MDM: The GPU Memory Divergence Model

Lu Wang[†] Magnus Jahre[‡] Almutaz Adileh[⟩*] Lieven Eeckhout[†]

[†]Ghent University, Belgium — {*luluwang.wang, lieven.eeckhout*}*@ugent.be*
[‡]Norwegian University of Science and Technology — *magnus.jahre@ntnu.no*
[⟩]Huawei, Toga Networks — *almutaz.adileh@huawei.com*

***Abstract*—Analytical models enable architects to carry out early-stage design space exploration several orders of magnitude faster than cycle-accurate simulation by capturing first-order performance phenomena with a set of mathematical equations. However, this speed advantage is void if the conclusions obtained through the model are misleading due to model inaccuracies. Therefore, a practical analytical model needs to be sufficiently accurate to capture key performance trends across a broad range of applications and architectural configurations.**

In this work, we focus on analytically modeling the performance of emerging memory-divergent GPU-compute applications which are common in domains such as machine learning and data analytics. The poor spatial locality of these applications leads to frequent L1 cache blocking due to the application issuing significantly more concurrent cache misses than the cache can support, which cripples the GPU's ability to use Thread-Level Parallelism (TLP) to hide memory latencies. We propose the GPU Memory Divergence Model (MDM) which faithfully captures the key performance characteristics of memory-divergent applications, including memory request batching and excessive NoC/DRAM queueing delays. We validate MDM against detailed simulation and real hardware, and report substantial improvements in (1) *scope:* the ability to model prevalent memory-divergent applications in addition to non-memory divergent applications; (2) *practicality:* 6.1× faster by computing model inputs using binary instrumentation as opposed to functional simulation; and (3) *accuracy:* 13.9% average prediction error versus 162% for the state-of-the-art GPUMech model.

I. Introduction

GPUs are the de facto standard platform for executing performance-critical applications. Their highly parallel execution model and high-performance memory system make GPUs a popular choice for emerging machine learning [17], [20], [27] and data analytics [3], [4] workloads. Several contemporary GPU applications differ from traditional GPU-compute workloads as they put a much larger strain on the memory system. More specifically, they are memory-intensive *and* memory-divergent. These applications typically have strided or data-dependent access patterns which cause the accesses of the concurrently executing threads to be *divergent* as loads from different threads access different cache lines. We refer to this application class as *Memory Divergent (MD)*, in contrast to the more well-understood *Non-Memory Divergent (NMD)* applications.

Analytical GPU Performance Modeling. Analyzing and optimizing GPU architecture for the broad diversity of modern-day GPU-compute applications is challenging. Simulation is arguably the most commonly used evaluation tool as it enables detailed, even cycle-accurate, analysis. However, simulation is excruciatingly slow and parameter sweeps commonly require thousands of CPU hours. An alternative approach is modeling, which captures the key performance-related behavior of the architecture in a set of mathematical equations, which is much faster to evaluate than simulation.

Modeling can be broadly classified in machine-learning (ML) based modeling versus analytical modeling. ML-based modeling [9], [36] requires offline training to infer or learn a performance model. A major limitation of ML-based modeling is that a large number (typically thousands) of training examples are needed to infer a performance model. These training examples are obtained through detailed simulation, which leads to a substantial one-time cost. Moreover, extracting insight from an ML-based performance model is not always straightforward. Analytical modeling [1], [14], [15], [31], [34], [38] derives a performance model from fundamentally understanding the underlying architecture and hence provide deep insight, i.e., the model is a white box, and the one-time cost is small once the model has been developed. The latter is extremely important when exploring large design spaces. Typically, the analytical model is used to quickly identify an interesting region in the design space which is then studied in more detail using accurate but time-consuming simulation. It is hence more important that the analytical model captures key performance trends than that it minimizes absolute error.

This work advances the state-of-the-art in analytical GPU performance modeling by expanding its scope, improving its practicality, and enhancing its accuracy. GPUMech [15] is the state-of-art analytical performance model for GPUs, however, it is highly inaccurate for memory-divergent applications, and in addition, it is impractical as it relies on functional simulation to collect the model inputs. We propose the MDM performance model which is highly accurate across the broad spectrum of MD and NMD-applications, and in addition is practical by relying on binary instrumentation for collecting model inputs. Before diving into the specific contributions of this work, we first point out the prevalence of memory-divergent applications.

Memory-Divergent Applications are Prevalent. We use Nvidia's Visual Profiler [30] on an Nvidia GTX 1080 GPU to categorize all benchmarks in the Rodinia [5], Parboil [32], Polybench [11], Tango [20], LonestarGPU [3], and Mars [13] benchmark suites (see Table I). We define an application as

*This work was performed while at Ghent University.

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00085

TABLE I: MD-applications across widely used GPU benchmark suites. *MD-applications are common.*

Suite	Ref.	#MD-app.	#NMD-app.	Sum
Rodinia	[5]	4	12	16
Tango	[20]	2	6	8
LonestarGPU	[3]	4	2	6
Polybench	[11]	4	8	12
Mars	[13]	6	0	6
Parboil	[32]	1	7	8
Sum		**21**	**35**	**56**

memory-divergent if it features more than 10 Divergent loads Per Kilo Instructions (DPKI). Note that this classification is not particularly sensitive to the DPKI threshold: most NMD-applications have a DPKI at, or close to, 0 (maximum 3) while the average DPKI for the MD-applications is around 64 (minimum is 10). DPKI is an architecture-independent metric and identifies the applications where memory divergence significantly affects performance since it captures both memory intensity and degree of memory divergence in a single number. Overall, we find that 38% (i.e., 21 out of 56) of the benchmarks are memory divergent (i.e., have DPKI larger than 10). Given the prevalence of MD applications, it is clear that analytical models must capture their key performance-related behavior.

Memory divergence is challenging to model analytically and existing models are highly inaccurate: in particular, GPUMech [15] incurs an average performance error of 298% for a broad set of MD-applications, even though GPUMech is accurate for NMD-applications. The key problem is that prior work does not model the performance impact of Miss Status Holding Registers (MSHRs) [22], nor does it accurately account for Network-on-Chip (NoC) and DRAM queueing delays. More specifically, concurrent cache requests in an MD-application (mostly) map to different L1 cache blocks which causes the L1 cache to run out of MSHRs. This cripples the ability of the GPU's Streaming Multiprocessors (SMs) to hide memory access latency through Thread-Level Parallelism (TLP), i.e., the SMs can no longer execute independent warps because the cache cannot accept further accesses until one of the current misses resolves. Furthermore, memory divergence incurs a flood of memory requests that severely congest the NoC and DRAM subsystems, which in turn leads to long queueing delays.

Paper Contributions. In this work, we propose the *Memory Divergence Model (MDM)*, the first analytical performance model for GPUs that accurately predicts performance for MD-applications. MDM builds upon two key insights. First, L1 cache blocking causes the memory requests of concurrently executing warps to be processed in *batches*. The reason is that MD-applications have more concurrent misses than there are MSHRs, and the cache can maximally service as many concurrent misses as there are MSHRs. More specifically, a cache with N MSHRs will process M requests in roughly M/N batches because it will block after the first N misses. Second, MD-applications saturate the NoC and the memory system which means that a memory request is queued behind all other concurrent requests. Because TLP-based latency hiding breaks down for MD-applications, the SMs are exposed to the complete memory access latency.

MDM faithfully models both MSHR batching and NoC/-DRAM queueing delays, which reduces performance prediction error by 16.5× on average compared to the state-of-the-art GPUMech [15] for MD-applications. At the same time, MDM is equally accurate as GPUMech for NMD-applications. Across a set of MD and NMD-applications, we report an average prediction error of 13.9% for MDM compared to detailed simulation (versus 162% for GPUMech). Moreover, we demonstrate high accuracy across a broad design space in which we vary the number of MSHRs, NoC and DRAM bandwidth, as well as SM count. Furthermore, we validate MDM against real hardware, using binary instrumentation to collect the model inputs as opposed to functional simulation as done in prior work. Thereby, we improve both model evaluation speed (by 6.1×) and accuracy (average prediction error of 40% for MDM versus 164% for GPUMech).

