2015 IEEE International Symposium on Workload Characterization

Revealing Critical Loads and Hidden Data Locality in GPGPU Applications

Gunjae Koo[1], Hyeran Jeon[2], Murali Annavaram[1]
[1]Ming Hsieh Department of Electrical Engineering, University of Southern California
[2]Department of Computer Engineering, San Jose State University
gunjae.koo@usc.edu, hyeran.jeon@sjsu.edu, annavara@usc.edu

***Abstract*—In grapichs processing units (GPUs), memory access latency is one of the most critical performance hurdles. Several warp schedulers and memory prefetching algorithms have been proposed to avoid the long memory access latency. Prior application characterization studies shed light on the interaction between applications, GPU microarchitecture and memory subsystem behavior. Most of these studies, however, only present aggregate statistics on how memory system behaves over the entire application run. In particular, they do not consider how individual load instructions in a program contribute to the aggregate memory system behavior. The analysis presented in this paper shows that there are two distinct classes of load instructions, categorized as *deterministic* and *non-deterministic* loads. Using a combination of profiling data from a real GPU card and cycle accurate simulation data we show that there is a significant performance impact disparity when executing these two types of loads. We discuss and suggest several approaches to treat these two load categories differently within the GPU microarchitecture for optimizing memory system performance.**

I. Introduction

Graphics processing units (GPUs) employ a large number of thread contexts and provide sufficient hardware resources to quickly switch between thread contexts. In spite of the enormous hardware resources expended to hide the memory access latency, it is still one of the primary performance bottlenecks of GPUs. Memory accesses to the DRAM memory typically take hundreds of GPU cycles. To mitigate the impact of long memory latency, several warp schedulers and prefetching algorithms have been proposed [15], [17], [22], [11]. Many of the proposed prefetching and scheduling algorithms are application oblivious mechanisms that do not take into account application specific behaviors. Recently, as the importance of application specific optimization is emphasized, detailed application characterization studies have been conducted by many researchers [14], [9], [7], [6], [23]. The purpose of these studies is to identify specific application characteristics that may not map well to the GPU microarchitecture. For instance, some of these studies analyzed graph analytics and other irregular applications [7], [6], [23], [16] and showed that these applications have many uncoalesced memory accesses which result in significant memory system bottlenecks. The common conclusions of these studies are that coalescing memory accesses can reduce memory traffic, DRAM bandwidth is the main performance bottleneck, and warp scheduling algorithm plays an important role in preserving data locality in small caches. Most of these studies, however, only present aggregate statistics on how memory system behaves over the entire application run. In particular, they do not consider how individual load instructions in a program contribute to the observed memory system behavior. However, as we show in this paper, within a single application execution run two different types of load instructions are executed and there is a significant performance impact disparity when executing these two types of loads. We make the case that GPU microarchitecture must handle these two types of loads differently in order to reduce memory system bottlenecks.

The followings are the contributions of this work:

- In this paper, we analyze 15 applications from four different GPU benchmark suites [8], [13], [6], [2] both on a GPU hardware and on GPGPU-Sim [5] simulator. We profile individual load instructions and their interactions with the underlying memory system.
- Using backward data flow analysis, we classify load instructions based on how their source addresses are computed. We define a load as $deterministic$ when its source address is generated from parameterized data such as CTA ids, thread ids, and constant parameters which are typically passed from the host to the device. These values are easily accessible during the start of the kernel execution using ld.param instruction. We define a load as $non-deterministic$ when its source address is generated from values obtained from prior load instructions executed in the kernel or user inputs that may vary from one run to another.
- We then re-examine memory system interactions of deterministic and non-deterministic loads separately. We show that a vast majority of the uncoalesced memory accesses originate from non-deterministic loads. The bursty memory traffic from these uncoalesced loads use up many memory management hardware resources to degrade overall system performance. We further provide an in-depth characterization of non-deterministic loads to reveal sources of their long turnaround time.
- Based on the above observations we discuss microarchitectural design changes to the GPU hardware for improving memory system performance.

The remainder of this paper is organized as follows. In Section II, we discuss previous studies and contrast them with our own observations. Section III explains the baseline GPU architecture. Section IV describes our evaluation methodology and the applications used in the experiment. In Section V, we describe how we classify the load instructions into two categories. We analyze the behavior of the two load categories

978-1-5090-0088-3/15 $31.00 © 2015 IEEE
DOI 10.1109/IISWC.2015.23

by using the evaluation results from Section VI to Section IX. We discuss the hardware optimization directions in Section X. We conclude in Section XI.

II. Related Work and Our Observations

There have been several studies that provide insights into the microarchitectural behavior of various GPU applications. In this section, we summarize their findings and then contrast them with our work.

Hestness et al. [14] characterized memory sub-system behavior when the CPU and GPU use a unified system memory. They studied the interference caused between the CPU and GPU when both processors access the same physical memory. They also presented various memory metrics such as spatial locality, bandwidth, and performance impact of coalescing memory accesses. They noted that memory coalescing reduces the number of spatially local accesses to cache lines and bursty memory accesses resulting from uncoalesced accesses leads to higher off-chip bandwidth demands.

Che et al. [7] implemented eight graph applications in OpenCL and analyzed cache hit ratio, execution time breakdown from various microarchitectural bottlenecks, speedup over CPU version execution, and SIMT lane utilization while running those applications on an AMD Radeon HD 7950. They observed that memory-coalescing optimization is a challenge for graph algorithms due to their random access patterns, and graph workloads pose significant challenges for efficient GPU resource utilization due to their irregular data structures.

Burtscher et al. [6] investigated the performance impact of irregular GPU programs on an NVIDIA Quadro 6000. They compiled eight irregular programs and compared the performance in several aspects with a set of regular programs. They measured two runtime-independent metrics, the control-flow irregularity and the memory-access irregularity at a warp level. The key findings in their study are that control-flow irregularity and the memory-access irregularity are independent of each other, the irregularity does not change drastically for different input sizes, and memory-access irregularity and performance need not necessarily be negatively correlated.

O'Neil and Burtscher [21] further analyzed pipeline and memory behaviors of irregular GPU kernels on GPGPU-Sim and found that insufficient bandwidth and long memory latency are the bigger limiters of performance than branch divergence, load imbalance and synchronization overhead.

Xu et al. [23] compiled a group of graph applications written in CUDA and measured graph application's microarchitectural behavior on both a real GPU and a cycle accurate simulator. To identify graph applications' unique characteristics, they also measured the same metrics for non-graph applications. They found that graph applications tend to run small kernels more frequently and hence the data copy overhead is significant, long latency memory operation is the most critical performance bottleneck, and cache size is not correlated to the performance improvement.

