2011

Power management and optimization

Hari Sundararajan
Louisiana State University and Agricultural and Mechanical College, hari.s@alumni.lsu.edu

POWER MANAGEMENT AND OPTIMIZATION

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

In

The Department of Electrical and Computer Engineering

By
Hari Sundararajan
B.S., Louisiana State University, 2008
December 2011

ACKNOWLEDGEMENTS

First and foremost, I would like to thank God for His gracious presence in enabling me to achieve everything I have so far, and for always having stood by me.

I would like to extend my most sincere gratitude to the following people without whom it would have been impossible to complete this work.

My sincere thanks to my advisor, **Dr. Thomas Sterling** who has been continuously supportive and patience with me. His constant guidance and advice throughout is the only reason I have been able to successfully complete this project. No amount of words will be sufficient to express my gratitude to Dr. Sterling. **Maciej Brodowicz ,** my supervisor, deserves special mention for having spent a lot of time in helping me throughout the project. **Dr. Ramanujam** has been a constant source of guidance and support throughout my research.

I would also like to extend my gratitude to my parents for their faith in me throughout my education.

A number of other people deserve mention for their unwavering support. Rajesh Sankaran, my senior who has always been someone I could rely upon, Richie Sajan, my roommate of 6 years who has always remained a constant companion throughout my graduate career, Chirag Dekate for his steering me in the right direction, Vinay Amatya for his help in setting up all the necessary hardware, Srikanth Jandhyala for motivating me to get into the field of High Performance Computing, Terrie Bordelon for his constant support, Nevin Thomas George and all my undergraduate friends for constantly cheering me on, Viji Balachandran for being a pillar of support, Uma Ramanujam, Karthik Omanakuttan, Dr. Horst Reinhard Beyer and Subramanian Venkatachalam for everything they have done for me.

Finally, I would like to make a special mention of Dhivya Manivannan, without whose love and motivation this work and document would have never taken shape.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

After many years of focusing on "faster" computers, people have started taking notice of the fact that the race for "speed" has had the unfortunate side effect of increasing the total power consumed, thereby increasing the total cost of ownership of these machines. The heat produced has required expensive cooling facilities.

As a result, it is difficult to ignore the growing trend of "Green Computing," which is defined by San Murugesan as "the study and practice of designing, manufacturing, using, and disposing of computers, servers, and associated subsystems – such as monitors, printers, storage devices, and networking and communication systems – efficiently and effectively with minimal or no impact on the environment" [1].

There have been different approaches to green computing, some of which include data center power management, operating system support, power supply, storage hardware, video card and display hardware, resource allocation, virtualization, terminal servers and algorithmic efficiency. In this thesis, we particularly study the relation between algorithmic efficiency and power consumption, obtaining performance models in the process. The algorithms studied primarily include basic linear algebra routines, such as matrix and vector multiplications and iterative solvers.

Our studies show that it if the source code is optimized and tuned to the particular hardware used, there is a possibility of reducing the total power consumed at only slight costs to the computation time. The data sets utilized in this thesis are not significantly large and consequently, the power savings are not large either. However, as these optimizations can be

scaled to larger data sets, it presents a positive outlook for power savings in much larger research environments.

CHAPTER 1 - INTRODUCTION

Since the dawn of computing, the only metric that has always been focused on for improving has been performance. Faster has always meant better and towards this end, there has been significant progress and research aimed at increasing processor speeds.

Computer algorithms have generally been constructed, and programs written for sequential computation, wherein the program is a sequence of steps to be followed in order to implement the algorithm. These steps are executed by a computer, or to be precise, the central processing unit of a computer. Traditionally, computers have always had a single central processing unit.

Efficiency of computers has always been measured in terms of the time the computer takes to solve and work through the algorithms it has been programmed for. Since the computer executes the steps of the algorithm in a sequential fashion, the total time taken by a computer to run a program is given by the product by the number of instructions in the program and the average time taken to execute one instruction. Therefore, reducing either of the above factors yields a "better-performing" computer.

Research in algorithms have led to their improvement, which effectively translates into smaller number of instructions required. However, there can only be so much improvement done to an algorithm before the effect of further research becomes minimal. As a result, the most popular way then of improving the performance of computers was through the process of frequency scaling.

Frequency scaling refers to the technique of increasing a processor's frequency. Between the period from mid-1980s through mid-2000s, processor frequency was increased every year. As mentioned earlier, the running time of a particular program is the product of the number of

instructions in the program, the average number of cycles required per instruction (a value dependent on the program) and the average time the processor takes to execute one cycle. The third value, the average time a processor takes to execute one cycle is the inverse of the advertised processor frequency, and therefore, an increase in processor frequency directly results in a reduced run time.

Up until around 2004, chip manufacturers got away with constantly increasing frequency of their microprocessors. An increase in the processor frequency allowed any program in the market to take lesser time to execute, and hence "run faster." However, the factor that was ignored in this process was power. The power consumed by a microprocessor is directly related to the capacitance being switched in every cycle, the square of the voltage and the frequency. Consequently, climbing higher up the frequency scale implied climbing up the power consumed as well. As the power consumed is dissipated as heat, measures had to be taken to cool the central processing units.

Around 2004, Intel stopped production of their single core line of microprocessors, and the termination of its Tejas and Jayhawk processors [2] is often heralded as the birth of multi core processors, and a new parallel programming paradigm. Manufacturers started including multiple cores within their chips, and this required a significant change in the way the algorithms were implemented. However, this still doesn't tackle the overall problem of cooling the microprocessors.

Over time, the birth of commodity clusters has led to super computers that are able to perform hundreds of trillions of floating point operations per second. However, the massive amount of power consumed by these computers necessitates the construction of massively extravagant cooling facilities as well. Seager of Lawrence Livermore National Laboratory observes that

energy bill required to run the supercomputers there are of the order of $8 million per year to power up and $6 million [3] per year to cool. Similarly, the building for the ASC Q supercomputer at Los Alamos National Laboratory costs nearly $100 million to construct.

1.1 BACKGROUND

One of the original attempts at building power efficient machines included Green Destiny [4], a 240 processor supercomputer that consumed 3.2 kiloWatts of power when booted diskless. One of the primary advantages of this computer was its completely lack of unscheduled downtime during its two-year run. Arrhenius equation as applied to microelectronics states that a compute node is twice as likely to fail if its temperature raises by 18 degrees F. As a result, keeping the power and temperature low also helps with reducing unscheduled downtimes, as Green Destiny was able to prove.

The concept of energy efficient supercomputing came into picture when Sharma et al. [5] made the case for a Green 500 List to supplement the Top 500 List, which ranks computers solely on the speed metric. They argued for a new list that would also rank computers based on new metrics that would indicate the power consumed, such as FLOPS / watt.

There have also been significant advances made in the industry towards increased power efficiency. PA Semi announced their PWRficient ™ Processor family that is a derivative of IBM's POWER Architecture ™ aims to "really drive a breakthrough in performance per watt" [6].

There have been other works conducted along similar topics. In their paper on power profiling, Feng et al. [20] have concluded that power profiles are regular corresponding to application

characteristics and that for fixed problem sizes increasing the number of nodes always increases energy consumption but does not always improve performance.

In this project, frequency is scaled within certain limited values, and studies are made. Similar research was performed by Freeh [21] , where they use high performance cluster nodes that are frequency and voltage scalable, and save energy by scaling down the CPU. Their paper primarily deals with the NAS benchmark and they conclude that the benchmark exhibits a better energy time trade off using multiple phases, where a pre set frequency is assigned to each phase heuristically. Similar research is also done by the same author in his paper on energy time tradeoffs in MPI programs [22].

In their paper on improvement of power-performance efficiency for high end computing [23], Ge et al. propose a novel approach to utilize parallel performance inefficiencies that are typically found in non-interactive, distributed scientific applications and conserve energy using dynamic voltage scaling. They present a framework to analyze and optimize the power performance while using dynamic voltage scaling. Similar work in the field has also been performed by Hsu et al. [24] where they leverage a commodity technology, "dynamic voltage and frequency" scaling to implement power aware algorithms in commodity HPC systems.

1.2 MOTIVATION

The aim of this project is to take a commercial off the shelf workstation, monitor and observe the power consumed for a variety of work loads and in the process of correlating the various factors involved, come up with a performance model that can be used to predict and improve power consumption depending on what the workstation is used for.

Even though certain factors are outside the programmer's control, we believe that knowledge of the hardware used and its specifications will enable programmers to write code that draws power efficiently. While efficiency in terms of time is still the primary metric sought after, certain compromises can be allowed if the result is an improvement along the power metric, since that would translate into increased up times and lowered total cost of ownership.

CHAPTER 2 - HARDWARE ENVIRONMENT DESCRIPTION

2.1 WORKSTATION DETAILS

The workstation used for testing purposes is a Hewlett Packard xw9400 workstation. The total RAM installed in the workstation is 16GB, with 8 Dual Inline Memory Modules (DIMM) of capacity 2GB each. The RAM modules DDR2 Synchronous, with a frequency of 667 Mhz each and are 64 bit wide.

