ENERGY CONSUMPTION ANALYSIS OF PARALLEL ALGORITHMS RUNNING ON MULTICORE SYSTEMS AND GPUS

THESIS

Presented to the Graduate Council of
Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Ivan Zecena, B.S.

San Marcos, Texas
August 2013

ENERGY CONSUMPTION ANALYSIS OF PARALLEL ALGORITHMS RUNNING ON MULTICORE SYSTEMS AND GPUS

Committee Members Approved:

Ziliang Zong, Chair

Jin Tongdan

Apan Qasem

Approved:

J. Michael Willoughby
Dean of the Graduate College

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

Dedicated to my parents, whose support and encouragement throughout my graduate studies has been the key to completing this work.

ACKNOWLEDGEMENTS

I would like to thank Dr. Ziliang Zong for his guidance and support during my time working with him as a research assistant; he has been an amazing advisor whose input and constant advise have made me a better student and researcher. I would also like to thank Dr. Apan Qasem, Dr. Tongdan Jin, and Dr. Martin Burtscher for their expertise, feedback, and overall support in putting together this work. This work was supported by the U.S. National Science Foundation under Grant No. CNS-1118043, and the U.S. Department of Agriculture under Grant No. 2011-38422-30803.

This manuscript was submitted on June 24, 2013.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF ACRONYMS

BH..... Barnes Hut

CPU..... Central Processing Unit

DRAM.... Dynamic Random Access Memory

FLOPS.... Floating-point Operations Per Second

GPU..... Graphics Processing Unit

HPC..... High Performance Computing

NB..... N-Body

ABSTRACT

ENERGY CONSUMPTION ANALYSIS OF PARALLEL ALGORITHMS RUNNING ON MULTICORE SYSTEMS AND GPUS

by

Ivan Zecena, B.S.

Texas State University-San Marcos

August 2013

SUPERVISING PROFESSOR: ZILIANG ZONG

As multicore computers and High Performance Computing systems in general continue to increase their number of processors and processing power, so too have the energy consumption and power requirements of these systems increased. The amount of dollars spent on providing energy to data centers continues to escalate and in the U.S. alone billions of dollars are spent each year. In fact, energy consumption has become so important in today's computing world that the need for energy efficient systems and applications has become critical. In this work, we analyze the energy efficiency of several parallel applications executed on multiple CPUs and GPUs. In chapter I, we discuss different parallel sorting algorithms and their energy efficiency. In particular, we show

how software optimization such as modifying the task granularity of a sorting algorithm can save energy. In chapter II, we look at several implementations of 2 famous N-Body particle simulators and profile their performance on CPUs and GPUs. Our results indicate that the GPU implementations provide applications that are orders of magnitude more energy efficient. Finally, in chapter III we show some of the common pitfalls and fallacies when measuring the energy consumption of GPU applications. In addition, we provide a methodology to successfully overcome these issues and accurately measure the energy consumption of GPU applications.

CHAPTER I

ENERGY CONSUMPTION ANALYSIS OF PARALLEL SORTING ALGORITHMS RUNNING ON MULTICORE SYSTEMS

1.1 Introduction

Significant energy consumption has become a top concern for many data centers and high performance computing applications. It is estimated that in 2011 alone, servers and data centers in the United States consumed more than 100 billion kWh, which amounts to a $7.4 billion energy bill [EPA, 2007; Lefurgy, 2011]. At the same time, as the processing power increases and newer and larger scientific applications are developed, computer scientists and engineers are turning their attention to examining ways to reducing energy consumption in their applications.

Sorting is one of the fundamental algorithms in computer science and sorting algorithms have been widely used in a myriad of fields; from scientific applications that need to manipulate and make sense of their data, to search engines like Google, which uses sorting to implement its PageRanking system that ranks billions of pages as it crawls the Web [Brin and Page, 2000]. In fact, it is estimated that 85% of scientific applications in the world depend exclusively on sorting algorithms [Pacheco, 2011]. Therefore, a massive amount of energy can be saved if we could improve the energy-efficiency of traditional sorting algorithms. However, previous studies primarily focused on reducing

the time complexity and improving the performance of sorting algorithms. The energy consumption behavior of sorting algorithms have not been fully explored.

In this chapter, we conduct an in-depth analysis on performance and energy consumption of three well known sorting algorithms: Odd-even sort, Shellsort, and Quicksort. These algorithms are carefully chosen to cover different implementation methods and time complexity categories. The Odd-even sort and Shellsort are implemented using iterative methods while the Quicksort is implemented recursively using OpenMP tasks. The time complexity of Odd-even sort, Shellsort, and Quicksort algorithms are O(n^2), O($n^{3/2}$) and O($nlogn$) respectively. We analyze both, the serial version and the parallel version of these three sorting algorithms, using a shared-memory system that contains two quad-core AMD 2380 Opteron processors. The size of data to be sorted scales from 10,000 to 1 billion elements.

The remaining parts of this chapter are organized as follows. Section 1.2 provides relevant background about sorting algorithms and parallel sorting. In Section 1.3, we explain our algorithm design and implementation details. The experimental results are illustrated and evaluated in Section 1.4. Related work is discussed in Section 1.5. And finally, we conclude this chapter in Section 1.6.

1.2 Background

1.2.1 Sorting Algorithms

Odd-Even, Shellsort, and Quicksort are three well known and commonly used sorting algorithms today [Kataria, 2008; Lang, 2010]. Odd-Even is a simple algorithm with a

O(n^2) time complexity that performs well on small data sets. Shellsort is a comparison-based sorting algorithm that improves upon Odd-Even sort in that it has an average time complexity of O($n^{3/2}$) and performs exceptionally well on medium-to-large data sets. Quicksort is known as one of the fastest sorting algorithms in practice today due to its O($nlogn$) time complexity and low memory requirements [Lang, 2010]. These three algorithms are selected to cover both iterative implementation (Odd-Even sort and Shellsort) and recursive implementation(Quicksort). We analyze the performance and energy consumption of the serial code and parallel code of each algorithm as well. In Section 1.3, we will provide more details on how to parallelize each algorithm.

1.2.2 Amdahl's Law

One of the guiding principles when improving and parallelizing applications is Amdahl's law. Amdahl's law is used as a guideline to find the maximum expected improvement in a program when only part of it can be improved. In essence, it gives us the theoretical maximum speedup when parallelizing a program across multiple cores [Breshears, 2009]. Amdahl's law states that if P is the proportion of a program that can be made parallel, and (1 - P) is the proportion that remains serial, then the maximum speedup that can be achieved with N processors is given by $Speedup = \frac{1}{(1-P)+\frac{P}{N}}$. In other words, the speedup of parallel code will not linearly grow with the number of cores. It is limited by the percentage of code that can be parallelized. For instance, if 95% of a program can be parallelized, the theoretical maximum speedup assuming an infinite number of cores would be 20x, regardless of how many processors we use. A more realistic approach would be using 8 cores, which yields a theoretical maximum of 6x. Thus, the less serial

code an application executes, the greater the speedup will be.

1.2.3 Task Decomposition and Task Granularity

In order to transform a sequential algorithm to a concurrent algorithm, the first step is to identify the code segment that can be executed in parallel. Task decomposition refers to the process of splitting a big task into smaller tasks that can be concurrently executed. Task granularity, accordingly, is defined as the amount of computation that a task needs to complete. The more work a task will do before synchronization, the coarser the granularity will be. Since task decomposition and task management is not free, we must avoid the danger of creating over fine-grained tasks because they do not have sufficient work assigned to threads to overcome the overhead cost. We will use Quicksort as an example to explain the impact of task granularity on performance and energy consumption in Section 1.4.

1.3 Algorithms Design and Environment

In order to compare the performance and energy consumption of the serial code and parallel code, we developed a parallel version for each of the aforementioned sorting algorithms using OpenMP [Pacheco, 2011]. The OpenMP extension allows us to add parallelism to our algorithms on our shared-memory based testbed with two quad-core AMD 2380 Opteron processors. This is accomplished by issuing sets of *#pragmas* that indicate the compiler which sections of code to parallelize. All three algorithms utilize a dynamically-allocated array structure to store the randomly-generated sample data. Our algorithms are implemented as follows.

```
void Odd_even(long a[], long n)
{
   long phase;
   long tmp,i;

#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++)
   {
      if (phase % 2 == 0)
#        pragma omp for
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp for
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }//end for
}  // end Odd_even function
```

Figure 1.1: Parallel Odd-Even sort algorithm.

1.3.1 Iterative Sorting

Both, Odd-even sort and Shellsort are implemented iteratively. Odd-even uses phases to compare all odd or even indexed pairs of adjacent elements in the list. Phases are incremented from 0 to n, and odd phases compare all odd-indexed elements and even phases compare all even-indexed elements. For each new phase, the list is partitioned in a block fashion and one chunk is assigned per core so that data dependencies are eliminated. Figure 1.1 contains a code snippet illustrating this algorithm.

Shellsort on the other hand, works by comparing elements that are separated by a *gap*. The first element is compared to the element located *gap* positions down the list, the next element is then located *gap* positions away, and so on. Each new sub-list of elements

```
void shell_sort_parallel(long sort_arr[], long size)
{
  long i,j,k;
  long gap=1;

   while(gap*3+1 < size)
     gap = gap*3+1;

    while (gap >= 1)
    {
      #pragma omp parallel for num_threads(thread_count) private(i,k,j)
        for (i=0 ; i<gap ; i++)
        {
            for (j=i; j<size-gap; j+=gap)
            {
              k = j;
              while(k >= i && sort_arr[k] > sort_arr[k+gap])
              {
                swap_shep( sort_arr, k, k+gap );
                k -= gap;
              }
            }
        }

        gap = gap / 3;
    }//end
```

Figure 1.2: Parallel Shellsort algorithm.

to be compared is assigned to a core, so cores comparing and swapping their sub-lists of elements do not have data dependencies with each other. At the beginning, the gap is roughly 1/3 of the array size. Then for each new iteration the gap is gradually reduced to 1. Figure 1.2 illustrates how the parallel Shellsort algorithm works.

1.3.2 Recursive Sorting

Quicksort is a recursive algorithm that selects a *pivot* and partitions the list into two disjoint groups. All elements in the left group are less than the *pivot* and elements in the right group are greater than the *pivot*. The partition is performed recursively for each sub-list until all sub-lists have been sorted. The parallel Quicksort algorithm is implemented recursively as well using OpenMP Tasks, which are defined and supported in the OpenMP 3.0 Standard [Duran, 2009]. Each time a partition is called with a different

```
void QuickSortOmpTask(long array[], const long left, const long right, const int deep)
{
    if(left < right){
        if( deep )
        {
            const long part = QsPartition(array, left, right);
            #pragma omp task
            QuickSortOmpTask(array,part + 1,right, deep - 1);
            #pragma omp task
            QuickSortOmpTask(array,left,part - 1, deep - 1);
        }
        else
        {
            const long part = QsPartition(array, left, right);
            QsSequential(array,part + 1,right);
            QsSequential(array,left,part - 1);
        }
    }
}


void QuickSortOmp(long array[], const long size)
{
    #pragma omp parallel
    {
        #pragma omp single nowait
        {
            QuickSortOmpTask(array, 0, size - 1 , 15);
        }
    }
}
```

Figure 1.3: Parallel Quicksort algorithm.

list, a new task will be generated to complete the partition job. The parallel Quicksort algorithm is illustrated in the code snippet shown in Figure 1.3.

By using OpenMP tasks in our implementation, we parallelize this irregular and recursive algorithm by creating a pool of tasks from which available cores pull tasks out. The number of tasks and the amount of work each task will do is determined by how *deep* we want the level of recursion to go. The greater the depth, the less amount of work each task will complete and the finer the task granularity will become. As we decrease the depth, less tasks will be created and each task will contain more work [Breshears, 2009]. We will tune the task granularity of parallel Quicksort algorithm in the experimental evaluation section by varying the depth and discuss the impact of task granularity on performance and energy.

Sorting Algorithms Performance (1 machine, 1-8 cores)									
Data Size	Algorithm	Time (s) 1 Core	Total Energy (J)	Time (s) 2 Cores	Total Energy (J)	Time (s) 4 Cores	Total Energy (J)	Time (s) 8 Cores	Total Energy (J)
10,000	Odd-Even	1.99	268.6J	1.00	134.8J	0.53	71.3J	0.29	39.0J
	Shellsort	0.01	1.3J	0.01	1.3J	0.01	1.3J	0.01	1.3J
	Quicksort	0.01	1.3J	0.01	1.3J	0.01	1.3J	0.04	5.4J
100,000	Odd-Even	199.57	27176.6J	99.26	13804.8J	53.05	7473.6J	27.31	4021.8J
	Shellsort	0.23	30.9J	0.14	18.8J	0.09	12.1J	0.07	9.4J
	Quicksort	0.07	9.4J	0.04	5.4J	0.03	4.0J	0.06	8.1J
500,000	Odd-Even	4968.73	677369.7J	2500.20	345726.8J	1302.90	184657.0J	675.19	100622.9J
	Shellsort	1.67	269.2J	0.93	134.9J	0.55	74.0J	0.37	49.8J
	Quicksort	0.37	49.8J	0.19	25.5J	0.11	14.8J	0.09	12.1J
1,000,000	Odd-Even	20091.48	2780763.0J	9998.62	1382570.4J	5232.57	743229.9J	2716.37	406571.3J
	Shellsort	3.69	539.6J	2.06	270.8J	1.16	135.7J	0.76	102.2J
	Quicksort	0.76	102.2J	0.40	53.8J	0.23	30.9J	0.15	20.2J
10,000,000	Shellsort	62.61	8591.2J	32.95	4563.7J	18.12	2713.2J	10.03	1497.1J
	Quicksort	9.30	1358.3J	4.79	684.6J	2.68	422.2J	1.64	282.6J
100,000,000	Shellsort	1115.57	152286.0J	575.33	79914.9J	298.02	42670.3J	157.18	23524.7J
	Quicksort	167.76	23024.2J	86.00	12067.0J	45.54	6541.4J	25.82	3783.8J

Figure 1.4: Performance of sorting algorithms.

1.4 Experimental Evaluation

In this section, we comprehensively evaluate our parallel sorting algorithms using the performance, speedups, and energy efficiency metrics. Before we dive into the detailed discussion, we first describe our testbed system.

1.4.1 Experimental Environment

Our experimental environment consists of a computer node composed of two Quad-Core AMD Opteron(tm) 2380 Processors running at 2.5 GHz. Since two Quad-core processors are used per node, there are a total of 8 processing cores in this shared-memory setup. In addition, this node is connected to a meter node that runs in the background and measures the energy consumption of the node. In order to properly evaluate the performance and energy efficiency of our algorithms, the meter measures the node's energy consumption every second and all the readings are collected into a file. For jobs that take less than 1 second to complete, the energy consumption is derived from the run time and the node's base energy consumption. For all our evaluations the base energy consumption of the node when idle is 134.47 Watts.

1.4.2 General Performance and Energy Consumption Analysis

We tested our parallel Odd-even, Shellsort, and Quicksort algorithms using 1,2,4, and 8 cores respectively. For each case, we scaled the unsorted sample data from 10,000 up to 1 billion elements. The performance data and energy consumption data for each sorting algorithm is shown in Figure 1.4.

In Figure 1.4, we observe that the execution time of each algorithm decreases as we increase the number of cores, which is predictable. We also notice that total energy consumption reduces as the execution time gets shorter, even if the number of cores doing work is increased. The reason is that the energy consumption of the extra cores is negligible compared to the base power being consumed by the machine. The only exception to this behavior occurs when the sample data is so small that the overhead of parallelization and synchronization is significantly larger than the performance gained by using more cores. The exception can be observed when sorting 10,000 elements. In this case, both Shellsort and Quicksort have no performance gain or energy savings. It is in fact the case that Quicksort performs worse and consumes more energy when sorting 10,000 elements and the number of cores is increased from 4 to 8. However, for all remaining data sets, the larger the data set is the more energy-efficient our parallel sorting algorithms become. Figure 1.5 and Figure 1.6 further detail the speedups and energy reductions obtained for each algorithm.

Again, as shown in these figures, the speedups and energy consumption percentages prove that in a shared-memory environment, using as many cores as possible to sort large data sets will yield a faster and more energy-efficient sorting process. In addition, as indicated by Amdahl's law, the maximum speedup is achieved when sorting the largest data set of 1 billion elements. Looking at Quicksort for example, a speedup of 6.50x was achieved compared to a 1.16x when sorting only 100,000 elements. Energy wise, sorting 1 billion elements consumed only 16.4% of the original energy compared to 86.2% when sorting 100,000 elements.

