UNDERSTANDING THE IMPACT OF HYBRID PROGRAMMING ON SOFTWARE ENERGY EFFICIENCY

by

Donna LaKomski, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
August 2016

Committee Members:

Ziliang Zong, Chair

Tongdan Jin, Co-Chair

Martin Burtscher

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplicate Permission

ACKNOWLEDGEMENTS

This thesis would not have been possible without my advisor Ziliang Zong. He has been instrumental in giving guidance and direction for the work done in this thesis. I would like to especially thank him for his patience.

I would also like to thank Tongdan Jin for his support, valuable feedback, and encouragement, and Martin Burtscher for sparking my interest in parallel computing.

Lastly, I want to thank my husband Greg LaKomski for his patience and support. I cherish his love and encouragement.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF ABBREVIATIONS

Abbreviation	Description
J	Joules
sec	second(s)

ABSTRACT

High performance computing systems today are heterogeneous in nature with multiple CPUs and accelerators/coprocessors in each computing node. The majority of today's programs only utilize single computing components (e.g. a CPU, GPU or Xeon Phi) while leaving other components idle (e.g. waiting for the results to be calculated). This may not be optimal for either performance or energy efficiency. Hybrid computing can solve this problem. Employing multiple device types can create more computing power on the platform, but can also create unexpected and unintended issues and challenges due to potential complex interactions of software and hardware. This thesis investigates the impact of hybrid computing on the performance and energy-efficiency of parallel applications, provides a guideline for hybrid work division, and develops a model to predict optimal performance or energy-efficiency.

1. INTRODUCTION

As tools for measuring power and energy of heterogeneous computing systems and parallel applications become available, the importance of measuring the energy efficiency of applications becomes both practical and paramount. For computer systems, the benefits of lower energy usage due to more multi-device-use applications include smaller-sized and lighter weight systems, and longer battery life. For high performance computing systems and data centers, hybrid computing can offer better energy efficiency, resulting in more computing performance per joule, lower energy and cooling costs.

A common practice today is to utilize the offload programming model (i.e. parallelize the code to it for execution). The major problem of this programming model is that the CPU remains idle, wasting computation cycles and energy, while waiting for the Xeon Phi or GPU to complete its work. For example, Figure 1.1 shows the percentage of CPU wasted energy for the applications without hybrid use from this thesis. The three sets of bars correspond to a CPU and coprocessor, CPU and GPU using CUDA, and CPU and GPU using OpenCL. Each set contains the applications: matrix multiplication (MM), fractal (F), breadth-first search (BFS), two different input graph sizes of 1 million nodes (1M) and 16 million nodes (16M) for breadth-first search, and where applicable two different compiling options Intel compiler (icc) and GNU compiler (gcc). The leftmost set of red bars show the CPU wastes from 15% to 20% of the total application energy while waiting for a Xeon Phi to complete the calculations. The set of pink bars display the percentage of energy the CPU wastes for the CPU and GPU (C++ and CUDA) calculations. The CPU in the fractal and breadth-first search applications

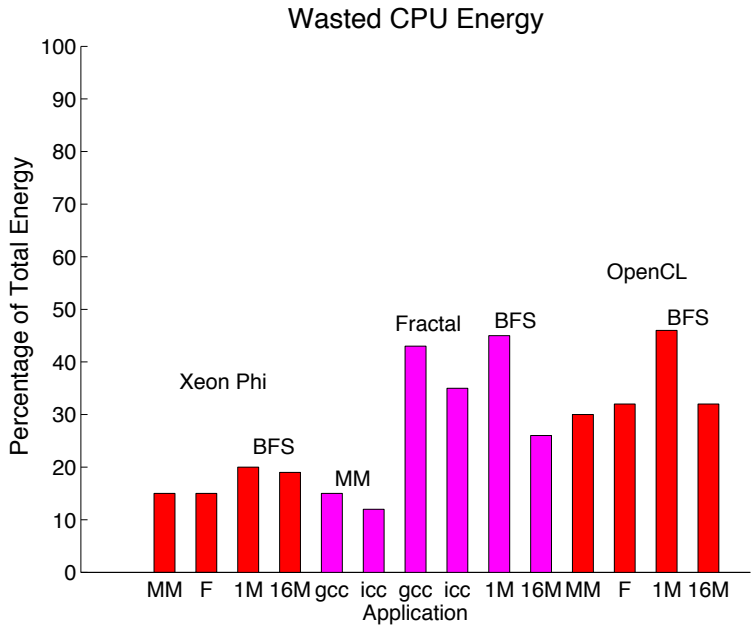

Figure 1.1: Wasted CPU energy

wastes upwards of 43% of the total energy. The rightmost set of bars display the wasted CPU energy for the C++ and OpenCL versions of the applications. The hybrid programming model can remedy this problem by partitioning the work to the CPU and the accelerator.

Unfortunately, few applications and very little research [1] have taken advantage of workload partitioning of hybrid computing to improve energy efficiency. Thus, it is important to explore how to reduce the energy consumption of heterogeneous computer systems and applications by enabling hybrid computation.

The bulk of current research has focused on performance of single CPU, GPU or Xeon Phi. Yet many current high performance and likely most future high-end computing systems will be heterogeneous with multiple CPUs and accelerators in one node. For example, four of today's top ten supercomputers are heterogeneous [2]. For the top ten on the Green 500 list [3], all are heterogeneous. Utilization data from one of the

heterogeneous systems, Stampede, show that the majority of applications use an individual computing component [4]. These systems provide an enormous opportunity for hybrid computing and improved energy efficiency.

Prior research looked at parameters such as memory use, memory transfer, and message passing, improved load balancing and compiler optimizations as methods for performance improvement. When energy efficiency is discussed, it is often inferred with performance. As more software is being coded to take advantage of hardware of heterogeneous systems, little research has been conducted on the energy efficiency of programming for hybrid software implementation. Hybrid computing systems offer an opportunity to take advantage of the available processors for better performance and energy efficiency.

In most cases of hybrid computing, the CPU consumes energy while waiting for the accelerator, to complete the task. To yield a more energy efficient solution, workload can be partitioned between the CPU and an accelerator. Without a model to base a prediction upon, partitioning the workload may performed with an exhaustive search, which can be time and energy consuming. This thesis aims to provide a model to predict workload partitioning for optimal performance or energy efficiency.

For development of the predict model for optimal partitioning, software applications were adapted and included an adjustable range of workload partitions to a CPU and to an accelerating device such as a Xeon Phi or GPU. This approach generates differing workloads and data points that are used to study the impact of workload partitioning on energy efficiency and performance.

In addition to the applications, a model-based prediction method for partitioning workload is proposed. This method uses measured data and predicts a suitable partition for the hybrid application. By doing so, the model copes with the sensitivity of the platform to the application, to the problem size, to the data transfer latencies, and to the drivers and compilers. By modeling the performance, energy and throughput, it predicts the optimal workload partitioning.

The workload partitioning prediction is then compared with the measured performance and energy parameters to validate accuracy. The results show that partitioning improves application performance and energy efficiency. The optimal partitioning point for energy and performance are correctly predicted in more than 90% of the cases, with various workloads, accelerators, and applications.

In order to further assist hybrid application development, this thesis also provides a guideline for creating a hybrid program from parallel applications. Most parallel applications that have already been compiled and executed on an Intel CPU can be readily adapted for hybrid use with a Xeon Phi. However, for use with a CPU and a GPU, both parallel applications for the CPU and GPU must already exist in order to combine and yield a hybrid application.

The main contributions of this thesis are: (1) a number of hybrid applications are developed to benchmark workload partitions, (2) a detailed guideline is provided on writing hybrid applications for CPU and GPU or CPU and Xeon Phi, and (3) a model is proposed to accurately predict the optimal workload partition between CPU and GPU or CPU and Xeon Phi for best performance or energy efficiency.

The rest of this thesis is organized as follows. Chapter 2 discusses background related work. Chapter 3 covers the experimental platform. Guidelines for developing hybrid code from parallel code can be found in Chapter 4. Chapter 5 includes application descriptions and results for the applications. Chapter 6 discusses and provides a model for predicting hybrid workload partitions. Finally, Chapter 7 concludes and offers direction for future work.

2. RELATED WORK

Traditionally, prior research has focused on energy and performance of homogeneous multicore CPU systems. More recently, some research and studies have addressed performance, with the added benefit of energy reduction on a homogeneous GPU system. Others have studied performance of workloads for heterogeneous CPU and GPU systems. Similarly for the Xeon Phi, few have studied and researched performance and energy efficiency of this accelerator. This chapter discusses pertinent work and research associated with performance and energy of CPUs, GPUs, and Xeon Phi.

Liu et al. [5] developed and provided a method to track processor idle times and adjusts the frequency of other processors to provide power savings without reducing performance. However, the technique uses a barrier, creating overhead due to waiting and consuming power. Ge et al. [6] developed an algorithm that relies on past statics, and then adjusts the CPU frequency to reduce energy. This method accounts for neither future workloads nor energy-consuming peak workloads. Hsu and Feng [7] have shown that CPU energy usage can be reduced with DVFS. Pallipadi and Starikovskiy [8] proposed to use the on demand governor in the Linux kernel to adjust CPU frequency based upon utilization thereby reducing energy use. Much attention has been given to CPU performance and energy of applications in papers published by Hsu and Kremer [9], Barroso and Holzle [10]. Vhdat et al. [11] direct their attention to energy efficiency in operating systems. Lee and Zomaya [12] offer scheduling algorithms for multiprocessor computer systems for the benefit for energy and performance. While the algorithm is applicable to multiprocessors, this paper and others listed prior only consider CPUs.

A number of previous studies presented the performance and energy consumption of homogeneous GPU usage. For example, Hong and Kim [13] propose to throttle a number of GPU cores to reduce energy consumption, which requires characterization of the GPU by micro-benchmarks. Collange et al. [14] found that memory access patterns and bandwidth can play a role in both energy use and performance in GPUs. Although useful in tuning and optimizing GPU code, it directs it attention only to the GPU. Similiarly, Che et al [15] discusses optimizing memory performance and hence, over all performance, by overlapping memory accesses with PCI-E transfers. Bailey et al. [16] proposes a model using kernel clustering and multivariate linear regressions to improve performance. Wu et al. [17] uses energy aware compiler optimizations. And Song et al. [18] applies machine learning to the performance and energy efficiency of GPU programs, employing only models and not the on-chip power sensor. However, only heterogeneous GPU applications were considered in these papers.

Few studies have been reported for performance improvement by splitting tasks in heterogeneous CPU-GPU systems. Luk et al. [19] proposed a scheme to minimize execution time based on computational task distribution to the GPU and CPU. In the same realm, Dean and Ghemawat [20] studied workload division based on the MapReduce Framework. In a variation, Ravis et al. [21] proposed to partition data dynamically to the CPU and GPU cores to improve performance. Che et al. [22] suggested splitting workload between CPU and GPU to improve performance with the GPU frequencies set to their peak level. While their work provided a wealth of heterogeneous applications, none provide hybrid application of a CPU and GPU nor do they scale to larger problem sizes. A proposed system by Scogland et al. [23] divide the

work between the CPU and GPU based upon the characteristics of the workload. Gee et al. [1] provide an energy and performance models for hybrid workload division to the GPU based upon both advertised and measured parameters. These studies address only CPU-GPU systems and not a comprehensive model for CPU's, GPUs and Xeon Phi's

With the recent release of the Xeon Phi, the studies on addressing energy usage and optimization of Xeon Phi are limited. Fang et al. [24] provided an empirical evaluation of the Xeon Phi, discussing cores, memory, and interconnects in the context of performance and peripherally energy. Lawson et al. [25] evaluated and recommended specific thread affinities to improve performance, thereby reducing energy usage. Wood et al. [26] characterized energy and power usage of various applications running on Xeon Phi. Lorenzo et al. [27] studied energy and power implications of thread numbers on Xeon Phi and Yao al. [28] proposed an instruction level energy model for the Xeon Phi. Li et al. [29] characterized performance and energy efficiency tradeoffs of HPC applications running on Xeon Phi.

While much work and effort in characterizing and modeling energy and performance has supported on CPU-GPU heterogeneous systems and on homogeneous systems, little work has compared energy and performance on heterogeneous systems with hybrid use of CPUs and accelerators. This thesis may be the first to attempt to explore the benefits of hybrid computing and offer a prediction model, which predicts workload partitions for optimal performance or optimal energy efficiency for both CPU and GPU and CPU and Xeon Phi systems.

3. EXPERIMENTAL PLATFORM

All experiments were performed on a node of the Marcher System, which is an NSF funded power measurable, high performance-computing platform [30]. Each node in the system has a dual 8-core Xeon Sandy Bridge E5-2670 processor, a Xeon Phi, and NVIDIA Tesla K20 GPU.

The Xeon Phi, used to accelerate parallel computing, was chosen due to many factors: its prevalence in HPC platforms; a higher degree of parallelism with respect to a CPU; and a single code model with an Intel CPU, facilitating code reuse and design. The Xeon Phi in our system has 61 cores with one reserved for the operating system, yielding 60 cores for parallel applications. It has a maximum computational performance of one TFLOPS and a maximum memory bandwidth of 352 GB/s [31]. The Xeon Phi is found in Tianhe-2 (Milky Way 2), ranked first on the Top500 List and Stampede, which was ranked tenth as of June 2016 [1].

The K20 has 2688 compute cores, operates at the default rate of 705 MHZ with a peak performance of 1.17 TFLOPS, and interacts with the parallel computing application-programming interface CUDA and OpenCL. The K20 and similar NVIDIA GPUs are popular in HPC systems. The parameters for the GPU, Xeon Phi and CPU are listed in Table 1.

For all of the measurements, the power data is collected directly with the following interfaces: the CPU Power is collected via the RAPL [32] interface, the Xeon Phi power via the Intel micsmc interface [33], and the GPU power via the NVIDIA smi interface [34]. Please refer to the papers published by Burtscher et al. [35] and Wood et al. [26] for detailed power measurement of K20 and Xeon Phi on the Marcher system.

The Marcher system generates files of power readings for the CPU, GPU, and Xeon Phi. These power readings record the real time power use of all three components (CPU, Xeon Phi, and GPU). The energy of each component is calculated as the accumulated products of execution time and profiled instantaneous power. For total energy, the CPU energy and the GPU or Xeon Phi energy are combined. The power readings of other system components such as the DRAM and disks are not included as they remain unchanged most of the time when running the Matrix Multiplication, Fractal, and Breadth-first Search codes.

