A Case Study in Reverse Engineering GPGPUs: Outstanding Memory Handling Resources

Ahmad Lashgar
ECE Department
University of Victoria
lashgar@uvic.ca

Ebad Salehi
ECE Department
University of Victoria
ebads67@uvic.ca

Amirali Baniasadi
ECE Department
University of Victoria
amiralib@ece.uvic.ca

ABSTRACT

During recent years, GPU micro-architectures have changed dramatically, evolving into powerful many-core deep-multi-threaded platforms for parallel workloads. While important micro-architectural modifications continue to appear in every new generation of these processors, unfortunately, little is known about the details of these innovative designs. One of the key questions in understanding GPUs is how they deal with outstanding memory misses. Our goal in this study is to find answers to this question. To this end, we develop a set of micro-benchmarks in CUDA to understand the outstanding memory requests handling resources. Particularly, we study two NVIDIA GPGPUs (Fermi and Kepler) and estimate their capability in handling outstanding memory requests. We show that Kepler can issue nearly 32X higher number of outstanding memory requests, compared to Fermi. We explain this enhancement by Kepler's architectural modifications in outstanding memory request handling resources.

1. INTRODUCTION

GPGPUs have evolved into energy-efficient platforms for executing massively-multithreaded OpenCL / CUDA workloads mainly as the result of innovative designs employed in their micro-architecture. Information for understanding new micro-architectures are limited to few white papers from manufacturers, which mostly explain a high-level view of the design. Previous studies [10, 11, 1], however, have introduced various micro-benchmarking solutions, successfully revealing few micro-architectural details.

In this study we extend previous studies and use micro-benchmarks to reverse engineer GPGPUs and analyze their capabilities in managing outstanding memory requests. First, we give an overview of two alternative architectural designs for handling outstanding memory requests in GPGPUs (Section 2). One design is based on the Miss Status/Information Holding Register (MSHR) structure, initially proposed for CPUs [3]. The other design, named Pending Request Table (PRT) is used specifically by many-thread GPUs [7]. Second, we investigate two different GPUs and report their approach to handling outstanding memory requests (Section 3). Finally, after identifying the employed approach, we extend our micro-benchmarking to understand design details (Section 5).

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2015) Boston, MA, USA, June 1-2, 2015.

Our micro-benchmarks rely on issuing a certain number of memory requests and measure the number of core clock ticks taking to complete the requests. The measured latency increases as we increase the number of memory requests. Observing the variations in this latency, we assume that a significant increase indicates a micro-architectural limitation in booking outstanding memory requests.

In summary, we make the following contributions:

- We develop micro-benchmarks to stress outstanding memory requests handling resources. The developed micro-benchmarks can be configured to generate different memory patterns, stressing various aspects of the resources.
- We run our micro-benchmarks on two GPGPUs which have different micro-architectures. We show that under Fermi architecture the maximum number of outstanding memory accesses is limited by the number of uncoalesced accesses. We also show that under Kepler architecture the maximum number of outstanding memory accesses is limited by the number of memory instructions that a warp can execute.
- We show that each SM in Fermi GPGPU can issue up to 128 concurrent memory requests to DRAM. We also show that this number is much greater in Kepler GPGPU, nearly 1408 requests. We present the micro-architectural design that can explain this huge change.

2. BACKGROUND

In this study we focus on NVIDIA Tesla M2070 and Tesla K20 GPGPUs. Tesla M2070 and Tesla K20 are based on Fermi [9] and Kepler architectures [6], respectively. Kepler, the recent architecture, differs from Fermi in various aspects; namely the number of Streaming Multiprocessors (SM), warps per SM, and SIMD engines per SM. In both architectures, there are 32 threads in a warp. Threads of the warp execute one common instruction at a time, referred to as *warp instruction*. Threads of the warp are executed in lock-step SIMD-like style and are synchronized at every instruction. Both architectures have two levels of cache hierarchy for data. We assume that the cache hierarchy is not blocking and can handle multiple outstanding memory requests concurrently. This is achieved mainly by using *outstanding memory request handling resources*. Since L2 cache is backing multiple L1 caches from different SMs, we assume that the L2 cache is richer than L1 in terms of outstanding memory request handling resources. Hence, during micro-benchmarking a single SM, the outstanding memory request handling resources at the L1 cache side will be the bottleneck as these resources saturate before the saturation of outstanding memory request handling resources at the L2 cache side.

