Energy, Power, and Performance Characterization of GPGPU Benchmark Programs

Jared Coplin
Department of Computer Science
Texas State University
coplin@txstate.edu

Martin Burtscher
Department of Computer Science
Texas State University
burtscher@txstate.edu

***Abstract*—This paper studies the effects on energy consumption, power draw, and runtime of a modern compute GPU when changing the core and memory clock frequencies, enabling or disabling ECC, using alternate implementations, and varying the program inputs. We evaluate 34 applications from 5 benchmark suites and measure their power draw over time on a K20c GPU. Our results show that changing the frequency or the program implementation can alter the energy, power, and performance by a factor of two or more. Interestingly, some changes affect these three aspects very unevenly. ECC can greatly increase the runtime and energy consumption, but only on memory-bound codes. Compute-bound codes tend to behave quite differently from memory-bound codes, in particular regarding their power draw. On irregular programs, a small change in frequency can result in a large change in runtime and energy consumption.**

I. Introduction

GPU-based accelerators are widely used in high-performance computing and are quickly spreading in PCs and even handheld devices as they not only provide higher peak performance but also better energy efficiency than multicore CPUs. Nevertheless, large power consumption and the required cooling due to the resulting heat dissipation are major cost factors in HPC environments. To reach exascale computing, a 50-fold improvement in performance per watt is needed by some estimates [1]. Moreover, battery life is a key concern in all types of handhelds such as smartphones.

For these and other reasons, energy-efficient computing has become an important research area. While many hardware optimizations for reducing power have been proposed or are already deployed, software techniques are lagging behind, particularly techniques that target accelerators like GPUs. However, to be able to optimize the energy efficiency of GPU code, we first need to develop a good understanding of how the power draw and performance of such programs interact. To this end, we investigate the energy consumption, the power, and the active runtime (i.e., the time the GPU is computing) of a large number of codes.

It is well known that code optimizations can improve GPU performance and that reducing the GPU's frequency lowers power and performance. But what about energy? Some studies report a one-to-one correspondence between active runtime and energy. But these studies focus on regular programs and only vary the inputs, not the code itself or the GPU frequency. So it is unclear whether there are implementations or GPU configurations that help energy more than active runtime, how big of an impact such changes can make, and whether there are differences between compute- and memory-bound or regular and irregular codes. The goal of this paper is to answer these and related questions to determine whether software has the potential to play an important role in making future accelerators more energy efficient.

To perform this study, we took 34 programs from 5 GPGPU benchmark suites. We ran each program on a Kepler-based K20c compute GPU at 3 different clock frequencies and with or without using error-correcting code (ECC) in main memory. In addition, select programs are run with alternate implementations and inputs.

This paper makes the following key contributions.

1) It is the first paper to measure the energy and power of a large number of GPGPU benchmark programs.

2) It experimentally validates some common expectations regarding GPU power and energy characteristics.

3) It shows that different frequencies, hardware configurations, inputs, and implementations can impact energy, power, and performance by different amounts, demonstrating that these metrics can be affected independently of each other.

4) It analyses what frequencies, settings, implementations, and inputs tend to help with which aspect and why.

5) It exposes key differences between regular and irregular codes as well as compute- and memory-bound codes.

6) It makes general recommendations for selecting meaningful subsets of these programs for conducting studies on GPGPUs using the built-in power sensor.

The rest of this paper is organized as follows. Section II discusses related work. Section III provides an overview of the GPU we study. Section IV describes the evaluation methodology. Section V presents and analyzes our most interesting measurements. Section VI summarizes our findings, provides guidelines, and draws conclusions.

II. Related Work

We are not aware of other extensive measurement-based studies on the energy, power, and runtime of GPU programs.

There are many papers that investigate Dynamic Voltage and Frequency Scaling (DVFS) on CPUs to reduce power and energy. For example, Kandalla et al. demonstrate the need to design software in a power-aware manner to minimize perfor-

mance overheads and to balance performance and power savings on a power-aware DVFS-capable cluster system [15]. Furthermore, Li uses DVFS across multiple clock domains, such as shaders and texture domains, to increase energy efficiency on GPUs [19]. Pan et al. also use DVFS-based solutions and show that sometimes expending more energy does not result in a large performance benefit [26]. In addition to varying the frequency, Korthikanti and Agha explore how changing the number of active cores affects the energy consumption and provide guidelines for the optimal core count and frequency for a given algorithm and input size [16]. Freeh et al. perform a related study on a cluster where they evaluate how using different frequencies and numbers of compute nodes affect the power and performance of MPI programs [10]. There are also papers that highlight the lack of a standardized power measurement methodology for energy-efficient supercomputing [31] or talk about how ignoring power draw as a design constraint in supercomputing will result in higher operational costs and diminished reliability [9].

Several publications propose and use analytical models to investigate power and energy aspects. For instance, Li and Martinez establish an analytical model for looking at parallel efficiency, granularity of parallelism, and voltage/frequency scaling [20]. Diop et al. and Choi et al. propose clusters of micro-benchmarks to model time, energy, and power [5] [8]. Wang conducted a similar study that uses a GPU simulator to model the effects on energy with frequency scaling and concurrency throttling [32]. Lorenz et al. explore compiler-generated SIMD operations and how they affect energy efficiency [22]. They acknowledge the need to optimize both hardware and software to obtain an energy efficient system. Some analytical models target GPUs. For example, Chen et al. institute a mechanism for evaluating and understanding the power consumption when running GPU applications [4]. Ma et al. use a statistical model to estimate the best GPU configuration to save power [24]. One simulation-based paper on thermal management for GPUs discusses methods for managing power through architecture manipulation [30]. Another simulator includes a modified and extended version of a CPU power model to model the power of compute GPUs [13].

Similar to our study, there are several papers that report the energy consumption of actual GPU hardware. For instance, Lee and Kim measure the energy consumption of GPUs using a digital multimeter [18]. Gosh et al. explore some common HPC kernels running on a multi-GPU platform and compare their results against multi-core CPUs [12]. Ge et al.'s study evaluates the effect of DVFS on the same type of GPU we are using [11]. A paper by Zecena et al. measures *n*-body codes running on different GPUs and CPUs [33]. Lucas et al. measure the power and energy to verify their GPU power simulator [23]. Jiao et al. investigate how changing the core and memory frequencies affect performance and power [14]. All of these publications investigate just a few programs. Our study is the first to characterize the energy behavior of a large number of GPU codes, which we believe is necessary to obtain general insights.

