Performance Modeling for Optimal Data Placement on GPU with Heterogeneous Memory Systems

Yingchao Huang
University of California, Merced
yhuang46@ucmerced.edu

Dong Li
University of California, Merced
dli35@ucmerced.edu

***Abstract*—A heterogeneous memory system (HMS) consists of multiple memory components with different properties. GPU is a representative architecture with HMS. It is challenging to decide optimal placement of data objects on HMS because of the large exploration space and complicated memory hierarchy on HMS. In this paper, we introduce performance modeling techniques to predict performance of various data placements on GPU. In essence, our models quantify and capture implicit performance correlation between different data placements. Given the memory access information and performance of a sample data placement, our models predict performance for other data placements based on the quantified correlation. We reveal critical performance factors that cause performance variation across data placements. Those factors include instruction replay, addressing mode, hardware queuing delay of memory requests, off-chip memory access latency, and caching effects. Those factors, which are often not sufficiently considered in the existing performance models, can significantly impact modeling accuracy. We introduce a series of techniques to model those factors. We extensively evaluate our models with a variety of benchmarks with various data placements. Our models are able to quantitatively predict the benefit or performance loss of data placements.**

I. Introduction

A heterogeneous memory system (HMS) consists of multiple memory components with different properties (e.g., bandwidth, latency, memory organization, preferable access patterns, and programming paradigms). HMS has the potential to provide performance benefits for various workloads; it can strike a good balance between multiple design goals (e.g., cost, area size, and performance). With the growing demands for efficient data accesses, HMS is becoming more and more common [1], [2], [3].

GPU is a representative architecture with HMS. An NVIDIA Kepler GPU, for example, has more than eight types of memory components (global, texture, shared, constant, and various caches), with some on-chip, some off-chip, some directly manageable by software, and some not. Recent studies [4] reveal that many GPU kernels manifest significant performance difference with different data placements on HMS. The performance difference is up to 208% (159% on average). Recent studies [5] also show that many memory-intensive GPU applications carefully written by developers cannot even reach half of the best performance achievable by exhaustive searches for optimal data placement. These studies all indicate that deciding optimal data placement on GPU with HMS is critical for performance optimization.

However, given GPU with HMS, it is challenging to decide optimal placement of data objects. First, a GPU kernel can easily have many data objects, each of which can have multiple data placement options. In theory, to decide data placement of n data objects on m programmable memory components ($m = 4$ on GPU), there are m^n possible data placements, subject to the limitation of memory capacities and read/write properties of memory components. Exploring the optimal data placement in such potentially large exploration space is a challenge. Second, caching mechanisms further complicate the data placement problem. Different memory components can use shared or separate cache hierarchy; They can also employ different caching policies (e.g., the 2D spacial locality for texture memory and LRU-like policy at L2 cache for off-chip memories). Moving data objects from one memory component A to B have non-trivial impact on the data caching of A and B. Third, different data placements have different implications on instruction issues and warp interleaving (i.e., the overlapping between computation and memory access). Quantifying and predicting the above implications is a challenge.

In this paper, we introduce performance modeling techniques to predict performance of various data placements on GPU. We focus on the data placement problem for those programmable memories (i.e., global, shared, texture, and constant memories). Given a GPU kernel, our models predict performance without the necessities of developing many versions of a CUDA kernel to implement different data placements. This allows programmers to easily explore the large search space and find the optimal data placement. Hence, our models can work as a tool to help programmers for GPU performance optimization and improve their productivity.

In essence, our models quantify and capture implicit performance correlation between various data placements. Given the memory access information and performance of a sample data placement, our models predict performance for other data placements based on the quantified correlation. We model the performance correlation between different data placements from three perspectives: computation cost, memory cost, and performance overlapping because of multi-threading.

First, the computation cost varies across data placements because of addressing mode difference and instruction replay. For each memory component, we characterize its addressing mode, and then quantify the difference in the number of instructions between different memory components. The instruction replay is another important factor that causes computation cost variation across data placements. The instruction replay happens when specific hardware events happen, such as share memory bank conflict and global memory address divergence. The instruction replay occupies extra instruction issue slots and impacts compute throughput of GPU. We reveal that the number of issued instructions including instruction replays is a better performance indicator to the computation cost than the number of executed instructions commonly employed in the existing performance models [6], [7].

Second, the memory cost varies across data placements because of the variation of caching effects and average memory access latency. As one changes data placement from one memory component to another, the cache interference and cache bandwidth consumption are affected, which impact performance. Furthermore, changing data placement will impact

the access latency of those off-chip memory accesses. The existing performance models assume a constant access latency for any off-chip memory accesses without caches [6], [7], [8], [4]. However, our performance analysis on a real GPU hardware reveals that the off-chip memory accesses without caches can have up to 110% difference in the access latency, depending on whether the memory accesses hit row buffers in memory banks. Changing data placement has impact on row buffer hit/miss, and hence impact the access latency of those off-chip memory accesses. Our evaluation shows that capturing the *large* variation of the off-chip memory access latency is critical for modeling accuracy, even for those GPU kernels with large memory-level parallelism to hide the off-chip memory access latency. Furthermore, we introduce a queuing model of the off-chip memories to capture memory request delays because of shared resource contention. The queuing model is highly composable and flexible, allowing us to model the combination of diverse memory systems. Our queuing model-based approach in combination with the modeling of row buffer hit/miss provides a more accurate estimation of the off-chip memory access latency. This effectively improves the modeling accuracy by 13.8% on average, comparing with the model assuming a constant off-chip memory access latency.

Third, the performance overlapping varies across data placements because of the variation of memory events that cause stall cycles. To model the overlapping time, we introduce an empirical model based on critical memory events.

Based on the above modeling techniques, we develop a framework that uses a sample data placement as input and predict performance for various data placements. The major contributions of our work are as follows.

- We develop performance models to predict performance for various data placements. Our modeling works as a tool for performance optimization for GPU with HMS, and provides foundation to explore other HMS systems.
- We reveal the critical performance factors that cause performance variation across various data placements. Those factors include instruction replays, addressing mode, hardware queuing delay of memory requests, off-chip memory access latency, and caching effects. Those factors, which are often not sufficiently considered in the existing performance models, can significantly impact modeling accuracy.
- We extensively evaluate our modeling with various data placements. We demonstrate that our models are able to quantitatively predict the benefits or performance loss of data placements. Our models demonstrate the tangible benefits of using a model-driven approach for performance optimization on HMS.

II. Background and Motivation

In this section we review the heterogeneous memory system on GPU and motivate our model construction.

A. GPU Memory Background

A typical NVIDIA GPU has four types of programmable memory components, including global, shared, texture, and constant memories. Deciding data placement on them is the focus of this paper. Global, texture, and constant memories are off-chip, and based on GDDR DRAM (GDDR3 or GDDR5), while shared memory is on-chip.

Texture memory and constant memory have their own caches. Texture, constant, and global memories share a last-level L2 cache distributed over multiple streaming multiprocessors (SM) on GPU.

Like regular DRAM devices, off-chip GDDR DRAM is hierarchically organized as channels, modules, ranks, and chips. On GPU, each channel has one memory rank that consists of multiple DRAM chips. Each chip in a rank is organized into a number (4, 8, or 16) of logically independent banks. A memory bank is a 2D array of cells organized into rows and columns. Banks can operate in parallel, providing bank level parallelism in accessing memory. Memory reads and writes are performed by selecting the location of a memory bank according to the row and column addresses.

For any memory request, a row of data is first read into a row buffer associated with each bank. If the request is to a currently open row (i.e., a *row buffer hit*), then the data is directly serviced from the row buffer. If the request is not to a currently open row (i.e., a *row buffer miss*), then the memory controller has to write back data in the open row and fetch a new row of data into the row buffer, which causes longer access latency.

We use a Tesla K80 (an NVIDIA's Kepler architecture) as an example throughout this paper, but our general modeling methodology is applicable to other GPUs with programmable memories. Figure 1 shows the memory system of a typical Kepler based on the above background information. Even though it is possible to extend our models to handle different types of data structures, our work focuses on the placement of data arrays, similar to [4], [9], because the data array is the most common data structure in GPU programming.