2.2 CPU DETAILS

The workstation is outfitted with 2 Quad Core AMD Opteron ™ Processor 2384. This provides the programmer with 8 cores to operate on. Additionally, if the operating system supports it, the processor can be made to run at different frequencies. This sets itself up for later testing, as frequency is a significant factor affecting power consumption. Additionally, the CPUs have 512 KB of L1 cache, 2MB of L2 cache and 6 MB of L3 cache. The CPUs are 64 bit capable

2.3 GPU DETAILS

The workstation also features a NVIDIA GPU. The GPU details, as provided by a *deviceQuery* listing of the NVIDIA CUDA API lists the following –

CUDA Device Query (Runtime API) version (CUDART static linking)

There is 1 device supporting CUDA

Device 0: "GeForce GTX 260"

CUDA Driver Version: 4.0

CUDA Runtime Version: 3.10

CUDA Capability Major revision number: 1

CUDA Capability Minor revision number: 3

Total amount of global memory: 939196416 bytes

Number of multiprocessors: 27

Number of cores: 216

Total amount of constant memory: 65536 bytes

Total amount of shared memory per block: 16384 bytes

Total number of registers available per block: 16384

Warp size: 32

Maximum number of threads per block: 512

Maximum sizes of each dimension of a block: 512 x 512 x 64

Maximum sizes of each dimension of a grid: 65535 x 65535 x 1

Maximum memory pitch: 2147483647 bytes

Texture alignment: 256 bytes

Clock rate: 1.35 GHz

Concurrent copy and execution: Yes

Run time limit on kernels: No

Integrated: No

Support host page-locked memory mapping: *Yes*

Compute mode: *Default (multiple host threads can use this device simultaneously)*

Concurrent kernel execution: *No*

Device has ECC support enabled: *No*

It must be observed that double precision is supported in CUDA compute capability 1.3 and above, and the GTX 260 used for testing purposes works with CUDA compute capability 1.3. However, CUDA's double precision support is turned off by default; the compiler converts doubles into floats inside of kernels. While this might not cause a significant issue in itself, the problem is more likely to arise because the host (CPU) code remains unchanged. This causes the double precision values on the CPU to be read as multiple single precision values on the GPU. In order to prevent this, the following flag is passed to the compiler when it is invoked –

nvcc –gpu-name sm_13

Additionally, the NVIDIA GTX 200 Technical Brief [7] states that while the multi processors on the GTX 200 series have 8 single precision floating point ALUs (one on each core), there is only one double precision ALU per multiprocessor that is shared between all the cores. Consequently, applications where the execution time is predominantly dependent on computations are likely to see a significant slow down of up to 8. However, the aim of this research is to focus on the power consumption, and the tests will be performed accordingly.

2.4 POWER MONITORING HARDWARE DETAILS

The following are the hardware being utilized for obtaining power measurements –

AC power meter – Yokogawa PR 300

DC meter – Texmate DI-503 E with IQD2 input board (quad channel 50 mV converter)

Hall effect transducers: LA 55-P / SP23 by LEM

Communications Bridge – Net485 from Gridconnect

A brief description of the hardware is provided below.

The Yokogowa PR300 is an industry standard panel mounted power and energy meter for monitoring energy consumption. For the purpose of this research, this device is used for measuring active power. It features a large triple display and provides for a wide choice of measurement items. It is equipped with a standard RS-485 communication function and it is capable of Ethernet communication, which is the method used in this research as well. The device can be used in a variety of different configurations, such as single phase two wire system, single phase three wire system, or three phrase systems. The device has been setup for a simple single phase two wire system [8].

The Texmate DI-503E is a programmable meter controller. It is equipped to handle upto 4 input channels, and in this research, the inputs are connected to a 3V rail, a 5V rail, and two 12V rails. The GPU is connected to a 12 V rail, and hence significant fluctuations are observed along this line when performing measurements of GPU computations. Similarly, the CPU is powered by the other 12V rail and this is the rail monitored when performing computations that involve CPU factors, such as number of CPUs and CPU frequency [9].

The LA 55 –P / SP23 is a closed loop compensated current transducer that uses the Hall effect. This eliminates the need for actual wire connections [10].

The NET485 bridge allows the connection between the monitoring computer and the RS485 interface on the Yokogowa PR300, which in turn enables remote serial communications [11].

2.5 EXPERIMENTAL SETUP

The power monitoring hardware as described above is connected directly to a workstation (also described earlier). A separate profiling computer is connected to the power meters through the Net485 bridge, and it is on this computer that the readings are measured. The readings are sampled, by default, every 0.5 seconds. The application used to obtain the values allows for a padding value, which is the amount of time for which measurements are taken before running the test, and after running the test, which allows us to study variations in the power draw. The power monitoring application is executed on the profiling machine. One of the arguments for this command is the name of the application to be executed on the test bed, and this happens via SSH over Ethernet.

The power monitoring meters are plugged into the same power strip that the test bed is connected to. It must be observed that this setup merely measures the wall socket power drawn by the motherboard, and doesn't measure the power draw of individual components such as the hard disk. This implies our power monitoring measurements should primarily be done around compute-intensive applications.

In order to generate the Green 500 List, the authors keep the software configuration similar across all systems, and measure the power drawn with a setup similar to the above setup across different systems. However, in this case, we take a single system and measure the power drawn with a variety of different software configurations to provide a different perspective into power consumption.

CHAPTER 3. - SOFTWARE ENVIRONMENT DESCRIPTION

3.1 OPERATING SYSTEM

Both the test bed and the profiling computer run the Fedora 12 operating system with the 2.6.31.5-127.fc12.x86_64 SMP Linux kernel. This is a SMP enabled kernel.

The operating system allows for dynamic modification of the CPU frequency. This is accomplished by the following command

cpufreq-selector -c $i -f $num

$i refers to the CPU number and can take on values between 0 and 7, for the 8 available cores. *$num* refers to the frequency speed. The file */sys/devices/system/cpu/cpuXX/cpufreq/scaling_available_frequencies* lists the frequencies that are supported for switching by the operating system, and for the test bed values, these values were found to be 2700000, 2000000, 1500000 and 800000.

The file */sys/devices/system/cpu/cpuXX/cpufreq/scaling_available_governors* lists the different modes that the CPU frequency switching can take on. Setting it *performance* will allow the CPU to run at maximum speed and setting it to *ondemand* allows the operating system to dynamically switch between the different available frequencies depending on the workload. This is not an option we will be utilizing in this research. The *userspace* governor setting allows the CPU to be at a particular frequency

3.2 BASIC LINEAR ALGEBRA SUBROUTINES – ATLAS

ATLAS [12] stands for Automatically Tuned Linear Algebra Software. It is an ongoing research effort that aims at providing a portable and optimal version of standard linear algebra routines

with C and Fortran interfaces. It provides a complete BLAS API and a small subset of LAPACK API. The version of ATLAS used in this research project is 3.8.4. It provides C and Fortran77 interfaces to routines such as GESV, GETRF, GETRS, GETRI, TRTRI etc. ATLAS provides a custom C interface to LAPACK, as LAPACK doesn't have an official C interface. Additionally, static libraries are provided by default. The interface provided by ATLAS is used by the HPL and HPCC benchmarks.

3.3 AMD CORE MATH LIBRARY – ACML

ACML is a set of math routines that are thoroughly optimized and threaded for HPC applications by AMD for their processors. It provides a full implementation of Levels 1,2 and 3 of Basic Linear Algebra Subroutines and LAPACK routines. The interface provided by ACML is used to compare against the similar interface provided by ATLAS for SGEMM calculations

3.4 COMPILER TOOLCHAIN – GCC

The compiler used for the CPU –based code is GCC version 4.4.3, built with posix threads support. This compiler also has support for OpenMP, which is described in a later section. The compiler allows for various optimizations at different optimization levels, some of which are described below. Measurements are taken at different optimization levels in order to study the effect of compiler optimization on power consumption. GCC supports three levels of optimization [13], with each higher level performing all optimizations done at the previous level and an additional set. At the first level, the compiler attempts to reduce code size and execution time without performing optimizations that require additional time. At the second level, GCC attempts all supported optimizations that do not involve space – time trade off. This option

increases compilation time. The third level performs loop unrolling and function inlining along with optimizations that would have been performed in the previous levels.

- Default Inline – Do not make member functions inline by default if they are defined inside the class scope (C++). Turned on in level 1.
- Loop optimize – Performs loop optimizations, such as moving constant expressions outside the loop, simplifying exit test conditions and attempts at loop unrolling
- If-conversion – Attempt to transform conditional jumps into branch less equivalents.
- Defer-pop – Always pop the arguments to each function call as soon as that function returns.
- Inline functions – The compiler heuristically decides which functions are simple enough to be worth integrating directly into their callers.

Other optimizations can be found in the GCC Manual.

3.5 PARALLEL PROGRAMMING API – OPENMP

OpenMP is an API for shared memory multiprocessing programming using C, C++ and Fortrain. It consists of compiler directives, library routines and environment variables [14] that influence and direct run time behavior. It uses a portable model that gives programmers a simple way to utilize all available cores on a shared memory symmetric multiprocessor based system.

OpenMP is managed by a technology consortium consisting of major computer hardware and software vendors such as AMD, IBM, Intel, Cray, HP, NVIDIA and others [15].

OpenMP implements multithreading , wherein the thread that is executing the program (also referred to as master thread) forks a specified number of threads (also referred to as slave threads) and the run time environment allocates the different threads to the different processing units available. This allows the programmer to control the number of cores being used in the code. The operating system allocates threads to the processing units depending on factors such as usage and machine load. The number of allocable threads is assigned by setting the values of certain environment variables before the execution of the code. OpenMP provides support for both task parallelism as well as data parallelism.

3.6 PARALLEL PROGRAMMING API – MPI

MPI stands for Message Passing Interface [16], and is the industry standard for parallel programming on super computers and computer clusters. While the previously described OpenMP API is primarily used as shared memory multithreaded programming library, MPI is primarily targeted at distributed memory programming, where the individual "threads" (referred to as processes in MPI) are typically run on different machines connected over a Local Area Network. Thus each process is likely to have access to only its own local memory.

MPI library functions include point to point rendezvous type send/receive operations, combining partial results of computations such as gather and reduce, exchanging data between processes and synchronizing processes through barriers.

There exist different implementations of MPI. For the purpose of this project, MPICH 1.2.7p1 was chosen. Additionally, even though MPI is traditionally meant for distributed memory machines, MPICH also provides for using shared memory as a channel for communication. This was accomplished using the ch_shmem option while compiling MPICH.

3.7 PARALLEL PROGRAMMING API – POSIX THREADS

In shared memory multiprocessors architectures such as the one used in the test bed, threads are often used to implement parallelism. Hardware vendors often implement their own proprietary implementations towards this end, however, the IEEE POSIX 1003.1 c standard specifies a standard C language interface for programming threads.

- The pthreads API consists of subroutines falling in to 4 major categories
- Thread management – Routines that create, detach, join and perform other operations on threads
- Mutexes – Routines that deal with synchronizations using mutual exclusions
- Condition Variables – Routines that deal with communications between threads that share mutexes
- Synchronization – Routines that deal with locks and barriers.