III. GPU Architecture

This section provides an overview of the architectural characteristics of the Kepler-based Tesla K20c GPU we use for our study. It includes an on-board power sensor that allows the direct measurement of the GPU's power draw.

The K20c consists of 13 streaming multiprocessors (SMs). Each SM contains 192 processing elements (PEs). Whereas each PE can run a thread of instructions, sets of 32 PEs are tightly coupled and must either execute the same instruction (operating on different data) in the same cycle or wait. This is tantamount to a SIMD instruction that conditionally operates on 32-element vectors. The corresponding sets of 32 coupled threads are called warps. Warps in which not all threads can execute the same instruction are subdivided by the hardware into sets of threads such that all threads in a set execute the same instruction. The individual sets are serially executed, which is called branch divergence, until they re-converge. Branch divergence hurts performance and energy efficiency.

The memory subsystem is also built for warp-based processing. If the threads in a warp simultaneously access words in main memory that lie in the same aligned 128-byte segment, the hardware merges the 32 reads or writes into one coalesced memory transaction, which is as fast as accessing a single word. Warps accessing multiple 128-byte segments result in correspondingly many individual memory transactions that are executed serially. Hence, uncoalesced accesses are slower and require more energy than coalesced accesses.

The PEs within an SM share a pool of threads called thread block, synchronization hardware, and a software-controlled data cache called shared memory. A warp can simultaneously access 32 words in shared memory as long as all words reside in different banks or all accesses within a bank request the same word. The SMs operate largely independently and can only communicate through global memory (main memory in DRAM). Shared memory accesses are much faster and more energy efficient than main memory accesses.

IV. Methodology

A. Benchmarks

We study programs from the LonestarGPU v2.0 [21], Parboil [27], Rodinia v3.0 [28], and SHOC v1.1.2 [29] benchmark suites as well as a few programs from the CUDA SDK v6.0 [7]. We chose which codes to evaluate based on their active runtimes to obtain a sufficient number of power samples. Several codes from these suites could not be used simply because of their short runtimes even with the largest provided inputs. Table 1 lists the number of global kernels each program contains as well as the inputs we used.

1) LonestarGPU

The LonestarGPU suite is a collection of commonly used real-world applications that exhibit irregular behavior.

a. Barnes-Hut *n*-body Simulation (BH): An algorithm that quickly approximates the forces between a set of bodies in lieu of performing precise force calculations.

b. Breadth First Search (L-BFS): Computes the level of each node from a source node in an unweighted graph using a topology-driven approach. In addition to the standard BFS implementation (topology-driven, one node per thread), we also study the atomic variation (topology driven, one node per thread that uses atomics) and the wla variation (topology driven, one flag per node, one node per thread). The wlw variation (data driven, one node per thread) and the wlc variation (data-driven, one edge-per-thread version using Merrill's strategy [25]) were not used because they reduced the active runtime to the point that insufficient power samples could be recorded even with the largest available input.

c. Delaunay Mesh Refinement (DMR): This implementation of the algorithm described by Kulkarni et al. [17] produces a guaranteed quality 2-D Delaunay mesh, which is a Delaunay triangulation with the additional constraint that no angle in the mesh be less than 30 degrees.

Table 1: Program names, number of global kernels (#K), and inputs

Program	#K	Inputs	Program	#K	Inputs
EIP	2	None	SGEMM	1	"small" benchmark input
EP	2	None	STEN	1	"small" benchmark input
NB	1	100k, 250k, and 1m bodies	TPACF	1	"small" benchmark input
SC	3	2^26 elements	BP	2	2^17 elements
BH	9	bodies-timesteps; 10k-10k, 100k-10, 1m-1	R-BFS	2	random graphs; 100k and 1m nodes
L-BFS	5	Roadmaps of Great Lakes Region (2.7m nodes, 7m edges), Western USA (6m nodes, 15m edges), and entire USA (24m nodes, 58m edges)	GE	2	2048 x 2048 matrix
DMR	4	250k, 1m, and 5m node mesh files	MUM	3	100bp and 25bp
MST	7	Roadmaps of Great Lakes Region (2.7m nodes, 7m edges), Western USA (6m nodes, 15m edges), and entire USA (24m nodes, 58m edges)	NN	1	42k data points
PTA	40	vim (small), pine (medium), tshark (large)	NW	2	4096 and 16384 items
SSSP	2	Roadmaps of Great Lakes Region (2.7m nodes, 7m edges), Western USA (6m nodes, 15m edges), and entire USA (24m nodes, 58m edges)	PF	1	row length-column length-pyramid height; 100k-100-20, 200k-200-40
NSP	3	clauses-literals-literals per clause; 16800-4000-3, 42k-10k-3, 42k-10k-5	S-BFS	9	default benchmark input
P-BFS	3	Roadmap of the San Francisco Bay Area (321k nodes, 800k edges)	FFT	2	default benchmark input
CUTCP	1	watbox.s1100.pqr	MF	20	default benchmark input
HISTO	4	image file whose parameters are "--20-4" according to documentation	MD	1	default benchmark input
LBM	1	3000 and 100 timestep inputs	QTC	6	default benchmark input
MRIQ	2	64x64x64 matrix	ST	5	default benchmark input
SAD	3	default input	S2D	1	default benchmark input

d. Minimum Spanning Tree (MST): This benchmark computes a minimum spanning tree in a weighted undirected graph using Boruvka's algorithm and is implemented by successive edge relaxations of the minimum weight edges.

e. Points-to Analysis (PTA): Given a set of points-to constraints, this code computes the points-to information for each pointer in a flow-insensitive, context-insensitive manner implemented in a topology-driven way.

f. Single-Source Shortest Paths (SSSP): Computes the shortest path from a source node to all nodes in a directed graph with non-negative edge weights by using a modified Bellman-Ford algorithm. In addition to the standard SSSP implementation (topology driven, one node per thread), we also used the wln variation (data driven, one node per thread) and the wlc variation (data driven, one edge per thread using Merrill's strategy adapted to SSSP).

g. Survey Propagation (NSP): A heuristic SAT-solver based on Bayesian inference. The algorithm represents the Boolean formula as a factor graph, which is a bipartite graph with variables on one side and clauses on the other.

