# Power and Performance Characterization of Computational Kernels on the GPU

Y. Jiao[1], H. Lin[1], P. Balaji[2], W. Feng[1]

[1]Department of Computer Science
Virginia Tech
{jiaoyang, hlin2, feng}@cs.vt.edu

[2]Mathematics and Computer Science,
Argonne National Laboratory
balaji@mcs.anl.gov

***Abstract*—Nowadays Graphic Processing Units (GPU) are gaining increasing popularity in high performance computing (HPC). While modern GPUs can offer much more computational power than CPUs, they also consume much more power. Energy efficiency is one of the most important factors that will affect a broader adoption of GPUs in HPC. In this paper, we systematically characterize the power and energy efficiency of GPU computing. Specifically, using three different applications with various degrees of compute and memory intensiveness, we investigate the correlation between power consumption and different computational patterns under various voltage and frequency levels. Our study revealed that energy saving mechanisms on GPUs behave considerably different than CPUs. The characterization results also suggest possible ways to improve the "greenness" of GPU computing.**

## I. Introduction

Recent years have seen the growth of accelerator-based supercomputers at a surprising rate. Together with their performance capabilities, GPU-based accelerators are also becoming popular because of their extraordinary energy efficiency, as illustrated by The Green500 List [7], where the first eight machines on the Nov. 2009 list are all built upon an accelerator-based architecture. Despite their significantly better energy efficiency, as compared to CPUs, GPUs are still considered *power-hungry* based on the fact that the thermal design power (TDP) of a high-end GPU, e.g., NVIDIA GeForce GTX 280, could be as high as 236 watts (W). Given that the TDP of a high-end quad-core x86-64 CPU is 125 watts, a GPU can still account for a substantial portion of the overall power usage of the system.

In light of he high power usage of GPUs, the natural question that arises is whether such power usage can be reduced, and if so, what its impact on application performance would be. To address this question, we consider two architectural characteristics of GPUs that allow it to achieve higher performance compared to CPUs: (1) massive on-chip parallelism and (2) higher memory bandwidth. For applications that are compute-bound and can utilize the massive on-chip parallelism, reducing the clock speed of the GPU cores or reducing the number of cores used can impact performance significantly, while the memory bandwidth might not impact them much. On the other hand, for applications that benefit solely from the high memory bandwidth, reducing the clock frequency of the GPU cores might not impact performance too much. Of course, for applications that rely on both, either change (i.e., GPU core frequency or memory frequency) can impact performance.

In this paper, we present a detailed performance and power characterization of three computationally diverse applications (i.e., compute-intensive, memory-intensive, and hybrid) running on the NVIDIA GeForce GTX 280 GPU with varying processor and memory frequencies. In addition, we characterize the impact of such variation on different application kernels.

We classify the applications based on two metrics: the rate of instruction issues and the ratio of global memory transaction to computation instructions. The former metric characterizes how intense the computation is in the application; the latter indicates how memory-dependent the application can be.

Specifically, we study three commonly used scientific computing application kernels – (a) dense matrix multiplication, (b) dense matrix transpose, and (c) fast fourier transform – and show their behavior while varying the frequencies of the GPU cores and memory. In this paper, we do not vary the number of GPU cores utilized because of the restrictions on current GPUs in "suspending" unused GPU cores so as to utilize lesser power, which can cause inconsistent results.

The rest of the paper is organized as follows. We present a brief overview of the current software abstractions for the GPU in Section II. In Section III, we demonstrate our experimental methodology and experimental setup as well as our approach for profiling the GPU applications studied in this paper. In Section IV, we characterize the performance and power of these different GPU applications under varying frequency settings. We present other related work in Section V and summarize the paper in Section VI, together with possible future directions for this work.

## II. Background

In this section, we provide a brief overview of the programming framework for GPUs that we use in our evaluation. Furthermore, we present some aspects related to dynamic voltage and frequency scaling (DVFS) on GPUs that we utilized in this paper.

We do not discuss details on the general GPU architecture itself, but refer the readers to other existing literature for such information [9], [13], [14], [19].

### A. OpenCL

While GPUs have become a prominent model for high-end computing, till recently there was no common programming abstraction that allowed applications to transparently utilize different types of GPU architectures. The Open Computing Language (OpenCL) was recently proposed, originally by Apple Inc., to provide a general software abstraction for different types of accelerators. As a programming framework, OpenCL can be used to program any existing computing device including CPU, GPU, and even FPGA. While OpenCL itself is designed to have a standard set of APIs, its implementation is left to vendors (e.g., AMD, NVIDIA, Intel) and other industry and community users. Thus, irrespective of the implementation, applications written with OpenCL can be designed as *write once, run anywhere*. Due to its extensive support from various vendors and promising cross-platform compatibility, OpenCL was chosen for our study and our test applications are all written with the OpenCL framework.

