Characterizing Power and Performance of GPU Memory Access

Conference Paper · November 2016

DOI: 10.1109/E2SC.2016.012

CITATIONS
9

READS
684

2 authors:

Tyler Allen
Clemson University
9 PUBLICATIONS **52** CITATIONS
SEE PROFILE

Rong none Ge
Clemson University
58 PUBLICATIONS **1,894** CITATIONS
SEE PROFILE

Characterizing Power and Performance Of GPU Memory Access

Tyler Allen and Rong Ge
School of Computing
Clemson University
Email: tnallen, rge@clemson.edu

***Abstract*—Power is a major limiting factor for the future of HPC and the realization of exascale computing under a power budget. GPUs have now become a mainstream parallel computation device in HPC, and optimizing power usage on GPUs is critical to achieving future goals. GPU memory is seldom studied, especially for power usage. Nevertheless, memory accesses draw significant power and are critical to understanding and optimizing GPU power usage. In this work we investigate the power and performance characteristics of various GPU memory accesses. We take an empirical approach and experimentally examine and evaluate how GPU power and performance vary with data access patterns and software parameters including GPU thread block size. In addition, we take into account the advanced power saving technology dynamic voltage and frequency scaling (DVFS) on GPU processing units and global memory. We analyze power and performance and provide some suggestions for the optimal parameters for applications that heavily use specific memory operations.**

***Index Terms*—High Performance Computing, GPU Memory Access, Heterogeneous Computing, Power and Performance Characterization**

I. Introduction

Power has become a first-order constraint for high performance computing and limits the design and employment of systems and components. The expected exascale computers around 2020 will operate under a 20 MegaWatts power budget [7], and computer components must perform within their thermal design point to behave normally. To accelerate computation with reduced power consumption, power efficient graphics processing units (GPU) are prevalent in HPC systems today, and accelerator based heterogeneous computing emerges as a major parallel computing paradigm [3]. Nonetheless, GPU-based systems are similarly facing power limits, and it is critical for the GPU accelerators to optimally use power for application performance and system throughput.

Understanding the power effects of GPU memory accesses is important to addressing the power challenge for GPU-based heterogeneous computing for two main reasons. First, GPU memory accesses play a tremendous role in real-world GPU-based applications [25]. For example, GPU memory is the only method of data communication between the host and the GPU. In addition, the highly parallelizable applications that can benefit from GPU for performance involve a large number of data accesses to the levels of GPU memory system [8], [21]. Poor memory accesses lead to poor application performance and inefficient use of power. Second, memory accesses could be the dominating power consumer among GPU operations. Unfortunately, research has focused on GPU computation and processing units, and little work has been done to investigate the power effects of GPU memory accesses [17], [26].

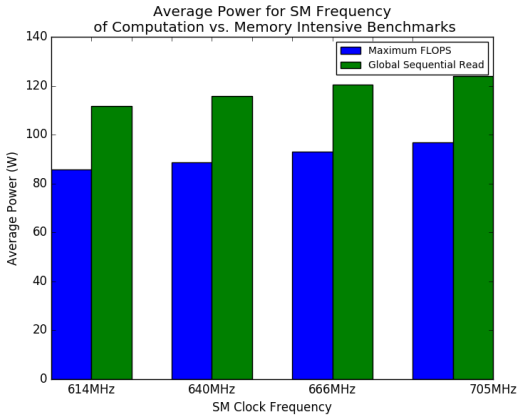

Fig. 1. Measured average power for maximum FLOPs benchmark [12] and global memory sequential read for each possible SM clock frequency. The memory-intensive benchmark consumes significantly more power than the compute intensive benchmark. Notice that this represents the total power consumed by the GPU, and that memory operations increase the average power by a substantial amount.

Figure 1 presents the total GPU power for a memory intensive kernel and a compute intensive kernel with floating point operations respectively. The memory intensive kernel consumes 20-30% more power than the compute intensive kernel. Note that the total GPU power consists of the power of the processing units and the power of GPU memory systems. GPU memory accesses affect power differently than CPU memory accesses. On host CPUs, memory intensive applications generally consume less power than CPU intensive applications [14].

The power effect of GPU memory accesses is complex and determined by many factors. Programming for general purpose GPU has been at a high level and the impact of specific low level instructions is unknown [18], [20]. In addition, energy efficient scheduling, which has been effective for CPU power management, is infeasible for GPU as the scheduler is embedded in the device firmware and not exposed to researchers [5], [20]. Furthermore, memory accesses involve different memory

E2SC2016; Salt Lake City, Utah, USA; November 2016
978-1-5090-3856-5/16/$31.00

levels and locations, and have different data patterns [5]. Consequently, they show different performance and power characteristics. The insights gained for CPU memory accesses are not applicable to GPU due to their different architectures.