We assume two possible designs for the organization of outstanding memory request handling resources at the SM or L1 cache side. Below we overview these two designs.

2.1 Miss Status Handling Registers

The Miss Status/Information Holding Registers (MSHR) structure is assumed in GPGPU-sim v3.2.2 [2] for modeling outstanding memory request handling resources. In this design, each SM has a certain number of MSHRs. Each MSHR tracks one pending memory request, fetching a memory block equal to the cache line size.

Upon executing a memory instruction for a warp, the Load/Store unit coalesces memory accesses of threads into memory blocks. For each coalesced access, one MSHR entry is booked and one memory request is issued. Each MSHR entry can also be shared among a certain number of threads, avoiding redundant MSHR allocations. The number of threads/warps that can share a common MSHR varies from GPU to GPU, impacting the capability of GPU in dealing with outstanding memory requests. Also the address offset (within the requested memory block) is stored for each of the threads merged in the same MSHR entry.

Once a reply is received from the lower memory level (L2 cache in this case), the requested data and identifier of MSHR are extracted from the reply. For the thread identifiers marked in the corresponding MSHR entry, the Load-/Store unit writes the fetched data into the private registers of the threads. The Load/Store unit releases the MSHR entry after completing this process. An operation example is illustrated in [4].

In this design, the number of outstanding memory requests depends on the number of available MSHRs. Accordingly, the SM may stall a warp once there is no free MSHR available. Since each MSHR has limited capability in merging redundant memory requests, it is important that the programmer develops programs that perform well-coalesced accesses to avoid such stalls.

2.2 Pending Request Table

The Pending Request Table (PRT) structure is introduced in [7]. PRT has a certain number of entries and each entry tracks one memory instruction per warp (memory instructions include load and store instructions from DRAM.)

Upon executing a memory instruction for a warp, only one PRT entry is reserved (if there is a memory miss). Before booking information in this entry, the Load/Store unit coalesces memory accesses of the threads into memory blocks. For each coalesced access, the Load/Store unit creates one memory request to fetch the data from the lower memory level. In the memory request packet, the Load/Store unit stores the address of the requested block, the identifier of assigned PRT entry, and the mask of threads needing this block. Then, the Load/Store unit sends this memory request to the lower memory level. In the PRT entry, the Load/Store unit stores the number of pending memory requests. It also stores the memory address offset (within the requested memory block) for each thread.

Once a reply is received from the lower memory level, the requested data, the identifier of PRT entry, the address of requested memory block, and the mask of threads are extracted from the reply. Based on the offsets recorded in the corresponding PRT entry, the Load/Store unit serves the memory instruction and stores the fetched data into private registers of the threads marked in the mask. The PRT entry is released upon completion of all memory requests. An example is illustrated in [7].

The number of outstanding memory requests is independent of the number of uncoalesced memory accesses. Since the entire memory accesses of a warp (including coalesced and uncoalesced accesses) are stored in just one PRT entry, the number of outstanding memory instructions is limited by the number of load/store instructions executed by warps. Essentially, the number of PRT entries specifies the number of warp memory instructions that can be executed concurrently.

3. MICRO-BENCHMARKING MECHANISM

We developed a set of micro-benchmarks to measure the maximum number of outstanding memory requests that every SM can handle. We report the information gathered by our micro-benchmark in the form of *Thread-Latency plots*. Thread-Latency plots report the latency of completing certain number of memory requests. Calculating the variance of latencies in this plot, we detect occasional saturations in outstanding memory request handling resources. *Saturation in outstanding memory request handling resources appears as a spike in the variance curve and indicates the maximum number of outstanding memory requests that an SM can handle.* We evaluate various memory access patterns to estimate the capability of an SM in handling pending memory requests. Below we explain our micro-benchmark, Thread-Latency plot, and how we derive information from this plot.

3.1 Micro-benchmark Skeleton

For generating Thread-Latency plots, we use the source CUDA code skeleton shown in Listing 1. The code measures the latency associated with completing a certain number of global memory requests. Within the kernel, we use *clock()* procedure call to measure the latency. The code varies two parameters to control the number of memory requests: i) the number of threads and ii) the number of loads per thread. These two parameters are controlled by *nThreads* and *N* in the code, respectively. We sweep *nThreads* from 2 to 1024 (maximum thread block size), incrementing steps of 2 threads at a time. We report a separate Thread-Latency plot for each *N*, swept from 1 to 4. We limit the number of loads per thread below the scoreboard limit. We avoid the configurations that exceed the capability of scoreboard since the performance in these cases is limited by the scoreboard, not the outstanding memory request handling resources [5].

