Effects of Source-Code Optimizations on GPU Performance and Energy Consumption

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 of source-code optimizations on the performance, power draw, and energy consumption of a modern compute GPU. We evaluate 128 versions of two *n*-body codes: a compute-bound regular implementation and a memory-bound irregular implementation. Both programs include six optimizations that can be individually enabled or disabled. We measured the active runtime and the power consumption of each code version on three inputs, various GPU clock frequencies, two arithmetic precisions, and with and without ECC. This paper investigates which optimizations primarily improve energy efficiency, which ones mainly boost performance, and which ones help both aspects. Some optimizations also have the added benefit of reducing the power draw. Our analysis shows that individual and combinations of optimizations can alter the performance and energy consumption of a GPU kernel by up to a factor of five.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming-Parallel Programming

General Terms

Algorithms, Management, Measurement, Performance, Design, Economics, Experimentation.

Keywords

GPU architectures, source-code optimization, power and energy efficiency, performance evaluation.

1. INTRODUCTION

GPU-based accelerators are widely used in supercomputers and are quickly spreading in PCs and even handheld devices as they not only provide high peak performance but also excellent energy efficiency. In HPC environments, large power consumption and the required cooling due to the resulting heat dissipation are major cost factors. Moreover, to reach exascale computing, a 50-fold improvement in performance per watt is needed by some estimates [2]. In all types of handhelds, battery life is a key concern.

GPGPU-8, February 07 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3407-5/15/02…$15.00
http://dx.doi.org/10.1145/2716282.2716292

These are just a few reasons why energy-efficient computing has become an important research area. While many hardware optimizations for reducing power have been proposed or are already in use, software techniques are lagging behind, particularly techniques that target accelerators like GPUs. Most of the published work focuses on changing the clock frequency (and supply voltage) using the DVFS support built into the latest GPUs. In this paper, we go a step further and study the effect of source-code optimizations on the active runtime, energy consumption, and power draw of a GPU.

It is well known that code optimizations can improve GPU performance a great deal. But what about energy or power? Some studies report a one-to-one correspondence between active runtime and energy [7, 9, 11]. But these studies only vary the program inputs, not the code itself. So it is unclear whether there are code optimizations that help energy more than active runtime or vice versa. And how much of a difference can code transformations make anyway? The goal of this paper is to answer these questions and to evaluate whether source-code optimizations have the potential to play an important role in making future GPUs more energy efficient.

To perform this study, we took a compute-bound regular and a memory-bound irregular *n*-body application written in CUDA. We then heavily modified the source code such that, through conditional compilation, 64 versions of each program could be generated by individually enabling or disabling six code optimizations. We measured the active runtime and power consumption of each version on three different inputs and five different configurations, including three clock speed settings, enabling ECC in main memory, and using double-precision instead of single-precision floating-point arithmetic.

This paper makes the following contributions. 1) It presents an extensive set of experiments to gain insight into the energy and performance impact of GPU source-code optimizations. 2) It studies source-code optimizations rather than the effect of compiler transformations. 3) It confirms several commonly held concepts that had yet to be validated through experimentation and points out cases where results are counter-intuitive. 4) It demonstrates that different source-code optimizations can have a very different effect on performance, energy, and power. 5) It shows that optimizations can drastically improve the energy efficiency. 6) It provides a detailed analysis of what types of optimizations tend to help which aspect and why. 7) It shows that optimizations cannot be assessed individually but must be assessed within the context of other present optimizations. 8) It exposes a heretofore unreported asymmetry in performance and energy impact, i.e., optimizing for performance is not the same as optimizing for energy and vice versa.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 provides an overview of the GPU we study. Section 4 describes the evaluation methodology. Section 5 presents and analyzes the measurements. Section 6 summarizes our findings and draws conclusions.

2. RELATED WORK

We are not aware of any other study on the effects of source-code optimizations on GPU energy, power, and active runtime. Even in the better explored CPU domain, there appears to be no study that evaluates the energy implications of all possible combinations of a set of code optimizations.

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 performance overheads, and to balance performance and power savings on a power-aware DVFS-capable cluster system [10]. Pan et al. also use DVFS-based solutions and show that sometimes expending more energy does not result in a large performance benefit [17]. 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 [11]. 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 [7]. There are also papers that highlight the lack of a standardized power measurement methodology in the HPC community for energy-efficient supercomputing [20] or talk about how ignoring power consumption as a design constraint in supercomputing will result in higher operational costs and diminished reliability [6].

Several publications propose and use analytical models to investigate power and energy aspects. Li and Martinez establish an analytical model for looking at parallel efficiency, granularity of parallelism, and voltage/frequency scaling [12]. Lorenz et al. explore compiler-generated SIMD operations and how they affect energy efficiency [15]. They acknowledge the need to optimize both hardware and software to get an energy-efficient system. Some analytical models target GPUs. For example, Chen et al. institute an efficient 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 [16]. Lim et al. propose a new power model for GPUs that is based on empirical data from a GTX 580 [13]. One simulation-based paper on thermal management for GPUs discusses methods for managing power through architecture manipulation (clock gating, fetch gating, dynamic voltage scaling, multiple clock domains, and floorplanning) [18], which is orthogonal to our study.

There are several papers that measure the power consumption on actual GPU hardware. For instance, Gosh et al. explore some common HPC kernels running on a multi-GPU platform and compare their results against multi-core CPUs [9]. Ge et al.'s study investigates the effect of DVFS on the same type of GPU that we are using [8]. A paper by Zecena et al. presents the probably most closely related study to ours. It measures *n*-body codes running on different GPUs and CPUs [20]. However, their study focuses on the effects of varying the CPU thread count and the differences between GPU generations. Like all the other papers mentioned in this section, it does not investigate the effect of source-code optimizations on the energy or power consumption.

3. GPU ARCHITECTURE

This section provides an overview of the architectural characteristics of the Kepler-based Tesla K20c compute GPU we study. It 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.

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/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. Part of the main memory, called constant memory, is reserved and can only be written by the CPU. GPU accesses to constant memory benefit from a special hardware cache.

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. Barrier synchronization between the threads in an SM can take as little as a couple of cycles per warp. The SMs operate largely independently. They can only communicate through global memory (main memory in DRAM). The SMs support special instructions such as voting, where all threads in a warp compute a combined predicate (i.e., a reduction and broadcast operation), and rsqrtf, which quickly computes an approximation of one over square root.

4. METHODOLOGY

4.1 Programs

We evaluate 128 different versions of two *n*-body codes (64 each). The first code, called NB, is regular and has $O(n^2)$ complexity. The second code, called BH, is irregular and has $O(n \log n)$ complexity. Both programs simulate the time evolution of a star cluster under gravitational forces for a given number of time steps. However, the underlying algorithm (see below) and the code base of the two implementations are completely different. *n* denotes the number of stars (aka bodies). Both of these codes have been written in such a way as there is essentially no execution taking place on the CPU.

The direct NB algorithm performs precise force calculations based on the $O(n^2)$ pairs of bodies. Since identical computations have to be performed for all bodies, the implementation is very regular and maps well to GPUs. The force calculations are independent and can be performed in parallel. In each time step, the $O(n^2)$ force calcula-

tion is followed by an O(n) integration where each body's position and velocity are updated based on the computed force. For the values of n we consider, the integration represents less than 1% of the overall execution time and is therefore insignificant. With all optimizations enabled (see below), our implementation outperforms the NB code that is included in the CUDA SDK [5].