Table 3.1: Experimental platform parameters

	CPU	**Coprocessor**	**GPU**
Host/Device	Intel E5 -2670	Xeon Phi	NVIDIA K20
Number of Cores	16	60	2688
Nominal Frequency	2.6 GHz	1.05 GHz	705MHz
DVFS Enabled	no	not available	no
Memory Size	32 GB	8 GB	5 GB
Threading API	OpenMP	OpenMP	CUDA 7.0
Peak Performance	332 GFLOPS	1 TFLOPS	1.17 TFLOPS (double precision)
Thermal Design Power	115 W	245 W	235 W
Memory Bandwidth	51.2 GB/s	352 GB/s	208 GB/s
Compiler	gcc, (icc)	icc	nvcc
Compiler Version	4.4.7 20120313 (Red Hat 4.4.7-16)	13.1.3 20130607	7.0, V7.0.27
Operating System	CentOS 6.5		

4. GUIDELINES TO CREATING HYBRID CODE

4.1 CPU and Xeon Phi

Parallelized OpenMP code compiled with the Intel compiler and which already runs on the CPU represents a good candidate for hybrid use on the CPU and Xeon Phi coprocessor. Typical applications include computational heavy code with sequentially accessed array data structures

There are two methodologies for hybridizing code for the CPU and Xeon Phi. The first uses the Intel Math Kernel Libraries (MKL) [36] that implements efficient and effective routines for math functions such as matrix multiplication. The MKL provides the ease of using a built in function, but not all functions are available. The second method is to hybridize OpenMP sections to the Xeon Phi. While this method requires considerable software time investment, most functions that have been parallelized with OpenMP can be changed to CPU and Xeon Phi hybrid implementation. Both methods can provide performance speedup and improve energy efficiency.

4.1.1 R Language and MKL

For the R scripting language, some of the functions such as matrix multiplication or matrix transposition have MKL implementation, refer to [37]. The MKL must be enabled via variables as follows:

```
export MKL_MIC_ENABLE=1
export OMP_NUM_THREADS=16
export MIC_OMP_NUM_THREADS=240
export MKL_HOST_WORKDIVISION=0.1
```

export OFFLOAD_REPORT=2

For some of the functions, the work division is fixed and pre-determined. This value is dependent upon application and workload. In that case, the export *MKL_HOST_WORKDIVISION* command will have no effect.

4.1.2 C++ and MKL

In order to use the MKL libraries, the application must first be identified and memory allocated specifically. The library reference guide includes listing of available functions [36]. The compiler directive *#include"mkl.h"* must be used at the beginning of the file. For memory allocation, instead of using *malloc*, the memory allocation command must be *mkl_malloc* and memory set on 64-bit boundary, and to free the allocation, the command is *mkl_free()*. Using the algorithm from Predicting Workload Distribution, calculate the percentage of workload on the CPU for either optimal performance or optimal energy efficiency.

To enable workload split, set the environment variables as listed in the previous section. The offload report options allows for verification of the CPU and Xeon Phi workload. The workload division can be adjusted with the *MKL_HOST_WORKDIVISION* command with the above example sending 10% of the workload to the CPU and 90% to the Xeon Phi. Figure 4.1 shows an example of MKL matrix multiplication.

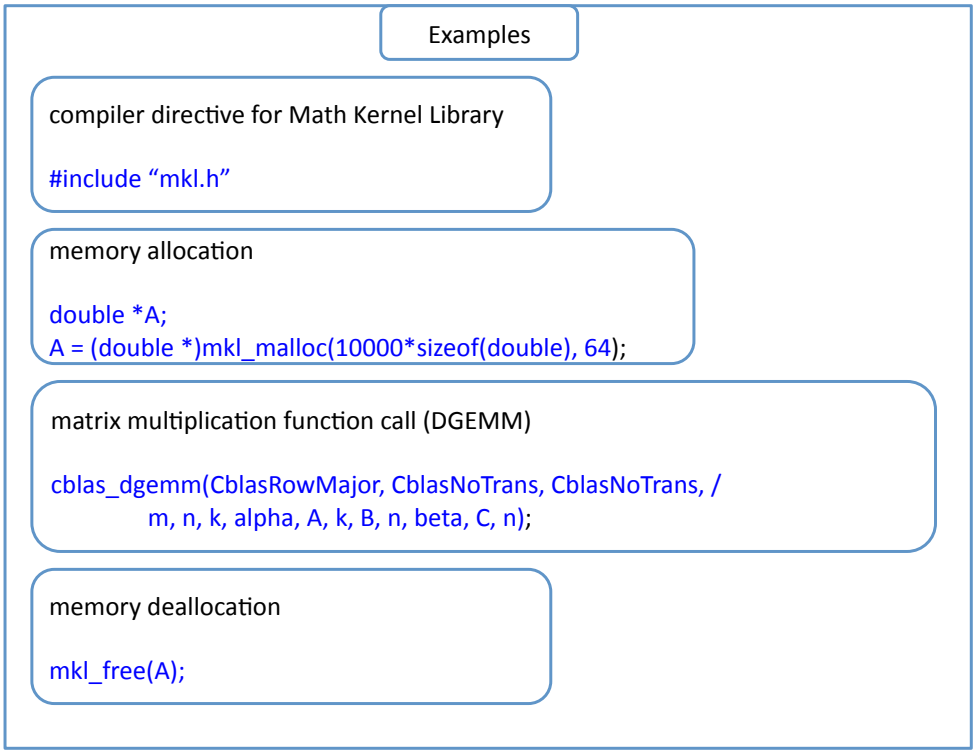

Figure 4.1: C++ MKL example

4.1.3 C++ and OpenMP

An additional method to partition the workload between a CPU and Xeon Phi within the program is to use OpenMP. Assuming that the parallel code has already been successfully compiled with the Intel compiler, icc, add *omp_set_nested(1)* before any OpenMP code. Next, add the code to divide the work between the CPU and Xeon Phi and shown in the Figure 4.2. The variable used to hold the thread number information, *thread_Id* must be private. As a good programming practice, defining which variables are private or shared can allow for predictable operation.

In the CPU and Xeon Phi code sections, the *omp_parallel_for* or *omp_parallel* can be used to further divide the work to the cores. For the Xeon Phi, offloading occurs with the compiler directive *#pragma offload target(mic:0) ...* . Refer to Intel offload resources for specifics on use of in, out, or inout options. However, each data transfer to the Xeon Phi will have allocation.

Because the Xeon Phi cores can support 4 threads to hide latency, the number of threads should be allocated as follows: *numberThreads = (numberCores -1) x 4* and set an environmental variable with *export MIC_KMP_AFFINITY=explicit,granularity=fine,proclist=[1-239:1]*, for 60 cores. One core should remain free to run the Linux operating system.

Set the environment variable OFFLOAD_REPORT =2 to allow for verification of Xeon Phi transfer.

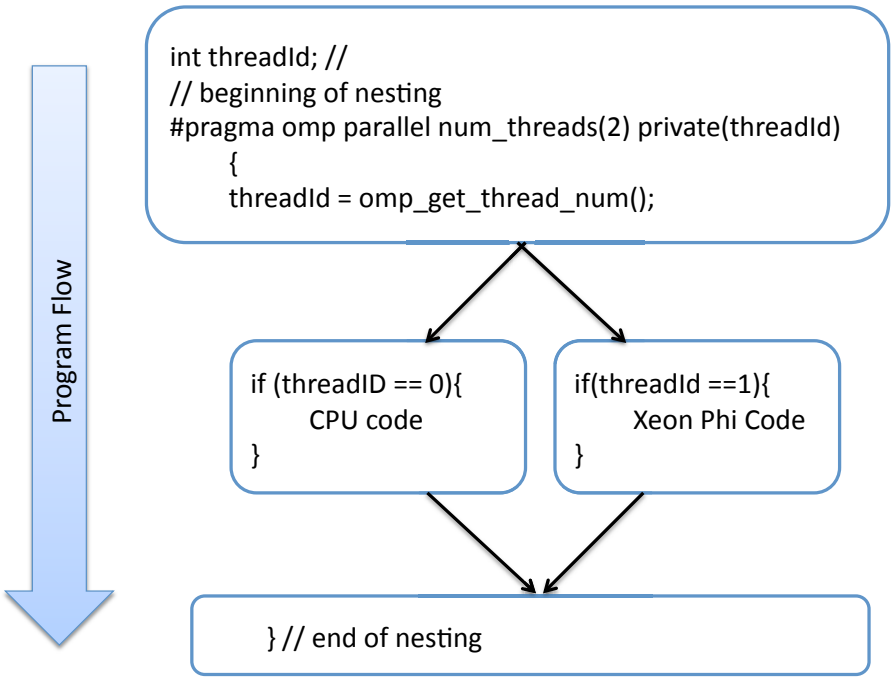

Figure 4.2: CPU and Xeon Phi OpenMP code division

4.2 CPU and GPU

The GPU has the capability to run two different kernel languages, CUDA and OpenCL. CUDA was created by NVIDIA, executing only on their GPUs. However, OpenCL is an open source language for parallel programming and executes on many CPUs and accelerators. Both languages were used to provide kernel programming for the GPU.

4.2.1 C++ and CUDA

For C++ and CUDA applications, there are two options to modify code for hybrid operation on with the CPU and GPU. The first option is to combine all code, the GPU kernel and CPU OpenMP code, into one file. While against the modern convention of modularity and abstraction, this approach can yield shorter execution times and more straightforward compilation and linking.

Assuming that both the GPU kernel and the CPU OpenMP parallelized code has been written, trouble shot and run separately, one can combine the two codes into one file, by adding the *omp_set_nested(1)* before any of OpenMP code. This allows the program flow to split between the CPU and GPU. Then add the code to divide the work between the CPU and GPU as shown in Figure 4.3. The *threadID* variable, which must be set to *private*, collects the thread number, and is used to split the work.

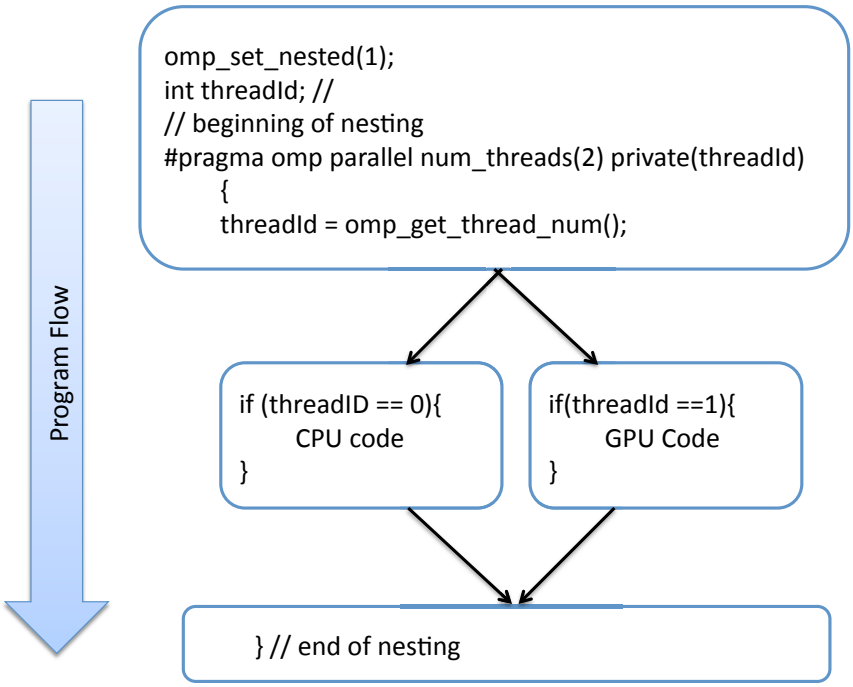

Figure 4.3: C++ and CUDA code division

After the code for hybrid operation has been added, the code can be compiled using the NVIDIA compiler (nvcc).

The second option for C++ and CUDA is to compile the codes separately, then link the object files together, creating an executable. The code to divide the work must reside in the CPU C++ portion of the code and must have visibility to the GPU kernel. Typically, visibility to CUDA code is implemented with a wrapper function that is declared but not defined in the C++ file. Combining and mixing CUDA and C++ code can be exasperating, tutorials have addressed this such as the one from Oak Ridge National Laboratory [40]. The code to divide the work is shown in Figure 4.4.

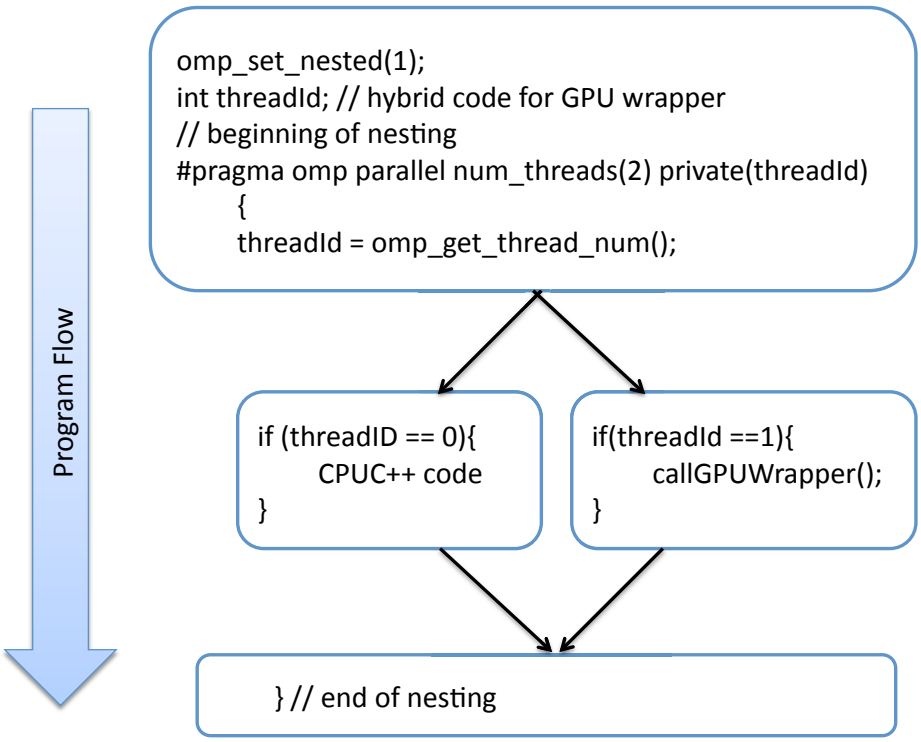

Figure 4.4: C++ and CUDA wrapper code division

4.2.2 C++ and OpenCL

While OpenCL enables task and data partitioning, one of the most straight forward ways is to use OpenMP in a similar method as used previously, shown in Figure 4.5. OpenMP has the advantage of allowing the programmers to specify which variables are shared or not shared. Although future work can compare and contrast different division methods and code, this thesis needs a consistent method to constrain the research. Thus, wrapping the kernel code allows for modularity in OpenCL.