To demonstrate the utility of MDM, we perform four case studies. First, we use MDM and GPUMech to explore the performance impact of changing the number of SMs and DRAM bandwidth for an NMD versus MD-workload. Overall, MDM predicts the same performance trends as simulation, while GPUMech erroneously predicts that providing more SMs and DRAM bandwidth significantly improves performance for the MD-workload. In a second case study, we demonstrate that the general-purpose MDM is equally accurate as the special-purpose CRISP model [24], in contrast to GPUMech, for predicting the performance impact of DVFS — CRISP is a dedicated model which relies on yet-to-be-implemented hardware performance counters, in contrast to the general-purpose MDM model. Third, we validate that MD-applications indeed suffer from MSHR batching and NoC/DRAM queueing and that MDM accurately captures these performance effects. Finally, we demonstrate that MDM is equally accurate for the aggressive streaming L1 cache [6], [21].

In summary, we make the following contributions:

- We identify the key architectural mechanisms that determine the performance of memory-divergent GPU applications. Poor spatial locality leads to widespread L1 cache blocking due to lack of MSHRs, which results in cache misses being processed in batches and the SMs being unable to leverage TLP to hide memory access latency and NoC/DRAM queueing delays.
- We propose the Memory Divergence Model (MDM), which faithfully models the batching behavior and NoC/-DRAM queueing delays observed in MD-applications. MDM significantly improves performance prediction accuracy compared to GPUMech: we report an average prediction error of 13.9% for MDM versus 162% for GPUMech when compared to detailed simulation.
- We validate MDM against real hardware, showing substantially higher accuracy than GPUMech (40% versus 164% prediction error), while in addition being 6.1× faster by

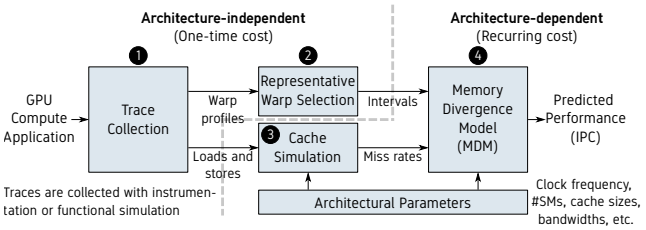


Fig. 1: MDM-based performance prediction.

collecting model inputs through binary instrumentation as opposed to functional simulation.

II. MDM Overview

MDM is based on interval modeling [10], [18], which is an established approach for analytical CPU performance modeling. The key observations are that an application will have a certain steady-state performance in the absence of miss events (e.g., data cache misses), and that miss events are independent of each other. Therefore, performance can be predicted by estimating steady-state performance and subtracting the performance loss due to each miss event. Interval modeling was originally proposed for single-threaded CPU workloads, and applying it to GPUs is not straightforward due to their highly parallel execution model [15].

Figure 1 provides a high-level overview of the MDM performance model. The first task is to collect instruction traces which can be accomplished through instrumentation on real hardware or through functional simulation (see ❶). Traces are architecture-independent and therefore only need to be gathered once for each benchmark, i.e., trace collection is a one-time cost. Instrumentation on real hardware is significantly faster than functional simulation.[1] Furthermore, binary instrumentation provides traces of native hardware instructions while functional simulation captures instructions at the intermediate PTX representation [25] (which may be inaccurate [12]). For these reasons, instrumentation on real hardware is generally preferable. An important exception is when validating the performance model across design spaces, which can only be done through simulation — real hardware enables evaluating a single design point only, and moreover, the difference in instruction abstraction level introduces error.

We next use the instruction trace to create interval profiles for all warps in the application (see ❷). Huang et al. [15] observe that the execution behaviors of the warps of a GPU-compute application are sufficiently similar so that a single representative warp can be used as input to the performance model. Following this observation, we select a representative warp based on its architecture-independent characteristics such as instruction mix and inter-instruction dependencies. Our analysis confirms that this approach is equally accurate for both MD- and NMD-applications — since memory divergence primarily affects the latency of the miss events.

[1]We extend the dynamic instrumentation tool NVBit [33] to capture per-warp instruction and memory address traces; see Section IV for details regarding our experimental setup.

TABLE II: Evaluation time as a function of design space size for detailed simulation and MDM. *MDM is orders of magnitude faster than detailed simulation for large design spaces.*

Architectural Configurations	Detailed Simulation	MDM	
		One-Time Cost	Recurrent Cost
1	9.8 days	3.6 hours	2 minutes
10	3.3 months	3.6 hours	20 minutes
100	2.7 years	3.6 hours	3.3 hours
1000	27 years	3.6 hours	1.4 days

In contrast to trace collection and representative warp selection which incurs a *one-time cost* per application, the next steps depend on both the application and the architecture, and hence need to be run once for each architecture-application pair; this is a *recurring cost* which is proportional to the number of architecture configurations to be explored. We first run the load and store instruction traces through a cache simulator to obtain the miss rates for all caches (see ❸). We consider all warps in the cache model as the accesses of concurrently executed warps significantly affect miss rates (both constructively and destructively). Finally, we provide the intervals and the miss rates to our MDM performance model to predict overall application performance (IPC) for a particular architecture configuration (see ❹).

MDM dramatically reduces evaluation time compared to simulation and makes exploring large GPU design spaces practical, i.e., hours or few days for MDM versus months or years for simulation. Table II explores MDM model evaluation time compared to simulation as a function of the number of architectural configurations in the design space. For each architectural configuration, we evaluate all our 17 benchmarks on a system with an Intel Xeon CPU and an NVIDIA GeForce GTX 1080 (see Section IV). The key take-away is that MDM-based evaluation is orders of magnitude faster than detailed simulation. More specifically, MDM speeds up evaluation by 65× when only considering a single configuration — less than 4 hours versus almost ten days — which then grows to 6371× when evaluating 1000 configurations — less than two days versus many years. Exploiting parallelism in a server farm speeds up simulation and MDM equally. The root cause of MDM's speed and scalability is that the recurring costs (i.e., steps ❸ and ❹ in Figure 1) are small compared to the one-time costs (i.e., steps ❶ and ❷), and MDM's one-time cost is much smaller compared to simulation.

III. Modeling Memory Divergence

We now describe how MDM captures the key performance-related behavior of MD-kernels. We first analyze the key performance characteristics of MD-applications (Section III-A), and we then use interval analysis to examine how this impacts performance (Section III-B). The observations that come out of this analysis then lead to the derivation of the MDM performance model (Section III-C).

Listing 1: Example kernel with strided access pattern.

```
1 int ix = blockIdx.x*blockDim.x+threadIdx.x;
2 __shared__ output[blockDim.x][N];
3 for (i=0; i<N; i++){
4   int index = GS*ix+i;       // Compute index
5   float t   = input[index]; // Load value
6   output[ix][i] = t*t;
7 }
```

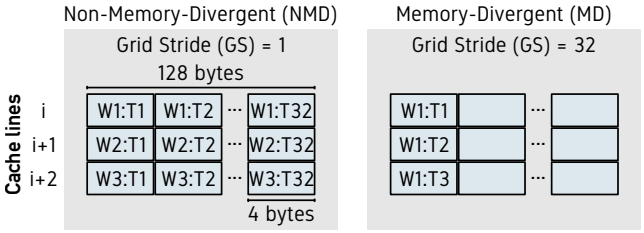


Fig. 2: Cache behavior for the example kernel in Listing 1.

A. Key Performance Characteristics

The basic unit of a GPU-compute application is a thread. Threads are organized into Thread Blocks (TBs), and the threads in a TB can execute sequentially or concurrently and communicate with each other. The threads are dynamically grouped into warps by the warp scheduler, and each warp executes the same instructions on different data elements. GPUs execute the instructions of all threads within a warp in lock-step across the cores of a Streaming Multiprocessor (SM). For loads, this means that each thread issues a load for a single data element. The per-thread requests within a warp are aggregated to cache requests by the coalescer. On a cache hit, the cache line is retrieved and provided to the SM's compute cores. On a miss, one or more MSHRs is allocated and a corresponding number of memory requests are sent to the lower levels of the memory hierarchy. If a warp cannot complete (e.g., due to an LLC miss), the warp scheduler will try to execute other warps to hide latencies (possibly from other TBs). The instructions within a warp are executed in program order, and the SM stalls when the pending instructions of all available warps are blocked.

Listing 1 shows a simple GPU kernel that we use to illustrate the key performance characteristics of MD and NMD applications. The example kernel squares the contents of an array and reorganizes it into a matrix. At runtime, each thread executes the instructions of the kernel on a subset of the application's data. The exact data elements are determined by the thread's position within the thread grid (line 1). For each iteration of the loop, the kernel computes the array index (line 4), loads the matrix value (line 5), and finally squares the value and writes it back to memory (line 6). The access pattern strides are determined by the constant Grid Stride (GS). In the following, we assume 32 threads per warp, 128-byte L1 cache lines, 4-byte floats, as well as that the input array is cache-aligned and much larger than the L1 cache. We use the notation W1:T1 to refer to thread 1 within warp 1.

