A Throughput-Aware Analytical Performance Model for GPU Applications

Zhidan Hu, Guangming Liu, and Wenrui Dong

College of Computer, National University of Defense Technology Hunan, China
{Huzd,liugm,dongwr}@nscc-tj.gov.cn

Abstract. Graphics processing units (GPUs) have shown increased popularity in general-purpose parallel processing. This massively parallel architecture allows GPUs to execute tens of thousands of threads in parallel to solve heavily data-parallel problems efficiently. However, despite the tremendous computing power, optimizing GPU kernels to achieve high performance is still a challenge due to the sea change from CPU to GPU and lacking of tools for programming and performance analysis.

In this paper, we propose a throughput-aware analytical model to estimate the performance of GPU kernels and optimizations. We construct a pipeline for global memory access servicing and redefine the compute throughput and memory throughput as the speed of memory requests arriving and leaving the pipeline. Based on concluding the kernel throughput limiting factor, GPU programs are classified into compute-bound and memory-bound categories and then we predict performance for each category. Besides, our model can provide useful information on the direction of optimization and predict the potential performance benefits. We demonstrate our model on a manually written benchmark as well as the matrix-multiply kernel and show that the geometric mean of absolute error of our model is less than 6.5%.

Keywords: GPU, compute-bound, memory-bound, performance prediction, performance bottleneck.

1 Introduction

In recent years, the ceiling of high performance computing has been updated multiple times by the GPU-based heterogeneous systems [1]. The GPU architecture has garnered wide popularity since the increasing programmability and the ever friendly programming model. Even though hardware is providing high performance computing, implementing and optimizing parallel programs to take full advantage of the potential computing power still remains a big challenge.

Several programming languages have been proposed to reduce programmer's burden in porting parallel applications to GPUs such as Brook++ [2], CUDA [3], and OpenCL [4]. However, even with these newly developed programming languages, programming and optimizing programs to achieve better performance is still time-consuming and error prone.

To provide insight into performance bottlenecks in massively parallel architectures, especially GPU architectures, we propose a simple analytical model. The model can be used statically without executing a GPU application. The basic intuition of our analytical model is that the ability to hide long latency memory operations with interleaving executions of computation from different thread warps can be obtained based on the warp level parallelism of both computations and memory operations. By constructing the memory pipeline model and extending the concept of compute throughput, we classify GPU applications into compute-bound and memory-bound categories, and then we estimate the execution time for each category.

We evaluate our analytical model based on the CUDA programming model, which is specific for the CUDA-enabled NVIDIA GPUs. We compare the results of our analytical model with the actual execution time collected on the NVIDIA GPUs. Our results show that the geometric mean of absolute error of our model is less than 6.5%.

The contributions of our work can be concluded as follows:

- We construct the memory pipeline model and extract the memory throughput based on capturing the performance factor of uncoalesced memory access
- We redefine the concept of compute throughput to be the frequency of global memory requests leaving the SMs and reaching the memory pipeline
- We classify GPU applications into two categories as memory-bound and compute-bound based on values of redefined compute throughput and memory throughput
- An analytical performance prediction model is proposed to estimate the performance of both compute-bound and memory-bound GPU kernels.

2 Background

We provide a brief background on the GPU architecture and the programming model that we have modeled. In this work, although we focus on a CUDA-enabled NVIDIA GPU, we believe our performance model is also applicable to any GPU architecture and GPU programming API.

2.1 Overview of GPU Architecture and CUDA Programming Model

Graphics Processing Units (GPUs) have emerged as a promising alternative building block for the construction of high performance supercomputers, due to their unique combination of outstanding performance, energy-efficiency, density and cost [5].

The GPU architecture consists of several streaming multi-processors (SMs), each containing a set of streaming processors (sp) that run threads in a SIMD manner. All SMs are connected to an off-chip DRAM memory via a interconnect network. Tesla M2050 has 14 SMs, each equipped with 32 streaming processors, which makes for a total of 448 processing cores [6]. The M2050 employs a dual-issue instruction dispatcher per each SM which can issue two instructions to 32 GPU cores every two clock cycles and thus an average speed of issuing one instruction per clock cycle is achieved. The global memory space is divided into 6 partitions, each with a memory controller.

The CUDA programming model groups GPU threads into a grid of thread blocks. Each thread block is mapped to a SM in a round-robin manner and multiple thread blocks can be running simultaneously on one SM. Each thread is assigned a thread ID (tid), which is used for the data distribution and control condition. Threads are created, managed, scheduled and executed at the granularity of thread warp, which contains 32 threads for most GPUs. The CUDA memory model has an off-chip global memory space, which resides in the DRAM memory and is accessible by all threads.

