Statistical Power Modeling of GPU Kernels Using Performance Counters

Hitoshi Nagasaka,*† Naoya Maruyama,*† Akira Nukada,*† Toshio Endo,*† Satoshi Matsuoka,*†,‡
*Tokyo Institute of Technology, {jin, naoya, nukada}@matsulab.is.titech.ac.jp, {endo, matsu}@is.titech.ac.jp
†JST, CREST
‡National Institute of Informatics

***Abstract*—We present a statistical approach for estimating power consumption of GPU kernels. We use the GPU performance counters that are exposed for CUDA applications, and train a linear regression model where performance counters are used as independent variables and power consumption is the dependent variable. For model training and evaluation, we use publicly available CUDA applications, consisting of 49 kernels in the CUDA SDK and the Rodinia benchmark suite. Our regression model achieves highly accurate estimates for many of the tested kernels, where the average error ratio is 4.7%. However, we also find that it fails to yield accurate estimates for kernels with texture reads because of the lack of performance counters for monitoring texture accesses, resulting in significant underestimation for such kernels.**

I. Introduction

Modern graphics processing units (GPUs) are being increasingly used to accelerate a wide variety of scientific applications such as physical simulations [1], bioinformatics [2], and medical analysis [3]. This trend has been made possible with the recent advancements in programmability embodied as CUDA and OpenCL, which greatly simplify exploiting much higher peak performance of GPUs than conventional multi-core CPUs.

Integration of GPUs into the already power-consuming HPC systems, however, must be carefully evaluated with respect to the impacts on system power efficiency [4]. The peak power of the latest high-end GPUs is as large as 250W, while the typical CPU consumes only 100W at maximum. This does not necessarily indicate that the GPU has lower energy efficiency since the advantage in performance can offset the larger power consumption. However, whether offloading computations to GPUs can actually yield better energy efficiency than solely using CPUs cannot be determined statically and depends on performance and power characteristics of specific applications.

This paper presents a statistical approach to estimating GPU power consumption. While the power behavior can be observed with hardware power sensors, one could not assume that such sensors are available in typical HPC machines, since they are not currently a standard component in commercially available GPUs. Therefore, identification of power consumption can only be done with a conjecture based on observable behavior. To do so, we study a statistical approach using GPU performance profiles that can be obtained without special hardware. More specifically, we collect performance profiles of various GPU kernels by using performance counters, and simultaneously measure the power consumption of the kernels using hardware power sensors. Once the profiles are collected, we derive a statistical model of the GPU that estimates the power consumption of a GPU kernel from its performance counters. The resulting model can then be used to estimate the power consumption without special hardware sensors. This can be especially useful in GPU clusters where all machines have the same type of GPU; we could derive a power model on one of the machine, which could then be used on the remaining machines.

We demonstrate the effectiveness of the above approach for NVIDIA CUDA GPUs. We collect both power and performance profiles from 49 CUDA kernels in publicly available programs, namely the CUDA SDK and the Rodinia benchmark suite [5]. Our linear regression modeling finds that instruction and memory throughputs are the two most highly correlated factors with power consumption. We evaluate the estimation accuracies of the model with cross validation and show that it can estimate the power consumption from the performance counters with an average error ratio of 4.7%. We also show that our model fails to accurately estimate power consumption of kernels with texture reads because of the lack of performance counters for texture accesses.

II. Background

A. Target GPU Architecture

While there are several types of GPUs available in the market, the current majority in HPC is a PCI-Express extension card that contains both a GPU itself with several hundred MB to a GB of DRAM. One example is the recent NVIDIA GPUs, such as GeForce GTX 285 and Tesla C1060, both of which essentially employ the same architecture with different frequency and capacity configurations. This section briefly describes its architecture for the discussion in the subsequent sections.

The architecture of NVIDIA GPUs supports both graphics and throughput-oriented general-purpose computing [6]. It consists of dozens to hundreds of Streaming Processor (SP) cores, of which eight are organized into a Streaming Multiprocessor (SM). The eight SP cores in an SM execute the same instruction but consume different data (i.e., SIMD parallelism). Each SM has own register files, software-managed on-chip cache, and read-only constant memory cache, and can be

controlled independently from other SMs (i.e., SPMD parallelism). Recent high-end models, such as GeForce 285 GTX and Telsa C1070, for example, have 30 SMs or 240 SPs in a single GPU, each clocked at 1.48 GHz and 1.44 GHz.

A single GPU board has an external off-chip DRAM of several hundreds of mega bytes to giga bytes. The GPU DRAM system is optimized for block data accesses: The bus width can be as large as 512 bits, which is eight times larger than the DDR3 interface used in standard PC and server memory systems. Thus, fine-grained random accesses are not as efficient as in DDR3 memories. Another notable difference between the standard CPU and the GPU memory systems is that the latter has only very limited data caches, and thus most of data loads and stores to DRAM in the GPU are not cached, taking 400-600 cycles of latency. However, unlike the standard CPU, each SM is also equipped with a software-managed scratch pad memory called shared memory, which can be accessed from the eight SPs with as low latency as registers.