Speedups (over serial)					
Data Size	Algorithm	Serial	Parallel		
		1 Core	2 Cores	4 Cores	8 Cores
10,000	Odd-Even	1x	1.98x	3.75x	6.79x
	Shellsort	1x	1.55x	2.00x	2.33x
	Quicksort	1x	0.00x	0.85x	0.17x
100,000	Odd-Even	1x	2.01x	3.76x	7.31x
	Shellsort	1x	1.65x	2.47x	3.52x
	Quicksort	1x	1.81x	2.48x	1.16x
500,000	Odd-Even	1x	1.99x	3.81x	7.36x
	Shellsort	1x	1.79x	3.03x	4.16x
	Quicksort	1x	1.89x	3.30x	4.02x
1,000,000	Odd-Even	1x	2.01x	3.84x	7.40x
	Shellsort	1x	1.79x	3.18x	4.81x
	Quicksort	1x	1.90x	3.34x	5.21x
10,000,000	Shellsort	1x	1.90x	3.45x	6.24x
	Quicksort	1x	1.94x	3.47x	5.67x
100,000,000	Shellsort	1x	1.94x	3.74x	7.10x
	Quicksort	1x	1.95x	3.68x	6.50x

Figure 1.5: Speedup of sorting algorithms.

Energy Consumption(compared to serial)					
Data Size	Algorithm	Serial	Parallel		
		1 Core	2 Cores	4 Cores	8 Cores
10,000	Odd-Even	100%	50.0%	26.5%	14.5%
	Shellsort	100%	100%	100%	100.0%
	Quicksort	100%	100%	100%	415.0%
100,000	Odd-Even	100%	50.0%	27.5%	14.7%
	Shellsort	100%	60.8%	39.2%	30.4%
	Quicksort	100%	57.4%	42.6%	86.2%
500,000	Odd-Even	100%	51.0%	27.3%	14.9%
	Shellsort	100%	50.1%	27.5%	18.5%
	Quicksort	100%	51.2%	29.7%	24.3%
1,000,000	Odd-Even	100%	49.7%	26.7%	14.6%
	Shellsort	100%	50.2%	25.1%	18.9%
	Quicksort	100%	52.6%	30.3%	19.8%
10,000,000	Shellsort	100%	53.1%	31.6%	17.4%
	Quicksort	100%	50.4%	31.1%	20.8%
100,000,000	Shellsort	100%	52.5%	28.0%	15.4%
	Quicksort	100%	52.4%	28.4%	16.4%

Figure 1.6: Energy consumption of sorting algorithms.

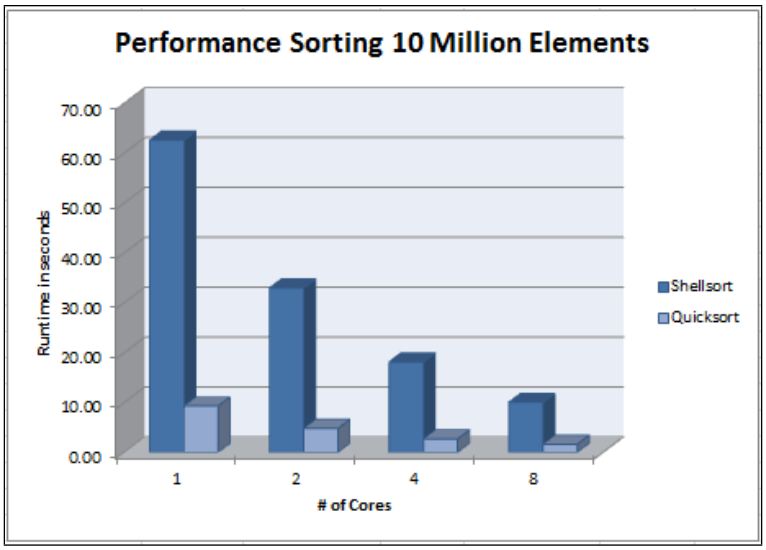

Figure 1.7: Performance comparison of Shellsort and Quicksort.

1.4.3 Shellsort and Quicksort Comparison

Due to the fact that clusters and HPC systems usually deal with large amounts of data, we decided to further analyze the energy-efficiency of Shellsort and Quicksort when sorting 10 million elements. We ignored the results obtained for Odd-even sort because this algorithm takes too long to run for data sets at this scale. Figures 1.7 and 1.8 show the performance and energy consumption of both Shellsort and Quicksort.

Quicksort, as the conventional wisdom indicates, outperforms Shellsort [Lang, 2010]. Shellsort performs better than Quicksort for small data sets but Quicksort runs faster than Shellsort for large data sets such as 10 million. Our results also prove that the energy consumption of this algorithm is much lower than Shellsort, and the same behavior can be observed as we scale the work across many cores. As shown on Figures 1.4 and 1.6, running Quicksort on 2 cores consumes only 50.4% of the original energy, 31.11% on 4 cores, and 20.8% on 8 cores.

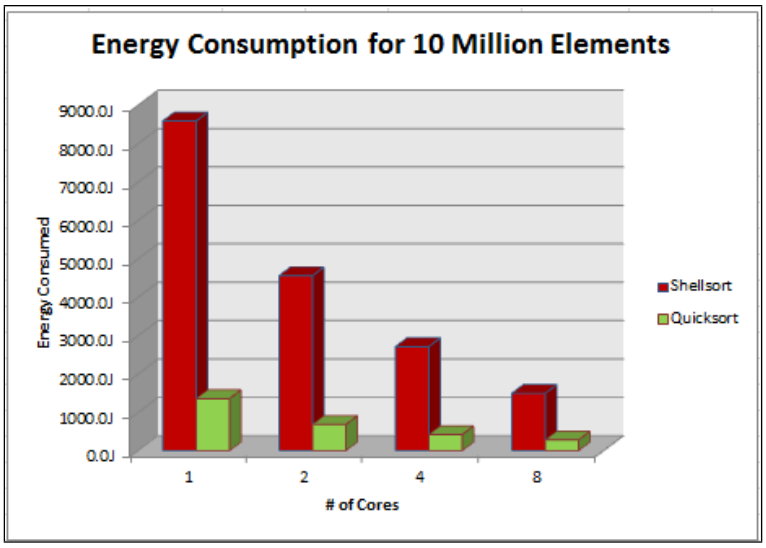

Figure 1.8: Energy consumption of Shellsort and Quicksort.

1.4.4 Impact of Changing Quicksort's Task Granularity

Since Quicksort proves to be the most efficient algorithm among these three algorithms, we also analyze the impact of task granularity on performance and energy by varying the *depth* parameter, as explained in Section 1.3. We ran Quicksort with recursion level depths of 5,10,15, and 20, and sorted the same 10 million elements as before. Figures 1.9 and 1.10 illustrate our results.

We can see that energy consumption of the algorithm is reduced the most when we reduce its recursion *depth* to 20. Using 8 cores and a *depth* of 20 levels produced an energy consumption of about 283.4 Joules, compared to 565.5 Joules when using a *depth* of 5 levels, this represents an energy savings of 50%. Although changing the *depth* to 10 also consumes about the same energy, the speedup increase with a *depth* of 20 significantly outperforms using a *depth* of 10 levels, as seen in Figure 1.9. In short, our results show that having a *depth* of 20 for the Quicksort algorithm increases performance,

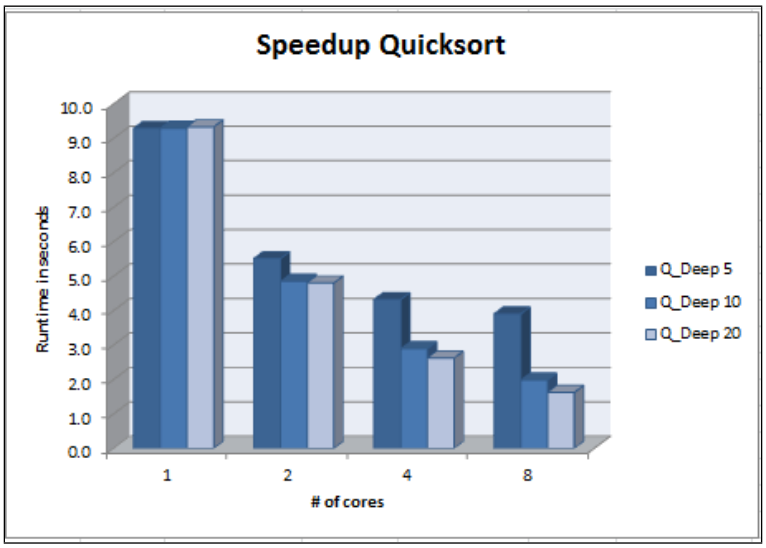

Figure 1.9: Performance comparison of different Quicksort depths.

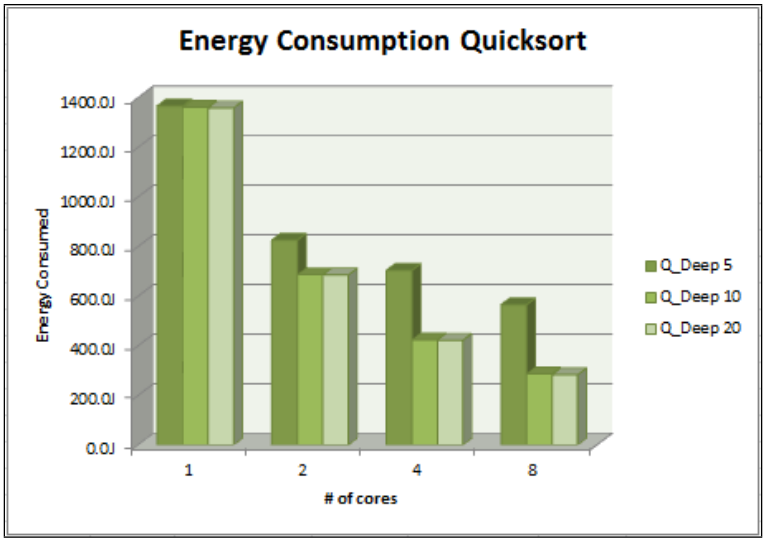

Figure 1.10: Energy consumption of different Quicksort depths.

reduces energy consumption, and provides the best energy-efficient performance.

1.5 Related Work

Most of the previous research in the HPC field has focused on improving performance and maximizing speed. But as energy costs continue to rise, the need for energy-aware applications has become critical. As a result, recent research started to shift focus to energy savings. For example, a number of recent work has proposed to optimize the energy consumption of processors [Weiser, 1994; Zong, 2011a] and disks [Douglis, 1995; Zong, 2011b] of high performance systems.

However, the research on investigating the energy consumption behavior of different algorithms is still in its infancy. To the best of our knowledge, there are only a few works that study the energy consumption of different sorting algorithms. Zent et. al. proposed a hybrid sorting algorithm (Quicksort + Heapsort) that is able to conserve energy by taking advantage of Dynamic Voltage Scaling (DVS) technology [Zeng, 2009]. Bunse et. al. published the energy consumption data of different sorting algorithms running on mobile devices [Bunse, 2009]. However, their work only analyzed the energy consumption of sequential sorting algorithms. In our study, we analyze the energy consumption of both sequential and parallel sorting algorithms through detailed power profiling. In addition, we discuss the impact of different parallel task designs and varying task granularities on the energy consumption of parallel Quicksort algorithm.

1.6 Conclusion

After analyzing the results generated by sorting large data sets with three parallel sorting algorithms, we can make the conclusion that the conventional wisdom of using more cores on a shared-memory system does in fact lead to more energy-efficient sorting algorithms. Our results also show that improving performance on a shared-memory system also produces an improvement in energy-efficiency. This is due to the fact that the energy consumed by the extra cores is minimal compared to the base energy consumption of the system, so reducing the running time of a program is the most efficient improvement. In this chapter, we also demonstrate that using a Quicksort algorithm on shared-memory systems can provide a significant increase in energy savings over other sorting algorithms. In addition, a fine-tuned Quicksort algorithm can also provide a 50% energy savings over another Quicksort algorithm that uses a different task granularity.

CHAPTER II

EVALUATING THE ENERGY EFFICIENCY OF PARALLEL N-BODY CODES ON MULTICORE CPUS AND GPUS

2.1 Introduction

Ever since Newton formulated his theory of gravitation to describe the motion of planets and stars (i.e., bodies) under mutual forces, n-body problems have attracted significant interest. In recent decades, researchers have started employing n-body simulations in a number of domains outside of astronomy, including for studying elementary particles that induce electric and magnetic forces upon each other. The n-body problem is simple in principle. Given the initial state (mass/charge, position and velocity) of n bodies at time T, we want to calculate the state of these bodies at a subsequent time T'. This is usually done incrementally by computing the evolution of the system in small time steps. For many real-world problems, the number of bodies, n, is very large (millions or billions). Hence, direct pair-wise n-body simulation may not be feasible due to its $O(n^2)$ algorithmic complexity. To make large problem sizes computationally tractable, several approximate n-body algorithms have been proposed, including the Barnes-Hut algorithm [Barnes, 1986] and the Fast Multipole Method (FMM) [Greengard, 1987]. In this chapter, we study two implementations of the direct $O(n^2)$ approach, which we call NB, as well as three implementations of the $O(nlogn)$ Barnes-Hut algorithm, which we call BH. We

chose the NB and BH methods because they represent interesting extremes. Of all the n-body algorithms we are aware of, NB is the most compute bound and BH is the most memory bound. Other fast algorithms, such as FMM, share similarities with BH but are less memory bound. Most of the existing work related to n-body simulation focuses on reducing the time complexity of the algorithm or on taking advantage of recent hardware improvements, in particular parallelization for multi-core CPUs and GPUs. However, the energy efficiency of n-body codes remains unexplored. In this chapter, we study the energy consumption of five parallel implementations of NB and BH. For BH, we analyze the P-Threads version written by Nicholas Chen at UIUC [Lonestar, 2009], the OpenMP version developed by Ricardo Alves at the University of Minho in Portugal, and the CUDA version from the LonestarGPU benchmark suite [Lonestar, 2010]. For NB, we study an OpenMP and a CUDA version we programmed. Our CUDA code outperforms the corresponding n-body implementation that ships with the CUDA 5.0 SDK [CUDA, 2013] and reaches over two teraflops on some inputs on a K20c GPU. Section 2.3 provides more detail about these implementations. Hereafter, we refer to the five codes as BH_{PThr}, BH_{OMP}, BH_{CUDA}, NB_{OMP}, and NB_{CUDA}.

We first evaluate the impact of the thread count on the energy efficiency of the multi-core CPU codes (BH_{PThr}, BH_{OMP}, and NB_{OMP}). We perform this analysis on two systems: one based on an Intel CPU (System 1) and the other based on two AMD CPUs (System 2). Our experiments show that up to 71% of the energy can be saved when all CPU cores are utilized. On the system that supports hyper-threading (System 1), we also evaluate the impact of hyper-threading on the energy consumption. Our results show that hyper-threading can improve the performance of BH_{PThr} and BH_{OMP} up to 30%, which

yields up to 21% energy savings when simulating one million bodies. However, the impact of hyper-threading on the energy efficiency of NB_{OMP} is negligible. Finally, we evaluate the benefit of using GPU acceleration on the energy efficiency of NB and BH. Our experiments show that the GPU codes outperform the CPU versions by orders of magnitude. For example, when simulating one million bodies, the BH_{CUDA} code running on a K20c GPU is 45 times faster and 97.6% more energy efficient than the BH_{OMP} code running on a Xeon E5620 CPU. When simulating one million bodies, the NB_{CUDA} code runs 424 times faster on the GPU and only consumes 0.27% of the energy compared to the NB_{OMP} code running on the multi-core CPU. In addition, we find our Kepler-based Tesla K20c GPU to outperform our Fermi-based GeForce GTX 480 GPU in energy efficiency on all tested programs and inputs. The remainder of this chapter is organized as follows. Section 2.2 discusses related work. Section 2.3 presents the CPU and GPU n-body implementations. Section 2.4 describes our systems and experimental methodology. Section 2.5 discusses and analyzes the results. Section 2.6 concludes this chapter.

2.2 Related Work

The n-body simulation problem has been studied extensively, and a variety of algorithms have been proposed. In particular, many previous studies focus on developing fast algorithms to break the $O(n^2)$ complexity boundary. In 1986, Barnes and Hut proposed the now well-known Barnes-Hut algorithm, which lowers the time complexity to $O(nlogn)$ using approximation [Barnes, 1986]. Based on the general principle of this algorithm, researchers have developed many variants to speed up the execution. Salmon

implemented a parallel version for distributed-memory machines [Salmon, 1990]. Later, Warren and Salmon proposed improved parallel implementations [M. Warren, 1993; Warren, 1992]. In 1997, Warren et al. exceeded one gigaflop when running their Barnes-Hut code on 16 Intel Pentium Pro processors [Warren, 1997]. Liu and Bhatt developed an algorithm that is based on a dynamic global tree that spans multiple processors [Liu, 2000]. Burtscher and Pingali wrote the first Barnes-Hut implementation that runs the entire algorithm on a GPU [Burtscher, 2011]. Bedorf and Zwart (2012) present a similar GPU implementation [Bedorf, 2012]. Several algorithms with a linear time complexity have also been proposed. The first was devised by Appel [Appel, 1985] in 1985. Greengard and Rokhlin developed the $O(n)$ Fast Multipole Method [Greengard, 1987]. Xue proposed another linear-time hierarchical tree algorithm for n-body simulations [Xue, 1998]. Very little work has been published on studying the energy efficiency of different n-body algorithms. The closest work we can find was published by Malkowski et al. [Malkowski, 2006]. They explore how to use low-power modes of the CPU and caches, and hardware optimization such as a load-miss predictor and data prefetchers, to conserve energy without hurting performance.