2.2 Related Work

A commonly introduced metric to characterize a program is arithmetic intensity which accounts operations per data transferred between the processor and the cache. The Roofline performance estimation model [7] introduces operational intensity as another metric which accounts operations per byte that transferred between DRAM and the processor. Zhang and Owen [8] constructed a GPU performance model in a quantitative way to estimate the execution time of arithmetic pipeline, shared memory, and global memory respectively. Performance bottlenecks are derived based on the modeled execution time of each component. Hong and Kim [9] authored an excellent study on analytical GPU performance modeling and using two metrics CWP and MWP to specify a program to be compute-intensive or memory-intensive, which is the most related to our method. However, we classify and predict performance of GPU kernels based on the kernel throughput which complies with the throughput-oriented GPU architecture.

In the past few years, many studies on GPU performance modeling have been proposed. Baghsorkhi et al. [10] proposed a work flow graph (WFG)-based analytical model to predict the performance of GPU applications. The WFG is an extension of a control flow graph (CFG), where nodes represent instructions and arcs represent latencies. Meng et al. [11] proposed a GPU performance projection framework to predict performance in a cross-platform style based on the abstraction of CPU code skeletons.

Hong and Kim [9] proposed the MWP-CWP based GPU analytical model, which shares the most common with our proposed model in the following two aspects: (1) the two analytical models extract parallelism from GPU kernels at the granularity of thread warps and overall execution time is counted on the ability of hiding the latency of global memory accesses by computations. (2) The latency of an uncoalesced global memory transaction can be synthesized as the sum of a base latency and multiple extra delays, each representing the departure delay between uncoalesced global memory transactions. Apart from that, we also see differences between the two models. First, in our work, the departure delay between two uncoalesced global memory accesses turns out to be the DRAM access latency of one memory transaction which can be calculated based on the values provided in the GDDR datasheet instead of profiling. Second, we construct a pipeline model for global memory accesses and utilize the pipeline throughput to describe the memory performance. Third, the computations and memory access operations in the kernels are separated and performances of both parts are represented by the extended compute throughput and memory throughput. As GPU programs are classified into compute-bound and memory-bound categories, the potential performance improving needs to emphasize on enhancing the value of compute throughput or memory throughput. In summary, our model predicts performance of GPU kernels in a more straightforward way and thus is more suitable for the throughput-oriented GPU architectures.

3 Program Classification

In this section, we first redefine compute throughput and memory throughput, and then classify GPU kernels into compute-bound and memory-bound categories.

3.1 Compute Throughput and Memory Throughput

Originally, the compute throughput refers to the throughput of arithmetic pipeline in a SM. We redefine the content of compute throughput as the time interval between warp switches to represent the frequency of memory requests being issued to the global memory interface. As all SMs can issue memory requests to global memory concurrently, the time interval should be divided by #SM, which is the number of SMs in a GPU. It is determined by the efficiency of executing one computation period which may be related to the performance factors of control flow divergence [12] and shared memory bank conflict [10] as we consider shared memory instructions have identical latency with compute instructions.

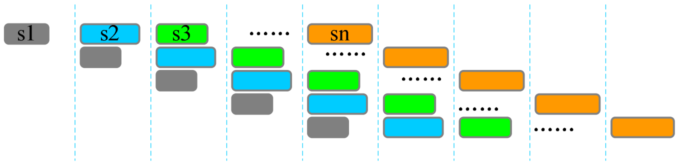


Fig. 1. A pipeline model for global memory accesses

The process of a global memory access includes several operations such as virtual address calculation, on-chip crossbar inter-connect traversal, virtual to physical address translation and physical to raw address translation, and DRAM access per each memory request. The DRAM access time here refers to the latency of reading/writing access to the DRAM chips and thus the latency is just small portion of the whole global memory access cost. In our model, the above operations are further divided into even more subtle steps that can be combined together to compromise a pipeline for memory request servicing, of which the DRAM access takes up the longest stage. According to the global memory coalescing rule implemented, multiple memory transactions may be caused per each request and multiple memory segments need to be transferred between DRAM and SMs, named uncoalesced memory access. In this case, the multiple transactions can be synthesized by one memory transaction with identical steps except a lengthened DRAM access stage due to the increased transferred memory segments. Thus, the global memory accesses can be serviced by the pipeline represented in Fig 1 and the global memory performance can be formulated via the pipeline throughput. The redefined memory throughput actually describes the frequency of memory accesses leaving the global memory.