The GPU can be programmed in C language with the Compute Unified Device Architecture (CUDA) extension [7], [8]. The primary abstractions include *kernels*, *threads*, *thread blocks*, and *grids*. A GPU kernel is a program that can be invoked on CUDA GPUs. A CUDA thread represents a single flow of scalar execution of a kernel, and has own set registers and constant memory. A CUDA thread block is a group of threads that cooperatively run on a single SM. The threads in the same thread block can communicate through the shared memory, where the weak consistency protocol is implemented via a block synchronization primitive. The typical number of threads in a thread block ranges from 64 to 256. A CUDA grid is the collection of all thread blocks that are launched for executing a single kernel. The number of blocks inside a grid can be as large as billions, depending on specific application data sizes.

The GPU memory system is directly available in the CUDA programming model. The off-chip DRAM can be accessed through several different abstractions in CUDA, including *global memory*, and read-only *texture memory* and *constant memory*. Among them, global memory is the most often used one for storing a large amount of data in GPU DRAM, which is analogous to the heap memory in standard CPU-based systems, except that the GPU does not have the memory swapping capability. The global memory is significantly optimized for block data accesses, since the underlying hardware always accesses a block of 32, 64 or 128 bytes. This grouping of memory accesses is called coalescing in the CUDA terminology.

The typical flow of kernel executions starts by transferring input data from host memory to the global memory by DMA through PCIe lanes. When the input data become available on the GPU side, a grid is launched with the user-specified configuration of numbers of threads and thread blocks. Each thread then loads its part of input data from the global memory, optionally shares it with the other threads in the same thread block to reduce load accesses, and performs computation using the loaded data, followed by writes of results to the global memory.

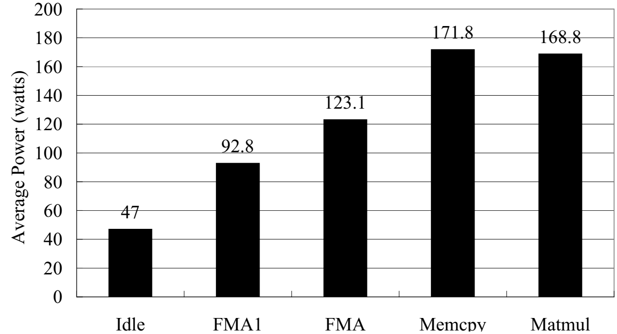

Fig. 1. Comparison of Average Power of CUDA Kernels.

B. GPU Power Consumption

A single GPU board has two sources of input power: PCI Express lanes and auxiliary connectors. The former contains 12V and 3.3V lanes, and the latter has one or two of 12V inputs. The aggregated power inputs support a board that can consume as much as 250W. Fundamentally, power measurement of a device can be done by clamping current probes around its power lines. This is possible with the auxiliary inputs to GPUs because they have separate, flexible cables for the power lines. To attach a clamp sensor to PCI power lanes, which cannot be separated with the rest, we use a PCI Express riser card where each lane can be separated by removing the covering layer of the lanes.

Fig. 1 shows the average power of various CUDA kernels running on a GeForce 285 GTX as well as its idle power (The detailed machine configurations can be found in Section IV). The maximum power of the GPU is listed as 204W in the specification. FMA is a CUDA program that fully exercises computation units in the GPU chip so as to achieve the near peak performance; FMA1 is a scaled-down version of FMA that only launches a single thread. Both are compute intensive and performs negligible memory operations. Memcpy copies a memory chunk of 200MB within GPU DRAM. Matmul computes a multiplication of two 8192^2 matrices. As shown in the graph, the average power varies from 92W to 171W, which is lower than the theoretical peak. Therefore, using a single fixed metric such as the maximum as an estimate of GPU power consumption can result in inaccurate estimation of energy efficiency of the GPU.

III. Statistical Power Modeling

To identify power consumption of GPU kernels without relying on special hardware probes, we propose a statistical methodology that learns correlation between performance and power profiles.

A. Performance Profiling

The latest CUDA release as of this work is CUDA version 2.3, which has 21 performance counters for analyzing

TABLE I
CUDA Performance Counters [9]

Name	Description
gld_32b	32-byte global memory load transactions
gld_64b	64-byte global memory load transactions
gld_128b	128-byte global memory load transactions
gst_32b	32-byte global memory store transactions
gst_64b	64-byte global memory store transactions
gst_128b	128-byte global memory store transactions
local_load	Local memory loads
local_store	Local memory stores
branch	Number of branches
divergent_branch	Number of divergent branches
instructions	Instructions executed
warp_serialize	Number of thread warps that has bank conflicts
tlb_miss	Number of TLB misses