To further our ability to optimize performance on power constrained GPU accelerated heterogeneous systems, in this work we investigate the power and performance characteristics of various GPU memory accesses. We take an empirical approach and experimentally examine and evaluate how GPU power and performance vary with data access patterns and software parameters including GPU thread block size. In addition, we take into account the advanced power saving technology dynamic voltage and frequency scaling (DVFS) on GPU processing units and global memory. We present our findings and provide some suggestions and insights to programmers and system designers.

In our experiments we examine a unique category of memory operations. These operations target performance critical memory locations including global memory and shared memory. They test typical data localities between threads and between blocks that appear in GPU kernels applications. These patterns include sequential accesses and regularly strided accesses. We use benchmarks from the SHOC benchmark suite [12] for these experiments.

II. Related Work

GPU memory accesses are commonly known performance bottlenecks, and have been the target of code analysis for performance improvement. Wu et al. examined the performance impact of memory operations using machine learning and static analysis and accordingly partitioned programs to generate efficient code for heterogeneous platforms [26]. Similarly, Chen et. al studied memory placement across uniform memory, and proposed a compiler to statically remap memory[9]. This work could be extended to analyze the power impact of memory operations. Jog et al. studied the memory interferences of multiple concurrent running application and discussed the limitation of schedulers in addressing such interferences [19]. Hong and Kim provided a method for predicting GPU kernel performance based on static analysis of parallelism, particularly with memory instructions in general [17]. Chen et al. show a statistical GPU power learning and prediction model [10]. Our work differs and studies the power characteristics of memory access at finer granularity.

Researchers have attempted to understand the performance and power consumption of GPU applications. McLaughlin et al. presented a power characterization derived from a case study of GPU graph traversals [23]. Jiao et al. experimentally studied several computational kernels and concluded that these characteristics vary more significantly on the GPU than on the traditional CPU, and that the ratio of memory accesses is a predictor for power consumption [18]. Coplin et al. made similar observations, noting that small increases to the dependency on memory accesses can decrease performance and increase power consumption [11]. Our work aims to provide power characterization on the impact of memory transactions at a deeper level.

Numerous studies have investigated how to manage GPU power consumption. Paul et al. used compute and bandwidth sensitiveness of programs to determine the number of computational units, computational unit clock frequency, and memory clock frequency for programs execution to reduce power at minimal performance cost [25]. GreenGPU scheduled DVFS of both GPU cores and memory in a coordinated manner based on their utilizations for energy savings with marginal performance degradation [22]. Similarly, other work proposed to use DVFS exclusively to change performance and power usage [6], [13], [15], [24]. We focus on the memory operations and investigate their important affecting factors including DVFS and software scheduling parameters such as block size.

GPU memory poses a new challenge that differs from traditional main memory. Previous work either studies GPU memory operations for performance improvement or manages power consumption at application level. We aim to study GPU memory operations in great detail and characterize their power and performance to aid algorithm designer and researchers for energy efficient heterogeneous computing.

III. Evaluation Methodology

We experimentally evaluate the power and performance characteristics of various GPU memory accesses, and examine their variations with resource scheduling and power saving technology. In this work we concentrate on the discrete NVIDIA GPGPU cards for high performance computing.

A. Memory Access Patterns

Memory level. We focus on two memory levels: *global memory* and *shared memory*, over which programmers have direct control. There are totally six types of memory in the GPU memory system for NVIDIA GPU cards in four levels from bottom to top: global memory, L2 cache, a read-only texture cache, L1 cache, shared memory, and register [5]. The L2 cache is smaller than global memory and serves to provide a performance increase for data that is read by multiple SMs [5]. The L1 cache is disabled in CUDA compute version 3.5 used in our experiments. Even though the L2 cache plays a role in the performance of global memory, it is not directly managed by programmers. We do not consider texture cache as it requires additional explicit optimizations, which vary by hardware [5].

Global memory is off chip and used by all applications. It stores data migrated to the GPU by the programmer or written back to main memory. As global memory I/O is the slowest form of I/O on GPU, applications move data in global memory into shared memory inside the kernel to improve performance.

Shared memory is on-chip and located in the SM. It is configurable by users and its scope is thread block. Shared memory is frequently used as an optimization for applications where data is reused once inside of a thread block.

Access locality. We test *sequential* access and *strided* access for global memory. With sequential access, a thread operates

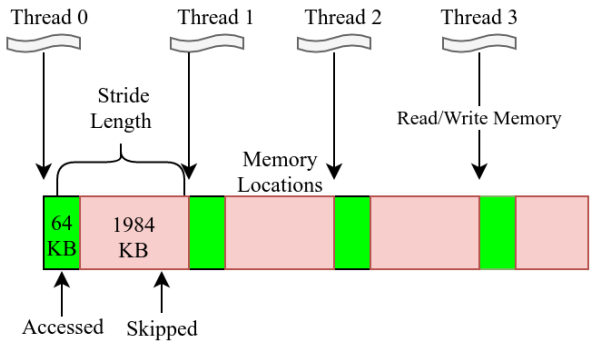

Fig. 2. Memory in the strided experiment is read locally by a single thread and strided by a fixed length between threads.

on sequential chunks of memory. Such access provides performance and power measurements for situations where there is a high measure of locality in data operations. With strided access, each thread reads from contiguous memory addresses, but the memory addresses of different threads have a stride with a fixed length. The strided access is illustrated in Figure 2. Typically, sequential accesses has much better locality than strided accesses.