The duration of each pipeline stage does not need to be equal but a guarantee should be made that DRAM access is the most inefficient among all pipeline stages and thus memory throughput is calculated as the reciprocal of DRAM access time of the synthesized memory transaction. The memory throughput is constraint by the global memory access patterns and partition camping.

For GPU with compute capability 2.0, it can be configured to enable L1 cache or not in SM through a compilation command `-Xptxas -dlcm`, and corresponded 32-byte or 128-byte transactions will be generated each with a different DRAM access time, denoted as $DRAM_{32B}$ and $DRAM_{128B}$. Let $\#partition$ and $comp_p$ each represents the number of memory partitions of the global memory and the number of clock cycles to execute a compute period, and $comp_{inst}$ and mem_{inst} represents the number of compute and memory instructions per each thread, $issue_{lat}$ denotes the Clock cycles needed to issue instructions to the SIMD pipeline while mem_{issue} denotes the Latency per memory transaction. Another two variables tpr_{32B} and tpr_{128B} each represents the number of 32-byte transactions and 128-byte transactions per each memory request. We also let DD represents the departure delay of the synthesized memory transaction .To put it together, we calculate the average DRAM access latency $DRAM_{lat}$ using equation 4. The compute throughput and memory throughput can be obtained using the following equations.

$$comp_p = \frac{comp_{inst}}{mem_{inst}} \times issue_{lat} \tag{1}$$

$$mem_{issue} = \frac{comp_p}{\#SM} \tag{2}$$

$$comp_{thr} = \frac{1}{mem_{issue}} = \frac{mem_{inst} \times \#SM}{comp_{inst} \times issue_{lat}} \tag{3}$$

$$DRAM_{lat} = \frac{DRAM_{32B} \times tpr_{32B} + DRAM_{128B} \times tpr_{128B}}{tpr_{32B} + tpr_{128B}} \tag{4}$$

$$DD = DRAM_{lat} \times (tpr_{32B} + tpr_{128B}) \tag{5}$$

$$mem_{thr} = \frac{\#partition}{DD} \tag{6}$$

3.2 GPU Program Classification

Based on the calculated compute throughput and memory throughput, the kernel throughput limited factors can be concluded and we have the following definitions:

- Compute-bound: it corresponds to the conditions where compute throughput is less than memory throughput, which means that the global memory requests arrive at the global memory interface at a relatively slow speed.
- Memory-bound: this category refers to the situation where the compute throughput is larger than the memory throughput, which means that memory requests arrive at the global memory more quickly than the leaving speed of previously arrived memory requests.

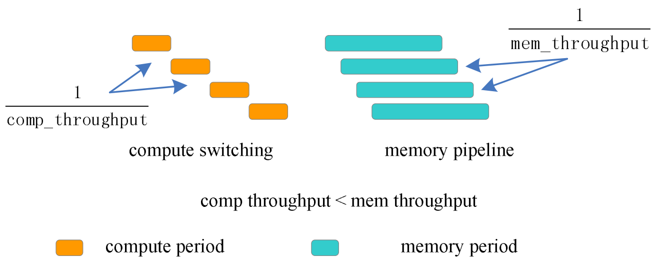
4 Analytical Performance Model

To illustrate how executing quantity of warps on SMs concurrently affects the total execution time, we will illustrate several scenarios covering both compute-bound and memory-bound cases. As the philosophy of the GPU architecture is to cover the long latency operations with interleaving execution of compute operations from a large amount of warps, the final performance is largely dependent on the effectiveness of latency hiding. The total execution time can be decomposed into two parts: duration of compute execution and uncovered memory latency.

4.1 Performance Prediction for Compute-Bound GPU Kernels

Due to a high compute-to-memory-access ratio or perfect global memory access coalescing, the compute throughput is larger than memory throughput, and memory requests can be handled at a faster speed than they arrive at the memory interface. Fig 2 shows an example of compute-bound kernels.

For case 1 in Fig 3a, we assume that each thread has only one memory access and thus one corresponding compute period per warp. Due to the relatively higher throughput of memory requests, the speed of memory requests handling is faster than the speed they are issued, and thus incoming memory requests will not accumulate latency to the final execution time. The resulting latency of case 1 in Fig 3a is 4 compute periods plus one memory period overhead.