The Barnes-Hut (BH) algorithm approximates the forces acting on each body [1]. It recursively partitions the volume around the n bodies into successively smaller cells and records the resulting spatial hierarchy in an octree (the 3D equivalent of a binary tree). Each cell summarizes information about the bodies it contains. For cells that are sufficiently far away from a given body, the BH algorithm only performs one force calculation with the cell instead of one force calculation with each body inside the cell, which lowers the time complexity to O(n log n). However, different parts of the octree have to be traversed to compute the force acting on different bodies, making the control flow and memory-access patterns quite irregular. The force calculation is by far the most time consuming operation in BH, which is why we only consider source-code optimizations that affect this kernel. We use the BH implementation from the LonestarGPU suite [14].

In summary, the NB code is relatively straightforward, has a high arithmetic intensity, regular control flow, and accesses memory in a strided fashion. In contrast, the BH code is quite complex (it repeatedly builds an unbalanced octree and performs various traversals on it), has a low arithmetic intensity, performs mostly pointer-chasing memory accesses, and has data-dependent control flow. Due to its lower time complexity, it is about 33 times faster on a K20c GPU than the NB code when simulating one million stars.

4.2 Source-Code Optimizations

We modified the two programs in a way that makes it possible to individually enable or disable specific optimizations. For NB, we chose the following six code optimizations. 1) *ftz* 'f' is a compiler flag that allows the GPU to flush denormal numbers to zero when executing floating-point operations, which results in faster computations. While strictly speaking not a code optimization, the same effect can be achieved by using appropriate intrinsic functions in the source code. 2) *rsqrt* 'r' uses the CUDA intrinsic "rsqrtf()" to quickly compute one over square root instead of using the slower but slightly more precise "1.0f / sqrtf()" expression. 3) *const* 'c' copies immutable kernel parameters once into the GPU's constant memory rather than passing them every time a kernel is called, i.e., it lowers the calling overhead. 4) *peel* 'p' separates the innermost loop of the force calculation into two consecutive loops, one of which has a known iteration count and can therefore presumably be better optimized by the compiler. The second loop performs the few remaining iterations. 5) *shmem* 's' employs blocking, i.e., it preloads chunks of data into the shared memory, operates exclusively on this data, then moves on to the next chunk. This drastically reduces the number of global memory accesses. 6) *unroll* 'u' uses a pragma to request unrolling of the innermost loop(s). Unrolling often allows the compiler to schedule instructions better and to eliminate redundancies, thus improving performance.

For BH, we selected the following six code optimizations. 1) *vote* 'V' employs thread voting instead of a shared-memory-based code sequence to perform 32-element reductions. 2) *warp* 'w' switches from a thread-based to a warp-based implementation that is much more efficient because it does not suffer from branch divergence and uses less memory as it records certain information on a per warp instead of a per thread basis. 3) *sort* 's' approximately sorts the bodies by spatial distance to minimize the tree prefix that needs to be traversed during the force calculation. 4) *rsqrt* 'r' is identical to its NB counterpart. 5) *ftz* 'f' is also identical to the corresponding NB optimization. 6) *vola* 'v' strategically copies some volatile variables into non-volatile variables and uses those in code regions where it is known (due to lockstep execution of threads in a warp) that no other thread can have updated the value. This optimization serves to reduce memory accesses.

4.3 Evaluation Test Bed

We measured the GPU's active runtime and power consumption with the K20Power tool [3]. 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 three: 1) the "default" configuration, which uses a 705 MHz core speed and a 2.6 GHz memory speed, 2) the "614" configuration, which uses a 614 MHz core speed and a 2.6 GHz memory speed, i.e., the slowest available compute speed at the default memory speed, and 3) the "324" configuration, which uses a 324 MHz core and memory speed, i.e., the slowest available frequency. It should be noted that the K20c only supports two memory frequencies, 2.6 GHz and 324 MHz. We use the NVIDIA Management Library (NVML) to change the GPU settings. Our GPU further supports enabling and disabling ECC protection of the main memory. The "ECC" configuration combines ECC protection with the default clock frequency. On the K20c, enabling ECC has the effect of increasing the number of memory accesses a particular program makes without significantly increasing the amount of computation. All other tested configurations have ECC disabled.

We compiled the CUDA codes with nvcc 5.5 using the -O3 and -arch=sm_35 flags as baseline. We ran the programs with three inputs. The first input is "N10k_10k" for NB, which has 10,000 stars and 10,000 time steps, and "B100k_100" for BH, which has 100,000 stars and 100 time steps. The second input is "N100k_100" for NB, which has 100,000 stars and 100 time steps, and "B1M_10" for BH, which has 1 million stars and 10 time steps. The third input is "N1M_1" for NB, which has 1 million stars and 1 time step, and "B10M_1" for BH, which has 10 million stars and 1 time step. Since the inputs with fewer bodies include more time steps, the running times of the three inputs are fairly similar. We chose these inputs because they yield multi-second active runtimes even in the most optimized cases. The stars' positions and velocities are initialized according to the empirical Plummer model [17], which mimics the density distribution of globular clusters.

We repeated our measurements on a second K20c GPU to make sure that we obtain the same results, which we did. The evaluated programs use single-precision floating-point arithmetic, but we also wrote double-precision versions for comparison. Since CUDA-enabled GPUs require two registers to hold a double value and can only store half as many doubles as floats in shared memory, the "double" configuration has to run with fewer threads than the single-precision codes. Because of this, the single and double precision configurations used in this study are not directly comparable.

We performed each experiment three times and report the median active runtime and average power, from which we compute the energy. The maximal difference we observed between the highest and the lowest of any set of three measurements is 1.0% in energy and 1.8% in active runtime. The average difference is around 0.2% for both energy and active runtime. In other words, the measurement variability is quite low. Hence, we believe any measured change in active runtime or energy that exceeds 2% to be a true change in program behavior rather than measurement noise.

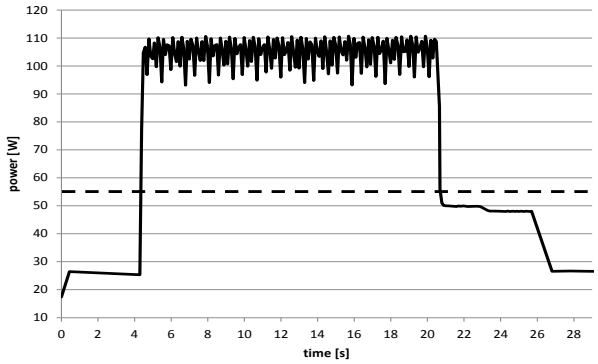

Figure 1: Sample power profile

4.4 Active Runtime

Throughout this study, we refer to the "active runtime", which is not the total application runtime but rather the time during which the GPU is actively executing kernel code. The K20Power tool defines this 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 code [3]. 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 few seconds (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 (which is negligible in all tested codes), as we are solely interested in the energy consumption and power draw of the GPU while it executes the programs. The power threshold is dynamically adjusted for each execution of a particular program to maximize accuracy for different GPU configurations, particularly the low-frequency settings that do not result in a high power draw.

5. 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 6. For reference, Table 7 in the appendix lists the raw data (the median of three experiments) for all studied optimization combinations on the default configuration.

5.1 Input Variability

Table 1 shows the active runtime in seconds, the energy in joules, and the average power in watts for the three inputs and the five configurations when all six optimizations are enabled.

The active runtime is over 10s in all cases. The average power increases for inputs with more bodies since they combine longer-running kernels with fewer kernel calls. For all configurations, the NB power is substantially higher than the BH power. This is because NB is a regular code that utilizes the GPU hardware more effectively than BH, where the hardware often has to wait because of the irregular nature of the code.

