Modeling Emerging Memory-Divergent GPU Applications

Lu Wang, Magnus Jahre, Almutaz Adileh, Zhiying Wang, and Lieven Eeckhout

Abstract—Analytical performance models yield valuable architectural insight without incurring the excessive runtime overheads of simulation. In this work, we study contemporary GPU applications and find that the key performance-related behavior of such applications is distinct from traditional GPU applications. The key issue is that these GPU applications are memory-intensive and have poor spatial locality, which implies that the loads of different threads commonly access different cache blocks. Such memory-divergent applications quickly exhaust the number of misses the L1 cache can process concurrently, and thereby cripple the GPU's ability to use Memory-Level Parallelism (MLP) and Thread-Level Parallelism (TLP) to hide memory latencies. Our Memory Divergence Model (MDM) is able to accurately represent this behavior and thereby reduces average performance prediction error by 14× compared to the state-of-the-art GPUMech approach across our memory-divergent applications.

1 Introduction

Quantitative evaluation is an essential part of the computer architect's tool box. Simulation is the most common evaluation tool since it enables detailed, even cycle-accurate, performance analysis. However, simulation is excruciatingly slow and hence parameter sweeps commonly require thousands of CPU hours. An alternative approach is analytical modeling, which captures the key performance-related behavior of the architecture with a set of mathematical equations. Analytical models are much faster than simulation — making them ideally suited for early-stage architectural exploration [1], [2] and helping programmers understand application performance [3], [4].

GPUs are the de facto standard platform for executing performance-critical applications. Their highly parallel execution model and high-performance memory system makes GPUs a popular choice for emerging applications such as data analytics [5], [6]. The diversity of modern-day GPU applications makes them challenging to model. Several contemporary GPU applications differ from traditional GPU-compute workloads because they put a much larger strain on the memory system. More specifically, they are memory-intensive and memory-divergent — i.e., the memory accesses from concurrently executing threads map to multiple cache lines. While simulators account for this behavior by modeling cycle-by-cycle activities, state-of-the-art GPU modeling approaches are unable to predict performance with sufficient accuracy.

Our objective is to provide an analytical performance model for GPUs that is able to accurately predict the performance of the various GPU applications, including divergent memory-intensive applications. Our starting point is interval modeling [7], which is a widely used approach for CPU performance evaluation. 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

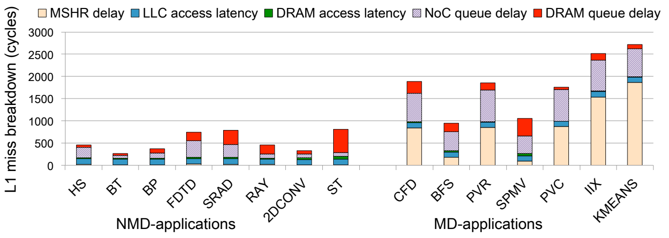

Fig. 1: L1 miss latency breakdown for select GPU-compute applications. *The key take-away is that delays due to insufficient MSHRs significantly affect the overall memory latency of MD-applications while NMD-applications are hardly affected.*

by predicting steady-state performance and subtracting the performance loss due to each miss event. GPUMech [1] applies interval modeling to GPUs. While GPUMech is accurate for traditional GPU-compute workloads, we find that it falls short for memory-divergent applications.

We propose the *Memory Divergence Model (MDM)* which captures the key performance-related behavior of modern, memory-divergent GPU applications. We find that the poor spatial locality of memory-divergent applications leads to inefficient utilization of the Miss Status Holding Registers (MSHRs). The number of MSHRs determines the number of concurrent misses the cache can sustain without blocking (a blocked cache cannot accept any requests). Blocking has a profound performance impact. First, a blocked cache limits the ability of the GPU core to hide memory latencies with Memory and Thread-Level Parallelism (i.e., MLP and TLP). Second, the memory system becomes saturated as the cores issue a large number of requests to fetch all required data. MDM accounts for these effects by accurately modeling MSHR behavior and the Network-on-Chip (NoC) and DRAM queuing latencies. Overall, MDM improves performance prediction accuracy by 14× on average compared to the state-of-the-art GPUMech [1] approach across our memory-divergent applications.