$\frac{1}{\text{comp\_throughput}}$
$\frac{1}{\text{mem\_throughput}}$
compute switching
memory pipeline
comp throughput < mem throughput
compute period
memory period

Fig. 2. An example of compute-bound kernels

For case 2 in Fig 3b, there are four warps and each warp has two compute periods and two memory periods. The second compute period can start only after the first memory period of the same warp is finished. The compute throughput and memory throughput are the same as case 1. Since the computation latency is dominant, memory accesses do not contribute to the overall execution time which equals to the sum of 8 compute periods and only one memory period.

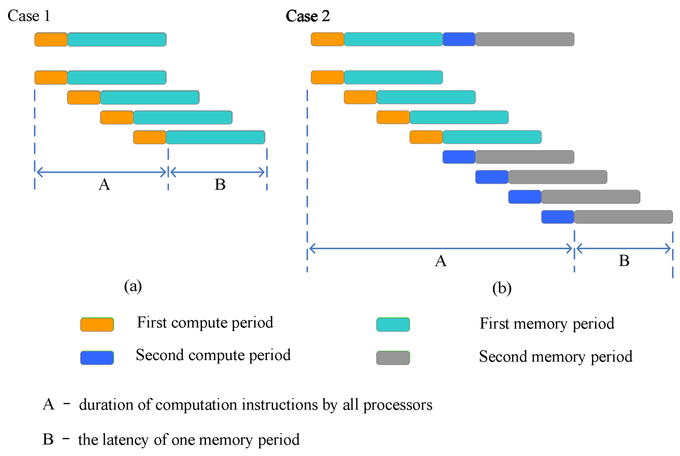


Fig. 3. Total execution time for the compute-bound GPU kernel

To be generally, the performance of compute-bound applications can be calculated using the following equations:

$$comp_{cycle} = \frac{comp_{inst} \times \#warp \times issue_{lat}}{\#SM} \tag{7}$$

$$exec_{cycle} = comp_{cycle} + mem_{lat} \tag{8}$$

where #warp represents the number of warps in a kernel which is defined by the kernel launch configurations and mem$_{lat}$ represents the latency of a synthesized memory transaction, as the value is not critical to the final performance, we constraint the latency to be 500 cycles.

4.2 Performance Prediction for Memory-Bound GPU Kernels

Figure 4 shows an example of memory-bound kernels where memory throughput is roughly a half of compute throughput. Equation 5 indicates that the departure delay between memory requests gets longer as more memory transactions are triggered for one memory request because of poor performance in memory coalescing. High throughput of computations will narrow down the interval of warp switching, and as a result, memory requests are issued more frequently to the global memory.

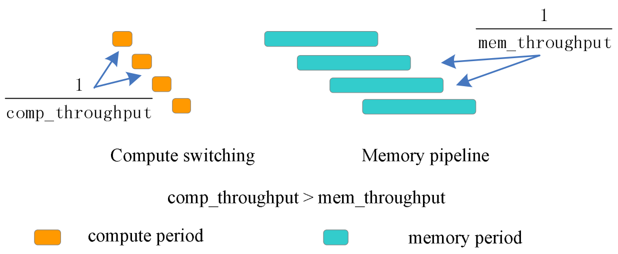


Fig. 4. An example of memory-bound kernels

For case 3 in Fig 5a, there are four warps and each warp has one compute period and one memory period. Since compute throughput is larger than memory throughput, memory access latency cannot be completely overlapped by computation, and thus each warp will accumulate extra latency of $(\frac{1}{mem_{thr}} - \frac{1}{comp_{thr}})$ cycles to the total execution time which equals to the sum of 4 compute periods and 4 extra latencies, which can also be represented as 4 departure delays plus one memory period and one compute period.

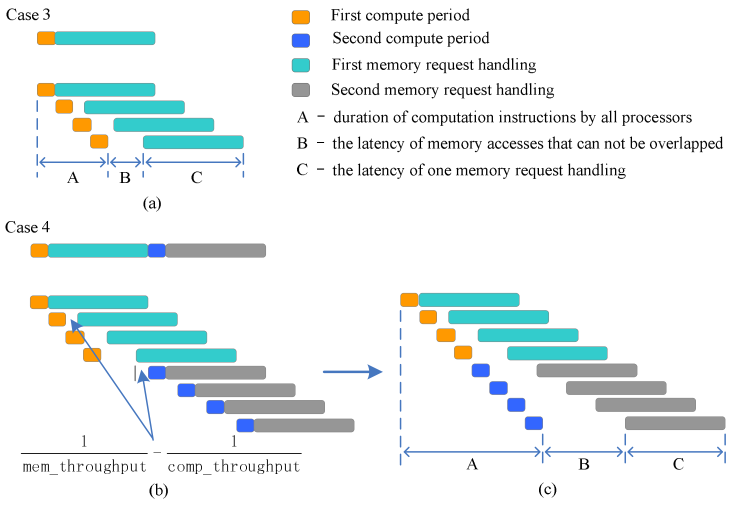

Case 3
First compute period
Second compute period
First memory request handling
Second memory request handling
A – duration of computation instructions by all processors
B – the latency of memory accesses that can not be overlapped
C – the latency of one memory request handling
A
B
C
(a)
Case 4
1
mem_throughput
1
comp_throughput
(b)
A
B
C
(c)