The 614 MHz clock frequency is 14.8% lower than the default. This is why the 614 configuration increases the active runtime by about 12% to 15%. At the same time, it decreases the energy by 4% to 5%, but only on the compute-bound NB. Since the memory frequency is not lowered, the energy of BH drops only insignificantly. Nevertheless, because the energy drops (slightly) in both cases while the active runtime increases, the 614 configuration lowers the average power by 15% to 17% on NB and by 12% on BH. Hence, this configuration represents a good power saving strategy but is only useful as an energy saving strategy on compute-bound codes.

The 324 configuration's core speed is 2.18 times lower than that of the default configuration. NB's active runtime is about 2.23 times slower with the 324 configuration, demonstrating that it is highly compute bound and not affected by the ten-fold lower memory frequency. In contrast, BH's active runtime is about 2.52 longer due to its larger dependence on memory speed, even when all code optimizations are enabled to minimize memory accesses. Surprisingly, BH's energy consumption increases by 9% with 324 whereas NB's energy drops by 14%. On both programs, the power drops to well under half of that of the default configuration, again showing that a reduction in GPU frequency is very useful for power savings (and somewhat useful for saving energy on compute-bound codes).

Table 1: Active runtime [s], energy [J], and power [W] when all source-code optimizations are enabled

		default			614			324			ECC			double		
		runtime	energy	power	runtime	energy	power	runtime	energy	power	runtime	energy	power	runtime	energy	power
NB	N10k_10k	16.27	2046	125.7	18.24	1959	107.4	34.98	1808	51.7	15.92	2036	127.9	68.29	8174	119.7
	N100k_100	12.10	1828	151.1	13.88	1745	125.7	26.97	1580	58.6	12.16	1840	151.4	47.33	7137	150.8
	N1m_1	11.56	1801	155.8	13.18	1709	129.7	26.92	1580	58.7	11.56	1822	157.6	49.00	7210	147.1
BH	B100k_100	12.33	1347	109.3	13.96	1346	96.4	30.85	1429	46.3	13.43	1519	113.1	22.52	2409	107.0
	B1m_10	16.15	1966	121.7	18.19	1952	107.3	40.72	2142	52.6	18.33	2277	124.2	26.18	3328	127.1
	B10m_1	19.61	2412	123.0	22.04	2406	109.2	50.06	2641	52.8	22.19	2775	125.1	41.58	4498	108.2

Table 2: Active runtime [s], energy [J], and power [W] with the second input when none and all of the optimizations are enabled

		default			614			324			ECC			double		
		runtime	energy	power	runtime	energy	power	runtime	energy	power	runtime	energy	power	runtime	energy	power
NB	none	63.88	7002	109.6	73.44	6902	94.0	135.85	6250	46.0	64.08	7078	110.5	154.64	17857	115.5
	all	12.10	1828	151.1	13.88	1745	125.7	26.97	1580	58.6	12.16	1840	151.4	47.33	7137	150.8
BH	none	209.09	26266	125.6	237.18	26255	110.7	739.92	34868	47.1	211.81	27301	128.9	304.25	37897	124.6
	all	16.15	1966	121.7	18.19	1952	107.3	40.72	2142	52.6	18.33	2277	124.2	26.18	3328	127.1

The impact of ECC is minimal on NB because it does not access main memory much, but it is high on BH, which accesses memory more often. On BH, the active runtime with ECC is 9% to 13% higher and the energy is 13% to 16% higher, indicating that ECC requires extra energy on top of the additional energy due to the longer active runtime. Hence, there is a small increase in power of up to 4% when ECC is turned on. In summary, ECC negatively impacts active runtime but affects energy more, resulting in a higher power draw.

The double configuration increases the active runtime and energy by almost a factor of four on NB. The computations and memory accesses are half as fast when processing double-precision values, but the higher resource pressure on registers and shared memory necessitates a lower thread count and thus less parallelism, which further decreases performance. In contrast, double "only" increases the active runtime and energy 1.65-fold on BH, which executes a substantial amount of integer code to traverse the octree. This code is not affected by the single- vs. double-precision choice. Despite the large differences in energy and active runtime, the power draw of both NB and BH are hardly affected by the double configuration.

5.2 None vs. All Optimizations

Table 2 shows the measured active runtime and energy consumption as well as the computed average power draw of the five evaluated configurations for the medium inputs when none and all of the optimizations are enabled.

Our first observation is that code optimizations can have a large impact not only on active runtime but also on energy and power. On NB, the energy consumption improves 2.5 to 4 fold due to the optimizations, and the performance improves 3.3 to 5.3 fold. At the same time, the power increases by up to 30% because the hardware is being used more effectively. On BH, the energy improves 11 to 16 fold and the performance 11.5 to 18 fold while the power increases by 10%. Clearly, code optimizations can drastically lower the active runtime and the energy but tend to increase power.

The improvements are substantially lower on NB than on BH. We believe there are two main reasons for this difference. First, NB is a much simpler and shorter code, making it easier to implement efficiently without having to resort to highly complex optimizations. Second, NB is a regular code with few data dependencies, enabling the compiler to generate quite efficient code even in the base case. (Note that we always compile with -O3.) As a consequence, the additional code optimizations we study provide less benefit.

Focusing on the default configuration, we find that the optimizations help the active runtime of NB much more than the energy consumption, which is why the power increases greatly. On BH, the situation is different. Here, the optimizations reduce the energy consumption a little more than the active runtime, resulting in a slight decrease in power draw.

Comparing the default and the 614 configurations, we find that the 15% drop in core speed yields a 15% active runtime increase on NB, as discussed above. The active runtime of BH only increases by 13% because the memory accesses are not slowed down. Hence, the drop in power draw is lower for BH than for NB. The 614 configuration consumes a little less energy than the default configuration. The reduction is within the margin of error for BH but the 5% drop with all optimizations enabled on NB is significant. Overall, the 614 configuration primarily serves to lower the power draw. Due to a concomitant increase in active runtime, it does not affect the energy much. The benefit of all the various optimizations is fairly similar for the default and 614 configurations.

The 324 configuration behaves quite differently because it not only reduces the core frequency by a factor of two but also lowers the memory frequency by a factor of eight relative to the 614 configuration. NB's increase in active runtime is in line with the two-fold drop in core frequency because it is compute bound whereas BH's increase in active runtime is much larger due to its greater dependence on memory speed. As a consequence, there is a decrease in energy consumption on NB but a substantial increase on BH when comparing the 324 to the 614 configuration. Since the 324 configuration affects the active runtime much more than the energy, it results in a large decrease in power draw. Reducing power is the primary strength of the 324 configuration, which is otherwise not very useful and can greatly increase the energy consumption of memory-bound codes. Nevertheless, the code optimizations are just as effective on it as they are on the other configurations. In fact, on BH, they are more effective when using the 324 configuration because the optimizations eliminate some of the now extremely costly memory accesses.

Looking at the ECC configuration, we find that the energy, active runtime, and power numbers are almost identical to the default for NB. Since NB has excellent locality, which translates into many cache/shared-memory hits and good coalescing, even the version without our optimizations performs relatively few main memory accesses, which is why it is hardly affected by ECC. BH has less locality and accesses memory more frequently, which explains why it is affected more. Its active runtime and energy become 13% and 16% worse, respectively, when ECC is turned on and all optimizations are used. However, the active runtime and energy are only 1% and 4% worse with ECC when none of the optimizations are enabled. The reason for this difference is that the optimizations improve the computation more than the memory accesses, thus making the optimized code, relatively speaking, more memory bound.