An OpenCL platform should contain at least one host that connects to one or more OpenCL devices. Each OpenCL device can contain a certain number of compute units (CUs), and each compute unit is further composed by several processing elements (PEs). The function of the host is to start an OpenCL kernel, and once the kernel is ready for execution on an OpenCL device, the host is responsible for transferring the task to the device accordingly.

Each OpenCL program can be generally divided into the host part running on the host and the kernels running on the OpenCL devices. Usually it is the host code which is in charge of setting up the execution environment for the kernels and managing their execution on the OpenCL devices. Each instance of a kernel is called a work-item in OpenCL terminology, and several work-items can be grouped together as a work-group. The mapping between the execution model and the platform model is that each work-item will run on one PE, while one work-group will be assigned to a CU.

Each work-item has access to 4 different types of memory spaces on the OpenCL device. They are listed as follows:

- Global memory, which is the memory space open to all work-items.
- Constant memory, which is part of global memory but for reading only.
- Local memory, which is dedicated to the work-items in the same work-group.
- Private memory, which is allocated for each work-item but not visible to other work-items.

For the purpose of a better visualization, Figure 1 illustrates the relationship between memory and work-items in the OpenCL framework.

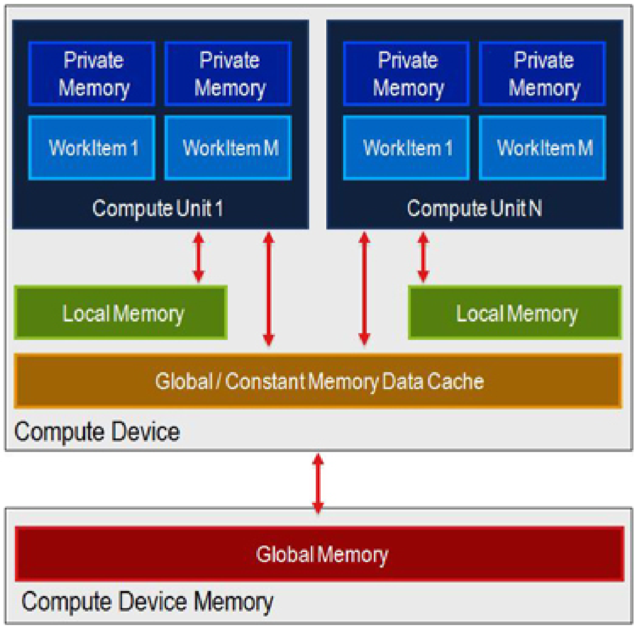

Fig. 1. Memory Hierarchy in OpenCL

Throughout the paper, when we refer to GPU memory, we mean global memory (including constant memory), and the bandwidth of global memory is determined by the memory-clock setting of the GPU.

### B. Dynamic Voltage and Frequency Scaling (DVFS) on GPU

The basic motivation for dynamic voltage and frequency scaling (DVFS) is captured by the equation below:

$$Power \propto Voltage^2 \times Frequency$$

When the frequency is lowered, the power consumption will be lowered proportionally. If the voltage is lowered, the power consumption will drops quadratically, and even cubically, because lowering the voltage also generally lowers the applicable frequency on which the chip can operate. However, there also exists a correlation between the clock frequency and achievable performance, which means that a decrease in frequency can also reduce processor performance. Therefore, as a first-order approximation, DVFS essentially provides us a tool to trade-off performance for power and energy reduction.

According to publicly available documentation [11] on NVIDIA's implementation of DVFS on the GPU, there are *two* predefined performance levels for their GPUs: 'Idle' and 'Maximum Performance.' While there is a third performance level called 'HD Video', it is typically designed for mobile GPU devices and is therefore not available for our GPU, the NVIDIA GeForce GTX 280.

As the name suggests, the 'Maximum Performance' setting fixes the clock speeds to the highest setting to achieve the best possible performance. In the 'Idle' setting, on the other hand, when the driver detects that the GPU is idling, it will automatically lower down the frequencies of the GPU cores and memory to the pre-defined 'Idle' level, which is just enough for the display to work. Once the driver sees a computational kernel that is ready to be executed, the driver will increase the clocks up to the highest level set by user. Both AMD/ATI and NVIDIA provide frameworks [1], [11] that allow the upper limit on the frequency and voltage to be scaled by users. In particular, the core clock of the NVIDIA GTX 280 can be set between 300 MHz and 700 MHz while the memory clock can be set to any value between 300 MHz and 1200 MHz.