2.3 Implementation Description

We evaluate five different n-body codes belonging to two different categories. The NB implementations have $O(n^2)$ complexity and the BH implementations have $O(nlogn)$ complexity. The goal of all five programs is to simulate the time evolution of a star cluster under gravitational forces for a given number of time steps.

2.3.1 BH Algorithm

The Barnes-Hut algorithm approximates the forces acting on each body. It hierarchically partitions the volume around the n bodies into successively smaller cells and records this information in an unbalanced octree, which is a tree data-structure where every internal node has exactly eight children. Each cell then forms an internal node of the octree and summarizes information about all the bodies it contains. The leafs of the octree are the individual bodies. This spatial hierarchy reduces the time complexity to $O(nlogn)$ because, for cells that are sufficiently far away, the algorithm only performs one force calculation with the cell instead of performing one force calculation with each body inside the cell, thus drastically reducing the amount of computation. However, differing parts of the octree have to be traversed to compute the force on each body, making the code's control flow and memory-access patterns quite irregular. The P-Threads, OpenMP and CUDA codes we study per-form six key operations in each time step to implement the BH algorithm. The first is an $O(n)$ reduction to find the minimum and maximum coordinates of all bodies. The second operation builds the octree by hierarchically dividing the space containing the bodies into ever smaller cells in $O(nlogn)$ time until there is at most one body per cell. The third operation summarizes information in all subtrees in $O(n)$ time. The fourth operation approximately sorts the bodies by spatial distance in $O(n)$ time to improve the performance of the next operation. The fifth operation computes the force on each body in $O(nlogn)$ time by performing prefix traversals on the octree. This is by far the most time consuming operation in the BH algorithm. The final operation updates each body's position and velocity based on the computed force in $O(n)$ time. Note that the

P-Threads code only parallelizes the force calculation. The OpenMP and CUDA codes parallelize all six operations. In case of OpenMP, parallel for and parallel pragmas are used, in some cases in combination with gcc-specific synchronization primitives, memory fences, and atomic operations to handle data dependencies. The force calculation code uses a block-cyclic schedule whereas the other operations use the default schedule. The CUDA code incorporates many GPU-specific optimizations that are described elsewhere [Burtscher, 2011].

2.3.2 NB Algorithm

The direct NB algorithm performs precise pair-wise force calculations. For n bodies, $O(n^2)$ pairs need to be considered, making the calculation quadratic in the number of bodies. However, identical computations have to be performed for all n bodies, leading to a very regular implementation that maps well to GPUs. As with BH, all force calculations are independent, resulting in a large amount of parallelism. Both the OpenMP and the CUDA implementations perform two key operations per time step. The first is the $O(n^2)$ force calculation and the second is an $O(n)$ integration where each body's position and velocity are updated based on the computed force. The OpenMP code uses a parallel for pragma to parallelize the outer loop of the force calculation and the loop of the integration. The default schedule is used in both cases. The CUDA code is very similar in structure and uses GPU threads to completely eliminate these two loops, i.e., each thread handles a different iteration. In addition, the force calculation code employs data tiling in shared memory (a software controlled L1 data cache) and unrolls the inner loop. In summary, the NB codes are relatively straightforward, have a high computational density, and only

access main memory infrequently because of excellent caching. In contrast, the BH codes are quite complex (they repeatedly build unbalanced octrees and perform various traversals on them), have a low computational density, and perform mostly pointer-chasing memory accesses. Due to the lower time complexity, the BH_{CUDA} implementation is about 33 times faster on a K20c than the NB_{CUDA} code when simulating one million stars.

2.4 Evaluation Methodology

2.4.1 Systems, Compilers, and Inputs

We conducted our experiments on two machines. System 1 is based on a 2.4 GHz Xeon E5620 CPU with four cores running 32-bit CentOS 5.9. It contains two GPUs. The first GPU is a previous generation Fermi-based GeForce GTX 480 with 1.5 GB of global memory and 15 streaming multiprocessors (SMs) with 480 processing elements (PEs) running at 1.4 GHz. The second GPU is a current generation Kepler-based Tesla K20c with 5 GB of global memory and 13 SMXs with 2,496 PEs running at 0.7 GHz. The idle power of the GTX 480 is 54W, the K20c draws 13W when idling, and the entire system consumes 165W in idle mode. System 2 contains two quad-core AMD Opteron 2380 CPUs running at 2.5 GHz. It has no GPUs. The idle power draw of System 2 is 134.5W. We compiled the CUDA codes on System 1 with nvcc 5.0 using the $-O3$, $-ftz = true$, and $-arch = sm\_20$ or $-arch = sm\_35$ flags. For the P-Threads and OpenMP codes, we used gcc 4.6.3 on System 1 and gcc 4.4.6 on System 2 with the $-O3$ flag on both systems. We ran the BH programs with 250,000, 500,000 and one million stars and the NB programs with 25,000, 50,000 and 100,000 stars. We used ten time steps for all

experiments. The stars' positions and velocities are initialized according to the empirical Plummer model [Plummer, 1911], which mimics the density distribution of globular clusters.

2.4.2 Energy Profiling

We measure the system-wide energy consumption when running the application codes using a WattsUp power meter [WattsUp, 2010]. The meter's software runs in the background as a daemon and samples the voltage and current to compute the power. It has a 1 Hz sampling frequency and a power resolution of 0.1W. The energy measurements are derived from the power readings by integrating the power over the runtime of the codes. To improve the accuracy of our measurements, we report the median of five runs for each experiment. Additionally, we removed high-energy outliers in the data caused by operating system jitter [Jones, 2009] (a.k.a operating system noise or operating system interference) and external user activities such as logging into the system, which are unavoidable in multi-user systems. Finally, we started our measurements with a four-second head delay before launching the program. This removes the overhead caused by starting the meter and our applications at the same time and allows the system to settle before the application program is launched. We then removed the first four seconds of power readings from our final calculations.

2.5 Experimental Evaluation

This section presents our experiments and analyzes the results. Subsection A discusses the impact of increasing the number of threads on the energy efficiency of the BH and NB

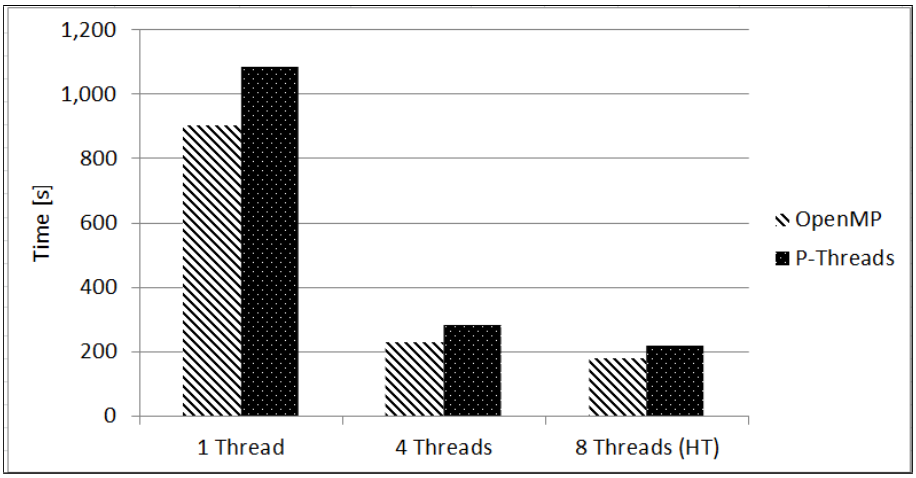

Figure 2.1: Runtimes of BH_{OMP} and $_{PThr}$ with 1 million bodies and 10 time steps running on System 1

codes. Subsection B studies the impact of hyper-threading on the performance and energy consumption. Subsection C evaluates the GPU results and compares them to the CPU results.

2.5.1 Impact of Thread Count

BH Algorithm Table 2.1 shows the runtime and the energy consumption of BH_{OMP} and BH_{PThr} on System 1 for one, four, and eight threads with 250,000, 500,000 and one million bodies over ten time steps. For all three input sizes and both implementations, we observe that the performance increases almost linearly (96% parallel efficiency) when going from one to four threads but sub-linearly (64% parallel efficiency) when going from four to eight threads, i.e., when going from one to two threads per CPU core using hyper-threading. We defer the discussion of the hyper-threading results to Subsection B.

Taking the 500,000-body experiments as an example, we observe that the OpenMP

Table 2.1: CPU BH results for 10 timesteps on the hyper-threaded 4-core system 1

# of Bodies	# of Threads	Algorithm	Runtime [s]	Energy [J]
250,000	1 Thread	OpenMP	183.0	30,776
		P-Threads	225.0	37,934
	4 Threads	OpenMP	47.5	9,035
		P-Threads	58.9	11,065
	8 Threads (HT)	OpenMP	38.3	7,539
		P-Threads	46.4	9,017
500,000	1 Thread	OpenMP	410.6	69,505
		P-Threads	497.6	83,643
	4 Threads	OpenMP	106.3	20,312
		P-Threads	130.1	24,811
	8 Threads (HT)	OpenMP	83.8	16,564
		P-Threads	101.5	19,818
1,000,000	1 Thread	OpenMP	902.7	152,462
		P-Threads	1,085.6	183,693
	4 Threads	OpenMP	232.7	44,723
		P-Threads	284.9	53,847
	8 Threads (HT)	OpenMP	180.7	35,937
		P-Threads	218.5	42,714

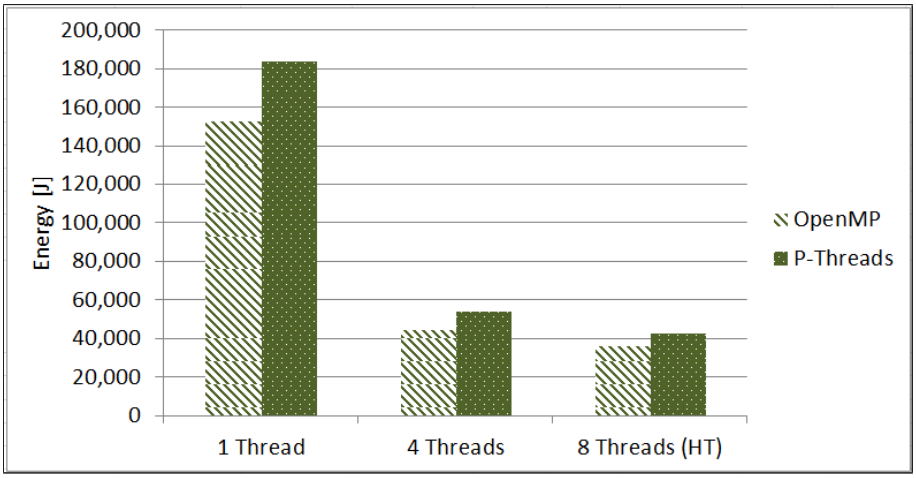

Figure 2.2: Energy consumption of BH_{OMP} and $_{PThr}$ with 1 million bodies and 10 time steps on System 1

code's runtime is 410.6 seconds with one thread and 106.3 seconds with four threads, which amounts to a speedup of 3.86 when quadrupling the number of cores used. Analyzing the energy consumption, we find that using one thread consumes 69,505 joules whereas using four threads only consumes 20,312 joules to compute the same result, a 71% reduction in energy usage. The P-Threads implementation behaves similarly but the absolute numbers are worse. It runs for 497.6 seconds with one thread and for 130.1 seconds with four threads, which amounts to a speedup of 3.82. Energy-wise, we see a 70% savings when using four threads instead of one. The other two inputs exhibit nearly identical trends. For all three inputs sizes, the OpenMP version of BH consistently outperforms the P-Threads version on System 1 in both performance and energy efficiency. On average, the OpenMP code is 22.2% faster and consumes 21.5% less energy than the P-Threads code. Figures 2.1 and 2.2 visualize the runtime and energy consumption, respectively, for the 1,000,000-body runs. The results for System 2 are provided in Table 2.2. Figures 2.3 and 2.4 graphically depict the runtimes and energy consumption for one million bodies. Since this system has eight physical cores, we obtained results for one, eight, and 16 threads and 250,000, 500,000 and one million bodies. On the middle input, the OpenMP implementation achieves a 7.89-fold speedup (98.6% parallel efficiency) and an 86% energy savings when going from one to eight threads. The P-Threads code yields a speedup of 7.17 (89.6% parallel efficiency) and an 85% reduction in energy consumption when going from one to eight threads. The results for the other inputs are again very similar.

Interestingly, the BH_{OMP} code scales substantially better on System 2 than the BH_{PThr} code. Whereas the OpenMP code is only 7.4% faster when using one thread, its

Table 2.2: CPU BH results for 10 time steps on the 8-core system 2

# of Bodies	# of Threads	Algorithm	Runtime [s]	Energy [J]
250,000	1 Thread	OpenMP	490.7	66,559
		P-Threads	528.4	71,743
	8 Threads	OpenMP	62.3	9,264
		P-Threads	75.1	11,025
	16 Threads	OpenMP	172.5	25,546
		P-Threads	81.9	11,979
500,000	1 Thread	OpenMP	1,039.6	141,165
		P-Threads	1,117.6	151,816
	8 Threads	OpenMP	131.8	19,498
		P-Threads	155.9	22,995
	16 Threads	OpenMP	344.6	51,134
		P-Threads	162.6	23,942
1,000,000	1 Thread	OpenMP	2,175.1	295,330
		P-Threads	2,325.7	316,155
	8 Threads	OpenMP	276.5	40,887
		P-Threads	321.2	47,221
	16 Threads	OpenMP	695.9	103,739
		P-Threads	324.1	47,598

advantage over the P-Threads code increases to 18.3% when using eight threads. The energy savings closely follow these percentages. In contrast, the two codes scale nearly identically on System 1. Comparing the single-thread runs across the two systems, we find that System 1 is, on average, 2.54 faster on the OpenMP code and 2.25 times faster on the P-Threads code. However, the energy-efficiency ratios are lower. System 1 outperforms System 2 by a factor of 2.04 on the OpenMP code and by a factor of 1.81 on the P-Threads code on average. Overall, System 1 is much faster and also more energy efficient than System 2, but System 2 has a lower average power consumption of 135.8 watts (compared to 168.7 watts for System 1). Note that these numbers are only a little above the idle power draw for both systems, showing that the one computation thread consumes relatively little extra energy. For reference, when using four threads on System 1 and eight threads on

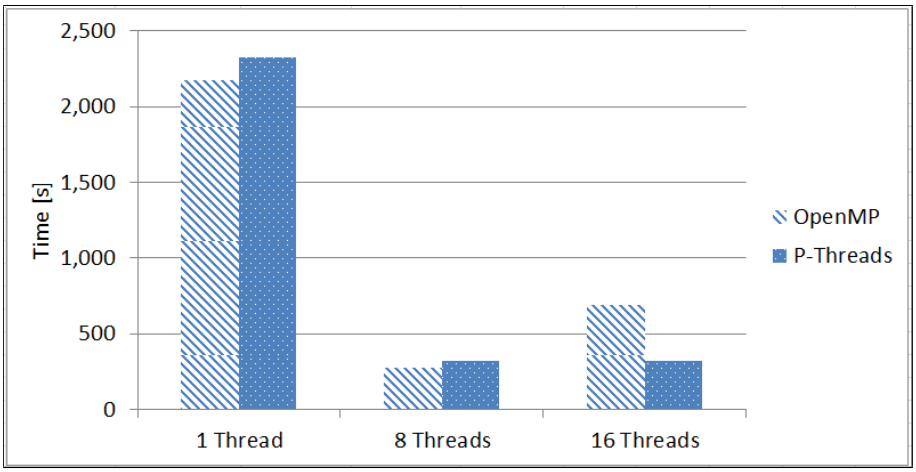

Figure 2.3: Runtimes of BH_{OMP} and $_{PThr}$ with 1 million bodies and 10 time steps on the 8-core System 2

System 2, the average power draw increases to 190.2 watts and 147.6 watts, respectively.