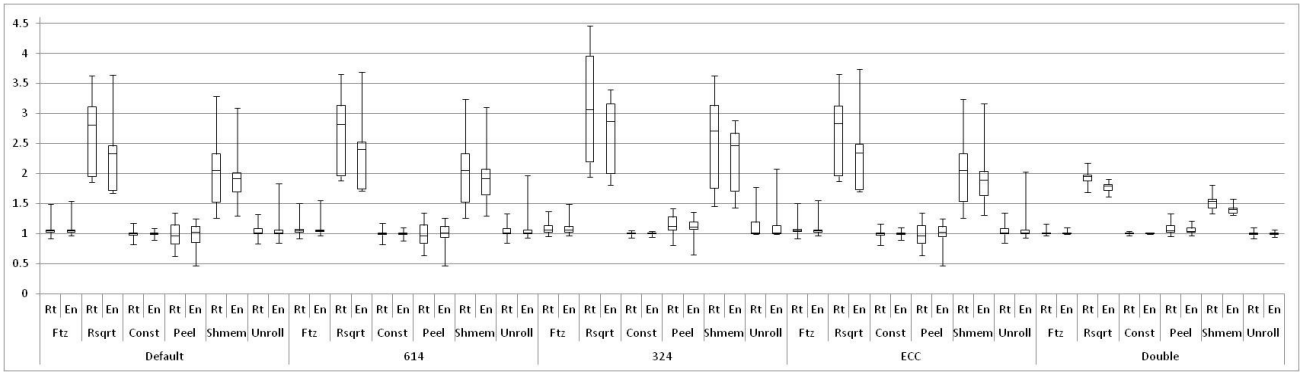

Figure 2a: Improvement range of the active runtime (Rt) and energy (En) on the second input for NB

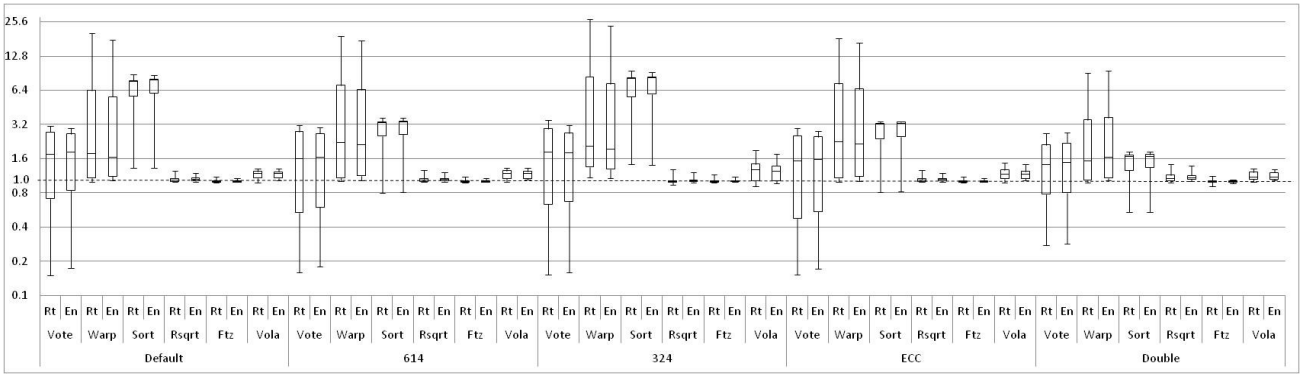

Figure 2b: Improvement range of the active runtime (Rt) and energy (En) on the second input for BH ($\log_2$ scale)

Comparing the double configuration to the default, we observe that the active runtime and energy are 1.5 to 3.9 times worse, but the power is almost unchanged. Clearly, the double-precision code is substantially slower. While still highly effective (a factor of 2.5 to 11.6 improvement), the benefit of the code optimizations is lower for the double configuration. This is because the double-precision code is more memory bound than the single-precision code and, as mentioned above, the optimizations improve the computation more than the memory accesses.

5.3 Effectiveness of Optimizations

Figures 2a and 2b show the range of the effect of each optimization on the active runtime and the energy when using the second input (N100k_100 or B1M_1). The 64 versions of each program contain 32 instances that do not and 32 that do include any given optimization. The presented data shows the maximum (top whisker), minimum (bottom whisker), and the median (line between the two boxes) change when adding a specific optimization to the 32 versions that do not already include it. The boxes represent the first and third quartiles, respectively. Points below 1.0 indicate a slowdown or increase in energy consumption. It is clear from these figures that the effect of an optimization can depend greatly on what other optimizations are present.

The most effective optimization on NB is *rsqrt*. Using this special intrinsic improves the energy and particularly the active runtime because it targets the slowest and most complex operation in the innermost loop. Since *rsqrt* helps the active runtime more than the energy, it increases the power substantially. The other very effective optimization is *shmem*, i.e., to use tiling in shared memory. It, too, improves the active runtime more than the energy, leading to an increase in power.

The impact of the remaining four optimizations is much smaller. *ftz* helps both energy and active runtime as it speeds up the processing of the many floating-point instructions. It is largely power neutral since it improves energy and active runtime about equally. Except with the double configuration, *unroll* helps active runtime a little and energy a lot, thus lowering the power quite a bit. We are not sure why unrolling helps energy more. *const* hurts the energy consumption and the active runtime, but it is close to the margin of error. After all, this optimization only affects the infrequent kernel launches. *peel* hurts performance and energy, except in the double-precision code, but does not change the power much. Clearly, there are optimizations that help active runtime more (e.g., *rsqrt*) while others help energy more (e.g., *unroll*). Some optimizations help both active runtime and energy equally (e.g., *ftz*).

Most of the optimization benefits are quite similar across the different configurations. Notable 324 exceptions are *peel*, which hurts significantly more than in the other configurations, and *ftz* and *rsqrt*, which are more effective on 324. *shmem* and *unroll* are also more effective. Interestingly, the double configuration exhibits almost exactly the opposite behavior. *peel* is more effective on it than on the other configurations whereas *ftz*, *rsqrt*, *shmem*, and *unroll* are substantially less effective.

The findings for BH are similar. There are also two optimizations that help a great deal. *warp*, the most effective optimization, improves energy a little less than active runtime, thus increasing the

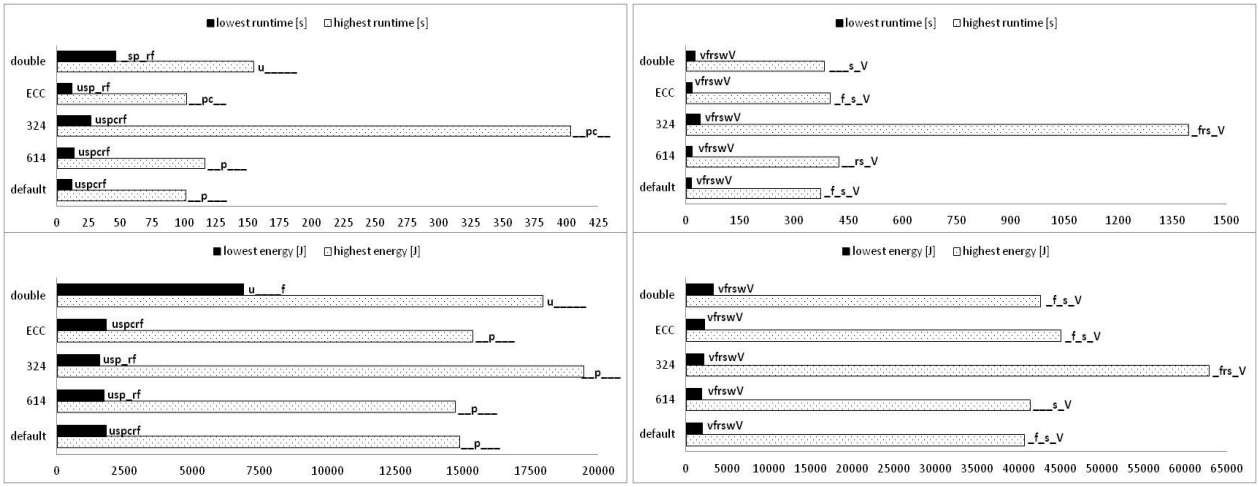

Figure 3: Sets of NB (left) and BH (right) optimizations that yield the highest and lowest runtime and energy on the second input