kernel performance [9]. Among them this study uses the 13 counters listed in Table I. The CUDA Profiler traces kernel invocations, collecting their performance profiles that include the performance counter values as well as the run time. Note that, because the number of simultaneously monitored counters is limited to four, we run each kernel multiple times to collect more than four counter values. Another limitation is that each counter records the number of specific events only on a single SM instead of the whole GPU.[1] Since a GPU has multiple SMs (e.g., 30 SMs in GeForce 285 GTX), the raw values of the counters do not directly indicate the actual number of total events on the GPU. If the workload is evenly distributed among multiple SMs, we can assume that the collected profile correctly reflects the behavior of the whole GPU; on the other hand, if it is imbalanced, i.e., the number of thread blocks executed on each SM differs significantly, we could end up with biased counter values. To remedy this limitation, we limit our modeling to those kernels that have enough thread blocks to avoid large load imbalance. The required number of thread blocks depends on the number of SMs in a specific GPU; in our study with GeForce 285 GTX, we only use kernels that launch at least 120 thread blocks.

Some important events in CUDA kernels are not monitored by the performance counters. For example, data reads from DRAM through texture hardware are not recorded. Since texture reads are cached in on-chip memory, they are especially beneficial when accessing irregular data, and thus some kernels extensively use texture reads instead of global memory reads. However, since texture reads are not monitored by any counters, the DRAM accesses in such kernels are not detected by our performance profiling. Thus, our modeling cannot accurately estimate power consumption of such kernels, as shown in the experimental evaluation. More accurate modeling remains to be a subject of future work.

[1]Although not documented in the official CUDA Profiler manual [9], counters related to memory loads and stores appear to be collected from a Texture Processing Cluster, which is a group of three SMs.

B. Power Profiling

To measure the power consumption of a GPU, we use three current clamps (URD HCS-20-10-AP): one for 12V PCI lane, one for 3.3V PCI lane, and another for 12V auxiliary input. The current clamps are connected to a 32-channel National Instruments analog-to-digital converter (NI PCIe-6259) via a Synergetech ST-30600 power measurement system. The A/D converter is attached to a different machine as a PCI Express extension card, where we monitor the GPU power consumption by reading samples from the probes at a particular interval. The A/D converter can read one sample in 800 ns, or 2.4 μs for the three probes. In this study, we configure the power monitor to sample the three probes at 10 μs interval.

We measure the power consumption of each GPU kernel invocation by using the current probes and assuming the voltage of each power line to be constant according to the specification. To do so, we compute the average of the sampling data for the kernel execution time. However, since the monitor and monitoree are two different machines, we cannot assume their clocks are completely synchronized. To minimize the error due to the clock difference, each kernel is repeated for multiple times so that the total time becomes longer than one second. Since both machines use Network Time Protocol, we expect errors due to clock difference is minimal.

C. Power Modeling

1) Linear Regression: Linear regression is a standard technique to model the correlation between independent and a dependent variable by assuming linearity between the variables. We attempt to derive a linear model where independent variables are based on performance counters of a kernel execution, and the dependent variable is the power consumption of the kernel. Although the linearity between the performance counters and power consumption cannot be proved, in practice linear regression often works effectively for a wide variety of real-world data.

Let p_k be the average power of a kernel and $c_i, 1 \leq i \leq n$ be the performance counter values converted to the per-second scale, where n denotes the number of counters. We convert the raw performance and power profiles to the per-second scale by dividing them with the kernel execution time recorded in the CUDA Profiler output. Linear regression allows us to derive a model as:

$$p = \sum_{i=1}^{n} \alpha_i c_i + \beta \quad (1)$$

where α_i denotes the contribution of counter c_i to the power consumption and β is the constant intercept.

The above model allows us to estimate the average power of a given kernel rather than instantaneous power at arbitrary timings. Since the performance counters are only readable after the completion of each kernel, our model is able to model GPU power consumption at the granularity of kernel calls.

2) Model Selection: Table I includes multiple counters to record accesses to global memory. Since linear regression requires more training data for models with a larger number

of dependent variables, we aggregate them to a single virtual counter. Specifically, we introduce a virtual counter named `gld`, which is defined as `gld_32b` + 2 × `gld_64b` + 4 × `gld_128b`. The coefficients of the latter two counters reflect the ratio of the chunk size to that of `gld_32b`. We aggregate `gst_32b`, `gst_64b`, and `gst_128b` to virtual counter `gst` in the same manner. Instead of the original counters, we use `gld` and `gst` so that the number of independent variables is reduced to nine, which would require less training data for robust regression analysis.

Linear regression with all the performance counters may not yield the most accurate model, especially when the size of training data is limited. For example, using a subset of counters, such as `instructions`, `gld`, and `gst`, thus reducing the complexity of the regression, could achieve better accuracy. We experimentally evaluate several variations of models in Section IV.

3) Avoiding Overfitting: Our linear modeling performs regularization to avoid overfitting to training data. Ridge regression is an extension of linear regression with regularization, where parameter λ adjusts the fitness of the derived model to the training data [10]. We use ridge regression with the λ value determined by n-fold cross validation.