2 Understanding Emerging GPU Applications

2.1 The Architectural Effects of Memory Divergence

GPUs use multiple Streaming Multiprocessors (SMs) to execute code. Each SM can run a limited number of software threads concurrently. Thus, software threads are divided into groups, called warps, that match the width of the SM. An SM executes

- *Lu Wang, Almutaz Adileh and Lieven Eeckhout are with the Dept. of Electronics and Information Systems (ELIS), Ghent University, Belgium. E-mail: {luluwang.wang, almutaz.adileh, lieven.eeckhout}@ugent.be*
- *Magnus Jahre is with the Dept. of Computer Science, Norwegian University of Science and Tech., Norway. E-mail: magnus.jahre@ntnu.no*
- *Zhiying Wang is with the School of Computer, National University of Defense Technology, Changsha, P.R. China. E-mail: zywang@nudt.edu.cn*
- *Manuscript submitted: 21-May-2019. Manuscript accepted: 06-Jun-2019. Final manuscript received: 14-June-2019.*

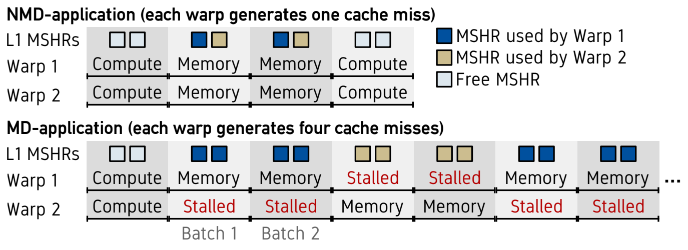

Fig. 2: MD-applications and NMD-applications have very different behavior. *The poor spatial locality of NMD-applications results in poor utilization of the L1 MSHRs — crippling the SM's ability to use TLP and MLP to hide memory latencies.*

the instructions of all threads within a warp in lock-step. For load instructions, each thread issues a load for a single data element. These per-thread requests are aggregated to cache requests by the coalescer. On a cache hit, the cache line is read by the SM. On a miss, an MSHR is allocated and a memory request is sent to the lower levels of the memory hierarchy. If the access pattern is favorable (e.g., sequential), the coalescer can map the misses of the warp's concurrent threads to a single cache request, consuming a single MSHR. However, a significant fraction of emerging GPU applications are memory-divergent (i.e., the threads of a warp tend to access different cache blocks), exerting significant pressure on the limited number of MSHRs. If the cache runs out of MSHRs, it blocks until an MSHR becomes available. A blocked cache causes SM stalls because no load instructions can be executed.

To understand how the poor spatial locality of memory-divergent applications affects the memory system, Figure 1 breaks down the average memory latency of GPU applications into the memory unit where it is incurred (see Section 4 for a description of our experimental setup). The key observation is that the benchmarks are clearly partitioned into two categories: The *Non-Memory Divergent (NMD)* benchmarks — where the latency due to insufficient MSHRs is negligible — and the *Memory Divergent (MD)* benchmarks — which on average spend hundreds and even thousands of cycles waiting for MSHRs to become available. Figure 1 also shows that MD-applications tend to experience significant queuing latencies in the NoC and DRAM subsystems. Thus, an effective performance model for MD-applications needs to accurately model MSHR behavior, NoC queuing and DRAM queuing.

2.2 Modeling Memory-Divergent Applications

Accurately modeling MSHR behavior and queuing in the NoC and DRAM requires understanding how the poor spatial locality of MD-applications interacts with the underlying architecture and how this interaction differs from NMD-applications. Figure 2 illustrates this with a simple example. We consider an NMD-application that has one L1 cache miss per warp and an MD-application with four cache misses per warp. Both warps of both applications first execute a couple of compute instructions. Then, they execute two load instructions (the second load depends on the first) before they finish by executing additional compute instructions. We assume that the SM can execute two warps concurrently and that the L1 cache has two MSHRs.

We first consider the NMD-application. For the first few cycles, Warp 1 and Warp 2 execute compute instructions at their steady-state IPC. Then, they both reach the load instructions that miss in the L1 cache (an L1 cache miss takes at least 120 cycles in our model, see Section 4). NMD-applications typically have good spatial locality across threads, and this enables the coalescer to combine the load instructions into a single cache request (one for each warp). The cache allocates an MSHR for each miss. Since there are two MSHRs in the cache, the misses are processed concurrently — uncovering MLP that successfully hides the latency of one of the requests. When the misses return, the MSHRs are freed. This enables the next misses to be issued in parallel as well. Since a realistic SM has a large number of warps in-flight and many MSHRs (128 in our model), MLP effectively hides memory latency in NMD-applications. The result is that memory latency has a limited impact on overall performance, and the simple memory latency models used in GPUMech [1] are sufficiently accurate to achieve low performance prediction errors for NMD-applications.