## III. Experimental Methodology

In this section, we detail the experimental methodology and setup that we employ in our study.

### A. *Experimental Setup*

The goal of the paper is to investigate the performance and power consumption of typical GPU application kernels under different core and memory clock frequencies. With DVFS support enabled, we have been able to adjust the working frequency of both core and memory of GPU. Specifically, for each configuration, we collect the data from multiple runs of the application and report their average. Only the data related to the kernel execution is recorded since our focus in this work is on GPU alone.

All the results in this paper were taken using the same workstation. The detailed configuration of this base workstation is as follows: Intel Core 2 Quad Q6600, running at 2.33 GHz with 1-GB DDR2 SDRAM*4 and 320-GB Seagate Barracuda 7200.11 hard disk. The GPU card is the NVIDIA GeForce GTX 280, where the default frequency of the computational core is 602 MHz and memory frequency is 1107 MHz. The software environment comprises Ubuntu Linux 9.04 64-bit Linux, CUDA toolkit 2.3a supporting OpenCL, and NVIDIA driver version 190.29. All power measurements in this section are collected with a "Watts Up? Pro ES" power meter and logged to a separate profiling system. Figure 2 shows the hardware setup.

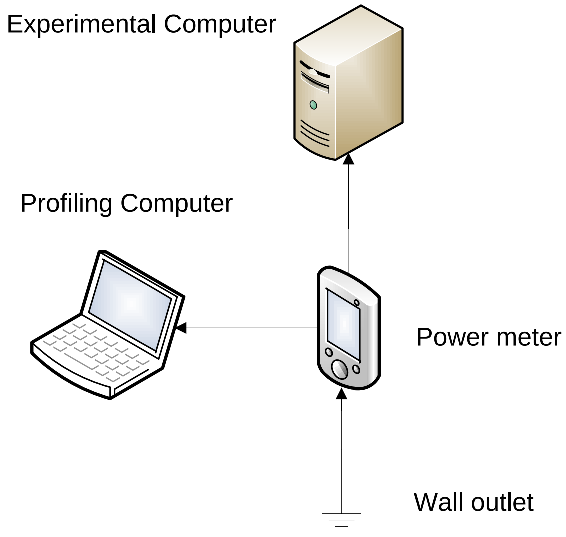

Fig. 2. Hardware setup for the experiment

### B. *Experimental Applications*

In our experiments, we tested three applications which represent compute-intensive, memory-intensive, and hybrid kernels to showcase the performance and power consumption of the NVIDIA GeForce GTX 280 GPU. The three applications that we consider are the compute-intensive *dense matrix multiplication*, the memory-intensive *dense matrix transpose*, and the hybrid *fast Fourier transform* (FFT).

Given that the GPU allows the frequencies of both the computational cores and the memory to be adjusted, we studied both. We did not consider communication-intensive applications since the major factor affecting communication performance on a GPU is neither computation speed nor memory throughput. Thus, we consider such applications to be out of the scope of this study. A detailed discussion about the performance characteristics of communication-intensive applications on GPUs can be found in [20], [21].

Below is a more detailed description of the three test applications:

*1) Dense Matrix Multiply:* The dense matrix multiply (MatMul) kernel is optimized to fully utilize the potential computational power of the GPU. The basic algorithm adopted here is the blocked version of matrix multiplication, which takes advantage of the GPU's coalesced global memory access and fast private/local memory. The bottleneck in this kernel is just the speed of issuing instructions [18].

*2) Dense Matrix Transpose:* The dense matrix transpose (MatTran) kernel is designed to be mostly memory manipulations with only a small amount of necessary computation for each thread's ID and memory indices. To fully utilize the parallel processing ability of GPU, multiple rows of the matrix are read-in and manipulated simultaneously. All intermediate results are stored in the local memory, waiting to written back to global memory.

*3) Fast Fourier Transform:* The fast Fourier transform (FFT) is an efficient algorithm for solving the discrete Fourier transform and its inverse. A recursive GPU implementation of this algorithm computes the result by launching the computing kernel multiple times during the run. For each kernel launch, the global memory chunk has to be read into local memory and computed by the computing unit. So, the speed of the FFT algorithm would be affected by both the GPU core and memory speeds.