2) Parboil

Parboil is a set of applications used to study the performance of throughput-computing architectures and compilers.

a. Breadth-First Search (P-BFS): Computes the shortest-path cost from a single source to every other reachable node in a graph of uniform edge weights.

b. Distance-Cutoff Coulombic Potential (CUTCP): Computes the short-range component of the Coulombic potential at each grid point over a 3-D grid containing point charges representing an explicit-water biomolecular model.

c. Saturating Histogram (HISTO): Computes a 2-D saturating histogram with a maximum bin count of 255.

d. Lattice-Boltzmann Method Fluid Dynamics (LBM): A fluid dynamics simulation of an enclosed, lid-driven cavity using the Lattice-Boltzmann Method.

e. Magnetic Resonance Imaging - Q (MRIQ): Computes a matrix Q, representing the scanner configuration for calibration, used in 3-D magnetic resonance image reconstruction algorithms in non-Cartesian space.

f. Sum of Absolute Differences (SAD): Sum of absolute differences kernel, used in MPEG video encoders.

g. General Matrix Multiply (SGEMM): A register-tiled matrix-matrix multiplication, with default column-major layout on matrix A and C, but B is transposed.

h. 3-D Stencil Operation (STEN): An iterative Jacobi stencil operation on a regular 3-D grid.

i. Two Point Angular Correlation Function (TPACF): Statistical analysis of the distribution of astronomical bodies.

3) Rodinia

Rodinia is designed for heterogeneous computing infrastructures with OpenMP, OpenCL, and CUDA implementations. Our study uses the following CUDA codes.

a. Back Propagation (BP): ML algorithm that trains the weights of connecting nodes on a layered neural network.

b. Breadth-First Search (R-BFS): Another GPU implementation of the breadth-first search algorithm, which traverses all the connected components in a graph.

c. Gaussian Elimination (GE): Computes results row by row, solving for all of the variables in a linear system.

d. MUMmerGPU (MUM): A local sequence alignment program that concurrently aligns multiple query sequences against a single reference sequence stored as a suffix tree.

e. Nearest Neighbor (NN): Finds the k-nearest neighbors in an unstructured data set.

f. Needleman-Wunsch (NW): A nonlinear global optimization method for DNA sequence alignment.

g. Pathfinder (PF): Uses dynamic programming to find a path on a 2-D grid with the smallest accumulated weights, where each step of the path moves straight or diagonally.

4) SHOC

The Scalable HeterOgeneous Computing benchmark suite is a collection of programs that are designed to test the performance of heterogeneous systems with multicore processors, graphics processors, reconfigurable processors, etc.

a. Breadth-First Search (S-BFS): Measures the runtime of breadth-first search on an undirected random *k*-way graph.

b. Fast Fourier Transform (FFT): Measures the speed of a single- and double-precision fast Fourier transform that computes the discrete Fourier transform and its inverse.

c. MaxFlops (MF): Measures the maximum throughput for combinations of different floating-point operations.

d. Molecular Dynamics (MD): Measures the performance of an n-body computation (the Lennard-Jones potential from molecular dynamics). The test problem is n atoms distributed at random over a 3-D domain.

e. Quality Threshold Clustering (QTC): Measures the performance of an algorithm designed to be an alternative method for data partitioning.

f. Sort (ST): Measures the performance of a radix sort on unsigned integer key/value pairs.

g. Stencil2D (S2D): Measures the performance of a 2-D, 9-point single-precision stencil computation.

5) CUDA-SDK

The NVIDIA GPU Computing SDK includes dozens of sample codes. We use the following programs in our study.

a. MC_EstimatePiInlineP (EIP): Monte Carlo simulation for the estimation of Pi using a Pseudo-Random Number Generator (PRNG). The inline implementation allows use inside GPU functions/kernels as well as in the host code.

b. MC_EstimatePiP (EP): Monte Carlo simulation for the estimation of Pi using a PRNG. This implementation generates batches of random numbers.

c. N-body (NB): All-pairs *n*-body simulation.

d. Scan (SC): Demonstrates an efficient implementation of a parallel prefix sum, also known as "scan".

B. Evaluation test bed

We measured the GPU's active runtime and energy consumption with the K20Power tool [3], which makes use of the K20's internal power sensor. We chose this approach over using an external sensor because we want our results to be directly applicable to anyone interested in performing power/energy-related GPU experiments and not just to those people with access to an external power sensor. We performed initial experiments on K20c, K20m, K20x, and K40 GPUs, all of which resulted in the same findings after appropriately scaling the absolute measurements, which is why we decided to only discuss the K20c results in this paper.

Our Tesla K20c GPU has 5 GB of global memory and 13 streaming multiprocessors with a total of 2,496 processing elements. It supports six clock frequency settings, of which we evaluate the following three: the "default" configuration, which uses a 705 MHz core speed and a 2.6 GHz memory speed. We found this to be the fastest speed at which the GPU could run long enough for our purposes without throttling itself down to prevent overheating. The "614" configuration, which uses a 614 MHz core speed and a 2.6 GHz memory speed, is the slowest available compute speed at the default memory speed. The "324" configuration uses a 324 MHz core and memory speed. We chose the 324 setting because it is the slowest available frequency configuration. The GPU further supports enabling and disabling ECC protection of the main memory. The "ECC" configuration combines ECC protection with the default clock frequency. All other tested configurations have ECC disabled. Thus, our four configurations are: 1) default, 2) 614, 3) 324, and 4) ECC. Note that we use the same K20c GPU for all measurements.

We repeated some of our measurements on a second K20c GPU to ensure that we obtain the same results, which we did. We were able to record accurate and reliable results from all of our selected programs with the default, 614, and ECC configurations, but we were unable to obtain the same consistency with the 324 configuration. This is because on such a low-frequency setting, the power draw rarely reaches a sufficient level for the GPU's power sensor to switch from the idle 1 Hz to the active 10 Hz sampling rate, which results in too few samples to draw conclusions from. Hence, those results are not reported here. The programs excluded from the 324 results are inline_p, p, P-BFS, cutcp, histo, mri-q, sad, stencil, tpacf-small, backprop, R-BFS, gaussian, mummergpu-25bp, nnlist, nw, pathfinder, S-BFS, fft, maxflops, md, and sort.

We performed each experiment three times and report the median active runtime, energy, and average power. Table 2 shows the maximum and average difference we observed between the highest and the lowest of any set of three measurements for each benchmark suite. We found the variability to be reasonable. Furthermore, while temperature swings can have a noticeable effect on power/energy results, our experiments were performed on a server that is constantly running under full load and is located in a climate-controlled room. Hence, we do not believe that temperature swings introduced any significant perturbations in our measurements. All benchmark codes were compiled with CUDA version 6.0.1 using the default switches prescribed by their authors.