The key performance-related behavior of the MD-application is significantly different. Initially, Warp 1 and Warp 2 execute their compute-instructions concurrently. However, the four L1 cache misses of Warp 1 exceed the MSHR capacity of the L1 cache. This results in the misses being executed in *batches*. When Warp 1 has issued two miss requests, the L1 cache blocks and Warp 1 cannot execute its remaining cache requests. Further, Warp 2 cannot make progress as the L1 cache cannot service its memory requests either. After a few hundred clock cycles, the two first requests of Warp 1 return and it can issue its final two cache requests. When these requests return, Warp 2 can issue its two first requests. This causes Warp 1 to stall since its next instruction is a load instruction and the cache is blocked. Execution continues in batches until both warps have executed their load instructions (only partially shown).

The example explains how poor spatial locality leads to widespread L1 cache blocking. This limits the SM's ability to use MLP to hide memory latencies since the number of concurrent loads is limited by the number of MSHRs. Further, it also destroys TLP as all available warps will stall on their first load instruction because the L1 cache is mostly blocked. At the same time, the memory system saturates because it is flooded with requests that fetch little useful data (e.g., 128 concurrent requests from each L1 cache in our model), causing excessive NoC and DRAM queuing. This illustrates that an effective performance model for MD-applications must accurately model batching and saturation behavior. GPUMech falls short of this requirement, leading to high prediction errors for MD-applications.

3 The Memory Divergence Model (MDM)

In this section, we explain how MDM models the batching and saturation behavior of MD-applications. We leverage the framework used by GPUMech [1] to collect the interval profile and select a representative warp. The starting point of MDM is the number of cycles it takes an SM to execute the instructions of interval i within 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 queuing 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 summing the number of instructions executed by the warp across all intervals and dividing them by 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 the overall IPC^{SM} of an SM:

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

We obtain the IPC of the entire GPU by multiplying by the number of SMs (i.e., $\text{IPC} = \#\text{SMs} \times \text{IPC}^{\text{SM}}$). MDM obtains C_i following the approach of GPUMech [1], but we provide new approaches for predicting S_i^{MSHR}, S_i^{NoC} and S_i^{DRAM}. The following sections explain how MDM predicts the stall cycles S_i per interval (and we drop the subscript i from the discussion).

3.1 MDM's Memory and NoC Queue Models

Memory contention occurs because the memory requests of all SMs queue up in the NoC and DRAM subsystems. 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}:

$$L^{\text{NoCService}} = f \times \frac{\text{BlockSize}}{B^{\text{NoC}}} \tag{2}$$

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

$$L^{\text{DRAMService}} = f \times \text{LLCMissRatio} \times \frac{\text{BlockSize}}{B^{\text{DRAM}}} \tag{3}$$

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

We now use the service latency predictions to predict the average queuing latency — and thereby the SM stall cycles caused by queuing latencies. The average queuing latency is determined by the average number of pending requests an arriving request must wait for times the average service latency. We first predict the average number of concurrent L1 misses M:

$$M = \min(M^{\text{Read}} \times W, \#\text{MSHRs}) + M^{\text{Write}} \times W \tag{4}$$

Read misses allocate MSHR entries and are therefore bounded by the number of L1 MSHRs. In other words, the application will either: (1) issue the number of read misses of the current interval of the representative warp times the number of 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.

The number of queued requests is determined by application behavior while the service latency is an architectural parameter. Thus, we can use the same model to predict both NoC and DRAM stalls by providing $L^{\text{NoCService}}$ ($L^{\text{DRAMService}}$) as input to compute S^{NoC} (S^{DRAM}):

$$S^{\text{NoC}} = \begin{cases} \#\text{SMs} \times M \times L^{\text{NoCService}}, & M^{\text{Read}} \times W > \#\text{MSHRs} \\ (1/2) \times \#\text{SMs} \times M \times L^{\text{NoCService}}, & \text{otherwise} \end{cases} \tag{5}$$

The equation formalizes the key observations of Section 2. For MD-applications, the number of MSHRs is the bottleneck and the high degree of divergence keeps the memory system saturated. Since the memory system is saturated, each request needs to wait for all other requests. For NMD-applications, the memory requests are not sufficient to keep the memory queue saturated. In this case, the first request is serviced directly and the last request needs to wait for all other requests. Thus, a request waits for approximately half the concurrent requests.