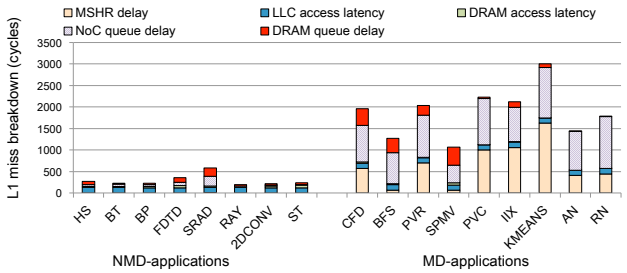


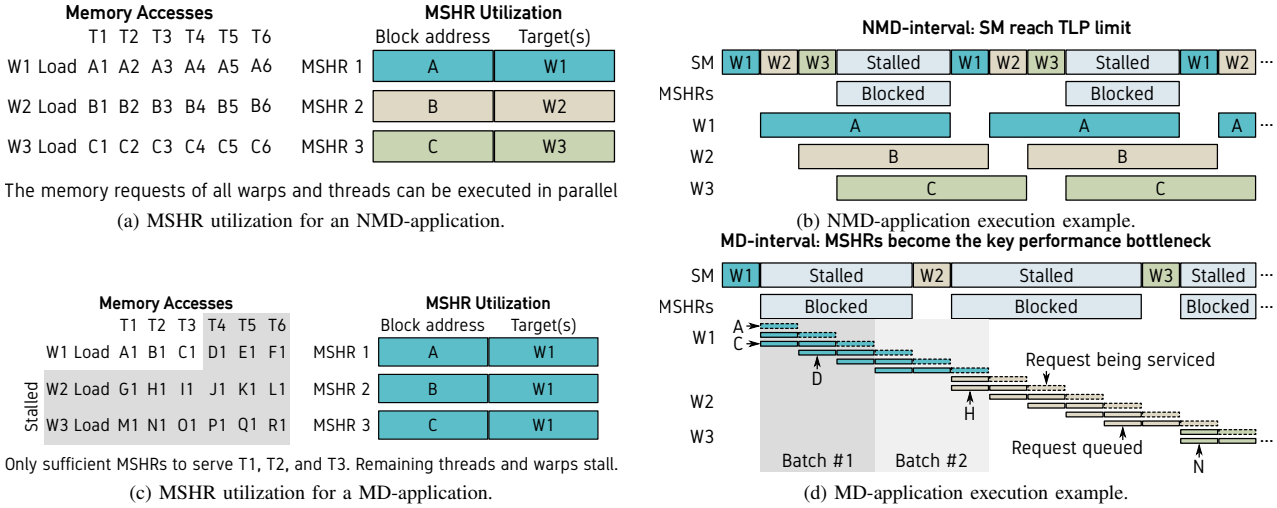
Fig. 3: L1 miss latency breakdown for selected GPU compute applications. *Delays due to insufficient MSHRs as well as queuing delays in the NoC and DRAM subsystem significantly affect the overall memory access latency of MD-applications, while NMD-applications are hardly affected.*

We now configure this example kernel as MD versus NMD by changing the GS parameter, which yields us two key observations regarding the performance characteristics of MD- versus NMD-workloads. We will later build upon these two observations when formulating the MDM model.

Observation #1: MD-kernels exhibit poor spatial locality which leads to widespread cache blocking due to lack of available MSHRs. Figure 2 illustrates this observation. When GS equals 1, the loads of each warp go to the same cache line, and the coalescer is able to combine these loads into a single cache request per warp. If GS equals 32, the kernel becomes memory-divergent because the memory accesses of each thread within the warp go to different L1 cache lines, and the coalescer can no longer combine the threads' cache requests. The poor spatial locality of MD-applications puts immense pressure on the L1 cache MSHRs which therefore become the predominant architectural bottleneck.

Figure 3 breaks down the L1 miss latency into the memory system units where the latency is incurred (see Section IV for details regarding our simulation methodology). We define MSHR delay as the number of cycles it takes from when a memory request is generated until its L1 miss is issued, and the NoC (DRAM) queuing latency is the number of cycles from a request entering a queue until it leaves. Only LLC misses enter a DRAM queue. Unsurprisingly perhaps, the overall memory latency is higher for MD-applications than for NMD-applications. Figure 3 shows that MD-applications spend a lot of time waiting for MSHRs to become available.

Observation #2: The poor spatial locality and high memory intensity of MD-applications causes widespread congestion in the NoC or DRAM subsystem. Figure 3 shows that MD-applications typically have much larger NoC or DRAM queuing delays than NMD-applications. To explain this behavior, we revisit Figure 2. When GS equals one, all words within the cache line are required by the SMs. Conversely, only a single word per cache line is required when GS equals 32. Put differently, the cache has to fetch 128 bytes to obtain the 4 bytes that are requested. This results in the NoC and DRAM being flooded with memory requests which in turn causes significant queueing delays.



(a) MSHR utilization for an NMD-application.

(b) NMD-application execution example.

(c) MSHR utilization for a MD-application.

(d) MD-application execution example.

Fig. 4: Example explaining why MSHR utilization results in significantly different performance-related behavior for NMD and MD-applications. *MD-applications puts immense pressure on the L1 cache MSHRs and thereby severely limit the GPU's ability to use TLP to hide memory latencies.*

B. Interval Analysis

It is clear from the above analysis that an accurate performance model should model both the impact of MSHR blocking and NoC/DRAM congestion. We first describe how these phenomena affect performance through interval analysis before describing the model in great detail in the next section. We do so using our example kernel, see Figure 4, while considering a single interval (i.e., a couple compute instructions followed by a long-latency load). Further, we assume 3 L1 cache MSHRs and that the warp scheduler can only consider 3 concurrent warps. Otherwise, the assumptions are the same as in Section III-A, including the assumption that all requests miss in the L1 cache.

A cache can sustain as many misses to different cache lines as there are MSHRs [22]. Each MSHR entry also tracks the destination (e.g., warp) for each request, and this parameter (commonly called the number of targets) determines the number of misses to the same cache line that the MSHR entry can sustain [16]. If the cache runs out of MSHRs (or targets), it can no longer accept memory requests until an MSHR (or target) becomes available. A blocked cache quickly causes the SM to stall because load instructions cannot execute.

The example NMD-kernel (i.e., GS equals 1) uses the MSHRs efficiently since the coalescer combines the loads of the threads within a warp into a single cache request. Therefore, Figure 4a shows that each warp occupies a single MSHR. Figure 4b shows execution behavior over time for the SM and the MSHRs as well as the cache requests of each warp. The SM first executes W1 which stalls when its threads reach the load instruction. To hide memory latency, the warp scheduler decides to execute W2. This enables W2's threads to calculate their addresses and issue their respective loads. The scheduler continues in a similar manner with W3. At this point, all concurrent warps available to the scheduler are stalled on loads, causing the SM to stall. The L1 cache also blocks because all its MSHRs are occupied, but this does not affect TLP since the SM has reached its TLP limit. Further, the performance impact of the stall is limited because execution resumes when W1's memory request returns.

Figure 4c shows the MSHR utilization for the example MD-kernel (i.e., GS equals 32). In this case, the cache misses of threads T1, T2, and T3 in warp W1 occupy all available MSHRs, and Figure 4d shows that the widespread cache blocking makes TLP-based latency hiding ineffective. More specifically, all forward progress in program execution occurs in response to memory requests completing. Therefore, switching to other warps does not help since their load requests cannot enter the memory system as long as the cache is blocked. This phenomenon occurs when the number of concurrent memory requests exceeds the number of L1 MSHRs. It thus depends on the application's memory access intensity and its access pattern, and its relationship with the number of MSHRs provided by the architecture. In other words, the application's characteristics and its interactions with the underlying architecture determines to what extent a kernel's memory divergence affects performance.

C. The Memory Divergence Model (MDM)

The above analysis illustrates that an accurate performance model for MD-applications needs to model the architectural effects of MSHR blocking (Observation #1)[2], and in addition needs to capture the effects of NoC and DRAM congestion (Observation #2). We now explain how our MDM model captures these observations.