We do not classify shared memory as sequential or strided access as there is generally no penalty for strided access. Strided memory can cause bank conflicts, which result in a performance penalty. It would take a larger study to examine the affect of the different combinations of bank conflicts. Bank conflicts can be avoided through clever memory layout [16].

In our experiments we include a short period to load a dynamically calculated value into the shared memory. Such processing is necessary because we can not read shared memory until data is written by threads.

Global memory and shared memory are both read-write, so our experiments use both operations.

B. Evaluation Metrics and Variables

We collect both performance and power information. Performance is measured by achieved memory bandwidth in GB/s. Power is measured on the entire GPU card, and consists of the power consumed by SMs, memory, and other devices such as fan. We do not isolate the memory power, as it is not available through the CUDA or NVML APIs. Instead, we present relative comparisons to show the impact of memory accesses. Each experiment is repeated 20 times and statistical average is reported.

We evaluate the power characteristics with the following variables:

Problem size. In general memory access bandwidth increases with problem size until it saturates. For the reported results, we choose a sufficiently large problem size which saturates memory bandwidth.

Block size. Block size, or the number of threads per block, is a software parameter that programmers can specify in programs. Block size is the smallest unit that can be assigned to a SM.

SM clock frequency. Users can use SM clock frequency to schedule GPU performance states. The higher frequency corresponds to a higher performance state with more power consumption.

C. Experimental Environment

All experiments are performed on a NVIDIA K20c GPU card hosted by an Intel Xeon E5-2670 v3 CPU. The SM clock frequencies supported by the K20c are 614 MHz, 640 MHz, 666 MHz, and 705 MHz. The card also supports 758 MHz, which is considered unstable and not used for our experiments. The K20c card also support 324 MHz for both SMs and global memory. We do not report results with this low frequency as the performance is unacceptably poor.

The maximum number of threads per block is limited to 1024 on the CUDA compute version 3.5 [5]. We use various block sizes in our experiments, by incrementing 32-thread between 32 and 1024 threads inclusively. The minimum warp size, which is also the basic unit of parallelism, is 32 threads [5].

The benchmarks are based on the Device Memory benchmark from the SHOC benchmark suite [12], [25]. The Device Memory benchmark from the SHOC benchmark suite performs several different memory operations: Global Memory Read, Strided Global Memory Read, Shared Memory Read, Global Memory Write, Strided Global Memory Write, and Shared Memory Write. Each operation is tested in a unique CUDA kernel. Each benchmark performs the same number of its respective memory operation. Read operations inherently require more registers, so read benchmarks use more registers. The implications of this will be discussed in detail in the Results section.

IV. Experimental Results

For all reported results in this section, though sometimes not shown, the problem size is sufficiently large to create stable results and saturate memory bandwidth. The comparison of these problem sizes can be seen in Figure 3.

A. Sequential Access to Global Memory

Read. Read performance varies with block size and the trend consists of four segments starting at block sizes of 32, 448, and 640, as shown in Figure 4. Each of these segments exists regardless of the clock speed. The first segment at block size 32 delivers the lowest performance due to unused hardware. With only 16 blocks queued on the SM simultaneously, the total number of threads in the SM is 512. *Occupancy*, the percentage of warps in each SM that are queued simultaneously, is 25% of the hardware capacity of 2048 threads [5]. As block size increases to 64, performance quickly reaches its maximum.

The next two segments are caused by thread limitations related to occupancy, shown in Figure 5. The first limitation is due to the number of threads. As mentioned previously, there are 2048 threads available per SM on this architecture. If we choose 672 threads per block, it does not divide the

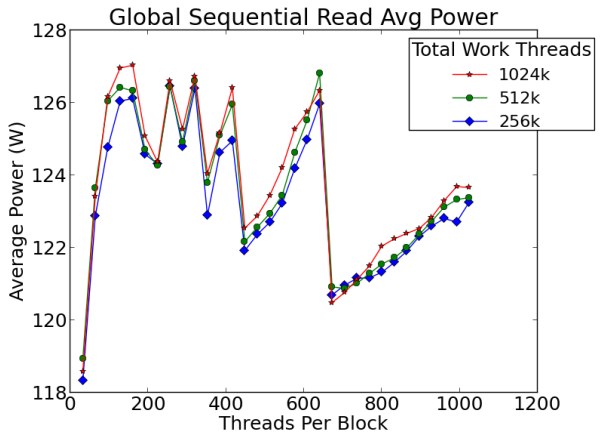

Fig. 3. Power consumption of global memory read benchmark with SM frequency 705MHz at three different problem sizes. Notice that they are statistically equivalent, and that the trend simply becomes more clear with the larger problem size.