Table 2: Maximum and average measurement variability

	max time	max energy	avg time	avg energy
CUDA SDK	7.1%	7.2%	1.1%	2.3%
LonestarGPU	8.7%	7.1%	0.9%	1.2%
Parboil	3.3%	5.5%	2.0%	2.3%
Rodinia	7.0%	4.5%	1.8%	2.0%
SHOC	7.0%	6.2%	1.4%	2.1%
Overall	8.7%	7.2%	1.4%	2.0%

C. Active runtime

Throughout this paper, we refer to the "active runtime", which is not the total application runtime but rather the time during which the GPU is actively computing, that is, running kernel code. For most of the studied programs, this is a combination of multiple GPU kernels. The K20Power tool defines active runtime as the amount of time the GPU is drawing power above the idle level. Figure 1 illustrates this.

Because of how the GPU draws power and how the built-in power sensor samples, only readings above a certain threshold (the dashed line at 55 W in this example) reflect when the GPU is actually executing kernel code [2]. Measurements below the threshold are either the idle power (less than about 26 W) or the "tail power" due to the driver keeping the GPU active for a while (in case another kernel call is made) before powering it down. Using the active runtime ignores any execution time that may take place on the host CPU, as we are only interested in the energy consumption and power draw of the GPU while it executes. The power threshold is dynamically adjusted for each execution of a particular program to maximize accuracy for different GPU configurations. For example, lower frequency settings require a lower threshold.

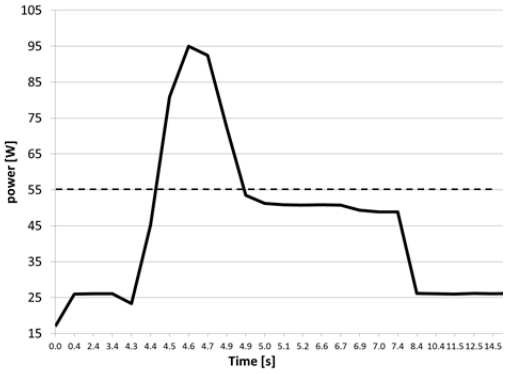

Figure 1: Sample power profile

V. Experimental Results

The following subsections discuss different aspects of our measurements. In each case, we present and analyze the general trends and highlight notable outliers. The most important findings and insights are summarized in Section VI. Detailed results for each individual program are given elsewhere [6].

A. Effects of lowering GPU frequency and enabling ECC

1) Default to 614

Figure 2 shows the relative change in active runtime, energy, and power when switching from default to 614. Values above 1.0 indicate an increase over the default. In the figure, the bars represent the medians, the boxes above and below the bars extend to the first and third quartiles, and the whiskers indicate the maximum and minimum of each suite.

The 614 configuration has a core speed that is about 15% lower than the default, so one might expect to see roughly the same increase in active runtime. However, only a couple of the benchmark medians (CUDA SDK and LonestarGPU) and a few individual codes slow down anywhere near 15%. This is because going from default to 614 only lowers the core frequency but not the memory frequency. Hence, programs showing little to no change are memory bound, including many codes in the Parboil, Rodinia, and SHOC suites. In contrast, most of the tested SDK programs are compute bound. There are also a few codes that show a slight speedup, but many of them are within the margin of error (i.e., the average variability of our measurements).

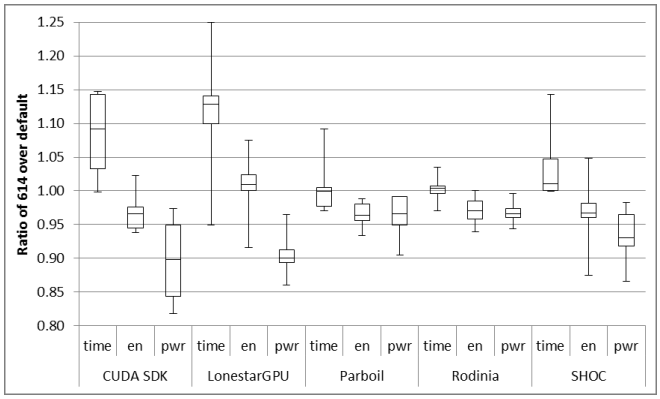

Figure 2: Range of effects on runtime, energy, and power with the 614 configuration relative to the default configuration

Switching to 614 affects the LonestarGPU suite the most. In fact, this suite includes both the worst increase and the best decrease in active runtime as well as the highest median increase. At first glance, this may appear surprising as LonestarGPU only contains programs with irregular memory accesses, which tend to be memory bound, particularly on GPUs where they result in many uncoalesced memory accesses [2]. Hence, lowering the core frequency while keeping the memory frequency constant should not affect the runtime much. However, irregular programs, by definition, exhibit data-dependent runtime behavior, i.e., the amount of parallel work and the load balance can change dynamically. Data-dependent behavior may lead to timing-dependent behavior because it matters when the values are calculated that dependent computations are waiting for. As a consequence, small changes in timing, e.g., due to lowering the core frequency, can have a large effect on the runtime. This is why the active runtime of some LonestarGPU programs changes by more than the clock frequency. Note that this effect can be positive or negative, which explains why LonestarGPU exhibits the widest range in runtime change and includes both the code that is helped the most and the code that is hurt the most by going to 614.

Interestingly, going from default to 614 results in a small beneficial effect on energy for most codes. Even though the programs tend to run longer, the amount of energy the GPU consumes decreases slightly in almost all cases except for LonestarGPU, where the energy consumption is mostly unaffected. Even in the worst cases, the energy does not increase by nearly as much as the active runtime.

This general decrease in energy is the result of a significant reduction in the power draw when switching to 614, which

outweighs the increase in runtime on every program we studied. Even in the worst cases, the power decreases slightly or stays roughly constant, and the median power decrease is between 3% and 10% depending on the benchmark suite. Note that two of the compute-bound CUDA SDK programs experience a power reduction of over 15%, that is, more than the reduction in core frequency. We surmise that this is because, in addition to the frequency, the voltage is also reduced as is commonly done in DVFS. Since power is proportional to the voltage squared, superlinear power reductions are possible.

NB from the CUDA SDK sees the greatest savings in power (22%) for the reasons stated above. It also has one of the largest increases in active runtime (15%) as it is highly compute bound, resulting in a middling decrease in energy (7%). Interestingly, just behind NB for decrease in power is the irregular MST from the LonestarGPU suite. It has the highest increase in active runtime of all the programs across all the benchmarks (25%) and the third highest increase in energy (8%) and saves 16% in power. SHOC's MF boasts the greatest savings in energy (14.3%) and one of the lowest increases in active runtime (1%). It saves 15% in power, making it the third best in terms of power reduction.