During evaluations, the code launches exactly one thread block (line 21), assuring all threads only stress memory handling resources of a single SM. Within the kernel in the code lines 7 to 14, we measure the latency of completing *N* memory requests. The addresses of these memory requests are calculated in the lines between 3 to 5. We change the address calculation to simulate different memory patterns. In this example, address calculation assures every memory address points to a unique 128-byte region (indicated by *UNQ* macro). We separate unique addresses by 128 bytes since the cache line size of micro-benchmarked GPUs are 128 bytes [8].

Listing 1: micro-benchmarking code.

```
1  void __global__
2     kernel(int *in, int *out, unsigned long int *time) {
3   unsigned long int addr1=(unsigned long int)in+UNQ(1,gtid);
4   ...
5   unsigned long int addrN=(unsigned long int)in+UNQ(N,gtid);
6
7   __syncthreads();
8   t1=clock();
9   int reg1, ..., regN;
10  asm("ld.global.s32 %0, [%1+0];" : "=r"(reg1) : "l"(addr1) );
11  ...
12  asm("ld.global.s32 %0, [%1+0];" : "=r"(regN) : "l"(addrN) );
13  __syncthreads();
14  t2=clock();
15
16  out[gtid]=reg1+...+regN;
17  time[gtid]=t2-t1;
18 }
19 int main()
20 {
21  kernel<<<1, nThreads>>>(in, out, time);
22 }
```

This addressing pattern forces the GPU to issue a unique memory request for each of the addresses. After measuring the time, the time is written back to global memory at line 17. The code writebacks the accumulation of loaded values at line 16, imposing instruction dependencies to prevent compiler optimizations from removing memory instructions. We will provide public access to our micro-benchmarks in the final version of the paper.

3.2 Methodology of Deducing Information from Thread-Latency Plot

We present the Thread-Latency plot for reporting the latency measured by Listing 1. We derive information from this plot and estimate the maximum number of pending memory requests. Each Thread-Latency plot reports the latency under certain fixed i) memory pattern and ii) number of loads per threads. The horizontal axis of this plot represents different number of threads (increasing the number of threads means increasing the number of memory instructions). Two vertical axises report the *latency* and *variance of latency*. We report *unbiased sample variance of latency* to make latency variations clearly visible. For every point in Thread-Latency plot, the variance curve reports the variance among the point and the two neighbors (right and left). For every point, the variance is calculated using the following equation:

$$Var(Lat_{thd}) = \frac{\sum_{i=thd-1}^{thd+1} (Lat_i - \mu)^2}{2}, \mu = \frac{\sum_{i=thd-1}^{thd+1} Lat_i}{3}$$

Along the positive direction of the horizontal axis, where memory requests increase, the *variance curve* highlights sudden changes in the *latency curve*, indicating the saturation of memory handling resources in the hardware.

Figure 1 shows the Thread-Latency plot for one load per thread, under the memory pattern where each thread sends a unique memory request. In each Thread-Latency plot, two factors can explain the spikes in the variance curve. First, rapid change of latency is caused by serialization in coalescing memory accesses of threads, issuing non-coalesced memory requests, and writing the data back after completion of memory requests. Second, latency variation is caused by the saturation of pending memory requests resources. The second latency variation is larger. Also we observe serialization latency rapid variations appear before the saturation of pending requests resources.

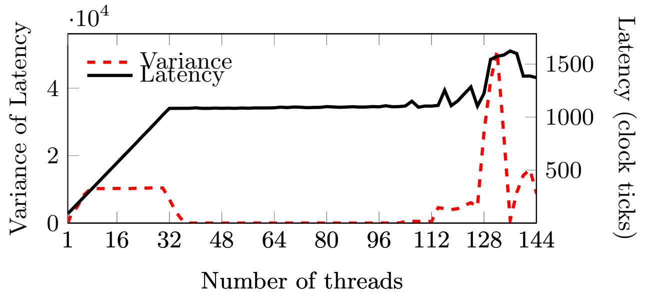

Figure 1: Thread-Latency plot under one load per thread and every thread requests one unique 128-byte block.