But most of the prior studies treated all the loads in an application uniformly. Hence, they presented the memory system behavior results aggregated over all load instructions. In our work we separate loads into two categories, namely deterministic and non-deterministic loads, and present detailed memory behavior results for the two load categories. Our key observations are as follows:

- Even in an application that has highly irregular memory access patterns not all load instructions are uncoalesced. In fact majority of load instructions are highly coalesced.
- It is possible to separate coalesced and uncoalesced load instructions by tracking their source address calculations. Deterministic loads are likely to have coalescing access pattern and non-deterministic loads tend to have uncoalescing access pattern.
- Most memory system bottlenecks are due to non-deterministic loads that quickly take up hardware resources such as cache tags.
- Irrespective of whether a data block is accessed in a coalesced or uncoalesced manner there is still a significant amount of data block sharing across neighboring CTAs.

We will explain these findings in detail in the following sections.

III. GPU Architecture and Memory Sub-system

We use the NVIDIA terminology to describe the GPU background, although the results presented in the paper are not vendor-specific. GPUs provide massive parallel processing power. As the host for the GPU device, CPU organizes and invokes GPU kernel functions. Communication between the CPU and the GPU is performed via PCI-Express bus. GPUs consist of a number of streaming multiprocessors (SMs). Each SM consists of simple processing engines, called the CUDA cores. Within each core there are three function units, stream processors (SPs) that perform integer and floating point arithmetic, special functional units (SFU) that perform complex functional arithmetic, and load/store units (LD/ST). For instance, NVIDIA Tesla M2050 consists of 14 SMs, each comprising of 32 CUDA cores and 64 KB shared memory shared among the SPs in an SM. In M2050, up to 1536 hardware threads are supported by each SM and 21,504 threads are supported on the entire GPU.

The kernel function called by the host CPU is divided into several independent thread blocks called cooperative thread arrays (CTA) each assigned to a specific SM. Inside a thread block, a set of threads (32 in M2050), referred to as a warp, is scheduled on the SM to run concurrently. A warp is a collection of threads that all run the same sequence of instructions but using different data operands. This execution model is referred to as single instruction multiple thread (SIMT) model and each thread within a warp has a dedicated set of execution resources which are referred to as SIMT lanes.

In addition, each SM has a private memory sub-system consisting of a very large register file, several level-1 caches, which consist of texture, constant, data and instruction caches. Part of the level-1 data cache can be configured as a shared memory that is shared among threads in the same CTA. Apart from the per-SM private memory sub-system, SMs also share a large level-2 cache which is partitioned and accessed by SMs

Category	Name	Data set	Description	No. of CTAs	No. of threads / CTA	No. of total instructions	No. of global load instructions	Fraction of global loads
Linear	2mm [13]	2048×2048 matrix	matrix multiplication	4096	256	1000278752	181012416	18.10%
	gaus [8]	matrix1024	Gaussian elimination	65536	16	2006088832	60909694	3.04%
	grm [13]	2048×2048 matrix	Gram-Schmidt decomposition	8	256	1007891584	249403688	24.75%
	lu [13]	2048×2048 matrix	LU decomposition	16384	256	1098632384	73080127	6.65%
	spmv [2]	Dubcova3.mtx.bin	sparce matrix dense vector multiplication	765	192	4715880000	552942200	11.73%
Image	htw [8]	104 609×590 images	Heartwall tracking	51	256	1299999936	111322544	8.56%
	mriq [2]	64_64_64_dataset.bin	MRI calibration	1024	256	10218416128	2625536	0.03%
	dwt [8]	1024×1024 BMP image	2D discrete wavelet transform	44	64	242465792	5852718	2.41%
	bpr [8]	-	back propagation for image recognition	4096	256	227672864	8454224	3.71%
	srad [8]	458×502 PGM image	Speckle reducing anisotropic diffusion	16384	256	1089418912	38868320	3.57%
Graph	bfs [8]	graph1M	breadth first search	2048	512	1251913792	101141243	1.17%
	sssp [6]	rmat.gr	single source shortest path	2048	512	8656550976	90667239	5.66%
	ccl	Trojan.dat	connected component labeling	2500	256	1195456864	63880699	5.78%
	mst [6]	rmat12.syn.gr	minimum spanning tree	56	384	2318710368	3070316	1.19%
	mis	512	maximal independent set	14	1536	2178021984	2073376	0.19%

TABLE I. APPLICATION CHARACTERISTICS IN THREE CATEGORIES

via an interconnection network. At the end of the memory hierarchy multiple external DRAM chips, called global memory, are directly connected with a GPU device through memory controllers. Each memory controller is associated with one or more level-2 cache partitions. Any memory access request that is not in the private cache is then sent to the appropriate level-2 cache partition which in turn may send it the appropriate DRAM channel on a level-2 cache miss. If data has to be served from external DRAM chips, it takes hundreds or even thousands of cycles [24].

IV. Methodology

A. Applications

As described in the introduction the purpose of this study is to characterize the behavior of individual load instructions within an application and identify the unique properties of different loads that result in vast disparity in memory system interactions across different loads. To aid in this goal we first select 15 GPU applications from various benchmark suites [8], [13], [6], [2]. Descriptions for the applications and their input data sets used in this study are briefly summarized in Table I. Since the goal of this study is to characterize memory system behavior we used large data sets where available. Large data sets stress the memory system as the working sets do not fit within the level-1 or even level-2 cache. The selected applications may be broadly grouped into three categories based on their functionality - linear algebra, image processing and graph applications.

1) Linear algebra: The linear algebra applications implement various matrix arithmetic operations. For effective parallel computation, large matrices are split into lots of smaller matrices, which are then mapped to parallel threads that run concurrently on a GPU. Typically splitting matrices and other vector elements leads to fairly simple indexing mechanisms for accessing the individual elements of a sub-matrix by a thread as such the sub-matrix data accesses by each thread are indexed using a linear function of thread ids and CTA ids. One exception is spmv which handles sparse matrix computations. Only non-zero elements from a large sparse matrix are stored and hence when splitting the sparse matrices the resulting sub-matrix indices are computed from non-linear equations of thread and CTA ids.

2) Image processing: Image or video data can be easily represented as 2-D or 3-D arrays. Thus input data sets for image processing applications can also indexed with thread ids and block ids similar to the linear algebra applications. However many image processing algorithms have multiple sub-tasks which are executed in a pipelined fashion. Thus once a sub-region of raw data (pixels or transformed data) is fetched into local memory in an SM, each sub-task processes the data and then passes on its output to the next sub-task in the pipeline. Even though image processing algorithms basically have series of linear algebra operations, control paths of the image processing applications may be diverged since specific algorithms are selectively applied based on the properties of image data. For instance data sets are replicated or padded with zeros for pixels out of a frame area as wavelet algorithm, applied to dwt, is performed for regions near frame boundaries. We also include bpr in the category of image processing applications although bpr applies a machine learning algorithm to neural network layers. The underlying computation layer, however, uses pattern recognition algorithms which have strong similarity with many image processing algorithms.