IV. Evaluation

We apply the proposed statistical modeling to a set of CUDA kernels to evaluate its accuracy. We evaluate modeling accuracy by ten-fold cross validation. The training data is divided into ten equally-sized subsets, and for each subset we compute squared errors of the model trained with the remaining nine subsets. The experimental platform is a 64-bit Linux machine with NVIDIA GeForce GTX 285. The detailed specification is listed in Table III.

A. Training and Evaluation Samples

For this evaluation, we use programs included in CUDA SDK and the Rodinia benchmark suite, as listed in Table II. We particularly select the Rodinia benchmark suite because it covers a wide variety of parallel application kernels (i.e., Berkeley dwarfs [11]), which would improve the effectiveness of statistical training. We take both power and performance profiles of 49 kernels in the programs by running each kernel for one second. Note that we do not use all the kernels included in CUDA SDK and the Rodinia benchmark suite, such as kernels that use less than the minimum number of thread blocks (120 in this work), and those that do not run on our evaluation platform.

We evaluate our modeling against two sets of kernels in the above programs. The first set contains all the kernels except for those that use texture reads, and another is the set of all kernels. As discussed in Section III-A, texture reads cannot be monitored by the current set of CUDA performance counters. Hence, kernels with texture reads would be difficult to estimate their power consumption accurately. Furthermore, including such kernels in model training could have adverse effects on the accuracy of the resulting model with respect

TABLE II
Profiled CUDA programs for evaluation.

CUDA SDK	Rodinia Benchmark
MersenneTwister	Back Propagation
MonteCarlo	Breadth-First Search
binomialOptions	Cell
boxFilter	CFD Solver
convolutionSeparable	Dynproc
convolutionTexture	Gaussian
dwtHaar1D	Hotspot
dxtc	Kmeans
fastWalshTransform	Leukocyte
histogram64	Needleman-Wunsch
matrixMul	Nearest Neighbor
quasirandomGenerator	SRAD
scalaProd	Streamcluster
scan	
scanLargeArray	
simpleCUBLAS	
transpose	

TABLE III
Evaluation Platform

GPU	NVIDIA GeForce GTX 285
CPU	AMD Phenom 9850 Quad-Core Processor 2.5 GHz
Chipset	AMD 790FX
Memory	4GB
OS	64-bit Fedora 10
GPU Driver	190.18
Compiler	gcc4.3.2 and CUDA Toolkit 2.3

to the other non-texture kernels. In the programs listed in Table II, six kernels in convolutionTexture, boxFilter, Kmeans, and Leukocyte use texture reads. We first evaluate the model accuracy with the remaining 43 kernels, and next discuss the limitation of the current modeling with all the kernels.

B. Results without Texture Reads

While the performance profiles contain nine counters, using all of them does not necessarily lead to the most accurate model since the more parameters in regression, the more training data is needed for robust analysis. We first evaluate the following seven sets of dependent variables. Here, let `mem` be the number of all the memory access events, i.e., `mem` = `gld` + `gst` + `local_load` + `local_store`.

1) *Mem:* This model considers only memory access events. To further simplify the model, we use `mem` rather than `gld`, `gst`, `local_load`, and `local_store`. While this does not take the other activities into account, simplifying modeling could derive a better model especially when the size of training data is limited.
2) *Inst:* Instead of memory accesses, this model uses the instruction count, `instructions`, for power estimation.
3) *Mem/Inst:* This model aims to improve the accuracy by using both `mem` and `instructions`.
4) *Mem/Inst/Branch:* This model uses `branch` in addition to Mem/Inst. Since executing more branches can lead to less efficient usage of GPU chip and memory, power might be reduced. Incorporating the `branch` counter to the model could better reflect such power behavior.

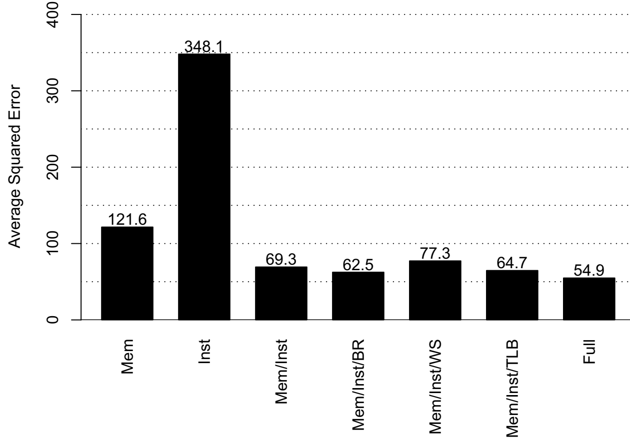

Fig. 2. Comparison of seven models in linear regression.