In summary, we find the OpenMP implementation to out-perform the P-Threads implementation in energy efficiency on both systems for all investigated inputs and thread counts. This is most likely because the P-Threads code only parallelizes the force calculation whereas the OpenMP code parallelizes all six algorithmic steps. Nevertheless, both BH codes scale well on our two multi-core CPU systems. Most importantly, we find on both systems that using all available cores results in large energy savings. The main reason for this is the high idle power. Since its contribution to the overall energy consumption is proportional to the runtime, reducing the runtime through parallelization saves energy. If the idle power were zero, using four cores to run the code, say, 3.86 times faster would actually increase the energy consumption relative to running the code on just a single core assuming all cores are independent (which they are not in current multi-core CPUs) and identical. As real systems have a non-zero idle power, there is a minimum

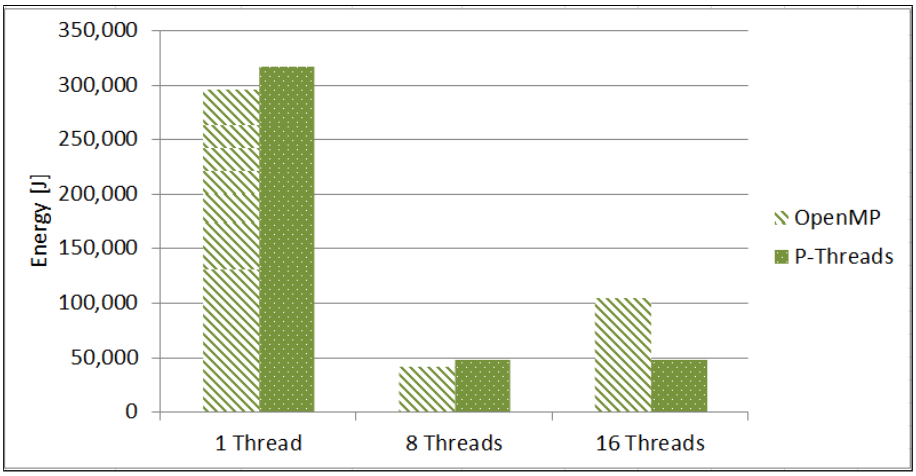

Figure 2.4: Energy consumption of BH_{OMP} and $_{PThr}$ with 1 million bodies and 10 time steps on the 8-core System 2

speedup that must be achieved by parallel code to be more energy efficient than serial code. For example, System 1 must execute the BH_{OMP} code at least 1.13 times faster using four cores to save energy over running the code on one core. The corresponding factor for System 2 is 1.10. Fortunately, a 10% or 13% speedup should be relatively easy to achieve on four or eight cores, meaning that parallelization is likely to be worthwhile to improve a program's energy efficiency on today's multi-core systems.

NB Algorithm Tables 2.3 and 2.4 contain the experimental results for our CPU implementation of the NB algorithm running on Systems 1 and 2, respectively. They show how the OpenMP code performs for different input sizes and thread counts. As before, we use 1, 4, and 8 threads on System 1 and 1, 8, and 16 threads on System 2. The input sizes are 25,000, 50,000 and 100,000 bodies on both systems. These sizes are smaller because the NB code is slower than the BH code. The NB_{OMP} results for the largest input are further illustrated in Figures 2.5 and 2.6 for System 1 and in Figures 2.7 and 2.8 for

Table 2.3: CPU NB_{OMP} results for 10 time steps on the hyper-threaded 4-core system 1

# of Bodies	# of Threads	Runtime [s]	Energy [J]
25,000	1 Thread	154.3	25,971
	4 Threads	39.9	7,344
	8 Threads (HT)	39.7	7,388
50,000	1 Thread	616.3	103,670
	4 Threads	159.2	29,563
	8 Threads (HT)	158.7	29,996
100,000	1 Thread	2,465.5	414,248
	4 Threads	637.6	119,447
	8 Threads (HT)	634.3	120,725

Table 2.4: CPU NB_{OMP} results for 10 time steps on the 8-core system 2

# of Bodies	# of Threads	Runtime [s]	Energy [J]
25,000	1 Thread	280.5	38,251
	8 Threads	35.2	5,157
	16 Threads	35.6	5,253
50,000	1 Thread	1,131.3	153,478
	8 Threads	141.6	20,809
	16 Threads	142.4	20,968
100,000	1 Thread	4,525.3	614,314
	8 Threads	567.9	83,875
	16 Threads	566.9	83,546

System 2. We leave the hyper-threading discussion for Subsection B.

On System 1, the NB code scales nearly identically on all three inputs. Going from one to four threads, we obtain an average speedup of 3.87 (corresponding to 96.7% parallel efficiency) and a 71.5% savings in energy. On System 2, the behavior is also very similar between the three inputs. Increasing the number of threads from one to eight yields a speedup of 7.975 (99.7% parallel efficiency) and energy savings of 86.4%.

Comparing the single-threaded runs on both systems, we find that System 1 is 1.83

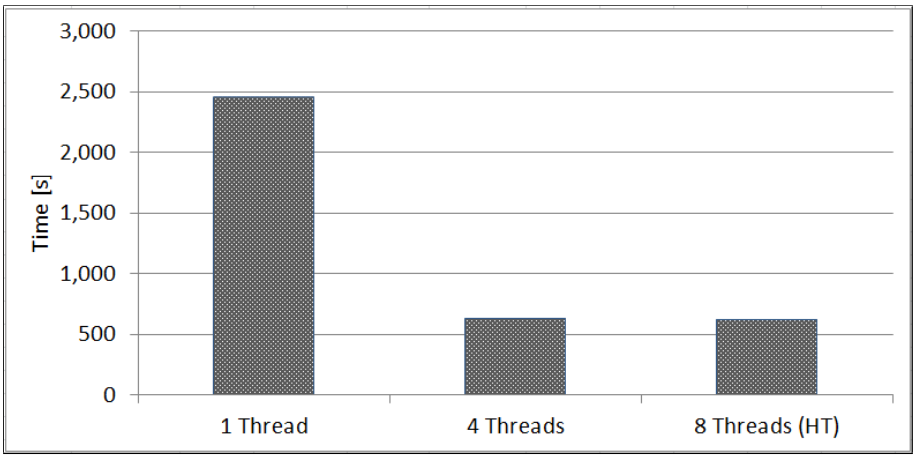

Figure 2.5: Runtime of NB_{OMP} with 100,000 bodies and 10 timesteps on System 1

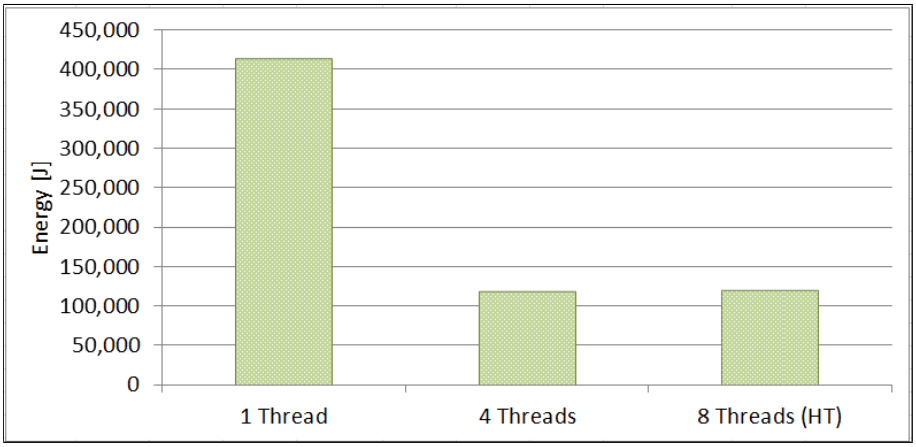

Figure 2.6: Energy consumption of NB_{OMP} with 100,000 bodies and 10 time steps on System 1

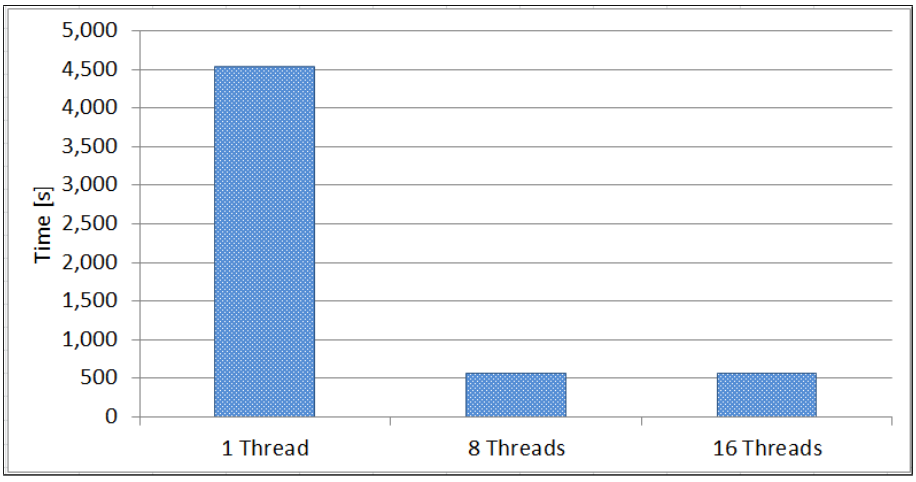

Figure 2.7: Runtime of NB_{OMP} with 100,000 bodies, 10 timesteps on the 8-core System 2

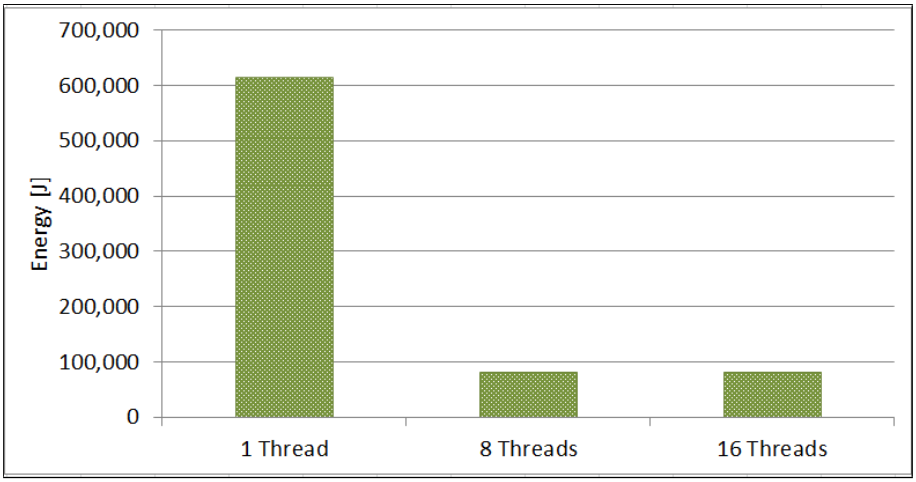

Figure 2.8: Energy consumption of NB_{OMP} with 100,000 bodies and 10 time steps on the 8-core System 2

times faster and 1.48 times more energy efficient than System 2 when running NB_{OMP}. The minimum speedup to be energy efficient using four cores on System 1 is 10.5%. System 2 only requires a minimum speedup of 8.2% on eight cores to be more energy efficient. In summary, the BH and NB CPU codes experience good scaling and boosts in energy efficiency on both tested systems. Clearly, parallelizing the BH and NB implementations is a very profitable way to improve their energy efficiency.

2.5.2 Impact of Hyper-Threading

In this subsection, we study the impact of hyper-threading (i.e., running two threads per CPU core) on the energy efficiency of System 1. System 2 does not support hyper-threading.

BH Algorithm Table 2.1 and Figures 2.1 and 2.2 above illustrate the results for BH. Whereas hyper-threading clearly provides a benefit, the benefit is smaller than the benefit of using more cores. For instance, going from one to two threads per core (i.e., increasing the thread count from 4 to 8) results in a parallel efficiency of about 64%. In contrast, going from one to four threads with one thread per core results in a parallel efficiency of about 96%. The results are nearly the same for both implementations and all inputs we tested. Note that the runtime as well as the energy consumption decrease when using hyper-threading. On average, hyper-threading yields additional energy savings of 18% on BH_{OMP} and 20% on BH_{PThr}. On System 2, running twice as many threads as the number of cores (i.e., increasing the thread count from 8 to 16) hurts the performance and the energy consumption, as the data in Table 2.2 and Figures 2.3 and 2.4 above show, because this system does not support hyper-threading. Interestingly, the degradation is small for

BH_{PThr} but large for BH_{OMP}. We are unsure what causes this difference in behavior between the two programs when oversubscribing threads to cores in System 2. Running more threads than there are (non-hyper-threading) cores hurts performance and increases the energy consumption on both codes and all inputs. Hence, we cannot recommend it.

NB Algorithm Surprisingly, hyper-threading does not help with NB as shown in Figures 2.5 and 2.6. Whereas it does result in a tiny speedup, as Table 2.3 reveals, it actually raises the energy consumption. Hence, hyper-threading does not further improve the energy efficiency of the NB code on System 1. The reason why hyper-threading improves the performance and the energy efficiency of the two BH implementations but not the NB implementation is the following: the BH codes are memory bound, meaning that they do not fully utilize the CPU cores because the memory system is the bottleneck. Hyper-threading enables each core to execute useful instructions from one thread whenever the other thread is stalled waiting for a memory request, thus boosting performance. In other words, hyper-threading helps hide some of the memory access latencies. In contrast, the NB code is compute bound, i.e., the CPU cores are already fully utilized when running one thread per core. Hence, there is no benefit from hyper-threading, but it should be pointed out that it also does not hurt performance. In summary, we conclude that adding extra cores is more useful than adding hyper-threading support from an energy-efficiency perspective. Yet, we find running two threads per core to save up to 21% energy (20% on average on BH_{PThr}). Moreover, the combination of both approaches results in the largest energy savings and should be used when available.

Table 2.5: GTX 480 GPU BH_{CUDA} and NB_{CUDA} results for 10 time steps

Algorithm	# of Bodies	Runtime [s]	Energy [J]
BH	250,000	1.3	376
	500,000	2.4	663
	1,000,000	4.8	1,223
NB	25,000	0.4	249
	50,000	1.1	371
	100,000	3.5	906

Table 2.6: K20c GPU BH_{CUDA} and NB_{CUDA} results for 10 time steps

Algorithm	# of Bodies	Runtime [s]	Energy [J]
BH	250,000	1.3	330
	500,000	2.2	547
	1,000,000	4.0	865
NB	25,000	0.5	196
	50,000	0.7	198
	100,000	1.5	329

2.5.3 Impact of GPU Acceleration

In this subsection, we study the energy efficiency of our CUDA implementations of BH and NB. We report the runtime over the entire application, including CPU and GPU code, and the energy consumption of the entire system when executing the accelerated code segments on a GTX 480 or a K20c GPU, both of which reside in System 1. As before, we run the BH code with 250,000, 500,000, and one million bodies and the NB code with 25,000, 50,000 and 100,000 bodies. Table 2.5 presents the results for the GTX 480 and Table 2.6 for the K20c. Figures 2.9 to 2.12 depict the same results graphically.

Comparing the runtimes between the two GPUs in Figures 2.9 and 2.10, we find that the K20c outperforms the GTX 480 by 1.6% to 20.4% on BH and by up to a factor of 2.35

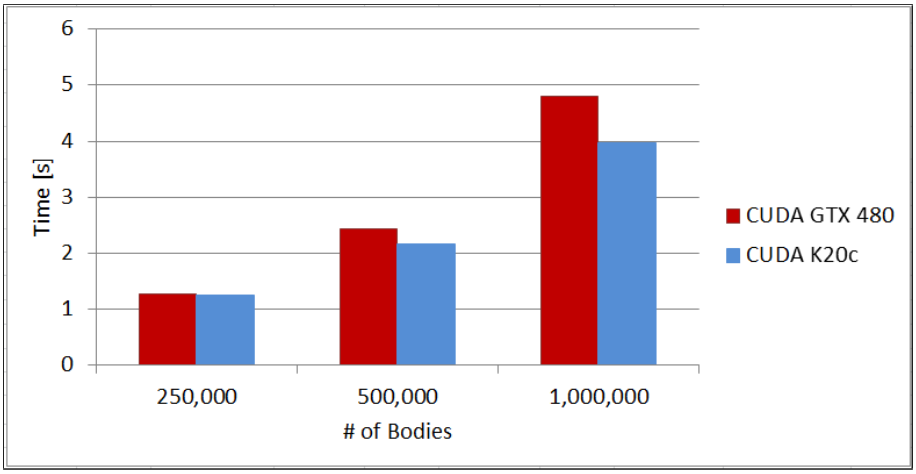

Figure 2.9: Runtime comparison of BH_{CUDA} code for 10 timesteps running on both GPUs

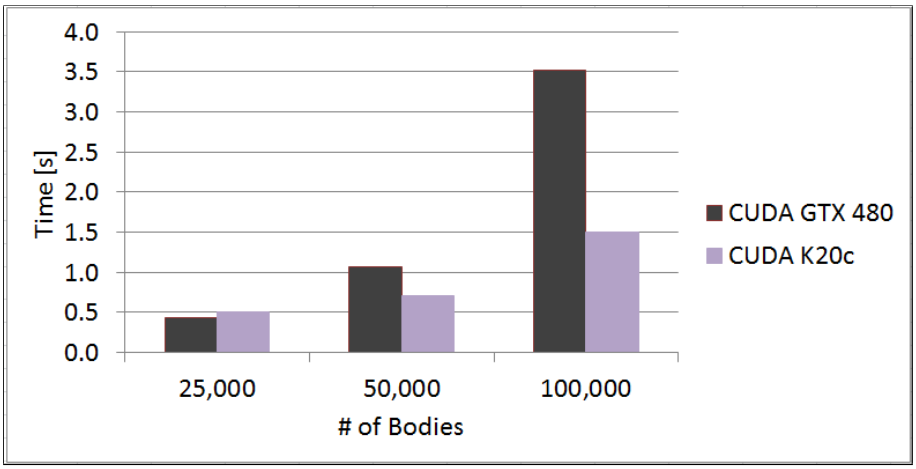

Figure 2.10: Runtime comparison of NB_{CUDA} code for 10 timesteps executed on both GPUs

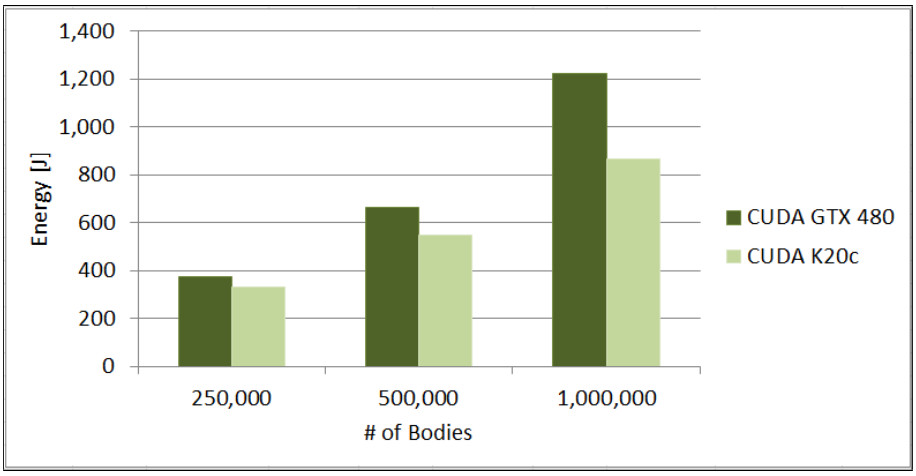

Figure 2.11: Energy consumption of BH_{CUDA} with 10 timesteps

on NB. However, on the smallest NB input, the GTX 480 is 13.6% faster. Nevertheless, the K20c is typically faster as it is based on the current-generation Kepler architecture whereas the GTX 480 is based on the previous-generation Fermi architecture.