power draw. *sort*, the second most effective optimization, helps energy a little more than active runtime. *vola* and *rsqrt* also help both aspects but are much less effective and power neutral. We are surprised by the effectiveness of *vola*. Apparently, it manages to reduce memory accesses significantly and therefore makes BH less memory bound. *vote* helps energy substantially more than active runtime. Hence, this optimization is particularly useful for reducing power. Interestingly, on some configurations, it hurts both energy and active runtime, on others only active runtime, and on yet others it helps both aspects. *ftz* is within the margin of error but seems to help a little. The reason why both *rsqrt* and *ftz* are much less effective on BH than on NB is because BH executes many integer instructions, which do not benefit from these optimizations.

Investigating the individual configurations, we again find that there is no significant difference in optimization effectiveness between default and 614 as well as between default and ECC on NB. For 324 on NB, *const* is unchanged, *peel* is less effective, and the other optimizations are more effective than with the default configuration. On BH, *warp* is much more effective with the 324 configuration and *vola* quite a bit more. This is because both optimizations reduce the number of memory accesses. *vote* is slightly less effective and the remaining optimizations' benefits are unchanged. For ECC on BH, *vote* is less effective, but the other optimizations are just as effective as they are with the default configuration. Once again, the double configuration behaves quite differently. Most notably, *warp* is much less effective and *vola* somewhat less. However, *rsqrt* and especially *vote* are more effective. Note that in the double-precision code, the rsqrtf() intrinsic is followed by two Newton-Raphson steps to obtain double precision, which is apparently more efficient, both in terms of active runtime and energy, than performing a double-precision square root and division. Thread voting helps because it frees up shared memory that can then be used for other purposes. For NB, the double configuration results in *peel* being much more effective and all other optimizations except *const* being much less effective than using the default configuration. In particular, *rsqrt* and *shmem* are substantially less effective; the former because of the Newton-Raphson overhead, which is substantial in the tight inner loop of NB and the latter because only half as many double values fit into the shared memory. Overall, the power is not much different for the double configuration with the exception of *unroll,* which does not lower the power draw in the double-precision code.

These results illustrate that the effect of a particular optimization is not always constant but can change depending on the regular or irregular nature of the code, the core and memory frequencies, the chosen floating-point precision, and the presence or absence of other optimizations.

5.4 Lowest and Highest Settings

Figure 3 displays the sets of optimizations that result in the highest and lowest active runtime and energy on the second input. In each case, a six-character string shows which optimizations are present (see Subsection 4.2 for which letter represents which optimization). The characters are always listed in the same order. An underscore indicates the absence of the corresponding optimization.

Table 3: Base setting and added optimization that yields the most significant positive/negative impact on the second input. A '+' indicates an increase and a '-' a decrease; "rt" stands for active runtime and "en" for energy

		default			614			324			ECC			double		
			Impr. factor			Impr. factor			Impr. factor			Impr. factor			Impr. factor	
		setting & opt	time	ener.	setting & opt	time	ener.	setting & opt	time	ener.	setting & opt	time	ener.	setting & opt	time	ener.
NB	rt+ en+	_____ & peel	0.63	0.47	_____ & peel	0.63	0.47	_____ & peel	0.34	0.32	_____ & peel	0.63	0.46	__pcr__ & unrol	0.86	0.91
	rt+ en-	__pc_f & unrol	0.86	1.25	__pc_f & unrol	0.86	1.24	__pc_f & unrol	0.97	1.26	__pc_f & unrol	0.86	1.28	us__rf & const	1.00	1.00
	rt- en+	_sp_rf & const	1.01	1.00	_sp_rf & const	1.01	1.00	_sp_rf & const	1.00	0.99	_sp_rf & const	1.01	1.00	us_cr_ & peel	1.00	0.97
	rt- en-	__p___ & rsqrt	2.97	3.65	__p___ & rsqrt	2.97	3.69	__pc__ & rsqrt	5.33	5.29	__p___ & rsqrt	2.98	3.74	__p___ & rsqrt	2.27	1.90
BH	rt+ en+	_frs__ & vote	0.16	0.18	_frs__ & vote	0.16	0.18	_frs__ & vote	0.15	0.16	_frs__ & vote	0.15	0.17	_frs__ & vote	0.22	0.24
	rt+ en-	___wV & vola	0.99	1.04	___wV & vola	1.00	1.05	v_rs__ & ftz	1.00	1.01	___wV & vola	0.99	1.04	_____ & vote	0.96	1.07
	rt- en+	n/a	n/a	n/a	v__swV & ftz	1.00	1.00	____w_ & ftz	1.00	1.00	____s__ & ftz	1.00	1.00	_____V & ftz	1.00	1.00
	rt- en-	_frs_V & warp	19.06	17.04	_frs_V & warp	19.28	17.48	_frs_V & warp	27.47	23.99	_frs_V & warp	18.31	16.64	_frs_V & warp	13.03	11.40

Interestingly, the worst performance and highest energy are obtained when some optimizations are enabled, showing that it is possible for "optimizations" to hurt rather than help. In the worst case, which is NB 324, the energy consumption increases more than 3 fold when going from none of the optimizations to enabling *peel*. For all tested configurations, the best performance and the lowest energy consumption on BH are always obtained when all optimizations are enabled. This is also mostly the case for NB. Comparing the highest to the lowest values, we find that the worst and best configurations differ by over a factor of 12 (energy) and 14 (active runtime) on NB and by over a factor of 29 (energy) and 34 (active runtime) on BH. This highlights how large an effect code transformations can have on performance and energy consumption. The effect on the power is modest in comparison. It changes by up to 23% on BH and up to 60% on NB.

Looking at individual optimizations, we find that, except for the worst case with the double configuration, *peel* is always included in the best and the worst settings in NB. Clearly, *peel* is bad when used by itself (as noted in Subsection 5.3) or in combination with the *const* optimization. However, *peel* becomes beneficial when grouped with some other optimizations, demonstrating that the effect of an optimization cannot always be assessed in isolation but may depend on the context. To reach the lowest power on NB, *unroll*, *peel*, and *const* need to be enabled together.

In case of BH, we find *sort* and *vote* to dominate the worst active runtime and energy settings. *sort* in the absence of *warp* does not help anything. *vote* generally hurts energy and often active runtime (cf. Subsection 5.3). Surprisingly, *ftz* is often included in the worst performance setting for BH. We expected *ftz* to never increase the active runtime. However, *ftz* also appears in all the best settings, showing that it does help in the presence of other optimizations.

Additionally, the best and worst settings are similar across the different configurations, illustrating that optimizations tend to behave consistently with respect to each other.

5.5 Energy Efficiency vs. Performance

Table 3 shows, for the second input, the largest impact that adding a single optimization makes. We consider four scenarios, maximally hurting both active runtime and energy, hurting active runtime but improving energy, improving active runtime but hurting energy, and improving both active runtime and energy.