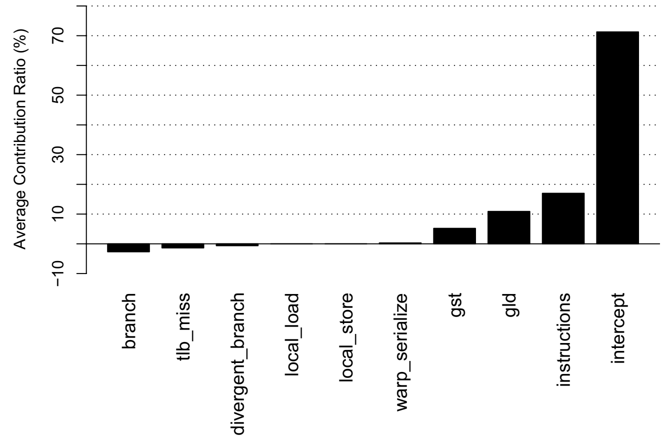

Fig. 4. Average contribution ratios of counters and model intercept to estimated power.

5) *Mem/Inst/WS:* This model uses `warp_serialize` in addition to Mem/Inst. Similar to branches, large `warp_serialize` reflects inefficient GPU execution, and might decrease instantaneous power.
6) *Mem/Inst/TLB:* This model attempts to capture the effects by TLB misses to power consumption by including `tlb_miss` counter in addition to memory and instruction counters.
7) *Full:* This model uses all the available counters. It uses `gld`, `gst`, `local_load`, and `local_store` instead of `mem`, as well as `instructions`, `branch`, `divergent_branch`, `warp_serialize`, and `tlb_miss`.

Fig. 2 shows the average squared error of the seven regression models. The most accurate model is Full model, whose average squared error is 54.9 and the average error ratio is 4.7%. Simpler models, especially Model/Inst/Branch, also yield comparable accuracies. This result indicates that in practice we could reduce the number of performances counters in the power modeling without significantly impacting the estimation accuracy, which would require a less number of profiling runs for power estimation. However, the result also suggests that memory accesses and instruction counts are essential in power modeling. Since they require nine counters in total, and CUDA allows only four counters to be logged in a single run, our power modeling requires at least three profiling runs for reasonable accuracy.

Fig. 3 compares the actual average watts, denoted by black bars, to the watts estimated by the Full model, denoted by the gray bars. We see that our power modeling achieves fairly accurate estimates in many kernels; however, several kernels do exhibit relatively large errors. One of the largest errors occurs in the estimation of a kernel, `RandomGPU`, in program MersenneTwister. The estimated power is lower than the actual by 23 watts. This error is likely to be caused by `local_load` and `local_store`. The particular kernel is the only sample in our data set that has loads and stores to local memory. In our cross validation, when estimating the power consumption of the kernel, no kernels in the training data set have local memory accesses, yielding a model without `local_load` and `local_load` terms. Therefore, power consumption due to local memory accesses in the MersenneTwister kernel cannot be estimated. This problem can be mitigated by collecting more training data with local memory accesses. Another kernel with a large error is a kernel in simpleCUBLAS (`sgemm`), where our estimate is lower than the actual by 18 watts. Although no exact details of the kernel are publicly known because its source code is not available, it is likely to extensively use the fused multiply-add instruction. The instruction would exercise more hardware transistors than the other instructions, consuming more power per instruction count. However, our power model is unable to reflect such differences in instruction types accurately, since they are not recorded by the current set of CUDA performance counters.

Fig. 4 shows the average contributions of the performance counters and the model intercept to the estimated power of each kernel. As shown in the graph, the constant factor (i.e., β in Equation 1) accounts for approximately 70% of power consumption. As expected, `instructions`, `gld`, and `gst` have the largest correlations with power. The most negatively correlated counters include `branch` and `tlb_miss`. These events imply inefficient usage of compute and I/O resources in GPU, which in turn could reduce instantaneous power consumption. Note that kernels with a larger number of such events do not necessarily consume less energy, since they would take longer time to perform their computation.

C. Results with Texture Reads

The programs listed in Table II includes six kernels that read GPU DRAM though texture hardware, namely `GICOV_kernel` and `dilate_kernel` in Leukocyte, `kmeansPoint` in Kmeans, `d_boxfilter_rgba_x` in boxFilter, and `convolutionColumnGPU` and `convolutionRowGPU` in convolutionTexture. Our performance profiling is unable to