This difference in GPU generations is also the reason for the better energy efficiency of the K20c, which was a primary design goal for the Kepler. Figures 2.11 and 2.12 show the energy consumption of BH_{CUDA} and NB_{CUDA}, respectively. The K20c is more energy efficient than the GTX 480 on all three inputs and both codes. On BH, the K20c saves between 12% and 29% energy. On NB, it saves between 21% and 64% energy. In summary, the K20c generally but not always outperforms the GTX 480. The difference in runtime seems to be more pronounced on compute-bound code like NB. More importantly, the K20c is substantially more energy efficient than the GTX 480, in particular on compute-bound code. Note that the higher performance of the K20c contributes to but is not the primary reason for its better energy efficiency. Rather, the K20c is based on a substantially lower power GPU design.

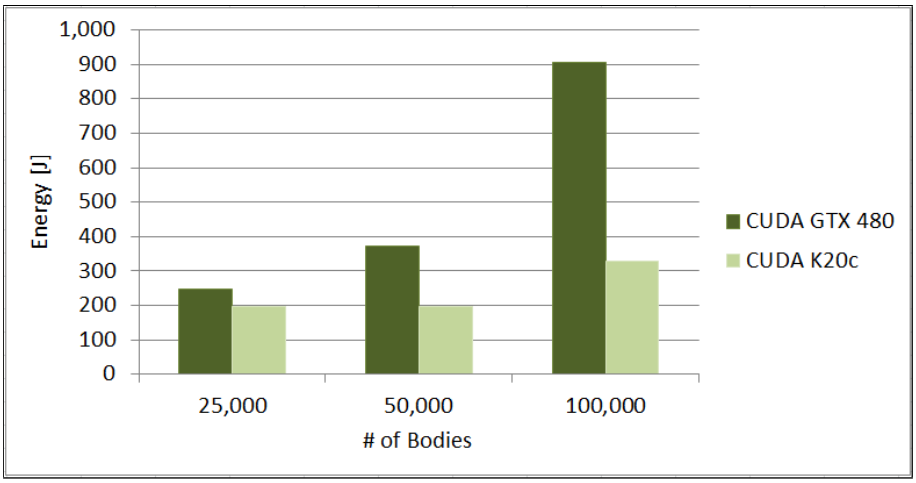

Figure 2.12: Energy consumption of NB_{CUDA} with 10 timesteps

To evaluate the benefit of using GPUs, we compare the GPU results to the results of the best-performing CPU code, which is the OpenMP version of BH and NB running on System 1 with hyper-threading. Figures 2.13 and 2.14 show the BH comparisons and Figures 2.15 and 2.16 show the NB comparisons.

The GPU codes are tremendously faster and much more energy efficient than the multi-core CPU codes on all inputs. Comparing BH_{OMP} running on System 1 to BH_{CUDA} running on the K20c, we find the GPU code to be 31 times as fast for 250,000 bodies, 39 times as fast for 500,000 bodies, and 45 times as fast for 1,000,000 bodies. Moreover, the GPU code consumes only 4.4% of the energy for 250,000 bodies, 3.3% for 500,000 bodies, and 2.4% for 1,000,000 bodies. This amounts to almost two orders of magnitude in energy savings. Clearly, GPUs are not only great at accelerating code but also nearly as effective at saving energy.

The GPU outperforms the CPU by even larger factors on NB. However, this is likely in part due to poor code generation. In particular, the gcc compiler does not exploit vector

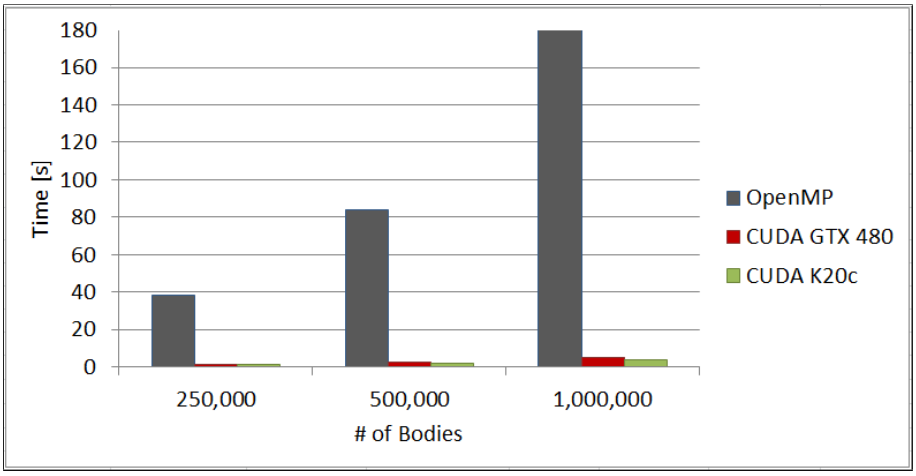

Figure 2.13: Runtime comparison between BH_{OMP} and BH_{CUDA} with 10 timesteps

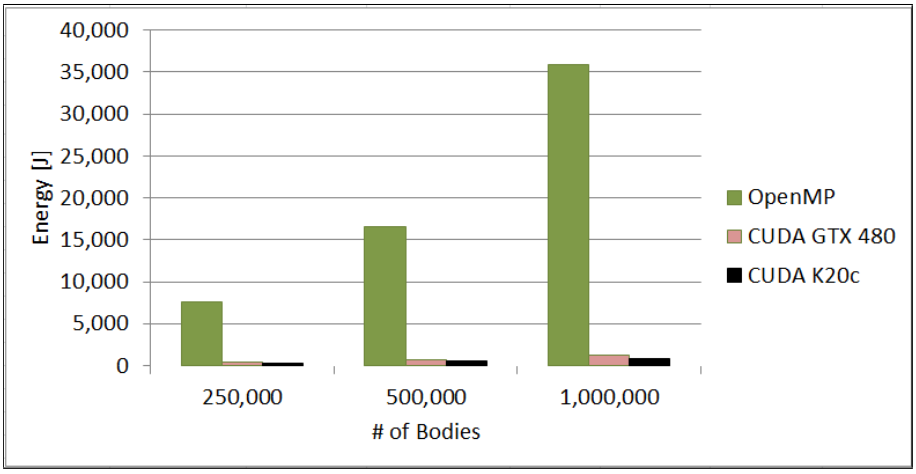

Figure 2.14: Energy consumption comparison between BH_{OMP} and BH_{CUDA} with 10 timesteps

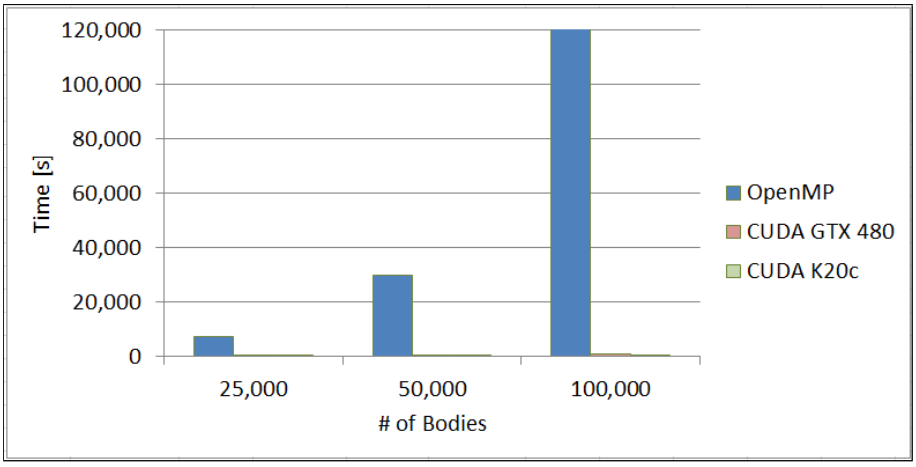

Figure 2.15: Runtime comparison between NB_{OMP} and NB_{CUDA} with 10 timesteps

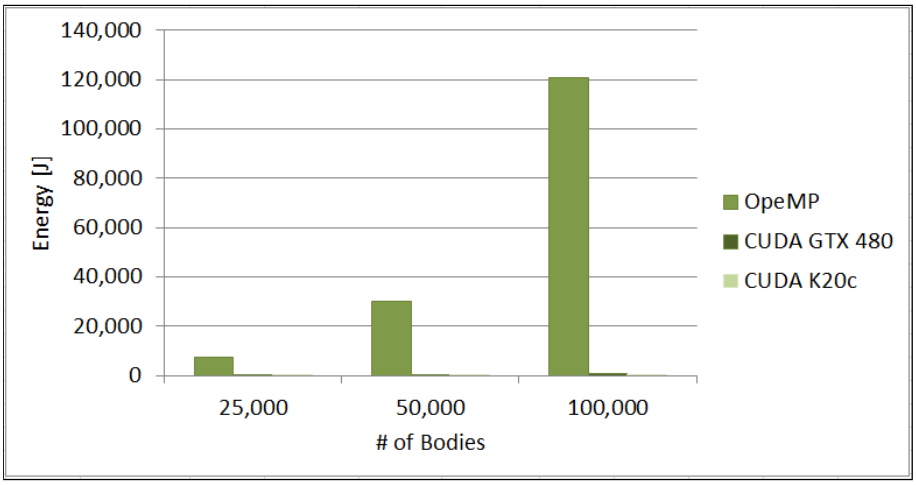

Figure 2.16: Energy consumption comparison between NB_{OMP} and NB_{CUDA} with 10 timesteps

instructions for this code. Hence, we find the NB_{CUDA} code running on the K20c to be 79 times as fast as the NB_{OMP} code running on System 1 for 25,000 bodies, 225 times as fast for 50,000 bodies, and 424 times as fast for 100,000 bodies. The GPU code consumes only 2.7% of the energy for 25,000 bodies, 0.66% for 50,000 bodies, and 0.27% for 100,000 bodies. This amounts to close to three orders of magnitude in energy savings. In summary, our results indicate that GPUs can improve the performance of n-body codes by one to three orders of magnitude compared to multi-core CPUs. Their energy consumption is lower by almost the same factor, making GPU acceleration also a very promising and effective approach for improving the energy efficiency of n-body codes.

2.6 Conclusion

This chapter studied the energy consumption and performance of five n-body codes running on two systems and two GPUs. The first system is based on a 2.4 GHz quad-core Intel Xeon E5620 CPU that supports 2-way hyper-threading. The second system contains two 2.5 GHz quad-core AMD Opteron 2380 CPUs. The two NVIDIA GPUs are a 1.4 GHz GeForce GTX 480 with 480 processing elements and a 0.7 GHz Tesla K20c with 2496 processing elements. Two of the five n-body codes we study implement an $O(n^2)$ algorithm and the remaining three programs implement the $O(nlogn)$ Barnes-Hut algorithm. The former category includes an OpenMP and a CUDA version whereas the latter category includes a P-Threads, an OpenMP and a CUDA version. Each code is tested on 3 inputs. The studied codes scale well on both multi-core CPUs, in-cluding the complex Barnes-Hut implementation. However, there are differences in the scaling of the

programs. Unexpectedly, we found some n-body codes to scale better on one system than on the other while other codes scale nearly equally on both systems. Hardly surprisingly, the OpenMP implementation that parallelizes every step of the algorithm is faster than the P-Threads implementation that only parallelizes one step. In general, the CPU power draw is relatively small compared to the system idle power, which makes the energy consumption of a program highly dependent on the runtime. As a consequence, any reduction in runtime results in nearly proportional savings in energy. This is why program parallelization is so important to achieving high energy efficiency. It also explains why our system with the lower power draw is less efficient as it is much slower. We conclude that shortening a program's runtime is paramount to improve its energy efficiency. Hence, it is crucial to parallelize the entire application and to utilize all cores. After all, we found very small speedups due to parallelization to be sufficient to improve the energy efficiency. Once all cores are used, running multiple threads per core on systems that support hyper-threading can result in significant additional improvements. Interestingly, we found hyper-threading to only help on the memory-bound code we studied but not on the compute-bound code. Nevertheless, since it does not appear to hurt, we recommend using hyper-threading when available. However, running too many threads can be detrimental to performance and energy efficiency. Thus, care must be taken to avoid oversubscribing threads to cores. Even though the power drawn by the GPUs is quite high (on the order of the system's idle power) and much higher than that of the CPUs, the GPUs are so much faster that they turn out to be very energy efficient. In fact, the GPU-accelerated n-body implementations we investigated consume two to three orders of magnitude less energy than the multi-core CPU codes. Whereas the current-generation

K20c GPU typically but not always outperforms the previous-generation GTX 480 GPU, the K20c appears to generally consume less energy, in particular on compute-bound code. Overall, we found GPUs to be great at speeding up our programs and nearly as effective at saving energy, making GPU acceleration the number one recommendation for improving energy efficiency.

CHAPTER III

ACCURATE ENERGY MEASUREMENT OF CODE RUNNING ON GPUS

3.1 Introduction

The HPC landscape has changed substantially with the introduction of Fermi-based compute GPUs [NVIDIA, 2012a], which now dominate the world's most powerful supercomputers [Meuer, 2012]. The recently released Kepler-based GPUs [NVIDIA, 2012b] will further improve the performance and energy-efficiency of upcoming HPC systems. Hence, it is likely that the number of computers with GPUs and the number of GPU-accelerated applications will increase rapidly over the coming years.

Titan, the currently fastest supercomputer in the Top500 list, contains nearly 300,000 CPU cores and over 50 million GPU cores. Undoubtedly, Titan and similar emerging systems will draw large amounts of power. However, to optimize the energy-efficiency of GPU-accelerated code, we need to first develop a good understanding about the energy-consumption behavior of programs running on GPUs, which requires accurate power measurement and profiling.

To make real-time power information available, the latest GPUs (e.g., the Tesla K20) have on-board sensors for profiling the power consumption at runtime. In contrast, most existing GPUs (e.g., the GeForce GTX 480) do not provide built-in sensors. Being able to obtain accurate power consumption data enables GPU programmers to evaluate

and improve the energy-efficiency of their code. However, the majority of published work in this area focuses on using models to estimate the power consumption [Choi, 2012; Hong, 2010; Isci, 2003; Nagasaka, 2010; Song, 2013], most likely because all but the latest generation of GPUs do not include power sensors. To the best of our knowledge, there is no published work that explains how to directly and correctly measure the power consumption of modern GPUs like the Tesla K20.

Such measurements are not as trivial as they might appear. In particular, the straightforward approach of sampling the power, computing the average, and multiplying by the runtime often results in large errors. This is because of the underlying assumption that all energy is consumed during program execution, that the sampling frequency is sufficiently high, that the power draw is independent of previous activities, and that the sampling intervals are equal. However, we show that these assumptions are typically wrong.