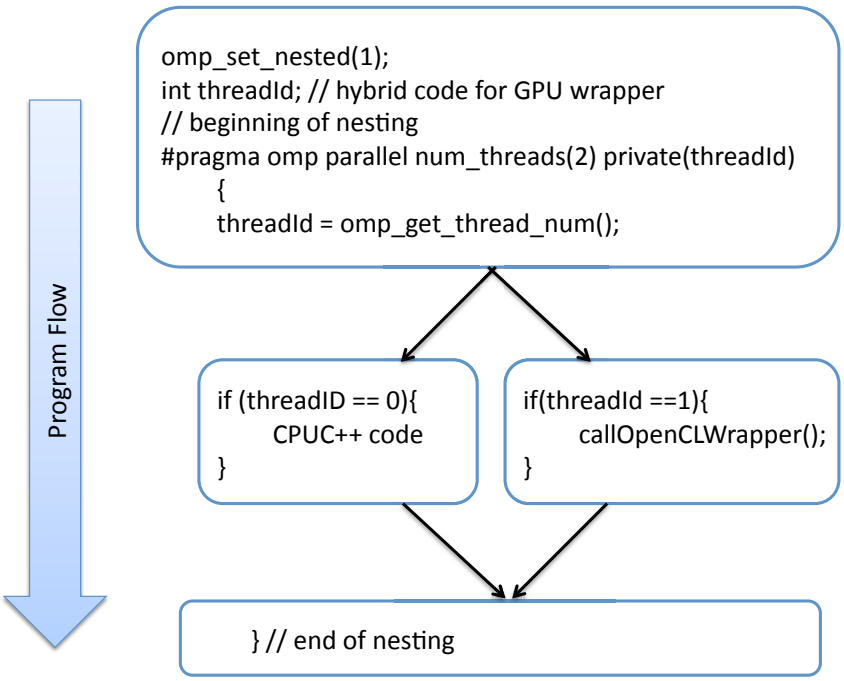

Figure 4.5: C++ and OpenCL wrapper code division

4.3 Dividing the Work

How to divide the work between the CPU and Xeon Phi or GPU depends upon the application. Task division makes sense when there are varying tasks with some suited for the CPU and others suited for the GPU or Xeon Phi. Data parallelism works when both do the same task, refer to *An Introduction to Parallel Programming* by Pacheco [41] for additional details.

The data should be partitioned by full rows, which allows for optimal memory access and ease of work division as shown in Figure 4.6.

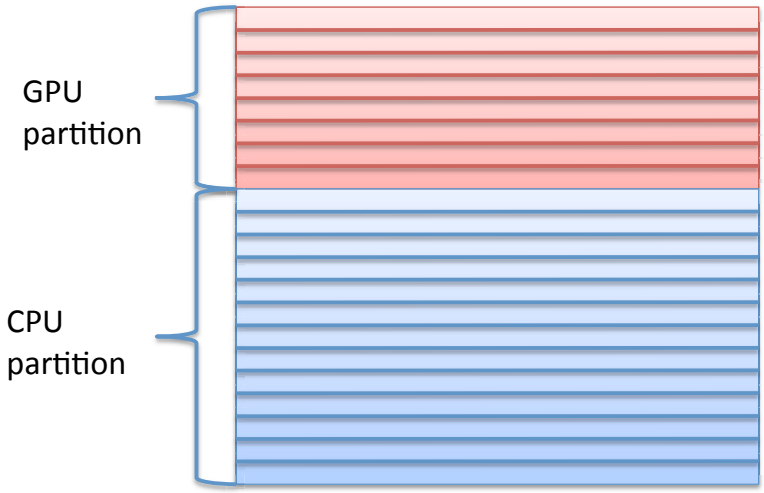

Figure 4.6: Full row partition

For example, the matrix multiplication work division code divides the A matrix, but not the B matrix. Thus, Figure 4.6 represents the division for matrix A.

5. RESULTS AND ANALYSIS

This section presents the performance and energy results of the hybrid applications running on either the CPU and GPU pair or the CPU and Xeon Phi pair. For each of the workload partitions, the effective execution time is the longest of the two times of the CPU and Xeon Phi or CPU and GPU. In hybrid applications, the CPU must be active with at least one thread operational to receive data or signals from the accelerator in order to conclude the calculations and finalize the tasks. In some cases the CPU may finish first, then wait for accelerator to finish. In others, the GPU or Xeon Phi may complete its workload before the CPU does. In either case, the CPU remains active for the entire application.

All code was compiled with the -O2 optimization flag.

The data was taken every 10% with exceptions noted in the results. The results listed are an averaged over three trials.

The optimal performance point is defined as workload division percentage with the shortest execution time. The optimal energy point is defined as the workload percentage of task splitting with the least amount of energy consumed.

5.1 Application Descriptions

5.1.1 Matrix Multiplication

Matrix multiplication was chosen because is it a widely used benchmark in scientific computing. For these tests, two randomly filled 10,000 square matrices are multiplied together to yield a 10,000 x 10,000 matrix. The number of calculations per each of the 10,000 final elements is 10,000 multiplications plus 9,999 additions. This

provides approximately 2,000,000,000,000 operations. Work division between the CPU and GPU or Xeon Phi is partitioned in full rows.

5.1.2 Fractal

Fractal is a computationally intensive application requiring minimal memory use. The fractal algorithm is important is that it can achieve high image compression ratios while retaining a high quality image. As implemented, the size is 20,000 by 20,000, provides 1,329,000,000 calculations and yields workable execution times.

5.1.3 Breadth-first Search (BFS)

Breath-first search is a graph traversal algorithm, searching each neighbor node for a specific value. It starts at the root node, visiting each node at that level before progressing to the next level. BFS is an important algorithm in that it is used to solve many problems such as finding a shortest path between two nodes, or cities. However, the algorithm as implemented from the Rodinia [22] does not search a tree structure for a specific value, but starts at the root node and visits each node throughout the entire structure to the very last node. The code for the application was modified for hybrid operation. The input graph data structures consist of an ordered list of nodes, numbered edges, and edges. This is essentially a linear array simulating a tree structure. These arrays allow for shared access amongst the threads enhancing parallel operation. In order to provide a non-computationally intensive algorithm and to further evaluation the proposed model, this BFS application has been modified to be data transfer intensive. The results are reset after completing the search to allow transfer and search again. In doing so, this requires the data to be transferred to and from the Xeon Phi each time. The

modified algorithm changes a one-time data transfer application by transferring data up to 100 times between the host and device, completing 100 times total searches.

The two input graph data text files sizes are 1,000,000 nodes with 5,999,950 edges and 16,777,216 nodes with 100,675,408 edges. The file is read only once and is stored for successive searches.

5.2 CPU and Xeon Phi Energy and Execution Time Results

5.2.1 R Matrix Multiplication

The R scripting language is an open source, popular, statistical computing and graphing language, which runs on a wide variety of platforms. R matrix multiplication application was chosen because it enables the Math Kernel Library that uses parallel matrix multiplication with an adjustable workload distribution. While the R package downloaded from a CRAN mirror [37] only provides serial implementation, an open source version with parallel version is available from Revolution Analytics [38]. This version implements some of the parallel MKL features. The environment variables were set to enable a range of workload distributions from CPU = 100% to CPU = 0%.

The R matrix multiplication application was adapted from an R benchmark originally from Splus Benchmark Test V.2 by Stephan Steinhaus and contains adaptations by Philippe Grosjean, Douglas Bates, and Simon Urbanek. The matrices are setup as 10,000 by 10,000 normal distributed floats about 0.0 with a mean of 1.0. The number of CPU OpenMP threads is set to 16, and the number of Xeon Phi OpenMP threads is set to 240.

Performance and energy results of R matrix multiply are tabulated in Table 5.1 and graphed in Figures 5.1 and 5.2. Both optimal performance and energy points occur in a range of workload distributions between CPU = 0% and 60. However, the workload distribution of CPU = 35% also represents a second optimal point in that the difference in executions times and energies are both less than 0.5%.

Table 5.1: R matrix multiplication

R Matrix Multiplication				
CPU %	**Total Execution Time (sec)**	**CPU Energy (J)**	**Xeon Phi Energy (J)**	**Total Energy (J)**
100	33.56	3388	224	3612
90	30.976	2823	766	3589
80	29.445	2572	821	3393
70	28.77	2397	639	3036
60	27.342	2264	844	3108
50	27.028	2084	784	2868
45	26.6	1976	712	2688
40	26.639	1988	861	2849
35	26.338	1815	824	2639
30	26.879	1774	882	2656
25	26.953	1733	1008	2741
20	27.462	1648	1047	2695
15	26.62	1572	1111	2683
10	27.325	1508	1102	2610
5	26.765	1449	1191	2640
0	26.214	1388	1242	2630

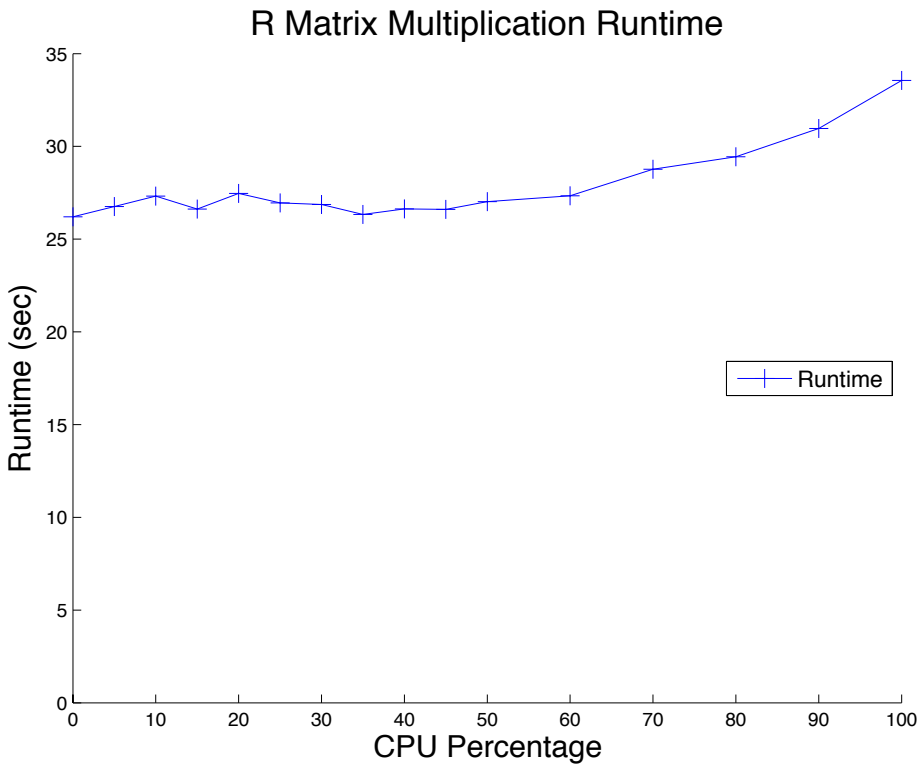

Figure 5.1: R matrix multiplication runtime

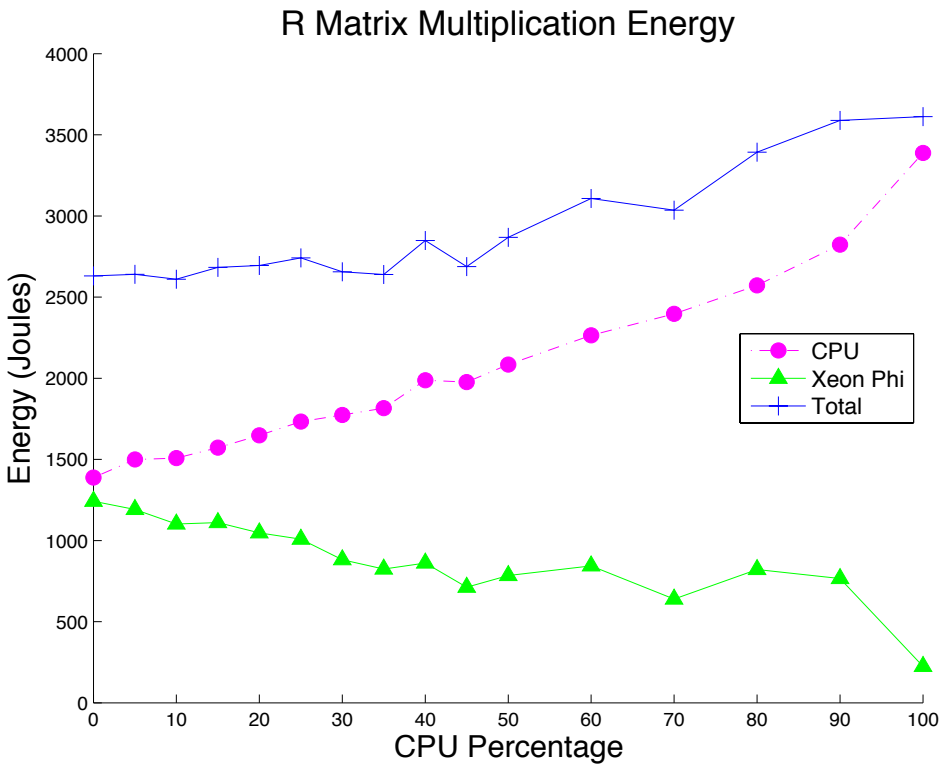

Figure 5.2: R matrix multiplication energy

5.2.2 C++ Matrix Multiplication

Performance and energy results for the C++ matrix multiplication application are provided in Table 5.2 and Figures 5.3 and 5.4. The optimal performance point at CPU = 40%. The optimal energy point is at CPU = 0%. For this application on the CPU and Xeon Phi, the optimal performance and energy points do not coincide.

Table 5.2: C++ matrix multiplication with CPU and Xeon Phi

C++ Matrix Multiplication with CPU and Xeon Phi						
CPU %	**CPU Execution time (sec)**	**CPU Energy (J)**	**PHI Execution time (sec)**	**PHI Energy (J)**	**Effective Execution time (sec)**	**Total Energy (J)**
100	42.59	7918	0	0	42.59	7918
90	37.00	6909	7.49	1151	37.00	8090
80	35.74	6395	9.59	1539	35.74	7935
70	29.84	5670	11.41	1859	29.84	7529
60	25.00	4798	13.76	2327	25.00	7126
50	21.12	3956	16.52	2939	21.12	6896
40	17.69	3356	18.57	3297	18.57	6564
30	12.79	2873	20.96	3692	20.96	6566
20	8.62	2412	23.14	4207	23.14	6620
10	4.38	1991	25.31	4623	25.31	6614
0	0	1429	26.99	4795	26.99	6224