With one exception, every configuration has examples of all four scenarios. The examples of decreasing the active runtime while increasing the energy consumption are within the margin of error and therefore probably not meaningful. However, the other three scenarios have significant examples. Excluding the double configuration for the moment, enabling just the *peel* optimization on NB increases the energy consumption by more than a factor of two and the active runtime nearly as much (as noted in the previous subsection). Clearly, this "optimization" is a very bad choice by itself. However, adding *rsqrt* to the *peel* optimization improves the active runtime and energy by a factor of three to five, making it the most effective addition of an optimization we have observed. Perhaps the most interesting case is adding *unroll* to *peel*, *const*, and *ftz*. It lowers the energy consumption by about 25% even though it increases the active runtime substantially. This example shows that performance and energy optimization are not always the same thing.

BH has similar examples. Adding *vote* to *ftz*, *rsqrt*, and *sort* greatly increases the active runtime and the energy consumption, making it the worst example of adding an optimization. In contrast, adding *warp* to *ftz*, *rsqrt*, and *sort* greatly reduces both active runtime and energy, making it the most effective addition of a single optimization we have observed. Again, decreasing the active runtime while increasing the energy is within the margin of error (or no such case exists). Finally, there are several examples of lowering the energy by a few percent while increasing the active runtime marginally. While not as pronounced as with NB, these examples again show that there are cases that only help energy but not active runtime.

The base settings and the optimizations added that result in significant changes in energy and active runtime are consistent across the five configurations. Only the double configuration on NB differs substantially. However, even in this case, the effects of the optimizations are large. In fact, this configuration exhibits the most pronounced example of adding an optimization that lowers the active runtime (albeit within the margin of error) while increasing the energy consumption (by three percent).

Table 4 shows similar results but varies the input (on the default configuration) instead of keeping the input fixed and varying the configuration. Again, the results are consistent for all cases that are not close to the margin of error. In particular, the settings yielding the largest effect are mostly the same. The resulting improvement factors vary a little, though, especially for the first input. Interestingly, the first input benefit more from these optimizations.

Table 4: Base setting and added optimization that yields the greatest positive/negative impact on the default configuration

		first			second			third		
			Impr. factor			Impr. factor			Impr. factor	
		setting & opt	time	ener.	setting & opt	time	ener.	setting & opt	time	ener.
NB	rt+ en+	`& peel`	0.81	0.64	`& peel`	0.63	0.47	`& peel`	0.63	0.47
	rt+ en-	`cr & peel`	0.99	1.01	`pc_f & unrol`	0.86	1.25	`pc_f & unrol`	0.84	1.16
	rt- en+	`& const`	1.00	0.99	`sp_rf & const`	1.01	1.00	`u & const`	1.00	1.00
	rt- en-	`uspc_f & rsqrt`	4.46	3.13	`p & rsqrt`	2.97	3.65	`p & rsqrt`	2.95	3.65
BH	rt+ en+	`frs & vote`	0.21	0.22	`frs & vote`	0.16	0.18	`frs & vote`	0.15	0.17
	rt+ en-	`v rsw & ftz`	1.00	1.01	`wV & vola`	0.99	1.04	`f & vote`	0.92	1.09
	rt- en+	`v V & ftz`	1.00	1.00	`n/a`	n/a	n/a	`n/a`	n/a	n/a
	rt- en-	`frs_V & warp`	14.42	14.52	`frs_V & warp`	19.06	17.04	`frs_V & warp`	20.32	17.70

5.6 Most Biased Optimizations

Whereas the previous subsection investigates cases where an optimization helps one aspect and/or hurts another, this subsection studies optimizations where the difference between how much they improve energy versus active runtime is maximal. Table 5 provides the results for the different configurations and Table 6 for the different inputs. The notation is the same as before.

On NB, we find that adding *unroll* to *peel* and *const* improves energy by 18% to 56% more than active runtime except for double, where there is no strong example. In contrast, adding *rsqrt* to *unroll*, *shmem*, *peel* and *const* improves active runtime by 31% to 46% more than energy, again with the exception of the double configuration, where the same optimization but on a different base setting yields 20% more benefit in active runtime than in energy.

Table 5: Base setting and added optimization that yields the most biased impact on energy over active runtime and vice versa on the second input

		default			614			324			ECC			double		
			Impr. factor			Impr. factor			Impr. factor			Impr. factor			Impr. factor	
		setting & opt	time	ener.	setting & opt	time	ener.	setting & opt	time	ener.	setting & opt	time	ener.	setting & opt	time	ener.
NB	max en over rt	__pc__ & unrol	1.19	1.82	__pc__ & unrol	1.19	1.80	__pc_f & unrol	0.97	1.26	__pc__ & unrol	1.19	1.86	_sp_r_ & ftz	1.01	1.02
	max rt over en	uspc__ & rsqrt	3.65	2.53	uspc__ & rsqrt	3.65	2.61	uspc__ & rsqrt	3.50	2.66	uspc__ & rsqrt	3.66	2.52	__p___ & rsqrt	2.27	1.90
BH	max en over rt	____wV & vola	0.99	1.04	__rs__ & warp	1.11	1.18	____wV & sort	3.01	3.16	_frs__ & warp	1.10	1.16	_______ & vote	0.96	1.07
	max rt over en	_____V & warp	4.36	3.81	v_rs_V & warp	18.13	16.13	_f___V & warp	7.56	6.25	_____V & warp	4.99	4.29	vfrs_V & warp	11.59	10.13

On BH, there is no consistent setting or optimization that yields the highest benefit in energy over active runtime. Nevertheless, some code optimizations help energy between 5% and 18% more than active runtime. The best optimization for helping active runtime more than energy is *warp*, but the base setting differs for the different configurations. It improves active runtime between 6% and 21% more than energy. Again, these results highlight that optimizations do not necessarily affect active runtime and energy in the same way. Rather, some source-code optimizations tend to improve one aspect substantially more than another.

Table 6: Base setting and added optimization that yields the most biased energy over active runtime and vice versa

		first			second			third		
			Impr. factor			Impr. factor			Impr. factor	
		setting & opt	time	ener.	setting & opt	time	ener.	setting & opt	time	ener.
NB	max en	__pc__ & unrol	1.65	1.95	__pc__ & unrol	1.19	1.82	__pc__ & unrol	1.18	1.70
	max rt	uspc__ & rsqrt	4.35	3.05	uspc__ & rsqrt	3.65	2.53	uspc__ & rsqrt	3.63	2.49
BH	max en	_f_____ & warp	1.14	1.22	____wV & vola	0.99	1.04	_f_____ & vote	0.92	1.09
	max rt	v___wV & rsqrt	1.28	1.21	_____V & warp	4.36	3.81	_____V & warp	3.19	2.67

6. SUMMARY AND CONCLUSIONS

This paper studies 128 versions of two different *n*-body simulations running on five GPU configurations using three inputs. For each program, version, configuration, and input combination, we measure the active runtime and power on a compute GPU and calculate the energy consumption. We would have liked to study more programs in this study, but writing conditionally compliable GPU code for so many optimizations is very difficult, error prone, and time intensive. We consider this work an initial study and more programs should be considered in future work.

While we cannot draw generalizations from two programs, this study already provides several interesting results. Some of the key takeaway points are that source-code optimizations tend to increase the power draw, that lowering the clock frequency is a good power saving strategy but not useful as an energy saving strategy, that enabling ECC negatively impacts active runtime and especially energy, that double-precision GPU code behaves quite differently from single-precision code, that optimizations can have a large impact on energy and power, that the effect of an optimization cannot always be assessed in isolation but may depend on the presence of other optimizations, and that source-code optimizations do not necessarily affect active runtime and energy in the same way.

Regarding the average power, we found that our regular, compute bound code draws substantially more power than the irregular, somewhat memory bound code for all configurations. As code optimizations generally increase power, the lowest power tends to be achieved when optimizations are disabled. Overall, our source-code optimizations change the power draw by up to 60%.