3.8 GPU PROGRAMMING API - CUDA

CUDA, which stands for Compute Unified Device Architecture is a proprietary architecture developed by NVIDIA for use with their Graphics Processing Units (GPUs). 'C for CUDA' provides programmers a C programming interface with NVIDIA specific extensions and restrictions) that allows for coding algorithms to be executed on the NVIDIA GPUs.

GPUs were traditionally used for applications such as gaming and graphics rendering. However, the GPU can also be viewed as a *compute capable* device [17] consisting of an extremely large number of processing units. This technique of using the GPU for performing computations in

applications that were once strictly the domain of CPU is known as GPGPU (General Purpose computing on Graphics Processing Units). The GPU acts as co-processor to the CPU, and allows data parallel, compute intensive portions of applications to be offloaded onto itself.

Data-parallel refers to the parallel programming paradigm where the same set of operations are carried out independently on different data. As long as the computations are independent, the task can be compiled into an instruction set for the GPU and can be downloaded onto the GPU.

The GPU itself is implemented as a set of multiprocessors, each having a Single Instruction Multiple Data architecture (SIMD). At any given clock cycle, the processors are all performing the same operation, but they are operating on different data.

For instance, in the GeForce GTX 260 used in the test bed, the number of multi processors is listed as 27 and the number of cores is listed as 216, thereby implying that each multiprocessor has 8 cores.

nvcc is NVIDIA's compiler driver that allows compilation of CUDA code. The workflow of this compiler initially involves separating the device (GPU) code from the host (CPU) code. The CPU code is eventually compiled by the host compiler. The GPU code is compiled into a binary object. This binary object is then loaded and executed on the GPU using the CUDA driver API.

Further details regarding CUDA and *nvcc*'s workflow can be found in the NVIDIA CUDA Programmer's Guide [17].

CHAPTER 4 – BENCHMARKS AND WORKLOAD DESCRIPTION

In this section, we describe the various applications being run on the CPU and the GPU. While some applications are industry standard benchmarks, some applications are smaller applications coded in order to expose and stress particular factors individually to monitor whether or not they have an impact on power consumption.

4.1 EFFECT OF CACHE ACCESS

We explore the impact of cache lines on power consumption. We run loops of the form

*for (int i=0; i<array.Length(); i+= K) arr[i] *= 3;*

In the above loop, we expect to see different run times for different values of *K*. CPU's fetch memory in chunks of typically 64 bytes called cache lines. When a particular memory location is accessed, the entire cache line is fetched from the main memory into the cache, and subsequent accesses to elements in the cache are much faster.

We then monitor the power consumed in different iterations.

4.2 BLOCK MATRIX MULTIPLICATION

The basic matrix multiplication algorithm is the one that involves three nested loops. However, with block multiplication, we take groups of elements (the number of elements in the group is equal to the block size) and then then perform multiplication / addition operation on those elements as required. The idea behind this implementation is that the group of elements are on the cache and this reduces cache misses.

This test is performed as an extension of the previous Cache Access test.

4.3 GCC OPTIMIZATION

The matrix multiplication code is compiled with varying GCC optimization levels and power consumption is monitored.

4.4 SYMMETRIC MULTIPROCESSOR PARALLEL PROGRAMMING

Matrix multiplication is carried out using OpenMP and pthreads. The least common multiple of 1,2,3,4,5,6,7 and 8 is taken as the smallest size of the matrix (840), which allows us to use any number of processors and have the tasks divided evenly between the processor cores.

4.5 HIGH PERFORMANCE COMPUTING CHALLENGE BENCHMARKS (HPCC)

The HPCC benchmarks [18] consists of 7 tests.

1 HPL – The Linpack benchmark, which measures floating point rate of execution while solving a linear system of equations.

2 DGEMM – Measures rate of execution of Double recision General Matrix Multiplication

3 STREAM - Measures sustainable memory bandwidth and corresponding computation rates for a vector kernel

4 PTRANS – Parallel Matrix Transpose, bandwidth that stresses the communications capacity of a network

5 RandomAccess – Measures the rate of integer random updates of memory (GUPS – Giga Updates Per Second)

6 FFT – Floating point rate of execution of double precision complex Discrete Fourier Transform

7 Communication bandwidth and latency – Tests and measures latency and bandwidth using various simultaneous communication patterns.

The HPCC benchmarks were designed with the goal of examining the performance of supercomputers with memory access patterns different from that of the standard HPL. The Top500 List is solely based on the performance of the supercomputer on the HPL benchmark, and these tests were designed to compare performances using additional metrics.

For this research project, special focus shall be given to HPL and RandomAccess, as we believe that the execution rate as well as memory updates can influence power consumption. DGEMM and FFT could have been used as well; however, HPL is easily ported to the GPU using CUDA and hence is a convenient way to measure power consumption across different scenarios involving the CPU and the GPU.

4.5.a HIGH PERFORMANCE LINPACK

LINPACK is a software library that provides a FORTRAN interface for performing numerical linear algebra. It utilizes the BLAS (Basic Linear Algebra Subroutines) libraries for performing basic vector and matrix operations. It solves linear systems whose matrices are general, banded, symmetric indefinite

Originally, LINPACK was used as a measure of a system's floating point computing power. The benchmark solves a dense system of linear equations using the Gaussian Elimination algorithm with partial pivoting. This result is then reported in millions of floating point operations per second.

With the advancements in super computing technology and the advent of beowulf clusters built out of Commercial Off The Shelf computers, Linpack was no longer a good super computer

benchmark as it focused primarily on floating point arithmetic units and cache memory and not on the shared memory or the node interconnect. Linpack's memory access patterns disregard the multi layered memory hierarchies of modern machines and hence more time than necessary was being spent on moving data as opposed to actual computation. LAPACK addresses this problem by reorganizing the algorithms to use block matrix operations in the inner loops. However, in order to obtain the fastest possible performance, LAPACK would require that optimized block matrix operations be already available on the host machine. This is where ATLAS is used.

Additionally, Linpack as a benchmark gave way to High Performance Linpack [19], which too is a software package that solves a random dense linear system in double precision arithmetic on distributed memory computers. This algorithm uses a two dimensional block cyclic data distribution, followed by a variant of the LU factorization with row partial pivoting and recursive panel factorization and finally a backward substitution. Thus, HPL not only times the calculation of a solution, but it also provides a way of quantifying the accuracy of the obtained solution.

4.5.b RANDOM ACCESS

The current trend in processor architecture is to focus on having longer cache lines and better stride performance. However, random memory access is also a metric that needs to be given importance.

RandomAccess (GUPS) is a benchmark that works by identifying the number of memory locations that can be randomly updated in one second, divided by 1 Billion. An update is defined as a read-modify-write operation on a table of 64 bit words.

The authors provide multiple variations of the GUPS benchmark, including Sequential and Multi threaded local versions and MPI versions for distributed computers.

4.6 CUDA COMPUTATIONS

In order to test the power monitoring characteristics of the GPU, we run a variety of programs on the GPU. Some basic tests are described below –

1. CuBLAS – BLAS for CUDA, performing basic linear algebra sub routines
2. Basic vector addition
3. Merge sort .
4. Computation with single precision and double precision values

CHAPTER 5 -EXPERIMENTAL METHODOLOGY

A typical output when using the power monitoring software for an application is as follows–

AC[02]: starting monitor thread

AC[02]: 0.037 -> active power: 177.60 W

← The above is repeated for as many samples as requested →

*******: 2.500 ** Starting: ssh c04 ../RemoteScript*

AC[02]: 2.516 -> active power: 177.60 W

*******: 2.743 ** Application exited*

AC[02]: 3.016 -> active power: 183.60 W

← The active power continues to get monitored for as many samples as requested →

DC Readings - DC[02]: starting monitor thread

DC[02]: 0.037 -> 3.3V rail: 7.61 W

DC[02]: 0.037 -> 5V rail: 9.38 W

DC[02]: 0.037 -> 12V rail1: 16.02 W

DC[02]: 0.037 -> 12V rail2: 70.19 W

← The above is repeated for as many samples as requested →

*******: 3.500 ** Starting: ssh c04 ~/Codes/xGPUvectorAdd*

DC[02]: 3.515 -> 3.3V rail: 7.60 W

DC[02]: 3.515 -> 5V rail: 9.38 W

DC[02]: 3.515 -> 12V rail1: 16.02 W

DC[02]: 3.515 -> 12V rail2: 69.46 W

← The above is once again shown at the sampling rate till the application exits →

*******: 4.892 ** Application exited*

DC[02]: 5.014 -> 3.3V rail: 7.58 W

DC[02]: 5.014 -> 5V rail: 9.38 W

DC[02]: 5.014 -> 12V rail1: 24.67 W

DC[02]: 5.014 -> 12V rail2: 97.93 W

← The above is again shown for as many samples as requested →

A couple of observations must be made at this point. In the following command

powermon –n 2 –a –p 7 – ssh "~/RemoteScript"

powermon represents the name of the executable run on the profiling computer, which is a custom written program that accesses the registers on the power monitors to obtain the power measurements.

The *–n* switch is used to indicate the number associated with the test bed. The test bed is node 2 of 4 in the configuration used in the lab, and hence the value.

The next switch indicates whether AC active power values are measured, or the power draw on individual rails are measured. This switch is optional. Not including either of *–a* or *–d* will

result in a combined output of both sets of values.

The *–p* switch indicates the amount of padding around application start and termination, in number of samples.

Additionally, the sampling period can be changed with a *–s argument* switch. The default value as can be observed from the time stamps is 500 milliseconds.

Even though it is not shown in the above output, the-*o* switch allows the output to be redirected to a file. The output from the execution of the remote program is redirected to *stderr*.

ssh is the protocol used to communicate with the test bed.

The command to be executed on the test bed is included within quotes. Since the running of CUDA programs, HPCC benchmarks and OpenMP programs require the setting of environment variables, the commands are included in a script file on the test bed, which is then sourced through the SSH command run on the profiling computer.

Since the entire execution happens on a terminal, it allows us to create scripts that execute the program multiple times with varying data, in order to obtain substantial data. Text processing is then done on the obtained data in order to extract the relevant information, which is then plotted and studied.