3) Graph: The last group of applications implement graph algorithms. Input data sets for the graph applications consist of large number of vertices connected via edges which have a weight. Graph applications visit various vertices by traversing the edges, and thus data fetched by a thread is dependent on graph connectivity. For example, bfs performs the breadth first search algorithm to traverse vertices of graphs, thus the index for fetching data of a next vertex is decided by the index of the current vertex and edge connectivity between the current and the next vertex. Note that edges are randomly distributed between vertices for real graph data. Therefore, indices for data fetching are irregular for the graph applications.

Table I summarizes those application characteristics which are relevant to the GPU execution model. The column titled No. of CTAs presents the number of thread blocks in each application. The column titled No. of threads/CTA shows the size of each CTA. The two columns combined quantify how much parallelism is exposed to the hardware by each application. The column titled No. of total instructions shows the dynamic count of warp instructions executed by the program. Almost all applications execute multiple billions of instructions. The column titled No. of global load instructions indicates the number of load instructions that access global memory. Note that in GPUs loads that access global memory use a different instruction mnemonic than loads that access shared memory and other specialized memories. Hence, it is easy to identify

GPU			
Model	Tesla M2050 [18]		
Core	14 CUDA SMs@1.15GHz		
Memory	2.6GB, GDDR5@1.5GHz		
Comm.	PCI-E GEN 2.0		
Simulator			
Version	GPGPU-Sim v3.2.2 [5]		
Configs	Tesla C2050		
Core	14 CUDA SMs@1.15GHz, 32 SIMT width		
Memory	GDDR5@1.5GHz		
Register file	128KB	Shared memory	48KB
Const cache	8KB	Texture cache	12KB
L1D cache	16KB, 128B line, 4-way, 64 MSHR entries [19]		
L2D cache	Unified, 786KB, 128B line, 8-way, 32 MSHR entries		
ROP latency	120	DRAM latency	100

TABLE II. EXPERIMENT ENVIRONMENTS

Counter	Description
gld_request	Number of executed global load instructions per warp in a SM
shared_load	Number of executed shared load instructions per warp in a SM
l1_global_load_hit	Number of global load hits in L1 cache
l1_global_load_miss	Number of global load misses in L1 cache
l2_subp0_read_hit_sectors	Number of read requests from L1 that hit in slice 0 of L2 cache.
l2_subp1_read_hit_sectors	Number of read requests from L1 that hit in slice 1 of L2 cache.
l2_subp0_read_sector_queries	Accumulated read sector queries from L1 to L2 cache for slice 0 of all the L2 cache units
l2_subp1_read_sector_queries	Accumulated read sector queries from L1 to L2 cache for slice 1 of all the L2 cache units

TABLE III. THE PROFILER COUNTERS USED IN THIS STUDY [1]

all loads that access the global memory. The last column shows the fraction of instructions that are global loads. The ratio of the global load instructions is on average 6.43% for all instructions. The average ratios of the global load instructions for each of the three categories is 12.85%, 3.66% and 2.80%, respectively.

B. Experiment environment

The selected applications were written in CUDA and were compiled with the NVIDIA CUDA Toolkit 4.0 [20]. We collected a wide range of statistics by running these applications on the native GPU hardware as well as on a cycle accurate software simulator. For measurement on real hardware, we utilize the CUDA Profiler [1] to measure memory characteristics of the application while running on an NVIDA Tesla M2050 GPU, which has 14 CUDA SMs operating with 1.15 GHz. The profiler counters used in this study are listed in Table III.

We also ran all the applications on the GPGPU-Sim simulator [5] with the NVIDIA Tesla C2050 configuration parameters in order to collect other statistics that were not supported by the CUDA Profiler. It is our understanding that Tesla C2050 and M2050 have identical architecture except they use different heat sinks. Since simulation is a very slow process, compared to running on native hardware, we only simulated applications until they commit the first billion instructions. Detailed specification of the native GPU hardware and the GPGPU-Sim simulator configuration is shown in Table II.

V. CLASSIFYING LOADS

While previous studies presented aggregate memory system behavior of each application we study the behavior of individual load instructions within each applications. In this section we present one approach we used to classify the load instructions into two categories. This approach uses how load instructions compute their effective memory address to distinguish the two load categories.

```
int tid = blockIdx.x * MAX_THREADS_PER_BLOCK + threadIdx.x;
if (tid<no_of_nodes && g_graph_mask[tid]) {
    g_graph_mask[tid]=false;
    for(int i=g_graph_nodes[tid].starting; ...) {
        int id = g_graph_edges[i];
        if(!g_graph_visited[id])
        ...
    }
}
```

Code 1. bfs code example

As briefly described in Section IV, the threads in each of the three application categories use different types of indexing operations to compute the effective address of a data item they need to access. Code 1 shows an example of how a thread in the bfs benchmark computes the effective address of a graph node that it accesses. In this code, an array *g_graph_mask* is indexed by *tid* calculated as a linear equation of *blockIdx.x* and *threadIdx.x*, which are thread-specific parameters. Thus the *g_graph_mask* is indexed entirely using thread ids and block ids. Since the thread and block ids are constant values that do not change from one run to another the load instruction that accesses this array is termed as a *deterministic* load. In other words deterministic loads compute their effective address based on parameters that are known at the time a kernel is launched from the host to the device and these parameter values do not change during the kernel execution. Furthermore, as *threadIdx.x* value increases by one between threads in a warp, it is expected that consecutive threads within a warp access consecutive elements in the array. Hence, deterministic load instructions tend to generate coalesced memory accesses. On the other hand, *g_graph_edges* is indexed by *id*, which is itself loaded from an other vector *g_graph_edges*[i]. Since the index is computed from non-parameterized data such as user inputs or indirectly accessed using indices stored in another vector we categorize such a load as a *non-deterministic* load in this paper.

In order to accurately classify a load instruction into the two categories we rely on backward data flow analysis which is routinely used in compiler analysis [4]. We trace the dependency graphs backwards for a source register that is used in the address computation of a load. We identify the parent instructions that define the source register. We then recursively trace the source registers of the parent instructions to identify the grand-parents of the load. Tracing back of source registers is continued until we reach the point when it becomes clear how the load's source register is being defined. If the source register of a load is defined from parameterized data such as block ids, thread ids and constant parameters then that load is classified as deterministic. In the CUDA environment it is easy to identify when a parameterized data value is used to define a load's source register. All the parameterized data values are loaded using a special *ld.param* instruction in CUDA. On the other hand, if a load's source register is defined from prior load instructions such as *ld.global*, *ld.local*, *ld.shared* or *ld.tex* we classify that load as non-deterministic. While any load in the program can be classified using this approach we only classify global load instructions into these two categories. The reason for this selection is that global load instructions cause the most significant performance hurdles in the memory system.