### C. *Experimental Profiling*

In order to understand the kernel execution, we collected profiling data for each kernel via the NVIDIA OpenCL Visual Profiler. To verify whether a kernel is compute-intensive or memory-intensive, the rule of thumb proposed in [18] is to use the "global memory to computation cycle" ratio to indicate the memory-transaction intensity of a GPU kernel. This ratio

| Application | instruction | gputime(us) | cputime(us) | gld_request | gst_request | gld_tran | gst_tran |
|---|---|---|---|---|---|---|---|
| MatMul | 203135064 | 991327 | 991360 | 5041152 | 4376 | 33566720 | 52448 |
| MatTran | 1118754 | 91527 | 91562 | 17480 | 17480 | 104864 | 1677824 |
| FFT | 2568388 | 17452 | 17611 | 52432 | 52432 | 314650 | 314650 |

TABLE I
PROFILING OF APPLICATIONS TESTED

$R_{mem}$ can be estimated using the following formula:

$$R_{mem} = \frac{Number(Global\ Memory\ Transactions)}{Number(Computation\ Instructions)}$$

Because the global memory on the GPU is two orders of magnitude slower than local memory [12], we only count the number of global memory accesses for the above ratio.

Another important ratio for measuring the performance of the GPU is the rate of issuing instruction ($R_{ins}$), which can be obtained by:

$$R_{ins} = \frac{Number(Computation\ Instructions)}{gputime}$$

where the $gputime$ is the actual execution time of the kernel on the GPU.

We also want to make sure that the kernel execution has minimal kernel launch overhead; this can be verified through the overhead ratio $R_{over}$ which is defined as:

$$R_{over} = \frac{cputime - gputime}{cputime}$$

where the $cputime$ is the corresponding time elapsed on the CPU.

The profiling information for $R_{mem}$, $R_{ins}$, and $R_{over}$ for each of the three applications is shown in Table I. Note that both the number of global memory requests and transactions ('_tran' in the table) are reported because a single global memory request might need multiple global memory transactions. Thus, it is the actual number of memory transactions that actually determines the latency on memory access. The memory request would be counted as one instruction that is deducted when computing the actual number of computation instructions.

Besides profiling the applications using the default factory clock setting, we also attempted to profile the application under different GPU clock settings. However, due to the overhead introduced by the profiler, the profiling data under different GPU clock settings look relatively similar. Therefore, we chose not to report this data in this study.

To obtain the accurate number of computation instructions, we deduct the number of memory requests from the total instruction count. The calculation is further complicated by the fact that the number of instructions is reported for one compute unit while the number of memory requests and transactions is reported for three compute units sharing the same memory control unit [12], which is accounted for in the profiling information provided in Table II.

| Application | $R_{mem}$ | $R_{ins}$ | $R_{over}$ |
|---|---|---|---|
| MatMul | 5.6% | 203215711 | 0.03% |
| MatTran | 53.7% | 12095895 | 0.04% |
| FFT | 8.3% | 145165788 | 0.9% |

TABLE II
APPLICATION CHARACTERIZATION

### D. Evaluation Metrics

We use the following evalutation metrics to compare the different DVFS system configurations for each application:

**Time:** The execution time is measured for the kernel execution of each application. To minimize noise, the same application is run multiple times under each setting, and the average time (T) is used.

**Performance:** A common metric for the high-performance computing (HPC) community is FLOPS, which stands for *floating-point operations per second*. However, this metric may not apply to certain applications that do not focus on computation, such as the matrix transpose kernel. In this case, we adopted another metric for performance of matrix transpose, using MBPS, which stands for *megabytes per second*, to indicate the throughput it has at run time.

**Energy:** Energy (E) is measured for the whole system when executing the kernel on the GPU.

**Power:** Power (P) is reported using the average power, calculated by the energy used per execution time unit:

$$Power = Energy/Time$$

**Power Efficiency:** We use the ratio of performance per power as the indicator of power efficiency:

$$Power\ Efficiency = \frac{Performance}{Power}$$

## IV. EXPERIMENTAL RESULTS

In this section, we present our experimental results from measuring the performance and power consumption of the three application kernels described in Section III. For any given problem with a fixed size, it is always true that the power efficiency metric, defined by the performance-to-power ratio, is equivalent to the energy efficiency, defined by the computation-to-energy ratio, as illustrated by the following equation:

$$\frac{Performance}{Power} = \frac{Computation}{Time \times Power} = \frac{Computation}{Energy}$$

Thus, with respect to the energy consumed by each application under the different settings in our experiment, it is the configuration with the best power efficiency that consumes the least energy.