2048 threads evenly. We are limited to a maximum occupancy of 98% (2016 threads). Additionally, in order to approach full occupancy, the number of registers available to each warp (32 threads) is 32 registers or less ($\frac{64k \text{ registers}}{2048 \text{ threads}}$). This kernel requires 44 registers per instance, meaning that we can have, at most, 1489 threads. We could fit 2 blocks here, so occupancy would be near 66% ($\frac{2*672 \text{ threads}}{2048 \text{ maximum threads}}$). However, CUDA version 3.5 allocates register resources for 4 warps at a time [1]. This is known as the *warp allocation granularity* [1]. Since there are 21 warps in a 672 thread block, we must allocate 24 warps worth of registers. This causes the required number of registers per block to be 33792. Since 33792 registers per block $> \frac{64k \text{ total registers}}{2}$, half of the total number of registers per SM, we can only allocate a single block. This leaves occupancy at approximately 33%. The drop after block size 640, as well as the other mentioned segments, is due to these limitations. This sudden, large decrease in occupancy accounts for why the 640-672 thread segment is more noticeable than the previous ones. NVIDIA provides an occupancy calculator that can be used to make these calculations for other parameters [1].

Performance increases with the SM clock speed for all segments. Performance gained between 614MHz and 640MHz is quite significant, and the performance gained by the increase from 640MHz to 666MHz is also proportional. However, the performance gain from 666MHz to 705MHz is much less.

Power consumption changes with block size similarly as performance, except for that it fluctuates and has several additional segments. The fluctuations are explained by changes in the amount of used hardware, occupancy, as shown in Figure 5. Power follows the same pattern as occupancy. The fact that these segments do not show up in the performance chart indicates that performance may already be saturated and additional hardware may not provide any benefit.

Power consumption increases proportionally with SM frequency as seen in Figure 4. This is expected, as a higher clock rate corresponds to a high performance state and consumes more power.

Between performance and power, running SMs at very high frequency may not be optimal, especially when power is limited. As frequency increases, power proportionally increases while performance may stop increasing or only marginally increase. In this case, power is not used to deliver performance but instead wasted.

Write. Performance is constant relatively to block size, as seen in Figure 6. Performance does not experience the same immediate increase after block size of 32. This implies that a hardware bottleneck is reached immediately with this block size. Performance in general increases with SM clock frequency. Increasing the clock speed from 614MHz to 640MHz provides the most significant performance increase. Further increasing frequency has diminishing return and performance difference between 666MHz and 705MHz is mostly negligible at some block sizes.

Power, similar as performance, stays relatively stable as block size increases. Unlike performance that has diminishing return from increasing frequency, power increases proportionally with clock frequency. Such difference suggests that it may not always be optimal to increase frequency for performance when power is limited.

Read vs. write. Overall, read and write for most cases have similar performance and power trends. In addition, they have similar performance. The block size 32 is an exception for global read with low performance. We note that power proportionally increases with frequency while performance has a diminishing return.

Read noticeably consumes more power than write. This can be attributed to the L1 cache being disabled on CUDA compute version 3.5. The texture cache is instead enabled, and only read-only objects are stored there [5]. This causes reads to check the texture cache first before hitting the L2 cache, causing performance loss and additional power usage. Write operations always go through the L2 cache, but never go through the texture cache as it is read-only. Our experiments show that reducing the number of registers used by the read kernel, specifically to the point where it can achieve greater occupancy, increases the power consumption and performance. These figures are not shown due to space. The discrepancy in achievable kernel occupancy was discussed earlier in the methodology section.

B. Strided Access to Global Memory

Read. Performance dramatically decreases as block size increases, shown in Figure 7. Interestingly 32 threads per block actually displays the best performance despite the inability to occupy hardware. Profiling with the NVIDA *nvprof* shows that as block size increases, L2 cache hit ratio decreases greatly and global memory bandwidth increases as more data need to be fetched from the global memory. Performance noticeably improves as SM clock frequency increases for all block sizes. Though the improvement becomes less for larger block sizes in general.

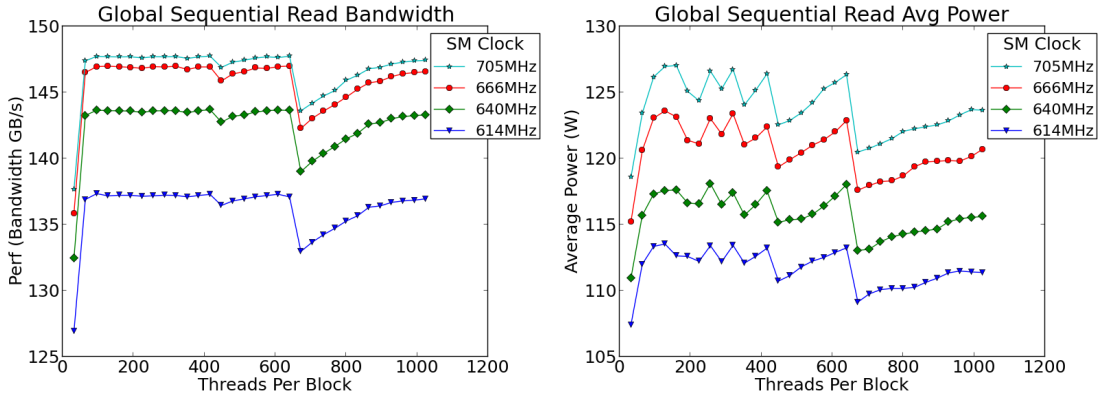

Fig. 4. Performance and power for global sequential read with problem size 1048576 work items.