Memory issue and writeback serialization. In Thread-Latency plot, a consistent increase can be seen in the latency when increasing the number of threads from 1 to 32, . This can be explained by the serialization in i) coalescing and issuing memory requests and/or ii) writing the fetched data back to register file for each thread of the warp. We refer to these two serializations as *memory issue serialization* and *writeback serialization*, respectively. As the number of threads strides beyond one warp (higher than 33), the latency does not change significantly. We can infer that memory issue and writeback of different warps can be processed in parallel (this can be achieved by parallel load/store units in an SM). We conclude from Figure 1 that up to 96 memory requests can be processed in parallel, as an increase in the latency is observed just after 96 threads.

Pending memory requests. Upon saturation of pending memory request resources, the memory unit stops issuing further requests and waits for completion of prior memory requests. This can cause a significant spike in the variation curve of Thread-Latency plot. For instance in Figure 1 and under one load per thread plot, we can see a significant change in the latency after 128 threads (which equals to 128 memory requests in this plot).

As reported in the variance plot of Figure 1, moving from left to right, the first and smaller spike corresponds to memory issue and writeback serialization and the second and larger spike corresponds to the saturation in pending memory requests resources. The second spike can be seen clearly on 128 for one load per thread in Figure 1. We conclude from this figure that the maximum number of outstanding memory requests possible to accommodate is 128.

3.3 L2 Cache

In NVIDIA Fermi and Kepler GPUs all memory transfers from host to device pass through the L2 cache. This means that once memory transfers are completed, kernels may access the data at the latency of L2 cache (not DRAM). As we rely on studying variations in memory latency, understanding the L2 cache is critical.

To understand the L2 cache, we wrote a kernel which simply copies a chunk of data from one part of the global memory to another. Each thread is responsible for copying a single 8-byte data element. Accordingly, threads fetch the data into the GPU cores and then write it back to another address. We measure the time of completing the kernel under different data sizes. We expected to see L2 cache hits so long the overall data size stays below the L2 cache capacity. In Figure 2, the plots labelled as M2070 and K20 show the result of this micro-benchmarking under Tesla M2070 (Fermi) and K20 (Kepler), respectively. To signify the performance impact of using the L2 cache, we compare this latency to cases where L2 cache is flushed. We flush the L2 cache by copying a large dummy data (76 MB) from host to device before kernel launch. This essentially clears the L2 cache before the kernel launch that follows. In Figure 2, the plots labelled M2070-flushed and K20-flushed represent this. Comparing the flushed and non-flushed plots, we conclude that the performance gap saturates beyond a certain data size. This suggests the saturation of L2 cache after that point. Under M2070 and K20, this occurs approximately at 768 KB and 1536 KB, which are the L2 cache sizes for these GPUs.

We use our findings and make sure that the input data size is below L2 cache size. Consequently the spikes observed our a result of the saturation of outstanding memory accessing resources and not the L2 cache.

4. METHODOLOGY

We run our evaluations on two systems with different GPGPUs; NVIDIA Tesla M2070 and NVIDIA Tesla K20. We measure the time using *clock()* procedure call of CUDA. Every reported point is a harmonic mean of three different runs. We use CUDA Toolkit 5.5 for compiling the benchmarks.

Our micro-benchmarking shows that every warp can issue a limited number of independent instructions concurrently. This limitation is imposed by the scoreboard structure used in each GPU [5]. In the evaluated micro-benchmark, we found that Tesla M2070 can issue up to four concurrent instructions per warp. We also found that Tesla K20 can issue up to three concurrent instructions per warp.

5. RESULTS

In this section, we use the micro-benchmark presented in Section 3 to estimate the capability of an SM in managing pending memory requests. We reconfigure and run the code under various configurations; varying memory patterns, number of threads, and number of load instructions per thread. In this section, we follow a two-step approach to understand outstanding memory request handling resources of Tesla M2070 and Tesla K20. At the first step, we specify whether the micro-architecture follows an MSHR-like design (refer to Section 2.1) or a PRT-like design (refer to Section 2.2). The micro-architecture may have a design similar to MSHR table, if the spikes in Thread-Latency plot of GPU depend on the i) number of concurrent threads and ii) number of loads per thread. The micro-architecture may have a design similar to PRT, if the spikes in Thread-Latency plot of GPU depend on the number of warp instructions. In either case, at the second step, we tune the micro-benchmarks to measure design specifics. For MSHR, we look for the number of MSHRs and mergers fields per MSHR entry. For PRT, we look for the number of PRT entries per SM.