The following tests were conducted, in the order of the workload descriptions provided in the previous chapter.

1. Simple loops with varying strides to measure cache sizes (Refer Chapter 4.1)

2. Block matrix multiplications (Refer Chapter 4.2)

    a. Vary block size

    b. Vary problem size

3. Block matrix multiplication with different GCC optimizations (Refer Chapter 4.3)

4. Symmetric Multiprocessor Parallel Programming with block matrix multiplication (Refer Chapter 4.4)

    a. Vary number of CPUs used

    b. Vary processor speed of CPUs

5. HPCC benchmarks

    a. HPL benchmark (Refer Chapter 4.5.a)

        i. Vary number of processes

        ii. Vary problem sizes

    b. RandomAccess benchmark (Refer Chapter 4.5.b)

6. Computations on the GPU, with varying problem sizes (Refer Chapter 4.6)

CHAPTER 6 – EXPERIMENTAL RESULTS

The first measurement taken involves the base line values. This shall be our reference point of origin for all further readings.

This reading is measured after a clean restart. Since the operating system has been configured to start in runlevel 3, the graphics card has not yet started drawing power. Once we manually change the runlevel to 5 with the following command

init 5

Linux starts its X Windows and enters its Graphical User Interface mode. At this point, the GPU begins to draw power and all further measurements (even after switching off the GUI) will be influenced by this. This will require calculating a new power base line as well.

After having obtained base line values, we proceed to conduct experiments that only use the CPU, reserving the GPU use for later.

Tables 1 and 2 represent the active power drawn and the DC power drawn for different base line values. Figures 1 and 2 are corresponding graphical representations.

6.1 EFFECT OF CACHE ACCESS

In this program, we step over an array incrementing every 16^{th} integer. Upon reaching the last value, we loop back to the beginning. We experiment with different array jumps and observe the power draw. It can be observed that there's a significant climb when the array sizes are 512 and 2048 kb, which are the sizes of the L1 and L2 caches.

Table 3 represents the active power drawn for different stride sizes, and figure 3 is its corresponding graphical representation.

6.2 BLOCK MATRIX MULTIPLICATION

In this section, we take the regular matrix multiplication code, and modify it take a group of elements (Block). The motivation behind this technique is to improve cache access, as the "group" of elements that we will be working on is expected to be in the cache, as opposed to the original triple nested loop where we have far more frequent cache misses.

Provided below are certain observations from varying the block sizes and the problem sizesThe CPU frequency was kept constant throughout, and the code used for the above was sequential block matrix multiplication.

Table 4 shows different values of active power consumed for different combinations of block size and data size, and the data is represented pictorially in figure 4.

6.3 GCC OPTIMIZATION

Tables 5, 6, 7 and 8 (and corresponding figures 5,6,7 and 8) represent the active power drawn for different GCC optimization levels for data sizes of 10000, 20000, 40000 and 80000 respectively.

6.4 SYMMETRIC MULTIPROCESSOR PARALLEL PROGRAMMING

In this section, we take the block matrix multiplication algorithm we have been using and observe that matrix multiplication is effectively embarrassingly parallel. As a result, it can be trivially parallelized.

As we are on a shared memory architecture machine, both of the matrices are available entirely to every thread / process. Each thread updates its individual results in the product matrix.

We vary the number of CPUs used between 1 and 8, which is the total number of processing cores in the test bed.

The processor speed of the CPUs is also varied between the allowed values of 800 MHz, 1500 MHz, 2000 MHz and 2700 MHz. The problem size is kept constant at 84000 and a block size of 64 is chosen.

Tables 9 and 10 represent the data measured when using OpenMP and Pthreads respectively.

The biggest observation that can be drawn is that in the case of OpenMP, when the 5^{th} thread is made available, there's a significant jump in the power draw. However, in the case of pthreads, the power draw is more or less uniformly increasing.

6.5.a HIGH PERFORMANCE LINPACK

The input for running the HPL benchmark is in the form of a file named HPL.dat. Some relevant factors are explained below –

Line 5 – This line specifies the number of problem sizes to be executed. In our case, we always keep it 1, since we are measuring power consumed for each variation.

Line 6 – The problem size. We vary this value (N)

Line 7 – The value of the block size for each run

Line 10 – Number of process grids. Again, we will be running one at a time.

Line 11 and 12 – Specifies the number of process rows and columns of the grid, which in turn corresponds to available cores. Therefore, we monitor power consumed for different values of P and Q such that P*Q <= 8

6.5.b GUPS – GIGA UPDATES PER SECOND

In this research, GUPS is run in a sequential fashion. The following results were obtained.

Begin of SingleRandomAccess section.

Node(s) with error 0

Node selected 1

Single GUP/s 0.190227

Current time (1310884140) is Sun Jul 17 01:29:00 20112

End of SingleRandomAccess section.

Active Power consumed = 191 W, without enabling GPU

6.6 CUDA COMPUTATIONS

Before we begin measuring power consumptions owing to GPU usage, we need to enable the GPU. This is done by switching the Linux operating system to run level 5. The following command accomplishes this -

init 5

This brings up the GUI mode. For the purposes of the experiment, we switch back to runlevel 3, under the assumption that we are eliminating any residual power draw for printing graphics on the screen. This way we are using the GPU strictly as a computational device.

However, once switched on, the GPU will draw a small power of its own. Hence we need to update our base line values. The new base line values are provided in the following tables –

For every one of the test kernels explained below, tests were run both on the host alone, as well as using the host and the GPU device. In most cases, visible speed ups could be noticed when

running the kernel on the GPU. However, as the aim of this project is not to measure speed up, the time taken to execute will not be considered. Instead, the power draw alone will be measured.

Additionally, it must be observed from the above tables that the mere process of turning on the GPU draws in an overall residual power. For some of the test kernels before, we try and find the median data set, such that for any data set smaller than the median, it would be optimal to run the code on the CPU itself without turning on the GPU, and similarly for any data set larger than the median it would be optimal to run the code on the GPU inspite of the residual power draw. Tables 11 and 12 represent this data.

6.6.1 SIMPLE CUBLAS

In this program, SGEMM is used. For comparison, the code used to run the program entirely on the CPU involves OpenMP. Table 13 represents this data.

6.6.2 BASIC VECTOR ADDITION

Table 14 represents this data.

6.6.3 MERGE SORT

In this case, the matrix multiplication done both on the host and on the device is the non blocked multiplication. However, the CPU version does use OpenMP and utilizes all 8 cores available. This data is presented in table 15.

6.6.4 SINGLE VERSUS DOUBLE PRECISION COMPUTATIONS

Basic vector operations are performed, such as addition and scaling. This data is presented in table 16.

Table 1: Base AC power versus CPU frequency

CPU freq (MHz)	Base AC power (Watt)
800	178.7
1500	183
2000	185.4
2700	192.6

Table 2: Base DC power versus CPU frequency

CPU freq (MHz)	Base 12V Rail2 (Watt)
800	52.3
1500	58.2
2000	64.3
2700	68.92

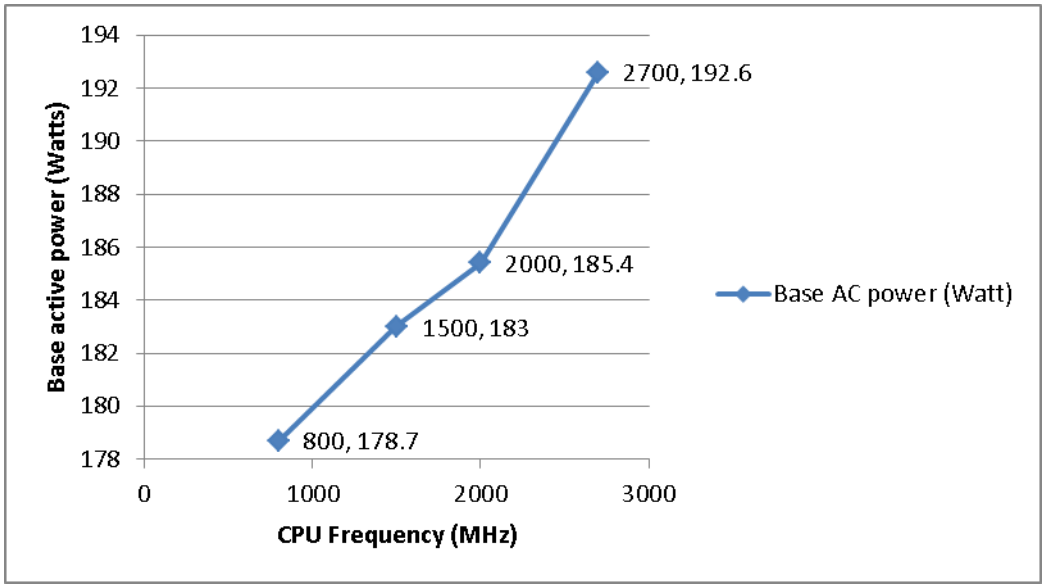

Figure 1:Base AC power versus CPU frequency

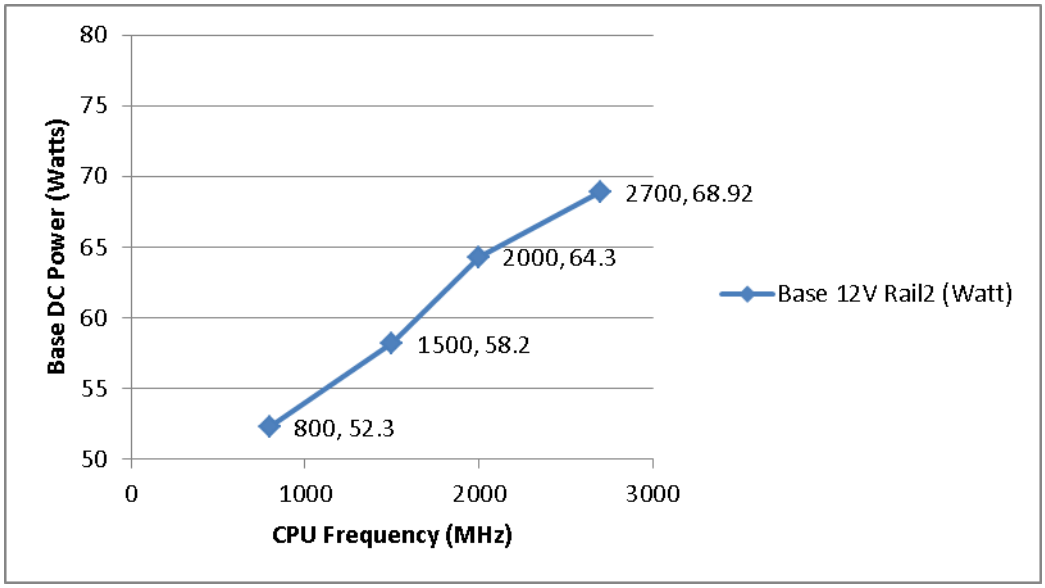

Figure 2:Base DC power versus CPU frequency

Table 3: Base AC power versus stride size

Stride size	Active Power (Watts)
1	178.7
2	179.6
4	179.6
8	180
16	180.9
32	181.5
64	181.9
128	182
256	182.6
512	185
1024	185.6
2048	190
4096	190.9