For each application running on the GPU, we present its performance, power consumption, and power efficiency as a set of graphs, where the x-axis plots the GPU core frequency and the legend denotes the GPU memory frequency. As a reference point, we record the static power consumed by our experimental GPU system. The power usage of the system while idling is 111.3 watts without the GPU and 142.7 watts when the GPU is installed in the system.

### A. *Matrix Multiply*

Figure 3 shows the performance, power consumption, and power efficiency when running the dense matrix multiplication kernel under different frequencies for the GPU cores and memory. As shown in the figure, the performance is determined solely by the GPU core frequency, which is not surprising because of the relatively low $R_{mem}$ and the highest $R_{ins}$ among the three application kernels that we ran, showing about two million computation instructions per compute unit. That is, the relatively low $R_{mem}$ and high $R_{ins}$ point to a compute-intensive application whose performance, relative to execution speed, is directly proportional to the frequency of the GPU computing cores and largely independent of GPU memory frequency. As a consequence, we can run the GPU at the highest GPU core frequency and the lowest GPU memory frequency to achieve nearly optimal performance while reducing the (absolute) power consumption by approximately 15 watts and improving the (absolute) power efficiency by nearly 20 MFLOPS/watt.

However, the *relative* reduction in power consumption is only 5%, and the *relative* improvement in power efficiency is only 4%. These results point to several potential inter-related reasons for the minor impact of GPU memory frequency on power consumption and power efficiency: (1) the memory intensity of matrix multiply is very low, i.e., $R_{mem}$ is only 5.6%; (2) the dynamic power range of GPU memory is relatively small, e.g., $\sim$20 watts; and (3) GPU memory is less power hungry than GPU cores.

### B. *Matrix Transpose*

Figure 4 shows the behavior of the matrix transpose kernel under various clock settings. The set of horizonal lines in the performance figure of matrix transpose clearly shows that its performance is determined solely by the GPU memory clock and is independent of GPU core frequency. The memory-bounded performance can be attributed to the high $R_{mem}$ of matrix transpose, which is approximately 10 times higher than the $R_{mem}$ for the matrix multiplication kernel.

As a consequence, we can run the GPU at the highest GPU memory frequency and the lowest GPU core frequency to achieve nearly optimal performance while reducing the absolute power consumption by more than 20 watts (or approximately 3 watts for every 50-MHz decrement in frequency) and improving the absolute power efficiency by approximately 100 MFLOPS/watt. In contrast to the matrix multiply scenario, the GPU core frequency has a larger relative impact on power consumption and power efficiency, improving each by 8-9%.