Given a GPU kernel to optimize the placement of its arrays, we call the kernel's existing data placement the *sample data placement*. We pick a data array as the *target data object*, and then predict the kernel performance if we move the array from the sample data placement to a new data placement. This new data placement is the *target data placement*. We use the above terms in the rest of the paper.

When we change the placement of an array, we assume that the dimension of the array in the target data placement remains the same as that in the sample data placement. When changing data placement, we do not consider those performance optimizations and any algorithm optimization in the target data placement (e.g., avoiding bank conflict and improving memory access coalesce). Exploring those optimizations for a specific data placement is available in the existing work (e.g., [10] for global memory coalesce and [11] for shared memory bank conflict). Our work aims to identify a promising data placement out of a large search space for performance optimization, and the existing work can be employed to further improve the performance of this promising data placement. Hence, our models work as a *performance advising tool* for HMS. Furthermore, if we assume optimized performance (e.g., optimized memory access patterns) in the target data placement in our models, our models would be misleading, because there could be an unbridgeable gap between the desired memory access patterns and GPU kernel implementation. In fact, the existing tool [4] uses the same strategy as our models, and has demonstrated success in performance advising.

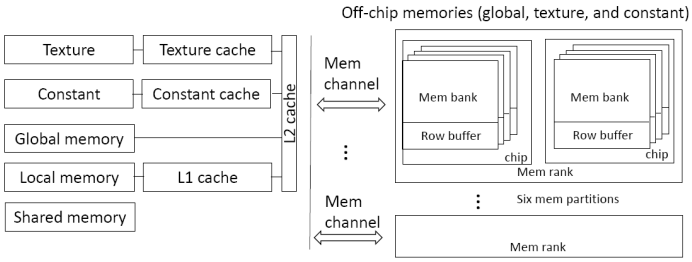


Fig. 1: The heterogeneous memory system of a Kepler GPU

B. Initial Observations for Model Construction

To gain some intuitions on how to build performance models for optimal data placement, we run six benchmarks (convolution, matrixMul, md, spmv, transpose, and cfd) with six GPU kernels, and test 34 data placements. These benchmarks and the data placement tests are summarized in our technical report [12]. For each data placement, we use nvprof to collect 265 performance events. Across those data placements, we attempt to identify those performance events that are the most sensitive to the change of data placement. Those events provide critical indications for the model construction.

Given a large amount of performance data, identifying meaningful performance events for performance modeling is challenging. We use a metric, the cosine similarity [13], to identify those sensitive performance events. The cosine similarity is a measure of similarity between two vectors. The cosine similarity of two vectors is bounded to [0,1]. with 1 for strong correlation and 0 for otherwise.

For a GPU kernel with N data placements, we construct a vector with the execution times of the N data placements as elements (i.e., the vector length is N). We call this vector the time vector. Furthermore, we construct 265 vectors, each of which corresponds to a performance event; Each vector uses the measured performance event for the N data placements as elements (i.e., each vector length is N). We call those vectors the performance event vectors. We calculate the cosine similarity between the time vector and each performance event vector to capture the correlation between the execution time and each performance event. A large cosine similarity value for a performance event indicates that the variation of the performance event closely follows the variation of the execution time.

We use 0.94 as a cosine similarity threshold, and then select those performance events that have cosine similarity values larger than 0.94. 0.94 is chosen as the threshold, such that at least two performance events are selected for each GPU kernel to ensure that the selected performance events are common across kernels.

After careful evaluation of alternatives, we choose 36 performance events. We further aggregate them, because some of them have the similar indications for performance modeling. For example, `L2_L1_write_transaction` and `L2_L1_read_transaction` are aggregated as `L2_L1_transaction`. We also try to capture those common performance events, and remove those events that only appear in two kernels, because those events may not be able to serve as a *general* indicator for performance modeling. The aggregation process reduces all performance events to five performance events, shown in Table I.

Based on Table I, we find the following events strongly correlated with the performance variance across data placement.

TABLE I: The cosine similarity values for five representative performance events. "N/A" means the value smaller than 0.94.

GPU kernel	issue slots	inst_ issued	inst_ integer	ldst_ issue	L2_ trans
cfd	0.993	0.992	0.961	0.946	0.941
convo1	N/A	N/A	N/A	0.949	0.986
convo2	0.949	N/A	0.969	0.957	0.996
md	0.956	0.963	0.963	N/A	0.995
matrixMul	0.980	0.977	0.943	0.990	N/A
spmv	0.983	0.989	N/A	N/A	N/A
transpose	0.992	0.99	N/A	0.980	0.948

(1) **The number of *issued* instructions** including those replayed instructions due to special hardware events (e.g., instruction cache misses and constant cache misses). The event `issue_slots`, affected by the instruction replays, also indicates the importance of the number of issued instructions for performance modeling.

(2) **The number of integer instructions.** The correlation between this event and the performance variance is especially pronounced in those data intensive workloads (e.g., md and convo2). Because the computation remains constant across different data placements, most of the variation of the number of integer instructions comes from the addressing mode difference between memory components (See Section III-B for a detailed discussion).

(3) **The number of memory transactions seen at L2.** Different memories on GPU have different caching mechanism. Placing data in different memory components impact data caching in L2, which in turn impacts performance.

We construct performance models to capture the above critical performance events, and predict performance for different data placements.

III. Performance Models

In this section, we discuss our models in details. Some model parameters are summarized in Table II.

A. General Description

The execution time of a GPU kernel consists of three components, T_{comp}, T_{mem}, and $T_{overlap}$, shown in Equation 1. Three components capture the computation cost, the memory cost, and the overlapping between the two costs because of multithreading effects of GPU. Given a GPU kernel to optimize its data placement, we measure and profile T_{comp}, T_{mem}, and $T_{overlap}$ of the sample data placement, based on which we predict T_{comp}, T_{mem}, and $T_{overlap}$ of target data placements to decide the best data placement.

$$T = T_{comp} + T_{mem} - T_{overlap} \quad (1)$$

We significantly extend the existing work [7] to calculate T_{comp} and T_{mem}. In particular, the calculation of T_{comp} is based on the quantification of the issued instruction variation across different data placements; the calculation of T_{mem} investigates the long access latency of off-chip memory components, and introduces queuing theory-based models to capture bank level parallelism and hardware queuing delays in the memory system. To make the paper self-contained, we list those models that come from the existing work [7] but employed in this paper in Appendix. This section only contains our models, unless we declare otherwise.

$T_{overlap}$ happens when a warp doing computation overlaps with another warp issuing a memory request and waiting for the requested data. $T_{overlap}$ is highly relevant to the memory events that cause idling cycles in a warp. We build a model

TABLE II: Summary of some model parameters

Model parameter	Definition	Source
T_{comp}	Computation cost	Equation 1
T_{mem}	Memory cost	Equation 1
$T_{overlap}$	Overlapped cost	Equation 1
W_{serial}	Serialization overhead	Equation 2 and Appendix
#inst	Total issued instructions per warp	Equation 2
AMAT	Average memory access latency	Equation 4

for $T_{overlap}$ to capture the relationship between $T_{overlap}$ and those memory events.

B. The Calculation of T_{comp}

Calculating T_{comp} is based on Equation 2 in [7]. In the equation, W_{serial} is the serialization overhead excluding the serialization caused by memory events, such as shared memory bank conflicts. The calculation of them can be found in Appendix.

$$T_{comp} = \frac{\text{\#inst} \times \text{\#total\_warps}}{\text{\#active\_SMs}} \times \text{Effective\_instruction\_ throughput} + W_{serial} \quad (2)$$

To calculate T_{comp} for a target data placement, the major challenge is to estimate #inst. Different from the existing modeling work using the number of executed instructions [6], [7], we estimate the number of issued instructions, because the number of issued instructions is a strong indicator on the performance variation across data placements, as discussed in Section II-B. To estimate #inst, we quantify the instruction difference between the sample and target data placements.