The starting point of MDM is the number of cycles it takes for an SM to execute the instructions of interval i within

[2]Aggressive L1 cache designs such as the streaming cache [6], [21] have sufficient MSHRs to avoid MSHR-induced blocking. In this case, blocking occurs when the SM's NoC queue is full, and the size of this queue should replace the #MSHRs in the equations.

the representative warp without contention (i.e., C_i). We then add the predicted MSHR-related stall cycles (i.e., S_i^{MSHR}) and the predicted stall cycles due to queueing in the NoC and DRAM subsystems (i.e., S_i^{NoC} and S_i^{DRAM}) to C_i to predict the number of cycles an SM would use to execute interval i with contention (i.e., S_i). We can obtain per-interval IPC predictions by dividing the number of instructions in the interval by the number of cycles we predict that it will take to execute them (i.e., $\text{IPC}_i = \#\text{Instructions}_i / [C_i + S_i]$).

We predict the IPC of the entire warp by dividing the total number of instructions executed by the warp across all intervals with the total number of cycles required to execute all intervals. Then, we multiply by the number of warps concurrently executed on an SM (i.e., W) to predict IPC^{SM}:

$$\text{IPC}^{\text{SM}} = W \times \frac{\sum_{i=0}^{\#\text{Intervals}} \#\text{Instructions}_i}{\sum_{i=0}^{\#\text{Intervals}} C_i + S_i^{\text{MSHR}} + S_i^{\text{NoC}} + S_i^{\text{DRAM}}}. \quad (1)$$

We obtain IPC for the entire GPU by multiplying with the number of SMs (i.e., $\text{IPC} = \#\text{SMs} \times \text{IPC}^{\text{SM}}$). MDM obtains C_i similarly to GPUMech [15] while we provide new approaches for predicting S_i^{MSHR}, S_i^{NoC} and S_i^{DRAM}. If the application consists of multiple kernels, we first obtain the instruction count and total cycles for each kernel. Then, we divide the sum of instruction counts across all kernels by the sum of cycles across all kernels to predict application IPC. The following sections explain how MDM predicts the stall cycles S_i per interval; we omit the subscript i in the below discussion to simplify the formulation.

MDM classifies each interval within the application's representative warp as an MD or an NMD-interval. More specifically, memory divergence occurs when the number of concurrent read misses exceeds the number of L1 cache MSHRs:

$$M^{\text{Read}} \times W > \#\text{MSHRs}. \quad (2)$$

In other words, Equation 2 is true if the application has the ability to make the MSHRs the key performance bottleneck within the current interval.

1) MDM's batching model: We now further analyze the performance of the MD and NMD-kernels in Figure 4. To keep the analysis simple, we assume a constant NoC transfer latency and that all requests enter the same NoC queue and hit in the LLC. Note that these assumptions only apply to the example; MDM models the highly parallel memory system of contemporary GPUs.

For the NMD-kernel in Figure 4b, there are sufficient MSHRs for the SM to issue the memory requests of all three warps concurrently and reach the TLP limit. For the MD-kernel in Figure 4d on the other hand, divergence results in only the three first requests of W1 being issued before the cache blocks. As each request completes, an MSHR becomes available and a new request is issued. For instance, the completion of request A enables issuing request D. The poor MSHR utilization of the memory-divergent warp results in it issuing its requests in *batches*, and that each batch contains as many requests as there are MSHRs. More specifically, the requests of W1 are serviced over two batches since there are six memory requests and three MSHRs. W1 cannot execute its next instruction until the memory requests of all threads have completed (threads are executed in lockstep).

We now explain how MDM models batching behavior. We first predict the average number of concurrent L1 misses M:

$$M = \min(M^{\text{Read}} \times W, \#\text{MSHRs}) + M^{\text{Write}} \times W. \quad (3)$$

Read misses allocate MSHR entries and are therefore bounded by the number of L1 MSHRs. In other words, the application will either issue (1) the number of read misses in the representative warp times the number of concurrently executed warps; or, (2) as many read misses as there are MSHRs. Since the L1 caches in our GPU models are write-through and no-allocate, write misses effectively bypass the L1 and are independent of the number of MSHRs.

To estimate the length of the batches, we start by determining the memory latency in the absence of contention:

$$L^{\text{NoContention}} = L^{\text{MinLLC}} + \text{LLCMissRate} \times L^{\text{MinDRAM}}. \quad (4)$$

Here, L^{MinLLC} is the round-trip latency of an LLC hit without NoC contention. The round-trip latency through the DRAM system is L^{MinDRAM} (again, assuming no contention), but only LLC misses incur this latency. We then combine $L^{\text{NoContention}}$ with the average stall cycles due to queueing in the NoC and DRAM subsystems (we will derive S^{NoC} and S^{DRAM} in Section III-C2):

$$S^{\text{Mem}} = L^{\text{NoContention}} + S^{\text{NoC}} + S^{\text{DRAM}}. \quad (5)$$

S^{Mem} is the predicted stall cycles due to L1 misses — considering both NoC and DRAM contention. We then use S^{Mem} to predict the SM stall cycles due to MSHR contention:

$$S^{\text{MSHR}} = \begin{cases} (\lceil \frac{M^{\text{Read}} \times W}{\#\text{MSHRs}} \rceil - 1) \times S^{\text{Mem}}, & \text{if MD-interval} \\ 0, & \text{otherwise.} \end{cases} \quad (6)$$

For MD-intervals, Equation 6 computes the number of batches needed to issue the memory requests of all warps by dividing the total number of read misses by the number of MSHRs. The latency of the final batch is covered by the queueing model (see Section III-C2), so we need to subtract one from this quantity to avoid adding this latency twice. Then, we multiply by S^{Mem} to obtain the combined SM stall cycles of these batches. NMD-applications are by definition able to issue requests of all warps in a single batch. Therefore, we set S^{MSHR} to zero for NMD-intervals.

2) MDM's memory contention model: We now return to the example in Figure 4. Here, both the NMD- and MD-interval saturate the memory system (see Figure 4b and 4d, respectively), and this results in each request waiting for all requests that were pending at the time it issued. More specifically, both kernels sustain three memory requests in-flight since the SM can consider three warps concurrently (NMD-kernel) and there are three MSHRs (MD-kernel). Therefore, each request has to wait for two other requests.

The relationship between SM stall cycles and memory latency is complex due to the highly parallel execution model of GPUs. To account for this overlap in NMD-applications, we heuristically assume that the SM stalls for half of the memory queueing latency. The half-latency heuristic works well because the SM hides latencies by exploiting TLP. The case where the memory latency is fully hidden is detected by the interval model. Conversely, the highly parallel execution model of GPUs means that a significant part of the latency is typically hidden. Therefore, assuming that the SM stalls for half of the latency is reasonable since this is the mid-point between the extremes (i.e., perfect versus no latency hiding). For MD-applications, the SM is not able to use TLP to hide memory latency due to a lack of MSHRs, and the application is exposed to the complete memory queueing latency.

We now describe how MDM formalizes the above intuition. In a real GPU, memory contention occurs because the memory requests of all SMs queue up in the NoC and DRAM, and the NoC and DRAM use a certain number of cycles to service each request. More specifically, the NoC service latency $L^{\text{NoCService}}$ is a function of the cache block size, the clock frequency f and the NoC bandwidth B^{NoC}:

$$L^{\text{NoCService}} = f \times \frac{\text{BlockSize}}{B^{\text{NoC}}}. \tag{7}$$

The DRAM service latency can be computed in a similar way. However, only the LLC misses access DRAM:

$$L^{\text{DRAMService}} = f \times \text{LLCMissRatio} \times \frac{\text{BlockSize}}{B^{\text{DRAM}}}. \tag{8}$$

We obtain the LLC miss ratio from the interval profile and adjust the service latencies to account for parallelism in the memory system. More specifically, we divide the average service latency by n to model an n-channel system.

We now use the service latency to predict the SM stall cycles caused by queueing latencies. The average queueing latency is determined by the average number of in-flight requests M times the average service latency. M is determined by application behavior, and we use Equation 3 to predict it. The service latency is an architectural parameter which means that we can use the same model for both NoC and DRAM stalls by providing $L^{\text{NoCService}}$ ($L^{\text{DRAMService}}$) as input to compute S^{NoC} (S^{DRAM}):

$$S^{\text{NoC}} = \begin{cases} \#\text{SMs} \times M \times L^{\text{NoCService}}, & \text{if MD and NoC saturated} \\ 0.5 \times \#\text{SMs} \times M \times L^{\text{NoCService}}, & \text{otherwise.} \end{cases} \tag{9}$$

Equation 9 formalizes the key observations of the above example. For NMD-intervals, the SM hides queueing latencies with TLP, and we assume that the SM stalls for half of the queueing latency. The half-latency heuristic works well for NMD-intervals because it is the mid-way point between the extremes of perfect latency hiding and no latency hiding. We know that these extremes do not occur in NMD-intervals because (1) the fully-hidden case is detected by the interval model of the representative warp, and (2) latency-hiding only completely breaks down in MD-intervals.