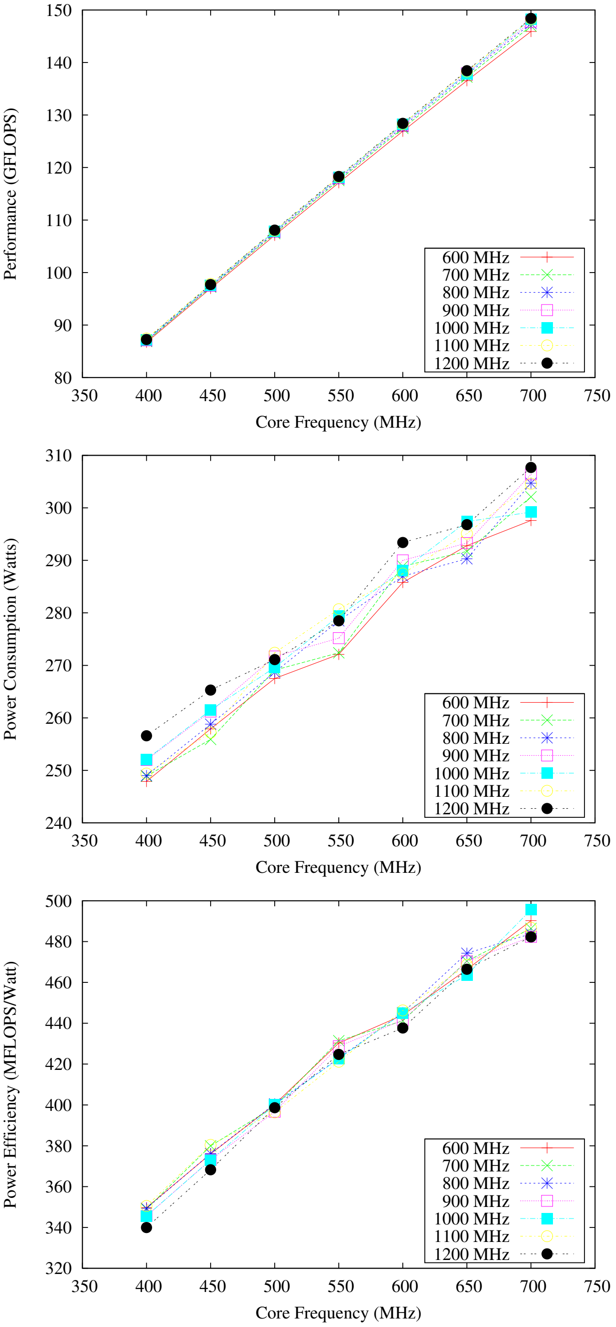

Fig. 3. Matrix Multiply on NVIDIA GeForce GTX 280

Further, compared to the compute-intensive matrix multiplication kernel, the memory-intensive matrix transpose kernel has a much lower power requirement and a narrower dynamic power range, as shown in Table III. With the GPU memory frequency fixed at 1200 MHz, the lowest power for executing the matrix transpose is 216.4 watts, the highest one is just 237.3 watts, i.e., only a difference of 20.9 watts. The generally low-power consumption of matrix transpose can be explained by the fact the $R_{ins}$ of matrix transpose is only a tenth of the

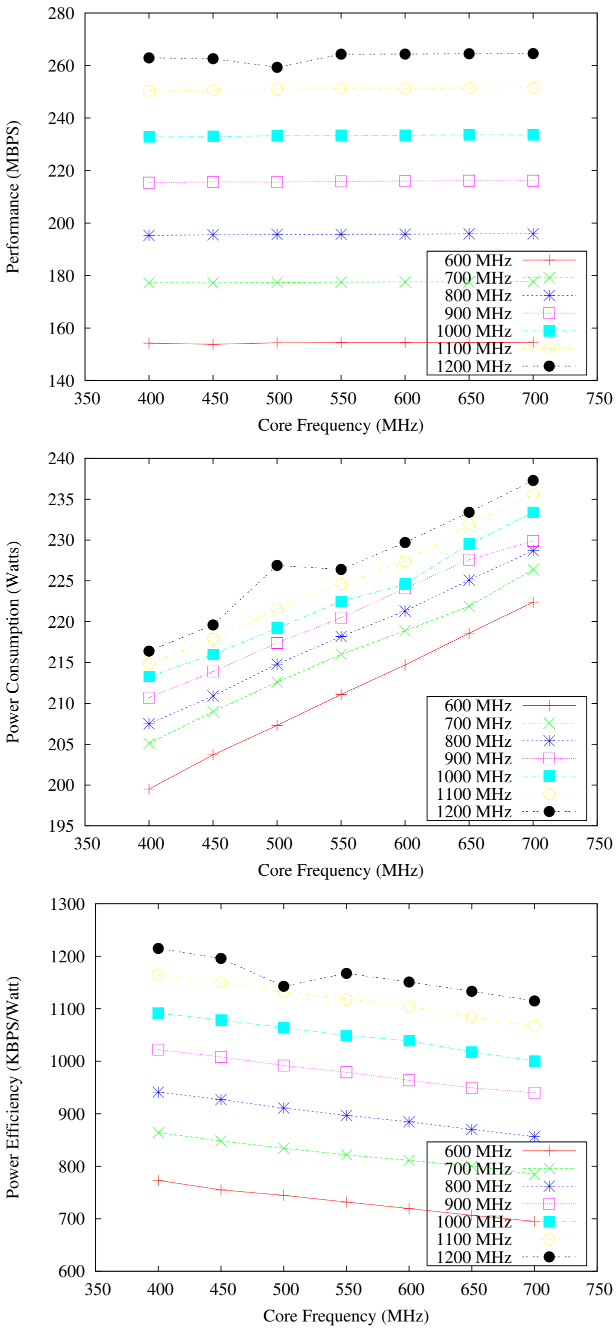

Fig. 4. Matrix Transpose on NVIDIA GeForce GTX 280

$R_{ins}$ of matrix multiply, which indicates that the compute units are almost idle when executing matrix transpose compared to matrix multiply. This result also shows the overall impact of the compute units on the total power consumption of the GPU.

### C. Fast Fourier Transform

Figure 5 shows the results of running a two-dimensional (2-D) FFT kernel. We see that the performance is bound not only by the speed of the compute units but also by the memory throughput. The multi-dimensional FFT algorithm requires a matrix transpose as a post-processing step after the computation. Because the size of fast local memory is very limited on GPU, a multi-dimensional FFT algorithm often requires significant data exchange between local memory and global memory on GPU. Since the global memory is usually two orders of magnitude slower than the local memory, it is oftentimes the bottleneck for exchanging data such that its throughput would largely determine the computation latency in the matrix transpose step. Thus, the performance of multi-dimensional FFT algorithm on GPU is bound by both the speed of compute unit and the throughput of global memory.