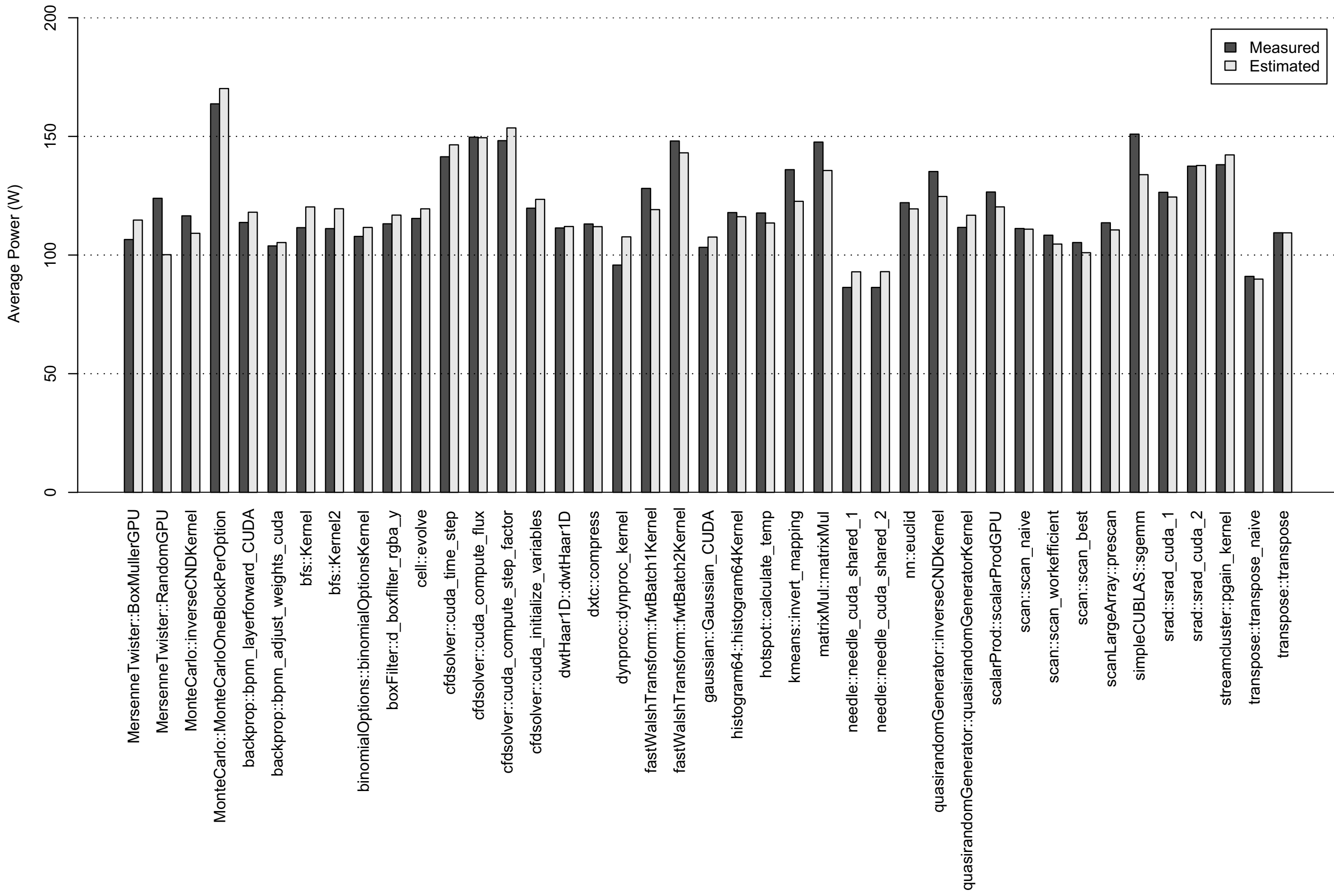

Fig. 3. Comparison of actual and estimated power of each kernel in Full linear regression without texture reads.

identify texture reads since no counters for monitoring texture accesses are available. We apply the same power modeling to the data set with texture read kernels, and identify the limitation of the current power modeling.

We apply the Full model to the data set including texture reads. Fig. 5 shows the actual and estimated power of each kernel. The average squared error is 140.4. The reason of the much larger error compared to the case with no texture kernels is that the power consumption of texture kernels are estimated to be much lower than the actual. For example, the actual power of `d_boxfilter_rgba_y` is 151W, while our estimate is 109W. The other texture kernels show similar discrepancies. The underestimation is caused by the lack of performance counters that record texture accesses in the CUDA profiler version 2.3. This could be mitigated with the recent profiler enhancements introduced in CUDA version 3.0, which adds counters for the numbers of texture hits and misses.

V. Discussions

The experimental results demonstrate that the GPU performance counters can be utilized to estimate power consumption of the GPU very accurately. These positive results encourage us to explore model-based techniques for reducing power consumption in GPU-based computing. One of the potential directions that we are currently investigating is to adjust GPU clock frequencies automatically so that power consumption is minimized within the given performance degradation threshold. For example, memory-bound kernels might have significant optimization opportunities for reducing power consumption with minimum performance degradation by scaling down non-memory-related clock domains such as the shader clock. By statically deriving multiple power models for different frequency settings, we envision that at run time we would first run the kernel for collecting performance counter values, and then determine the optimal clock configurations for the kernel, which would be subsequently used in the remaining runs. This approach requires that the kernel can be run multiple times with the same performance behavior, which would be a reasonable assumption in many of common HPC applications.