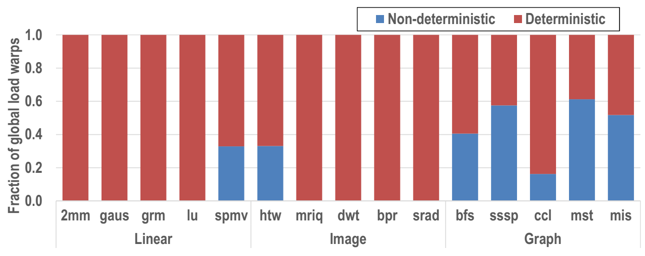


Fig. 1. Deterministic and non-deterministic load distribution

Figure 1 shows the distribution of global loads into deterministic and non-deterministic load instructions. Non-deterministic loads are frequently found in graph applications which traverse vertices indexed by user data. Most linear algebra and image processing applications require deterministic loads since those applications fetch data arrays in a regular fashion. However, spmv uses some non-deterministic loads as it fetches irregularly indexed arrays from a sparse matrix representation. Even in graph applications more than 50% of the global load instructions on average are deterministic loads, which typically produce coalesced memory accesses. Hence, the large number of uncoalesced memory accesses which were observed to cause significant performance hurdles in prior studies [23] actually originate from a smaller fraction of non-deterministic loads.

VI. Impact of Non-Deterministic Loads on Memory Traffic

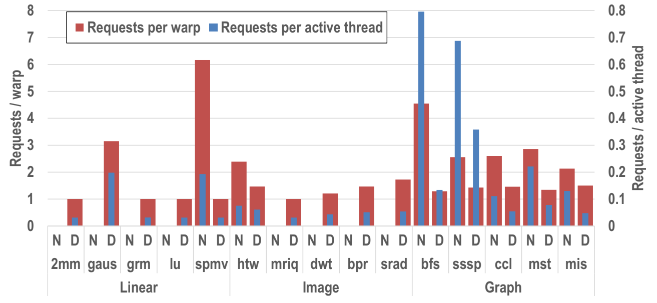


Fig. 2. Average number of memory requests per active thread and warp for deterministic and non-deterministic loads

We instrumented GPGPU-Sim to track non-deterministic and deterministic loads separately. We counted the number of memory requests generated by each warp as well as each active thread within a warp for the two load categories. As discussed earlier non-deterministic loads rely on non-parameterized data to compute effective address and hence it is highly probable that the addresses of such loads are not coalesced. Thus non-deterministic loads tend to generate multiple memory requests. GPUs coalesce data accesses from multiple threads in a warp if they all access consecutive memory locations. The coalescer sits before the L1 cache and hence each coalesced request generates one memory access request to the L1 cache.

The average number of memory requests generated to L1 per warp and per active thread, for both the non-deterministic and deterministic loads, is shown in Figure 2. An active thread denotes a thread whose active mask is set as valid in a warp. The non-deterministic load data is shown under the label N and deterministic load data is indicated under the label D in the figure. This notation is used from now on whenever we present data for the two load categories. It is clear from this data that for benchmarks that have non-deterministic loads the number of memory requests generated per each load instruction is significantly higher for non-deterministic loads than deterministic loads. The figure also plots the number of memory requests generated per load instruction per active thread. Ideally when all the 32 threads within a warp are active (when there is no branch divergence) and when a load request can be coalesced perfectly then each active thread may generate only $1/32$ memory requests per thread per each load instruction. However, non-deterministic loads generate vastly higher number of memory requests per each thread. For instance, bfs generates on average 0.8 memory requests per active thread per each non-deterministic loads. Thus about 26 memory access requests can be generated from single non-deterministic warp load instruction, if all threads of a warp are activate. The large number of memory requests generated per each non-deterministic load is agnostic to the benchmark category. Spmv, even though is a linear algebra program, generates six memory requests per warp for each non-deterministic load instruction.

The number of memory requests generated per load is a critical parameter that determines overall memory system behavior. If a larger number of warps issue multiple non-deterministic loads within a short time window they cause significant resource contention in the memory hierarchy. GPU memory system is well designed to handle coalesced memory accesses where a single memory request reads a wide cache line worth of data and feeds all the threads within a warp. However, when non-deterministic loads are encountered multiple narrow-width read requests stress cache access resources. When a memory request accesses the cache line it encounters one of three possible outcomes: *hit*, *miss* and *hit reserved*. A *hit reserved* outcome implies the case when the current request address is found in cache tags but the data for that cache line is still in-flight from a previous cache miss request . Clearly hit or at least a hit-reserved outcome is preferable for each memory request.

On a cache miss outcome the memory request tries to evict the cache line and fetch its own data. But such an eviction process may suffer an extremely long delay in GPUs. There are three reasons for why a cache line cannot be evicted. First is the case of *reservation fail by tags* which means that the current memory request cannot evict a cache line since the eviction candidate cache line itself is currently fetching in-flight data for a previous memory request. Until the in-flight data comes into the cache and is provided to the prior memory request the new memory request cannot be issued due to the lack of available cache tags. Second, even if there are available cache tags, the current memory request cannot be handled if the miss request has no available miss status handling registers (MSHR) to track the request. We call this event as *reservation fail by MSHRs*. Finally, even if the current miss has a tag and MSHR available there is a limited interconnection bandwidth between L1 and L2. If the new request cannot be injected into the input buffers of the interconnection network then the miss cannot be serviced either. We call this event *reservation fail by interconnection*. If the memory request cannot be handled due to any of the above three cases of reservation failures, cache access is retried at a future time. Hence, cycles are wasted until the corresponding hardware resources become available in the cache.