To study where the difference in the number of issued instructions between data placements comes from, we use nvprof to collect the number of issued instructions for 10 benchmarks (convolution, spmv, md, matrixMul, transpose, bfs, triad, reduction, md5hash, and stencil2d) with 42 data placements (see [12] for data placement details). Our study shows that among 32 data placements (we exclude 10 default data placements as the baseline for calculating instruction difference), 23 of them have large variations in instruction replays and integer instructions. These large variations account for at least 75% of the total instruction differences between data placements. The integer instruction variation mostly comes from the addressing mode difference between memory components. 5 of 32 data placements have small variation (less than 20%) in instruction replays and integer instructions, but the implementation of these data placements have relatively big algorithm changes upon the implementation of the default data placement. Those changes introduce a lot more instructions than the variation of instruction replays and integer instructions.

In conclusion, to quantify the difference in the number of issued instructions between data placements, we must quantify: (1) replayed instructions; and (2) those instructions for data addressing (those instructions do not have replays).

Quantification of the variation of executed instructions (no replay). To investigate how the addressing mode changes between different data placements, we examine CUDA code and the corresponding SASS code (a native ISA for NVIDIA GPU). Figure 2 shows an example GPU kernel doing vector addition (single-precision floating point typed) with four data placements for two arrays (a and b in the figure). In this example, across the four data placements, we do not change the vector addition algorithm, aiming to set up a straightforward one-to-one mapping between the four data placements.

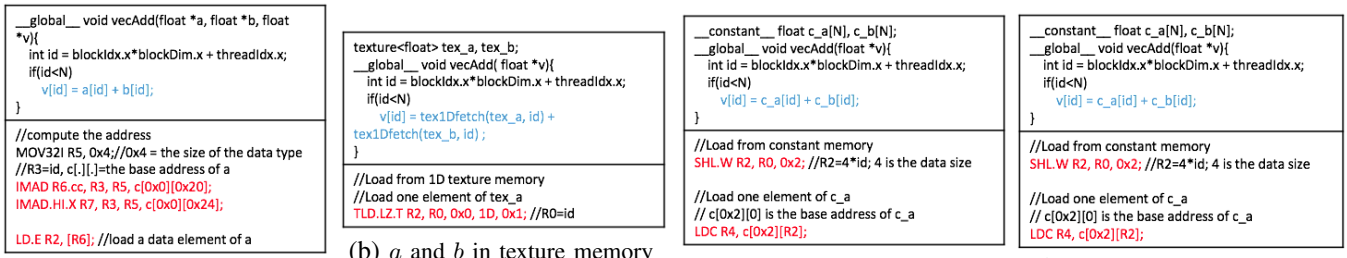
The figure reveals that the addressing mode changes between data placements. But, after data is loaded from a memory component to a register, the computation remains the same in all data placements, and there is no instruction difference afterwards. Also, different memory components use different load/store instructions (e.g., LDS for shared memory and LDC for constant memory), but those load/store instructions do not change the number of executed instructions across the four data placements. There is also an initialization phase for certain memory components before the data is ready to access. But the initialization phase happens only once. For the shared memory, the initialization phase copies data between global memory and shared memory. We can directly estimate the data copy time based on memory bandwidth and data size instead of counting instructions.

In this example, the addressing mode is particular for referencing an array element using the element index. The address mode includes the calculation of the effective address of an array element by using the element index and other information (e.g., the base address of the array) in registers or specific memory regions. Other methods exist to reference an array element (e.g., directly using the virtual address of the element instead of using the element index). However, using the element index is the most common method in GPU code. Our investigations on 12 SHOC benchmarks [14] and 4 CUDA SDK benchmarks (62 kernels in total) reveal that all of them use the element index to reference array elements. This is different from regular CPU programs that often employ flexible and complex methods to reference an array element.

We compare the addressing mode of the four memory components. To address an array element, some instructions have to be introduced to transform the element index into a new index or an actual data address to reference the data, depending on which memory component is used to place data. Global memory uses the register indirect addressing mode, and needs to transform the array index to the actual data address. On a 64-bit address space of the Kepler architecture, the actual memory address is calculated with 2 instructions using 32-bit registers. Constant and shared memories, however, use the indexed absolute addressing mode and only need one instruction to calculate a new index based on the array index and the base address of the array. Obtaining the base address for constant and shared memories does not consume any instruction, because the base address is pre-determined in a fixed space of the constant or shared memory (e.g., c[0x2][0] in the figure). Texture memory also uses the indexed absolute addressing mode. However, the texture memory may or may not use extra instructions to calculate a new index. For a 1D array on 1D texture memory in this example, the element index can be directly used as the index to reference the element without any extra instruction.

As a summary, the numbers of instructions required to calculate the address of a 1D-array element (single-precision floating point) are 2, 0, 1, 1 for global, 1D texture, constant, and shared memories, respectively. We enumerate and analyze common data types (double-precision floating point and integer) for the four memories and use that to quantify the variance of executed instructions because of the addressing mode changes between data placements.

Discussion: The above example is ideal for analyzing the addressing mode difference, because the example does not change the vector addition algorithm across the four data



(a) a and b in global memory (b) a and b in texture memory (c) a and b in constant memory (d) a and b in shared memory

Fig. 2: CUDA code and SASS code for four data placements for vector addition ($v = a + b$). We change the placement of input data vectors a and b. v is always in global memory. The red texts indicate the major difference in SASS code between data placements. Blue texts indicate the major difference in CUDA code between data placements.

placements. However, to implement a data placement, sometimes the algorithm change is inevitable. This is especially true when changing data placement between shared memory and other memory components, because shared memory is scoped to each thread block, while the other memory components are scoped to all thread blocks. The algorithm change will introduce new instructions and highly depend on application details. To build a general model, we do not consider those algorithm changes. Instead, in our models, when we change data from shared memory to other memories, we assume that the array index in shared memory is replaced with a global thread ID; when we change data from other memories to shared memory, we conservatively assume no extra instruction change, besides the above addressing mode difference.

In general, to compute the difference in the number of executed instructions between different data placements, we first identify those instructions addressing elements of the target data object in the sample data placement. Then we calculate the instruction difference between the sample and target data placements based on the above analysis of addressing mode.

Quantification of instruction replays. Instruction replays consume instruction issue slots reducing the compute throughput of SMs; instruction replays also increase the number of issued instructions. Hence, instruction replays has strong correlation to the performance variation across data placements. Many reasons cause instruction replays. Based on [15] and hardware counter events related to instruction replays, we summarize the reasons for instruction replays as follows:

1) Global memory address divergence, or when total number of words accessed by all threads in a warp exceed the number of words that can be loaded in one cycle;
2) Constant cache misses;
3) Address divergence in an indexed constant load;
4) Bank conflict in load/store for shared memory;
5) Complicated instructions issued over 2 cycles (including DFMA, DADD, DFMA, and DMUL);
6) Address conflict in a reduction or atomic operation;
7) L1 cache misses due to local memory accesses (e.g., register spill and stack data);
8) Instruction cache misses;
9) Address divergence in a local memory load or store;
10) Load/store buffer (LSU) full.

The instruction replays in (1)-(4) listed as above are directly related to memory references in the four memory components. As we move a data object from one memory component A to B, we reduce instruction replays for accessing A, but add instruction replays for accessing B. So we must count the number of instruction replays for referencing the target data object on both A and B.

To count the number of instruction replays for accessing a specific memory component, we collect the instruction trace of the sample data placement. Then, when an instruction references the target data object, we reason the instruction replays in (1)-(4) based on the analysis of memory references in each warp and cache models (see Section IV for details).

In particular, for (1), when analyzing a specific load or store instruction, we count the total number of words for all threads in a warp, and then divide the number by memory transaction size. Then, we use the result minus 1 as the number of replayed instructions. For (2), we count constant cache misses, and one constant cache miss adds one instruction replay. For (3), we count the number of constant memory address divergences in load instructions for all threads in a warp. Each memory divergence adds one instruction replay. For (4), we count the number of bank conflicts in shared memory, and one bank conflict adds one instruction replay.