TABLE III: Simulator configuration.

Parameter	Value
Clock frequency	1.4 GHz
Number of SMs	28
No. warps per SM	64
Warp size	32 threads
No. threads per SM	2048
Warp scheduling policy	GTO scheduling
SIMT width	32
L1 cache per SM	48 KB, 6-way, LRU, 128 MSHRs
Shared LLC	3 MB total, 24×128 KB banks 8-way, LRU, 128 MSHRs, 120 cycles
NoC bandwidth	1050 GB/s
DRAM	480 GB/s, 220 cycles 24 memory controllers

TABLE IV: Benchmarks.

Benchmark	Source	#Knl	Abbr.	Type
Hotspot	Rodinia	1	HS	NMD
B+trees	Rodinia	2	BT	NMD
Back Propagation	Rodinia	2	BP	NMD
FDTD3d	SDK	1	FDTD	NMD
Srad	Rodinia	2	SRAD	NMD
Ray tracing	GPGPUsim	1	RAY	NMD
2D Convolution	Polybench	1	2DCONV	NMD
Stencil	Parboil	1	ST	NMD
CFD solver	Rodinia	10	CFD	MD
Breadth-first search	Rodinia	24	BFS	MD
PageView Rank	MARS	258	PVR	MD
PageView Count	MARS	358	PVC	MD
Inverted Index	MARS	158	IIX	MD
Sparse matrix mult.	Parboil	1	SPMV	MD
Kmeans clustering	Rodinia	1	KMEANS	MD
AlexNet	Tango	22	AN	MD
ResNet	Tango	222	RN	MD

We have already established that a kernel issues more concurrent misses than there are L1 cache MSHRs in an MD-interval (see Equation 2). For this behavior to disable TLP-based latency-hiding, the NoC must saturate. More specifically, the predicted NoC queue latency must be larger than the minimum round-trip DRAM access latency:

$$L^{\text{NoCService}} \times M \times \#\text{SM} > L^{\text{MinLLC}} + L^{\text{MinDRAM}}. \tag{10}$$

If the NoC queue does not saturate, memory requests that hit in the LLC will be serviced quickly. This unblocks the L1 cache and enables the SM to issue a new memory instruction. Without NoC saturation, this occurs frequently enough for TLP-based latency-hiding to work also for MD-applications. Interestingly, detecting NoC saturation also enables MDM to model the streaming L1 cache [6], [21] (e.g., used in NVIDIA's Volta architecture) which has sufficient MSHRs to avoid L1 cache blocking. In this case, batching behavior occurs because only a finite number of requests can be buffered in the queues of the NoC at any given time and detecting NoC saturation is sufficient to identify MD-intervals.

IV. Experimental Setup

Simulation Setup. We use GPGPU-sim 3.2 [2], a cycle-accurate GPU simulator, to evaluate MDM's prediction accuracy. We choose the same baseline GPU architecture con-

TABLE V: Evaluated performance models.

Scheme	Memory/NoC Model	MSHR Model
GPUMech	GPUMech	GPUMech
GPUMech+	GPUMech with NoC	GPUMech w/ queue
MDM-Queue	MDM (Section III-C2)	GPUMech w/ queue
MDM-MSHR	GPUMech with NoC	MDM (Section III-C1)
MDM	MDM (Section III-C2)	MDM (Section III-C1)

figuration as in [15] for fair comparison against GPUMech, while scaling the number of SMs, NoC bandwidth and DRAM bandwidth to match a GPU architecture that is similar to Nvidia's Pascal GPU [29], see Table III.

Workloads. We select 17 applications: 8 NMD-applications and 9 MD-applications, from the main GPU benchmark suites, including Rodinia [5], SDK [26], Polybench [11], Parboil [32], MARS [13] and Tango [20] (see Table IV for details). We simulate the benchmarks to completion with the (largest) default input set. The inputs for AlexNet and ResNet [19] are pre-trained models from ImageNet [8].

Real Hardware Setup. We extend the NVBit [33] binary instrumentation framework to capture per-warp memory address and dynamic instruction traces. We collect our traces on an NVIDIA GeForce GTX 1080 [28] with caching of global data enabled, and we determine undisclosed parameters such as NoC bandwidth using microbenchmarks.

Performance Models. The original GPUMech proposal includes a DRAM queueing model which assumes that each request waits for half the total number of requests on average. However, GPUMech does not model NoC queueing delay, nor does it account for the DRAM and NoC queueing delays when estimating the MSHR stall latencies. We enhance GPUMech minimally and call it GPUMech+: it models the NoC queueing delay similarly to its DRAM queueing model, and also accounts for the NoC and DRAM queueing delays when estimating the MSHR waiting time. MDM-Queue improves upon GPUMech+ by using MDM's NoC and DRAM queue model. MDM-MSHR improves upon GPUMech+ by using MDM's MSHR batching model. MDM is our final model and includes both the NoC and DRAM queue model from Section III-C2 and the MSHR batching model from Section III-C1 (see Table V). This breakdown enables us to independently evaluate MDM's queue model and MSHR model.

V. Simulation Results

We first evaluate MDM's prediction accuracy for our baseline configuration through simulation, and consider real hardware validation in the next section (simulation enables us to demonstrate MDM's accuracy across a broad design space).

A. Model Accuracy

We evaluate MDM's accuracy by comparing the model's prediction against detailed cycle-accurate simulation with the absolute relative prediction error as our metric:

$$\text{Error} = \left| \frac{\text{IPC}_{\text{model}} - \text{IPC}_{\text{simulation}}}{\text{IPC}_{\text{simulation}}} \right| \quad (11)$$

$\text{IPC}_{\text{simulation}}$ is obtained through cycle-accurate simulation, and $\text{IPC}_{\text{model}}$ is obtained through modeling. Ideally, a model should have low absolute error, but in many cases it can be sufficient to accurately track relative performance trends across the design space (see Section V-B).

Figure 5 reports prediction error for the NMD and the MD-applications for our baseline configuration. GPUMech is largely inaccurate, especially for the MD-applications with an average prediction error around 298% and as high as 750%. MDM improves prediction accuracy by 16.5× compared to GPUMech for the MD-applications: MDM reduces the prediction error to 18% on average, and at most 50%. Similar prediction accuracy is achieved by GPUMech and MDM for the NMD-applications (average prediction error around 9%). On average across all benchmarks, MDM achieves a prediction error of 13.9% versus 162% for GPUMech.

The alternative performance models, GPUMech+, MDM-Queue and MDM-MSHR shed light on the relative importance of the different MDM model components. Although GPUMech+ improves accuracy significantly compared to GPUMech, it still incurs a high average prediction error of 131% for the MD-benchmarks. This shows that minorly modifying GPUMech is insufficient and that MD-applications need a fundamentally new modeling approach. MDM-Queue improves upon GPUMech+ by applying the saturation model described in Section III-C2 to memory-divergent intervals, thereby reducing the average prediction error to 63%. Similarly, MDM-MSHR improves upon GPUMech+ by applying the batching model of Section III-C1 to memory-divergent intervals which reduces the average prediction error to 60.3%. Neither MDM-Queue nor MDM-MSHR are able to accurately predict MD-benchmark performance in isolation, indicating that modeling both queueing effects and MSHR behavior is critical to achieve high accuracy.

B. Sensitivity Analyses

The prediction accuracy numbers reported in the previous section considered the baseline GPU configuration. Although it strengthens our confidence to know that MDM is accurate in a specific design point, a computer architect typically cares more about the accuracy of a performance model across a design space. This section evaluates this aspect by varying various important configuration parameters including NoC bandwidth, DRAM bandwidth, number of MSHR entries, and SM count. We focus on the MD-applications in this section because model accuracy for MDM is similar to GPUMech for the NMD-applications as previously shown for the baseline configuration — we observe similar results across the design space (not shown here because of space constraints).