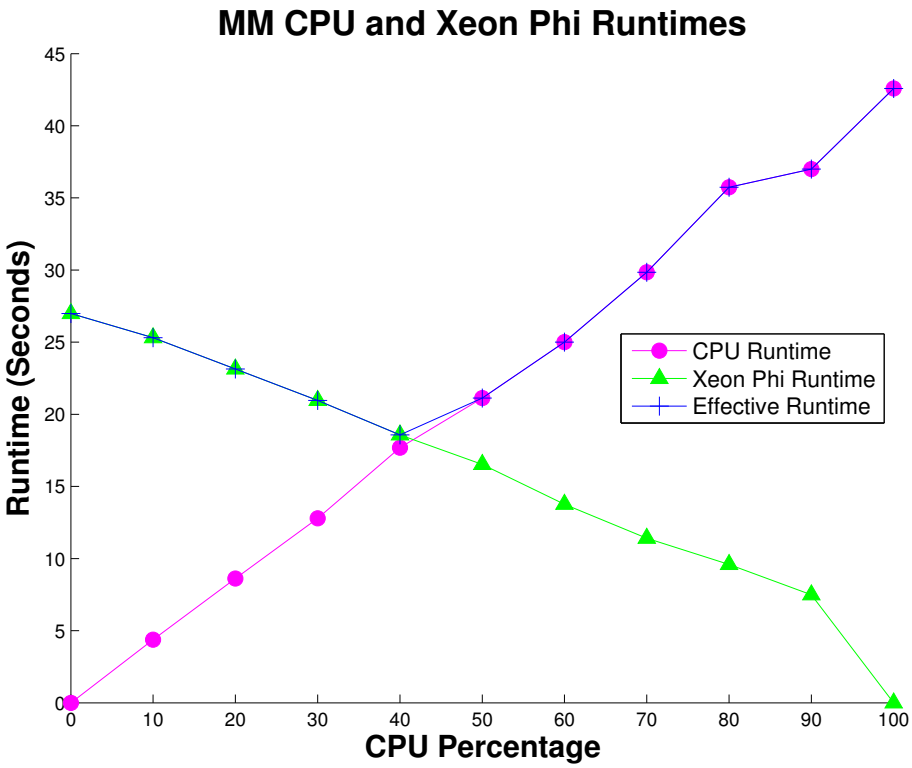

Figure 5.3: CPU and Xeon Phi C++ matrix multiplication runtime

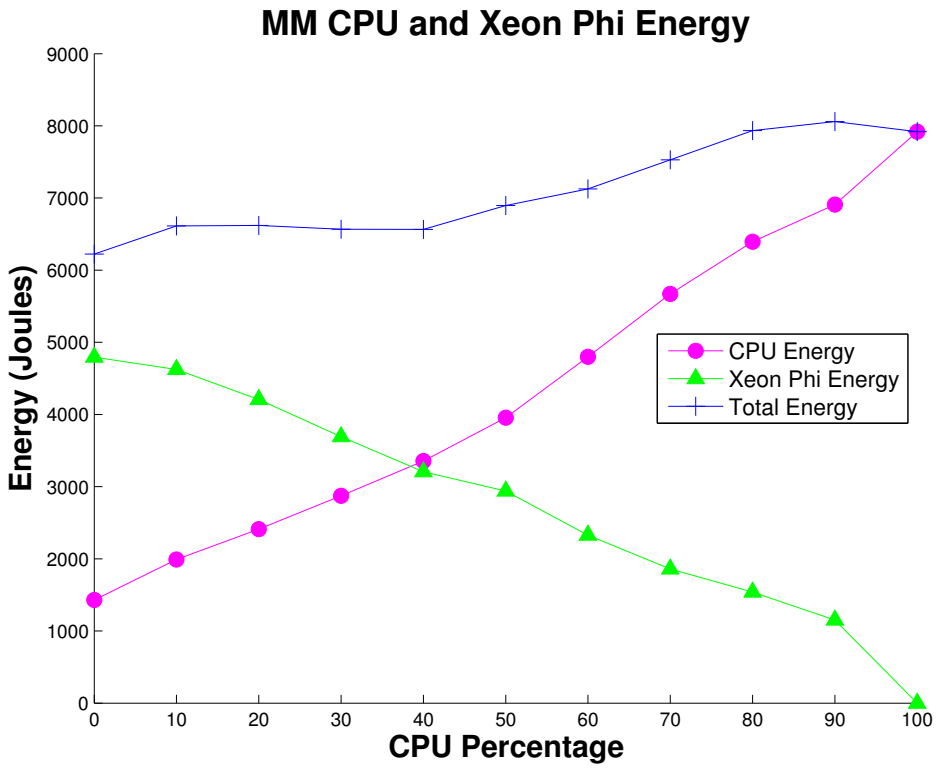

Figure 5.4: CPU and Xeon Phi C++ matrix multiplication energy

5.2.3 C++ Fractal

The execution time and energy data is provided in Table 5.3 and graphed in Figures 5.5 and 5.6. The optimal performance point is where the CPU = 30% with the optimal energy point at CPU = 40%. For this application on the Xeon Phi, the optimal points do not coincide.

Table 5.3: C++ Fractal with CPU and Xeon Phi

C++ Fractal with CPU and Xeon Phi						
CPU%	**CPU Execution time (sec)**	**CPU Energy (J)**	**PHI Execution time (sec)**	**PHI Energy (J)**	**Effective Execution time (sec)**	**Total Energy (J)**
100	47.39	6613	0	0	47.39	6613
90	42.57	6159	5.84	962	42.57	7121
80	39.45	5477	6.66	1115	39.45	6592
70	35.48	4912	10.19	1650	35.48	6562
60	32.13	4480	10.47	1738	32.13	6218
50	29.50	4007	12.61	2067	29.50	6074
40	26.13	3603	15.36	2419	26.13	6022
30	20.61	2895	19.83	3222	20.61	6117
20	13.36	2470	24.11	3894	24.11	6364
10	6.12	2104	27.32	4447	27.32	6551
0	0	1692	30.49	5029	30.49	6721

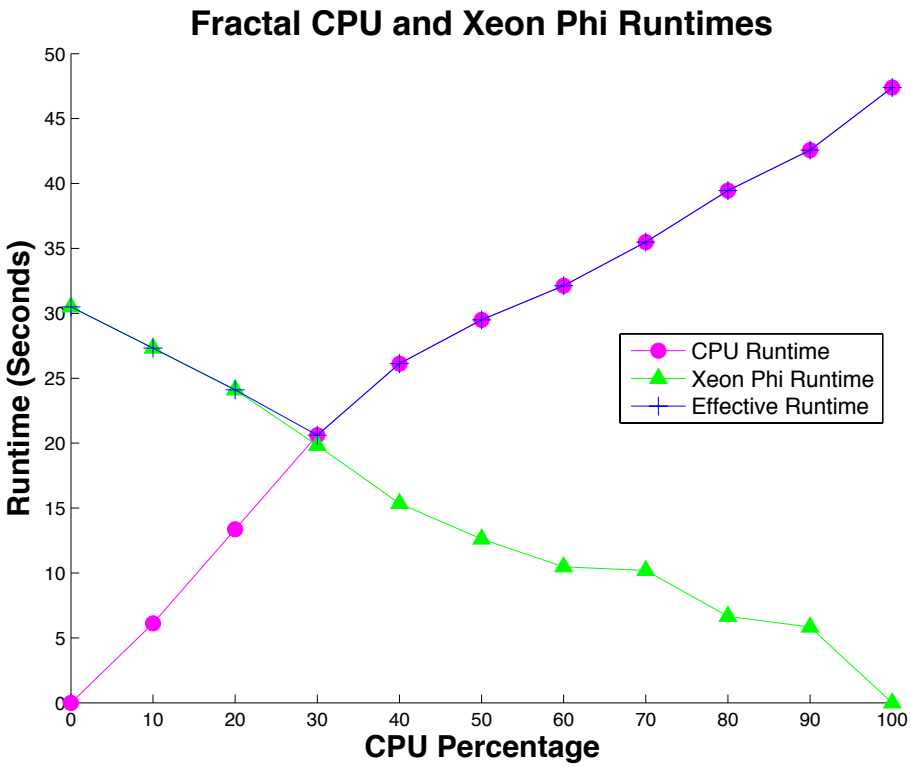

Figure 5.5: CPU and Xeon Phi C++ fractal runtime

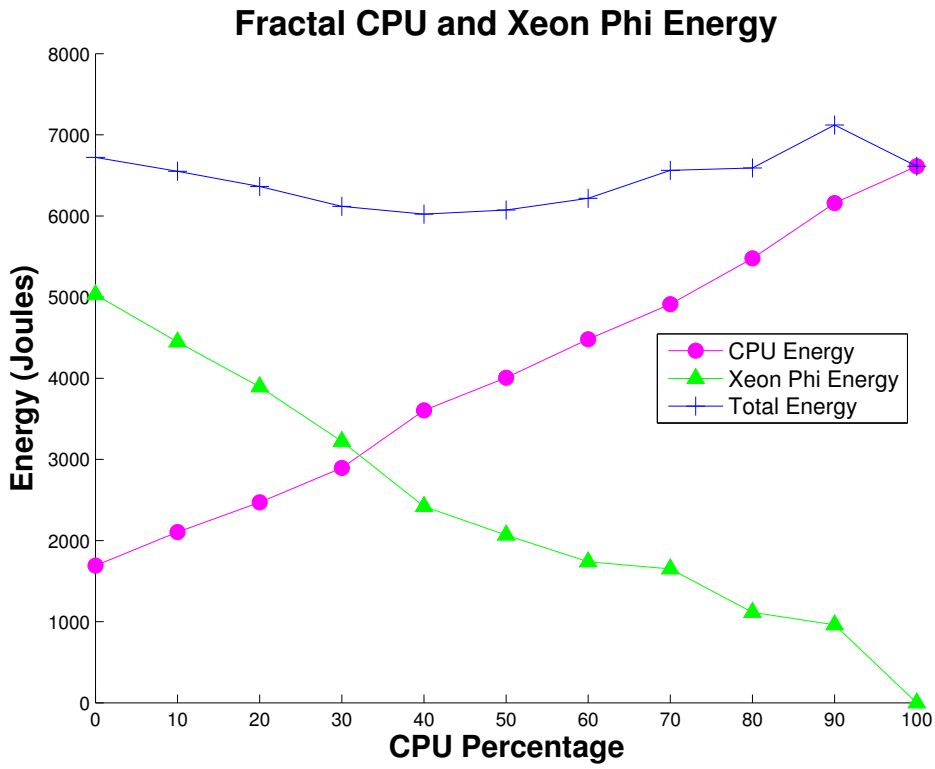

Figure 5.6: CPU and Xeon Phi C++ fractal energy

5.2.4 C++ Breadth-first Search (BFS)

Performance and energy data for BFS on the CPU and Xeon Phi for 1 million nodes is shown in Table 5.4 and for 16 million nodes in Table 5.5. The graphed data is shown in Figures 5.7 and 5.8. Data points were taken every 10%. However, upon additional testing, the optimal performance point for 1 million nodes is CPU = 86%. The optimal energy point is CPU = 100%. For the larger size is when the optimal performance point occurs when CPU = 60% and the optimal energy point when CPU = 100%. An additional performance point was selected at 86%, which provided the best runtime of 3.57 seconds.

Table 5.4: C++ Breadth-first search with CPU and Xeon Phi with 1M

C++ BFS with CPU and Xeon Phi 1M						
CPU %	**CPU time (sec)**	**CPU Energy (J)**	**MIC time (sec)**	**MIC Energy (J)**	**Execution time (sec)**	**Total Energy (J)**
100	3.71	530	0	0	3.714	530
90	3.62	471	3.09	285	3.623	756
80	3.31	463	4.167	350	4.167	813
70	3.10	467	5.267	480	5.267	947
60	2.83	514	6.287	607	6.287	1121
50	2.58	558	7.412	761	7.412	1319
40	2.34	577	8.472	891	8.472	1468
30	2.08	622	9.685	1050	9.685	1672
20	1.82	652	10.698	1188	10.698	1840
10	1.58	691	11.768	1341	11.768	2032
0	1.31	700	12.843	1453	12.843	2153

Table 5.5: C++ Breadth-first search with CPU and Xeon Phi with 16M

C++ BFS with CPU and Xeon Phi 16M						
CPU %	**CPU time (sec)**	**CPU Energy (J)**	**MIC time (sec)**	**MIC Energy (J)**	**Execution time (sec)**	**Total Energy (J)**
100	112.8	15014	0	0	112.8	15014
90	103.8	12928	38.34	2706	103.8	15634
80	94.3	11631	51.03	4766	94.3	16397
70	85.6	11140	63.79	6331	85.6	17471
60	76.07	9253	76.57	10300	76.6	17363
50	66.64	8558	89.23	10451	89.2	19009
40	58.38	7851	101.6	11168	101.6	20022
30	49.46	5747	114.5	12171	114.5	21743
20	40.26	8676	127.6	15287	127.6	23963
10	31.49	8449	140.2	16995	140.2	25444
0	22.47	8285	152.9	18944	152.9	27299

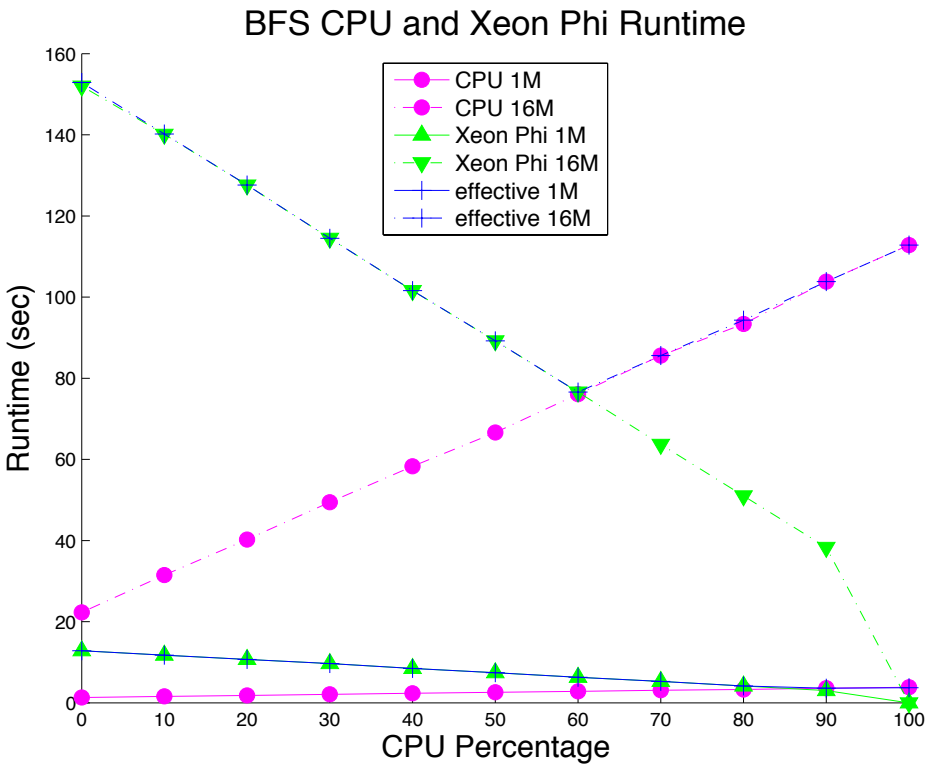

Figure 5.7: CPU and Xeon Phi C++ BFS runtime

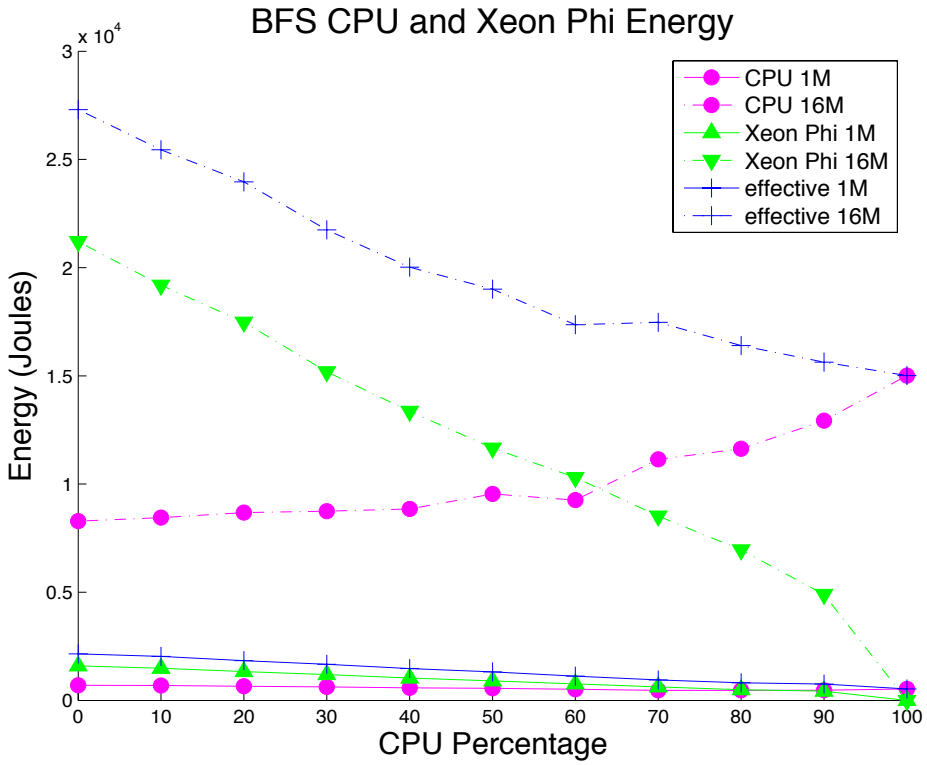

Figure 5.8: CPU and Xeon Phi C++ BFS energy

5.3 CPU and GPU Execution Time and Energy Results

5.3.1 C++ and CUDA Matrix Multiplication

The same code for the matrix multiplication algorithm was compiled with both the Intel compiler (icc) and GNU Compiler Collection (gcc). For both applications, both optimal performance and energy points coincide at CPU = 0%. At CPU = 100%, the icc compiler provides 2.3 speedup with an approximately 65% energy savings. The data is tabulated in Tables 5.6 and 5.7 and graphed in Figures 5.9 and 5.10.