For (5)-(10), we assume that the sample and target data placements have the same number of instruction replays. For (8), this assumption can introduce some inaccuracy when counting instruction replays, because the sample and target data placements have different number of executed instructions. A solution would be to build an instruction cache simulator and estimate the instruction cache misses when analyzing the instruction trace. However, we do not include it in our implementation to strike a balance between the model complexity and accuracy.

Based on the above discussion, the number of replayed instructions is calculated in Equation 3. In the equation, $\text{inst\_replay}_{\text{target}}$ and $\text{inst\_replay}_{\text{sample}}$ are the number of instruction replays per SM for the target and sample data placements, respectively. $\text{inst\_replay}_{\text{sample\_1-4}}$ and $\text{inst\_replay}_{\text{target\_1-4}}$ are the number of instruction replays due to (1)-(4) for the sample and target data placements, respectively.

$$\text{inst\_replay}_{\text{target}} = \text{inst\_replay}_{\text{sample}} - \text{inst\_replay}_{\text{sample\_1-4}} + \text{inst\_replay}_{\text{target\_1-4}} \quad (3)$$

C. Calculation of T_{mem}

The calculation of T_{mem} is based on Equation 4 in [7]. The calculation of effective memory requests per SM in Equation 4 can be found in Appendix. The effective memory requests include the memory requests to all of the four memory components. Counting them should consider the difference in memory request size between those memory components [15].

$$T_{mem} = \text{Effective\_memory\_requests per SM} \times \text{AMAT} \quad (4)$$

$$\text{AMAT} = \text{DARM}_{\text{lat}} \times \text{miss\_ratio} + \text{hit\_lat} + \text{shmem}_{\text{lat}} \times \text{shmem}_{\text{ratio}} \quad (5)$$

The calculation of AMAT (average memory access latency) for Equation 4 is shown in Equation 5, which requires miss_ratio, hit_lat, DRAM_{lat} for off-chip memories, and $\text{shmem}_{\text{lat}}$ and $\text{shmem}_{\text{ratio}}$ for shared memory. miss_ratio accounts for the L2 cache misses for global, constant, and texture memories. $\text{shmem}_{\text{ratio}}$ accounts for the shared memory requests. hit_lat is the cache hit latency (constant, texture and L2 caches). The previous work [16], [17] reveals that different GPU caches have different access latencies. We could extend Equation 5 to consider the latency difference. We omit the difference and assume the same cache access latency (L2 cache latency) in our model. This has very limited impact on modeling accuracy while simplifying the model. The same method has been used in [7].

DRAM_{lat} is the access latency for off-chip DRAM, and it is the most expensive one comparing with other memory access latencies. Previous work assumes that each memory request has the same DRAM access latency, and measures it through micro-benchmarks [16], [17]. However, memory requests can be queued at memory controller in bank-specific queues until the bank becomes available [18]. The queuing delay adds extra latency into the memory request. For those memory intensive workloads, GPU can especially increase the queuing delay of those memory requests than CPU, because of massive memory requests from massive thread-level parallelism. Furthermore, the DRAM access latency is affected by row buffers in memory banks. Our evaluation (see Section III-C2) reveals that there is up to 110% variation in DRAM access latency, depending on whether a row buffer miss/hit happens.

Altogether, the queuing delay and row buffer hit/miss cause the variation of the DRAM access latency across memory requests. As shown in the evaluation section, assuming a constant DRAM access latency can result in 13.8% higher prediction error (on average) than otherwise. To model the effects of the queuing delay and row buffer miss/hit, we employ a queuing model to calculate the average DRAM_{lat}, and do not assume a constant DRAM access latency.

1) T_{mem} Model Overview: Our queuing model considers GDDR5 of GPU with M memory controllers ($M = 6$ for Kepler and Fermi and six memory partitions in total) as a set of queues, shown in Figure 3. Following the standard of GDDR, each controller has one rank consisting of B banks. Each memory bank works as a server (shown as a bank server in Figure 3), and also has a specific queue (a G/G/1 queue discussed in Section III-C3) before the server. A memory request, after the last level cache, is distributed to a memory bank. If the memory request cannot be serviced by the memory bank immediately, the memory request is placed into the queue associated with the memory bank. When the memory request is serviced, the service time depends on whether the requested data is in the row buffer of the memory bank (a hit) or not (a miss). Memory banks can operate in parallel, depending on the amount of parallelism existing in memory requests.Figure 3 shows a conceptual description of the queuing model.

For a specific memory bank i, the average memory access latency includes the average queuing delay and the average service time (see Equation 6). We apply the queuing theory to calculate the average queuing latency (Section III-C3),

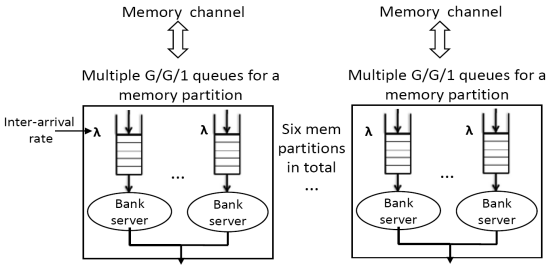


Fig. 3: A conceptual description of our queuing model.

and estimate the average service time based on row buffer miss analysis. The system-wide average DRAM access latency (DRAM_{lat}) is calculated based on the average memory access latency for NB memory banks, shown in Equation 7. In the equation, λ_i is the memory request arrival rate for the bank i. To calculate λ_i, we must know how memory requests are distributed to memory banks. We discuss it in Section III-C2.

The queuing theory has been applied to study memory system performance on CPU [19], [20] (Equation 7 is also employed in [19]). However, our work is significantly different from them from two perspectives: First, we reveal that the queuing models based on the exponential distribution, which have been applied on CPU, cannot be applied to GPU. We introduce a new queuing model customized to the workload characteristics of GPU. Second, we introduce a unique mechanism to *detect* the distribution of memory requests between memory banks, while the existing work assumes that memory requests have the same possibility to distribute on all memory banks, ignoring the real distribution of memory requests. We discuss our models in details in the following two sections.

$$\text{DRAM}_{\text{lat\_bank}_i} = \text{ave\_queuing\_delay}_i + \text{ave\_service\_time}_i \quad (6)$$

$$\text{DRAM}_{\text{lat}} = \sum_{i=1}^{NB} \frac{\lambda_i}{\sum_{j=1}^{NB} \lambda_j} \times \text{DRAM}_{\text{lat\_bank}_i} \quad (7)$$

2) Memory Request Distribution: We must know how memory requests are distributed between memory banks. This is important to determine row buffer hit and miss for calculating the average service time; this is also important for calculating the inter-arrival rate (λ) of memory requests for each memory bank (see Section III-C3).

The distribution of memory requests to memory banks is determined by the address mapping scheme implemented in hardware. The address mapping scheme denotes how a given memory address is resolved into indexes in terms of channel ID, rank ID, bank ID, row address, and column address.

The address mapping policy has been explored on CPU based on hardware tools [21] and micro-benchmarks [22]. However, the address mapping on GPU is largely unexplored. We introduce an algorithm to explore the address mapping scheme. This algorithm also measures the latencies of row buffer miss and hit on GPU.

Based on the needs of our models, our algorithm detects the bit locations for row or column addresses in a memory address, and does not distinguish the other bits (the address bits for channel, rank, and bank). *A combination of the other bits* identifies a unique memory bank in a specific rank and a specific channel. Given two memory addresses, to determine if they are mapped into the same memory bank, we only need to

check if the combination of the other bits in the two addresses are the same or not. Detecting which address bits are for row or column addresses is sufficient to determine which bits belong to the combination of the other bits.