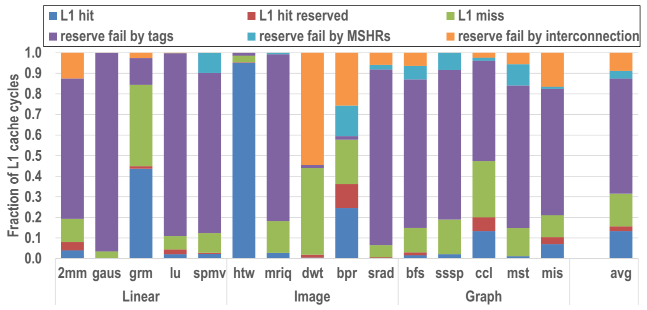


Fig. 3. Breakdown of L1 data cache cycles

Since each memory request, irrespective of the width of the request, needs to reserve all three resources – cache tag, MSHR, and an interconnection input buffer – each non-deterministic load that generates many more memory requests per warp has a debilitating impact on overall memory performance. Figure 3 shows the breakdown of how L1 cache access cycles are distributed across the various cache access outcomes described earlier. On average about 70% of cache cycles are wasted due to reservation failures, and a majority of reservation fails are due to the lack of available cache tags. This data sheds light into why graph applications suffer significantly higher memory bottlenecks, even though the fraction of the global loads over the total number of executed instructions is smaller than linear algebra and image processing applications. The high reservation failures in graph applications are generated by the non-deterministic loads resulting in significant memory access bottlenecks.

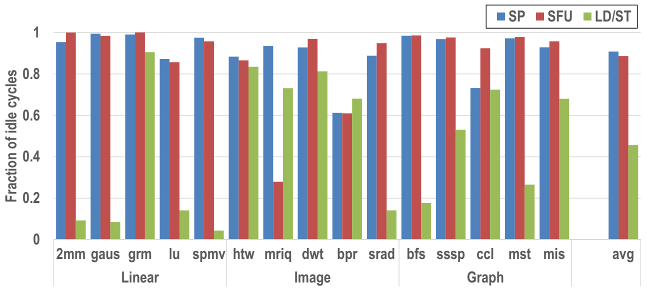


Fig. 4. Fraction of idle times

Figure 4 shows the fraction of time each of the three main function units within a SM are idle. To plot this data every cycle we check if the first pipeline stage in each of the three main functional units, SP, SFU and LD/ST is idle. If the first pipeline stage of an execution unit is still occupied by the previous instruction, the corresponding execution unit is unavailable for current ready warps. Therefore, the occupation of the first pipeline stage signifies that the execution unit is in busy state. Figure 4 shows the LD/ST unit is occupied more frequently than other execution units for the most of applications. SP and SFU units are busy only 9.3% and 11.5% respectively, while LD/ST unit is occupied on average for 54.4% of total execution time even though the portion of global load instruction is 6.43% of total instructions as described in Section IV. Furthermore, the LD/ST unit is busier for applications that have a large fraction of global loads and have a high cache miss rate. For instance, 2mm has 18% global load instructions and its L1 cache hit rate is low (shown later in Figure 3). On the other hand grm has nearly 25% global load instructions but its L1 cache hit rate is very high. Hence, the LD/ST units while executing grm are not as busy as in 2mm. Graph applications have a much smaller fraction of global load instructions compared to other applications but depending on how many non-deterministic loads they have, the LD/ST unit occupancy may be disproportionately higher.

VII. Impact of Non-Determinism on Load Instruction Latency

As memory traffic increases, latency of memory operation is elongated due to resource contention, such as queueing delays, congestion in memory partitions and interconnection network [12]. The bursty issuance of multiple memory requests from a single non-deterministic load can significantly degrade the service time of the memory sub-system. Furthermore, the longer latency of a non-deterministic load itself degrades overall performance since data dependency stalls warps until the non-deterministic load completes its execution.

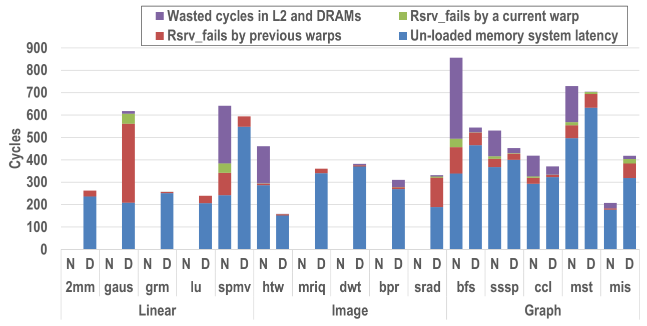


Fig. 5. Average turnaround time of non-deterministic and deterministic loads

Figure 5 shows the turnaround time of a global load warp on average, which is the time from when the warp is issued to LD/ST units to the time when the load data is written back to the destination register. The turnaround time is the minimum delay in executing an instruction that is dependent on a global load. We present the data for deterministic and non-deterministic loads separately. The bottommost component is the memory system latency when the memory system is not loaded and a global load does not encounter any reservation failures or interconnections network queueing delays. The second bar (Rsrv_fails by previous warps) represents the waste cycles while a current warp is waiting until resources of L1 data cache is available. Recall that a miss request needs an available cache tag, MSHR and interconnection input buffers before it can be issued. The third component from the bottom shows the required cycles until the last request from the current warp is reserved in the L1 cache. All requests cannot be reserved instantaneously in the L1 cache if multiple requests are generated from a warp while most of cache resources are already occupied. Then trailing requests must wait even longer until cache resources are available even if former requests could access the cache. Since non-deterministic loads generate multiple memory requests, overheads due to these two types of reservation fails are higher. The topmost component represents wasted cycles in memory partitions including interconnection network, L2 caches and DRAMs. Due to imbalanced traffic in memory channels and difference in data paths (L2 caches and

DRAMs), flight time of memory requests diverge, however, the overall time is determined by the lastly arrived data packets since completion of thread execution is synchronized within a warp.

Not surprisingly the biggest difference between deterministic and non-deterministic loads is that non-deterministic loads are delayed for longer periods of time waiting for resource reservations as well as wasted cycles by imbalanced service time in memory partitions. As discussed earlier, as non-deterministic loads generate multiple memory request they are more likely to encounter resource reservation stalls. Also, as the overall turnaround time is determined by the lastly serviced requests, non-deterministic loads have higher probability of getting delayed when they touch critical paths in the memory partition suffering from heavy data traffic. Due to these reasons, non-deterministic loads have longer turnaround time than deterministic loads and result in significant performance bottleneck. Furthermore, bursty memory requests from a non-deterministic load can also adversely impact deterministic loads by increasing memory congestion.