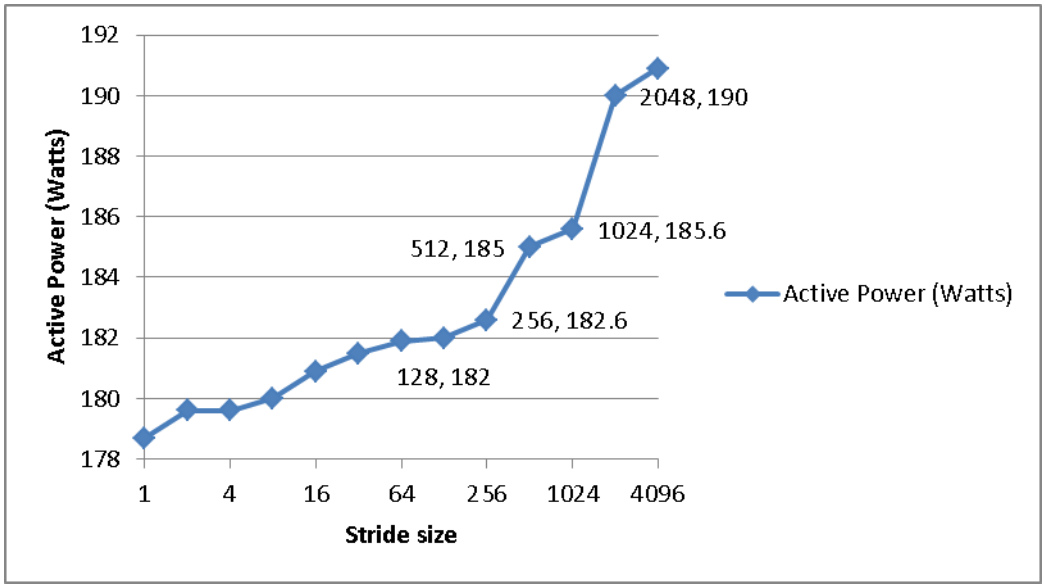

Figure 3:Base AC power versus stride size

Table 4: Base AC power for different block and data sizes

Block Size	10000	20000	40000	80000
1	192.2	200	212	220
2	192.8	200.3	212.4	220.1
4	192.8	200.4	212.8	220.3
8	193	201	213.3	221
16	193.4	201.3	213.9	221.5
32	193.5	201.4	214.4	221.6
64	193.6	201.9	214.5	221.8
128	193.8	202.4	215	223
256	193.9	202.5	215.5	223.5
512	196	203	216.8	225
1024	196	203.4	217	226
2048	198	204	218.9	228
4096	198.6	205	219	230

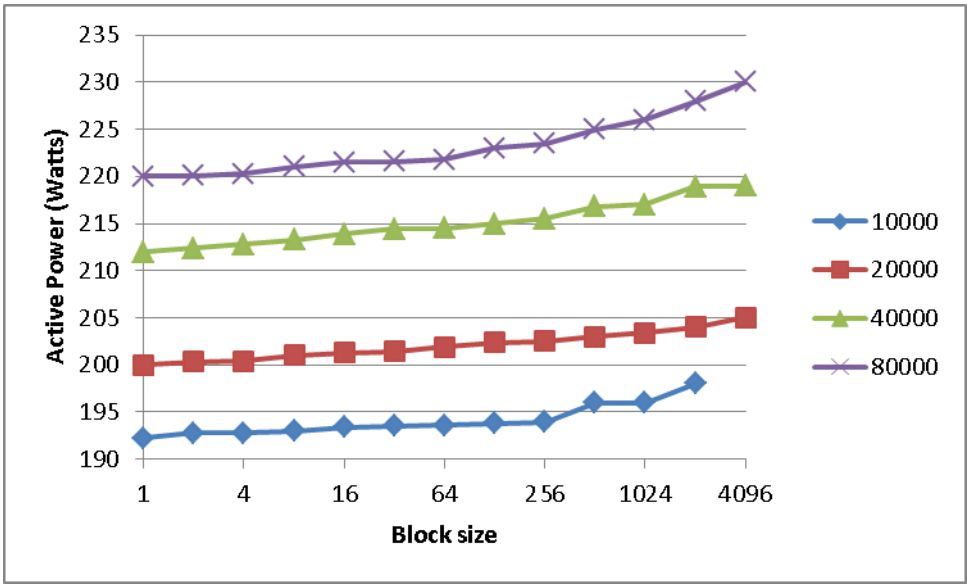

Figure 4:Base AC power for different block and data sizes

Table 5: Base AC power for data size 10000

Block Size	No optimization	"-O1"	"-O2"	"-O3"
1	192.2	192	191.5	189.3
2	192.8	191	191.3	189.5
4	192.8	192.6	192.2	189.5
8	193	192.7	192	190
16	193.4	192.5	192.2	191.4
32	193.5	193.7	192.1	191.8
64	193.6	193.6	192.8	192.3
128	193.8	194	193.2	192.8
256	193.9	194.2	193.5	193
512	196	195.6	194.2	193.6
1024	196	195.8	195.3	194.2
2048	198	197.4	195.2	194.6
4096	198.6	198	196.2	195.2

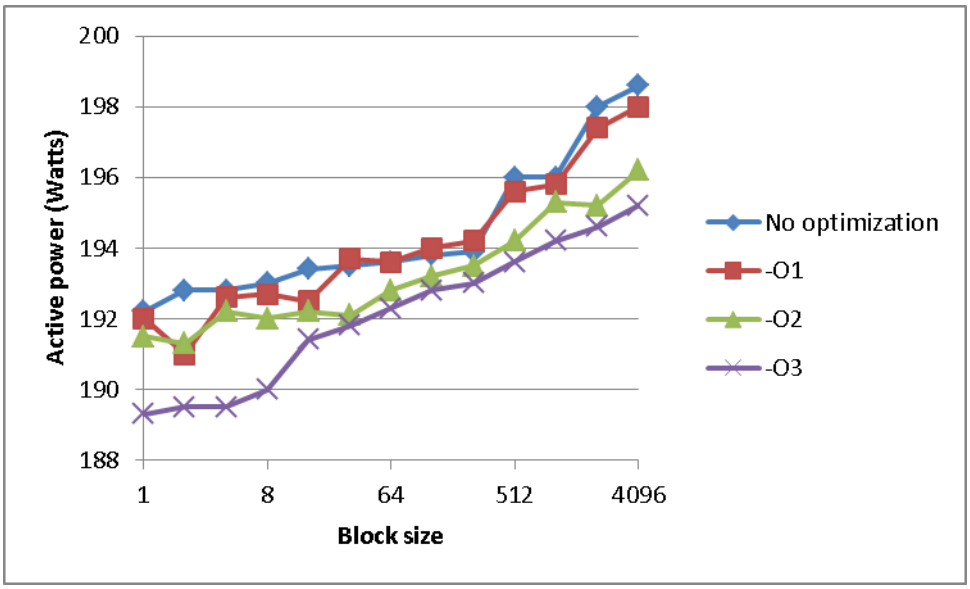

Figure 5:Base AC power for data size 10000

Table 6: Base AC power for data size 20000

Block Size	No optimization	"-O1"	"-O2"	"-O3"
1	200	199.2	198.2	194.5
2	200.3	199.6	198.2	194.9
4	200.4	199.6	198.8	195.2
8	201	199.9	199.3	195.6
16	201.3	200.4	199.7	196.4
32	201.4	200.6	200	196.8
64	201.9	201	200.6	197.3
128	202.4	201.4	201.2	197.4
256	202.5	201.8	201.5	197.4
512	203	201.9	201.7	198.3
1024	203.4	202.8	202.8	198.7
2048	204	203.2	202.9	198.9
4096	205	203.5	203	200.1

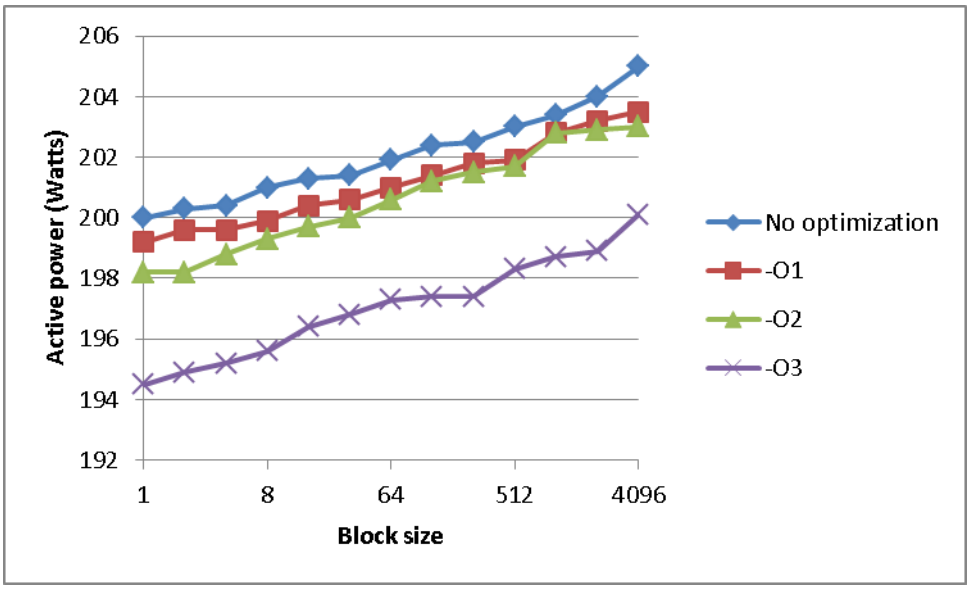

Figure 6:Base AC power for data size 20000

Table 7: Base AC power for data size 40000

Block Size	No optimization	"-O1"	"-O2"	"-O3"
1	212	212.2	212	211
2	212.4	212.4	212.4	211.1
4	212.8	213	212.8	212
8	213.3	213.3	213.3	212.4
16	213.9	213	213.9	212.8
32	214.4	214.4	214.4	213
64	214.5	214.5	214.5	213.8
128	215	214.9	215	214
256	215.5	215	215.5	214.4
512	216.8	216.8	216.8	214.4
1024	217	217	217	215
2048	218.9	218.9	218.9	216
4096	219	219	219	215.6