Algorithm 1 depicts our algorithm. The basic idea of the algorithm is to generate two memory requests whose addresses only differ by one bit (assuming this bit location is x). The first memory request always has a row buffer miss, because the requested data is never accessed before. The first memory request latency is the row buffer miss latency. If the x bit is neither a row bit nor a column bit, then the second memory request will also have a row buffer miss, because the two memory requests access two different memory banks. If the x bit is a column bit, then the second memory request will have a row buffer hit, because the two memory requests access the same row. The second memory request latency is the row buffer hit latency, the shortest access latency of all memory requests. If the x bit is a row bit, then the first and second memory requests go to the same bank but access different rows (i.e. the *row conflict*). For the second memory request, the row of the first memory request is written back first and then a new row of data is fetched. Hence, the second memory request has a row buffer miss, but its latency is the longest access latency of all memory requests because of the row conflict.

We implement the above algorithm and run it on NVIDIA Tesla K80 (Kepler architecture). We measure the row buffer hit and miss latencies as 352ns and 742ns, respectively. When the row conflict happens, the memory access latency is 1008ns. Furthermore, for a 64-bit address, the row and column address bits are 8th-21st bits and 30th-32nd bits. Except the last 3 bits (the byte address bits) and the row and column address bits, all the other bits uniquely identify memory banks.

The above algorithm uses the virtual addresses to detect the address mapping scheme. However, the address mapping scheme maps a physical address into a specific location in the main memory. How a virtual address is mapping to a physical address on GPU is unknown. We extensively test different virtual addresses with largely different address ranges, and find that they manifest the same row and column address bits. We reason that the mapping between the virtual address and the physical address on GPU do not change the positions of row and column address bits.

In general, we use the combination of non-row/column bits as a global memory bank ID to determine the distribution of memory requests among all memory banks. Then we decide the row buffer miss and hit based on the distribution of memory requests, and calculate the average service time for each bank (ave_service_time) based on Equation 8.

$$\begin{aligned}\text{ave\_service\_time} = {} & \text{row\_buffer\_miss\_lat} \times \text{miss\_ratio} \\ & + \text{row\_buffer\_conflict\_lat} \times \text{conflict\_ratio} \\ & + \text{row\_buffer\_hit\_lat} \times \text{hit\_ratio} \end{aligned} \quad (8)$$

3) Queuing Modeling: We use queuing modeling to calculate ave_queuing_delay for each memory bank. The calculation of the average queuing delay requires the knowledge of inter-arrival times and service times of memory requests and the bank utilization. Depending on the statistical distribution of inter-arrival times and service times, different queuing models should be employed. We must examine the statistical distribution to determine an appropriate queuing model.

The existing work shows that the inter-arrival times of memory requests on CPU approximately follow the exponential distribution (a *Markov arrival stream*) [19]. However, we find that the inter-arrival times of memory requests of some GPU kernels cannot be modeled like that. On a GPU platform with massive thread-level parallelism, memory requests tend to arrive in clumps and their inter-arrival times can have a very higher level of variability than those on CPU. Our conclusion is based on the following experiment.

Algorithm 1 Addresses mapping detection.

```
1:  Launch the following GPU kernel with a single thread block with a single thread
2:
3:  procedure GPU KERNEL:
4:      Input: The address bits
5:      Output: the positions of row and column bits
6:
7:      for Each bit x in the address bits do
8:          Generate 2 memory requests with one request's x bit as
              0 and the other request's x bit as 1;
9:          Access the two addresses using uncached GPU load
              instructions ("ld.global.cs") and record the latency
              of the two accesses;
10:
11:     Classify the address bits into three groups according to the
          access latency;
12:     Output the groups with the shortest and longest latency as
          the column and row bits;
```

We collect the inter-arrival times of memory requests with GPGPUSim [23] (using the default configurations for GPU Tesla C2050) for three benchmarks (spmv, md, and matrixMul). Figure 4 shows the measured and theoretical (exponential distribution) inter-arrival time distribution for those benchmarks.The figure shows that the inter-arrival times do not always follow an exponential distribution. In particular, spmv approximately follows, while md and matrixMul do not.

We further calculate the coefficient of variation of inter-arrival time (c_a) to quantify arrival variability based on Equation 10 (we discuss how to calculate c_a in the next paragraphs). We find that the average c_a of all memory banks (96 banks) is 1.11, 2.22, and 1.72 (the standard deviation is 0.19, 0.35, and 0.058) for spmv, md, matrixMul, respectively. In general, the value of c_a varies across benchmarks, but can be much larger than 1 in some kernels (for an exponential distribution c_a should be 1). This indicates that the inter-arrival time of GPU memory requests does not always follow an exponential distribution. Since a higher value of c_a indicates more variability or burstiness in the arrival stream, we conclude that the inter-arrival time of memory requests in a GPU kernel can be bursty. We attribute such bursty memory requests to the massive thread-level parallelism and the similarity of memory access patterns across threads.

Furthermore, the distribution of service times on GPU is also not an exponential distribution, because the service times are clustered into three values, corresponding to the row buffer hit, row buffer miss without conflict, and row conflict. Such distribution of service times is general with large variation.

In general, the inter-arrival times and the service times are *general*. We should not assume they are Markov. Hence, we use a G/G/1 queuing model to estimate the average queuing delay (the inter-arrival time and the service time are represented as G in G/G/1).

Since we choose G/G/1 queuing model, we use Kingman's equation [24] to approximate the average queuing delay, shown in Equations 9 and 10. The model notations are summarized in Table III.

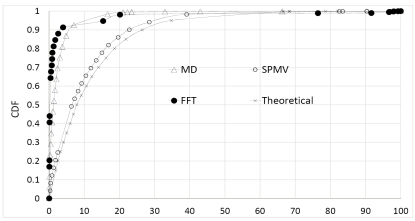


Fig. 4: The measured and theoretical inter-arrival time distribution for three GPU kernels (compute_lj_force for md, vector_kernel for spmv, FFT512_device for fft). We use the default data placements in these kernels.

TABLE III: Notations for our queuing modeling

Model Parameter	Definition
c_a, c_s	Coefficients of variation for interarrival time and service time
τ_a, τ_s	The mean interarrival time and service time
σ_a, σ_s	The standard deviation of interarrival time and service time
λ	The average arrival rate $\lambda = 1/\tau_a$
ρ	Utilization of the server (memory bank), $\rho = \lambda \times \tau_s$

$$\text{ave\_queuing\_delay} = W_q^{G/G/1} \approx (\frac{c_a + c_s}{2})(\frac{\rho}{1-\rho}\tau_a) \quad (9)$$

$$c_a = \frac{\sigma_a}{\tau_a} \qquad c_s = \frac{\sigma_s}{\tau_s} \qquad \rho = \tau_s/\tau_a \quad (10)$$

The calculation of $W_q^{G/G/1}$ (average queuing delay) depends on the approximation of τ_a, σ_a, τ_s and σ_s, which is shown in Equation 10. To calculate τ_a and σ_a, we need to know the interarrival time between any two consecutive DRAM memory requests. We approximate the inter-arrival time for two consecutive DRAM memory requests with the number of instructions between them. To calculate τ_s and σ_s, we need to know the service time for each DRAM memory request. Given the estimation of memory request distribution (Section III-C2), approximating the service time for each memory request is straightforward based on the analysis on row buffer hit and miss.

D. Calculation of $T_{overlap}$

$T_{overlap}$ quantifies the overlapping between T_{comp} and T_{mem}. As we change data placement, the occurrences of memory events, such as cache misses and bank conflicts, change. These memory events impact the extent of the overlapping. Different memory events with different stall cycles can result in different extent of overlapping.