5.1 Tesla M2070

MSHR or PRT? Figure 3 shows a Thread-Latency plot where each thread sends a load request for one unique 128-byte memory block. As reported, a significant variation in latency can be observed upon increasing the number of threads from 128 to 130. From this figure, we infer that SM allows up to 128 unique memory requests. A similar trend can be seen in Figure 4 which shows the Thread-Latency for two loads per thread. We observe a significant change in the latency after 64 threads (which equals to 128 memory requests in this plot). As the spikes in Thread-Latency plot directly relate to the number of threads and the number of loads per thread, we can conclude that each SM of Tesla M2070 has a structure similar to MSHR table for handling outstanding memory requests.

Parameters of MSHR. Knowing each SM exploits an MSHR table, we look for the number of redundant memory requests that each MSHR entry can merge. Based on our earlier micro-benchmarking, we increase the number of redundant memory requests within the warp to measure the merging capability. We evaluate the following memory patterns:

- **Merger2:** every two neighbor threads of the warp send a load request for the same address (total of 16 unique requests per warp),
- **Merger4:** every four neighbor threads of the warp send a load request for the same address (total of eight unique requests per warp),
- **Merger8:** every eight neighbor threads of the warp send a load request for the same address (total of four unique requests per warp),
- **Merger16:** every 16 neighbor threads of the warp send a load request for the same address (total of two unique requests per warp),
- **Merger32:** every 32 threads of the warp send a load request for the same address (total of one unique request per warp),

Figure 5 shows Thread-Latency plot under *Merger2* memory pattern where each thread executes *one* load instruction. As shown in the figure, there is a huge spike upon moving beyond 256 threads, which equals to 128 unique memory requests. Hence, we can infer that SM allows up to 128 unique memory requests and can also merge up to 128 redundant requests. In other words, SM has enough resources to keep 128 pending memory requests and can merge at least one other redundant request with each of those pending requests.

Figure 6 reports Thread-Latency plot under *Merger4* memory pattern where each thread executes *two* load instructions. There is a significant variation in the latency when increasing the number of threads beyond 256. This point equals to a total of 512 requests (*256*2*) from 256 threads, which are later merged into 128 unique memory requests (*512/4=128* according to the memory pattern). We can infer that each SM can keep up to 128 pending memory requests and can merge four other requests with each of the pending requests.

Figure 7 reports Thread-Latency plot under *Merger8* memory patter where each thread executes *four* load instructions.

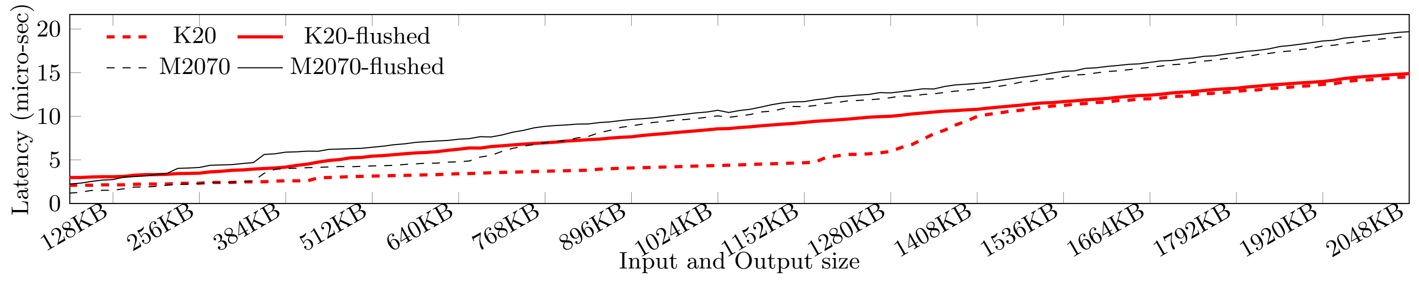

Figure 2: Micro-benchmarking L2 cache under Tesla M2070 and K20. Comparing flushed to non-flushed plots clearly shows the saturation of L2 cache after certain data size.