This chapter makes the following contributions:

- We discuss and explain a number of unexpected behaviors when measuring the energy consumed by GPUs.
- We provide guidelines for programmers on how to obtain accurate energy consumption data using built-in power sensors (if available) or external power meters.
- We make our GPU energy-measurement tool, which has been tested on the Stampede supercomputer at TACC, publicly available at http://cs.txstate.edu/ecl.
- We show that, using our proposed methodology, the on-board GPU sensors and the

external power meter result in nearly identical energy measurements.

The remainder of the chapter is organized as follows. Section 3.2 discusses related work. Section 3.3 describes our test bed. Section 3.4 presents common pitfalls and fallacies. Section 3.5 explains our proposed solution. Section 3.6 summarizes our findings and draws conclusions.

3.2 Related Work

The energy consumption of computer components can be obtained either directly or indirectly. The basic idea of indirect measurement is to estimate the power consumption using a power model, which correlates the power consumption with hardware performance-counter measurements or other events. This approach has already been widely used for estimating the power consumption of CPUs before GPGPUs became available [Contreras, 2005; Isci, 2003]. Two widely-used CPU power models are Wattch [Brooks, 2000] for single-core CPUs and McPAT [Li, 2009] for multi-core CPUs. Research on developing power models for GPUs is still in an early stage. Most of the GPU power models that utilize performance counters rely on statistics to correlate power to performance [Chen, 2011; Choi, 2012; Ghosh, 2009, 2013; Lal, 2013; Leng, 2013; Lucas, 2013; Ma, 2009; Song, 2013]. Hong and Kim proposed an integrated power and performance prediction model for a GPU architecture to predict the optimal number of active processors for a given application [Hong, 2010]. Song et al. proposed a GPU power model that is based on the training results of an artificial neural network [Song, 2013]. Choi and Vuduc proposed a roofline model to estimate the GPU energy

consumption [Choi, 2012]. Very recently, Sohan et al. released GPUSimPow [Lal, 2013; Lucas, 2013], a framework for modeling GPU power consumption, and Leng et al. developed GPUWattch [Leng, 2013], a configurable GPU power model based on cycle-level calculations. The advantage of using models to estimate power consumption is that a model can be deployed in a large-scale system with low cost. However, the parameters used in these models need to be retrained or reset for new hardware. In addition, the power estimation of these models may be relatively inaccurate. For example, GPUWattch differs by 9.9% to 13.4% from the power measured on actual hardware [Leng, 2013].

In contrast to indirect measurement using power models, direct measurement methods periodically collect samples of the current and the voltage of computer components. The power is computed by multiplying the current with the voltage, and the total energy consumption is calculated as the integral of the power over the execution time. Prior to the availability of built-in power sensors, external power meters had to be used to directly measure the power consumption. Several tools are available to directly measure the power draw of an entire node or of individual components like GPUs. The best known such tool is probably PowerPack [Ge, 2010], which was developed at Virginia Tech for System G, the world's largest power-aware cluster [Tech, 2008]. PowerPack is able to measure the power consumption of individual components (e.g., CPUs, DRAM, and disks). However, its hardware-software power profiling approach is expensive and difficult to implement. An alternative is the use of a power monitoring card such as PowerMon [Bedard, 2009, 2010] or a simple power meter like WattsUP [WattsUp, 2010]. WattsUP is easy to use but its maximum sampling frequency is rather low (1Hz).

Moreover, it can only measure the power consumption of the entire node. To obtain the power consumption of only the GPU, the base power of the node must be subtracted out. Recently released compute GPUs such as the Tesla C2075 (Fermi architecture [NVIDIA, 2012a]) and the Tesla K20 (Kepler architecture [NVIDIA, 2012b]) include on-board power sensors that allow us to directly measure the power consumption of the GPU card.

Table 3.1: GPU Description

Device	Core Speed	Memory Speed	Number of Cores	Idle Power
GeForce GTX 480	1.401 GHz	1.848 GHz	480	54 W
Tesla K20c	0.706 GHz	2.6 GHz	2,496	13 W

3.3 Benchmarks, System, and Energy Measurement

3.3.1 Benchmark Description

We use two GPU implementations of different n-body algorithms in this study to generate the experimental results. Both programs simulate the gravity-induced motion of stars (a.k.a. bodies) in a star cluster for a few time steps.

The first code, called NB, performs precise pair-wise force calculations. With n bodies, $O(n^2)$ interactions need to be considered, i.e., the computation is quadratic in the number of bodies. However, the same operations need to be performed for all n bodies, leading to a very regular implementation that maps well to GPUs. Moreover, the force

calculations are independent, resulting in large amounts of parallelism. Our NB code reaches over 2 teraflops on a single K20c GPU and exceeds the performance of the n-body code in the CUDA 5.0 SDK.

The second code, called BH, uses the Barnes-Hut algorithm to approximately compute the forces [Barnes, 1986]. This algorithm hierarchically partitions the volume around the n bodies into successively smaller cubes, called cells, until there is exactly one body per innermost cell. The resulting spatial hierarchy is recorded in an unbalanced octree. Each cell summarizes information about the bodies it contains. This hierarchy reduces the complexity to $O(n\ log\ n)$ because, for cells that are sufficiently far away, it suffices to perform only one force calculation with the cell instead of performing one calculation with every body inside the cell. We obtained the BH code from the LonestarGPU benchmark suite [Lonestar, 2010].

The NB code is relatively straightforward, has a high computational density, and only accesses main memory infrequently due to excellent caching. In contrast, the BH code is quite complex, has a low computational density, and performs mostly irregular pointer-chasing memory accesses. As a result, it 'only' reaches some 200 gigaflops. Nevertheless, because of its lower time complexity, it is about 33 times faster than the NB code when simulating one million stars.

3.3.2 System Description

Our system is based on a 2.4GHz Xeon E5620 CPU with 4 cores running 32-bit CentOS 5.9. Table 3.1 provides information about the two high-end GPUs it contains. The Fermi-based GeForce GTX 480 has 1.5GB of global memory and 15 streaming

multiprocessors (SMs) with 480 processing elements (PEs) running at 1.4GHz. The Kepler-based Tesla K20c has 5GB of global memory and 13 SMs with 2496 PEs running at 0.7GHz. The idle power of the GTX 480 is 54W, the K20c draws 13W when idling (15W when querying the internal power sensor at 100Hz), and the entire system consumes 165W in idle mode. Whereas compute nodes typically do not contain multiple GPUs of different type, this setup allows us to perform a broader range of measurements.

We compiled our CUDA codes with nvcc 5.0 and the '-O3 -arch=sm_20 -ftz=true' flags for the GTX 480 and the '-O3 -arch=sm_35 -ftz=true' flags for the K20c. Depending on the experiment, we ran the programs with either 500,000 or 1 million stars (bodies) and either 1 or 33 time steps. The stars' positions and velocities are initialized according to the empirical Plummer model [Plummer, 1911], which mimics the density distribution of globular clusters.

3.3.3 Energy Measurement

We employ two distinct approaches to obtain the energy consumption of GPU applications. The first approach is to use the WattsUp power meter [WattsUp, 2010]. It runs a background process to regularly record the power consumption. The meter has a 1Hz sampling frequency and a power resolution of 0.1W. The energy is computed from the readings by integrating the power as explained in Section 3.4. The second approach is to use the GPU's power sensor. The K20c comes with a sensor that captures the power consumption of the GPU, its memory, and everything else on the GPU card. We wrote our own software tool to query the sensor via the NVIDIA Management Library (NVML) interface [NVIDIA, 2011]. The tool uses a sampling frequency of about 100Hz and a

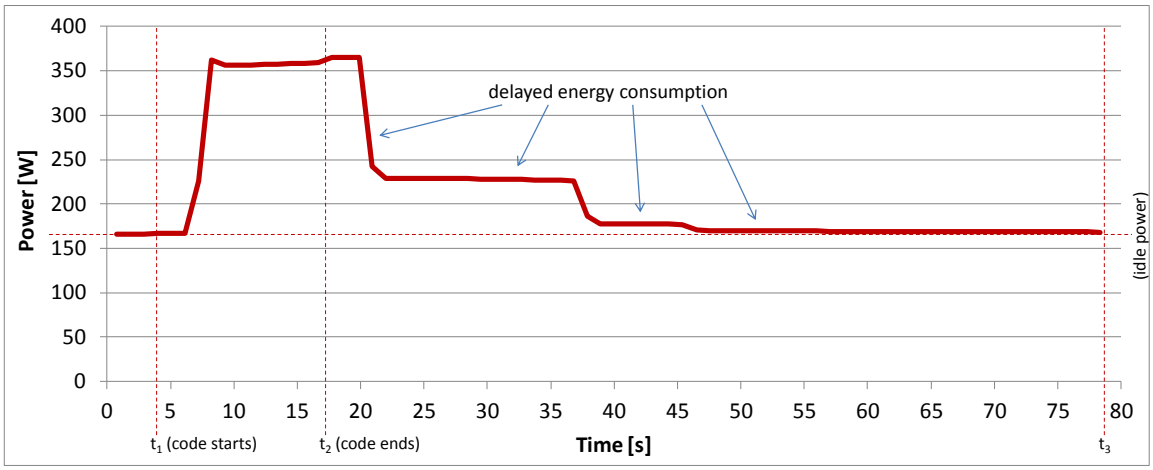

Figure 3.1: Node power profile showing delayed energy consumption when running BH on the GTX 480

resolution of 1W, and it computes the energy consumption by integrating the power readings. Note that the sampling activity incurs almost a 2W power overhead.

3.4 Pitfalls and Fallacies

In this section, we discuss possible pitfalls when measuring the energy consumption of GPU code. Section 3.6 describes our proposed methodology to avoid these pitfalls.

Pitfall #1: Measuring the power consumption only during program execution

A common assumption is that programs do not consume energy after they finish running. Based on this assumption, measurements might be stopped when the execution completes. However, GPU code generally consumes more energy than would be measured with this approach because of what we refer to as the *delayed* energy consumption.

Figure 3.1 depicts the node-wide power draw a little before, during (from time t_1 to t_2), and after running BH with 1M bodies for 33 time steps on the GTX 480. Clearly, the

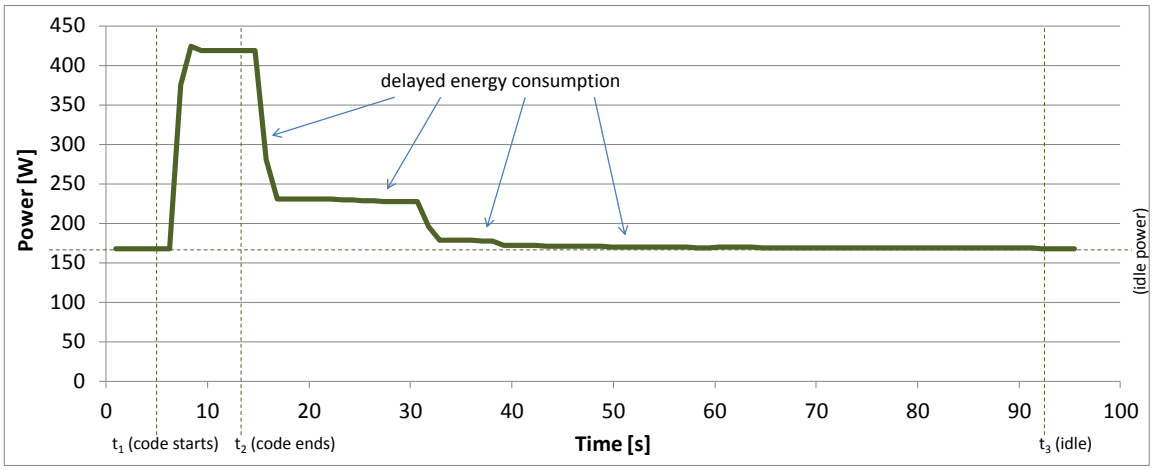

Figure 3.2: Node power profile showing delayed energy consumption when running NB on the GTX 480

program execution causes a substantial amount of energy consumption after the code has finished. In fact, it takes 62s from the program's termination until the node returns to its near-idle power draw (at time t_3). Only measuring the energy consumed during program execution yields 1873J (without the idle power contribution) whereas including the delayed energy yields 2947J. In other words, without accounting for the delayed energy, merely 64% of the computation energy consumed by the code is captured. Figure 3.2 shows the same effect on the NB code with 500,000 bodies and 1 time step. In this case, not counting the delayed energy yields 1568J, which is only 61% of the 2563J we obtain when including the delayed energy.

We believe the delayed energy consumption to be caused by capacitors, which are widely-used electrical components for storing and releasing energy to smoothen out rapid power changes. As shown in Figure 3.3, capacitor charging (and discharging) takes time. Hence, applications may still consume energy after they have stopped running because the

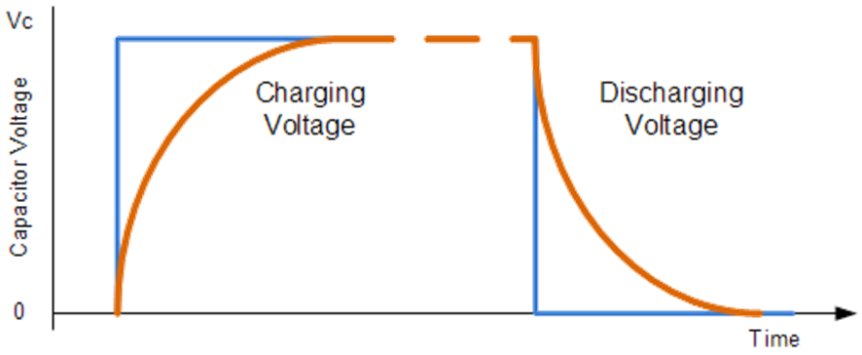

Figure 3.3: Capacitor charge/discharge behavior

capacitors that provided some of the energy during program execution need to be recharged. Note that Figure 3.3 is for illustration purposes only. The capacitors in a GPU may follow a different charge/discharge curve.

Figure 3.4 shows the GPU-only power profile of BH with 1M bodies and 33 time steps measured by the K20c's on-board sensor, which makes the discharging and recharging more obvious as it does not include any power consumption due to CPU activities. In fact, the power curve follows the typical capacitor behavior very closely. After an initial near-linear power increase during which memory is allocated and initialized on the GPU, the GPU starts computing and the power draw rapidly increases as less and less energy is provided by the capacitors. Once the capacitors have largely been discharged, we see a relatively stable power draw for a while. The small 'waves' are due to the 33 time steps. As soon as the GPU computation terminates, the power draw starts dropping. However, it does not drop instantly as the capacitors first have to be recharged. After the power consumption has leveled off at about 50W, it drops in distinct steps down to the idle level. We believe these steps are due to the GPU powering down components

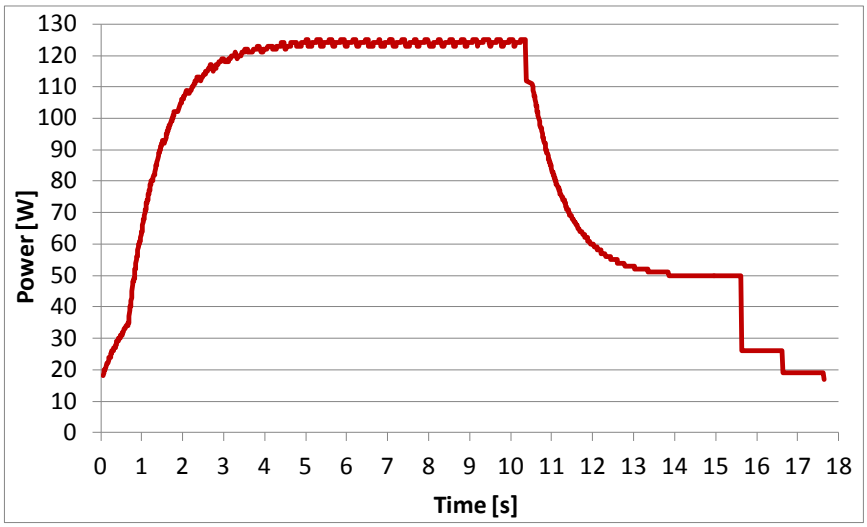

Figure 3.4: K20c power profile showing charging and discharging behavior on BH

after they have not been used for a certain period of time.

To summarize, the energy consumption of a GPU does not end instantly when the code terminates. Rather, the power draw is delayed due to capacitors, making it necessary to keep measuring past the end of program execution.

Pitfall #2: Continuing the power measurements for a fixed time past the program's termination

Recognizing that there is a delay in energy consumption, one might continue the measurement for a constant amount of time past the end of the program execution to capture this energy. However, the time it takes to return to the idle power level, which we call the *recovery* time, depends on the type of GPU as well as on how long the GPU was active.