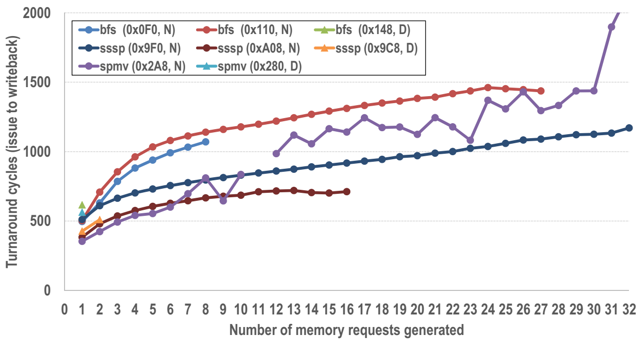


Fig. 6. Load instruction turnaround time w.r.t number of generated requests

To take a closer look at how non-determinism impacts total turnaround time we identify a few deterministic and non-deterministic loads from the bfs, sssp and spmv benchmarks. Figure 6 plots the turnaround time as a function of the number memory requests generated by each of the identified load in the three benchmarks. Each line graph is labeled by the benchmark name followed by (in parenthesis) the PC of the specific load instruction and whether that load is categorized as non-deterministic (N) or deterministic (D) load. Each deterministic load creates one or two memory requests, irrespective of which benchmark that load is part of. Hence, even in graph applications which have been demonstrated to exhibit poor memory system behavior deterministic loads do not generate large memory traffic. On the other hand, the number of memory requests generated by the non-deterministic loads varies from one instance of that load instruction execution to another instance. The same non-deterministic load instruction generates one to 32 memory requests per each warp during different instances of its execution. The randomness of the number of memory requests per each load is both a function of how many active threads are in the warp as well as how much coalescing is possible at some specific instances.

Figure 6 shows that the average turnaround time of a warp increases with the number of generated memory requests. An interesting point is that the average turnaround time of a warp of the deterministic load is similar to the average turnaround time of a non-deterministic load that creates one memory request. Hence, when non-deterministic loads are able to coalesce there is no significant difference between the two load categories. Consequently, longer turnaround time of the non-deterministic loads is primarily a function of the larger number of uncoalesced memory requests. Difference of arrival time between the first and the last data causes the increased turnaround time for a warp even if the average memory latency per each request is similar.

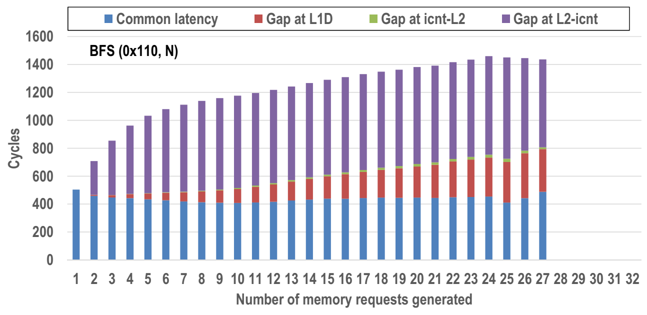


Fig. 7. Example of turnaround time breakdown for the non-deterministic load instruction (PC: 0x110 in bfs)

To provide further insights in to how the number of memory requests generated impacts the total turnaround time we breakdown the turnaround time of one specific non-deterministic load that missed in L1, (PC: 0x110) from bfs as shown in Figure 7. The bottommost component is labeled as common latency, which represents the latency that each individual request pays if it encounters miss in L1 cache. The common latency is equivalent to a warp's turnaround time subtracted by all wasted cycles from reservation fails and unbalanced paths in memory partitions. The additional latency encountered when the number of requests increase is broken down into three categories. The second component from the bottom is labeled as Gap at L1D, which is the added latency paid waiting for all the resource reservations to complete before initiating the cache miss request. This component of the latency increases as the number of requests increase. This data confirms our prior observations that as non-deterministic loads generate more memory requests per each load instruction the reservation stall time increases. The next component is labeled as Gap at icnt-L2, which is the wasted cycles in accessing the interconnection network between L1 and L2 cache. This component does not change very much as the memory request count increases. The top most component is labeled as Gap at L2-icnt, which is the cycle gap between the firstly and the lastly serviced data in accessing the network between L2 and interconnection network. The large average cycle gap at L2-icnt increases with the number of memory requests as increasing traffic leads to congestion and imbalanced used of memory partitions and usage differences across different data paths between L2 cache partitions and their associated DRAMs.

VIII. Cache Miss Rate

Recall that all the data presented in the previous sections focused on global loads that missed in the L1 cache. In this section we show how often a global load hits in the L1 cache, in which case it would not have to suffer the long latencies. Even though lots of memory requests are generated per warp

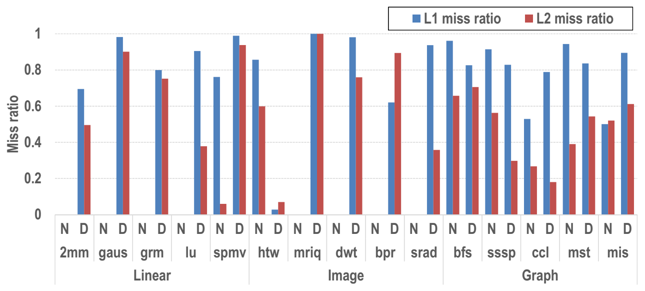


Fig. 8. L1 and L2 cache miss ratio for the non-deterministic and deterministic loads

for a non-deterministic load, the burden on the memory subsystem is mitigated if requested data hits in the cache. Figure 8 shows L1 and L2 cache miss ratios of all the applications for both categories of load instructions. The miss rates of both deterministic and non-deterministic loads exceed 50% in most cases, and in particular there is no significant improvement in the cache hit rate for deterministic loads. Furthermore, both non-graph and graph applications suffer from high cache miss rates. We surmise that prior studies that showed poor memory system behavior of graph applications [7], [6], [23], [21], [16] are essentially observing the poor behavior of non-deterministic loads that tend to occur more often in graph applications. Nearly in all cases it appears that L1 cache is highly ineffective in filtering accesses to the L2 cache.

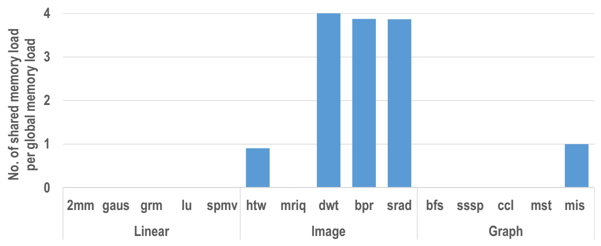


Fig. 9. Ratio of shared memory load and global memory load

One way to reduce the need for global memory is to use shared memory. GPU's provide programmers the ability to manage shared memory by making explicit memory allocation requests to the shared memory. With such explicit control data can be reused efficiently in shared memory with programmers' effort. Figure 9 shows the number of shared memory accesses per each global memory access. This data is collected by the CUDA Profiler running on the GPU hardware. Image applications use share memory 2.5 times frequently than global memory. However, other application categories do not use shared memory effectively, and in most applications shared memory is entirely unused. Shared memory can be efficiently exploited in image processing because the same set of image data is processed multiple times through several steps of processing. This data also explains the lower overheads in LD/ST unit for the image processing application in spite of very high miss ratio in the L1 data cache.