TABLE 1: Simulator configuration.

Parameter	Value
Clock frequency	1.4 GHz
Number of SMs	28
Number of mem. ctrl.	24
Warp schedulers per SM	4 (LRR)
Issue width per sched.	2 warp-instructions/cycle
L1 cache per SM	48 KB, 6-way, LRU, 128 MSHRs
L2 cache per mem. ctrl.	128 KB, 8-way, LRU, 128 MSHRs
NoC bandwidth	1050 GB/s
DRAM bandwidth	480 GB/s
Maximum warps per SM	64
Minimum L2 hit latency	120 cycles
Minimum DRAM latency	220 cycles

TABLE 2: Benchmarks.

Benchmark	Suite	Abbr.	Type
Hotspot	Rodinia [8]	HS	NMD
B+trees	Rodinia	BT	NMD
Back Propagation	Rodinia	BP	NMD
FDTD3d	SDK [9]	FDTD	NMD
Srad	Rodinia	SRAD	NMD
Ray tracing	GPGPUsim [10]	RAY	NMD
2D Convolution	Polybench [11]	2DCONV	NMD
Stencil	Parboil [12]	ST	NMD
CFD solver	Rodinia	CFD	MD
Breadth-first search	Rodinia	BFS	MD
PageView Rank	MARS [13]	PVR	MD
RageView Count	MARS	PVC	MD
Inverted Index	MARS	IIX	MD
Sparse matrix mult.	Parboil	SPMV	MD
Kmeans clustering	Rodinia	KMEANS	MD

3.2 MDM's MSHR Contention Model

The warps of MD-applications send their requests to the memory subsystem over consecutive batches (see Section 2.2). To estimate the length of these batches, we start by determining the memory latency in the absence of contention:

$$L^{\text{NoContention}} = L^{\text{MinLLC}} + \text{LLCMissRate} \times L^{\text{MinDRAM}} \tag{6}$$

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 queuing in the NoC and DRAM subsystems (obtained with Equation 5):

$$S^{\text{Mem}} = L^{\text{NoContention}} + S^{\text{NoC}} + S^{\text{DRAM}} \tag{7}$$

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:

$$S^{\text{MSHR}} = \begin{cases} (\lceil \frac{M^{\text{Read}} \times W}{\#\text{MSHRs}} \rceil - 1) \times S^{\text{Mem}}, & M^{\text{Read}} \times W > \#\text{MSHRs} \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

Equation 8 checks whether the number of requests of the current warps exceeds the number of MSHRs. If it does, we compute 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 queuing model, 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 typically able to issue the requests of all warps in a single batch (see Figure 1). Therefore, we set S^{MSHR} to zero for non-divergent intervals.

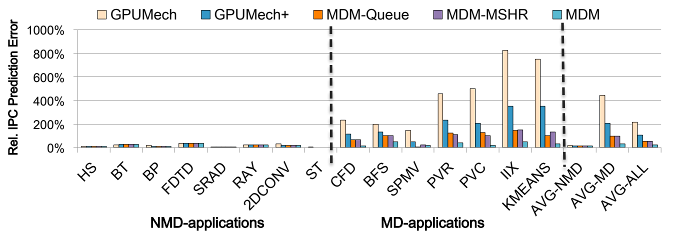

Fig. 3: IPC prediction error for our NMD and MD-benchmarks and the different performance models. *The key take-away is that MDM significantly reduces prediction error for MD-applications while minorly reducing error for the NMD-applications.*

4 Experimental Setup

We use GPGPU-sim 3.2 [10], a cycle-accurate GPU simulator, to evaluate MDM's prediction accuracy. We model an architecture similar to Nvidia's Pascal [14] as shown in Table 1. We select 15 applications: 8 NMD-applications and 7 MD-applications, from the main GPU benchmark suites. Table 2 provides details on the selected benchmarks. We simulate the benchmarks to completion with the (largest) default input set and report performance prediction error relative to simulated performance.