2) 614 to 324

Figure 3 is similar to Figure 2 except it shows the relative change in active runtime, energy, and power when switching from 614 to 324. Note, however, that the two figures should not be compared directly as a number of programs did not yield sufficient power samples with the 324 configuration. Hence, Figure 3 is based on fewer programs.

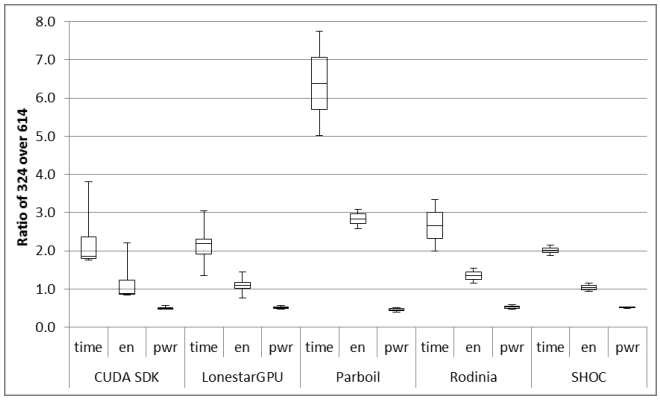

Figure 3: Range of effects on runtime, energy, and power with the 324 configuration relative to the 614 configuration

Switching from the 614 to the 324 configuration not only involves a 1.9x decrease in core frequency but, much more importantly, also an 8x decrease in memory frequency. The cumulative effect is an average increase in active runtime of over 110% across the studied benchmark programs. Except for some irregular programs, where the change in timing affects the load balance and the amount of parallelism, all investigated programs are slowed down by at least a factor of 1.9 as expected.

Going from 614 to 324 increases the energy of two-thirds of the programs, in one case by over 200%. Even in the best case, the energy savings do not exceed 30% and, on average, going from 614 to 324 results in an increase in energy.

This increase in energy can be attributed to the increase in active runtime, which more than outweighs the decrease in power draw. In contrast, on the 614 configuration evaluated in the previous subsection, the active runtime increases less than or roughly in proportion to the decrease in power draw, so the energy consumption decreases or stays roughly the same. This difference in behavior is due to the fact that the 614 configuration only lowers the core frequency whereas the 324 configuration lowers both the core and, in particular, the memory frequency.

According to Figure 3, the Parboil suite experiences by far the largest increase in active runtime and energy. This is misleading, though, as only one of the Parboil programs (LBM) yielded usable results on the 324 configuration. In fact, the two LBM inputs resulted in the two highest increases in runtime and energy out of all the programs we studied. This is a good example of how big an impact altering the memory frequency of the GPU can have on a memory-bound code.

The power decreases quite uniformly to about half the 614 values. Also, it is worthwhile pointing out that the range by which the power changes is much narrower than the range of the energy and in particular the active-runtime changes.

Looking at the individual codes, PTA from the LonestarGPU suite sees the smallest decrease in active runtime and the largest decrease in energy when going from 614 to 324, but because of the small drop in active runtime compared to the other programs, it also has one of the smallest savings in power. LBM from the Parboil suite sees the largest increase in active runtime (7.75x) and energy (2x).

Overall, lowering the core and memory frequencies increases the active runtime of programs and tends to increase the energy. However, it consistently yields lower power levels across all the benchmark programs we have studied.

3) Enabling ECC

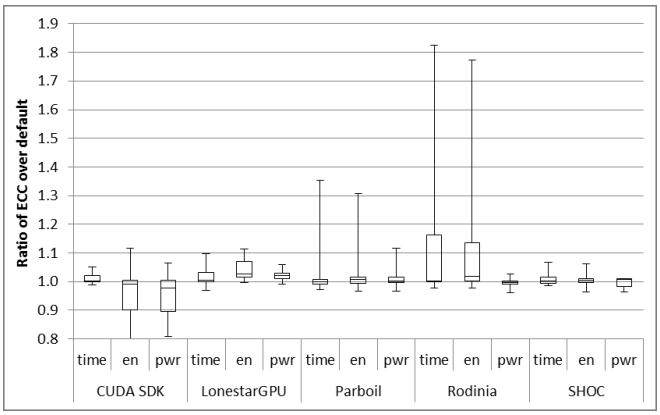

Figure 4: Range of effects on runtime, energy, and power relative to the default configuration when turning on ECC

Figure 4 shows how enabling ECC affects the active runtime, energy, and power relative to the default configura-

tion. Some programs exhibit a very small speedup when turning on ECC, but these speedups are within the margin of error and we therefore do not believe them to be significant.

For all suites, the median active runtimes are the same with and without ECC. However, there are a few programs, most notably in Rodinia, where the active runtime increases drastically. These codes are heavily memory bound. After all, ECC only affects main memory accesses. On our GPU, enabling ECC reduces the available main memory by 12.5% to set aside storage for the ECC information. While we do not know how ECC is implemented (NVIDIA does not publish hardware details about their GPUs), we expect the change in runtime to be within 12.5%, which it is for almost all of the studied programs.

As expected, ECC has a negative effect on the energy consumption. However, since the increase in energy scales closely with the increase in active runtime, ECC primarily increases energy consumption because it slows down main memory accesses and not because the actual ECC operations performed by the hardware consume much energy. Thus, the power is largely the same whether ECC is used or not.

LonestarGPU is the only suite where the energy increase is substantially higher than the increase in active runtime. This is likely a consequence of the many uncoalesced memory accesses, which probably exercise the ECC machinery more than coalesced accesses. As a result, the power also increases.

ECC has a large effect on three of the Rodinia and one of the Parboil programs. As we have no information on how ECC is implemented on our GPU, we can only surmise that it may affect the cache performance, if the ECC bits are copied into the cache, or the memory-controller performance by interrupting otherwise efficient memory access patterns due to intermittent accesses to ECC information.

Looking at individual programs, we find NB from the CUDA SDK to behave unexpectedly. Not only does its active runtime not appear to be affected by ECC much, but it also sees a significant decrease in energy and, consequently, in power. NB is highly regular and compute bound with excellent caching in shared memory, so we did not expect ECC to matter, but to have a positive effect that exceeds the margin of error is surprising. Note that NB's power and energy savings are less pronounced with larger inputs, which result in a higher computation-to-memory-access ratio.