Our current realization of the power modeling, however, has several challenges toward such online optimizations. One of such issues is that the current modeling uses 13 counters, whereas the GPU only allows four counters to be monitored simultaneously. We obtain more than four counter values by running the same kernel multiple times with the same input data. This could be acceptable if the model is used solely in offline scenarios; however, in online settings, minimizing the number of data-collecting runs would be important for

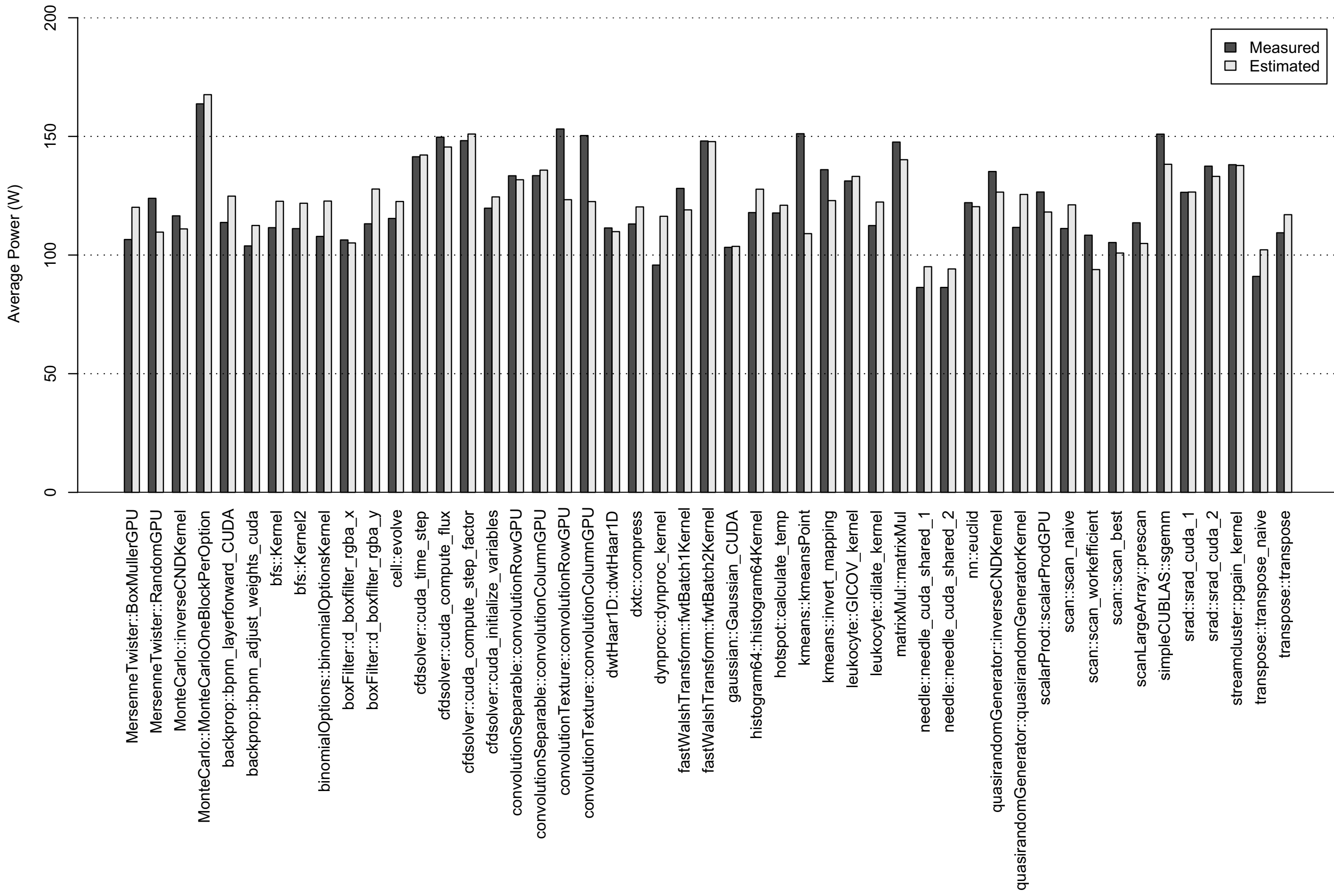

Fig. 5. Comparison of actual and estimated power of each kernel in Full linear regression with texture kernels.

reducing the associated run-time overhead. The model accuracies with the different sets of performance counters shown in Fig. 2 imply that while using all the available counters would lead to the most accurate model, several other models with smaller sets of performance counters would exhibit comparable accuracies. In our future work, we will further investigate the contributions of performance counters in improving model accuracies and study the technique for finding accurate models while minimizing the number of counters.

VI. Related Work

While performance acceleration with GPUs have been demonstrated with various applications [12], power consumption of GPUs has been little studied. Collange et al. studied power consumption of several generations of NVIDIA GPUs executing several primitive operations such as memory reads, arithmetic operations, and texture accesses [13]. Their experiments on memory accesses showed that energy per memory request is much lower when the request is serviced by texture cache than when DRAM is involved. Ma et al. studied correlation of power consumption and performance of graphics applications [14]. They also used performance profiles to build a statistical model of GPU power consumption, but unlike us they used NVIDIA PerfKit, which is designed to identify the usage of GPU components such as vertex and pixel shader usage, texture units, and ROP units, by conventional graphics applications, and thus their performance profiles do not contain GPGPU-specific events such as global memory accesses. Our experiments showed that accesses to global memory has the largest factor in power consumption of GPU kernels. Huang et al. compared a GPU accelerated version of a bioinformatics application with a multithreaded CPU version, and showed that the GPU version outperforms the CPU version in performance, energy consumption, and energy efficiency [15]. While they needed hardware power sensors in their evaluation, once our statistical model is derived, it can provide an accurate estimate of the GPU kernel without relying on hardware sensors.