Based on the above discussion, we build up an empirical model for $T_{overlap}$ shown in Equations 11 and 12. In Equation 12, we calculate $T_{overlap}$ based on $T_{overlap\_ratio}$. This makes the model based on the event ratio instead of the absolute event numbers. Calculating $T_{overlap\_ratio}$ makes models independent of applications and results in better modeling accuracy.

$$T_{overlap\_ratio} = \sum_i g_i \times e_i + \sum_j c_j \times e_j + \sum_m t_m \times e_m + \sum_n s_n \times e_n + \sum_k r_k \times e_k + w \times \text{\#warps} + c \quad (11)$$

$$T_{overlap} = T_{overlap\_ratio} \times T_{mem} \quad (12)$$

In essence, Equation 11 captures the relationship between a set of performance-critical memory events and $T_{overlap\_ratio}$. For global memory, these memory events (e_i) are L2 cache misses plus global memory requests; For constant memory, these memory events (e_j) are constant cache misses plus constant memory requests; For texture memory, these memory events (e_m) are texture cache misses plus texture memory requests; For shared memory, these memory events (e_n) are bank conflicts plus shared memory requests. We also include row buffer miss and conflict events (e_k) and the number of warps per SM into Equation 11. The number of warps per SM is necessary, because it provides the information on the availability of threads to cover the stall cycles.

There are also a set of coefficients (i.e., g_i, c_j, t_m, s_n, r_k and w) and a constant factor (c) in Equation 11. To construct the model, those coefficients and the constant factor are derived using linear regression with a set of benchmarks. When making prediction for $T_{overlap}$ for a target data placement, those memory events in Equation 11 are calculated by the instruction trace analysis and cache models (see Section IV).

E. Other Details

To predict performance for a target data placement, we need to know the addresses of target data objects in the target memory component. This is necessary to determine the distribution of memory requests among memory banks and quantify cache misses. In our model, if the location of the target data object is changed between the off-chip memories, the address of the target data object remains the same. If the location of the target data object is changed between an off-chip memory and shared memory, we assign an address range to the target data object after the allocated largest memory addresses on the target memory component. Also, the assigned address range follows the requirements of memory alignment and data object size.

To identify memory accesses to the target data object, we need to know the addresses of the target data object in the sample data placement. This can be done by instrumenting the GPU kernel with SASSI [25] and output the address range of the target data object at the beginning of the kernel call.

IV. Implementation

Our performance models require a variety of information on the execution of the sample data placement. Based on the information, our models predict performance for a target data placement. The implementation of the models is a framework including instruction and memory trace generation, cache models, and trace analysis.

We develop an instruction trace generator and a memory trace generator based on SASSI [25]. The instruction trace generator generates an SASS instruction trace for all threads, and the memory trace generator collects load and store instructions to generate a memory trace. The memory trace is then processed to replace load and store operations of the sample data placement with those of the target data placement accommodating the addressing mode difference.

We further develop cache models (including the texture cache, constant cache, and L2 cache) based on the cache models in GPGPUSim. Our cache models take the processed memory trace as input, and then output a new memory trace filtered by our cache models. The memory requests in the new memory trace include the dynamic instruction IDs that issue memory requests. The new memory trace is fed into the T_{mem} model to count inter-arrival times and row buffer misses/hits based on the distribution of memory requests among memory banks. Our cache models also count disruptive memory events

TABLE IV: Benchmarks for evaluation. We use the notation "kernel_name [data_object_name(mem1→mem2), ...]" to represent a specific data placement test. "mem1" is the original data placement and "mem2" is the new data placement. In the first column, the numbers after each benchmark name give the number of data placement tests. Those numbers include the sample data placements. G, T, C, S and 2T stand for global, 1Dtexture, constant, shared, and 2Dtexture memories, respectively. Convol. stands for the benchmark convolutionSeparable.

Benchmark	Data placement test
Benchmarks for evaluation	
SHOC:bfs(2)	BFS_kernel_warp[edgeArray(G→T)]
SHOC:fft(2)	FFT512_device[smem(S→G)]
SHOC:neuralnet(5)	kernelFeedForward1[weights(G→C, G→S, G→T, G→2T)]
SHOC:reduction(2)	reduce[sdata(S→G)]
SHOC:scan(2)	reduce[g_idata(G→2T)]
SHOC:sort(2)	reorderdata[sBlockOffsets(S→G)]
SHOC:stencil2d(2)	StencilKernel[data(G→T)]
SHOC:md5hash(2)	Find.[foundKey(G→S)]
SHOC:S3D(3)	gr_base[gpu_p(G→T), gpu_y(G→T), gpu_p,gpu_y(G→T)]
Benchmarks for training $T_{overlap}$	
SDK:convol.(5)	convolutionRowsKernel[(d_Src(G→2T)], convolutionRowsKernel[(d_Src(G→T)], convolutionRowsKernel[(c_Kernel(C→G)], convolutionRowsKernel[(c_Kernel(C→T)]
SHOC:md(6)	compute_lj_force[d_position(T→G), neighList(G→)T], compute_lj_force[d_positioin(T→G)], compute_lj_force[d_position(T→G)], compute_lj_force[d_position(T→G), neighList(G→)T], compute_lj_force[neighList(G→)T]
SDK:matrixMul(8)	matrixMul[A(G→2T), B(G→2T)], matrixMul[A(G→2T)], matrixMul[A(G→T)], matrixMul[A(G→T), B(G→2T)], matrixMul[B(G→2T)], matrixMul[A(G→T),B(G→T), matrixMul[B(G→T)]
SHOC:spmv(10)	vector_kernel[rowD.(G→S), d_vec(T→)G], vector_kernel[rowD.(G→C), d_vec(T→)G], vector_kernel[rowD.(G→T), d_vec(T→)G], vector_kernel[rowD.(G→S)], vector_kernel[val(G→T), d_vec(T→)G], vector_kernel[rowD.(G→T), d_vec(T→)C], vector_kernel[rowD.(G→S)], vector_kernel[val(G→T), cols(G→T), rowD.(G→C), d_vec(T→)G], vector_kernel[val(G→T), cols(G→T)],
SDK:transpose(3)	transposeNaiv[odata(G→2T)], transposeNaiv[idata(G→T)]
SDK:cfd(2)	cuda_compute_flux[variables(G→T)]
SHOC:triad (2)	triad[B(G→S)]
SHOC:QTC(2)	QTC_device[distance_matrix_txt(G-¿2T)]

(e.g., the cache miss and memory bank conflict). The statistics of those memory events is fed into the T_{comp} model to estimate instruction replays and into the $T_{overlap}$ model.

V. Evaluation

We test our performance models on an NVIDIA Tesla K80 (Kepler architecture). To test the model accuracy, we predict performance for various data placements. Then, we implement those data placements and measure their performance on GPU. We compare the predicted and measured results. Table IV lists benchmarks we use for evaluation and model training. Those benchmarks include all of benchmarks in SHOC benchmark suite [14] and some CUDA SDK benchmarks. To train the $T_{overlap}$ model, we use 38 data placements (see Table IV). Those training benchmarks are selected with various memory access patterns and data placements. We use the trained model to make prediction for other benchmarks. Hence the training benchmarks and evaluation benchmarks are separate.

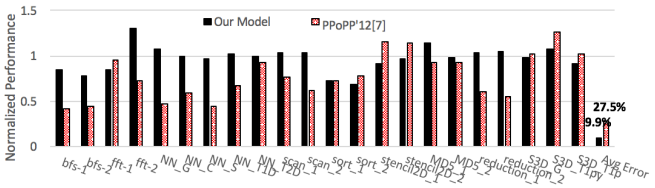


Fig. 5: Predicted performance for various data placements based on [7] and our models. The predicted performance is normalized by the measured performance.

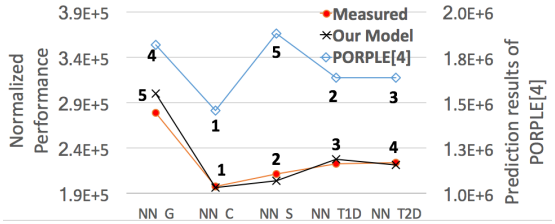


Fig. 6: Prediction accuracy comparison between PORPLE [4] and our work. The numbers above markers in the figure are performance ranks based on performance modeling. Performance ranking based on our model is consistent with that based on the measured performance.