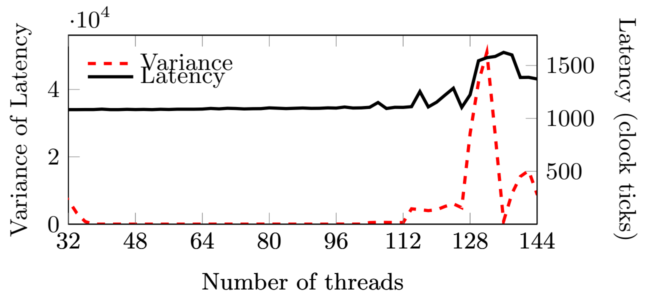

Figure 3: Thread-Latency plot under Tesla M2070, one load per thread, and every thread requests one unique 128-byte block.

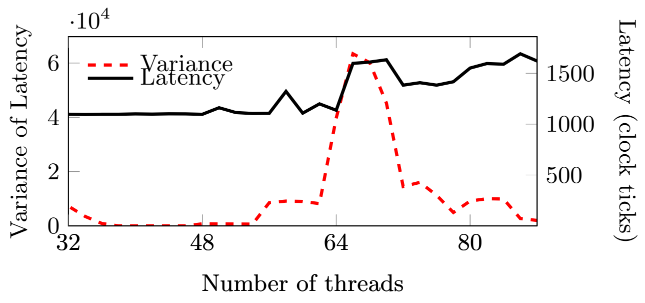

Figure 4: Thread-Latency plot under Tesla M2070, two loads per thread, and every thread requests two unique 128-byte blocks.

There is a significant spike in variation plot just after 256 threads. This point equals to 1024 requests (*256*4*) from 256 threads, which are later merged into 128 unique requests (*1024/8=256* according to the memory pattern). We can infer that each SM can keep up to 128 pending memory requests and can merge eight other requests with each of the pending requests.

For *Merger16* and *Merger32*, we could not observe a significant spike in Thread-Latency plots. Our guess is another resource may have been saturated, flattening MSHR saturation spikes.

Finding. According to the observations above, we infer that each SM of Tesla M2070 has a 128-entry MSHR table to handle outstanding memory requests. We can also conclude each MSHR entry can merge up to eight requests.

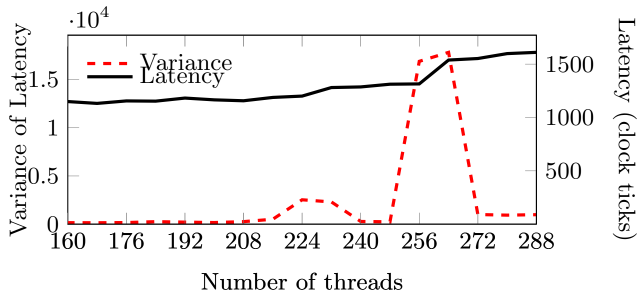

Figure 5: Thread-Latency plot under Tesla M2070, one load per thread, and every two neighbor threads request one unique 128-byte block.

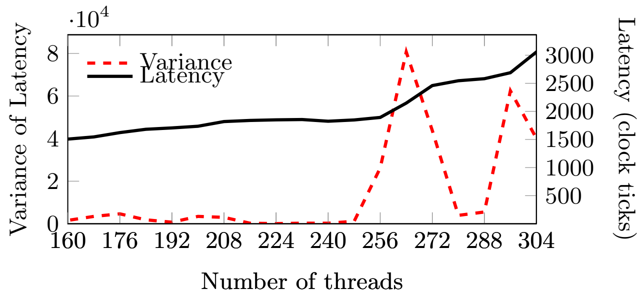

Figure 6: Thread-Latency plot under Tesla M2070, two loads per thread, and every four neighbor threads request two unique 128-byte blocks.

5.2 Tesla K20

MSHR or PRT? Figure 8, 9, and 10 show Thread-Latency plot under Tesla K20 GPGPU. All figures report numbers for a memory pattern where every load instruction has a unique address. Figures differ in the number of load instructions per thread. Figure 8, 9, and 10 report numbers under one, two, and three load instructions per thread. Under one load instruction per thread, we do not observe a significant spike in Thread-Latency. This can be explained by the fact that outstanding memory handling resources are not saturated. Under two load instructions per thread, we observe a significant spike just after 680 threads. At this point, we have launched 22 warps. Assuming two load instruction per warp, we can also say we issue 44 warp load instructions at this point. Under three load instructions per thread, we observe a significant spike just after 454 threads.