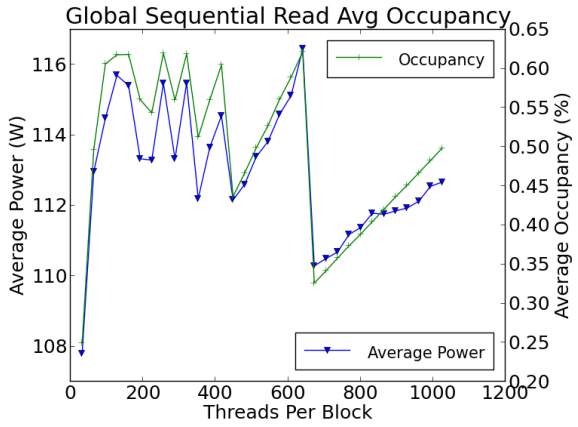

Fig. 5. Occupancy and power of the global memory read at 705MHz SM clock frequency with problem size of 180224 work items. There is a strong correlation between power and occupancy. A smaller problem size is used here due to limits on the amount of occupancy data that can be recorded. Occupancy registers overflow too quickly to be recorded for large data sets.

Power increases with block size in general except after certain block sizes, such as 608, even though performance decreases. This decrease in power consumption is attributed to occupancy, as explained earlier in the global read section. Power also increases with SM clock frequency.

Smaller block sizes are preferable for strided global accesses due to their relatively higher performance and lower power consumption. Our explanation is that smaller block sizes have improved locality and therefore increased L2 cache hits. The larger block sizes incur more cache misses and thus involve more hardware to complete the operations and fetch data from memory.

Write. Performance drastically decreases with block size, as shown in Figure 8. The decline from 32 threads to 64 threads is significant, but afterwards the rate of decline is relatively small. Performance increases with the SM clock speed proportionally for all block sizes.

Power is relatively constant to block size but increases with SM clock frequency. This fits the pattern of occupancy until it is able to fully occupy the hardware, with minor variations due to occupancy afterwards.

Read vs. write. For either read or write, performance does not directly correlate to power. For both operations, block size of 32 is optimal with the best performance. As block size increases, power tends to flatten or increase while performance decreases.

Write achieves significantly lower performance than read while consuming more power. Profiling shows that the number of transactions with the L2 cache is significantly higher for write than read. As a result, bandwidth for both the L2 cache and global memory are reduced. Increased occupancy for strided hardware is associated with a decline in performance due to L2 cache misses. The fact that power stays relatively high for write implies that occupancy still has a large impact on total power consumption even when memory accesses are stalled.

Sequential vs. strided. As expected, we observe that performance for strided read and write are significantly lower than the sequential operations. The poor performance is due to the inability to coalesce multiple accesses to global memory, forcing them to be serialized [4]. Interestingly, strided read consumes less power than sequential read, but strided write consumes more power than sequential write.

C. Access to Shared Memory

Read. Performance increases dramatically with block size until block size reaches 128. With this block size, SM can be fully occupied. This indicates that occupancy plays an important role in shared memory performance. As block size further increases, performance almost stays stable with little variations. The small variation is predictable with occupancy calculation. Performance increases with clock frequency, shown in Figure 9. The increase is not exactly proportional, and the increment from 666MHz to 705MHz is slightly greater than the others.

Power basically follows the similar trend as performance. It increases drastically with block size until block size of 128. It also increases with SM frequency.

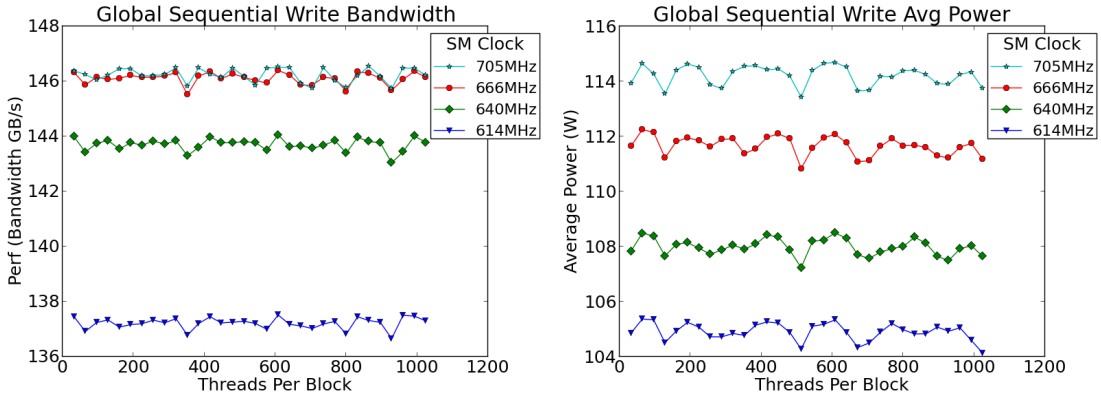

Fig. 6. Bandwidth and power for global sequential write with problem size 1048576 work items.