NoC Bandwidth. Figure 6 reports model accuracy as we vary NoC bandwidth. GPUMech does not model NoC bandwidth and is hence highly sensitive to this parameter. At low NoC bandwidth, the NoC is a critical performance bottleneck, and GPUMech shows the highest performance prediction error. For GPUs with high NoC bandwidth, the NoC does not impact performance as significantly, which leads to a relatively low

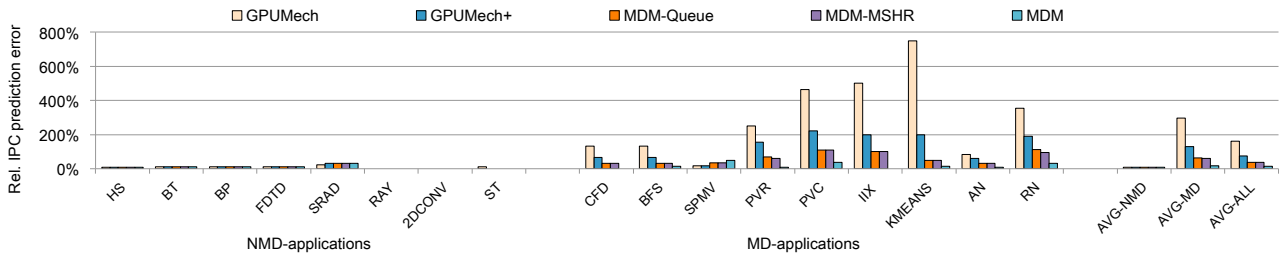


Fig. 5: IPC prediction error for our NMD and MD-benchmarks under different performance models. *MDM significantly reduces the prediction error for the MD-applications.*

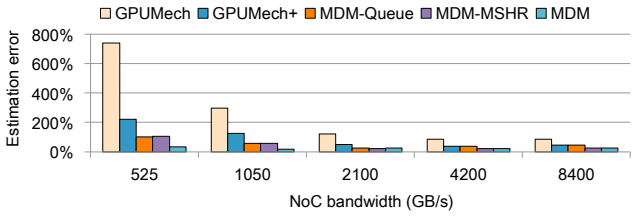


Fig. 6: Prediction error as a function of NoC bandwidth for the MD-applications.

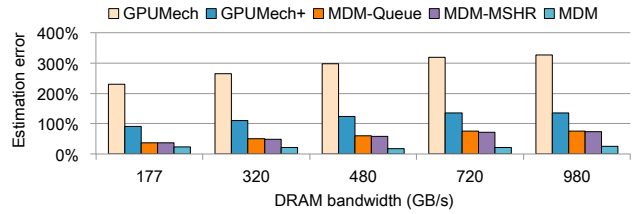


Fig. 7: Prediction error as a function of DRAM bandwidth for the MD-applications.

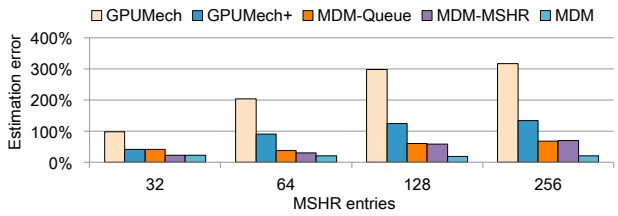


Fig. 8: Prediction error as a function of the number of MSHR entries for the MD-applications.

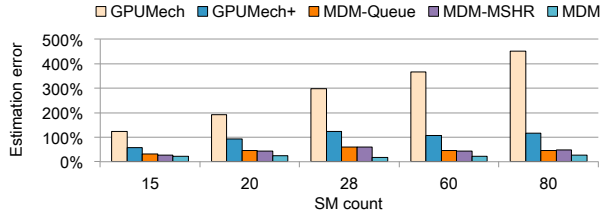


Fig. 9: Prediction error as a function of SM count for the MD-applications.

prediction error for GPUMech. GPUMech+ incorporates a basic NoC model which is improved upon by MDM-Queue. As a result, GPUMech+ and MDM-Queue are less sensitive to constrained NoC bandwidth configurations, yielding lower prediction errors. However, none of these models capture the impact of MSHRs. MDM-MSHR improves accuracy, especially at larger NoC bandwidths, where performance is less bound by NoC bandwidth. MDM significantly improves model accuracy at smaller NoC bandwidths, because it accounts for the impact MSHRs have on NoC bandwidth pressure. Overall, MDM is accurate across the range of NoC bandwidths.

DRAM Bandwidth. Figure 7 reports model accuracy across DRAM bandwidth configurations. GPUMech's prediction error increases with DRAM bandwidth: increasing DRAM bandwidth puts increased pressure on NoC bandwidth, which GPUMech does not model. GPUMech+ models the NoC queueing delay and the corresponding L1 miss stall cycles, which significantly decreases the prediction error. MDM-Queue and MDM-MSHR further improve accuracy through improved queueing and MSHR models, respectively. However, prediction error still increases with DRAM bandwidth. MDM counters this trend by synergistically modeling saturation and batching behavior.

MSHR Entries. Figure 8 shows model accuracy sensitivity to the number of MSHRs. GPUMech's prediction accuracy deteriorates with increasing MSHR entries ranging from 98% (32 MSHRs) to 317% (256 MSHRs). More MSHR entries leads to an increase in NoC and DRAM queuing delays because the system has to process more in-flight memory requests. GPUMech+ incorporates NoC and DRAM queueing delays when calculating the MSHR waiting time, which decreases the prediction error. However, the error is still high for a large number of MSHRs (e.g., 124% for 128 MSHRs and 133% for 256 MSHRs). MDM-Queue and MDM-MSHR significantly decrease the prediction error compared to GPUMech+ by using MDM's NoC/DRAM queue model and MSHR model, respectively. MDM achieves the highest accuracy of all models across the range of MSHRs.

SM Count. Figure 9 reports prediction error as a function of SM count. In general, increasing the number of SMs increases NoC and DRAM contention, leading to longer L1 miss stall cycles. GPUMech significantly underestimates the L1 miss stall cycles, which leads to prediction errors ranging from 124% (15 SMs) to 450% (80 SMs); the increase in prediction error is a direct result of increased queueing delays which GPUMech

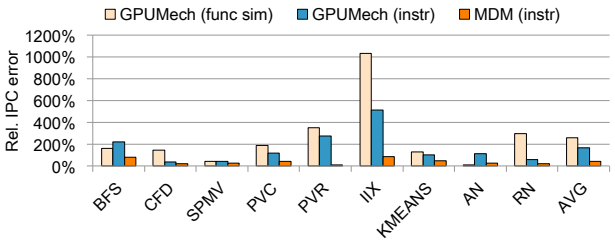


Fig. 10: Hardware validation: relative IPC prediction error for GPUMech and MDM compared to real hardware. *MDM achieves high prediction accuracy compared to real hardware with an average prediction error of 40% compared to 164% for GPUMech (using binary instrumentation).*

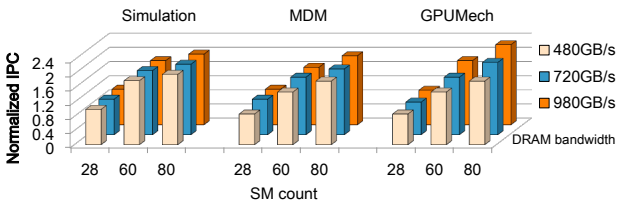


Fig. 11: Normalized performance for BT as a function of SM count and DRAM bandwidth. We normalize to the simulation results at 28 SMs and 480 GB/s DRAM bandwidth. *Both GPUMech and MDM capture the performance trend.*

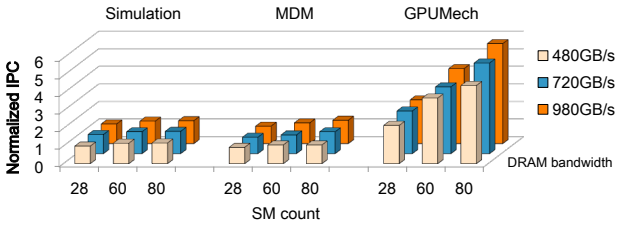


Fig. 12: Normalized performance for CFD as a function of SM count and DRAM bandwidth. All results are normalized to the simulation results at 28 SMs and 480 GB/s DRAM bandwidth. *GPUMech not only leads to high prediction errors, it also over-predicts the performance speedup with more SMs and memory bandwidth, in contrast to MDM.*

does not model. In contrast, GPUMech+ (partially) accounts for NoC and DRAM queuing delays which significantly decreases the prediction error. MDM-Queue and MDM-MSHR further improve accuracy by modeling memory saturation and batching behavior. By combining both model enhancements, MDM reduces the prediction error to less than 26% on average compared to detailed simulation across the different SM counts.