Fig. 5. Total execution time for memory-bound kernels

For case 4 in Fig 5b, there are four warps in each SM and each warp has two compute periods and two memory periods. The second compute period can start only after the first memory period of the same warp is finished. Compute throughput and memory throughput are the same as case 3. Even though idle cycles are introduced to the arithmetic pipeline, the execution time remains stable since the memory access time is dominant. The timing model of case 4 can be equivalently transformed as depicted in Fig 5c by moving the latter compute periods forward. As a result, the latency of memory accesses can only be partially overlapped by the computations. The final execution time is composed of three parts: parallel execution of compute instructions by all process units, uncovered overhead of memory requests and one memory period.

To be generally, the total execution time of memory-bound kernels can be calculated as the following two forms:

$$DD_{sum} = DD \times \#warp \times mem_{inst} \tag{9}$$

$$extra_{lat} = \frac{1}{mem_{thr}} - \frac{1}{comp_{thr}} \tag{10}$$

$$mem_{uncover} = extra_{lat} \times \#warp \times mem_{inst} \tag{11}$$

$$exec_{cycle} = comp_{cycle} + mem_{uncover} + mem_{lat} \tag{12}$$

or

$$exec_{cycle} = \frac{comp_p}{\#SM} + DD_{sum} + mem_{lat} \tag{13}$$

where DD represents the DRAM access time for a single transaction, DD_{sum} represents the overall DRAM access time for all memory transactions in the kernel. The content of $extra_{lat}$ points to the extra latency introduced by one memory access in memory-bound kernels. The $mem_{uncover}$ counts for the latency of global memory that cannot be hidden by computations. Equation 12 and Equation 13 have the same result but from different aspects. Equation 12 calculates execution time from the aspect of latency hiding while Equation 13 calculates execution time based on the memory access efficiency as memory accesses dominant.

5 Methodology

We conduct experiments on one NVIDIA Tesla M2050 GPU and the CUDA programming model, and we believe that the result of this work is still suitable for other chips and programming models as long as modifications are made to the value of input parameters.

To evaluate the effectiveness of our model, we predict performance for a manually written GPU benchmark and a commonly used kernel matrix-multiply.

5.1 Benchmark

The manually written benchmark we used contains 100 iterations, each consisting of one compute period and one memory period. The variable comp_iter controls the amount of compute instructions in a compute period, and the change of its value can simulate optimizations toward computation. Another variable *tran_per_req* presented as a parameter in the calculation of index indicates the number of memory transactions caused by each global memory access, and also its value can simulate optimizations toward memory access pattern. The variable index in the benchmark spreads the footprints of one memory request over multiple memory segments based on the value of *tran_per_req* and data type. For simplicity, single point float numbers are generated in the host CPU and transferred to the GPU global memory, and the generated compute instruction occupies one clock cycle each on Tesla M2050. The number of both the computation and memory access instructions is counted from the assembly code, which is obtained through the cuobjdump tool provided by NVIDIA. As can be concluded from the above assembly code, each iteration of the inner loop will generate 3 compute instructions and there are 7 other instructions in each iteration of the outer loop. The instruction LD.E performs 32 global memory load operations for 32 threads in a warp, which may result in multiple memory transaction according to the performance of memory access coalescing, that is, the value of *tran_per_req*.

5.2 Matrix-multiply

The matrix-multiplication is commonly applied in various applications. The shape of two input matrixes A[M*TILE] and B[TILE*M] are rectangular instead of square shape. The work load of each thread can be decomposed into several memory requests and plenty of computations per each request.

Figure 6 shows two cases of tiled matrix multiplication each corresponds to *TILE_WIDTH*=8 and *TILE_WIDTH*=32. For each iteration of the inner loop, the memory requests per warp of the case in Fig 6(a) consists of 4 addresses across four rows of A and 8 addresses along a row of B tile, which results in 4 32-Byte memory transactions to A and one 32-byte memory transaction to matrix B. The situation has been much improved when the tile width is 32 as each iteration of the inner loop only incurs one 32-byte transaction to A and 4 consecutive 32-byte transactions to B which can be combined into one 128-byte memory transaction, as we will show in the next section.