A. Model Accuracy

Figure 5 shows the prediction accuracy. We evaluate multiple data placements for each benchmark (see [12]). The figure shows the predicted performance normalized to the measured performance. In general, the models achieve high prediction accuracy. The arithmetic average prediction error is 9.9%.

We also compare our work with the existing work [7] (see Figure 5). Our work is based on [7] but with significant improvement. As a result, we improve performance prediction accuracy by 17.6% (on average). Our models introduce the detailed instruction counting (counting instruction replays and addressing mode difference) and account for latency variation of off-chip memory accesses based on the queuing model and address mapping scheme. As a result, our models perform particularly better than [7] for benchmarks NN_C and SCAN_2 (improving accuracy by 41% and 38% respectively), because the two target kernels in the two benchmarks have a large number of instruction replays. Our models also perform particularly better for Reduction_2 (improving accuracy by 48%), because the target kernel in this benchmark has significant number of row buffer misses which cause off-chip memory access latency variation.

To further demonstrate our modeling accuracy, we compare our model with another related work PORPLE [4]. PORPLE uses a memory latency-oriented performance model. The model aims to rank performance of different data placements instead of predicting execution time. To compare with PORPLE, we use our models to rank performance of five data placements of a kernel (kernelFeedForward1) in a SHOC benchmark "neuralnet". Figure 6 shows the results. PORPLE cannot correctly rank different data placements, especially because of its poor performance modeling result for a data placement (NN_S). Our models correctly rank the performance of those data placements, because of high prediction accuracy.

B. Exploring Model Construction

To investigate the impact of various performance factors on modeling accuracy, we remove them from our models, and then observe how the modeling accuracy changes. This

evaluation provides an important indication on how and which we should model on HMS

In this evaluation, we introduce a "baseline" model, which is our models without the detailed instruction counting and the queuing model, and with even memory distribution between memory banks (no consideration of address mapping). Hence, the baseline model does not include those critical performance factors in our models. This baseline model is different from [7], in that the baseline model uses Equation 11 to calculate $T_{overlap}$, while [7] uses CWP and MWP formulation [6]. Without the consideration of those critical performance factors, the baseline model performs even worse than [7], because it incorrectly calculates the numbers of those memory events needed by Equation 11.

Figure 7 shows the modeling results with and without the detailed instruction counting. In our models, we consider the difference in the number of issued instructions across data placements in details. In particular, we count instruction replays and address mode differences. We notice that introducing the detailed instruction counting improves modeling accuracy by 17% on average (see "Baseline" vs. "Baseline+instr replay&addr mode diff"). Some benchmarks are very sensitive to accurately counting of issued instructions. For example, fft_1, NN_S, and bfs_2 have 142%, 106%, and 67% difference in modeling accuracy between "Baseline" and "Baseline+insts replay&addr mode diff".

We further introduce the queuing model into "Baseline+insts replay&addr mode diff" to study the impact of the queuing model with the detailed instruction counting method in place. To separate the effect of memory request distribution from the effect of the queuing model, we do not consider address mapping when employing the queuing model, and assume even distribution of memory requests between memory banks (see "Baseline+insts replay&addr mode diff+queuing model(even mem requests)"). Figure 8 shows the result. With the employment of the queuing model (assuming even distribution of memory requests), we improve modeling accuracy by 31%, comparing with the baseline. With the consideration of address mapping, we further improve the modeling accuracy of the queuing model by 8.1% (see "Our Model" vs. "Baseline+insts replay&addr mode diff+queuing model(even mem requests)"). These results demonstrate that accurately modeling off-chip memory access latency is very helpful to improve modeling accuracy, and using queuing model we can effectively capture the variation of off-chip memory access latency across memory requests.

To further study the impact of the queuing model, we apply the queuing model to the baseline without the detailed instruction counting method. We want to separate the effect of the queuing model from the effect of the instruction counting method, but with the consideration of address mapping. Figure 9 shows the result. In general, the queuing model alone improves modeling accuracy by 13.8% on average. With the queuing model in place, applying other modeling techniques improves modeling accuracy by 25.3% on average. Combining the results from Figures 7 and 9, we see the necessity of using multiple modeling techniques. Using the detailed addressing counting method alone or using the queuing modeling alone, we improve the baseline by 17% and 13.8% respectively, but when employing both of them, we improve the baseline by 39.1%, larger than the combination of the improvements of using the two techniques alone.

VI. Related Work

GPU Performance Models. Hong and Kim [6] introduce an analytical model based on the quantification of memory warp-level and thread-level parallelism. Sim et.al [7] further improve this model by considering cache effects and effective instruction throughput. However, they do not model the queuing delay due to resource contention in the memory system.

Baghsorkhi et al. [26] introduce a work flow graph by extending the traditional control flow group to estimate performance. However, they have limited consideration for modeling the overlap between computation and memory access, and cannot study the data placement problem. Zhang and Owens [8] propose a performance framework to measure the execution time spent on global and shared memories and instruction pipeline. However, their work cannot quantify the performance benefits of data placement optimizations. Huang et. al [27] propose GPU performance modeling based on interval analysis to capture the effects of multithreading and resource contentions caused by memory divergence. Their models can be used to study the performance effects of different architecture design options, which is complementary to our study. Tang et al. [28] use the reuse distance theory to model the cache miss rate. Nugteren et al. [29] also employ the reuse distance theory but with extra considerations of thread block scheduling and MSHR. Choi et al. [30] introduce automated performance tuning to optimize sparse matrix-vector multiplication on GPU. In Sectoin V, we compare our models with two of the state-of-the-art models [4], [7].

Data Placement Problem for Heterogeneous Memory. Chen et al. [4] introduce a framework to enable automatic data placement on GPU, based on a series of compiler and runtime techniques and a performance model. However, their performance model to direct data placement does not sufficiently consider memory level parallelism and the overlapping between computation and memory accesses. Jang et al. [5] propose a list of rules to guide users in placing data when writing a program. Ma et al. [31] consider the optimal data placement on shared memory only. Wang et al. [2] study the energy and performance tradeoff for placing data on DRAM versus non-volatile memory on GPU. Agarwal et al. [1] study data placement strategies for GPU with high bandwidth-based heterogeneous memory. They introduce a bandwidth-aware data placement strategy and profile data structure accesses to direct data placement. Our work complements the above by modeling performance to direct data placement.

VII. Conclusions

This paper introduces performance models to predict performance for various data placements on GPU with HMS. We reveal performance factors that are critical for performance modeling for HMS. Our models can work as a tool to help programmers for GPU performance optimization.

Acknowledgement. This work is partially supported by U.S. NSF (CNS-1617967 and CCF-1553645). We thank anonymous reviewers for their valuable feedback.

VIII. Appendix

We use Equation 13 to calculate effective instruction throughput for T_{comp}. Similar to [7], we use the latency of FP operations to approximate average instruction latency in the equation. ITILP is the inter-thread instruction-level parallelism, which represents how much ILP (instruction-level

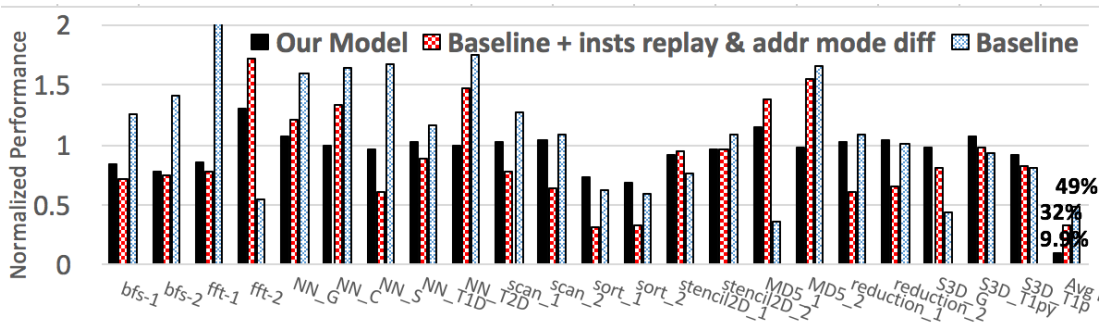


Fig. 7: The impact of instruction counting on model accuracy.