An interesting finding with respect to the performance is that the higher the GPU core clock, the more the extent that memory frequency affects performance. This implies that when compute units get faster they spend more time waiting for the memory modules to feed them. Compared to the completely memory-bounded or completely compute-bounded kernels, the multi-dimensional FFT is bound by both compute speed and memory throughput. Because DVFS scheduling on the GPU can be done along two dimensions, i.e., GPU core frequency and GPU memory frequency, it is possible to achieve similar performance with different core and memory settings. For example, in Table IV, all three frequency settings produce similar performance numbers, but the power usage is obviously higher when the GPU core speed is higher at 650 MHz.

| Core (MHz) | Performance (MBPS) | Power (Watts) |
|---|---|---|
| 400 | 262.9 | 216.4 |
| 450 | 262.6 | 219.6 |
| 500 | 259.3 | 226.9 |
| 550 | 264.3 | 226.4 |
| 600 | 264.4 | 229.7 |
| 650 | 264.5 | 233.4 |
| 700 | 264.6 | 237.3 |

TABLE III
PERFORMANCE AND POWER WITH GPU MEMORY AT 1200 MHZ

| Core (MHz) | Memory (MHz) | Performance (GFLOPS) | Power (Watts) |
|---|---|---|---|
| 550 | 1100 | 69.60 | 289.64 |
| 600 | 1000 | 71.01 | 290.57 |
| 650 | 900 | 69.70 | 298.36 |

TABLE IV
SIMILAR PERFORMANCE WITH DIFFERENT POWER

## V. RELATED WORK

Prior to the emergence of OpenCL, the most fundamental work related to general-purpose GPU computing was with Stanford's Brook [3], [5] for the AMD Stream Processor [3] and NVIDIA's Compute Unified Device Architecture (CUDA) [10].

There has been a lot of previous work that evaluates the performance of different application kernels using the

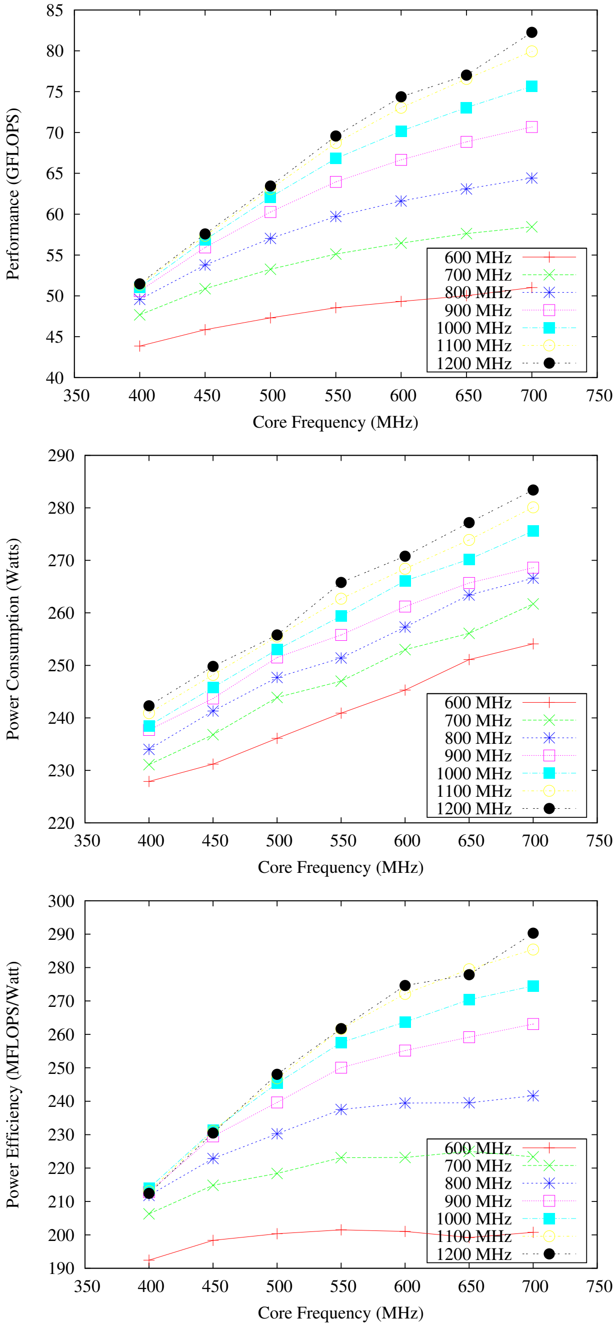

Fig. 5. 2-D FFT on NVIDIA GeForce GTX 280

Compute Unified Device Architecture (CUDA) [17], which is a precursor to OpenCL. FFT has also been studied in-depth in the paper [8], resulting in a five-fold speedup over NVIDIA's implementation of FFT algorithm. In [4], the authors proposed a "divide-conquer" approach for solving multi-dimensional FFTs using a matrix transpose to cope with limited memory space on computers—we adopted this implementation of FFT for our work.

