UC Davis
IDAV Publications

Title

A Quantitative Performance Analysis Model for GPU Architectures

Permalink

https://escholarship.org/uc/item/8gp0x7tc

Authors

Zhang, Yao
Owens, John D.

Publication Date

2011

DOI

10.1109/HPCA.2011.5749745

Peer reviewed

eScholarship.org

Powered by the California Digital Library
University of California

A Quantitative Performance Analysis Model for GPU Architectures

Yao Zhang and John D. Owens
Department of Electrical and Computer Engineering
University of California, Davis
{yaozhang, jowens}@ece.ucdavis.edu

Abstract

We develop a microbenchmark-based performance model for NVIDIA GeForce 200-series GPUs. Our model identifies GPU program bottlenecks and quantitatively analyzes performance, and thus allows programmers and architects to predict the benefits of potential program optimizations and architectural improvements. In particular, we use a microbenchmark-based approach to develop a throughput model for three major components of GPU execution time: the instruction pipeline, shared memory access, and global memory access. Because our model is based on the GPU's native instruction set, we can predict performance with a 5–15% error. To demonstrate the usefulness of the model, we analyze three representative real-world and already highly-optimized programs: dense matrix multiply, tridiagonal systems solver, and sparse matrix vector multiply. The model provides us detailed quantitative analysis on performance, allowing us to understand the configuration of the fastest dense matrix multiply implementation and to optimize the tridiagonal solver and sparse matrix vector multiply by 60% and 18% respectively. Furthermore, our model applied to analysis on these codes allows us to suggest architectural improvements on hardware resource allocation, avoiding bank conflicts, block scheduling, and memory transaction granularity.

1 Introduction

Since 2002, the massively parallel GPU has evolved from a graphics-specific accelerator to a general-purpose computing device. With the advent of C-based programming environments like CUDA [1] and OpenCL [2], an active research and development community has formed to develop high-performance general-purpose applications for GPUs [3]. However, compared to an enormous amount of efforts devoted to application development, little has been done on supporting tools for performance profiling and analysis. Commercial program profiling tools such as ATI Stream Profiler [4] and NVIDIA Parallel Nsight [5], along with academic GPU functional simulators [6, 7], are limited to providing program statistics only, but do not relate these statistics to program performance. Therefore, the hard work of identifying program bottlenecks and estimating the benefits of potential optimizations is done by programmers' paper-and-pencil analysis. As well, GPU architects need to evaluate their architecture designs against real-world applications to help guide architectural improvements.

In this paper, we describe a workflow that analyzes GPU program performance in a quantitative way, and thus allows programmers and architects to identify the performance bottlenecks and their causes, and predict the benefits of potential program optimizations and architectural improvements. In particular, we make the following contributions in this paper. First, we develop a microbenchmark-based performance model for three major components of GPU application runtime: the instruction pipeline, shared memory access, and global memory access. Second, we demonstrate how the model could guide programmers to optimize three representative real-world applications respectively limited by these three components. Third, by analyzing the performance of these applications, our model suggests architectural improvements on hardware resource allocation, avoiding bank conflicts, block scheduling, and memory transaction granularity. Fourth, to the best of our knowledge, this is the first modeling work based on a native GPU instruction set, which is critical for the accuracy of our model.

Baghsorkhi et al. [8] and Hong and Kim [9] recently authored excellent studies on GPU performance modeling. We see several major differences between these studies and our work. First, instead of trying to build an analytical model based on an abstraction of GPU architecture and then verifying the model by microbenchmarks, we adopt the reverse strategy. We first design microbenchmarks, observe the benchmark results, and then derive a simple throughput model respectively for instruction pipeline, shared memory, and global memory costs. This microbenchmark-based approach allows us to observe and consider only the architecture and programming factors that are most relevant to

performance, and make sure our model complies with the real program behavior. Second, the focus of these previous two studies is the prediction of program execution time, while our focus is on identifying performance bottlenecks in a quantitative way and guiding programmers and architects for optimizations. Third, our model is based on a native GPU instruction set instead of the intermediate PTX [10] assembly language or a high-level language. Simulating only the PTX instruction set leads to poor accuracy, because PTX code is not run directly on GPU hardware but instead is further compiled to native machine instructions where significant compiler optimizations are applied [6]. Fourth, these two studies are mainly based on static program statistics, while ours is based on dynamic program statistics collected from the Barra simulator, which enables us to handle data-dependent applications.

Compared to a set of recent studies on performance autotuning by empirical search [11, 12, 13, 14, 15], we provide an alternative optimization solution. Certainly search-based approaches are a powerful tool for optimization, but we note two disadvantages of such an approach. First, they provide little insight into real program behavior and architectural evaluation. Second, they require programmers to write programs in a parameterized way to accommodate various tuning parameters such as the granularity of parallelism, flexible memory access patterns, the level of loop unrolling, and application-specific parameters. In contrast, our goal is to build a performance model that guides programmers to write the right program directly, rather than write all possibilities of a program and then search for the best.

The rest of the paper is organized as follows. Section 2 reviews the GPU architecture and programming model. Section 3 describes our modeling methodology. Section 4 conducts micro-benchmarks for GPU performance modeling. Section 5 demonstrates the usefulness our model against three case studies. Section 6 concludes and describes future work.

2 GPU Architecture and Programming Model

Modern GPUs are throughput-oriented devices made up of hundreds of processing cores. They maintain a high throughput and hide memory latency by multi-threading between thousands of threads. The GPU is a two-level hierarchical architecture. It is made of vector processors at the top level, termed streaming multiprocessors (SMs) for NVIDIA GPUs and SIMD cores for AMD GPUs. Each vector processor contains an array of processing cores, termed scalar processors (SPs) for NVIDIA GPUs and stream processing units for AMD GPUs. All processing cores inside one vector processor can communicate through an on-chip user-managed memory, termed shared memory for NVIDIA GPUs and local memory for AMD GPUs.

The CUDA [1] and OpenCL [2] APIs share the same SPMD (Single Program Multiple Data) programming model. CUDA virtualizes SMs as blocks (equivalent to work-groups in OpenCL) and SPs as threads (equivalent to work-items in OpenCL), which enable programmers to run thousands of threads and blocks across different generations of GPUs regardless of the number of physical processors. A key concept of the CUDA programming model is the *warp*, equivalent to the *wavefront* in AMD GPUs. A warp is a group of 32 threads that execute in lockstep in a SIMD fashion. Because the GPU architecture shares a single instruction unit for all threads in a warp, a warp is the smallest unit of work a GPU issues. As a result, problems with less than 32-way parallelism will still have 32 threads running, some of which will not do useful work.