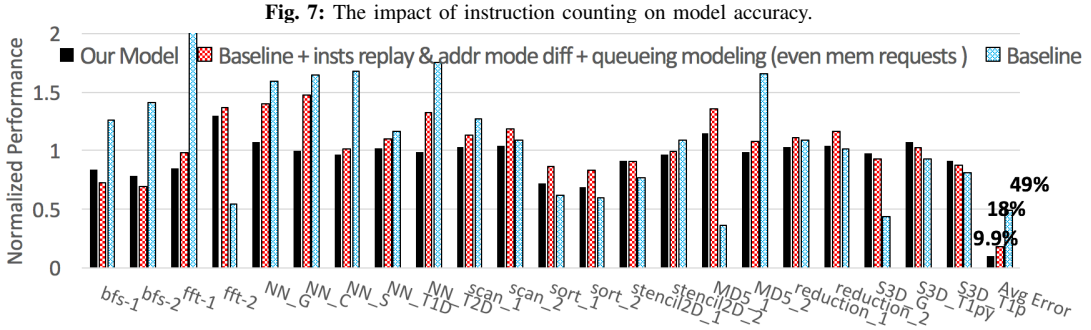


Fig. 8: The impact of the queuing modeling on model accuracy. The detailed instruction counting method (considering addressing mode and instruction replay) is in place with the queuing model.

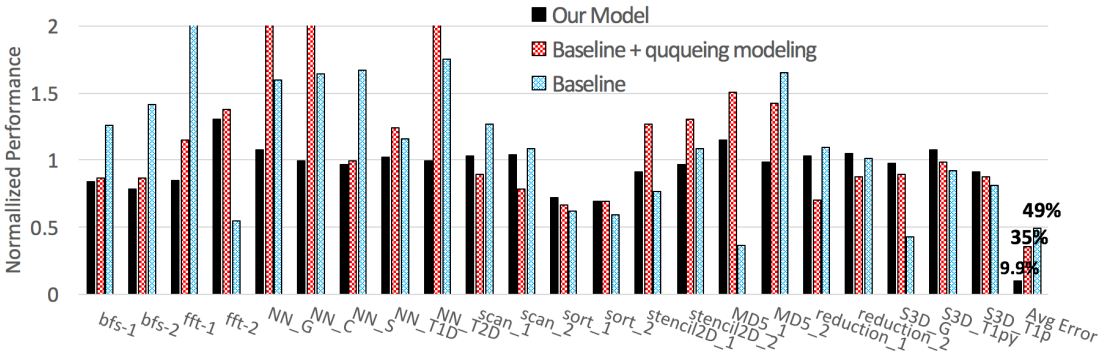


Fig. 9: The impact of queuing modeling on model accuracy.

parallelism among warps) is available among N warps to hide pipeline latency (Eq. 14 and 15).

$$\text{Effective\_instruction\_throughput} = \frac{\text{avg\_inst\_lat}}{\text{ITILP}} \tag{13}$$

$$\text{ITILP} = min(\text{ILP} \times N, \text{ITILP}_{max}) \tag{14}$$

$$\text{ITILP}_{max} = \frac{\text{avg\_inst\_lat}}{\text{warp\_size/SIMD\_width}} \tag{15}$$

The serialization overhead (W_{serial}) for computing T_{comp} is modeled in Equation 16. W_{serial} includes sync overhead (O_{sync}), SFU resource contention overhead (O_{SFU}), and control flow divergence overhead (O_{CFdiv}). We assume that these overhead are the same between different data placements.

$$W_{serial} = O_{sync} + O_{SFU} + O_{CFdiv} \tag{16}$$

Effective_memory_requests per SM for computing T_{mem} is modeled in Equation 17. ITMLP in the equation defines the number of memory requests per SM concurrently serviced. ITMLP is calculated in Equations 18 and 19. The calculation of ITMPL is based on [6].

$$\text{Effective\_memory\_requests per SM} = \frac{\text{\#mem\_instrs} \times \text{\#total\_warps}}{\text{active\_SMs} \times \text{ITMLP}} \tag{17}$$

$$\text{ITMLP} = min(\text{MLP} \times \text{MWP}_{cp}, \text{MWP}_{peak\_bw}) \tag{18}$$

$$\text{MWP}_{cp} = min(max(1, \text{CWP} - 1), \text{MWP}) \tag{19}$$

REFERENCES

[1] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page Placement Strategies for GPUs Within Heterogeneous Memory Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[2] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter, "Exploring Hybrid Memory for GPU Energy Efficiency through Software-Hardware Co-Design," in *PACT*, 2013.

[3] F. X. Lin and X. Liu, "Memif: Towards Programming Heterogeneous Memory Asynchronously," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[4] G. Chen, B. Wu, D. Li, and X. Shen, "PORPLE: An Extensible Optimizer for Portable Data Placement on GPU," in *IEEE/ACM International Symposium on Microarchitecture*, 2014.

[5] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 105–118, 2011.

[6] 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 Annual International Symposium on Computer Architecture*, ser. ISCA '09, 2009.

[7] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[8] Y. Zhang and J. D. Owens, "A Quantitative Performance Analysis Model for GPU Architectures," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[9] G. Chen and X. Shen, "Coherence-Free Multiview: Enabling Reference-Discerning Data Placement on GPU," in *International Conference on Supercomputing (ICS)*, 2016.

[10] N. Fauzia, L.-N. Pouchet, and P. Sadayappan, "Characterizing and Enhancing Global Memory Data Coalescing on GPUs," in *International Symposium on Code Generation and Optimization (CGO)*, 2015.

[11] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[12] Y. Huang and D. Li, "Performance Modeling for Optimal Data Placement on GPU with Heterogeneous Memory Systems," 2016, Technical Report, PASA Lab, University of California, Merced.

[13] A. Singhal, "Modern Information Retrieval: A Brief Overview," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 24, no. 4, 2001.

[14] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) benchmark suite," in *GPGPU*, 2010.

[15] "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[16] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU Microarchitecture through Microbenchmarking," in *ISPASS*. IEEE Computer Society, 2010, pp. 235–246.

[17] X. Mei and X. Chu, "Dissecting GPU Memory Hierarchy through Microbenchmarking," *CoRR*, vol. abs/1509.02308, 2015.

[18] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *International Symposium on Computer Architecture (ISCA)*, 2000.

[19] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "ANATOMY: An Analytical Model of Memory System Performance," in *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2014.

[20] N. Gulur, M. Mehendale, and R. Govindarajan, "A Comprehensive Analytical Performance Model of DRAM Caches," in *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2015.

[21] Y. Bao, M. Chen, Y. Ruan, L. Liu, J. Fan, Q. Yuan, B. Song, and J. Xu, "HMTT: A Platform Independent Full-system Memory Trace Monitoring System," in *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2008.

[22] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[23] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[24] J. Kingman, "On the Algebra of Queues," *Journal of Applied Probability*, vol. 3, pp. 285–326, 1996.

[25] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible Software Profiling of GPU Architectures," in *International Symposium on Computer Architecture (ISCA)*, 2015.

[26] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An Adaptive Performance Modeling Tool for GPU Architectures," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[27] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "GPUMech: GPU Performance Modeling Technique Based on Interval Analysis," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[28] T. Tang, X. Yang, and Y. Lin, "Cache Miss Analysis for GPU Programs Based on Stack Distance Profile," in *International Conference on Distributed Computing Systems (ICDCS)*, 2011.

[29] C. Nugteren, G. J. van den Braak, H. Corporaal, and H. Bal, "A Detailed GPU Cache Model Based on Reuse Distance Theory," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[30] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-Driven Autotuning of Sparse Matrix-vector Multiply on GPUs," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[31] W. Ma and G. Agrawal, "An Integer Programming Framework for Optimizing Shared Memory Use on GPUs," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.