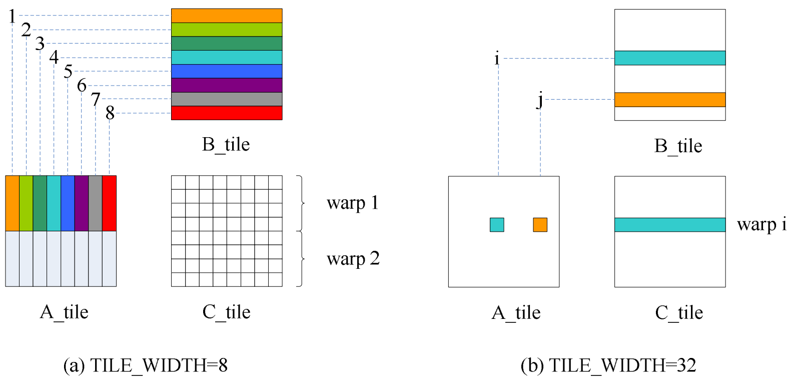


Fig. 6. Matrix multiply : (a) A_tile[8][8] ×B_tile[8][8], (b) A_tile[32][32] ×B_tile[32][32]

6 Experiment Results

6.1 Benchmark

As the address index in the manually written benchmark is carefully assigned that no repetition occurs in the loaded data for each thread, we bring out an experiment for both L1 cache enabled and disabled situations, each corresponds to the compilation command `-Xptxas -dlcm=ca` and `-Xptxas -dlcm=cg`, and 128-byte and 32-byte transactions are triggered. We gradually increase the spectrum of address requirement of a single warp to increase the number of memory transactions per each warp's request for a memory-bound program (with higher ratio of memory requests per computation) and the result is concluded in the Fig 7. When *tran_per_req* bellows 8, both the cached and uncached cases follow the same curve and it can be inferred that the 4 32-byte memory transactions are combined into a single 128-byte transaction, even in the uncached conditions. While the *tran_per_req* is above 32, the memory transactions caused by one warp's memory request will not increase. Otherwise, the execution time shows a linear growth to the value of *tran_per_req*, although each with a different value. Based on the memory-bound classification information, the increased latency of the kernel can be attributed to the reduction of the memory throughput, due to the increased memory transactions per each warp's memory request.

For both cached and uncached cases, the DRAM access time can be calculated by dividing the increment of kernel latency by the number of increased memory transactions. The calculated DRAM access latencies for 32-byte and 128-byte memory transaction of Tesla M2050 are 0.67 cycles and 1.53 cycles respectively. The number of instructions is counted in the assemble code.

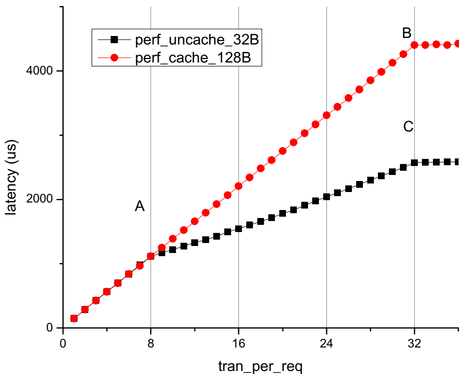


Fig. 7. Measurement of DRAM access time for 32-byte and 128-byte memory transaction

We measured kernel execution time under different compute throughput and memory throughput by varying the value of *comp_iter* and *tran_per_req*. The numbers of compute and memory instructions, as well as memory access pattern specified by the *tran_per_req*, are served as inputs to the performance prediction model. Table 1 lists parameters and predicted performance of two cases of our benchmark, and the results show that the error of prediction is no more than 6.5%.