In this paper, although we focus on the CUDA-enabled NVIDIA GTX 285 GPU, we believe our performance modeling methodology is also applicable to any GPU architecture and GPU programming API. However, certain adaptations may be required. For example, AMD GPUs use VLIW (not scalar) cores in each vector processor. In this case, we need to add the ability to handle packed VLIW instructions.

3 Performance Modeling and Analysis Methodology

The traditional GPU performance model widely used for program optimization is at the algorithmic level. In this work, we redefine this model at the instruction execution and GPU architecture level for more accurate performance analysis. In the traditional model, programmers calculate the sustained computational rate and memory bandwidth based on the measured program's execution time and algorithmic complexity. From this, they infer if the program is compute-bound or memory-bound by comparing the sustained compute/memory performance with the peak GPU performance. If the program is memory-bound, possible optimizations are to reduce the number of memory transactions, or to trade computations for memory bandwidth, and vice versa for compute-bound program.

However, this high-level model can only give a rough idea of performance and in many cases fails to identify performance bottlenecks. We note several reasons for this. First, instructions used for actual computations are only a part of the entire executed instruction sequence; others include instructions for control, address calculation, memory operation, and so on. Therefore, a program can be instruction-throughput-bound instead of compute-bound if it has a low computational density. Second, a set of memory transactions at an algorithmic or program level may split into multiple transactions at the hardware level if they are not in a continuous memory segment. Third, the tradi-

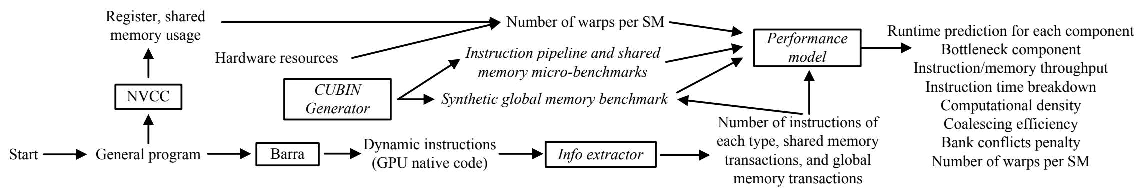


Figure 1: Our performance modeling workflow. Tools in italics are developed by us. The CUBIN generator generates synthetic benchmarks based on GPU native code. The info extractor takes the dynamic instruction count from Barra and generates inputs for the three components of our performance model: the instruction pipeline, shared memory, and global memory.

tional model does not take into account the effects of on-chip shared memory and bank conflicts.

To address these issues, we develop a performance model at an instruction and architecture level. Our model simulates the performance of three major components of GPU performance: the instruction pipeline, shared memory access time, and global memory access time. By estimating the time spent on each component, our model identifies the program bottleneck as the component spending most of the time. We assume the time spent by non-bottleneck components is covered by the bottleneck component, based on two reasons, (1) the GPU allows simultaneous instruction, shared-memory, and global-memory operations, and (2) the design philosophy of GPU is to hide memory latency by context switching between independent warps, and thus able to achieve near-perfect overlapping, instead of being held by intra-warp dependencies. This assumption will under-estimate the total execution time when there are insufficient warps and scarce independent instructions inside a warp. We divide a program into multiple stages by synchronization barriers. If there is only one block on a streaming multiprocessor (SM), we serialize all stages divided by synchronization barriers, and identify a performance bottleneck for each stage. Because GPU synchronization is local to a block, if there are multiple blocks, we assume different stages could still be overlapped, and we estimate a single performance bottleneck for the whole program. This treatment of synchronization's effect on multiple blocks will give better-than-reality performance as we will show later in the case studies, because synchronization decreases the number of independent warps in a block and thus drops the instruction and shared memory throughput.

We base each component's model on micro-benchmarks. For instruction pipeline modeling, we classify instructions into different types based on how expensive they are. Then we run micro-benchmarks to estimate the pipeline throughput of each instruction type at a different amount of warp-level parallelism. For a given general program, we calculate its execution time as a linear combination of the time spent on each instruction type. For shared memory modeling, we measure the sustained bandwidth at different amounts of warp-level parallelism. For a given general program, we first use bank conflict information to correct the number of memory transactions derived by program statistics, and then we estimate the time by using the bandwidth at a corresponding amount of warp-level parallelism. To estimate the global memory bandwidth of a given general program, we run a synthetic benchmark of the same configuration. We develop a memory transaction simulator to compute the number of transactions at the hardware level. We use the functional simulator Barra [6] to generate the dynamic program execution information on how many times each instruction is executed. Then we use this information to generate the number of dynamic instructions of each type, the number of shared memory transactions, the number of global memory transactions, and the number of stages divided by synchronization barriers. Figure 1 shows our performance modeling workflow.

Our model guides programmers and architects by providing them detailed quantitative performance information on each of the architecture components: instruction pipeline, shared memory, and global memory. By comparing the time spent on each component, we identify which component is the performance bottleneck. We can further infer if this bottleneck is removed, what will be the next component that becomes the new bottleneck. After a bottleneck is identified, we provide information to track down the causes of the bottleneck. For instruction-pipeline-bound programs, we can identify the following possible causes: (1) low computational density, (2) expensive instructions such as rcp, cos, log, and (3) insufficient parallel warps. For shared-memory-bound programs, the possible causes are: (1) bank conflicts, (2) shared memory traffic generated by bookkeeping instructions, and (3) insufficient parallel warps. For global-memory-bound programs, the causes can be: (1) insufficient parallelism to cover the memory latency, and (2) uncoalesced memory accesses and large memory transaction granularity. By identifying these causes of the bottleneck, our model motivates programming or architectural solutions.

Table 1: Instruction types

Instruction type	Number of functional units	Example instructions
Type I	10	mul
Type II	8	mov, add, mad
Type III	4	sin, cos, log, rcp
Type IV	1	double precision floating point

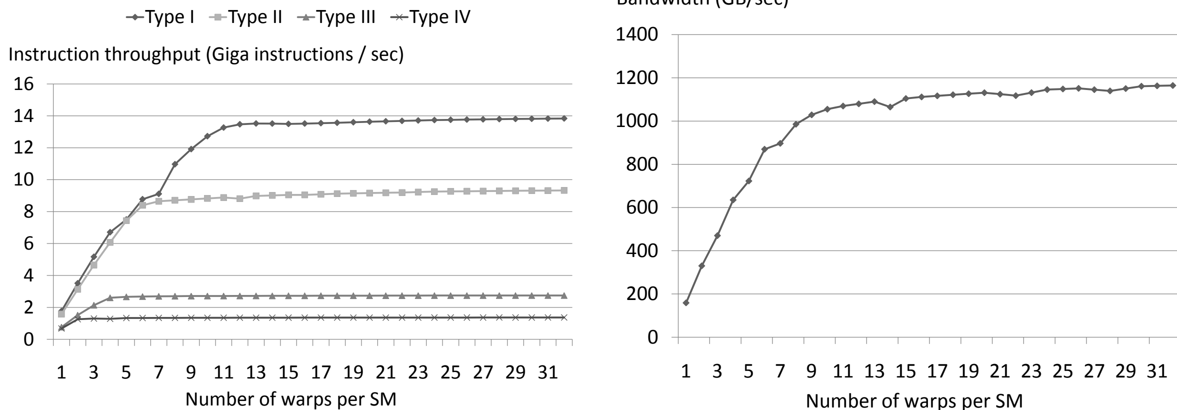


Figure 2: Instruction throughput for various instruction types (left) and shared memory bandwidth (right) as a function of warps per SM.