Figure 3.5 shows the recovery time of our two GPUs when running the BH code with 500,000 bodies and 33 time steps. We start the power measurements 4s before the

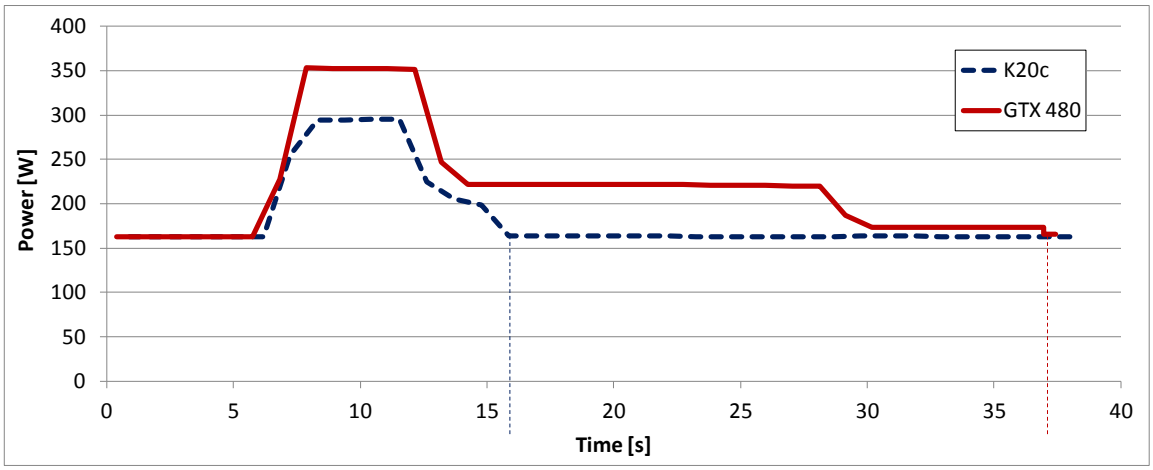

Figure 3.5: Recovery times for BH with 500,000 bodies and 33 time steps

program is launched. The GPU code runs for 4.7s on the K20c and for 6.4s on the GTX 480. Even though these runtimes only differ by 36%, the K20c returns to its near-idle power consumption after 7.1s whereas the GTX 480 takes 26.6s, a difference of 375%. Figure 3.6 shows the results of the same experiment but with 1 million bodies. Due to the larger input, the execution times increase to 9.7s on the K20c and 13.1s on the GTX 480. As a consequence, the recovery times also become longer for both GPUs. It now takes the K20c 11.9s and the GTX 480 61.3s. Again, there is a big difference between the two GPUs. In this case, it takes almost 50s longer for the GTX 480 to recover. The reason for this difference is primarily that the previous-generation GTX 480 consumes more energy than the current-generation K20c when running the same task.

In summary, these results show that the GPU architecture and the application runtime can substantially affect the power-consumption behavior of a program after (and during) execution. Hence, the measurement time past the end of the program execution may have to be dynamically adapted.

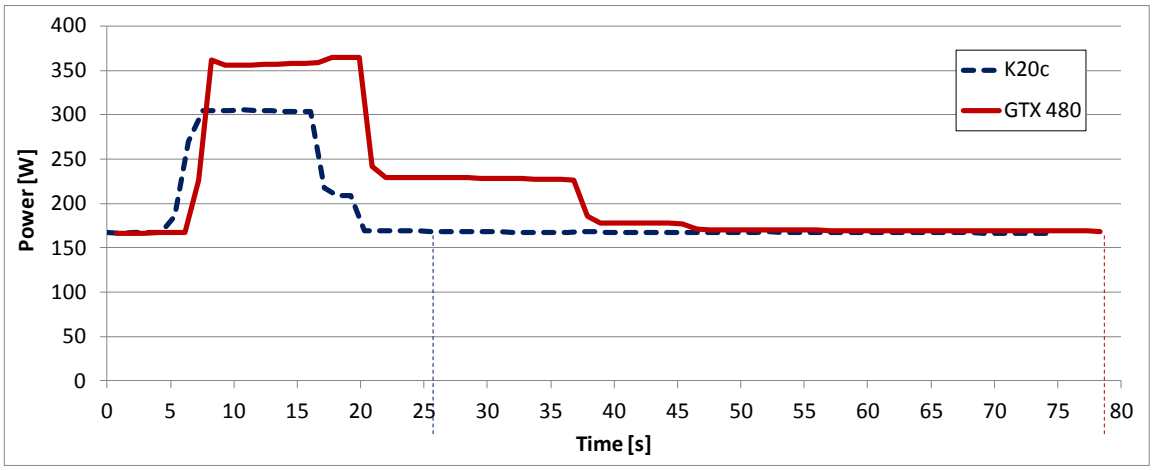

Figure 3.6: Recovery times for BH with 1 million bodies and 33 time steps

Pitfall #3: Not distinguishing between 'cold' and 'warm' runs on a GPU

When a program is launched on a node whose power consumption is at the idle level, we call it a *cold* run. If a program is launched on a node that has just been busy and has not yet returned to its idle power, we call it a *warm* run.

A pitfall in energy measurement is to assume that cold and warm runs of the same program consume the same amount of energy. While this may even be true, the problem with warm runs is that the delayed energy consumption of the previous activity overlaps, by definition, with the current application's execution. To illustrate the resulting apparent difference in energy consumption this can cause, we measured a cold and a warm run of the BH code on the GTX 480, both times with 1 million bodies and 33 time steps. For the cold run, we waited for the GPU to return to its idle power before launching the code. For the warm run, we first ran the BH code five times in a row before measuring the immediately following sixth run. Figure 3.7 displays the resulting power profiles.

The profiles show that there is, indeed, a substantial difference in the measured

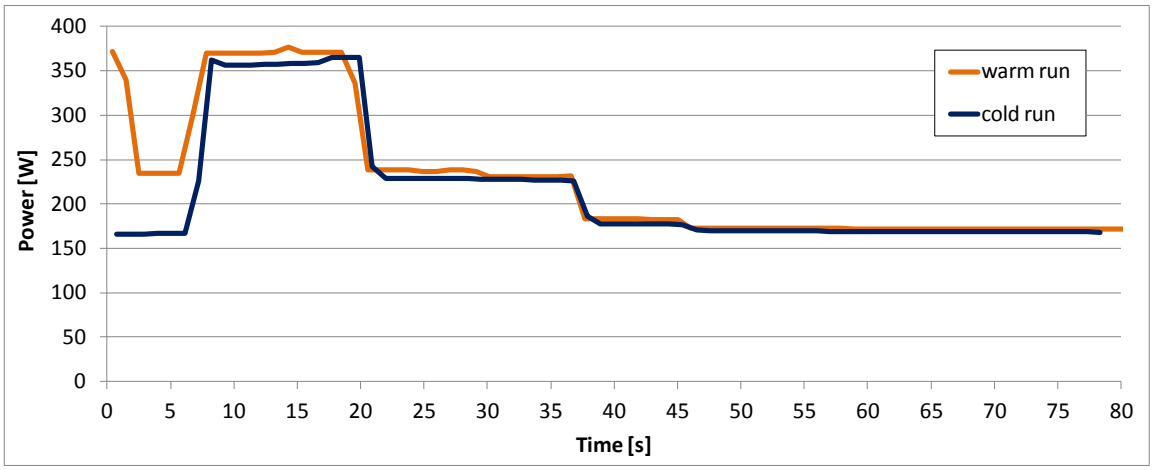

Figure 3.7: Difference in power consumption between a ‘cold’ and a ‘warm’ run of BH on the GTX 480

energy between the two runs. In fact, the warm run seems to consume 25% more energy than the cold run (not counting the idle energy). This clearly demonstrates that the state of the GPU (warm or cold) plays an important role when determining the energy consumption of a GPU application. Note that warm GPU runs are common on HPC clusters and supercomputers, which strive to maximize the system utilization through job scheduling.

In summary, warm runs make it difficult to determine the true energy consumption because the measured program run overlaps with the delayed energy consumption of the previously executing code, which artificially inflates the power readings. Since there is no way to distinguish between the power contributions of the previous and the current application, the only solution is to postpone the current application’s execution until the power has returned to the idle level as only cold runs yield accurate energy measurements.

Fallacy #4: Not being able to measure the GPU's energy consumption for lack of a built-in sensor

Whereas the latest compute GPUs come with on-board power sensors, the vast majority of GPUs do not yet include such sensors. Nevertheless, it would be wrong to assume that the absence of a built-in power sensor makes accurate measurements of GPU energy consumption overly difficult or unachievable. In fact, our experimental results show that the energy consumption of an application running on a GPU can be accurately obtained using a simple external power meter like WattsUP that measures the entire node's power.

Table 3.2: Median energy obtained using the K20c's on-board sensor and the WattsUp power meter

Application	**Runtime [s]**	**Energy [J]**
Meter: BH (1M bodies)	9.69	1272.86
Sensor: BH (1M bodies)	9.69	1248.40
Meter: NB (1M bodies)	9.59	1724.59
Sensor: NB (1M bodies)	9.58	1655.75

Table 3.2 compares the energy of BH and NB measured with the GPU sensor and with WattsUp following the methodology described in Section 3.5. The two methods yield very similar results. For BH, the difference is 1.96% and for NB, it is 4.16%. These results not only indicate a strong correlation between the two methods but also independently verify the amount of energy consumed by these codes.

Although the external power meter suffices to measure our code's energy consumption, it is unable to capture power changes at a fine granularity, as is evident when comparing the graphs in Figures 3.4 and 3.6. The former is changing much more smoothly than the latter, primarily because of the difference in sampling frequency. The lower this frequency is, the more likely it is that important power changes are missed. Figure 3.8 shows how the computed BH energy consumption depends on the sampling frequency of the K20c's sensor. To make these results independent of the power consumption of the sampling itself, which varies from 0.0W below 2Hz to 1.62W at 100Hz, we always sample at 100Hz but do not use all sampled values. As we can see, 1Hz is the minimum frequency required to obtain an accurate result (within 5%) for this code and input. Below 1Hz, the result quickly becomes useless. The reason for this behavior is the runtime of approximately ten seconds, which amounts to ten samples at 1Hz. For a relatively smooth power curve, such as the one of the BH code, one can expect the error to be around half a sample or about 5% at 1Hz, which is in line with our experimental results. Note, this means that even at 100Hz the kernel runtime should be at least 100ms to obtain accurate energy numbers.

To summarize, it is possible to accurately measure the energy consumption of GPUs without built-in power sensors using a simple external power meter like WattsUP. However, care must be taken to only measure programs whose running time is long enough to span many power samples. This is much more of a concern with WattsUp's relatively low sampling frequency of 1Hz than with the K20c sensor's sampling frequency of about 100Hz. Moreover, the rest of the node should be mostly idle so as not to perturb the measurements.

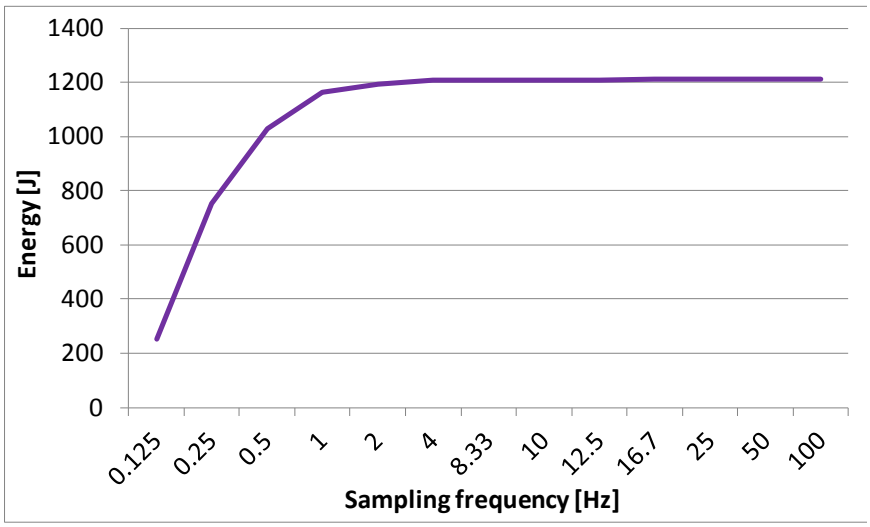

Figure 3.8: Computed BH energy consumption on the K20c as a function of the power sampling frequency

Pitfall #5: Assuming a constant sampling frequency with the built-in power sensor

When sampling the power consumption at a constant rate, one can easily obtain the energy by summing up the sampled values, dividing by the number of samples (i.e., computing the average power), and multiplying by the time over which the samples were acquired. However, this approach may lead to incorrect results when using the GPU's built-in sensor.

The problem is that the code reading the power values executes on the CPU whereas the sensor resides on the GPU. Hence, the measurements must be transmitted over the PCI Express bus from the GPU card to the CPU. But the same bus is also used to transfer the program's data to and from the GPU as well as other information such as kernel invocations. Whenever the bus is busy with such a transfer, it is not available for power

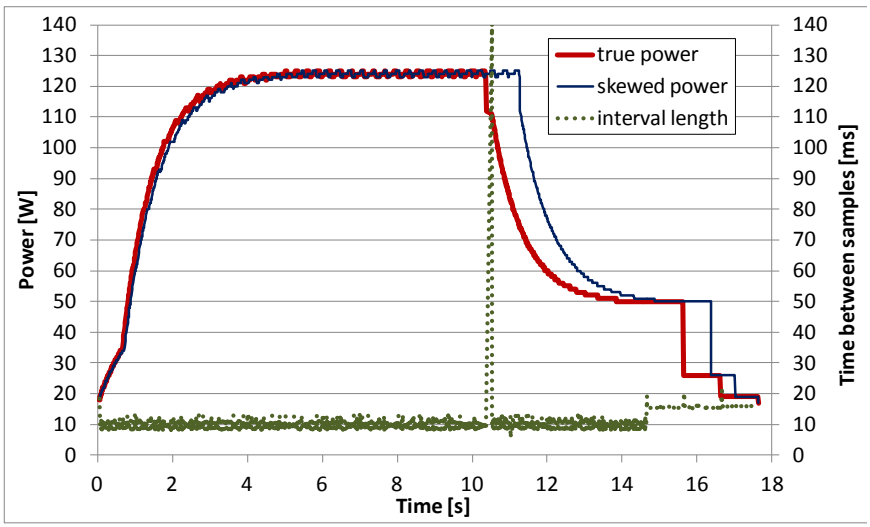

Figure 3.9: BH energy consumption on the K20c with and without assuming a constant sampling rate as well as the variability of the sampling intervals

readings. It is therefore likely that the sampling intervals will vary depending on bus activity.

Figure 3.9 illustrates the problem on the example of BH with 1M bodies and 33 time steps. The bold solid curve shows the power consumption when taking the variability in the sampling intervals into account. In contrast, the thin solid curve shows the same power readings but displayed at equidistant intervals. Clearly, the two curves do not overlap well, and the thin curve results in a 2.8% larger energy consumption.

To point out the cause of this discrepancy, Figure 3.9 also displays the actual sampling interval lengths (i.e., the dots). There is a huge spike at about 10.5s, which occurs immediately after the GPU code is done computing and coincides with the transfer of the computed data back to the CPU. Whereas most intervals are close to 10ms, which corresponds to a 100Hz sampling frequency, the longest interval is 140.4ms,

corresponding to a sampling frequency of only 7.1Hz. There is also a significant spike due to a data transfer just before the GPU code starts executing. Finally, there are spikes whenever the GPU switches power levels. Interestingly, the sampling intervals are longer (about 16ms or 63Hz) when the GPU is in a low-power state.

In summary, PCI Express bus activity affects the frequency at which the GPU's power sensor can be queried. Not accounting for the resulting fluctuations in the sampling interval lengths yields an error of 2.8% in the energy consumption of our code. The error is small because BH does not transfer much data. However, it could be much higher for applications that send large amounts of data to/from the GPU. Hence, power sensor samples should include time stamps so that the energy consumption can be computed accurately.

3.5 Proposed Methodology

Based on the observations described in the previous section, we designed the following energy-measurement methodology for GPU code that avoids all mentioned pitfalls. Figure 3.10 visualizes the key aspects of this methodology.

3.5.1 When to Start

We recommend always initiating the power measurements a few seconds before launching the GPU code (at time t_1 in Figure 3.10) to ensure that the measurement code is running smoothly and that the system has settled into a steady state. We use a fixed 4s head delay for this purpose. However, we found the power readings to stabilize in less than 4s, so a shorter head delay would also work on our system.