Table 5.6: C++ matrix multiplication with CPU and GPU (gcc)

Matrix Multiplication with CPU and GPU (gcc)						
CPU %	**CPU Execution time (sec)**	**CPU Energy (J)**	**GPU Execution time (sec)**	**GPU Energy (J)**	**Effective Execution time (sec)**	**Total Energy (J)**
100	100.98	15606	0	0	100.98	15606
90	88.55	13749	0.33	16	88.55	13765
80	77.00	12021	0.36	19	77.00	12040
70	66.37	10700	0.39	21	66.37	10720
60	56.47	8972	0.46	25	56.47	8997
50	48.77	7535	0.51	27	48.77	7562
40	36	6175	0.60	30	36	6207
30	27.88	4704	0.62	33	27.88	4737
20	18.74	3177	0.64	34	18.74	3210
10	9.65	2038	0.69	37	9.651	2075
0	2.30	475	0.78	41	2.38	516

Table 5.7: C++ matrix multiplication with CPU and GPU (icc)

Matrix Multiplication with CPU and GPU (icc)						
CPU%	**CPU Execution time (sec)**	**CPU Energy (J)**	**GPU Execution time (sec)**	**GPU Energy (J)**	**Effective Execution time (sec)**	**Total Energy (J)**
100	43.62	7937	0	0	43.62	7937
90	37.88	6919	0.33	16	37.88	6935
80	36.51	6406	0.36	19	36.51	6425
70	29.78	5662	0.39	21	29.78	5683
60	25.79	4821	0.46	25	25.79	4846
50	21.38	4017	0.51	27	21.38	4044
40	17.89	3378	0.60	32	17.89	3410
30	12.79	3899	0.625	33	12.79	2932
20	8.82	2459	0.64	34	8.82	2493
10	4.55	2012	0.69	37	4.55	2049
0	1.75	465	0.78	41	1.75	506

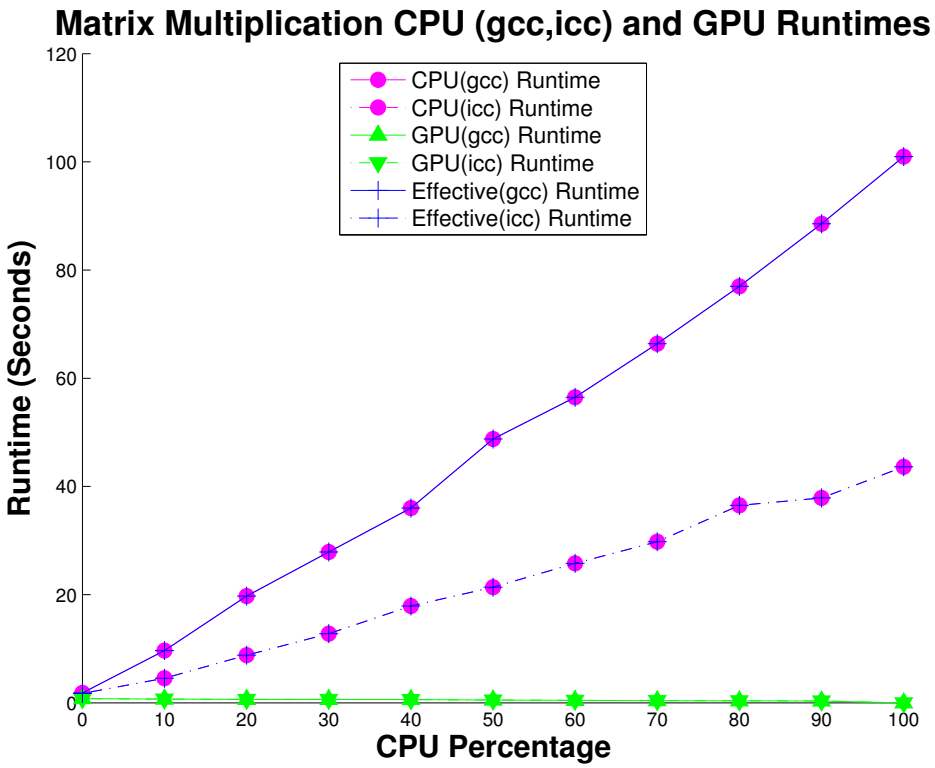

Figure 5.9: CPU and GPU matrix multiplication with icc, gcc runtime

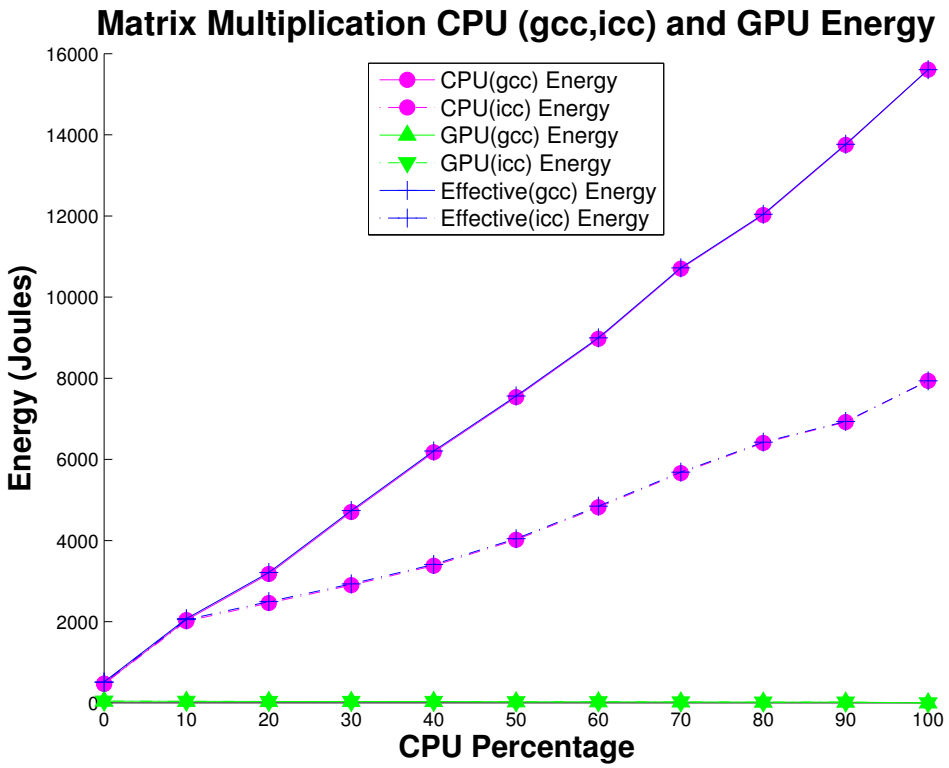

Figure 5:10: CPU and GPU matrix multiplication with icc, gcc energy

5.3.2 C++ and CUDA Fractal

As with matrix multiplication application, this application was compiled with the two compilers gcc and icc. The optimal performance and energy points for both of the compiler choices are the same at CPU = 0%. However, at all other CPU workload distribution percentages, icc provides better execution times resulting less energy used. The data for the gcc compiler are tabulated in Table 5.8 and the icc compiler in Table 5.9. The results are graphed in Figures 5.11 and 5.12.

Table 5:8 Fractal with CPU and GPU (gcc)

Fractal with CPU and GPU (gcc)						
CPU%	**CPU Execution time (sec)**	**CPU Energy (J)**	**GPU Execution time (sec)**	**GPU Energy (J)**	**Effective Execution time (sec)**	**Total Energy (J)**
100	55.74	8211	0	0	55.74	8211
90	50.21	7432	0.23	15	50.21	7447
80	46.52	7190	0.33	15	46.52	7205
70	41.62	6129	0.27	15	41.62	6144
60	38.07	5943	0.28	15	38.07	5958
50	35.03	5367	0.27	15	35.03	5383
40	31.05	4799	0.35	18.8	31.05	4807
30	24.73	3568	0.40	21.5	24.73	3589
20	16.51	2364	0.48	25.9	16.51	2390
10	8.31	1165	0.55	27.0	8.31	1192
0	2.01	123	0.59	32.4	2.01	155.4

Table 5.9: Fractal with CPU and GPU (icc)

Fractal with CPU and GPU (icc)						
CPU %	**CPU Execution time (sec)**	**CPU Energy (J)**	**GPU Execution time (sec)**	**GPU Energy (J)**	**Effective Execution time (sec)**	**Total Energy (J)**
100	47.45	6653	0	0	47.45	6653
90	42.65	6025	0.23	15.0	42.65	6040
80	39.27	5724	0.33	15.0	39.27	5739
70	35.39	5204	0.26	15.1	35.39	5219
60	32.16	4539	0.28	15.0	32.16	4553
50	29.48	4166	0.27	15.0	29.48	4180
40	26.07	3712	0.35	18.3	26.07	3730
30	20.56	2934	0.40	20.8	20.56	2924
20	13.30	1946	0.48	25.9	12.13	1971
10	6.12	967	0.55	30.1	6.12	997
0	1.96	158	0.59	32.4	1.96	190

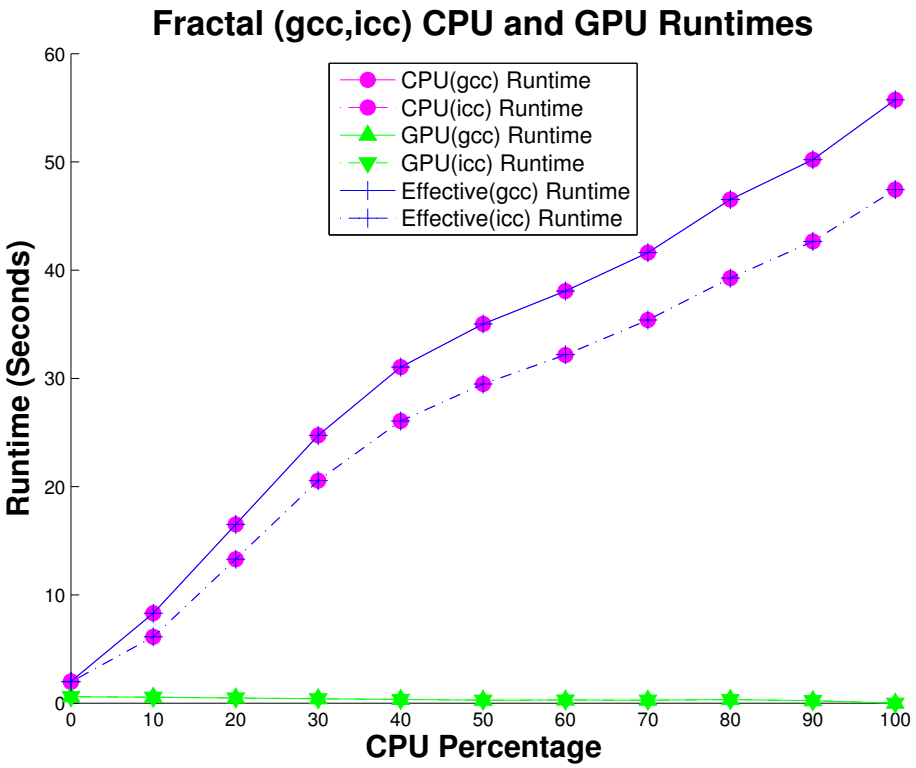

Figure 5.11: CPU and GPU fractal with icc, gcc runtime

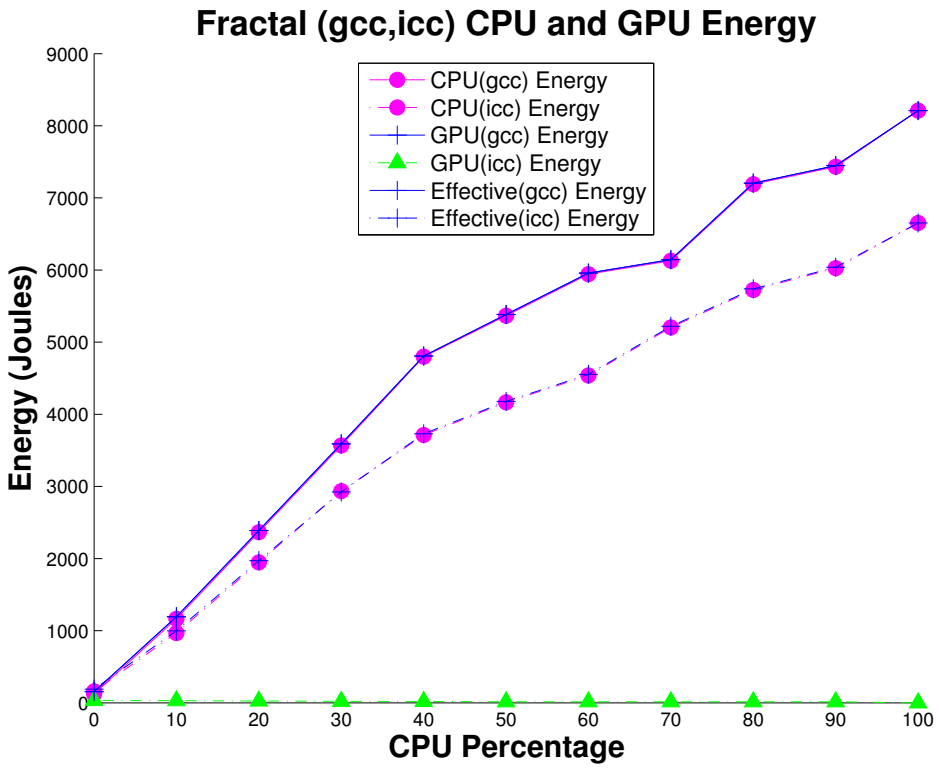

Figure 5.12: CPU and GPU fractal with icc, gcc energy

5.3.3 C++ and CUDA Breadth-first Search

The results for the two input file sizes of the applications are tabulated in Table 5.10 and graphed in Figures 5.13 and 5.14. For the smaller data size, optimal performance and energy points are similar, but not the same. They are CPU = 50% and 60%, respectively. For the larger data size, both the optimal performance and energy points coincide at CPU = 30%.