4 Performance Modeling

In this section, we describe the micro-benchmarks we designed to understand and model the performance of the instruction pipeline, shared memory, and global memory. Since the instruction set of native machine code is not publicly documented, we use the disassembler Decuda developed by van der Laan [16], on which Barra [6] is based as well. With the assistance of Decuda, we build a tool to modify the original binary instructions, assemble the modified instructions back to the binary code sequence, and finally embed the modified code into the execution file. This tool enables us to avoid compiler interference such as dead code elimination, and thus to develop binary code that exercises the GPU exactly the way we intend.

4.1 Instruction Pipeline

GPUs hide pipeline and memory latency by executing many parallel threads in an interleaved fashion. If the instruction pipeline is fully saturated, we know it runs at around the peak performance. The difficulty of modeling the instruction pipeline performance lies in the non-ideal situations, when the pipeline is under-utilized. The goal of this section is to explain how we model these under-utilized situations.

As discussed in Section 2, the smallest unit of work a GPU issues is a warp of 32 threads. The GPU primarily relies on inter-warp, rather than intra-warp, instruction-level parallelism, and the instruction window inside a warp is very small. Therefore, the major source of insufficient instructions to feed the pipeline is from insufficient warps. The GPU architecture we study here has several hardware ceilings per SM (streaming multiprocessor) that may limit the number of available warps to run: 16,384 registers; 16 kB of memory; 512 threads; 8 resident blocks; and 32 warps. Programs that reach these limits, either within one or across many blocks on a single SM, cannot launch more work.

We classify all instructions by the number of functional units that can run that instruction in an SM as shown in Table 1. Note that there are 10 multipliers in a SM (8 from the floating point units and 2 from the special functional units). The theoretical peak throughput of an instruction is calculated by $\frac{\text{numberFunctionalUnits} \cdot \text{frequency} \cdot \text{numberSM}}{\text{warpSize}}$. For example, the peak throughput of MAD (fused multiply-add) is $\frac{8 \cdot 1.48\,\text{GHz} \cdot 30}{32} = 11.1$ Giga instructions/s. Since one MAD is 2 floating point operations, the theoretical peak floating-point performance is $11.1 \cdot \text{warpSize} \cdot 2 = 355.2 \cdot 2 = 710.4$ GFLOPS. Each type of instruction has a different cost depending on how many functional units there are for

this instruction. We write micro-benchmarks that repeatedly run an instruction of each type. By choosing the size of blocks and the number of blocks, we can control the number of warps resident in a SM. We measure the instruction throughput for various amounts of parallel warps as shown in Figure 2, left. Note that the more functional units there are, the more parallel warps we need to cover the pipeline latency. The saturation point of type II instructions is 6 warps, which suggests the number of instruction pipeline stages is around 6.

4.2 Shared Memory

Each SM has 16 KB of shared memory organized in 16 banks. The theoretical peak throughput is calculated as $\text{numberSP} \cdot \text{numberSM} \cdot \text{frequency} \cdot 4\,\text{B} = 1.48\,\text{GHz} \cdot 8 \cdot 30 \cdot 4\,\text{B} = 1420\,\text{GB/s}$. The shared memory micro-benchmarks repeatedly move data from one shared memory region to another. We measure the shared memory throughput for various numbers of warps (Figure 2, right), as we did for modeling the instruction throughput. Comparing the two graphs in Figure 2, we notice that shared memory has a longer memory pipeline than the instruction pipeline, and thus needs more parallel warps to cover its latency.

In shared memory, adjacent 4-byte words are stored in adjacent banks. If multiple threads access different locations in the same bank, all memory accesses will be serialized. For example, if 3 threads read from different locations in the same bank, there would be 3 memory transactions, instead of 1 in the case they read from different banks. Since the functional simulator Barra [6] does not collect bank conflicts information, we wrote an automated program to derive the effective number of shared memory transactions by specifying the degree of bank conflicts of each shared memory access.

4.3 Global Memory

Since global memory is shared across SMs, we do not model it against the number of parallel warps as we did for the instruction pipeline and shared memory. We found that the global memory bandwidth is sensitive to three major factors: the number of blocks, the number of threads per block, and the number of memory transactions per thread, as shown in Figure 3. To saturate the bandwidth, we need a sufficient number of total memory transactions, which we can increase by using either more blocks, or more threads per block, or more memory requests per thread. The theoretical peak bandwidth is calculated as $\frac{\text{memoryFrequency} \cdot \text{busWidth}}{8\,\text{bits/byte}} = \frac{2.484\,\text{GHz} \cdot 512\,\text{bits}}{8\,\text{bits/byte}} = 160\,\text{GB/s}$. The 30 SMs on the GTX 285 are grouped into 10 clusters, where the 3 SMs in a cluster share a single memory pipeline. This is why we see sawtooth patterns with a period of 10 when the bandwidth is near the peak, and for the best throughput, the number of blocks should be a multiple of 10. The figure also suggests that blocks are scheduled uniformly to the clusters. When the number of blocks is getting larger, this leftover effect becomes weaker (note the figure shows that the fluctuation becomes smaller as the number of blocks grows). When there are insufficient memory transactions to cover the latency of the memory pipeline, it is almost free to have more memory transactions. This is why the plot is almost linear with the number of blocks when it is far below the peak bandwidth.

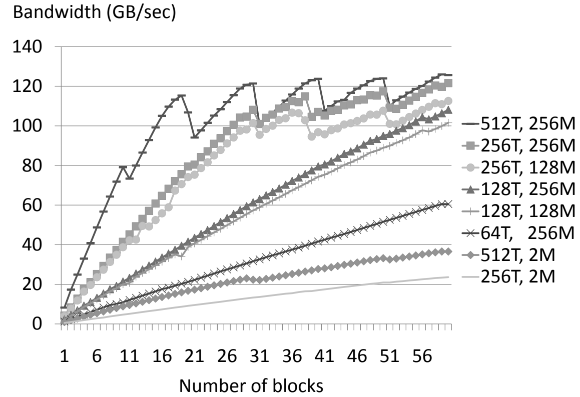


Figure 3: Global memory throughput at various block sizes, numbers of blocks, numbers of memory transactions per thread. In the legend, T stands for threads, and M stands for memory transactions per thread.