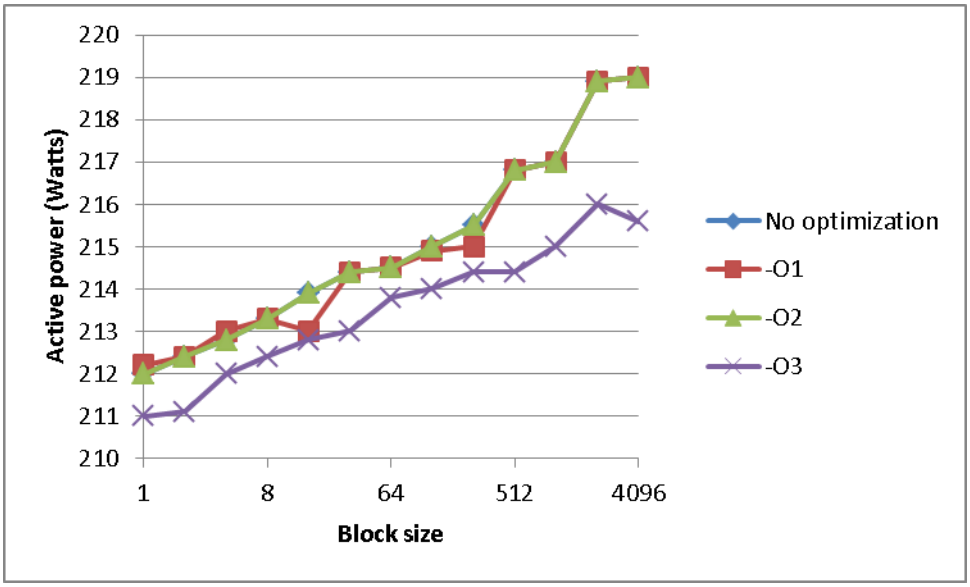

Figure 7:Base AC power for data size 40000

Table 8: Base AC power for data size 80000

Block Size	No optimization	"-O1"	"-O2"	"-O3"
1	220	220.1	218.2	214.2
2	220.1	219.2	219	214.9
4	220.3	219.7	219.2	215.1
8	221	220.1	220	215.9
16	221.5	220.7	220.5	215.8
32	221.6	221	220.9	216.1
64	221.8	221.3	221.3	216.8
128	223	221.9	222.1	217.3
256	223.5	222.5	222.8	218
512	225	223.1	223	219.1
1024	226	223.9	223.8	219.5
2048	228	225.3	224.9	220.1
4096	230	226.3	226	220.9

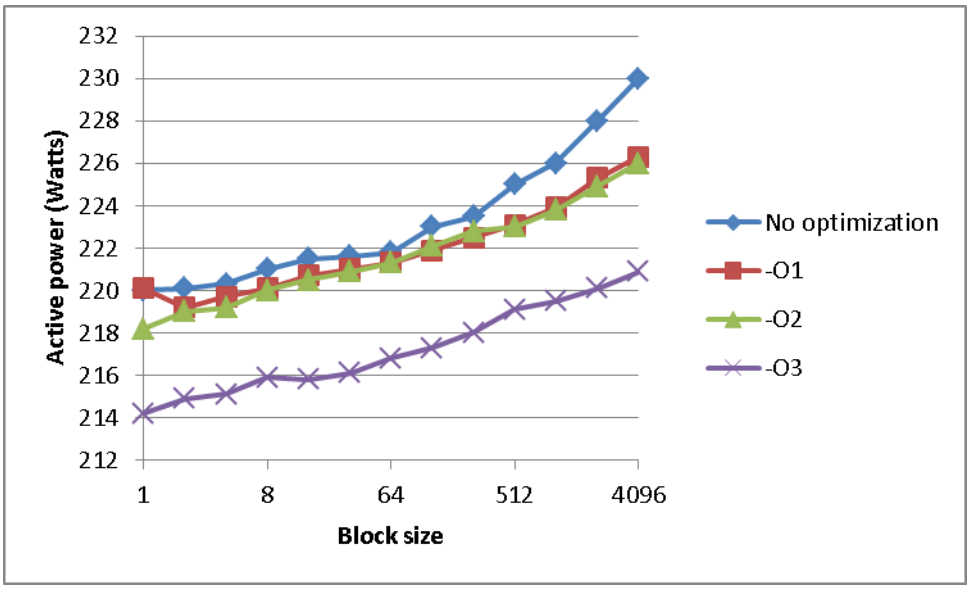

Figure 8:Base AC power for data size 80000

Table 9: Base AC power using OPENMP

number of CPUs	800 Mhz	1500 Mhz	2000 Mhz	2700 Mhz
1	178.7	185	189.5	196.6
2	179	185.5	190	197
3	179.5	185.7	190	197.6
4	179.7	186	190.6	198
5	181	190	195	204
6	181.3	191.3	195.1	205
7	182	192	195.2	205.6
8	182.4	192.4	195.5	206

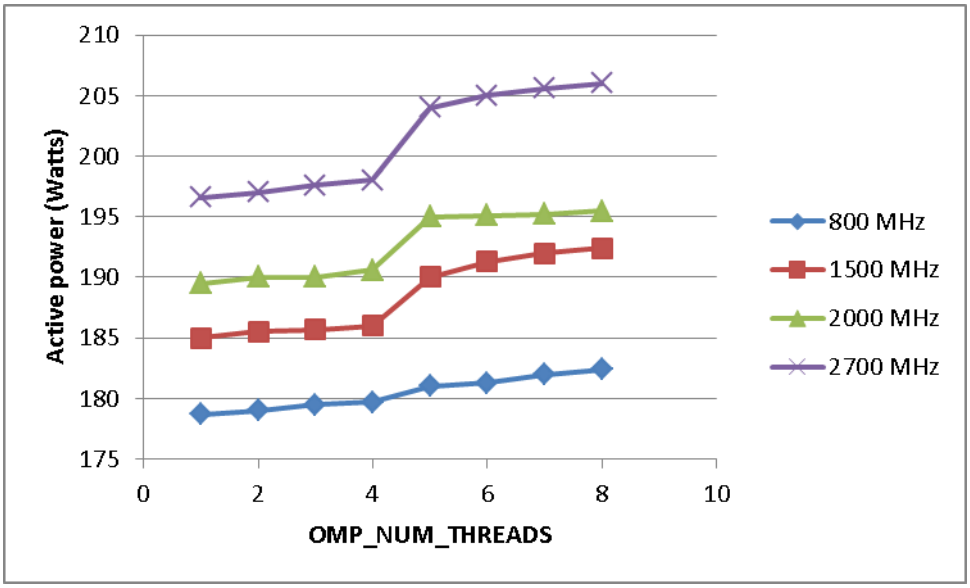

Figure 9:Base AC power using OPENMP

Table 10: Base AC power using PTHREADS

number of CPUs	800 Mhz	1500 Mhz	2000 Mhz	2700 Mhz
1	178.7	185	189.5	196.6
2	178.7	185.5	189.5	197
3	178.9	185.7	190	197.6
4	179	186	190.5	198
5	180	186.5	192	200
6	181	187	192.5	200.2
7	181.2	187.6	192.7	200.5
8	182.4	188	193	201