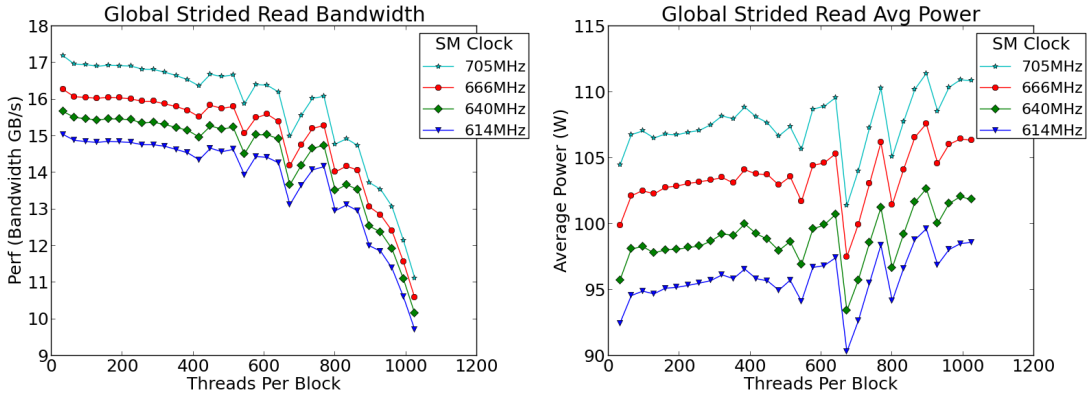

Fig. 7. Bandwidth and power for global strided read with problem size 1048576 work items. Performance is significantly different than those for sequential read. The greatest performance occurs at 32 block size.

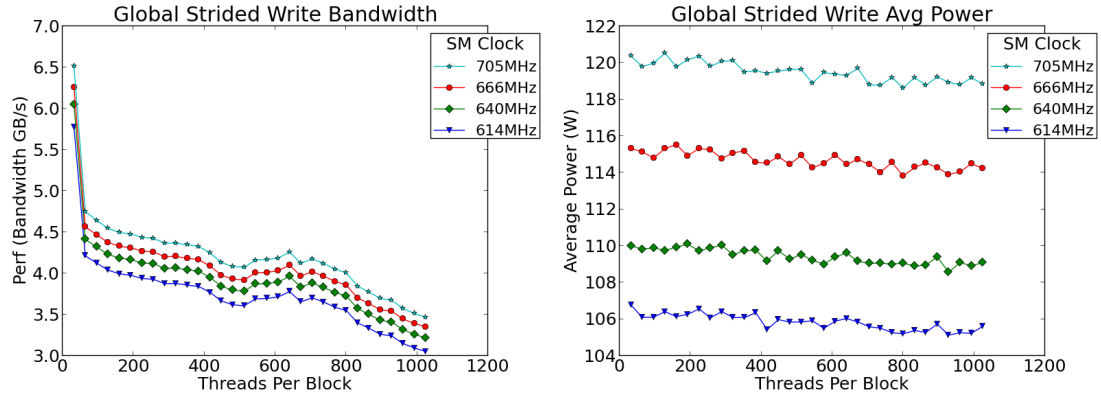

Fig. 8. Bandwidth and power of the global strided write with problem size 1048576 work items. Performance decreases by up to 25% at large block sizes. Power noticeably increases even though performance degrades as block size increases.

For shared read, both performance and power are predictable with specific patterns. Essentially, the range of possible performance and power consumption is large. Depending on programmers' preference to high performance or low power, they can select large or small block sizes.

While not shown, the write operation has similar trends and quantities as the read operation for both performance and power. Limited by space, we don't include those results.