Since the global memory behavior is fairly complex, it is hard to accurately simulate it with a simple model at a high level as we did for the instruction pipeline and shared memory. To accurately estimate the global memory bandwidth of a general program, we instead run a synthetic benchmark of the same number of blocks, block size, and the number of memory transactions per thread. However, this approach does not tell the whole story, because a memory transaction at instruction level may split into multiple transactions at hardware level based on the memory coalescing rule. To address this issue, we developed a memory transaction simulator to simulate the number of hardware transactions.

CUDA issues memory transactions at a granularity of a half-warp. For CUDA architectures of version 1.2 and 1.3, the following coalescing protocol is used: (1) for each memory transaction, find the memory segment that contains the address requested by the lowest numbered thread; (2) find all other threads whose requested address is in this segment; (3) reduce the segment size if possible; (4) repeat the above process until all threads in a half-warp are served. Currently the minimum segment size CUDA supports for floating point numbers is 32 bytes. We implement this protocol in our transaction simulator whose input is the requested memory addresses of all threads.

Table 2: Register and shared memory (smem) usage per thread for various sub-matrix sizes in dense matrix multiply, and the number of blocks that can fit into one multiprocessor given the particular per-thread constraint. The maximum number of blocks per multiprocessor is 8. A block consists of 64 threads or 2 warps for all three cases.

sub-matrix size	register	smem	# blocks (register)	# blocks (smem)	# blocks	# active warps
8×8	16	348	16	47	$\min(16, 348, 8) = 8$	$8 \cdot 2 = 16$
16×16	30	1088	8	15	$\min(8, 15, 8) = 8$	$8 \cdot 2 = 16$
32×32	58	4284	3	3	$\min(3, 3, 8) = 3$	$3 \cdot 2 = 6$

5 Case Studies

In this section, we use our performance model to study and optimize three applications: dense matrix multiply, tridiagonal systems solver, and sparse matrix vector multiply. These three applications represent classes of applications that are respectively bound in performance by the instruction pipeline, shared memory, and global memory.

5.1 Dense Matrix Multiply

We show in this section that our model identifies the performance bottlenecks for various sub-matrix sizes, and demonstrates the additional costs of shared memory accesses for larger sub-matrices. The model further suggests hardware optimizations to improve performance.

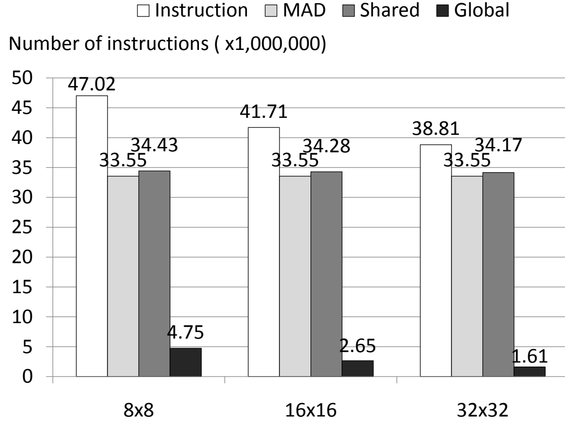
We study a computational procedure developed by Volkov and Demmel [17]. The procedure divides the result matrix into sub-matrices, with each sub-matrix mapped to a block. The major improvement introduced by Volkov and Demmel is reordering the computational loops so that the sub-matrix of only one input matrix needs to be stored in shared memory, instead of sub-matrices from both input matrices. However, their implementation uses a sub-matrix of fixed size 16x16. It is not obvious why this size outperforms other sizes. Furthermore, this computational procedure only achieves 56% of GPU peak performance, and lacks a performance analysis on why this is the case. The goal of this section is to answer these questions.

We first measure and simulate the performance of 8×8, 16×16, and 32×32 sub-matrix sizes. Ideally a larger sub-matrix would increase the performance for two reasons: (1) it decreases the redundant memory loads; Figure 4(a) shows that the number of global memory transactions are reduced by 45% and 40% respectively from 8×8 to 16×16 and from 16×16 to 32×32, and (2) it increases the computation density; Figure 4(a) also shows that the total dynamic instruction count decreases as we use larger sub-matrices, while the MAD instruction count remains constant ($\frac{\text{matrixSize}^3}{\text{warpSize}}$). However, in reality, a size of 16×16 actually achieves the best performance (Figure 4(b)).

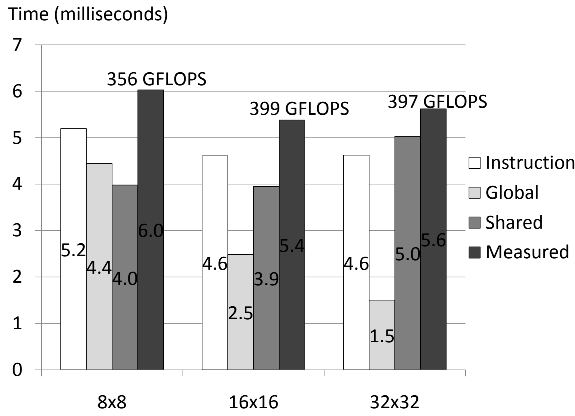
Figure 4(b) also shows the simulated performance for the three sub-matrix sizes. We note that for the sub-matrix sizes 8×8 and 16×16, the performance is bottlenecked by the instruction pipeline, but for the sub-matrix size 32×32, the performance bottleneck shifts to shared-memory access. Although the 32×32 case has a similar shared memory count as the 16×16 case (Figure 4(a)), its total time spent on shared memory access is significantly larger (Figure 4(b)). The reason for this lies in the hardware resource usage for the three cases shown in Table 2. The register and shared memory demands for the 32×32 case are so high that the number of resident blocks in a multiprocessor is reduced from 8 to 3, which is equivalent to 6 warps. This leads to insufficient parallelism to hide the latency of the instruction pipeline and the shared memory pipeline, with a corresponding decrease in performance.

Now that we know how many warps can run per SM (streaming multiprocessor), how does this translate to overall performance? For this, we return to the microbenchmarks we calculated earlier (Figure 2). For {6, 16, 32 (max)} active warps per block, we expect an instruction throughput of {8.39, 9.05, 9.33} gigainstructions/s and a shared memory bandwidth of {870, 1112, 1165} GB/s. Note the sustained memory bandwidth with 6 warps for the 32×32 case is considerably lower than that with 16 warps for the 8×8 or 16×16 cases. We also note that shared memory is more vulnerable to insufficient parallelism than the instruction pipeline, since the instruction throughput does not drop much from 16 warps to 6 warps. It is also worth mentioning that our simulated time is about 14% less than the measured time, as shown in Figure 4(b), because we did not consider the synchronizations' effects on performance. Synchronization barriers in a block hold the matrix computation until all necessary data is loaded to shared memory, so the actual amount of parallel warps is fewer than the total number of warps, which results in even lower instruction and shared memory throughput in reality.