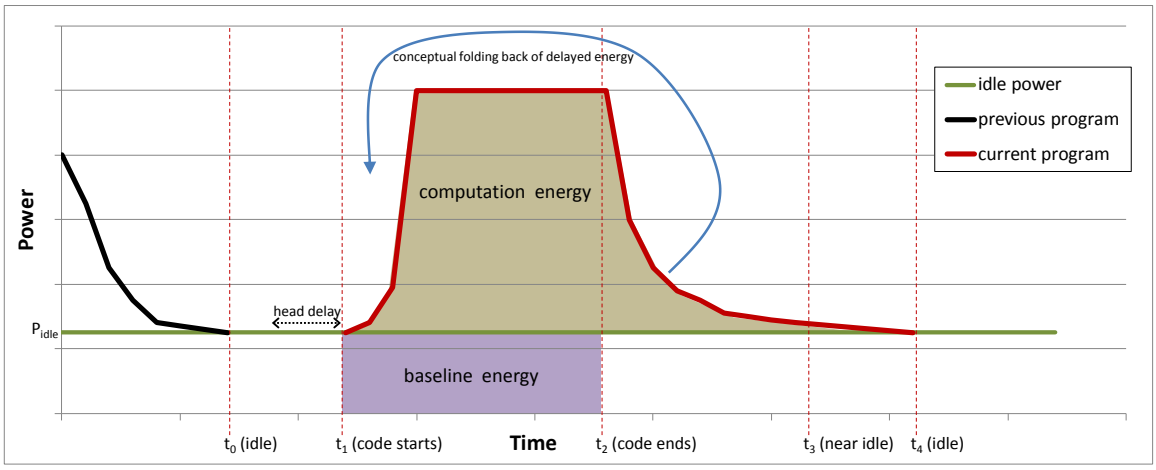

Figure 3.10: Proposed energy-measurement methodology for compute GPUs

More importantly, the GPU code should only be launched when the power draw is stable and at the idle level (after time t_0). If the idle power P_{idle} is known, as is the case in our experiments, the launcher script can simply wait for that power level to be reached and maintained for the duration of the head delay before starting the GPU code. If the idle power is unknown, the script might sample the power at, say, 30s intervals until it remains stable, record that power level, and then launch the GPU code.

3.5.2 How to Measure

Since CPUs typically run various activities (e.g., interrupt handlers, daemons, and other background jobs), it is advisable to repeat the energy measurements. For example, the results we present in this paper are the medians of five runs. Dropping outlier measurements in this or a similar fashion is important to obtain accurate energy consumption results in the presence of possible other system activities.

Another important factor is to measure a sufficiently long-running program.

Because the power is only sampled at discrete points in time, care must be taken that enough samples are obtained to capture the shape of the power curve at a fine enough granularity. Our results indicate that, for a relatively smooth power curve, at least ten samples are necessary to compute the energy consumption accurately. How long the code has to run to reach ten samples depends on the sampling frequency. For our external power meter with a 1Hz sampling frequency, a runtime of at least 10s is needed, which is approximately how long our codes run.

If the best available sampling rate is too low, it may be possible to improve the accuracy of the energy measurement by running the same kernel multiple times in a row or by using a larger program input to increase the running time. Of course, the resulting energy has to be scaled appropriately to obtain the energy consumption of a single run or for a smaller input.

When using the GPU's built-in sensor for measuring the power, every sample should be accompanied by a time stamp from a high-resolution (millisecond or better) timer. This is particularly important for applications that transfer large amounts of data to or from the GPU, which affects the sampling rate. By properly accounting for the difference in interval length when integrating the power readings, the energy consumption can be computed quite accurately even if the sampling rate fluctuates significantly.

3.5.3 When to Stop

As there can be a substantial amount of delayed energy consumption, it is paramount to continue the power measurements past the end of the GPU code's execution. To capture all the energy, the measurements have to continue until the power draw returns to the idle

level (time t_4 in Figure 3.10).

The delayed energy often exhibits a long tail that is just above the idle power (cf. Figure 3.1). To speed up the measurements without introducing a significant error, we stop taking power samples once the power has returned to a *near-idle* level (at time t_3), which is a small amount above the idle power. We use a 2W threshold for this purpose, which we verified to produce less than a 2% error on our codes.

3.5.4 How to Calculate the Energy

As mentioned, we either need to already know the idle power or establish it. Moreover, we have to time the execution on the GPU. This can either be done by querying the state of the GPU when reading the power samples from the built-in sensor or by instrumenting the source code, that is, reading a timer just before and just after the kernel call to compute the runtime. Note that GPU kernel calls are asynchronous, so extra code may be needed to wait for the GPU to finish computing before stopping the timer.

Once we have obtained the idle power, the GPU runtime, and the power samples from before, during, and after the program run on the GPU, we can compute the energy consumption as follows. First, we compute the 'baseline' energy, which is the idle power times the kernel runtime (without the head or tail delay). Second, we calculate the 'computation' energy by integrating the power *above idle* from the moment the power increases (at or after time t_1) until it drops back down to the idle level (at time t_3 or t_4). The total energy consumption is simply the sum of the baseline energy and the computation energy. Note, very importantly, that the computation energy is measured over a different time span than the baseline energy.

This methodology essentially folds the delayed energy back into the time span during which the GPU is actually running the code, thus giving the correct amount of energy consumed. In particular, it will yield twice the energy consumption when the same GPU code is executed twice in a row, as it should, irrespective of how far apart the two calls happen. If they are back to back, the second execution will be a warm run that is distorted because it includes some of the delayed energy of the first run. But in this case we want to include the delayed energy of both runs, so this is correct. If the two calls happen far apart, they will result in two cold runs, which also yield the correct total energy when added. Similarly, when increasing the input size and therefore the runtime, our methodology will correctly capture the disproportionate change in the delayed energy. Finally, for very long-running GPU codes, the tail energy becomes insignificant, causing our approach to result in the same energy consumption as the straightforward approach.

Our methodology works with the built-in power sensor as well as with an external power meter. However, the idle power is, of course, different. Moreover, the external meter will also measure all CPU, disk, network etc. activity. So, in general, one can only measure the GPU energy consumption with the GPU's sensor and the whole system energy consumption with the external power meter. However, our codes require very little CPU activity, which is why almost all of the computation energy consumption is due to the GPU, allowing us to accurately measure the GPU energy with an external power meter.

3.5.5 How the Pitfalls are Avoided

The above methodology addresses all of the pitfalls and fallacies we identified in the previous section. Pitfall #1 is avoided because we continue measuring the power

consumption after the GPU code stops executing. Pitfall #2 is avoided by taking power samples until the power returns to its near-idle level. Pitfall #3 is addressed by only launching the GPU code after the system has 'cooled' down from its prior activities. Fallacy #4 is addressed by computing the energy as described above and running the GPU code multiple times or on a larger input if necessary to make the runtime sufficiently long to obtain enough power samples. Finally, Pitfall #5 is avoided by including time stamps with each power sample and accounting for the difference in the sampling interval lengths when computing the energy.

3.6 Conclusion

Compute nodes can consume large amounts of energy, even when using accelerators such as GPUs. To enable programmers to study and reduce the energy consumption of their codes, accurate energy measurements are needed. However, the simplistic approach of sampling the power, computing the average, and multiplying by the running time often results in large errors. This is because the simplistic approach assumes that all energy is consumed during the program's execution, that the sampling frequency is sufficiently high, that the sampling intervals are equal, and that the power draw is independent of previous system activities.

Our study demonstrates that all of these assumptions are wrong. In actuality, the energy consumption is significantly delayed relative to the program execution due to capacitances in the hardware. If the GPU has recently been computing, this delay will cause the power measurements of the currently running code to be inflated. Moreover, the

sampling frequency at which the GPU's internal power sensor can be accessed fluctuates substantially because the power readings have to compete for the PCI Express bus, which is also used for transferring data to and from the GPU. Finally, many samples are needed to accurately reflect the power consumption, i.e., the inverse of the sampling frequency must be much shorter than the program's execution time. This may not be the case with an external 1Hz power meter because, in practice, many GPU kernels do not run for tens of seconds. Finally, we found different GPU architectures to behave quite differently, i.e., the same code results in different power profiles.

This chapter proposes an energy measurement methodology that is accurate in spite of the above mentioned problems. It correctly accounts for the delayed energy consumption, it waits until the GPU has cooled down before starting the measurements, it time stamps the samples to handle variations in sampling frequency, it provides guidelines on how long the program runtime should minimally be, and it automatically adapts to the GPU used. We verified that our methodology gives the same results with an external power meter as it does with the GPU's built-in power sensor.

CHAPTER IV

CONCLUSION

In this study, we have performed an extensive analysis on the energy efficiency of a wide range of commonly-used software algorithms executed on several hardware devices. Our results show that not only can software optmizations help conserve energy in HPC systems, but utilizing graphics cards can also play an important role in reducing the energy consumption of these applications. Our work has also provided a methodology for successfully and accurately measuring the energy consumed by novel algorithm implementations that make use of new computing devices such as GPUs. The use of graphics processing units is a trend that is revolutionizing the supercomputing industry by providing performance and energy efficiency that current CPU systems cannot match. It is because of this, that our research provides important insight and knowledge towards better understanding the energy consumption behavior of today's and tomorrow's high performance computing systems. Future work in this area is an exciting and promising direction. As we reach the exascale supercomputing era, the need for energy-efficient applications will become even more critical. Therefore, a broader analysis can be performed, one that covers a wider variety of systems and accelerators, including heterogeneous systems and the new Intel Xeon Phi. On the software side, we want to extend our work to distributed-memory implementations as well as other commonly used n-body algorithms and scientific applications in general.

BIBLIOGRAPHY

Appel, A. (1985). An efficient program for many-body simulation. In *SIAM J. Scientific and Statistical Computing*, volume 6.

Barnes, J., H. P. (1986). A hierarchical o(n log n) force-calculation algorithm. In *Nature*, volume 324.

Bedard, D., e. a. (2009). Powermon 2: Fine-grained, integrated power measurement. In *RENCI Technical Report TR-09-04, Renaissance Computing Institute*.

Bedard, D., e. a. (2010). Powermon: Fine-grained and integrated power monitoring for commodity computer systems. In *Proceedings of the IEEE Southeastcon Conference*.

Bedorf, J., e. a. (2012). A sparse octree gravitational n-body code that runs entirely on the gpu processor. In *Journal of Computitational Physics*, volume 231.

Breshears, C. (2009). O'Reilly Media, California, USA.

Brin, S. and Page, L. (2000). The anatomy of a large scale hypertextual web search engine. *http://infolab.stanford.edu/ backrub/google.html.*

Brooks, D., e. a. (2000). Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture(ISCA)*.

Bunse, C., e. a. (2009). Exploring the energy consumption of data sorting algorithms in embedded and mobile environments. In *Proceedings of the 10th International Conference on Mobile Data Management: Systems, Services and Middleware*.

Burtscher, M., P. K. (2011). An efficient cuda implementation of the tree-based barnes hut n-body algorithm. In *GPU Computing Gems Emerald Edition*, pages 75–92.

Chen, J., e. a. (2011). Statistical gpu power analysis using tree-based methods. In *Proceedings of the Green Computing Conference(IGCC)*.

Choi, J.W., V. R. (2012). A roofline model of energy. In *Technical Report GT-CSE-2012-01*, Georgia Tech.

Contreras, G., M. M. (2005). Power prediction for intel xscale processors using performance monitoring unit events. In *ISLPED*.

CUDA (2013). Cuda sdk. *https://developer.nvidia.com/cuda-toolkit*.

Douglis, P., e. a. (1995). Adaptive disk spin-down policies for mobile computers. In *Proceedings of MLICS*.

Duran, A. (2009). Tasking in openmp. *https://iwomp.zih.tu-dresden.de/downloads/omp30-tasks.pdf*.

EPA (2007). *Environmental Protection Agency: Report to Congress on Server and Data Center Energy Efficiency*, chapter Public Law 109-431. Public Law.

Ge, R., e. a. (2010). Powerpack: Energy profiling and analysis of high-performance systems and applications. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, volume 21, pages 658–671.

Ghosh, S., e. a. (2009). Statistical power and energy modeling of multi-gpu kernels. In *http://www2.cs.uh.edu/ hpctools/pub/SC12-Poster.pdf*.

Ghosh, S., e. a. (2013). Power and energy prediction of multi-gpu kernels using non-linear regression. In *GTC 2013 Poster*.

Greengard, L., R. V. (1987). A fast algorithm for particle simulations. In *Journal of Computitational Physics*, volume 73, pages 325–348.

Hong, S., K. H. (2010). An integrated gpu power and performance model. In *Proceedings of the 37th International Symposium on Computer architecture (ISCA)*.

Isci, C., M. M. (2003). Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

Jones, T., e. a. (2009). Linux os jitter measurements at large node counts using a blue gene/l. In *Oak Ridge National Laboratory*, volume 303.

Kataria, P. (2008). Parallel quicksort implementation using mpi and pthreads. *http://www.winlab.rutgers.edu/ pkataria/pdc.pdf*.

Lal, S., e. a. (2013). A framework for modeling gpus power consumption. In *LPGPU Workshop on Power-Efficient GPU and Many-core Computing in conjunction with the HiPEAC Conference*.

Lang, H.W., F. F. (2010). Sorting algorithms: Quicksort. *http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/quick /quicken.htm.*

Lefurgy, C., e. a. (2011). Energy-efficient data centers and systems. In *International Symposium on Workload Characterization*.

Leng, J., e. a. (2013). Gpuwattch: Enabling energy optimization in gpgpus. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

Li, S., e. a. (2009). Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

Liu, P.F., B. S. (2000). Experiences with parallel n-body simulation. In *IEEE Transactions on Parallel and Distributed Systems*, volume 11.

Lonestar (2009). P-threads barnes-hut implementation with uniform data distribution. *https://wiki.engr.illinois.edu/display/transformation/Barnes-Hut+TBB*.

Lonestar (2010). Barneshut gpu application. *http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu*.

Lucas, J., e. a. (2013). How a single chip causes massive power bills gpusimpow: A gpgpu power simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

M. Warren, J. S. (1993). A parallel hashed oct-tree n-body algorithm. In *Supercomputing*.

Ma, X., e. a. (2009). Statistical power consumption analysis and modeling for gpu-based computing. In *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*.

Malkowski, K., e. a. (2006). Toward a power efficient computer architecture for barnes-hut n-body simulations. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*.

Meuer, H., e. a. (2012). Top 500 supercomputer sites. *http://top500.org/lists/2012/11/*.

Nagasaka, H.N., e. a. (2010). Statistical power modeling of gpu kernels using performance counters. In *Proceedings of the Green Computing Conference*.

NVIDIA (2011). Nvidia management library. *http://developer.nvidia.com/nvidia-management-library-nvml*.

NVIDIA (2012a). Fermi compute architecture whitepaper. *http://www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf*.

NVIDIA (2012b). Kepler compute architecture whitepaper. *http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf*.

Pacheco, P. (2011). *Shared-Memory Programming With OpenMP*. MA: Morgan Kaufmann Publishers.

Plummer, H. (1911). On the problem of distribution in globular star clusters. In *Monthly Notices of the Royal Astronomical Society*, volume 71.

Salmon, J. (1990). *Parallel Hierarchical N-Body Methods*. PhD thesis, California Institute of Technology.

Song, S.W., e. a. (2013). A simplified and accurate model of power-performance efficiency on emergent gpu architectures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

Tech, V. (2008). System g. *http://www.cs.vt.edu/node/4666*.

Warren, M., e. a. (1997). Parallel supercomputing with commodity components. In *Proceedings of the International Conference in Parallel and Dristributed Processing Techniques and Applications*.

Warren, M., J. S. (1992). Astrophysical n-body simulations using hierarchical tree data structures. In *Supercomputing*.

WattsUp (2010). Wattsup power meter. *https://www.wattsupmeters.com/*.

Weiser, M., e. a. (1994). Scheduling for reduced cpu energy. In *Proceedings of OSDI*.

Xue, G. (1998). An o(n) time hierarchical tree algorithm for computing force field in n-body simulations. In *Theoretical Computer Science*, volume 197, pages 157–169.

Zeng, G., e. a. (2009). Analyzing and optimizing energy efficiency of algorithms on dvs systems a first step towards algorithmic energy minimization. In *Proceedings of the Asia and South Pacific Design Automation Conference*.

Zong, Z.L., e. a. (2011a). Ead and pebd: Two energy-aware duplication scheduling algorithms for parallel tasks on homogeneous clusters. In *IEEE Transactions on Computers*, volume 60, pages 360–374.

Zong, Z.L., e. a. (2011b). Heat-based dynamic data caching: A load balancing strategy for energy-efficient parallel storage systems with buffer disks. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST 2011)*.

VITA

Ivan Zecena was in born in Guatemala City, Guatemala, on February 27, 1988. He moved to the United States in 2006 and obtained his Bachelor's degree in Computer Science from Texas State University-San Marcos in Fall 2011. In 2012, he was accepted by the Broader Engagement program of the Supercomputing Conference (SC12) to attend SC12 in Salt Lake City, Utah. Ivan was also accepted by the 2012-2013 Extreme Science and Engineering Discovery Environment (XSEDE) program to attend XSEDE12 in Chicago, Illinois. He continued on at Texas State pursuing a Master's degree in Computer Science. During his graduate studies, Ivan was awarded the Graduate Research Excellence Award by the Texas State Computer Science department, and he won the Best Poster Award at the 2013 HSI Research Symposium at Texas State.

Permanent Address: 5607 Viewpoint Dr.
Austin, Texas 78744

This thesis was typed by Ivan Zecena.