Table 1. Applying the model for benchmark

Parameter	Case 1	Case 2	Description
comp_iter	100	20	Parameter
tran_per_req	4	8	Parameter
#warp	(64*256)/32=512	(64*256)/32=512	Warp number
$comp_{inst}$	31000	7000	Compute instruction
mem_{inst}	1*100	1*100	Memory instruction
$comp_{p}$	310	70	Instructions per compute period
mem_{issue}	310/14=22.1	70/14=5	Frequency of memory request issuing
tpr_{32B}	0	0	32-byte transaction per request
tpr_{128B}	4	8	128-byte transaction per request
$DRAM_{lat}$	1.53*4=6.12	1.53*8=12.24	Average departure delay
$comp_{thr}$	1/22.1=0.045	1/5=0.2	Compute throughput
mem_{thr}	1/6.12=0.163	1/12.24=0.082	Memory throughput
classification	Compute bound	Memory bound	Program classification
$comp_{cycle}$	1134393	256641	Execution time of compute instructions
$mem_{uncover}$	0	370688	Uncovered latency of memory accesses
$exec_{cycle}$	1184393	627829	calculated execution time
measured	1267534	642108	Measured execution time
Error	6.5%	2.2%	Prediction error

6.2 Matrix-multiply

We conduct experiments to predict the performance of matrix multiplication C=AB with different matrix scale as depicted in Fig 6.

The case in Fig 6a shows the calculation of a tile of matrix C where A of dimension 1024×8, B of dimension 8×1024, and C of dimension 1024×1024. Each element of C is assigned a thread and 1024×1024 threads are created. The matrix C is decomposed into multiple tiles and each tile contains 8×8 elements. As a result, the threads are organized as 128×128 blocks and each block contains 8×8 threads. Each block calculates the elements of a different tile in C based on a single tile of A and a single tile of B. The 64 threads in a block are organized into two warps, each of which calculates 4 rows of C_tile. As presented in the figure, each iteration of the inner loop will generate 4 32-byte transactions to load A_tile and one 32-byte transaction to load B_tile per warp, thus 40 32-byte memory transactions are generated for each warp.

For the case in Figure 6b, it shows another matrix multiplication C=AB where A of dimension 1024×32, B of dimension 32×1024, and C of dimension 1024×1024. For each warp, an iteration of the inner loop requires one element of A_tile and one row of B_tile, thus one 32-byte transaction of A and four 32-byte transactions of B will be generated. Due to the data locality of L2 cache, the unused element of the last accessed row of A_tile will be used by the next 7 iterations. The generated 4 32-byte transactions for a row data of B_tile can be combined into a 128-byte transaction. As a result, 4 32-byte transactions and 32 128-byte transactions will be generated.

The number of compute instructions per warp can be obtained from the assemble code. The calculated compute throughput and memory throughput is presented in Table 2. Surprisingly, the matrix multiplication code is specified as memory-bound according to our classification method.

Table 2. Parameters for matrix-multiply

	TILE_WIDTH=8	**TILE_WIDTH=32**
32B trans per warp	40	4
128B trans per warp	0	32
Avg_dep_delay_per_req	(40*0.67)/40=0.67	(4*0.67+32*1.53)/36=1.43
Comp_inst	12*8+26=122	12*32+26=410
Mem_req_issue_dist	122/(40*14)=0.218	410/(36*14)=0.813
1/Comp_throughput	0.218	0.813
1/Mem_throughput	0.67	1.43
Program classification	Memory-bound	Memory-bound

To verify the effectiveness of program classification, we manually add multiple compute instructions in the inner loop by increasing the value of *comp_iter* to the case in Fig 8b. As presented in the Fig 8, the overall latency of the kernel starts to rise at a point around 6 along the x-axis, which means that the kernel is not bounded by the compute operations before that point. We also calculate an expected point at which compute throughput equals to memory throughput, and the result turns out to be *comp_iter*=7, which is pretty close to the measured value.

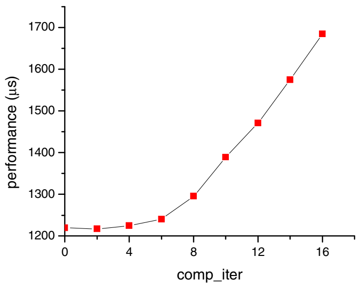


Fig. 8. Verification of program classification for A[1024][32] ×B[32][1024]

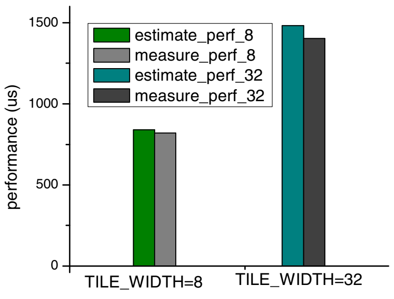


Fig. 9. Comparison of estimated and measured latency for two cases

6.3 HotSpot and Gaussian Elimination

We also applied our analytical model to another two GPU programs: HotSpot and Gaussian Elimination, both from the Rodinia benchmark suits [13], which are specifically developed for the GPU-accelerated heterogeneous systems.