Finally, we can also address the reasons why matrix multiply only achieves 56% of its theoretical peak performance: (1) the sustained instruction throughput is only 81% of the peak throughput, because the instruction pipeline is not per-



(a) Numbers of total instructions, MAD, shared memory and global memory transactions (measured per warp).



(b) Measured performance and simulated performance breakdown in terms of instruction execution, shared memory access, and global memory access.

Figure 4: Performance and program statistics comparison for various submatrix sizes: 8×8, 16×16, and 32×32. The two input matrices have the same dimensions of 1024×1024.

fectly saturated, and (2) although 80% of the total instructions are MAD instructions that are used for actual computations, the rest are bookkeeping instructions used for program control, address calculation, and memory operations.

With these results, we can draw two conclusions for possible architectural improvements. First, although the maximum number of resident warps in a multiprocessor is 32, the maximum number of resident blocks is only 8, which limits the number of blocks to 8, or the number of warps to 16, for the 8×8 and 16×16 cases. If the maximum number of blocks was increased to 16 (without changing any other resources), there would be more resident parallel warps to achieve better instruction and shared memory throughput. Second, if we increase the register and shared memory resources per multiprocessor, we can fit more warps onto a multiprocessor to keep the same shared memory throughput for the 32×32 case, but achieve better performance because of its higher computational density.

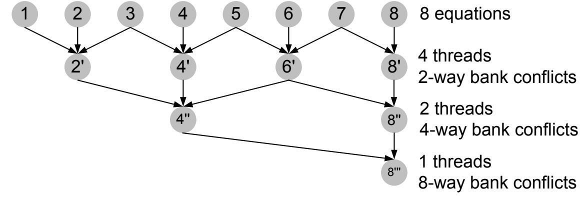


Figure 5: Communication pattern of cyclic reduction (forward reduction phase only) for solving an 8-equation system. A dot stands for an equation and n' stands for an updated equation n.

5.2 Tridiagonal Solver

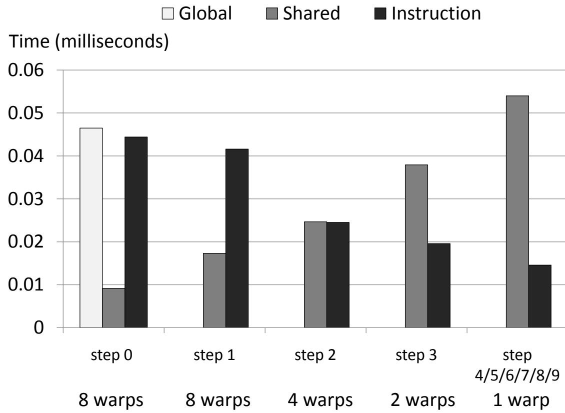
In this section, we demonstrate our model's usefulness on simulating shared memory throughput, quantifying the effects of bank conflicts on performance, and estimating the potential benefit of an optimization technique to remove bank conflicts. We further verify that this technique does improve performance by 1.6× as expected. Our performance analysis also indicates a need for hardware improvements on avoiding bank conflicts and block scheduling.

Tridiagonal linear systems are of importance to many problems in numerical analysis and computational fluid dynamics, and cyclic reduction is one of the most popular parallel algorithms to solve such a system. A traditional either-compute-or-memory bound performance analysis approach is not applicable to this application, because the application is neither computation-bound nor memory-bound, and can only achieve a computational rate of 6 GFLOPS and a bandwidth of 7 GB/s.

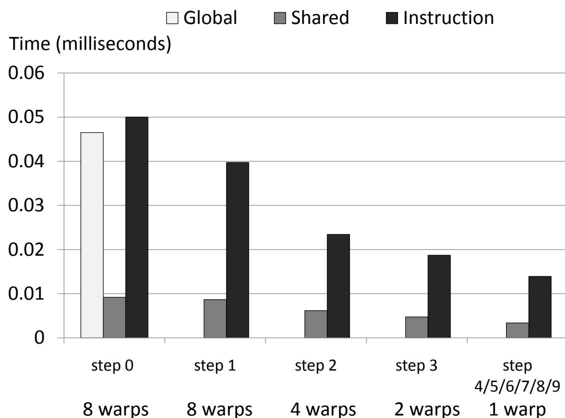
In previous work in this area, Zhang et al. [18] note that GPU-based cyclic reduction suffers from shared memory bank conflicts and propose a more complex hybrid solver that combines cyclic reduction and its variant to alleviate this problem. Goeddeke et al. [19] instead propose an interleaving addressing scheme to completely eliminate the bank conflicts. Alternatively, we propose and evaluate a simple but effective padding technique to remove bank conflicts.

Cyclic reduction has two phases: forward reduction and backward substitution. Because the two phases have a similar communication pattern, we only show the forward reduction phase in Figure 5. For a system of size n, forward reduction requires $\log_2(n)$ steps to consecutively reduce the original system to a 1-equation system. All systems are first loaded into shared memory, then solved on chip. As the memory access stride doubles every step, the number of bank conflicts are doubled as well, from 2-way bank conflicts in step one, to 4-way in step two, to 8-way in step three, and so on.

Figure 6(a) shows the simulated performance breakdown for pure cyclic reduction (CR). In this example, we solve 512 512-equation systems in parallel with systems mapped to blocks and equations mapped to threads. There are

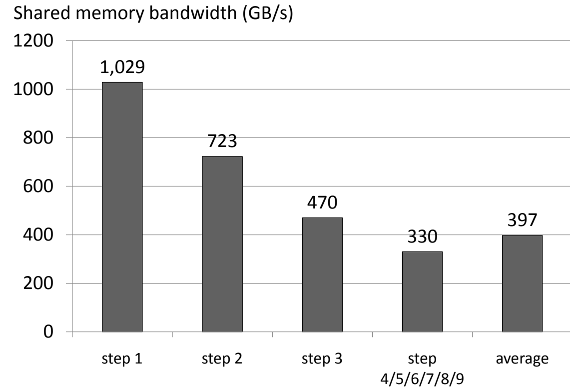


(a) Cyclic reduction

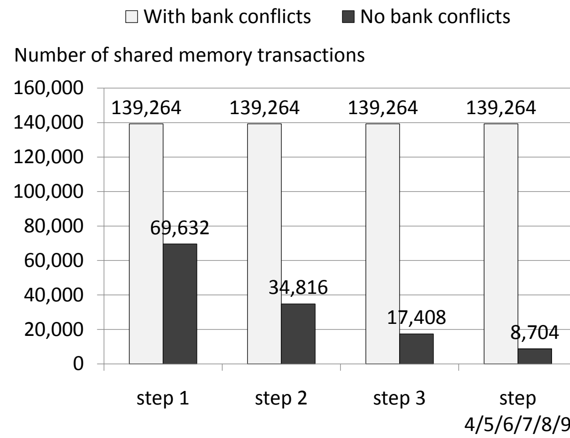


(b) Cyclic reduction with no bank conflicts

Figure 6: Simulated performance breakdown for CR and CR-NBC for solving 512 512-equation systems (forward reduction phase only). Step 0 loads the system into shared memory. The system solving begins at Step 1.