IX. Data Locality Among CTAs

The large L1 cache miss rate shown in the previous section may not be that surprising given the fact that thousands of threads must share a small L1 cache within an SM. For instance, only 10.67 Bytes (16 KB / (48×32)) of L1 data cache space can be assigned to each thread on the assumption that the maximum number of threads are allocated in an SM under the hardware configuration discussed in Section IV. Furthermore, since GPUs provide allow explicit management of special purpose data using constant and texture caches and programmers may explicitly move frequently used data into shared memory (which is not cached in L1), one may expect that L1 cache data is likely to exhibit poor locality.

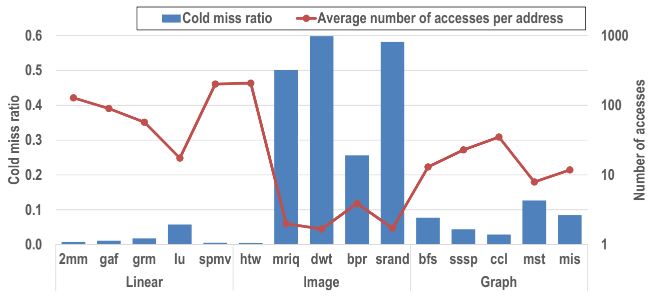


Fig. 10. Cold miss ratio and average number of accesses for data space

To verify this intuition and to reveal data reuse patterns in L1 cache, if any, we computed the cold miss ratio across the different GPU applications. To compute the cold miss ratios we divided the memory region into 128 byte cache line sized chunks and counted the first time any of the 15 L1 caches in the GPU brought that data into the cache while executing a given application. We divided the total count of first access to each block of data divided by total L1 cache accesses across all 15 L1 caches to compute the cold miss ratio. Note that our GPU configuration has 15 SMs and each SM has a private L1 cache, and all the SMs are assigned a subset of CTAs for execution.

Figure 10 shows the cold miss ratio for the GPU applications. Surprisingly, the cold miss ratio is only 16% on average. Only image processing applications exhibit high cold miss rates, on average 38.8%. The other two categories of applications exhibit very low cold miss ratios. The line graph in Figure 10 shows the average number of accesses to each 128 bytes data blocks. On average, data blocks are accessed repeatedly over 100 times for 2mm, gaf, spmv and htw. For graph applications, the average number of repeated accesses to a memory block is 18.1. Further analysis of the application source code revealed that global data accesses of the image processing application have low locality because repeatedly used data is captured into shared memory through explicit programmer help. However, in the other two categories of applications shared memory is rarely used even though there

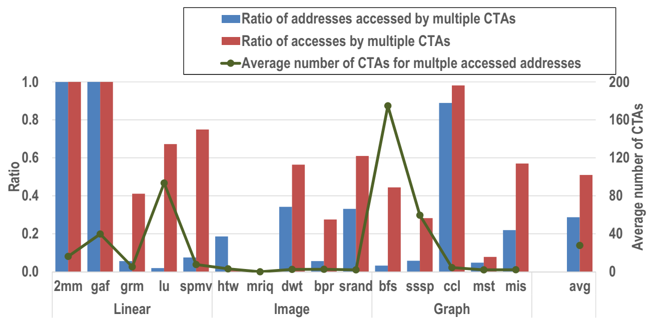
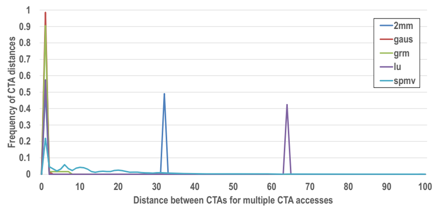
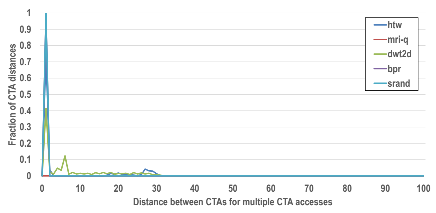


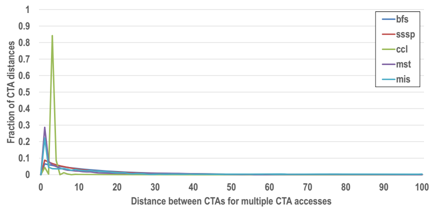
Fig. 11. Data space accessed by multiple CTAs



(a) Linear



(b) Image



(c) Graph

Fig. 12. Frequency of CTA distances for data space accessed by multiple CTAs

appears to be a large amount of temporal reuse of data. Thus the general intuition that high cache miss rates observed in GPU are due to random data accesses may not be accurate. So the question is where is this data locality gone? To answer this question we look at inter-CTA data locality.

A CTA is a group of threads and a basic workload unit assigned to an SM. If one warp within a CTA fetches data into L1 cache and other warps within the CTA reuse that data then those accesses may hit the private L1 cache. However, data fetched to the L1 data cache by one CTA may not be reused by another CTA if the two CTAs are assigned to different SMs since the L1 caches are private. Figure 11 shows ratio of data blocks shared by multiple CTAs. The blue bars show the ratio of 128 byte data blocks accessed by two or more CTAs compared to all the 128 byte blocks accessed by all CTAs. The red bar represents the ratio of access count for the blocks shared by multiple CTAs to total access count. In 2mm and gaf every block of data is is accessed by multiple CTAs. On average, 28.7% of data blocks are requested by multiple CTAs. Even though the amount of data space shared by multiple CTAs is not large, the fraction of access count for such shared blocks is high. On average, 50.9% among total data accesses are memory requests for data blocks shared by multiple CTAs. It means that data blocks used by multiple CTAs exhibit good inter-CTA locality. The line graph in the figure shows the average number of CTAs accessing data blocks if the data space is requested by multiple CTAs. Except for image processing applications shared data blocks are in fact shared across dozens of CTAs. For image processing applications since the programmer is able to move reused data to shared memory, there is no significant inter-CTA reuse of global memory.

Given the high data reuse in linear algebra and graph applications the high cache miss rate of these applications seems counter-intuitive. However, inter-CTA sharing of data does not necessarily decrease cache miss rates if the CTAs are assigned to different SMs. In order to validate this assumption Figure 12 shows the frequency of CTA distances for data blocks accessed by multiple CTAs. The X-axis is the distance between two linearized CTA ids and Y-axis shows that fraction of total shared data accesses that were shared across two CTAs whose linear ids differ by the amount specified on the X-axis. We define a linearized CTA id as *CtaId.x* + (*CtaId.y* × *CtaDim.x*) + (*CtaId.z* × *CtaDim.y* × *CtaDim.x*). Figure 12 shows sharing of data across CTAs occur within a close range of CTA ids. For instance, two neighboring CTAs (CTA distance = 1) have the highest probability of sharing data. Since linear algebra applications perform matrix arithmetic, high frequency of sharing is observed for the specific CTA distances based on the matrix dimensions. For 2mm, shared data space is accessed by multiple CTAs with a distance of either 1 or 32, and in lu sharing occurs across CTA distances of 1 and 64. Even for the image processing applications where data sharing is not as prominent whenever data is shared it is used across adjacent CTAs with a CTA distance of one. But in the graph applications even though there is a large amount of data sharing it is more dispersed across CTAs. We measured the data sharing patterns for non-deterministic and deterministic loads in graph applications and observed non-deterministic loads are the main reason why data sharing is distributed across a wide range of CTA distances.