With respect to the power usage of GPUs, there has been a surprising dearth of work in this area, with only a handful of papers published on the subject. However, given the importance of the power usage, all the major GPU vendors have attempted to address this issue by adding built-in power management to their GPUs. For example, NVIDIA's PowerMizer [11] and ATI's PowerPlay [2] throttle the GPU clock speeds when idle and attempt to minimize the run-time power usage without affecting performance. Others have attempted to model the power efficiency of GPUs [15] as well as investigating energy-aware high-performance computing on GPUs [16]. Rodina [6] is a benchmark suite focusing on characterizing heterogeneous computing, which also touches the power efficiency of different applications on different platforms. However, none have investigated these properties with the DVFS approach we adopted in our study.

## VI. Conclusion and Future Work

As a promising architecture for supercomputers, GPU-based computing is receiving significant attention for its performance and power efficiency. In order to help the HPC community better understand the power characteristics of different application kernels on the GPU, we have conducted this study to characterize the performance and power of various application kernels under varying frequency settings. Based on our findings, the GPU application kernels' performance and power consumption are largely determined by two characteristics: the rate of issuing instructions and the ratio of global memory transactions to computation instructions.

For future work, we intend to conduct our study on other GPUs, including from AMD and Intel. Given the software abstraction used in this work is OpenCL which is supported by most accelerator platforms, a fair comparison running the same code across platforms is possible. In addition, we believe a complete and comprehensive modeling of GPU performance and power consumption would be very useful.

## Acknowledgements

We would like to thank the NSF Center for High Performance Reconfigurable Computing (CHREC) for their support through NSF I/UCRC Grant IIP-0804155. This work was also supported in part by the following NSF grants: CNS-0915861 and CNS-0916719.

## References

[1] AMD. OverDrive, 2008. http://game.amd.com/us-en/drivers_overdrive.aspx.

[2] AMD. PowerPlay, 2008. http://ati.amd.com/products/powerplay/index.html.

[3] AMD. AMD Stream Processor. http://ati.amd.com/products/streamprocessor/index.html, 2010.

[4] David H. Bailey. Ffts in external or hierarchical memory. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, pages 211–224, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *International Conference on Computer Graphics and Interactive Techniques*, pages 777–786. ACM New York, NY, USA, 2004.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC '09: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Wu-chun Feng and Kirk Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12):50–55, 2007.

[8] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[9] Nvidia. CUDA C Programming Guide, version 3.1. *NVIDIA Corporation*.

[10] NVIDIA. NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html, 2007.

[11] NVIDIA. PowerMizer 8.0 Intelligent Power Management Technology. Technical report, June 2008.

[12] NVIDIA. *NVIDIA OpenCL Programming Guide*. 2010.

[13] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879 –899, May 2008.

[14] John Owens. Gpu architecture overview. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 2, New York, NY, USA, 2007. ACM.

[15] K. Ramani, A. Ibrahim, and D. Shimizu. PowerRed: A Flexible Power Modeling Framework for Power Efficiency Exploration in GPUs. In *Worskshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2007.

[16] M. Rofouei, T. Stathopoulos, S. Ryffel, W. Kaiser, and M Sarrafzadeh. Energy-Aware High Performance Computing with Graphic Processing Units. In *Workshop on Power Aware Computing and System*, December 2008.

[17] S. Ryoo, C.I. Rodrigues, S.S. Stone, S.S. Baghsorkhi, S.Z. Ueng, and W.W. Hwu. Program optimization study on a 128-core GPU. In *The First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[18] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[19] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235 –246, 2010.

[20] Shucai Xiao, Ashwin M. Aji, and Wu-chun Feng. On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit. In *15th International Conference on Parallel and Distributed Systems (ICPADS)*, Shenzhen, China, December 2009.

[21] Shucai Xiao and Wu-chun Feng. Inter-Block GPU Communication via Fast Barrier Synchronization. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Atlanta, Georgia, USA, April 2010.