(a) Sustained shared memory bandwidth under various numbers of parallel warps.



(b) Number of shared memory transactions for the algorithmic steps in forward reduction.

Figure 7: Sustained shared memory bandwidth and the number of shared memory transactions.

$\log_2(512) = 9$ steps to reduce the system to a 1-equation system. In the first step, 256 parallel threads work on 256 even-indexed equations and their two neighbors. In the second step, the number of parallel threads is reduced to 128. The algorithm reduces the amount of parallel threads each step until it reaches a single thread. However, since the minimum unit of work on the GPU is a warp of 32 threads, steps 4–9 have identical performance characteristics, so we show them together in Figure 6. Note that Figure 6(a) shows that a pure cyclic reduction implementation has a large cost from shared-memory access.

Due to the limited amount of shared memory, we can only fit one block per multiprocessor. Since there is only one resident block on a multiprocessor, and we must place a synchronization barrier between loading a system to shared memory and solving the system, the global memory loads and subsequent computations are serialized. This is in contrast to the previous matrix multiply example, in which global memory access and computation are overlapped, because of multiple resident blocks (although there are synchronization barriers as well). Because of additional synchronization barriers between neighboring algorithmic steps, all steps are serialized. As shown in Figure 6(a), the performance of CR is bound by global memory access in step 0, by instruction throughput in step 1, and by shared memory access in all subsequent steps.

As the algorithmic step keeps reducing the amount of work by half each step, the number of shared memory transactions should have been reduced by half as well. However, because the number of bank conflicts doubles each step, the number of shared memory transactions remains constant, as illustrated in Figure 7(b). Making this bad situation worse, the sustained shared memory bandwidth drops, as there are fewer and fewer active warps after each step (Figure 7(a)). The average sustained bandwidth is only 397 GB/s; steps 4–9 suffer from the lowest sustained bandwidth. Figure 6(a) shows that if we could remove the bank conflicts, the new bottleneck for steps 2–9 would be the instruction pipeline,

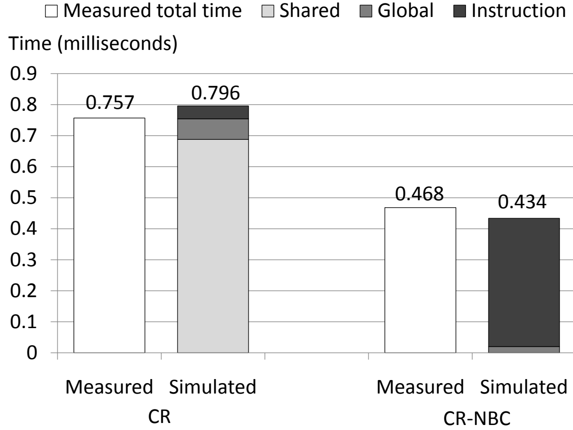


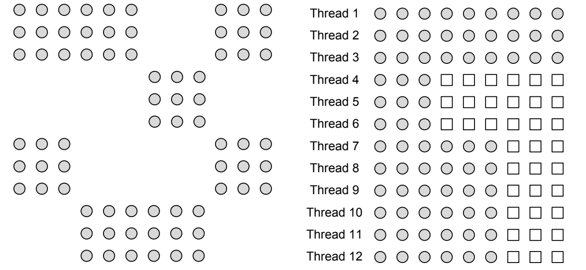
Figure 8: Measured and simulated performance of CR and CR-NBC.

and the performance could be significantly improved.

We implemented a new CR with no bank conflicts (CR-NBC) by a padding technique. Since there are 16 shared memory banks, we pad 1 element per 16 elements, which redirects all conflicted accesses to available banks. Figure 6(b) shows that for CR-NBC, the first step is bound by instruction throughput because of its more complex addressing calculation, and now all subsequent steps are bound by instruction throughput as well because bank conflicts are removed. CR-NBC effectively removed all bank conflicts and at the same time introduced minimal extra instruction overhead: CR-NBC has a similar instruction count to CR. The padding technique has shifted the bottleneck from shared memory to the instruction pipeline, which improves the performance of CR by 1.6×. However, the effective computational rate is still low because of two reasons: (1) the insufficient warp-level parallelism during the later stages of CR's forward reduction phase, and (2) the low computational density of CR/CR-NBC (there are only about one tenth of total instructions are doing actual computations).

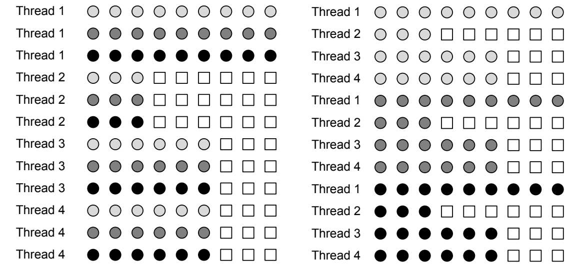
Figure 8 shows the measured and simulated performance, which agree closely within a 7% error. The time of CR is mainly dominated by shared memory access and the time of CR-NBC is mainly dominated by instruction execution. There is a small amount of time for global memory access that cannot be overlapped by instruction execution, and a small amount of instruction execution time from step 1 in CR when bank conflicts have not yet become a bottleneck.

It is fair to say that the current shared memory organization is well-suited for simple, structured problems, but as GPUs increasingly target more irregular, complex problems, our performance model can help identify the impact of changing this organization. Two architectural improvements could deliver better performance: (1) change the number of shared memory banks from 16 to a prime number to avoid bank conflicts; (2) introduce a mechanism to release unused hardware resources early as a block uses fewer and fewer threads. With this, we could schedule subsequent blocks onto SMs to increase warp-level parallelism and deliver better instruction and shared memory throughput.



(a) A 12x12 sparse matrix.

(b) Matrix stored in the ELL format. Padded entries are represented as squares.



(c) Straightforward ELL storage with block processing. Rows are divided into three interleaved groups, each specified by a gray level.

(d) Interleaved ELL storage with block processing. Rows of the same group are placed together.