X. Discussion

In this section, we discuss some potential hardware optimization for better memory performance.

A. *Instruction-specific optimization*

Based on the quantitative data shown, not all memory operations executed by an application have the same characteristics. Non-deterministic load instructions typically spawn more memory traffic than deterministic loads. As a consequence, those applications having more non-deterministic load instructions tend to encounter more reservation fails due to memory resource shortage, and longer turnaround time to handle multiple memory requests.

We suggest to design instruction-feature-aware mechanisms that can be selectively applied to load instructions according to their characteristics. A recent study which proposes memory prefetching algorithm for indirectly referenced addresses in graph applications [16] is a good example of such a specialization. We believe their algorithm targets prefetching only the non-deterministic loads unlike other prefetching methods that do not take into account load instruction type.

To avoid bursty memory traffic generation by non-deterministic loads, we suggest exploring techniques that partition non-deterministic loads into multiple sub-loads using warp splitting algorithms [10], [3]. Each sub-warp then generates only a subset of memory requests. Through careful scheduling of sub-warps it may be possible to avoid bursty resource usage and traffic congestion.

B. *CTA scheduling*

According to our evaluation, adjacent CTAs tend to share data block accesses in all three application domains. However, to maximize the parallelism, current GPUs assign CTAs to SMs in round-robin fashion. For example, suppose ten CTAs are to be assigned to five SMs and each SM can accommodate two CTAs then the first five CTAs are mapped to the five SMs, one CTA per each SM, and then the remaining five CTAs are again assigned in round-robin fashion. Therefore, CTA0 and CTA4 are assigned to SM0 (CTA1 and CTA5 to SM1 and so on). However, it would be better to assign neighboring two CTAs to the same SM (i.e. CTA0 and CTA1 to SM0, and CTA2 and CTA3 to SM1, and so on) for better data locality in L1 cache.

C. Cache hierarchy

Cache hierarchy can be redesigned for better locality. As adjacent two to five CTAs share data blocks, a shared L2 cache that is spans only a few SMs, rather than sharing across all SMs, can reduce interconnection costs and improve access latency. The semi-global L2 caches would have fewer conflict and capacity misses because of higher data locality. Alternatively, it would be also possible to have three level cache hierarchy by adding semi-global L2 cache between private L1 and globally shared L3 caches.

XI. Conclusion

This study provides a detailed characterization of the memory system behavior of GPU applications. Rather than looking at the memory system behavior aggregated over all loads, this study classifies loads into deterministic and non-deterministic categories and presents how these two load categories exhibit vastly different memory system behavior. Deterministic loads whose address is calculated by the parameterized values are likely to have coalescing memory access patterns. On the other hand, non-deterministic loads whose address is determined by user input data and other non-parameterized values tend to have uncoalesced memory access patterns. By measuring memory behaviors of these two types of load instructions separately, we found that non-deterministic loads are the primary performance bottleneck due to large number of memory requests generated by their uncoalesced accesses. Contrary to prior intuitions we show that even in graph applications data blocks are in fact shared across multiple CTAs but that inter-CTA sharing does not translate into L1 cache hit rate improvements. Based on these observations, we discuss and suggest several approaches to optimize GPGPU hardware for better memory performance.

Acknowledgment

This work was supported by DARPA-PERFECT-HR0011-12-2-0020, NSF-CAREER-0954211 and NSF-0834798.

References

[1] "Nvidia cuda profiler user's guide." [Online]. Available: http://docs.nvidia.com/cuda/profiler-users-guide/

[2] "Parboil benchmark suite." [Online]. Available: http://impact.crhc.illinois.edu/parboil.php

[3] M. Abdel-Majeed, W. Dweik, H. Jeon, and M. Annavaram, "Warped-re: Low-cost error detection and correction in gpus," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Pearson Education, 2006.

[5] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174.

[6] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *Proceedings of IEEE International Symposium on Workload Characterization*, 2012, pp. 141–151.

[7] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *Proceedings of IEEE International Symposium on Workload Characterization*, 2013, pp. 185–195.

[8] 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 IEEE International Symposium on Workload Characterization*, 2009, pp. 44–54.

[9] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Proceedings of IEEE International Symposium on Workload Characterization*, 2010, pp. 1–11.

[10] W. Dweik, M. Abdel-Majeed, and M. Annavaram, "Warped-shield: Tolerating hard faults in gpgpus," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 431–442.

[11] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proceedings of International Symposium on Computer Architecture*, 2011, pp. 235–246.

[12] P. Gratz, B. Grot, and S. W. Keckler, "Regional congestion awareness for load balance in networks-on-chip," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, 2008, pp. 203–214.

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

[14] J. Hestness, S. W. Keckler, and D. A. Wood, "A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior," in *Proceedings of IEEE International Symposium on Workload Characterization*, 2014, pp. 150–160.

[15] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for gpgpus," in *Proceedings of International Symposium on Computer Architecture*, 2013, pp. 332–343.

[16] N. B. Lakshminarayana and H. Kim, "Spare register aware prefetching for graph algorithms on gpus," in *Proceedings of 2014 IEEE 20th International Symposium on High Performance Computer Architecture*, 2014, pp. 614–625.

[17] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for gpgpu applications," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 213–224.

[18] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, 2008.

[19] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal, "A detailed gpu cache model based on reuse distance theory," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, 2014, pp. 37–48.

[20] NVIDIA, "Cuda toolkit 4.0." Available: https://developer.nvidia.com/cuda-toolkit-40

[21] M. A. O'Neil and M. Burtscher, "Microarchitectural performance characterization of irregular gpu kernels," in *Proceedings of IEEE International Symposium on Workload Characterization*, 2014, pp. 130–139.

[22] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 72–83.

[23] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on gpus: Where are the bottlenecks?" in *Proceedings of IEEE International Symposium on Workload Characterization*, 2014, pp. 140–149.

[24] G. L. Yuan, A. Bakhoda, and A. T. M., "Complexity effective memory access scheduling for many-core accelerator architecture," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 34–44.