VI. Real Hardware Results

We now move to real-hardware validation. As mentioned in Section II, the model's input can be collected using either binary instrumentation on real hardware or through functional simulation. Because architectural simulation in GPGPUsim also operates at the PTX level, we used functional simulation in the previous section, for both MDM and GPUMech. Note that functional simulation is done at the intermediate PTX instruction level, and that GPUMech collects traces through functional simulation. In this work, we novelly collect traces using binary instrumentation, which is a better alternative, both in terms of accuracy and modeling speed, when comparing against real hardware which executes native-hardware instructions. We now evaluate modeling speed and accuracy.

Modeling Speed. Overall model evaluation time consists of trace collection, representative warp selection, cache simulation and computing the model's equations, as previously described in Section II. Collecting traces using functional simulation takes up around 84% of the total model evaluation time. Because binary instrumentation using NVBit is around 612× faster than functional simulation using GPGPUsim to collect traces, we achieve an overall model evaluation speedup of 6.1× through binary instrumentation.

Accuracy. Figure 10 validates MDM's (and GPUMech's) model accuracy against real hardware. We consider binary instrumentation for MDM and both instrumentation and functional simulation for GPUMech. There are two key observations to take away from this result. First, binary instrumentation improves the accuracy of GPUMech considerably compared to functional simulation, i.e., the average prediction error reduces from 260% to 164%. The reason is that profiling happens at the native instruction level using instrumentation instead of the intermediate PTX level using functional simulation. In spite of the improved accuracy, GPUMech still lacks high accuracy compared to real hardware. Second, MDM significantly improves the prediction accuracy compared to real hardware with an average prediction error of 40%. Not only is MDM accurate compared to simulation as reported in the previous section, MDM is also shown to be accurate compared to real hardware.

VII. Case Studies

We now consider four case studies to illustrate the usefulness of the MDM performance model.

Design Space Exploration. The most obvious use case for the MDM model is to drive design space exploration experiments to speed up the design cycle. Note we are not arguing to replace detailed cycle-accurate simulation. Instead, we are advocating to use the MDM model as a complement to speed up design space exploration, i.e., a computer architect could use MDM to quickly explore the design space and identify a region of interest which can then be further analyzed in more detail through cycle-accurate simulation.

Figures 11 and 12 report performance results for a typical design space exploration study in which we characterize performance as a function of SM count (X axis) and DRAM bandwidth (Y axis) for a representative NMD-application (BT) and MD-application (CFD), respectively. Performance numbers are reported for simulation, GPUMech and MDM; all results are normalized to simulation with 28 SMs and 480 GB/s memory

bandwidth. GPUMech and MDM capture the trend well for BT, the NMD-application. However, for CFD, the MD-application, GPUMech grossly overestimates performance as a function of both SM count and DRAM bandwidth. MDM shows that the performance improvements obtained by increasing SM count and/or DRAM bandwidth is small, which simulation confirms. GPUMech on the other hand suggests that there are significant performance gains to be obtained by increasing the number of SMs and DRAM bandwidth, which is a misleading conclusion. This reinforces that an accurate performance model such as MDM is needed to accurately predict performance trends for memory-divergent workloads.

Dynamic Voltage and Frequency Scaling (DVFS). DVFS is a widely used technique to improve energy efficiency [23]. Reducing the operating voltage and frequency dramatically decreases power consumption (i.e., dynamic power consumption decreases cubically with voltage and frequency) while only linearly decreasing performance. Moreover, memory-bound applications might observe only a slight performance degradation. Hence, DVFS can offer significant energy savings while incurring a (relatively) small loss in performance.

CRISP [24] is an online DVFS performance model for GPUs that predicts performance at a lower clock frequency based on statistics measured using special-purpose hardware performance counters at a nominal frequency. Based on the statistics measured at nominal frequency, CRISP then predicts how stall cycles scale when lowering clock frequency. Note that CRISP is a special-purpose model, i.e., it can only be used to predict the performance impact of DVFS. MDM and GPUMech on the other hand are general-purpose models that can be used to predict performance across a broader design space in which we change SM count, NoC/DRAM bandwidth, etc., as previously shown.

The difference in scope between MDM and CRISP is important because the modeling problem for a special-purpose model is much simpler than for a general-purpose model. A special-purpose model such as CRISP *measures* the stall component at the nominal frequency and then predicts how this component scales at a lower frequency. In contrast, a general-purpose model needs to *predict* the various stall components at the nominal frequency *and* at the target frequencies to then predict how performance scales with clock frequency. Predicting how a stall component scales with frequency is much easier than predicting the absolute value of the stall component at different frequencies. In this sense, it is to be expected that CRISP is more accurate than MDM (and GPUMech) for predicting DVFS scaling trends.

Figure 13 reports the error for predicting the execution time at 1.4 GHz compared to 2 GHz. For MDM and GPUMech, this means we predict performance at both frequency points and then compute the performance difference. For CRISP, we first run at 2 GHz and then predict performance at 1.4 GHz. CRISP is the most accurate model (average prediction error of 3.7%), closely followed by MDM (average prediction error of 4.6%); GPUMech on the other hand leads to much higher inaccuracy

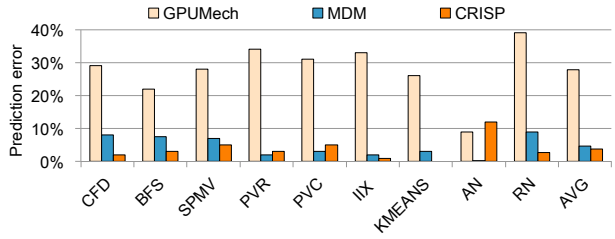


Fig. 13: Error when predicting the relative performance difference at 1.4 GHz versus 2 GHz for the MD-applications for GPUMech, CRISP and MDM. *The general-purpose MDM model achieves similar accuracy as the special-purpose CRISP.*

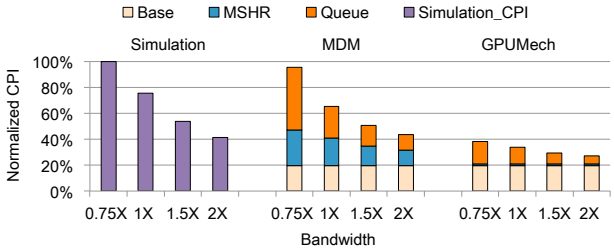


Fig. 14: Normalized CPI as a function of NoC/DRAM bandwidth for the memory-divergent BFS benchmark. *MDM accurately captures MSHR batching and NoC/DRAM queueing delays, in contrast to GPUMech.*

(average prediction error of 28%) because it underestimates the memory stall component. We find that CRISP's relatively high error for AN is due to CRISP assuming memory access time (in seconds) being constant whereas in reality it increases because of reduced overlapping with computation at lower clock frequency. Overall, we conclude that the general-purpose MDM model is only slightly more inaccurate than the special-purpose CRISP model for predicting the performance impact of DVFS.

Validating the Observations. In our third case study, we validate the observations that underpin the MDM model. Figure 14 reports CPI for BFS while varying NoC and DRAM bandwidth relative to our baseline configuration for simulation. Again, we observe that MDM accurately predicts performance compared to simulation, in contrast to GPUMech. Moreover, we note that the CPI component breakdown for MDM shows that MD-application performance is indeed sensitive to MSHR blocking (Observation #1) and NoC/DRAM queueing delay (Observation #2). GPUMech models some DRAM queueing effects, but it lacks an MSHR batching model as well as an accurate NoC/DRAM queueing model.

Modeling a Streaming L1 Cache. In our final case study, we show that MDM is able to model the streaming L1 cache used in the Volta GPU [6], [21]. MDM performs equally well as for the conventional L1 caches and improves prediction accuracy by 8.66× compared to GPUMech (see Figure 15). The streaming cache is interesting because it adds a large number of MSHRs to each L1 cache (4096 MSHRs in this case study) and adopts an on-fill line allocation policy. This

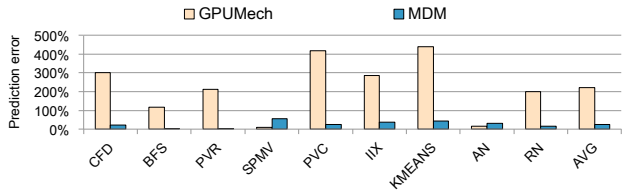


Fig. 15: Relative IPC prediction error with a streaming L1 cache. *MDM improves accuracy compared to GPUMech because it models batching behavior caused by NoC saturation.*

practically removes L1 blocking, but batching behavior still occurs because the queues in the NoC can only buffer a finite number of concurrent requests.