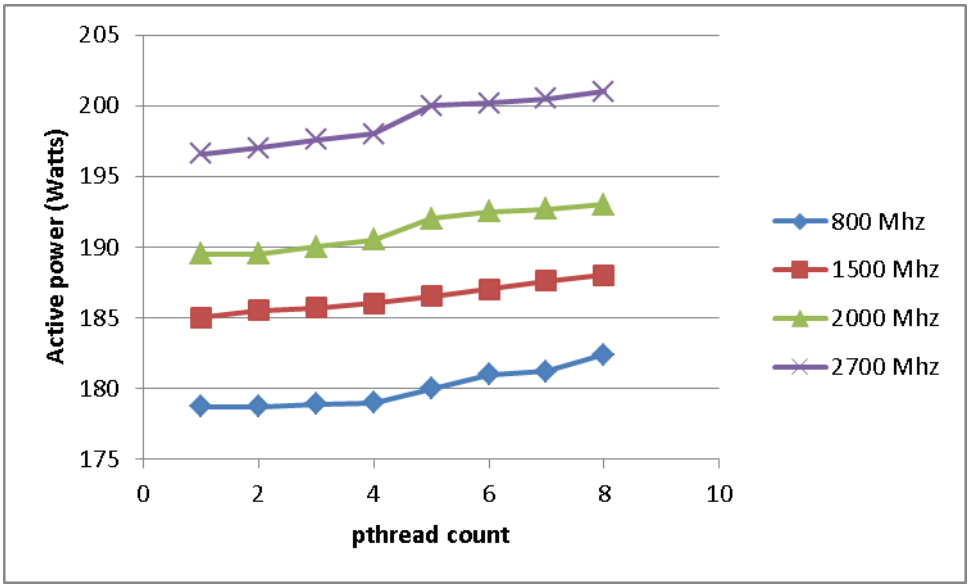

Figure 10:Base AC power using PTHREADS

Table 12: Base AC power using GPU

CPU freq (Mhz)	Base AC power (Watt)
800	198
1500	200
2000	208
2700	210

Table 11: Base DC power using GPU

CPU freq (MHz)	Base 12V Rail2 (Watt)
800	70
1500	74.3
2000	78
2700	80

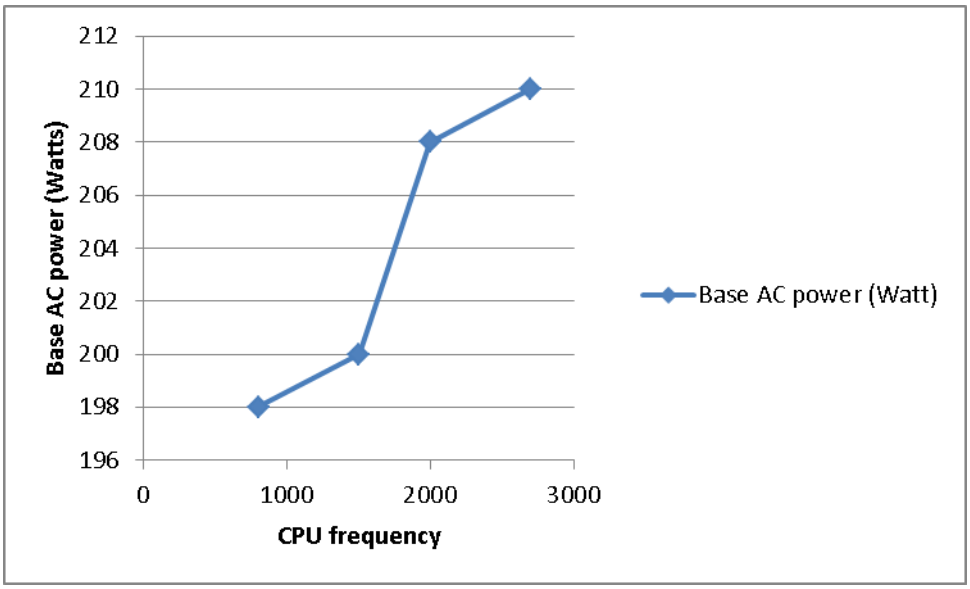

Figure 11:Base AC power using GPU

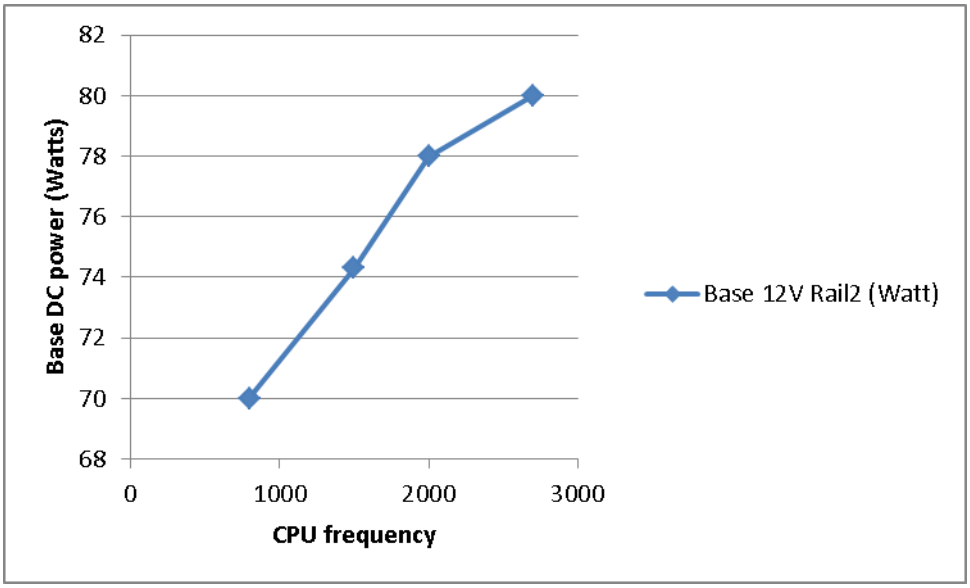

Figure 12:Base DC power using GPU

Table 13: Base AC power for different SGEMM runs

Matrix Size	ATLAS	cuBLAS	ACML
5000	210	220	209.8
10000	213.5	220.4	211.7
15000	215	220.6	214.9
20000	218	221	216.7
25000	218.9	224	217.8
30000	220	224.8	218.9
35000	224.3	226	222.1
40000	226.8	228	224.2
45000	228.4	228.9	225.6
50000	230.6	229	228.4
55000	233	230.4	231.1

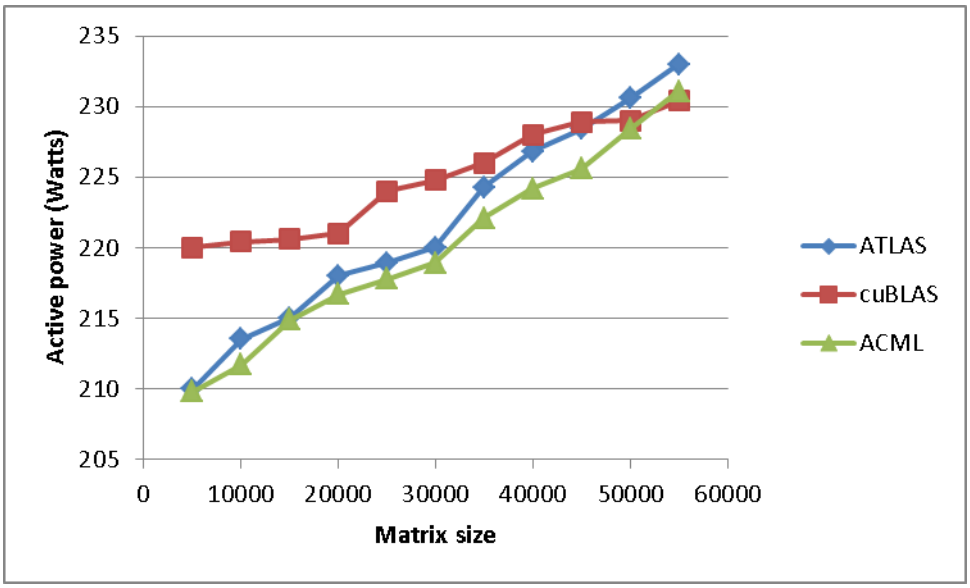

Figure 13:Base AC power for different SGEMM runs

Table 14: Base AC power for vector addition

Vector Size	Host only Power	Host+GPU Power
5000	211	220
10000	212.4	220.4
15000	214.6	220.6
20000	216	221
25000	218.7	221.5
30000	219	222.3
35000	221.4	223.4
40000	224.7	223.9
45000	225.1	224.5
50000	225.9	225
55000	226.4	225.8

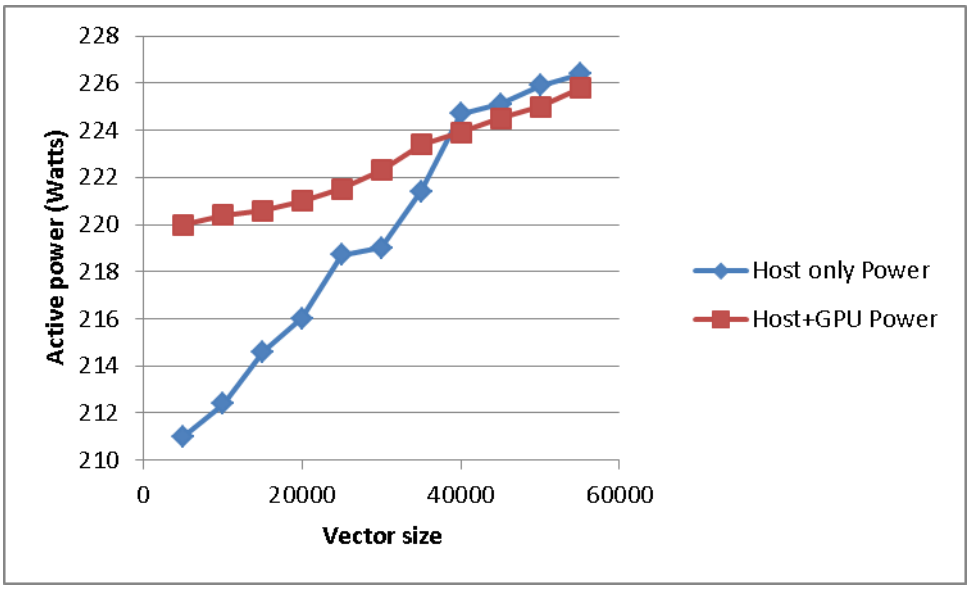

Figure 14:Base AC power for vector addition

Table 15: Base AC power for Merge sort

Vector Size	Host only Power	Host+GPU Power
2500	211	220.1
5000	214.2	220.5
10000	216.3	220.7
15000	217.4	221.5
20000	219.5	222.3
25000	221.2	223.7
30000	223.5	224.8
35000	224.3	225.9
40000	225.6	226.6
45000	226.8	228
50000	230.4	228.9
55000	231	229.1