Estimation of power consumption of conventional CPUs have also been studied. Isci and Martonosi presented a power model for Pentium4 processors using performance counters [16]. They decompose the power consumption of the target processor into 22 components, and manually derive a model for each component using processor performance counters. Unlike us, their model derivation extensively relies on knowledge of internal architectural details of the target processor. A more black-box approach to power modeling is proposed by Bellosa et al., where statistical linear regression is employed to estimate power consumption of CPUs

using their performance counters [17]. Lee and Brooks also uses regression-based statistical modeling with performance counters for accurate and efficient microarchitectural performance and power modeling [18]. This paper applies a similar methodology to significantly different hardware, i.e., the GPU, and demonstrated its effectiveness for a wide variety of GPU kernels. Kansal et al. proposes a power modeling technique for virtual machines to account power usage in data-center environments accurately [19]. Their JouleMeter framework infers power dissipation of virtualized hardware components, such as CPUs, memory, and hard disks, by measuring actual physical resource usage by those virtual components and identifying its linear correlation to power dissipation. The modeling presented in this paper could be employed in the JouleMeter framework to account the GPU power consumption.

VII. Conclusions

This paper proposed a statistical approach for estimating power consumption of GPU kernels. It uses the CUDA performance counters to obtain performance profiles of kernels and finds linear correlation between the performance profiles and power consumption by statistical model learning. Our experimental evaluation demonstrated high estimation accuracy for many of the tested kernels: The average error ratio was 4.7%. However, we also identified a limitation in the proposed approach because of the lack of performance counters on texture accesses.

Our modeling still has several challenges. First, texture accesses need to be included into our power model for better accuracy, which would be possible with the recent enhancements to the CUDA profiler. Also, we apply the modeling method developed in this work to the latest NVIDIA Fermi GPU architecture and evaluate its effectiveness. Next, we will extend the current modeling technique for the other parts of GPU-accelerated systems, such as data transfers between host and GPU memory so as to estimate the total power consumption of GPU applications. Finally, we will explore potential applications of the presented power model for optimizing power consumption in GPU-based computing, such as dynamic frequency scaling and CPU-GPU adaptive scheduling.

Acknowledgments

We thank Masashi Sugiyama at Tokyo Institute of Technology for his valuable feedback on our statistical modeling. This work is partially supported by the JST-CREST Ultra-Low-Power HPC Project, MEXT Grant-in-Aid for Young Scientists (22700047), HPC-GPGPU Microsoft Technical Computing Initiative, and NVIDIA CUDA Center of Excellence.

References

[1] L. Nyland, M. Harris, and J. Prins, "Fast N-body simulation with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley Professional, August 2007, ch. 31.

[2] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformatics*, vol. 8, no. 1, pp. 474+, 2007.

[3] S. S. Stone, J. P. Haldar, S. C. Tsao, W. Mei, Z. P. Liang, and B. P. Sutton, "Accelerating advanced MRI reconstructions on GPUs," in *Proceedings of the 5th conference on Computing frontiers*, 2008, pp. 261–272.

[4] T. Mudge, "Power: A first-class architectural design constraint," *Computer*, vol. 34, no. 4, pp. 52–58, 2001.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. pp. 44–54.

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

[7] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[8] "NVIDIA CUDA Programming Guide," http://developer.nvidia.com/object/cuda.html.

[9] NVIDIA Corporation, "The CUDA Profiler," 2009.

[10] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[11] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, W. L. P. David A. Patterson, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, December 18 2006.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370–1380, 2008.

[13] S. Collange, D. Defour, and A. Tisserand, "Power consumption of GPUs from a software perspective," in *Workshop on Using Emerging Parallel Architectures for Computational Science (in conjunction with ICCS'09)*. Springer, 2009, vol. 5544, ch. 92, pp. 914–923.

[14] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical power consumption analysis and modeling for GPU-based computing," in *Workshop on Power Aware Computing and Systems (HotPower '09)*, 2009.

[15] S. Huang, S. Xiao, and W. Feng, "On the energy efficiency of graphics processing units for scientific computing," in *Fifth Workshop on High-Performance, Power-Aware Computing (HPPAC'09)*, 2009, pp. 1–8.

[16] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[17] F. Bellosa, A. Weißel, M. Waitz, and S. Kellner, "Event-driven energy accounting for dynamic thermal management," in *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, September 2003.

[18] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM, 2006, pp. 185–194.

[19] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. Bhattacharya, "Virtual machine power metering and provisioning," in *ACM Symposium on Cloud Computing (SOCC)*, June 2010, pp. 39–50.