VIII. Related Work

Analytical GPU Performance Modeling. Several analytical models have been proposed to predict performance and identify performance bottlenecks. Unfortunately, none of these prior works targets memory-divergent applications. Hong and Kim [14] propose a model that estimates performance based on the number of concurrent memory requests and the concurrent computations done while one warp waits for memory. Baghsorkhi et al. [1] concurrently proposed a work-flow graph (WFG) based analytical model to predict performance. WFG is an extension to control-flow graph analysis in which nodes represent instructions while arcs represent various latencies. Sim et al. [31] build a performance analysis framework that identifies performance bottlenecks for GPUs by extending the Hong and Kim model. Similarly, Zhang and Owens [38] use a microbenchmark-based approach to model the performance of the instruction pipeline, shared memory and global memory of a GPU. In general, all these models, as well as the state-of-the-art GPUMech [15], make several simplifying assumptions regarding the GPU's cache hierarchy. Not modeling MSHR batching and inaccurately modeling NoC/DRAM bandwidth queueing delays leads to significant prediction errors for MD-applications, as reported in this paper. Volkov [34] studies GPU performance using synthetic benchmarks and confirms that several of these recently-proposed GPU models do not accurately capture the effects of memory bandwidth, non-coalesced accesses, and memory-intensive applications.

ML-Based GPU Performance Modeling. A separate line of research exploits machine learning (ML) for GPU performance modeling. These techniques are black-box approaches that do not reveal the impact of GPU components on performance nor help identify performance bottlenecks. Their effectiveness for design space exploration requires extensive training involving a wide range of applications and hardware configurations. Wu et al. [36] build GPU performance and power models by using numerous runs on several hardware configurations, and by feeding relevant performance counter measurements to the trained model for power and performance prediction. Poise [9] is a technique to balance the conflicting effects of increased TLP and memory system congestion due to high TLP in GPUs. Poise trains a machine learning model offline on a set of benchmarks to learn the best warp scheduling decisions. At runtime, a prediction model selects the warp scheduling decision for unseen applications using the trained performance model.

Special-Purpose Models. Several runtime optimization techniques rely on special-purpose models, in contrast to MDM which is a general-purpose model suitable for broad design space exploration. For example, CRISP [24] predicts the performance impact of varying the operating frequency for GPUs. It accounts for artifacts that affect GPU performance more than CPUs (thus ignored in CPU frequency scaling work), such as store-related stalls and the high overlap between compute and memory operations. Dai et al. [7] propose a model to estimate the impact of cache misses and resource congestion as a function of the number of thread blocks. They use these models to devise mechanisms to bypass the cache to improve performance.

Simulation-Based Approaches. Detailed GPU architecture simulation is time-consuming and researchers have proposed novel approaches for speeding up simulation. Wang et al. [35] propose a modeling approach that relies on source code profiling to extract warp execution behavior and an execution trace. They feed this information to a fast high-abstraction trace-based simulator for performance estimation. Yu et al. [37] propose a framework for generating small synthetic workloads that are representative for long-running GPU workloads. In contrast, MDM is an analytical model and is therefore much faster than a simulator-based approach.

IX. Conclusion

We have now presented the GPU Memory Divergence Model (MDM) which accurately models the key performance-related behavior of emerging MD-applications while retaining high accuracy for NMD-applications. MDM faithfully models that the poor spatial locality and high memory intensity of MD-applications leads to frequent L1 cache blocking which cripples the ability of the SMs to use TLP to hide memory latencies. MDM achieves high accuracy compared to detailed simulation (13.9% average prediction error versus 162% for the state-of-the-art GPUMech model) and on real hardware (40% prediction error versus 164%) for our MD-applications. We further demonstrate MDM's usefulness for driving design space exploration, DVFS scaling and analyzing performance bottlenecks. Overall, MDM significantly advances the state-of-the-art in analytical GPU performance modeling in terms of scope, practicality and accuracy.

X. Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This work is supported through the European Research Council (ERC) Advanced Grant agreement No. 741097 and Research Foundation Flanders (FWO) grants No. G.0434.16N and G.0144.17N. Lu Wang is supported through a CSC scholarship and UGent-BOF co-funding. Magnus Jahre is supported by the Research Council of Norway (Grant no. 286596).

REFERENCES

[1] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 105–114, 2010.

[2] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 163–174.

[3] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2012, pp. 141–151.

[4] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonia, "Managing DRAM latency divergence in irregular GPGPU applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014, pp. 128–139.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[6] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and programmability," *IEEE Micro*, vol. 38, no. 2, pp. 42–52, 2018.

[7] H. Dai, C. Li, H. Zhou, S. Gupta, C. Kartsaklis, and M. Mantor, "A model-driven approach to warp/thread-block level GPU cache bypassing," in *Proceedings of the Design Automation Conference (DAC)*, 2016, pp. 1–6.

[8] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009, pp. 248–255.

[9] S. Dublish, V. Nagarajan, and N. Topham, "Poise: Balancing thread-level parallelism and memory system performance in GPUs using machine learning," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 492–505.

[10] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Transactions on Computer Systems*, vol. 27, no. 2, pp. 1–37, 2009.

[11] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[12] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers, "Lost in Abstraction: Pitfalls of analyzing GPUs at the intermediate language level," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 608–619.

[13] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce framework on graphics processors," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 260–269.

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

[15] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "GPUMech: GPU performance modeling technique based on interval analysis," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014, pp. 268–279.

[16] M. Jahre and L. Natvig, "A high performance adaptive miss handling architecture for chip multiprocessors," *Transactions on High-Performance Embedded Architectures and Compilers IV*, vol. 6760, 2011.

[17] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the International Conference on Multimedia (ICMM)*, 2014, pp. 675–678.

[18] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2004, pp. 338–349.

[19] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon, "Detailed characterization of deep neural networks on gpus and fpgas," in *Proceedings of the Workshop on General Purpose Processing Using GPUs (GPGPU)*, 2019, p. 12–21.

[20] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon, "Tango: A deep neural network benchmark suite for various accelerators," *CoRR*, vol. abs/1901.04987, 2019. [Online]. Available: http://arxiv.org/abs/1901.04987

[21] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[22] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 1981, pp. 81–87.

[23] P. Macken, M. Degrauwe, M. Van Paemel, and H. Oguey, "A voltage reduction technique for digital systems," in *Proceedings of the International Conference on Solid-State Circuits (ISSCC)*, 1990, pp. 238–239.

[24] R. Nath and D. Tullsen, "The CRISP performance model for dynamic voltage and frequency scaling in a GPGPU," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2015, pp. 281–293.

[25] CUDA parallel thread execution ISA. NVIDIA Corp. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[26] CUDA SDK code samples. NVIDIA Corp. [Online]. Available: https://developer.nvidia.com/cuda-downloads

[27] GPU Accelerated Libraries for Computing. NVIDIA Corp. [Online]. Available: https://developer.nvidia.com/gpu-accelerated-libraries.

[28] NVIDIA GeForce GTX1080. NVIDIA Corp. [Online]. Available: https://developer.nvidia.com/introducing-nvidia-geforce-gtx-1080

[29] NVIDIA GP100 Pascal Architecture. NVIDIA Corp. [Online]. Available: https://www.nvidia.com/object/pascal-architecture-whitepaper.html

[30] Profiler user's guide. NVIDIA Corp. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[31] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012, p. 11–22.

[32] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois, Tech. Rep., 2012.

[33] O. Villa, M. Stephenson, D. W. Nellans, and S. W. Keckler, "NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019, pp. 372–383.

[34] V. Volkov, "Understanding latency hiding on GPUs," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html

[35] X. Wang, K. Huang, A. Knoll, and X. Qian, "A hybrid framework for fast and accurate GPU performance estimation through source-level analysis and trace-based simulation," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 506–518.

[36] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 564–576.

[37] Z. Yu, L. Eeckhout, N. Goswami, T. Li, L. K. John, H. Jin, C. Xu, and J. Wu, "GPGPU-MiniBench: Accelerating GPGPU micro-architecture simulation," *IEEE Transactions on Computers*, vol. 64, no. 11, pp. 3153–3166, 2015.

[38] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 382–393.