Table 5.10: Breadth-first search with CPU and GPU

BFS with CPU and GPU				
CPU %	**1M Time (sec)**	**1M Energy (J)**	**16M Time (sec)**	**16M Energy (J)**
100	5.844	380	95.533	8789
90	5.297	406	99.857	9912
80	5.083	409	91.004	9493
70	4.067	405	83.482	8867
60	4.036	368	74.657	8378
50	3.932	377	67.473	7883
40	4.24	413	59.129	7359
30	4.55	402	51.079	6495
20	4.606	386	54.016	6701
10	4.801	415	59.062	6803
0	5.152	460	63.355	8102

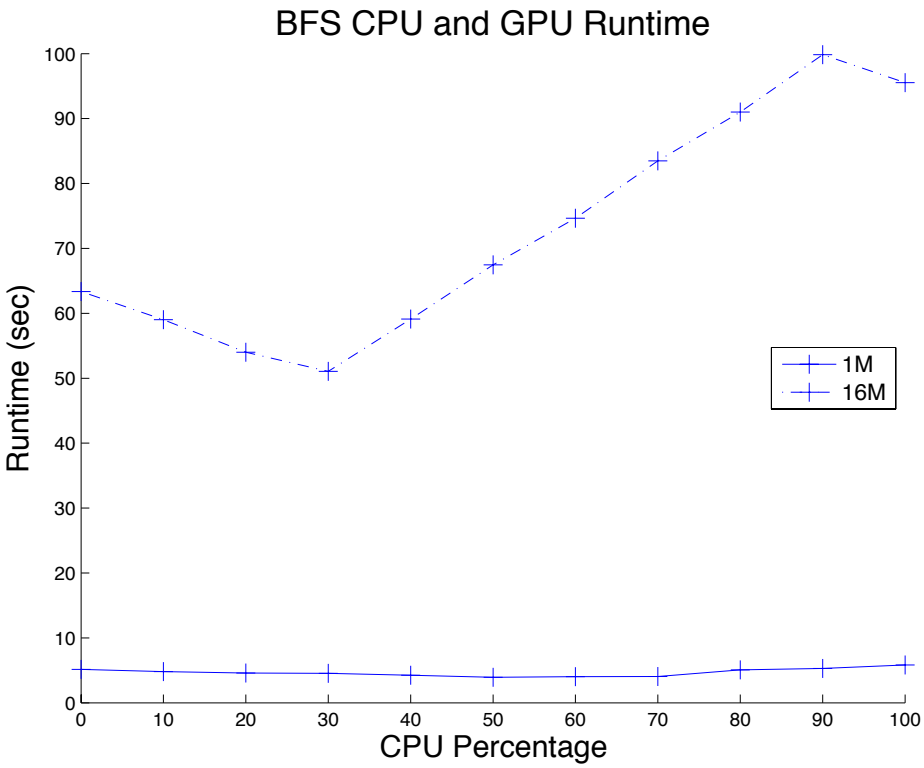

Figure 5.13: CPU and GPU BFS runtime

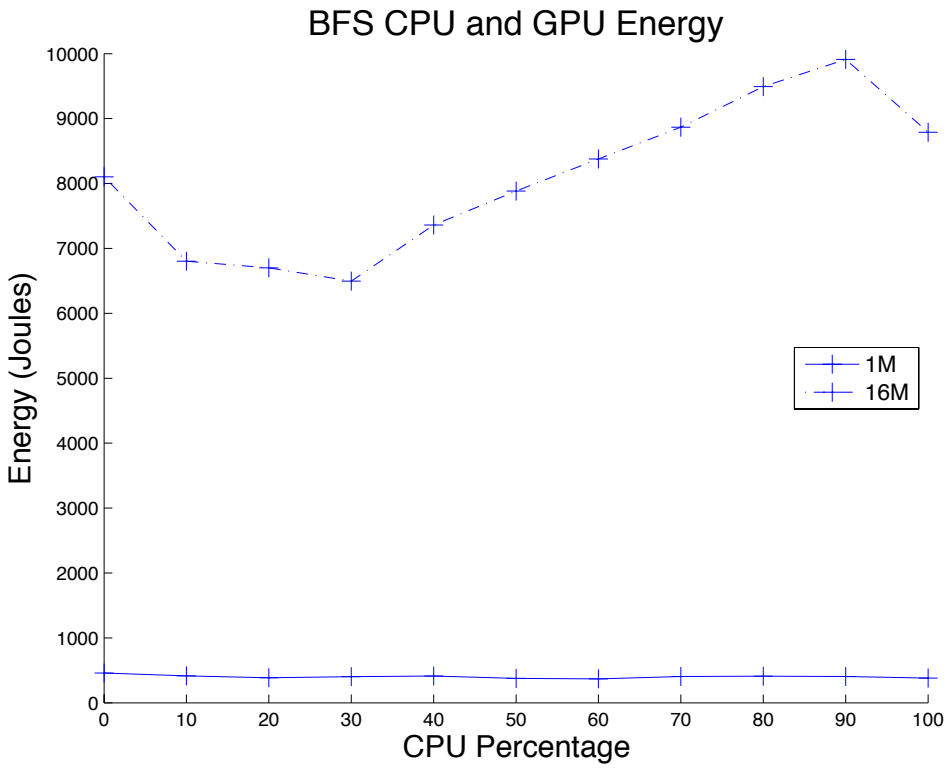

Figure 5.14: CPU and GPU BFS energy

5.3.4 C++ and OpenCL Matrix Multiplication

The results for the OpenCL version of the matrix multiplication are shown in Table 5.11 and graphed in Figures 5.15 and 5.16. Both optimal performance and energy points are when the work distributed to the CPU is 20%.

Table 5.11: Matrix multiplication C++ and OpenCL

Matrix multiplication C++ and OpenCL						
CPU%	**CPU Execution time (sec)**	**CPU Energy (J)**	**GPU Execution time (sec)**	**GPU Energy (J)**	**Effective Execution time (sec)**	**Total Energy (J)**
100	104.73	16118	0	0	104.73	16118
90	94.68	14292	0.06	5.5	94.68	14297
80	82.55	12291	0.44	23.97	82.55	12315
70	69.70	10963	1.45	91.97	69.70	11055
60	59.98	9189	3.41	265.55	59.98	9454
50	53.44	7749	6.63	592	53.44	8341
40	40.81	6406	11.42	1081	40.81	7487
30	32.57	5013	18.10	1771	29.26	6784
20	21.88	3271	26.95	2694	26.95	5965
10	12.87	2219	38.48	3884	38.48	6103
0	5.27	681	52.78	5388	52.78	6069

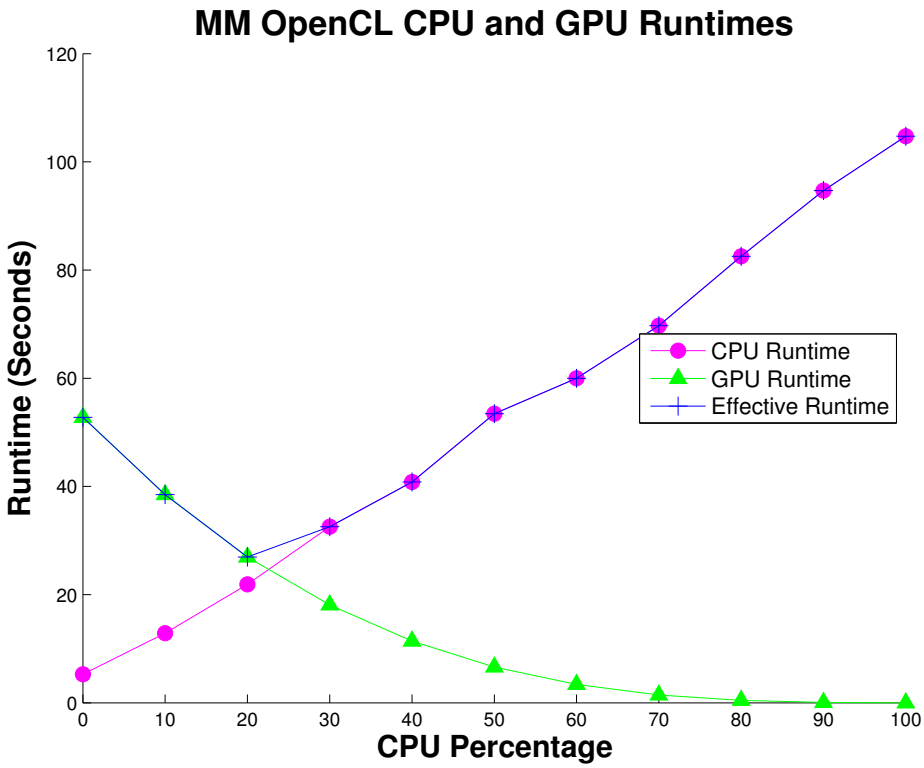

Figure 5.15: C++ and OpenCL matrix multiplication runtime

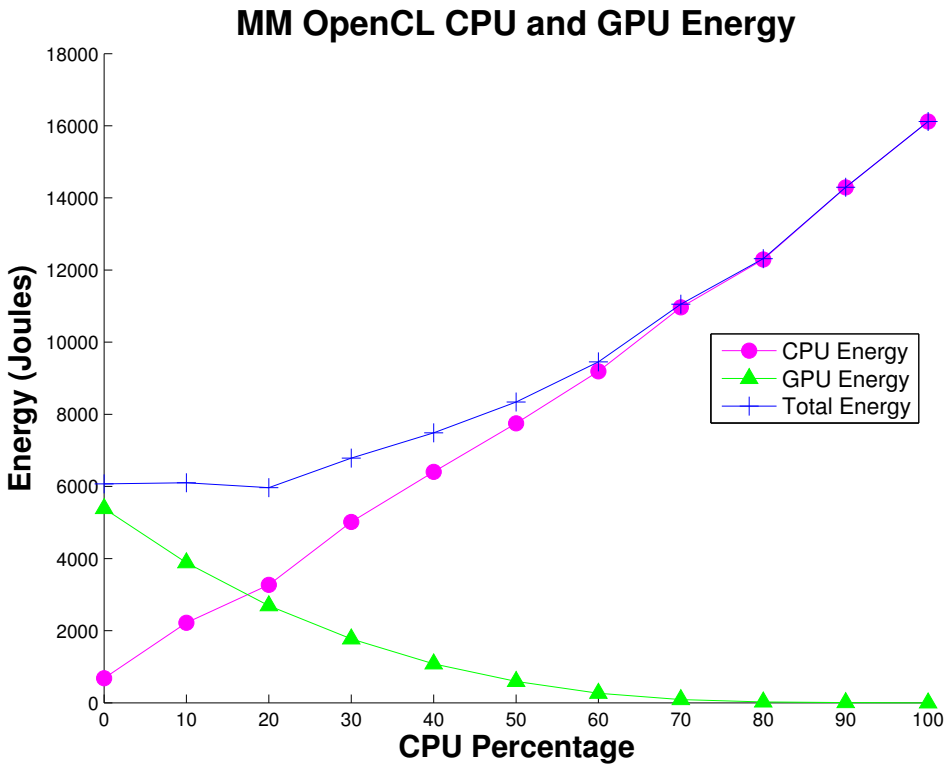

Figure 5.16: C++ and OpenCL matrix multiplication Energy

5.3.5 C++ and OpenCL Fractal

The results are tabulated in Table 5.12 and graphed in Figures 5.17 and 5.18. The optimal performance is when CPU = 20% and optimal energy is when CPU = 0%.

Table 5.12: Fractal C++ and OpenCL

Fractal C++ and OpenCL						
CPU%	**CPU Execution time (sec)**	**CPU Energy (J)**	**GPU Execution time (sec)**	**GPU Energy (J)**	**Effective Execution time (sec)**	**Total Energy (J)**
100	56.85	8383	0	0	56.85	8383
90	51.21	7580	0.05	5.12	51.21	7585
80	47.65	7331	0.38	20.8	47.65	7352
70	42.68	6269	1.17	74.2	42.68	6343
60	38.99	6085	2.51	191	38.99	6276
50	35.87	5463	4.86	396	35.87	5859
40	31.61	4909	8.11	649	31.61	5558
30	25.20	3628	11.76	920	25.20	4548
20	16.75	2413	16.44	1342	16.75	3755
10	7.42	1182	19.43	1981	19.43	3163
0	2.04	124	24.80	2521	24.80	2645