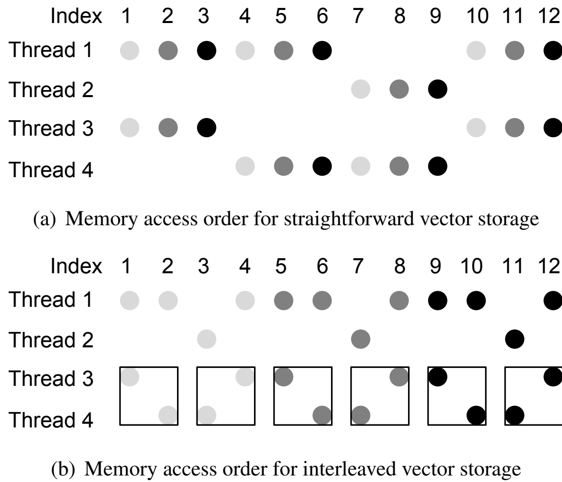
Figure 9: ELLPACK (ELL) storage formats.

5.3 Sparse Matrix Vector Multiply

In this section, we use a memory-bound application to show our model's ability to simulate the number of hardware transactions. This ability enables us to make an optimization that otherwise would not be noticed. Our model also suggests that a smaller transaction granularity would improve performance further.

Sparse matrix vector multiply (SpMV) lies in the heart of iterative methods for numerous scientific computing applications. Bell and Garland [20] experimented with a variety of matrix formats to increase SpMV performance on the GPU. They found that the ELLPACK (ELL) format [21] enjoys coalesced memory access, and generally achieves superior performance for matrices with more uniform numbers of row entries. The state-of-the-art work by Choi et al. [14] improved the performance of ELL format by using a blocked ELLPACK format (BELL). In this section, we will show how our performance analysis tool guides us to further improve the performance of BELL by 18%.

Figure 9(a) and Figure 9(b) respectively show a sparse



(a) Memory access order for straightforward vector storage

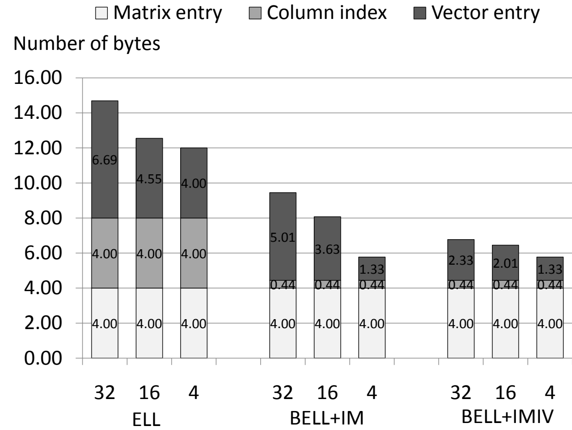
(b) Memory access order for interleaved vector storage

Figure 10: Memory access order for straightforward vector storage and interleaved vector storage. The vector entries grouped in a square share the same memory transaction. In this example, we use a memory transaction granularity of 8 bytes and a transaction issue granularity of 2 threads.

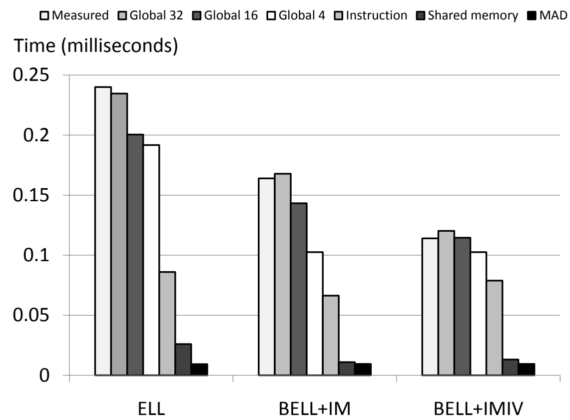
matrix and the same matrix stored in the ELL format. In the ELL format, we first compress the matrix entries to the left side and then pad all rows so that we have a rectangular ELL matrix. We store this ELL matrix column by column, and pad at the end of each column to meet the alignment requirement. We also store the corresponding column index for each matrix entry. The GPU program maps each row to a thread. The column-by-column matrix storage allows coalesced memory access, because each continuous thread accesses a continuous memory region (Figure 9(b)). To process each matrix entry in the ELL format, we load three values from global memory: a matrix entry, a column index, and a vector entry. The goal of the BELL format is to reduce the amount of memory loads if a matrix has a block structure. In the BELL format, we only store the column index of the top-left entry for each block, and each GPU thread processes a row of blocks. For the matrix shown in Figure 9(a), to process each 3x3 block, we load 9 matrix entries, 1 column index, and 3 vector entries. Thus the BELL format reduces the column index loads to 1/9 and vector entry loads to 1/3 of the ELL format.

Since each thread processes three continuous rows and continuous threads access different memory regions, memory access becomes uncoalesced (Figure 9(c)). For coalesced memory access, we must reorder the rows in an interleaved fashion, so that continuous threads access contiguous memory regions (Figure 9(d)). This interleaving technique is natural for the matrix storage and is the same as what Choi et al. suggest in their work [14].

Our tools indicated that global memory access was the key to SpMV performance and specifically, that a significant impediment to peak throughput was the number of uncoalesced memory accesses to vector entries. Thus we



(a) Simulated average number of bytes per matrix entry processing.



(b) Measured performance and simulated performance breakdown for three storage formats.

Figure 11: Simulated number of memory transactions, performance breakdown, and performance comparison. 32, 16, 4 respectively denote a memory transaction size of 32 bytes, 16 bytes, and 4 bytes. IM stands for interleaved matrix. IV stands for interleaved vector. The analysis is for the single precision case, in which we use 4 bytes per matrix/vector entry.

concentrated on optimizing these accesses. The key insight from this focus was that it is better to store not only the matrix but also the vector in an interleaved way. Memory access to vector entries is uncoalesced anyway and depends on the sparsity of the matrix. However, the intuition here is that neighboring rows have similar entry positions, and the more apart two rows are, the less chance they will share a single memory transaction for vector entries. Interleaving scatters the vector entries around so that they have a better chance to be grouped in a single memory transaction.