Since ECC protects main memory, its cost is entirely dependent upon how many main memory accesses a program makes. Our results show that using ECC with even slightly memory-bound programs not only results in a slowdown but also in a concomitant increase in energy consumption. However, compute-bound codes are largely unaffected by ECC, as one might expect.

B. Effects of varying program implementation and input

1) Changing the implementation

Table 3 shows how the alternate implementations of L-BFS and SSSP included in the LonestarGPU suite perform when compared to the active runtime, energy, and power of the default implementation of L-BFS. Values below 1.0 indicate that the alternate implementation is better than the default.

The atomic version of L-BFS shows an over 2x reduction in active runtime and energy in all cases. Because the reductions in active runtime and energy scale relatively closely, the power is less affected but still sees a modest decrease (11%-20%), meaning that the atomic version results in improved energy and power efficiency in addition to speedup.

Table 3: Effects of different implementations of L-BFS and SSSP on active runtime [s], energy [J], and power [W]

L-BFS	atomic/default			wla/default		
	time	en	pwr	time	en	pwr
default USA	0.31	0.27	0.85	0.66	0.36	0.55
324 USA	0.29	0.26	0.89	0.39	0.27	0.68
614 USA	0.32	0.27	0.86	0.55	0.33	0.60
ECC USA	0.32	0.27	0.85	0.68	0.36	0.54
SSSP	wlc/default			wln/default		
	time	en	pwr	time	en	pwr
default USA	0.56	0.54	0.97	2.38	2.16	0.91
324 USA	0.70	0.67	0.95	1.92	1.83	0.95
614 USA	0.55	0.54	0.99	2.38	2.21	0.93
ECC USA	0.58	0.57	0.99	2.36	2.18	0.92

The wla version of BFS also yields a decrease in active runtime, though not as much as the atomic version. The energy in the wla version, however, decreases far more than the active runtime. This translates into a much higher power reduction than the atomic version. Thus, wla is the better choice for users concerned with power whereas atomic is better for performance and energy consumption. Hence, different implementations of the same algorithm can benefit energy, power, and performance differently.

We cannot show results for the data-driven wlc version of L-BFS, which uses one edge per thread, nor the data-driven wlw version, which uses one node per thread, because these implementations are so fast that, even on the largest input, the power sensor does not log enough samples to accurately analyze the code's energy and power consumption.

For SSSP, the wlc version shows good improvements in performance, ranging from 42% to an over 2x decrease in active runtime. The energy also improves and scales almost perfectly with the decrease in active runtime, which is why the power is basically flat or decreases slightly. Interestingly, the wln version of SSSP is worse than the default by roughly a factor of two in terms of both active runtime and energy and exhibits only small improvements in power if any, making it an uninteresting implementation.

Again, these results demonstrate that it is possible for different implementations of the same program to not only affect the energy consumption, power draw, and performance but to affect them in different ways. In other words, it is possible to optimize code for these three aspects and the optimizations might differ depending on which aspect is most important.

Table 4 shows the per-vertex and per-edge computation

costs on the largest input of each of the BFS implementations from the different benchmark suites on the default configuration. We use this metric because the same or similar inputs could not be used across all implementations and still produce usable results. For example, the smallest input for L-BFS that would produce sufficient power samples is larger than any of the other implementation's largest inputs.

Table 4: Cross-benchmark BFS comparison, performance per 100k processed vertices (top) and per 100k processed edges (bottom)

	per 100k vertices		
	time	energy	power
L-BFS	0.13	13.61	3.78
P-BFS	1.97	95.78	15.65
R-BFS	3.40	171.35	50.42
S-BFS	341.09	16785.53	4921.14
	per 100k edges		
	time	energy	power
L-BFS	0.05	5.25	1.46
P-BFS	0.76	37.11	6.07
R-BFS	0.34	17.14	5.04
S-BFS	341.43	16802.33	4926.07

LonestarGPU's default BFS implementation (L-BFS) is faster and more energy efficient than the other suite's BFS implementations, taking almost 7x less time and over 3x less energy per edge than the next best implementation (R-BFS). In contrast, S-BFS consumes by far the most energy and runtime, both per vertex and per edge. These results again show that different implementations can yield very different energy and performance behaviors. More importantly, they illustrate that changes in the implementation may help one aspect more than another. For instance, going from R-BFS to L-BFS saves twice as much runtime as it saves energy.

2) Changing the program input

Figure 5 shows how the power draw changes when going from one program input to another on the default configuration. Since the active runtime and energy depend on the input, we chose power as the metric for comparison between different inputs. Most of these programs were run using three inputs, but a few were run with just two. Programs from suites that only include one input each are not included in the figure. Values above 1.0 indicate the power draw got higher.

The results show that power tends to increase when going to larger inputs. The only exceptions are some of the irregular codes, where a change in input can greatly change the program's behavior as discussed earlier. BH, LBM, MUM, NB, NW, NSP, and PTA all see a power increase of over 20% when going from a smaller to a larger input. It is probable that these larger inputs exercise the hardware more, e.g., by accessing more memory locations or utilizing more of the GPU's processing elements, thus raising the power draw.

C. Power efficiency

Figure 6 shows the range of the average power draw of each benchmark suite for different GPU configurations. Note that this figure displays absolute values, not ratios.

There is a relatively large differential between the best and worst cases (~60% to over 3x) in each suite, highlighting that some codes are more power intensive than others. Surprisingly, many of the Parboil, Rodinia, and SHOC programs do not exceed 52 W with any GPU configuration. We believe this is because these programs spend most of their runtime waiting for memory accesses or do not fully occupy the GPU. In contrast, the highly regular and compute-bound SDK codes draw about 100W on average and peak at over 160W.