Switching from single- to double-precision arithmetic has little effect on the power but drastically increases both the active runtime and the energy consumption of the GPU. The benefit of code optimizations tends to be lower for the double-precision version of a program. However, some optimizations are more effective on double-precision code. Emulating a double-precision division and square root using the rsqrtf intrinsic followed by two Newton-Raphson steps to obtain double precision is more efficient, both in terms of active runtime and energy, than executing true double-precision square root and division instructions.

We have observed improvements by a factor of five in both active runtime and energy consumption due to source-code optimization. Often, the worst performance, energy, and power are obtained when some optimizations are enabled, showing that it is possible for "optimizations" to hurt rather than help. Comparing the highest to the lowest measurements, we find that the worst and best configurations differ by over a factor of 29 in energy and 34 in active runtime. Some optimizations hurt when used by themselves but can become beneficial when grouped with other optimizations.

We have identified several examples where optimizations lower the energy while increasing the active runtime, showing that there are optimizations that only help energy but not active runtime. In one case, the improvement in energy is 56% higher than the improvement in active runtime. In another case, the active runtime improvement is 46% higher than the energy improvement.

Our results demonstrate that programmers can optimize their source code for energy (or power), that such optimizations may be different from optimizations for performance, and that optimizations can make a large difference. Clearly, source-code optimizations have the potential to play an important role in making accelerators more energy efficient.

7. ACKNOWLEDGMENTS

The work reported in this paper is supported by the U.S. National Science Foundation under Grants 1141022, 1217231, 1406304, and 1438963, a REP grant from Texas State University as well as a gift and equipment donations from NVIDIA Corporation.

8. REFERENCES

[1] J. Barnes and P. Hut. "A Hierarchical O(N log N) Force-Calculation Algorithm." *Nature*, vol. 324. 1986.

[2] 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 & Study Lead Peter Kogge, 2008.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] J. Li and J. Martínez. "Power-performance considerations of parallel computing on chip multiprocessors." *ACM Transactions on Architecture and Code Optimization*. December 2005.

[13] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung. "Power Modeling for GPU Architectures Using McPAT." ACM Transactions on Design Automation of Electronic Systems, 19:3, Article 26. June 2014.

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

[15] 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.

[16] 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.

[17] F. Pan, V. Freeh, and D.M. Smith. "Exploring the energy-time tradeoff in high-performance computing." *Parallel and Distributed Processing Symposium.* April 2005.

[18] H. C. Plummer. "On the Problem of Distribution in Globular Star Clusters." *Mon. Not. R. Astron. Soc.*, 71:460. 1911.

[19] 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.

[20] 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.

[21] 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.

APPENDIX

Table 7: Raw data (the median of three experiments) for all studied optimization combinations on the default configuration