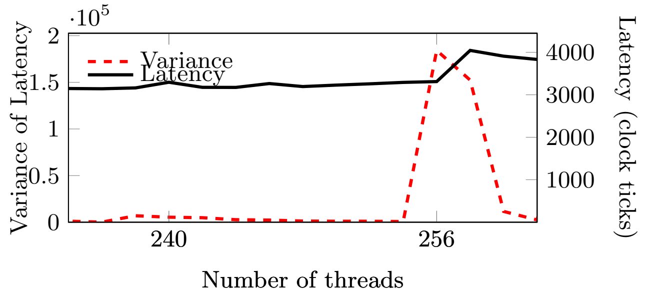

Figure 7: Thread-Latency plot under Tesla M2070, four loads per thread, and every eight neighbor threads request four unique 128-byte blocks.

At this point, we have launched 15 warps. Assuming three instruction per warp, we can also say we issue 45 warp load instructions at this point. Comparing our findings in Figures 9 and 10, we can conclude that each SM can concurrently issue at least 45 warp load instructions. Since the number of outstanding memory requests in this GPU depends on the number of warp instructions, not the number of threads and load instructions per thread, we can infer that this GPU uses a design similar to PRT.

Finding. We conclude that there is strong evidence that outstanding memory requests handling resources in Tesla K20 may have a design similar to PRT described in 2.2 and the table has at least 45 entries. Since each entry should store 32 requests to support the entire threads of the warp, we can also conclude that each SM can support up to 1408 (44 x 32) outstanding unique memory requests.

6. RELATED WORK

Wong et al. [10] introduced a set of micro-benchs, revealing wide set of micro-architectural details on NVIDIA GTX280 GPU, including the latency and throughput of different instructions and cache hierarchy of different memory spaces. They also investigated the behavior of intra- thread block synchronization and intra-warp branch divergence.

Zhang et al. [11] developed a set of micro-benchmarks to understand the impact of VLIW utilization on the power consumption of ATI GPUs. They found that increasing VLIW utilization from 20% to 100% can increase the power consumption by nearly 33%.

Anderson et al. [1] used micro-benchmarking to understand shared memory, global memory, and intra- thread block synchronization in NVIDIA Fermi architecture. They reported that the achievable bandwidths are 75% and 85.4% for shared and global memory respectively. They also found that the latency of shared memory and global memory are 27 and 570 cycles, respectively. They showed that address calculation overhead in Fermi architecture significantly increases the shared memory access time, while the address calculation overhead is negligible for global memory accesses. They also presented that the latency of intra- thread block synchronization depends heavily on the thread block size, if the thread block size is below 256 threads.

7. CONCLUSION

We used micro-benchmarking to understand GPGPU resources in handling outstanding memory requests. We investigated two generations of NVIDIA GPUs; Fermi and Kepler. First, we explained two alternative designs for the micro-architectural details of outstanding memory requests handling resources, MSHR and PRT. Then, we used micro-benchmarking to understand whether the GPU may use either of these designs. We showed that our evaluated Fermi GPU uses a design working similar to a 128-entry MSHR table and each entry can merge at least eight requests. On the evaluated Kepler GPU, we showed that the GPU could be using a 45-entry PRT-like design. Our findings can be useful to programmers in optimizing the memory access pattern of the applications in respect to hardware capability.

8. REFERENCES

[1] M. Anderson et al. A predictive model for solving small linear algebra problems in gpu registers. In *IPDPS 2012*.

[2] A. Bakhoda et al. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS 2009*.

[3] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA 1981*.

[4] A. Lashgar and A. Baniasadi. A case against small data types in gpgpus. In *ASAP 2014*.

[5] S. Moy and J. Lindholm. Across-thread out of order instruction dispatch in a multithreaded graphics processor, June 23 2005. US Patent App. 10/742,514.

[6] NVIDIA Corp. Nvidia's next generation cuda compute architecture: Kepler gk110. *Available: http://www.nvidia.ca/content/PDF/kepler/ NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.*

[7] L. Nyland et al. Systems and methods for coalescing memory accesses of parallel threads, Mar. 5 2013. US Patent 8,392,669.

[8] Y. Torres et al. Understanding the impact of cuda tuning techniques for fermi. In *HPCS 2011*.

[9] C. Wittenbrink et al. Fermi gf100 gpu architecture. *IEEE Micro*, 31(2):50–59, March 2011.

[10] H. Wong and others. Demystifying gpu microarchitecture through microbenchmarking. In *ISPASS 2010*.

[11] Y. Zhang et al. Performance and power analysis of ati gpu: A statistical approach. In *NAS 2011*.