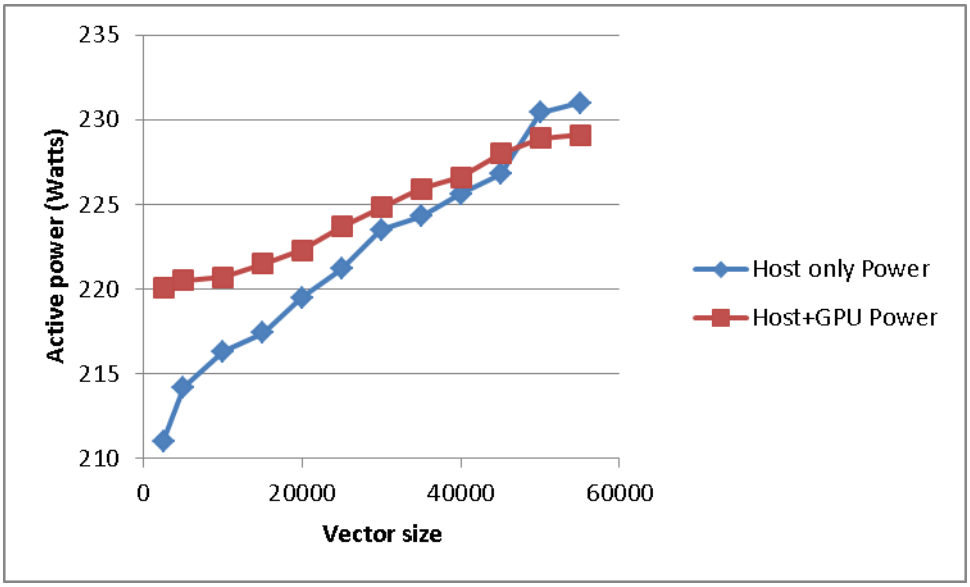

Figure 15:Base AC power for Merge sort

Table 16: Base AC power for single and double precision

Vector Size	Single Precision, CPU	Single Precision, GPU	Double Precision, CPU	Double Precision, GPU
2500	211	220.1	220	230.1
5000	214.3	220.3	222.1	231.6
10000	215.8	219.6	223.5	232.8
15000	217.3	220.7	224.2	233.5
20000	218.1	222.1	224.9	234.1
25000	221.1	223.7	225.8	234.8
30000	223	224.7	227.1	235.2
35000	224.1	225.4	228.2	236.1
40000	225.8	226.1	229.9	236.6
45000	226.7	227.1	231.1	237.1
50000	228.1	227.8	232	237.9
55000	229.1	228.5	234	238.1

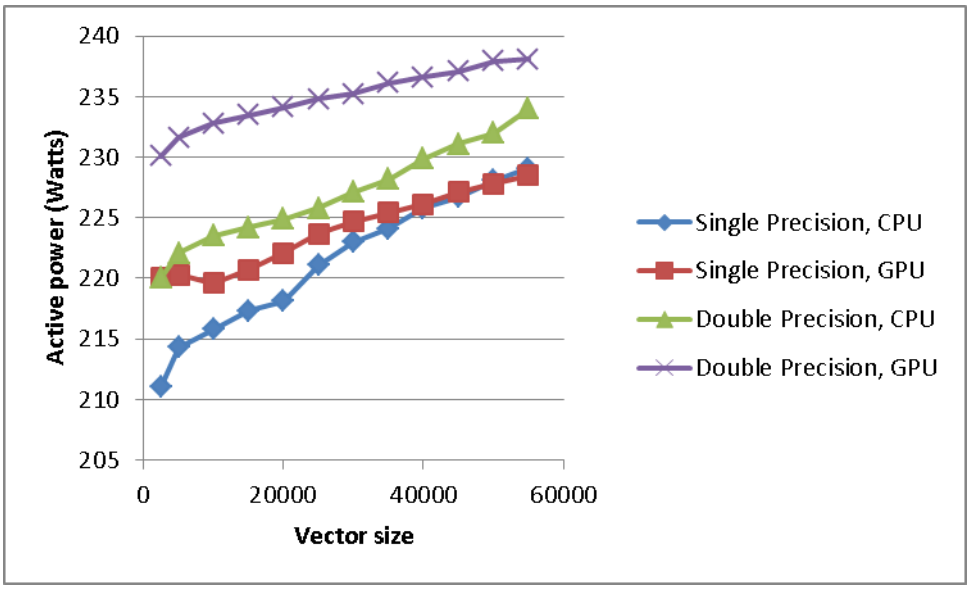

Figure 16:Base AC power for single and double precision

CHAPTER 7. DISCUSSION

Tables 1 and 2 establish the dependence of power consumption on the cpu frequency. Most modern CPUs, especially the ones used in laptop computers provide for a dynamic modulation in the cpu frequencies depending on the load. The Figures establish that the relation between the frequency and base power consumed is more or less linear. Thus, from a power point of view, it would be a wiser idea to let CPUs idle at the lowest possible frequency they can support.

Section 6.1 establishes a relation between cache misses and power consumption. Every time there's a cache miss, there's a small jump in power as the next set of elements are brought into the cache.

Section 6.2 deals with a commonly seen routine in scientific computing, matrix multiplication. Power consumption, as inferred from previously, jumps if the number of cache misses are high. Consequently, irrespective of the problem size, keeping the block size smaller than or equal to the cache line size would yield optimum results.

As explained in section 3.3, GCC performs most supported optimizations that do not involve a space speed trade off at the second level, and performs optimizations such as inline functions, unswitch loops at the third level. Even though the third level of optimizations provide a significant gain in power consumption, the compilation time is drastically increased as well.

Section 6.4 reveals certain interesting results. One of them is that there is not a significant difference in overall power consumption between OpenMP and Pthread APIs for multi threaded programming. This provides programmers with higher flexibility when it comes to a choice of APIs.

Additionally, there's a significant jump in power consumed when the number of cores increase from 4 to 5 in the case of OpenMP, but this characteristic of the graph is absent when using pthreads. Given that the architecture of the machine is dual quad-core, a possible explanation for this is that the run time environment allocated 4 cores of a single processor for OpenMP execution, and when the 5^{th} thread was required, the second processor was brought in for computation thereby resulting in a slight increase in the power consumption.

The HPL and RandomAccess benchmarks from the HPCC suite of benchmarks predict what we have already learned from the above test runs. Different combinations of P, Q and N were tried before concluding at 4.14 GFLOPS as the peak performance of the machine. A sudden jump in the active power consumed confirms this result.

Section 6.6. provides certain interesting conclusions, and probably a reminder of what we are likely to see more of in the future. In almost all of the test examples, for smaller sizes using the CPU alone easily proves more beneficial than using the GPU. This could probably be explained by the fact that merely transmitting data to and from the GPU utilizes time and power, and the CPU outperforms the GPU before this process finishes. However, as all the graphs indicate, the power usage curves of the CPUs are always steeper than those of the GPUs, and there's always a point in the graph beyond which the GPUs out perform the CPUs in terms of power consumption.

Additionally, a special note must be made of the double precision floating point capabilities of the hardware being used in the test bed. The GTX 260, while supporting double precision, is more likely to suffer from a degradation in performance since the double precision arithmetic logic unit is shared between all cores of each multiprocessor.

CHAPTER 8. CONCLUSION AND FUTURE WORK

In this thesis, we have looked at some of the factors that are likely to affect power consumption. As a end user or a developer, there are certain tools in our hands that allow us to write more energey efficient programs, while we continue relying simultaneously on hardware vendors to build energy efficient systems. Heterogenous computing with GPUs is definitely a viable alternative in terms of power consumption, and is already becoming quite popular.

This thesis revolves primarily around scientific computing kernels, all run via the console. We have not studied the power consumption trends in GUI applications, which are more likely to affect the non-technical computer users directly. Additionally, we have not studied the power consumption trends among different operating systems either. With most open source software being cross platform, and vendors making an effort to release compatible software, the onus is on the developer to look for energy efficient operating systems.

REFERENCES

[1] San Murugesan, "Harnessing Green IT: Principles and Practices," IEEE IT Professional, January–February 2008, pp 24-33.

[2] http://www.nytimes.com/2004/05/08/business/intel-halts-development-of-2-new microprocessors.html?pagewanted=1. Retrieved July 15, 2011

[3] M. Seager. What are the future trends in highperformance interconnects for parallel computers? In IEEE Symposium on High-Performance Interconnects (HotI), Aug. 2004

[4] M. Warren, E. Weigle, and W. Feng. High-density computing: A 240-node beowulf in one cubic meter. In Proceedings of SC2002, Nov. 2002.

[5] Sharma, S. and others. Making a case for a green500 list. In Proceedings of 20th IEEE International Parallel and Distributed Processing Symposium, 2006, pp 343.

[6] A. Vance. DEC veterans prepare chip challenge for Intel, Amd, Ibm and Sun.http://www.theregister.co.uk/2005/10/24/pasemi power/,October 2005.

[7] NVIDIA GTX 200 Technical Brief; see http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf

[8] See ftp://ftp.yokogawa.co.jp/nsfl/com/pr/jkmxc/gs/GS77C01E01-01E_005.pdf

[9] See http://www.texmate.com/media/pdf/2011/03/DI-503_NZ304_L6-24-04.1.pdf

[10] See http://www.lem.com/docs/products/la%2055-p%20sp23%20e.pdf

[11] See http://site.gridconnect.com/docs/PDF/NET485_UM_800240_c.pdf

[12] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In Proceedings of Ninth SIAM Conference on Parallel Processing for Scientific Computing, 1999, CD-ROM Proceedings

[13] See http://gcc.gnu.org/onlinedocs/gcc-4.3.3/gcc/

[14] Dagum , L., and Menon , R. 1998. OpenMP: An industry-standard API for shared-memory programming. IEEE Comput.Sci. Eng. 5, 1, 46–55.

[15] See http://openmp.org/wp/about-openmp/

[16] W. Gropp, E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, 1999.

[17] See http://developer.nvidia.com/cuda-toolkit-40

[18] Luszczek, P., Bailey, D., Dongarra, J., Kepner, J., Lucas, R., Rabenseifner, R., Takahashi, D.: The HPC Challenge (HPCC) Benchmark Suite. In: SC 2006 Conference Tutorial. IEEE, Los Alamitos (2006)

[19] Antoine Petitet, R. Clint Whaley, Jack J. Dongarra, and Andy Cleary. HPL – A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. Innovative Computing Laboratory,September 2000. Available at http://icl.cs.utk.edu/hpl/ and http://www.netlib.org/benchmark/

VITA

Hari Sundararajan was born in Kumbakonam, India, in September 1986. Hari came to the United States to pursue his higher education and joined Louisiana State University (LSU) in August 2004. In August of 2008, he completed his undergraduate education with a Bachelor of Science in Electrical Engineering. Hari then joined the graduate program in LSU in August 2008 and completed the requirements for his master's degree in August 2011. Hari is the elder son of Sundararajan Seshadri and Latha Sundaraarjan and has a brother, Badri Sundararajan.