HotSpot is an ordinary differential equation solver used in simulating microarchitecture temperature. Every element is computed as a function of 3*3 neighborhood of elements from the input array (as stencil). For each thread's one element computation, 9 elements need to be loaded into the processor unit and thus heavy stresses are assigned to the global memory bandwidth. However, the Hotspot in the version Rodinia 2.4 is optimized throughput caching, by way of utilizing shared memory to store neighborhood data so it can be reused among neighboring threads in the same thread block. As the shared memory accessing has the identical latency as normal compute instruction, shared memory access instructions is treated as compute instructions. There are only two global memory loads and one global memory store instruction in each iteration of a thread, and all three global memory accesses are coalesced due to the shared memory. We calculate the values of compute throughput and memory throughput in the assembly code and the kernel calculate_temp turns out to be compute-bound. We estimate the execution time of all kernel runs using equation (8) and compare the results with the measured latency as listed in Table 3. The input data are also provided in the benchmark suit.

Table 3. Benchmark result for HotSpot

Input size	Measured (s)	Predicted (s)	Error
64	0.021	0.017	19.05%
512	0.040	0.035	12.5%

Gaussian Elimination solves systems of equations using the Gaussian elimination method and contains multiple iterations of two kernels: Fan1 and Fan2. the algorithm must synchronize between iterations, but the values calculated in each iteration can be computed in parallel. For both kernels, parameterized size-strided-accesses to matrix a_cuda and m_cuda lead to uncoalesced accesses which result in tremendous global memory transactions. In the L1 cache-enabled case, 32 128B-memory-transactions will be incurred while the value of size above 32. According to GPU program classification method presented above, the two kernels are defined as memory-bound and we estimate the execution time using equation (12). A comparison of measured and predicted execution time is shown in the Table 4.

Table 4. Benchmark result for Gaussian Elimination

Input size	Measured (s)	Predicted (s)	Error
16	0.000410	0.000324	20.94%
64	0.001643	0.001540	6.27%
512	0.063546	0.056340	11.33%

As can be seen from both benchmarks, the estimated values tend to be constantly smaller than the actual execution time. The inaccuracy in the projected performance can result from various sources, such as synchronization, kernel initialization, CPU execution of loop control instructions, etc. In the following work, all those factors will be considered in our performance model.

7 Conclusion

In this paper, we propose a throughput-aware analytical performance prediction model for the GPU applications. We predict performance of GPU kernels based on the throughput determined by the compute throughput and memory throughput redefined in the paper. Experiment results illustrate high accuracy of our performance prediction model in capturing impaction of performance bottlenecks such as control flow divergence and uncoalesced memory access.

We believe our model has captured the GPU's primary performance factors, and it can provide some useful hints in the future performance optimization. Our work has several limitations that we hope to address in future research: (1) model the cost of double-precision computations and other complex operations, (2) figure out an upper bound of performance based on the model research, (3) automatic memory transaction number detection, (4) model the synchronization barrier's effect on warp-level parallelism.

References

1. Keckler, S.W., Dally, W.J., Khailany, B., et al.: GPUs and the future of parallel computing. IEEE Micro 31(5), 7–17 (2011)
2. Advanced Micro Devices, Inc. AMD Brook+
3. NVIDIA Corporation. CUDA Programming Guide, Version 4.0
4. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in Science & Engineering 12(3), 66 (2010)
5. Owens, J.D., Houston, M., Luebke, D., et al.: GPU computing. Proceedings of the IEEE 96(5), 879–899 (2008)
6. Lindholm, E., Nickolls, J., Oberman, S., et al.: NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro 28(2), 39–55 (2008)
7. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM 52(4), 65–76 (2009)
8. Zhang, Y., Owens, J.D.: A quantitative performance analysis model for GPU architectures. In: 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA), pp. 382–393. IEEE (2011)
9. Hong, S., Kim, H.: An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. ACM SIGARCH Computer Architecture News 37(3), 152–163 (2009)
10. Baghsorkhi, S.S., Delahaye, M., Patel, S.J., et al.: An adaptive performance modeling tool for GPU architectures. ACM Sigplan Notices 45(5), 105–114 (2010)
11. Meng, J., Morozov, V.A., Kumaran, K., et al.: GROPHECY: GPU performance projection from CPU code skeletons. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, p. 14. ACM (2011)
12. Cui, Z., et al.: An accurate GPU performance model for effective control flow divergence optimization. In: 2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS). IEEE (2012)
13. Che, S., Boyer, M., Meng, J., et al.: Rodinia: A benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization, IISWC 2009, pp. 44–54. IEEE (2009)