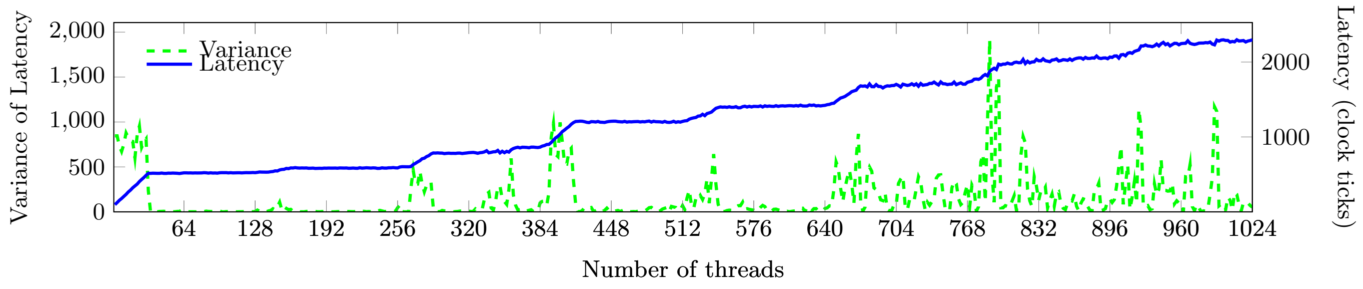

Figure 8: Thread-Latency plot under Tesla K20, one load per thread, and every thread requests one unique 128-byte block.

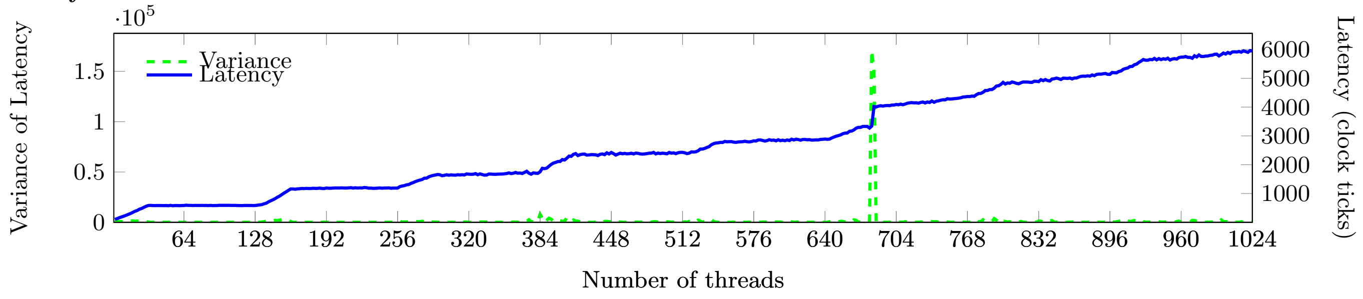

Figure 9: Thread-Latency plot under Tesla K20, two loads per thread, and every thread requests two unique 128-byte blocks.

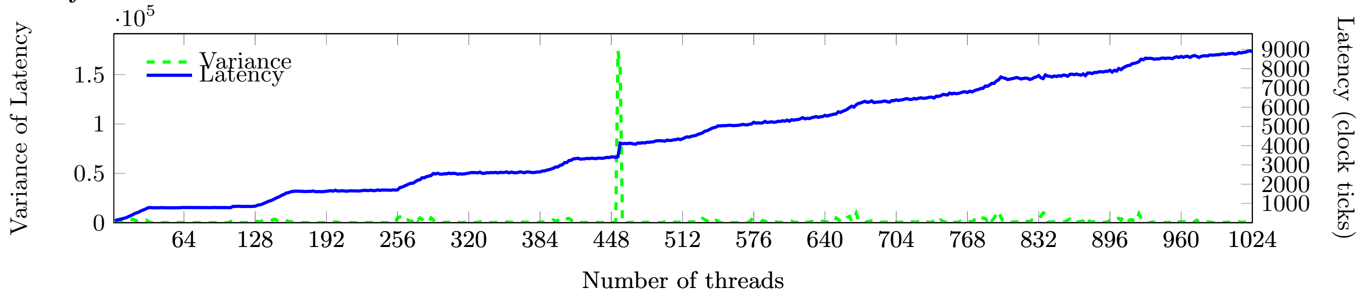

Figure 10: Thread-Latency plot under Tesla K20, three loads per thread, and every thread requests three unique 128-byte blocks.