V. Discussion

Memory accesses play an important role in the performance and energy usage of GPUs. The primary memory device units that programmers can control are off-chip global memory, and

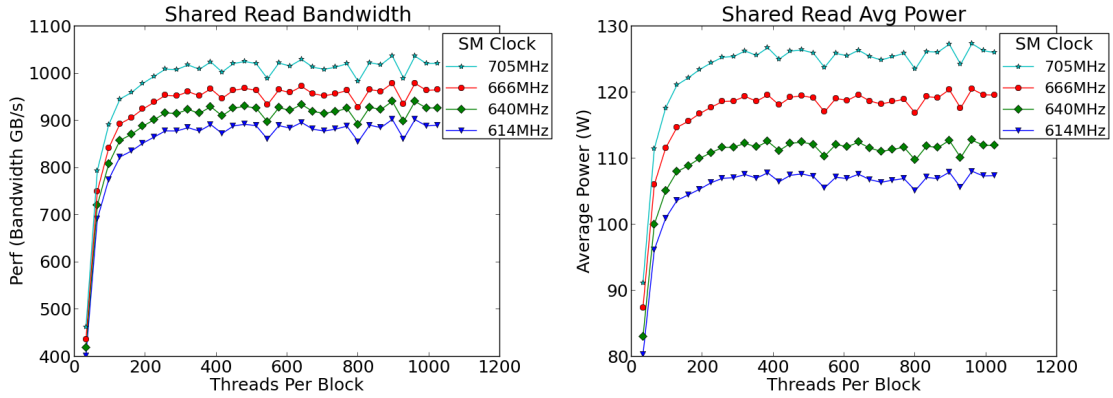

Fig. 9. Bandwidth and power for shared memory read with problem size 1048576 work items. Compared to global sequential read, shared memory read is more stable, achieves higher performance, and consumes slightly more power.

on-chip shared memory. Programmers can read and write to both types of memory. Both types of memory have their own advantages and disadvantages, and the performance and power depend on the data layout in memory, clock speed of the SMs on the GPU, and the block size of the kernel. Problem size does not have any affect once the hardware is sufficiently saturated.

Accessing global memory in a sequential pattern provides predictable results with stable performance above a certain occupancy threshold across all other block sizes. Power is stable as well, with writes using less power than reads due to the read-only texture cache enabled in place of the L1 cache in CUDA compute version 3.5 and read having a higher threshold for performance. It appears that there is room for future study in the trade off between power and performance for read. Read has higher potential for performance at the cost of power if the code is optimized for registers.

Performance gains are proportional to power increases when SM clock frequency increases with the exception of the increase to 705MHz from 666MHz. This final performance gain may not be worth the increased power consumption when power is a limited if the kernel is largely global sequential operations.

Accessing global memory in a strided pattern causes a severe performance degradation from sequential. Interestingly, performance of strided memory accesses rapidly decrease as block size increases for both read and write due to reduced L2 cache hit ratio. Write performance decreases at a slower rate than read performance, but still substantially. The difference in performance is due to difference in occupancy between our benchmarks and possibly the texture cache. Increased block size is directly associated with decreased performance in strided operations. Programmers should limit block size if these operations are necessary.

Power tends to be relatively stable as block sizes increase in this situation, indicating that the least number of threads possible is ideal if strided global memory accesses are the primary operation in a kernel. Strided write uses most instantaneous power, and memory accesses use more power than pure computation. In addition, power is not reduced proportionally with the loss of performance. Increasing the SM clock frequency increases power proportionally to performance at every interval.

Shared memory has greater overall performance for both read and write operations compared to global memory, as expected. Both operations have a similar trend to sequential global memory accesses in that performance increases dramatically until a certain occupancy threshold is met, and then stabilizes. The power usage of shared memory shows a strong relationship with occupancy. Every increment of SM clock frequency shows a proportional increase in power and performance. Shared memory has a power advantage over global memory when theoretical occupancy is unaffected by the change. Shared read has similar power consumption to global read, but shared write shows a slight increase in some cases. Strided accesses are not a concern on shared memory, but are replaced with bank conflicts which may be reduced or avoided by the programmer [16].

Theoretical occupancy can be used to judge the small variations in power consumption variation in general. Theoretical occupancy is also useful for measuring small variations in performance for operations where performance is stable across block sizes, including sequential and shared reads and writes. Occupancy is still a factor in final power usage, even when kernels spend large amounts of time performing memory accesses. Changes in the quantity of registers used, especially when making adjustments between types of memory and number of accesses, can have a large impact on power and performance if occupancy is affected. Large drops in power, and possibly performance, due to occupancy can occur at large block sizes due to the number of resources required for each block. We notice that occupancy is a good predictor of power consumption. We also confirm that occupancy is a good indicator of performance only if the kernel is bound by bandwidth and bandwidth is not saturated [2]. Occupancy can be tricky to calculate theoretically due to the warp allocation granularity. We notice that power still increases for every warp

within the warp allocation granularity, meaning that register warp resources are allocated but power does not increase just from this allocation. Furthermore, power is directly influenced by occupancy regardless of performance limitations. Occupancy, power, and performance show the same trend when performance is not limited. It may be possible to decrease power consumption below a certain limit by using a thread count targeting a particular occupancy.