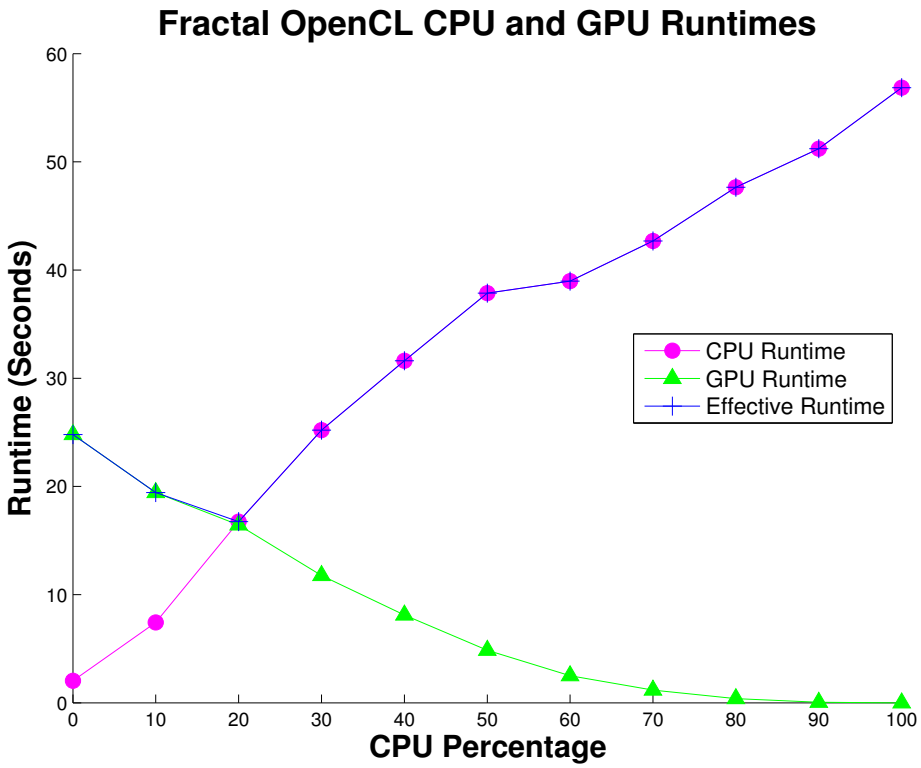

Figure 5.17: C++ and OpenCL fractal runtime

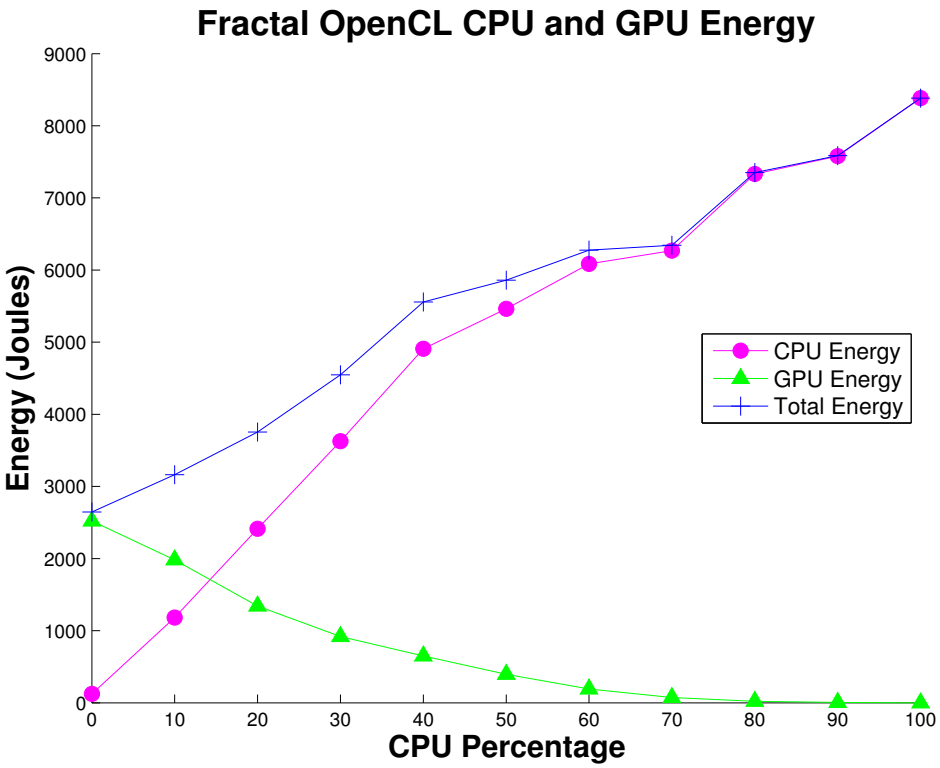

Figure 5.18: C++ and OpenCL fractal energy

5.3.6 C++ and OpenCL Breadth-first search (BFS)

For the smaller data set size, optimal performance point is when CPU = 10% and optimal energy point is when CPU = 0%. For the larger one, both performance and energy points are when CPU = 0%. The results are shown in Table 5.13 and graphed in Figures 5.19 and 5.20.

Table 5.13: Breadth-first search C++ and OpenCL

BFS C++ and OpenCL				
CPU %	**1M Time (sec)**	**1M Energy (J)**	**16M Time (sec)**	**16M Energy (J)**
100	5.979	546	142.75	11984
90	5.512	505	104.31	11398
80	5.064	479	94.84	10482
70	4.663	442	86.82	9595
60	4.278	416	77.93	8554
50	3.781	348	69.85	7811
40	3.285	341	61.77	6906
30	3.107	312	53.72	6008
20	3.098	274	45.16	5162
10	3.043	258	36.67	4352
0	3.425	248	29.31	3080

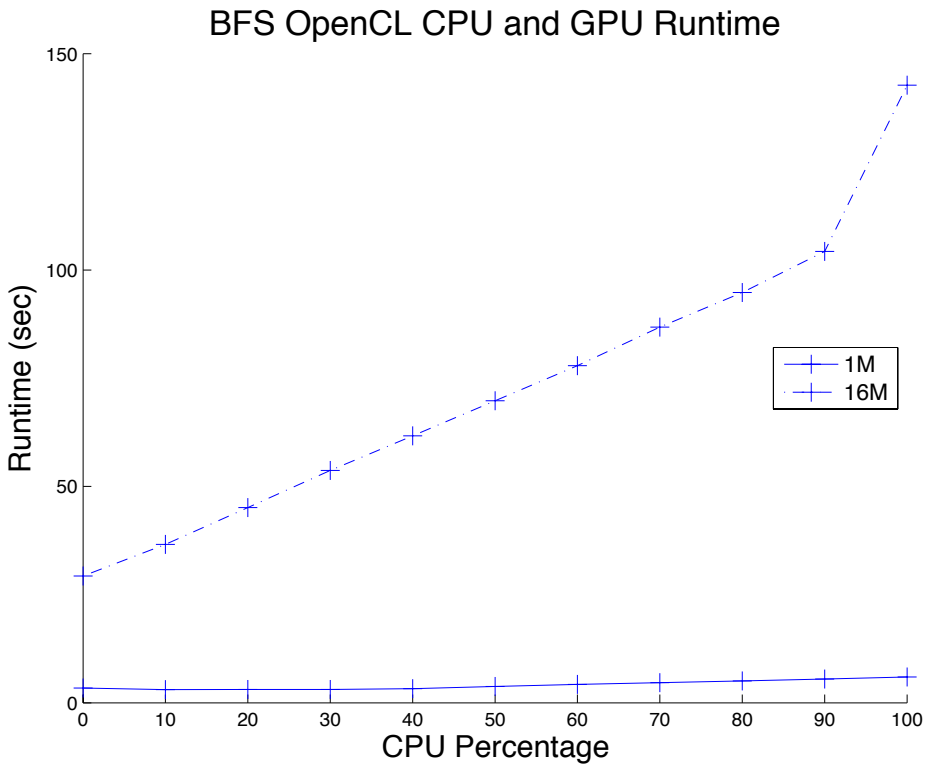

Figure 5.19: C++ and OpenCL BFS runtime

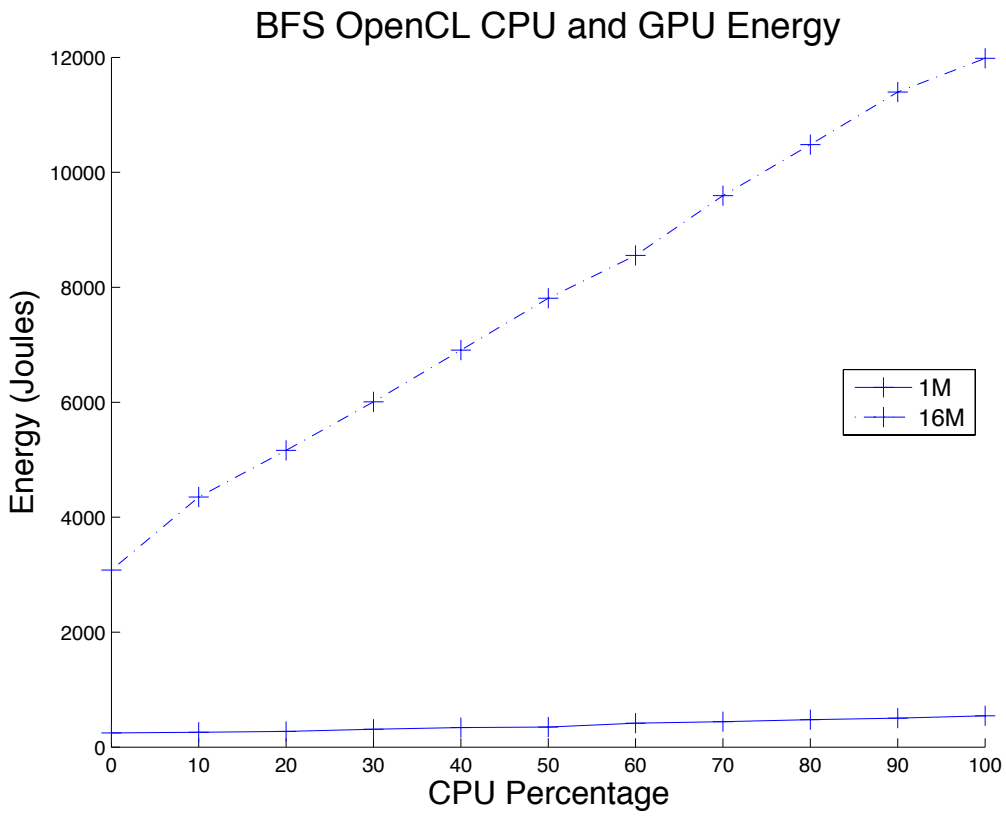

Figure 5.20: C++ and OpenCL BFS energy

5.4 Summary of Optimal Performance and Energy Points

A summary of the optimal performance and energy efficiency results from the previous sections for the applications is provided in Table 5.14.

Table 5.14: Summary of optimal points

Summary of Optimal Points			
Application	**Accelerator**	**Performance**	**Energy**
R Matrix	Xeon Phi	30:70	0:100
Matrix	Xeon Phi	40:60	40:60
Fractal	Xeon Phi	30:70	50:50
BFS 1M	Xeon Phi	86:14	100:0
BFS 16M	Xeon Phi	60:40	100:0
MM gcc	GPU	0:100	0:100
MM icc	GPU	0:100	0:100
Fractal gcc	GPU	0:100	0:100
Fractal icc	GPU	0:100	0:100
BFS 1M	GPU	50:50	60:40
BFS 16M	GPU	30:70	30:70
MM OpenCL	GPU	20:80	0:100
Fractal OpenCL	GPU	20:80	0:100
BFS 1M OpenCL	GPU	30:70	0:100
BFS 16M OpenCL	GPU	0:100	0:100

6. PREDICTING OPTIMAL HYBRID WORKLOAD DISTRIBUTION

Predicting optimal workload partitions can eliminate the exhaustive search which tests all different combinations to arrive at an optimal distribution. With matrix multiplication, fractal and breadth-first search hybrid applications, this chapter attempts to model and predict workload partitions for optimal performance and optimal energy based upon measured parameters.

A hybrid application can be modeled with five unique regions: 1) pre-computation, 2) CPU computation, 3) Xeon Phi or GPU computation, 4) CPU waiting for Xeon Phi or GPU, and 5) post-computation regions. The pre-computation region includes allocation or initialization of memory such as allocation and initialization of matrices used in matrix multiplication, or reading data from as reading the nodes and edges in BFS, or user input. The CPU is active in this time while the GPU and Xeon Phi have yet to be activated. The CPU computation region is when CPU performs the calculations associated with hybrid workload distribution. The GPU or Xeon Phi are active in the GPU or Xeon Phi computation region, which in general occurs at the same time as the CPU computation region. If the CPU completes it work allocated during the CPU computation region, it will wait for the GPU or Xeon Phi to finish. This region is the CPU wait region. The post-computation region occurs after the hybrid calculations are complete. Memory is de-allocated or reset for another computation, and files are closed are included in this region. The CPU is active in this region while the GPU or Xeon Phi is not.

The parameters associated with these distinct regions are time, average power, energy, work, and rate. These parameters are used in prediction of optimal workload partitions and are listed in Table 6.1.

Table 6.1: Parameters used for predictions

	Units	**Parameter**	**Description**
Time	seconds	T_C	CPU computation time
		T_X	Xeon Phi computation time
		T_G	GPU computation time
		T_{pre}	pre-computation time
		T_{post}	post-computation time
		T_{wait}	CPU waiting time
Energy	Joules	E_C	CPU computation energy
		E_X	Xeon Phi computation energy
		E_G	GPU computation energy
		E_{pre}	pre-computation energy
		E_{post}	post-computation energy
Work	10^9 calculations per second	W	All hybrid work
		W_C	CPU work
		W_X	Xeon Phi work
		W_G	GPU work
		W_{pre}	pre-computation work
		W_{post}	post-computation work
Power	Watts	P_C	CPU computation power
		P_X	Xeon Phi computation power
		P_{wait}	power consumed by CPU while waiting for GPU or Xeon Phi
		P_{pre}	pre-computation power
		P_{post}	post-computation power
Rate	10^9 calculations per second	R_C	CPU computation rate
		R_G	GPU computation rate
		R_X	Xeon Phi computation rate

Application time is the total time the application executes on the computing system. It is the sum of the pre- and post- computation times plus the longer of the Xeon Phi/GPU or CPU computation time and described with the following equation:

$$Application\ Time = T_{pre} + \max\left(T_C, T_X\ or\ T_G\right) + T_{post}$$

The application energy can be described similarly. The energy of the pre- and post- computation regions, the CPU computation energy, the energy of the CPU wait region if applicable, and energy of the Xeon Phi or GPU are summed together to give the total energy. This is the total energy the application consumes.

$$Application\ Energy = E_{pre} + E_C + \left(E_X\ or\ E_G\right) + E_{wait} + E_{post}$$

Power is the average power of the region and is the energy of the region divided by the time for that region. Each region has an associated average power.

$$Power = \frac{Energy}{Time}$$

Similar to time and energy, the application work can be described as the sum of the pre- and post-computation, the CPU computation plus the Xeon Phi or GPU computation work. The CPU does no work during the CPU wait region.

$$Application\ Work = W_{pre} + W_C + \left(W_X\ or\ W_G\right) + W_{post}$$

With the exception of the CPU wait region, each region has a rate of work. Rate of work is work divided by time.

$$R = \frac{W}{T}$$

For the application, the pre- and post-computation regions remain constant while the CPU, GPU or Xeon Phi computation regions vary based on the selected partition. Thus, in predicting and modeling, the parameters pre- and post-computation regions may

be discarded from the calculations of the optimal work partitions. Thus, the measurements of application power, time, energy and work must account for all the regions to accurately predict optimal performance or energy partitions.

6.1 Hybrid Code Running on CPU and Xeon Phi

Hybrid computation is a subset of the entire application. Thus, the time in the hybrid computation time does not include pre- or post- computation time, but just the maximum of either the CPU or Xeon Phi computation times.

$$Hybrid\ time = \max(T_C, T_X)$$

The data transfer to the Xeon Phi is included in the Xeon Phi computation time. Optimal performance occurs when the time periods of the two processors overlap exactly, when the Xeon Phi computation time equals CPU computation time.

$$T_C = T_X$$

The sum of the CPU work and Xeon Phi work is the total hybrid work. This can be described by the following equation:

$$W = W_C + W_X .$$

Work divided by time yields rate. Rate describes the computation rate of the CPU or Xeon Phi and is dependent on hardware, software, memory and cache use, and data transfer. The rate of the CPU and Xeon Phi is given by the two following equations, respectively.

$$R_C = \frac{W_C}{T_C}$$

$$R_X = \frac{W_X}{T_X}$$

Using the previous equations, optimal performance for the CPU and Xeon Phi are related by the following equation.

$$\frac{W_X}{R_X} = \frac{W_C}{R_C}.$$

Because work of the CPU and work of the Xeon Phi are related to the total work, they can be described by the two following equations respectively.

$$W_C = \alpha W$$

$$W_X = (1-\alpha)W$$

$$where\ \alpha : \alpha \in [0:1]$$

Combining the previous equations yields the computational mapping for optimal performance is as follows:

$$\frac{W(1-\alpha)}{R_X} = \frac{\alpha W}{R_C}$$

$$\alpha_{perf} = \frac{R_C}{R_X + R_C}$$

$$where\ \alpha_{perf}\ represents\ optimal\ performance\ partiton.$$

For optimal performance, the partition depends upon the measured rates of the CPU and Xeon Phi.