NB	N10k_10k			N100k_100			N1m_1			BH	B100k_100			B1m_10			B10m_1		
Setting	Runtime	Energy	Power	Runtime	Energy	Power	Runtime	Energy	Power	Setting	Runtime	Energy	Power	Runtime	Energy	Power	Runtime	Energy	Power
_____	149.10	12136	81.4	63.88	7002	109.6	60.56	6782	112.0	_____	79.11	9558	120.8	209.09	26266	125.6	587.20	74770	127.3
____f	127.60	10969	86.0	59.91	6860	114.5	56.92	6687	117.5	____v	118.62	13222	111.5	311.11	34002	109.3	634.88	68411	107.8
___r_	67.94	6287	92.5	32.66	4139	126.7	31.05	4037	130.0	___w_	69.65	7882	113.2	183.35	22245	121.3	516.88	63586	123.0
___rf	65.69	6108	93.0	31.85	3982	125.0	30.29	3887	128.3	___wv	25.69	2911	113.3	71.33	8932	125.2	199.13	25593	128.5
__c__	148.80	12247	82.3	63.78	7022	110.1	60.47	6795	112.4	__s__	47.05	5419	115.2	62.51	7767	124.3	75.24	9406	125.0
__c_f	128.45	10994	85.6	59.93	6828	113.9	56.90	6651	116.9	__s_v	218.75	24042	109.9	373.77	40620	108.7	478.23	51416	107.5
__cr_	67.90	6292	92.7	32.67	4141	126.7	31.04	4036	130.0	__sw_	42.84	4682	109.3	56.32	6709	119.1	67.96	8163	120.1
__crf	65.63	6104	93.0	31.88	3982	124.9	30.31	3886	128.2	__swv	18.16	1940	106.8	23.22	2774	119.5	27.76	3364	121.2
__p___	184.75	18839	102.0	101.27	14877	146.9	96.16	14529	151.1	__r___	78.13	9393	120.2	203.42	25479	125.3	568.69	72227	127.0
__p__f	135.71	12688	93.5	67.44	9632	142.8	64.35	9438	146.7	__r__v	118.40	13160	111.1	311.47	34003	109.2	637.61	68512	107.5
__p_r_	68.78	6227	90.5	34.06	4080	119.8	32.58	3983	122.3	__r_w_	68.83	7787	113.1	179.26	21714	121.1	503.51	61961	123.1
__p_rf	58.88	5479	93.1	33.08	3870	117.0	31.49	3783	120.2	__r_wv	25.19	2818	111.8	71.02	8674	122.1	199.39	24897	124.9
__pc__	184.66	18837	102.0	100.95	14861	147.2	95.91	14510	151.3	__rs__	45.07	5199	115.4	60.38	7490	124.0	72.67	9060	124.7
__pc_f	136.09	12716	93.4	68.25	9685	141.9	64.92	9476	146.0	__rs_v	218.64	23919	109.4	373.98	40506	108.3	477.48	51156	107.1
__pcr_	68.77	6230	90.6	34.09	4080	119.7	32.59	3983	122.2	__rsw_	40.79	4429	108.6	54.02	6375	118.0	65.51	7797	119.0
__pcrf	58.84	5478	93.1	33.09	3867	116.8	31.48	3784	120.2	__rswv	15.22	1670	109.7	19.75	2417	122.4	23.85	2955	123.9
_s____	80.08	7153	89.3	49.87	5378	107.8	47.52	5243	110.3	_f____	79.11	9577	121.1	209.16	26292	125.7	588.68	74895	127.2
_s___f	80.09	7155	89.3	47.49	5237	110.3	45.26	5111	112.9	_f___v	118.94	13255	111.4	311.40	34060	109.4	637.49	68600	107.6
_s__r_	23.47	2692	114.7	16.14	2291	141.9	15.38	2251	146.3	_f__w_	69.53	7871	113.2	182.86	22202	121.4	514.81	63581	123.5
_s__rf	20.88	2390	114.5	13.95	2007	143.9	13.31	1975	148.3	_f__wv	26.02	2931	112.6	72.22	8998	124.6	201.72	25791	127.9
_s_c__	79.80	7112	89.1	42.72	4908	114.9	40.70	4797	117.8	_f_s__	47.22	5437	115.1	62.76	7791	124.1	75.60	9448	125.0
_s_c_f	79.14	7003	88.5	46.28	5080	109.7	44.08	4953	112.4	_f_s_v	218.64	24034	109.9	374.13	40681	108.7	478.42	51455	107.6
_s_cr_	22.32	2589	116.0	15.95	2230	139.9	15.24	2194	144.0	_f_sw_	42.97	4693	109.2	56.48	6724	119.1	68.51	8198	119.7
_s_crf	20.06	2329	116.1	15.12	2063	136.4	14.46	2034	140.7	_f_swv	17.90	1917	107.1	22.89	2745	119.9	27.38	3332	121.7
_sp___	75.45	6680	88.5	43.93	4812	109.5	41.79	4691	112.2	_fr___	77.66	9309	119.9	202.55	25288	124.8	566.21	71658	126.6
_sp__f	74.90	6638	88.6	42.50	4713	110.9	40.44	4592	113.6	_fr__v	118.45	13146	111.0	311.14	33934	109.1	636.17	68389	107.5
_sp_r_	19.24	2321	120.6	13.67	2011	147.1	13.04	1979	151.8	_fr_w_	68.54	7730	112.8	178.79	21596	120.8	502.32	61600	122.6
_sp_rf	18.33	2169	118.3	13.42	1907	142.1	12.86	1881	146.2	_fr_wv	25.17	2788	110.7	71.09	8585	120.8	199.71	24647	123.4
_spc__	75.26	6660	88.5	46.66	4978	106.7	44.36	4850	109.3	_frs__	44.93	5154	114.7	60.17	7426	123.4	72.44	8986	124.1
_spc_f	76.40	6730	88.1	44.39	4830	108.8	42.19	4709	111.6	_frs_v	218.66	23897	109.3	373.86	40436	108.2	480.67	51374	106.9
spcr	20.30	2397	118.1	14.20	2061	145.1	13.54	2031	150.0	_frsw_	40.62	4398	108.3	53.84	6321	117.4	65.20	7736	118.6
_spcrf	18.11	2167	119.7	13.26	1909	144.0	12.64	1880	148.8	_frswv	15.16	1646	108.6	19.61	2373	121.0	23.66	2903	122.7
u_____	149.07	12211	81.9	63.90	7002	109.6	60.51	6779	112.0	v_____	68.96	8216	119.1	180.83	22488	124.4	506.83	63862	126.0
u____f	127.67	10977	86.0	59.93	6863	114.5	56.92	6688	117.5	v____v	85.84	9722	113.3	231.87	25626	110.5	489.96	52935	108.0
u___r_	67.94	6293	92.6	32.65	4136	126.7	31.11	4039	129.8	v___w_	66.39	7497	112.9	178.72	21509	120.3	504.85	61621	122.1
u___rf	65.63	6107	93.1	31.87	3981	124.9	30.30	3884	128.2	v___wv	24.37	2655	108.9	72.11	8584	119.0	203.56	24810	121.9
u__c__	148.65	12238	82.3	63.74	7023	110.2	60.30	6784	112.5	v__s__	41.27	4663	113.0	54.48	6671	122.5	65.78	8088	123.0
u__c_f	128.49	10998	85.6	59.96	6828	113.9	56.92	6647	116.8	v__s_v	158.98	17623	110.8	288.40	31538	109.4	370.96	40084	108.1
u__cr_	67.92	6289	92.6	32.67	4138	126.7	30.98	4029	130.0	v__sw_	40.81	4446	108.9	53.92	6397	118.6	65.44	7794	119.1
u__crf	65.70	6109	93.0	31.85	3981	125.0	30.29	3886	128.3	v__swv	15.29	1639	107.2	19.49	2343	120.2	23.21	2841	122.4
u_p___	104.88	9150	87.2	75.73	7517	99.3	72.79	7915	108.7	v_r___	66.40	7827	117.9	171.62	21193	123.5	478.86	59973	125.2
u_p__f	110.20	9488	86.1	78.46	7712	98.3	75.59	8130	107.5	v_r__v	85.25	9584	112.4	231.87	25505	110.0	490.00	52731	107.6
u_p_r_	53.74	5169	96.2	40.23	4351	108.1	39.09	4707	120.4	v_r_w_	62.97	7132	113.3	168.02	20287	120.7	470.80	57804	122.8
u_p_rf	49.03	4777	97.4	38.22	4124	107.9	36.67	4431	120.8	v_r_wv	19.03	2188	115.0	56.53	7070	125.1	163.89	20787	126.8
u_pc__	112.20	9648	86.0	85.17	8182	96.1	81.52	8554	104.9	v_rs__	38.49	4342	112.8	51.11	6234	122.0	61.64	7554	122.5
u_pc_f	108.86	9371	86.1	79.45	7736	97.4	77.17	8191	106.2	v_rs_v	158.79	17498	110.2	288.47	31354	108.7	370.62	39817	107.4
u_pcr_	52.51	5089	96.9	40.44	4367	108.0	39.25	4721	120.3	v_rsw_	37.79	4148	109.8	50.39	6011	119.3	60.63	7306	120.5
u_pcrf	48.80	4763	97.6	38.22	4121	107.8	36.88	4442	120.5	v_rswv	12.41	1369	110.3	16.18	1997	123.4	19.63	2451	124.8
us____	79.99	7132	89.2	49.91	5376	107.7	47.54	5245	110.3	vf____	68.98	8210	119.0	180.99	22513	124.4	507.39	63954	126.0
us___f	80.10	7142	89.2	47.50	5237	110.2	45.29	5113	112.9	vf___v	85.79	9725	113.4	232.01	25666	110.6	490.45	52986	108.0
us__r_	23.45	2686	114.6	16.14	2291	141.9	15.37	2253	146.6	vf__w_	66.19	7488	113.1	178.31	21496	120.6	503.81	61576	122.2
us__rf	20.88	2388	114.4	13.92	2004	144.0	13.29	1978	148.9	vf__wv	22.14	2484	112.2	64.48	7957	123.4	182.94	23080	126.2
us_c__	79.78	7100	89.0	42.73	4912	114.9	40.65	4792	117.9	vf_s__	41.43	4676	112.9	54.74	6694	122.3	66.11	8120	122.8
us_c_f	79.14	6997	88.4	46.28	5079	109.7	44.08	4952	112.3	vf_s_v	159.24	17656	110.9	288.45	31555	109.4	371.67	40151	108.0
us_cr_	22.32	2586	115.9	15.95	2230	139.8	15.25	2195	144.0	vf_sw_	40.68	4436	109.1	53.70	6392	119.0	65.26	7793	119.4
us_crf	20.07	2329	116.0	15.12	2062	136.4	14.46	2035	140.7	vf_swv	15.27	1638	107.2	19.46	2343	120.4	23.32	2848	122.2
usp___	72.29	6401	88.5	37.16	4290	115.5	35.34	4188	118.5	vfr___	66.14	7787	117.7	171.03	21067	123.2	477.22	59593	124.9
usp__f	71.24	6305	88.5	37.72	4306	114.1	35.91	4212	117.3	vfr__v	85.15	9561	112.3	232.18	25515	109.9	491.18	52771	107.4
usp_r_	17.34	2164	124.8	12.50	1910	152.8	11.97	1887	157.7	vfr_w_	62.80	7036	112.0	167.38	19985	119.4	468.63	56835	121.3
usp_rf	16.26	2046	125.8	12.11	1829	151.0	11.58	1802	155.7	vfr_wv	18.92	2158	114.1	56.23	6963	123.8	163.24	20483	125.5
uspc__	75.19	6588	87.6	45.62	4827	105.8	43.33	4691	108.3	vfrs__	38.36	4310	112.4	50.96	6188	121.4	61.33	7497	122.2
uspc_f	72.53	6407	88.3	40.90	4512	110.3	38.97	4410	113.2	vfrs_v	159.03	17488	110.0	288.32	31300	108.6	371.63	39827	107.2
uspcr_	17.29	2162	125.0	12.51	1909	152.5	11.93	1886	158.1	vfrsw_	37.87	4102	108.3	50.54	5938	117.5	60.79	7210	118.6
uspcrf	16.27	2046	125.7	12.10	1828	151.1	11.56	1801	155.8	vfrswv	12.33	1347	109.3	16.15	1966	121.7	19.61	2412	123.0