VI. Conclusion

We presented performance and power characteristics of a set of data access patterns that test specific individual memory operations. We provided some observations and conclusions about the results of these benchmarks, and provided some suggestions for programmers. The main findings are that SM clock frequency is generally proportional to power increases, and that performance may be proportional to the power increase. Global reads and writes may not see the same performance benefit at higher clock frequencies. Global strided operations should try to use smaller block sizes to increase performance. Shared operations are generally stable, and performance is optimal past the 128 thread per block mark. It may be possibly to slightly tune performance for some operations by changing the block size to achieve a desired occupancy. Occupancy has the largest impact on global sequential operations. Global read shows greater performance and power consumption than write when hardware utilization is equal. However, some situations cause reads to use more registers, and therefore more hardware, than write.

In future work, we will study newer GPU architectures for performance differences. We will also study other families of hardware for the same characteristics. We plan to use the findings from this work to design energy efficient heterogeneous computing for real applications.

References

[1] "Cuda occupancy calculator." [Online]. Available: developer.download.nvidia.com/compute/.../CUDA_Occupancy_calculator.xl

[2] "Cuda warps and occupancy." [Online]. Available: http://developer.download.nvidia.com/CUDA/training/cuda_webinars_WarpsAndOccupancy.pdf

[3] "TOP500 Supercomputer Site," http://www.top500.org, accessed: 2016-09-08.

[4] "Whitepaper nvidia's next generation cuda compute architecture: Fermi," NVIDIA, Tech. Rep., 2009.

[5] "Whitepaper nvidia's next generation cuda compute architecture: Kepler gk110," NVIDIA, Tech. Rep., 2012.

[6] Y. Abe, H. Sasaki, S. Kato *et al.*, "Power and performance characterization and modeling of gpu-accelerated systems," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 113–122.

[7] K. Bergman, S. Borkar, D. Campbell *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead," Dept. of Computer Science and Eng., University of Notre Dame, Tech. Rep., 2008.

[8] S. Che, J. W. Sheaffer, M. Boyer *et al.*, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, ser. IISWC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2010.5650274

[9] G. Chen, B. Wu, D. Li, and X. Shen, "Porple: An extensible optimizer for portable data placement on gpu," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 88–100.

[10] J. Chen, B. Li, Y. Zhang, L. Peng *et al.*, "Tree structured analysis on gpu power study," in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, Oct 2011, pp. 57–64.

[11] J. Coplin and M. Burtscher, "Energy, power, and performance characterization of gpgpu benchmark programs," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1190–1199.

[12] A. Danalis, G. Marin, C. McCurdy *et al.*, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735702

[13] R. Ge, R. Vogt, J. Majumder *et al.*, "Effects of dynamic voltage and frequency scaling on a k20 gpu," in *2013 42nd International Conference on Parallel Processing*, Oct 2013, pp. 826–833.

[14] R. Ge, X. Feng, S. Song *et al.*, "Powerpack: Energy profiling and analysis of high-performance systems and applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 5, pp. 658–671, May 2010. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2009.76

[15] J. Guerreiro, A. Ilic, N. Roma *et al.*, "Multi-kernel auto-tuning on gpus: Performance and energy-aware optimization," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, March 2015, pp. 438–445.

[16] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with cuda," in *Gpu Gems 3*, 1st ed., H. Nguyen, Ed. Addison-Wesley Professional, 2007, ch. 39.2.3, pp. 855–866.

[17] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 152–163. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555775

[18] Y. Jiao, H. Lin, P. Balaji *et al.*, "Power and performance characterization of computational kernels on the gpu," in *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, Dec 2010, pp. 221–228.

[19] A. Jog, O. Kayiran, T. Kesten *et al.*, "Anatomy of gpu memory system for multi-application execution," in *Proceedings of the 2015 International Symposium on Memory Systems*, ser. MEMSYS '15. New York, NY, USA: ACM, 2015, pp. 223–234. [Online]. Available: http://doi.acm.org/10.1145/2818950.2818979

[20] S. W. Keckler, W. J. Dally, B. Khailany *et al.*, "Gpus and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sept 2011.

[21] E. Lindholm, J. Nickolls, S. Oberman *et al.*, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar. 2008. [Online]. Available: http://dx.doi.org/10.1109/MM.2008.31

[22] K. Ma, X. Li, W. Chen *et al.*, "Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures," in *Proceedings of the 2012 41st International Conference on Parallel Processing*, ser. ICPP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 48–57. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2012.31

[23] A. T. McLaughlin, "Power-constrained performance optimization of gpu graph traversal," Master's thesis, Georgia Institute of Technology, 2013.

[24] X. Mei, L. S. Yung, K. Zhao *et al.*, "A measurement study of gpu dvfs on energy conservation," in *Proceedings of the Workshop on Power-Aware Computing and Systems*, ser. HotPower '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:5. [Online]. Available: http://doi.acm.org/10.1145/2525526.2525852

[25] I. Paul, W. Huang, M. Arora *et al.*, "Harmonia: Balancing compute and memory power in high-performance gpus," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 54–65. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750404

[26] G. Wu, J. L. Greathouse, A. Lyashevsky *et al.*, "Gpgpu performance and power estimation using machine learning," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 564–576.