The original GPUMech proposal does not model NoC queuing delay and does not account for the DRAM and NoC queuing delays when estimating the MSHR stall latencies. GPUMech+ models a NoC queuing delay that resembles GPUMech's DRAM queuing model, whereby each request waits for half the total number of requests on average. GPUMech+ also accounts for the NoC and DRAM queuing 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 model. This enables us to independently evaluate MDM's queue model and MSHR model. MDM incorporates the improved NoC, DRAM and MSHR queuing delays.

5 Results

Figure 3 reports the relative IPC prediction error for our NMD and MD-benchmarks for all model combinations. MDM reduces prediction error by 14× on average compared to GPUMech for the MD-benchmarks, from 444% to 32%. For the NMD-benchmarks, MDM reduces prediction error marginally compared to GPUMech, from 19% to 16% on average. Across all benchmarks, GPUMech has an average performance prediction error of 217%. MDM achieves an average prediction error of 23%. The execution times of the MDM and GPUMech models are practically equal.

GPUMech+, MDM-Queue and MDM-MSHR shed light on the relative importance of the different components of MDM for the MD-applications. Although GPUMech+ improves accuracy significantly compared to GPUMech, it still has a high average prediction error of 206%. This reinforces 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 3.1 to memory-divergent intervals, thereby reducing the average prediction error to 100%. Similarly, MDM-MSHR improves upon GPUMech+ by applying the batching model of Section 3.2 to memory-divergent intervals, which reduces the average prediction error to 98%. Neither MDM-Queue nor MDM-MSHR are able to accurately predict MD-application performance in isolation, indicating that modeling both queuing effects and MSHR behavior is critical to achieve low prediction error.

6 Related Work

Prior work uses GPU modeling techniques to guide runtime optimizations (e.g., DVFS configuration [15] and cache miss-related optimizations [16]) or GPU resource scaling analysis [2]. Our work provides an accurate model for fast design space exploration. In general, prior performance modeling efforts make simplifications that lead to inaccuracies when modeling the cache hierarchy [4] and divergent applications [1], [3], or do not provide insight [2]. Volkov [17] studies GPU performance using simple synthetic benchmarks and shows that recent GPU models do not accurately capture the effects of memory bandwidth, non-coalesced accesses, and memory-intensive applications.

7 Conclusion

In this paper, we analyze the key performance characteristics of contemporary GPU applications and find that the poor spatial locality of these applications cause them to be memory-divergent. The modeling assumptions made by state-of-the-art GPU performance models such as GPUMech do not capture the characteristics of such applications. Applying GPUMech to memory-divergent applications leads to significant performance prediction errors (444% on average). We propose the Memory Divergence Model (MDM), which accurately models the batching and saturation behavior caused by high memory intensity and poor spatial locality. MDM significantly improves performance prediction accuracy compared to GPUMech, by 14× on average across a set of memory-divergent applications.

References

[1] J.-C. Huang *et al.*, "GPUMech: GPU performance modeling technique based on interval analysis," in *MICRO*, 2014.
[2] G. Wu *et al.*, "GPGPU performance and power estimation using machine learning," in *HPCA*, 2015.
[3] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *ISCA*, 2009.
[4] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *HPCA*, 2011.
[5] M. Burtscher *et al.*, "A quantitative study of irregular programs on GPUs," in *IISWC*, 2012.
[6] N. Chatterjee *et al.*, "Managing DRAM latency divergence in irregular GPGPU applications," in *SC*, 2014.
[7] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *ISCA*, 2004.
[8] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.
[9] NVIDIA CUDA SDK Code Samples. NVIDIA Corp. [Online]. Available: https://developer.nvidia.com/cuda-downloads
[10] A. Bakhoda *et al.*, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.
[11] S. Grauer-Gray *et al.*, "Auto-tuning a high-level language targeted to GPU codes," in *InPar*, 2012.
[12] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois, Tech. Rep., 2012.
[13] B. He *et al.*, "Mars: A MapReduce framework on graphics processors," in *PACT*, 2008.
[14] NVIDIA GP100 Pascal Architecture. NVIDIA Corp. [Online]. Available: https://www.nvidia.com/object/pascal-architecture-whitepaper.html
[15] R. Nath and D. Tullsen, "The CRISP performance model for dynamic voltage and frequency scaling in a GPGPU," in *MICRO*, 2015.
[16] 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 *DAC*, 2016.
[17] V. Volkov, "Understanding latency hiding on GPUs," Ph.D. dissertation, UC Berkeley, 2016.