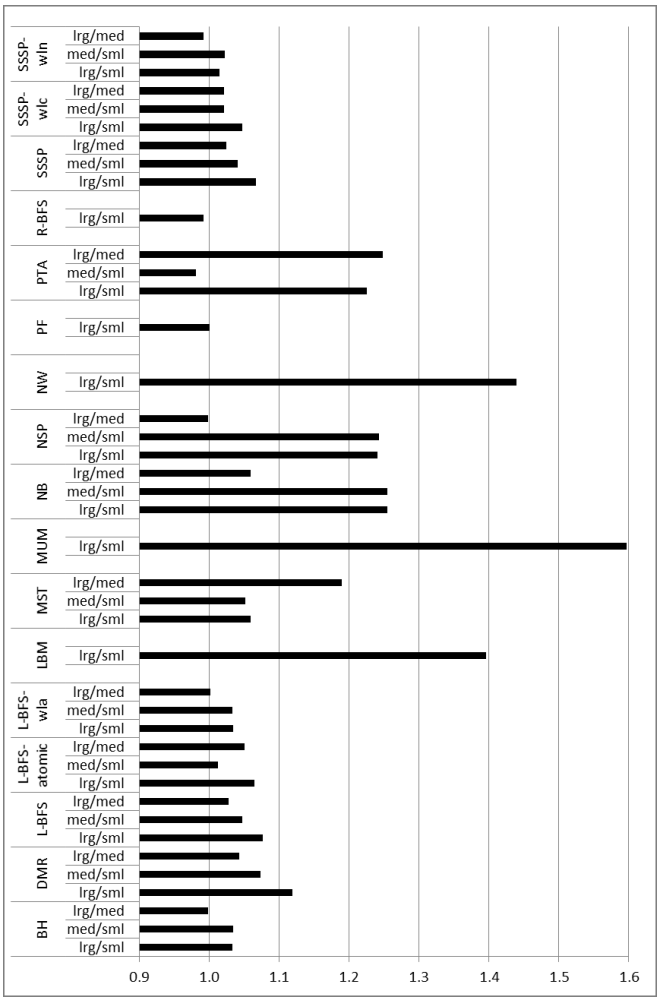

Figure 5: Effects on power when varying the program inputs (values below 1.0 indicate a decrease in power draw)

Most of the irregular LonestarGPU programs draw substantially more power than the regular memory-bound codes. It appears that the irregularity of these programs exercises power-hungry hardware components more, thus increasing the activity per instruction and therefore the power draw.

In all cases, the power consumption decreases when lowering the GPU frequency. In particular the 324 configuration is very effective at reducing power. ECC is generally similar to the default, but in the CUDA SDK and especially the LonestarGPU suite, the range of ECC is smaller than the range with the default configuration.

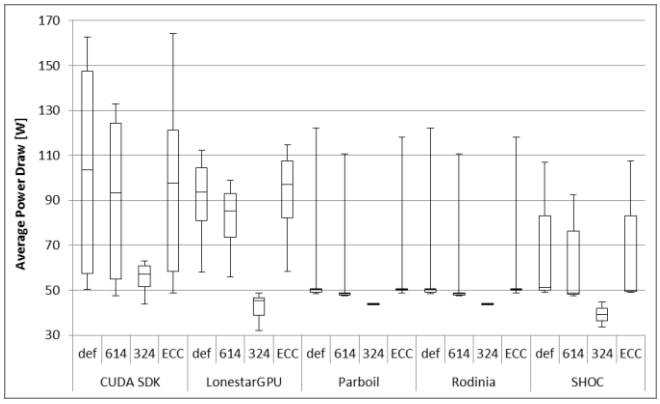

Figure 6: Range of power consumption

VI. Summary and Conclusions

This paper studies the energy consumption, power draw, and runtime of 34 programs from 5 GPGPU benchmark suites. Each program was run at 3 different GPU core frequencies, 2 different GPU memory frequencies, and with ECC enabled or disabled. Where available, the programs were run with alternate implementations and inputs. The goal of this study is to gain general insight into the energy and power behavior of code running on a compute GPU, which can only be achieved by studying a large number of distinct programs.

One important question is whether software has the potential to play an important role in making future accelerators more power and energy efficient. Based on our results, the answer is affirmative. Our measurements demonstrate that changes in the implementation of GPU code can not only drastically improve the performance, as is well known, but also make the energy consumption several times better and reduce the power consumption by over a factor of two. More importantly, different implementations and GPU configurations affect the energy, power, and runtime by different amounts. Hence, there are code transformations that primarily target one aspect and other code transformations that primarily target a different aspect. In other words, it is possible to optimize code and the hardware configuration for energy efficiency and these optimizations may be different from optimizations for performance or power. Of course, some program implementations and GPU configurations are strictly inferior to others and should be avoided.

We made the following main observations when changing the GPU's core and memory frequencies. 1) Different frequencies can impact performance, energy, and power by different amounts. 2) When lowering the core frequency, the energy does not increase significantly to scale with the increase in active runtime. 3) On highly compute-bound codes, power reductions that exceed the reduction in core frequency are possible. 4) Altering the memory frequency of the GPU can have a drastic impact on the runtime and energy of memory-bound code but does not affect compute-bound code by nearly as much. 5) The range by which the power changes is generally smaller than the range by which the energy and active runtime change. 6) Lowering the clock frequency consistently results in lower power levels.

Enabling ECC protection of the main memory has the following effects. 1) Using ECC with even slightly memory-bound programs not only results in a slowdown but also in a concomitant increase in energy consumption. 2) ECC's cost is entirely dependent upon how many main-memory accesses a program makes, so code optimizations that reduce the number of memory accesses are especially useful when ECC is active. Compute-bound codes are mostly unaffected by ECC in terms of energy consumption, power draw, and performance.

Regarding regular and irregular codes, we observed the following. 1) Small changes in timing, e.g., due to a change in frequency, can have a large effect on the runtime of irregular codes, whose behavior is data- and timing-dependent, but not on regular codes. 2) Power tends to increase when going to larger inputs except on some irregular codes where any change in input can greatly alter the program's behavior.

In conclusion, we make the following recommendations for selecting meaningful subsets of GPGPU benchmark programs for anyone interested in performing power and energy studies on GPUs using the built-in power sensor. 1) Use program inputs that result in long runtimes to obtain enough power samples to accurately analyze the energy and power behavior. However, avoid poor implementations that run slowly. 2) Measure a broad spectrum of codes, including memory- and compute-bound programs as well as regular and irregular codes since they exhibit different behaviors. However, avoid topology-driven implementations of irregular graph-traversal codes, as these tend to hide irregularity by doing many unnecessary computations (e.g., L-BFS, SSSP, and NSP). 3) As none of the studied benchmark suites include all of these types of applications, use programs from multiple suites for conducting power and energy studies. Having said that, the codes from Rodinia, Parboil, and SHOC exhibit relatively similar behavior. 4) To make the results comparable, especially with irregular codes or different implementations of the same algorithm, employ metrics like power or energy per processed item. 5) Run irregular codes such as PTA that show input-dependent behavior across several inputs to fully profile the behavior of the code. 6) Repeat experiments at different frequency settings if desired as the findings might change. This is particularly true when lowering the memory frequency of memory-bound codes.