Table 6.2: CPU and Xeon Phi partitions

Parameter	Description
α	CPU partition
α_{perf}	Optimal CPU partition for performance

The optimal energy partition is defined by the minimum energy expended by the application. Because the pre- and post- computation energies remain fixed as the hybrid work is partition, those energy parameters may be discarded when calculating optimal energy. Optimal energy can be related by the following equation:

$$Optimal\ Energy = \min E\left(\alpha : \alpha \in [0:1]\right) = E_C + E_X + \max\left(E_{wait}, 0\right)$$

While the CPU waiting energy cannot be negative, it is included to derive the next equation. Consequently, to determine the optimal energy distribution, only the CPU and Xeon Phi computational energies and the CPU waiting energy effect the calculation. In substituting the equations above, the optimal energy is given with the following equation:

$$\min E\left(\alpha : \alpha \in [0:1]\right) = \frac{\alpha \cdot P_C}{R_C} + \frac{(1-\alpha)\cdot P_X}{R_X} + \max\left(\frac{1-\alpha}{R_X} - \frac{\alpha}{R_C}, 0\right)\cdot P_{wait}$$

The CPU waiting power, CPU rate and computation power, Xeon Phi rate and computation power were measured for the applications and are listed in Table 6.3.

Table 6.3: CPU and Xeon Phi measured parameters

	CPU			Xeon Phi	
	P_{wait}	R_C	P_C	R_X	P_X
R MM	47.3	150.3	170	322.5	196.5
MM	55.5	46.96	185.9	74.10	177.7
Fractal	55.5	28.9	139.3	43.6	154
BFS	54.9	179.6	182	38.7	126.2
BFS	53	5.12	153	3.55	141

Performance and energy optimal points were calculated and predicted from the measured parameters. They are shown in Table 6.4 with very good agreement between measured and predicated. While data was taken every 10%, BFS with 1M nodes was tested at additional workload distributions to determine the optimal performance point of 86%.

Table 6.4: CPU and Xeon Phi measured and predicted optimal performance and energy

Predicted and Measured Hybrid CPU + Xeon Phi Optimal				
	Performance		**Energy**	
	Measured	**Predicted**	**Measured**	**Predicted**
R Matrix	30:70	32:68	0:100	0:100
Matrix	40:60	39:61	40:60	38:62
Fractal	30:70	39:61	50:50	40:60
BFS 1M	86:14	82:18	100:0	100:0
BFS 16M	60:40	60:40	100:0	100:0

For matrix multiplication in the R scripting language, a second optimal performance point exists at 0:100, representing a difference of just 0.3%

6.2 Hybrid Code Running on CPU and GPU

The CPU and GPU follow similarly for work partition prediction. The optimal performance is given as follows:

$$\alpha_{perf} = \frac{R_C}{R_G + R_C}$$

And the optimal energy for the CPU and GPU workload partition is given below:

$$\min E\left(\alpha : \alpha \in [0:1]\right) = \frac{\alpha \cdot P_C}{R_C} + \frac{(1-\alpha)\cdot P_G}{R_G} + \max\left(\frac{(1-\alpha)}{R_G} - \frac{\alpha}{R_C}, 0\right)\cdot P_{wait}$$

The measure parameters for the applications were tallied and are listed in Table 6.5.

Table 6.5: CPU and GPU measured parameters

CPU and GPU Measured Model Parameters					
	CPU			GPU	
	P_{wait}	R_C	P_C	R_G	P_G
MM gcc	206.5	19.8	154.5	864	52.5
MM icc	261	45.8	181.9	1014	52.5
Fractal gcc	61.1	23.8	147.3	661	54.9
Fractal icc	80.6	28.0	140.2	678	54.9
BFS 1M	57.9	97.6	126	45	115
BFS 16M	72.3	6.23	102.6	11.0	92.9
MM OpenCL	31.8	19.1	153.8	37.9	102.1
Fractal OpenCL	45.3	23.4	147.4	53.5	101.6
BFS 1M OpenCL	42.3	95.3	75.8	208	30.0
BFS 16M OpenCL	44	3.8	76.9	67.8	55.3

The measured parameters were then used the model to predict the optimal partition for performance and energy and are listed in Table 6.6. The granularity for these measurements is 10%.

Table 6.6: CPU and GPU measured and predicted optimal performance and energy

CPU and GPU Optimal				
	Performance		**Energy**	
	Measured	**Predicted**	**Measured**	**Predicted**
MM gcc	0:100	3:97	0:100	2:98
MM icc	0:100	4:96	0:100	4:96
Fractal gcc	0:100	3:97	0:100	2:98
Fractal icc	0:100	3:97	0:100	2:98
BFS 1M	50:50	46:54	60:40	46:54
BFS 16M	30:70	36:64	30:70	36:64
MM OpenCL	20:80	33:67	0:100	2:98
Fractal OpenCL	20:80	30:70	0:100	2:98
BFS 1M OpenCL	30:70	31:69	0:100	2:98
BFS 16M OpenCL	0:100	5:95	0:100	2:98

The performance in the ranges between 10% and 30% CPU present at most a 2% difference and yield a range of minimal runtimes. In general, there is very good agreement between predicted and measured.

7. CONCLUSION

This section summarizes the contributions and work done in this thesis, and provides recommendations for work in the future

7.1 Contribution

Historically, it has been difficult to predict the best performance and energy efficiency operating points of hybrid applications. For this thesis, applications are developed to enable measurement of hybrid performance and energy efficiency over the entire range of workload distributions. From the results, for many cases, the optimal performance and optimal energy operating points do not coincide. And for these applications, the selection of the workload distribution for either the optimal performance or energy is not obvious. Thus, this thesis presents prediction methodologies for optimal performance or optimal energy operating points for these sample applications, which can likely be extended to other applications. The prediction equations not only extend to the CPU and GPU but also CPU and Xeon Phi combinations for a variety of sample applications. In addition, the methodology accurately predicts the CPU and Xeon Phi workload division in an R scripting language application using an entirely different workload division technique, the environment variables. The optimal hybrid performance operating point may benefit other R applications, which typically run with multiple hours long execution time, providing a execution time performance and allowing larger data sets to be run. In addition, power data from the on-chip power monitoring of the CPU and accelerators and measured performance data can be applied to the equations to provide accurate predictions.

Many times, programs can be rather difficult to adapt to hybrid use. This thesis presents a general guideline on adapting already-parallel applications to hybrid use of a CPU and Xeon Phi or CPU and GPU. These methods provide guidelines and workload division examples. In addition, for the R scripting language, this thesis provides the information on open source R language enabled to use MKL and a detailed list of environment variables that allow parallel and hybrid use.

7.2 Future Work

In this work, I presented a method to predict workload division for hybrid applications, enabling optimal energy or performance predictions. There are many directions to extend this work. First, verify and confirm predictions with other applications. Although the applications cover computational intensive applications and a data intensive transfer application, there are many other applications for which the prediction methodology can be verified. Second, because the measurements on these applications were performed on a node on a single system, measurement on other platforms using other CPUs and other GPUs, and next generation of Xeon Phi. Third, the applications can be optimized for more efficient memory accesses, execution time or energy, thread affinity and hyperthreading. All of these can tweak the optimal points. High transfer rates between CPU and Xeon Phi or GPU can be optimized. And finally, a software enabled machine learning to automatically sample, measure and provide optimal hybrid distribution for performance or energy can be added to programs. This would automate the steps done in this thesis to create a very quick, efficient, application.

LITERATURE CITED

[1] R. Gee, X. Feng, M. Burtscher, Z. Zong, "Performance and Energy Aware Coorperative Hybrid Computing", in *Proceedings of the 11th ACM Conference on Computing Frontiers*, May 2014.

[2] top500.org. Accessed June 2016.

[3] green500.org. Accessed June 2016.

[4] http://glennklockwood.blogspot.com/2013/09/intels-xeon-phi-uptake-measured-from.html. Accessed June 2016.

[5] C. Liu, A. Sivasubramaniam, M. Kandemir, M.J. Irwin, "Exploiting Barriers to Optimize Power Consumption of CMPs," in *Proceedings of the IEEE 19th International Parallel and Distributed Processing Symposium*, p. 5a, 2005.

[6] R. Ge, X. Feng, W.-c. Feng, K.W. Cameron, "Cpu miser: A Performance-directed, Run-time System for Power-aware Clusters," in *International Conference on Parallel Processing (ICPP)*, p. 18, IEEE, 2007.

[7] C. -h. Hsu, W. -c. Feng. "A Power-aware Run-time System for High Performance Computing," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, p. 1. IEEE Computer Society, 2005.

[8] V.Pallipadi, A. Starikovskiy, "The Ondemand Governor," in *Proceedings of the Linux Symposium,* vol. 2, pp. 223-238, July 2006.

[9] C. -H. Hsu, U. Kremer, "The design, implementation and evaluation of a compiler algorithm for CPU energy reduction," in *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 38-48. ACM 2003.

[10] L. A. Barroso, U. Holzle. “The case for energy-proportional computing,” *IEEE Computer* vol. 40, no. 12, pp. 33-37, 2007.

[11] A. Vhdat, A. Lebeck. C. S. Ellis. “Every joule is precise: The case for revisiting operating system design for energy efficiency,” in *Proceedings of the 9th workshop on ACM SIGOPS European Workshop: beyond the PC: new challenges for the operating system*, pp. 31-36, ACM 2000.

[12] Y.C. Lee and A.Y. Zomaya. "Energy Conscious Scheduling for Distributed Computing Systems under Different Operating Conditions", in *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, issue 8, pp. 1374-1381, December 2010.

[13] S.Hong and H. Kim. “An Integrated GPU power and Performance Model,” in *Proceedings of the 37th Annual International Symposium Computer Architecture*, vol. 38, no. 3, pp. 280-289, 2010.

[14] S. Collange, D. Defour, A. Tisserand, “Power Consumption of GPUs from a Software Perspective,” in *9th International Conference on Computational Science*, pp. 922-931, 2009.

[15] S. Che, J.W. Sheaffer, K. Skadron, "Dymaxion: optimizing memory access petterns for heterogeneous systems", in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC),* November 2011.

[16] P. Bailey, D.K. Lowenthal, V. Ravi, B. Rountree, M. Schulz, B.R. de Supinski, “Adaptive Configuration Selection for Power-Constrained Heterogeneous Systems,” in *Parallel Processing (ICPP) 2014 43rd International Conference*, pp. 371-380. IEEE, 2014.

[17] Q. Wu, M. Martonosi, D.W. Clark, V.J. Reddi, D. Connors, Y. Wu, J. Lee, D. Brooks, "A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 271-282, Nov. 2005.

[18] S. Song, C. -Y. Su, B. Rountree, K.W. Cameron, "A Simplified and Accurate Model of Power-Performance Efficiency on Emergent GPU Architectures," *IEEE 27th International Symposium on Parallel And Distributed Processing (IPDPS)*, pp. 20-24, May 2013.

[19] C.-K.Luk, S. Hong, H.Kim. "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42*, pp. 45-55, 2009.

[20] J.Dean and S. Ghemawat. "MapReduce," Communications of the ACM, vol. 51, p. 107, Jan. 2008.

[21] V.T. Ravi, W. Ma, D. Chiu, G. Agrawal. "Compiler and Rutime Support for Enabling Generalized Reduction Computations on Heterogeneous Parallel Configurations," in *Proceedings of the 24th ACM International Conference on Supercomputing (ICS)*, pp. 137-146, 2010.

[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," In *Proceedings of the IEEE International Symposium on Workload Characteriztion (IISWC)*, pp. 44-54, Oct. 2009.

[23] T. Scogland, B. Rountree, W. -c. Feng, B. de Supinski, “Heterogeneous Task Scheduling for Accelerated OpenMP,” in *Proceedings of the IEEE 26th International Parallel Distributed Processing Symposium*, pp. 144- 155, May 2012.

[24] J. Fang, A.L. Varbanescu, H. Sips, L. Zhang, Y. Che, C. Xu, "An Empirical Study of Intel Xeon Phi", in *arXiv preprint arXiv:1310.5842vs [cs.DC] 20 Dec 2013.*

[25] G. Lawson, M. Sosonkina, U. Shen, “Energy Evaluation for the Applications with Different Thread Affinities on the Intel Xeon Phi,” in *2014 International Symposium on Computer Architecture and High Performance Computing workshop (SBAC-PADW*), pp. 54-59, Oct 2014.

[26] J. Wood, Z.L. Zong, Q.J. Gu. “Energy and Power Characterization of Parallel Programs Running on Intel Xeon Phi,” *in Proceedings of the Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, in conjunction with ICPP 14, pp. 265- 272, Minneappolis, MN, Sept. 2014.

[27] O.G. Lorenzo, T.F. Pena, J.C. Cabaleiro, J. C. Picel, F.F. Rivera, D.S. Nikolopoulos, “Power and Energy Implications of the Number of Threads Used on the Intel Xeon Phi. Multicore and GPU Programming,” Paper presented at Second Congress of GPU Programming, Caceres, Spain, pp. 1-8, 2015.

[28] Y. Shao, D. Brooks, “Energy Characterization and Instruction Level Energy Model of Intel’s Xeon Phi Processor,” in *Proceedings of 2013 International Symposium on Low Power Electronics Design*, pp. 389- 394, 2013.

[29] B. Li, H.-C. Chang, S. Song, C. Su, T. Meyer, J. Mooring, K. Cameron, "The Power-Performance Tradeoffs of the Intel Xeon Phi on HPC Applications," *2014 IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 1448-1456, 2014.

[30] http://nsf.gov/awardsearch/showAward?AWD ID=1305359

[31] "Intel Xeon Phi Coprocessor System Software Developers Guide," http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf. Accessed July 2014.

[32] "Intel RAPL," https://01.org/blogs/tlcounts/2014/running-average- power-limit-?-rapl. Accessed June 11, 2015.

[33] "Intel Xeon Phi coprocessor Power management Configuration: Using the micsmc command-line Interface," https://software.intel.com/en- us/blogs/2014/01/31/intel-phi-coprocessor-power-management- configuration-using-the-micsmc-command. Accessed July 9, 2014.

[34] https://developer.nvidia.com/nvidia-system-management-interface. Accessed November 2014.

[35] M. Burtscher, I. Zecena, Z. L. Zong, "Measuring GPU Power with the K20 Built-in Sensor," in *Proceedings of the 7th Workshop on General Purpose Processing Using GPUs (GPGPU 7)*, in conjunction with ASPLOS 14, Salt Lake City, Mar. 2014.

[36] https://software.intel.com/en-us/articles/intel-math-kernel-library-documentation. Accessed April 2016.

[37] https://www.r-project.org. Accessed January 2016.

[38] http://www.revolutionanalytics.com. Accessed January 2016.

[39] J. Shen, A.L. Varbanescu, P. Zou, Y. Lu, H. Sips, "Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms", in *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS 2014, pp. 241-250.

[40] https://www.olcf.ornl.gov/tutorials/compiling-mixed-gpu-and-cpu-code/. Accessed April 2016.

[41] *An Introduction to Parallel Programming*, Peter Pacheco, ISBN-13: 978-0123742605