Figure 10 shows the effects of interleaving. In this example, for simplicity, we use a memory-transaction-issue granularity of 2 threads, instead of 16 threads in the CUDA case. We also use a memory transaction size granularity of 8 bytes (2 4-byte vector entries), instead of 32 bytes in the CUDA case. Figure 10(a) shows which vector entries are accessed by each thread, which is decided by the sparse

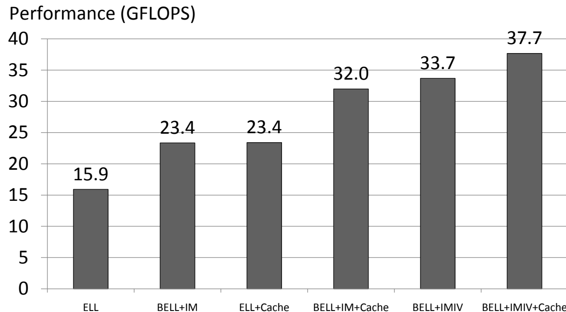


Figure 12: Performance comparison for different combinations of optimization techniques in single precision.

matrix in Figure 9(a). In this case, there are no memory transactions shared by the first group of two threads or the second group of two threads. For example, the first element accessed by thread 1 (entry 1) is too far from the first element accessed by thread 2 (entry 7) to be grouped in a single contiguous 8-byte memory transaction. However, the interleaved vector storage shows 6 memory transactions shared by thread 3 and thread 4 (Figure 10(b)). The distance between the first element accessed by thread 1 and the first element accessed by thread 2 is decreased as well.

Figure 11(a) shows the average number of bytes required to process a matrix entry generated by our memory transaction simulator. We use a naturally 3x3 blocked sparse matrix named QCD, from the benchmark suite of 14 sparse matrices used in prior work by others [14, 20, 22]. In an ideal situation, where all memory accesses are coalesced (equivalent to a memory transaction size granularity of 4 bytes), the ELL format requires $4 + 4 + 4 = 12$ bytes to process a single matrix entry. However, in reality the CUDA memory transaction size is 32 bytes, and the memory access to vector entries is uncoalesced. This results in a memory access requirement of 6.69 bytes per vector entry. We also simulated a smaller transaction size, 16 bytes, which is not supported by the current GPU. The smaller size reduces the average number of bytes per vector entry to 4.55 bytes. For the BELL format, because $3 \times 3 = 9$ entries share a single column index, the bytes per column index is reduced to $1/9$. The interleaved vector storage significantly reduces the bytes per vector entry. Figure 11(b) shows the measured performance and simulated performance breakdown. The error between the measured and the simulated performance of bottleneck factor is within 5%. In all three cases, the performance is bottlenecked by global memory access. We also show that with a smaller transaction size of 16 bytes, the performance would be improved. If the time of global memory could be reduced even further, the new bottleneck would be the instruction pipeline. However, we will be still far from the peak GFLOPS rate of the GPU, because the computational density of the program is so low that only about 1/10 of total instructions (all of them are MAD instructions) are devoted to actual computations.

Although we have not built a texture cache simulator into our model, we tried using texture cache for vector entries, as what the two previous studies [14, 20] did. Figure 12 compares the performance of ELL, BELL+IM, and BELL+IMIV in two cases, using texture cache or without using texture cache. ELL+Cache and BELL+IM+Cache respectively represents the best performance achieved by Bell and Garland [20] and Choi et al. [14]. Our vector interleaving optimization BELL+IMIV is very effective and it outperforms the previous method even without using the texture cache. Our BELL+IMIV+Cache is 18% faster than the the previous best BELL+IM+Cache.

6 Conclusion

Our quantitative performance model for the GPU allows programmers and architects to identify optimization possibilities in modern GPU programs and architectures. Today, programmers do not know how effective an potential optimization will be until they try it out. In contrast, our performance analysis tool enables programmers to identify the performance bottlenecks, foresee the benefit of removing a certain bottleneck in a quantitative way, and decide if a potential optimization is worth the programming efforts. From an architecture design point of view, our performance analysis tool is able to identify architectural shortcomings against real-world applications, and suggest architectural improvements on hardware resources allocation, block scheduling, memory transaction granularity, and so on.

We believe our model has captured the GPU's primary performance factors, and we have showed a simulation accuracy within 5–15% on three representative case studies. Our work has several limitations that we hope to address with future research: (1) incorporate a cache model in memory system simulation (for texture memory and Fermi hardware caches), (2) develop a bank-conflict simulator for more general cases, (3) model the synchronization barrier's effects on warp-level parallelism, and (4) identify and model situations of non-perfect overlap of instruction execution, shared memory, and global memory access.

Acknowledgments

Thanks to Sylvain Collange for his support on the Barra simulator, without which this work is impossible. Thanks to Anjul Patney, Shubho Sengupta, Jonathan Cohen, Peng Wang, Nathan Bell, Paulius Micikevicius, Everett Phillips, and the anonymous reviewers for their helpful discussions and suggestions. Thanks also to our funding agencies, the HP Labs Innovation Research Program, the National Science Foundation (Award 0541448), and the SciDAC Insti-

tute for Ultrascale Visualization, and to NVIDIA for equipment donations.

References

[1] "NVIDIA CUDA compute unified device architecture, programming guide," http://developer.nvidia.com/.

[2] "The OpenCL specification," http://www.khronos.org/registry/cl/.

[3] "General-purpose computation using graphics hardware," http://www.gpgpu.org/.

[4] "ATI Stream Profiler," http://developer.amd.com.

[5] "NVIDIA Parallel Nsight," http://developer.nvidia.com.

[6] S. Collange, D. Defour, and D. Parello, "Barra, a parallel functional GPGPU simulator," Université de Perpignan, Tech. Rep. hal-00359342, Jun. 2009.

[7] G. Diamos, A. Kerr, and M. Kesavan, "Translating GPU binaries to tiered SIMD architectures with Ocelot," Georgia Institute of Technology, Tech. Rep. GIT-CERCS-09-01, 2009.

[8] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*. ACM, Jan. 2010, pp. 105–114.

[9] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009)*, Jun. 2009, pp. 152–163.

[10] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of PTX kernels," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC 2009)*, Oct. 2009, pp. 3–12.

[11] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S. Z. Ueng, S. S. Baghsorkhi, and W. W. Hwu, "Program optimization carving for GPU computing," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1389–1401, Oct. 2008.

[12] Y. Liu, E. Z. Zhang, and X. Shen, "A cross-input adaptive framework for GPU program optimizations," in *Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing (IPDPS '09)*, May 2009.

[13] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs," in *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*, Jun. 2009, pp. 256–265.

[14] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*. ACM, Jan. 2010, pp. 115–126.

[15] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for GPGPUs," in *ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing*, Jun. 2008, pp. 225–234.

[16] W. J. van der Laan, "Decuda and Cudasm, the cubin utilities package," 2009, http://github.com/laanwj/decuda.

[17] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Nov. 2008, pp. 31:1–31:11.

[18] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the GPU," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*, Jan. 2010, pp. 127–136.

[19] D. Göddeke and R. Strzodka, "Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 22–32, Jan. 2011.

[20] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *SC '09: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*, Nov. 2009, pp. 18:1–18:11.

[21] J. R. Rice and R. F. Boisvert, *Solving elliptic problems using ELLPACK*. New York, NY, USA: Springer-Verlag New York, Inc., 1984.

[22] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007, pp. 38:1–38:12.