiplication kernels [2], known for indirect and irregular memory accesses. For this set of kernels, computation distribution and thread mapping can affect memory bandwidth and SIMD utilization.

We start with a version (`Global`) in which one thread is allocated to each row of the matrix. Based on the number of nonzero elements per row, this version can introduce extra compute latency into the computation, as not all SIMD cycles are used efficiently.

In another version of the kernel (`Shared`), the array that stores the starting position of each row (in the dense representation of the sparse matrix) is loaded into shared memory and is reused across a thread-block.

The third implementation (`Consecutive`) uses multiple threads (a half-warp size) to compute each row. Consecutive threads access consecutive nonzero elements to promote coalesced memory accesses and possibly reduce the effect of control flow divergence on compute latency of the kernel. Each thread computes the partial product for one nonzero element. A parallel sum reduction is used to compute the final result for each row. Memory accesses are coa-

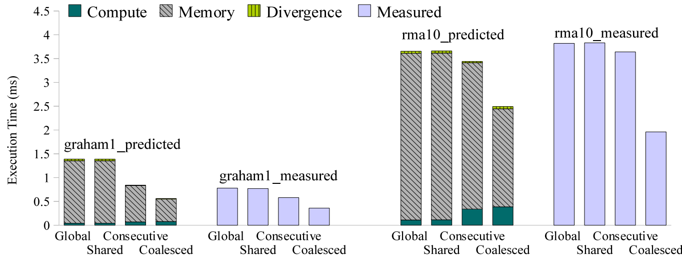


Figure 10. Sparse matrix-vector kernels: a breakdown of predicted execution time versus measured execution time.

lesced in this version if the first element of the array (which stores the starting position of each row) is well-aligned.

To guarantee coalesced accesses, in a new implementation (`Coalesced`), a few threads load the first few unaligned memory accesses before proceeding to accessing the aligned portion of the loads.

The warp latency computed for each of the above four kernels is parameterized over factors that are determined by structural properties of the input matrix, e.g., the trip count of the main loop for the computation of each row is determined by the number of nonzero elements of the row and the thread mapping scheme. As a results, we generate a light-weighted backward slice of the kernels that includes statements that their execution reveals partial structural properties of the matrix. Statistics were collected by running these micro-kernels to examine one warp on each GPU streaming multiprocessor. The instrumentation cost for each kernel, which is mainly dominated by the micro-kernel launch time, is in the order of a few microseconds. Based on the results collected from kernel instrumentation, we specialize the parametric warp latency for each input set.

Figure 10 shows the predicted and measured execution times for two sparse matrices from the University of Florida Sparse Matrix Collection [5]. Notice that in this experiment, we conservatively designate a portion of memory accesses as uncoalesced accesses. However, the model is capable of capturing the trends and relative merit of code versions, as shown in Figure 10.

5. Conclusions

We have presented a compiler-based approach to application performance modeling on GPU architectures. Our model is equipped with an efficient symbolic evaluation module to determine the effects of the structural conditions and complex memory access expressions on the performance of a GPU kernel. Our approach combines the effects of different performance factors into a coherent framework. In cases where it cannot statically determine performance information, a parametric latency is derived which can be customized later, according to the kernel inputs. In the case of data-dependent conditions or access patterns, it employs a light-weight dynamic instrumentation approach to specialize the parametric latency.

Our model allows a compiler to determine the relative merits of parallel kernel configurations without running all the variations. More importantly, the model identifies the bottlenecks and can guide the the compiler through the optimization process.

We validated our performance model for the matrix multiply, prefix sum scan, FFT, and sparse matrix-vector benchmarks. These benchmarks exhibit challenging conditional and memory access patterns. Our evaluation shows that there is good agreement between predicted and observed performance rankings for the various tuning versions of these kernels and that the model captures the effect of all major performance factors for GPU architecture.

Acknowledgments

We would like to acknowledge the support of the Gigascale Systems Research Center, funded under the Focus Center Research Program. Experiments were also made possible by NSF CNS grant 05-51665 and the CUDA Center of Excellence, sponsored by NVIDIA Corporation. This work was also supported in part by the NSF award 0837719. In addition, we wish to thank Mrs. Laurie Talkington for her help in editing this paper.

References

[1] ATI Stream Computing. http://developer.amd.com/gpu-assets/Stream-Computing-Overview.pdf.
[2] M. Baskaran and R. Bordawekar. Optimizing Sparse Matrix-Vector multiplication on GPUs. *IBM Research Report*, December 2008.
[3] M. Clement and M. Quinn. Analytical Performance Prediction on Multicomputers. In *ACM/IEEE Conference on Supercomputing*, November 1993.
[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, pages 451–490, 1991.
[5] T. Davis. University of Florida Sparse Matrix Collection. http://www.cise.u.edu/research/sparse/matrices/.
[6] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the Efficiency of GPU Algorithms for Matrix-matrix Multiplication. In *Conference on Graphics Hardware*, August 2004.
[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, pages 319–349, 1987.
[8] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A Memory Model for Scientific Algorithms on Graphics Processors. In *ACM/IEEE Conference on Supercomputing*, November 2006.
[9] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. In *ACM/IEEE Conference on Supercomputing*, November 2008.
[10] S. Hong and H. Kim. An model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *International Symposium on Computer Architecture*, June 2009.
[11] C. Jiang and M. Snir. Automatic Tuning Matrix Multiplication Performance on Graphics Hardware. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2005.
[12] W. Liu, W. Muller-Wittig, and B. Schmidt. Performance Predictions for General-Purpose Computation on GPUs. In *International Conference on Parallel Processing*, September 2007.
[13] M. Kongstad . *An Implementation of Global Value Numbering in the GNU Compiler Collection with Performance Measurements*, October 2004.
[14] G. Marin and J. Mellor-Crummey. Cross-architecture Performance Predictions for Scientific Applications Using Parameterized Models. In *International conference on Measurement and modeling of computer systems*, June 2004.
[15] A. Munshi. The OpenCL Specification. http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.
[16] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 2008.
[17] NVIDIA Corporation. *NVIDIA CUDA Programming Guide: Version 1.0*, June 2007.
[18] Z. Pan and R. Eigenmann. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *International Symposium on Code Generation and Optimization*, March 2006.
[19] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Program Optimization Space Pruning for a Multithreaded GPU. In *International Symposium on Code Generation and Optimization*, April 2008.
[20] D. Schaa and D. Kaeli. Exploring the Multi GPU Design Space. In *International Symposium on Parallel and Distributed Processing*, October 2009.
[21] D. B. Whalley. Tuning High Performance Kernels through Empirical Compilation. In *International Conference on Parallel Processing*, June 2005.