Acknowledgments

This work was supported by the U.S. National Science Foundation under grants 1217231, 1406304, and 1438963, a REP grant from Texas State University, and grants and hardware donations from NVIDIA Corporation. The authors acknowledge the Texas Advanced Computing Center for providing some of the HPC resources used in this study.

References

[1] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling,

R. S. Williams, and K. Yelick. “ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems.” Editor and Study Lead Peter Kogge, 2008.

[2] M. Burtscher, R. Nasre, and K. Pingali. “A Quantitative Study of Irregular Programs on GPUs.” *2012 IEEE International Symposium on Workload Characterization*, pp. 141-151. November 2012.

[3] M. Burtscher, I. Zecena, and Z. Zong. “Measuring GPU Power with the K20 Built-in Sensor.” *Seventh Workshop on General Purpose Processing on Graphics Processing Units*. March 2014.

[4] J. Chen, B. Li, Y. Zhang, L. Peng, and J. Peir. “Statistical GPU power analysis using tree-based methods.” *2011 International Green Computing Conference and Workshops*. July 2011.

[5] J. Choi, M. Dukhan, X. Liu, and R. Vuduc. “Algorithmic Time, Energy, and Power on Candidate HPC Compute Building Blocks.” *Parallel and Distributed Processing Symposium.* May 2014

[6] J. Coplin and M. Burtscher. “Energy, Power, and Performance Characterization of GPGPU Benchmark Programs.” Technical Report TXSTATE-CS-ECL-2016-1. March 2016. http://cs.txstate.edu/~burtscher/papers/tr1ecl16.pdf

[7] CUDA SDK: https://developer.nvidia.com/cuda-toolkit

[8] T. Diop, N. Jerger, and J. Anderson. “Power Modeling for Heterogeneous Processors.” *Seventh Workshop on General Purpose Processing Using GPUs*. March 2014.

[9] W. Feng, X. Feng, and R. Ge. “Green Supercomputing Comes of Age.” *IT Professional*. February 2008.

[10] V. Freeh, F. Pan, N. Kappiah, D. Lowenthal, and R. Springer. “Exploring the Energy-Time Tradeoff in MPI Programs on a Power-Scalable Cluster.” *19th IEEE International Parallel and Distributed Processing Symposium.* March 2005.

[11] R. Ge, R. Vogt, J. Majumder, A. Alam, M. Burtscher, and Z. Zong. “Effects of Dynamic Voltage and Frequency Scaling on a K20 GPU.” *2nd International Workshop on Power-aware Algorithms, Systems, and Architectures*. October 2013.

[12] S. Ghosh, S. Chandrasekaran, and B. Chapman. “Energy Analysis of Parallel Scientific Kernels on Multiple GPUs.” *2012 Symposium on Application Accelerators in High Performance Computing.* July 2012.

[13] GPUWattch Energy Model Manual: http://www.gpgpu-sim.org/gpuwattch/

[14] Y. Jiao, H. Lin, and W. Feng. “Characterizing Performance and Power of GPU Applications with DVFS.” *IEEE/ACM International Conference on Green Computing and Communications*. December 2010.

[15] K. Kandalla, E.P. Mancini, S. Sur, and D.K. Panda. “Designing Power-Aware Collective Communication Algorithms for InfiniBand Clusters.” *39th International Conference on Parallel Processing.* September 2010.

[16] V. Korthikanti and G. Agha. “Towards optimizing energy costs of algorithms for shared memory architectures.” *22nd annual ACM symposium on Parallelism in algorithms and architectures.* June 2010.

[17] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala and L. Paul Chew. “Optimistic Parallelism Requires Abstractions.” *ACM Conference on Programming Languages Design and Implementation,* 211 - 222, June 2007.

[18] Y. Lee and S. Kim. “Empirical Characterization of Power Efficiency for Large Scale Data Processing.” *2015 International Conference on Advanced Computing Technology*. July 2015.

[19] J. Li. “Application-Directed DVFS using Multiple clock domains on Graphics Hardware.” MS Thesis, Worcester Polytechnic Institute. 2008.

[20] J. Li and J. Martínez. “Power-Performance Considerations of Parallel Computing On Chip Multiprocessors.” *ACM Transactions on Architecture and Code Optimization*. December 2005

[21] LonestarGPU: http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu

[22] M. Lorenz, P. Marwedel, T. Dräger, G. Fettweis, and R. Leupers. “Compiler based exploration of DSP energy savings by SIMD operations.” *2004 Asia and South Pacific Design Automation Conference.* January 2004.

[23] Lucas, J., Lal, S., Andersch, M., Alvarez-Mesa, M., Juurlink, B. “How a Single Chip Causes Massive Power Bills GPUSimPow: A GPGPU Power Simulator”. *IEEE International Symposium on Performance Analysis of Systems and Software*. 2013.

[24] X. Ma, M. Rincon, and Z. Deng. “Improving Energy Efficiency of GPU based General-Purpose Scientific Computing through Automated Selection of near Optimal Configurations.” *Technical report UH-CS.* August 2011.

[25] Duane Merrill, Michael Garland, and Andrew Grimshaw. “Scalable GPU Graph Traversal.” *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. pp. 117-128. 2012.

[26] F. Pan, V. Freeh, and D.M. Smith. “Exploring the Energy-Time Tradeoff in High-Performance Computing.” *Parallel and Distributed Processing Symposium.* April 2005.

[27] Parboil: http://impact.crhc.illinois.edu/Parboil/parboil.aspx

[28] Rodinia: http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page

[29] SHOC: https://github.com/vetter/shoc/wiki

[30] J. Sheaffer, K. Skadron, and D.P. Luebke. “Studying Thermal Management for Graphics-Processor Architectures.” *IEEE International Symposium on Performance Analysis of Systems and Software.* March 2005.

[31] B. Subramaniam and W. Feng. “Understanding Power Measurement Implications in the Green500 List.” *Green Computing and Communications, 2010 IEEE/ACM Int’l Conference on & Int’l Conference on Cyber, Physical, and Social Computing*. December 2010.

[32] Wang, G. “Power Analysis and Optimizations for GPU Architecture Using a Power Simulator.” *3rd International Conference on Advanced Computer Theory and Engineering.* 2010.

[33] I. Zecena, M. Burtscher, J. Tongdan, and Z. Ziliang. “Evaluating the Performance and Energy Efficiency of N-Body Codes on Multi-Core CPUs and GPUs.” *2013 IEEE 32nd International Performance Computing and Communications Conference.* December 2013