The LNM Institute of Information Technology

Performance Analysis and Optimization of Individualised CPU-GPU Core Systems

A report submitted for the Operating Systems

Done by

Names of Students	Roll No
Anshul Gupta	Y11UC047
Ashish Agarwal	Y11UC062
Nikunj Gupta	Y11UC155
P Sai Krishna	Y11UC159
Parvinder Singh Kalsi	Y11UC161

Submitted on : April 16,2013

Preface

This project **"Performance Analysis and Optimization of Individualised CPU-GPU Core Systems"** has been undertaken keeping in minds the heights "Information Technology has reached" and when everything is powered with computers does make a great difference.

Initially, the CPUs are responsible for handling all of the computing and instructions that it receives from the user and the system. However, with the increaseof technology and the demand of technology, it was best to take some of the pressure of the CPU and give it other processors. In comparison to CPUs, GPUs have more transistors that can handle more work and offers greater resolutions. Most of the GPUs transistors perform calculation related to 3D technologies. They wereoriginally used to accelerate the memory-intensive work of texturemapping and rendering polygons. Many GPUs also support technologies for advanced gaming or digital playback, offering better and advanced systems.

Acknowledgement

We would like to take opportunity to express our humble gratitude to Prof. Gaurav Somani who motivated and guided us to executed this project. His constant guidance and willingness to share his vast knowledge made us understand this project and its manifestations in great depths and helped us to work on the tasks assigned to us.

We are highly thankful to our project internal guide Miss. Anamika Sharma whose invaluable guidance helped us understand the project better.

Although there may be many who remain unacknowledged in this humble note of gratitude there are none who remain unappreciated.

April 2013

Abstract

Today's supercomputers are computers of tomorrow and GPU processors are bridge between them today. **A graphics processing unit** or **GPU** is a specialized processor that offloads 3D or 2D graphics rendering from the microprocessor.

GPU computing is the use of aGPU to do general purpose scientific andengineering computing. The model for GPU computing is to use a CPU and GPU together in a heterogeneous computingmodel. The sequential part of the application runs on the CPU and the computationally-intensive part runs on the GPU. From the users perspective, the application just runs faster because it isusing the high-performance of the GPU to boost performance. Computing is evolving from"central processing" on the CPU to "co-processing" on the CPU and GPU. To enable this newcomputing paradigm, **NVIDIA invented the CUDA (Compute Unified Device Architecture)** parallel computing architecture.

The **NVIDIA ®Tesla 20-series** is designed from the groundup for high performance computing. Based on the next generation CUDA GPU architecture. When compared to the latest quad-core CPU, Tesla 20-series GPU computingprocessors deliver equivalent performance at **1/20th the power consumption and 1/10th the cost** i.e. **it gives thepower of super computer in a pc based workstation.**

Hardware Specifications

GPU : Nvidia Tesla C2050
FORM FACTOR : 9.75 PcIe x16 form factor
Number Of Tesla GPUs: 1
Number of CUDA cores: 448
Frequency of CUDA cores: 1.15 GHz
Double Precision Floating Point Performance (Peak): 515 Gflops
Single Precision Floating Point Performance (Peak): 1.03 Tflops
Total Dedicated: 3 GB GDDR5
Memory Speed: 1.5 GHz
Memory Interface: 384-bit
Memory Bandwidth: 144 GB/sec
Power Consumption: 238W TDP
System Interface: PCIe x216 Gen2
Display Support: Dual-Link DVI-I:1
Maximum Display Resolution: 60Hz, 2560x1600

CPU : Intel Xeon Processor E5630
Number of Cores : 4
Number of Threads : 8
Clock Speed : 2.53 GHz
Max Turbo Frequency : 2.8 GHz
Intel Smart Cache : 12 MB
Intel QPI Speed : 5.86 GT/s
Number of QPI Links : 2
Instruction Set : 64-bit
Instruction Set Extensions: SSE4.2
Lithography: 32 nm
Max TDP : 80 W
VID Voltage Range: 0.750V-1.350VPU

GPU : Nvidia GeForce GT 525M CUDA Cores: 96
Processor Clock Tester(MHz): 1200 MHz
Texture Fill Rate (billion/sec): 9.6
Memory Clock: 900 Mhz
Memory Interface: DDR3
Memory Interface Width: 128-bit
Memory Bandwidth: 28.8

CPU : Intel Core i5-2450M Processor
Number of Cores: 2
Number of Threads: 4
Clock Speed: 2.5 GHz
Max Turbo Frequency: 3.1 GHz
Intel Smart Cache: 3 MB
DMI: 5 GT/s
Instruction Set: 64-bit
Instruction Set Extensions: AVX
Lithography: 32 nm
Max TDP: 35 W

GPU : Nvidia Geforce GT 630m
CUDA Cores: 96
Graphics Clock (MHz): 800 MHz
Texture Fill Rate (billion/sec): Up to 12.8
Memory Interface: DDR3 GDDR5
Memory Interface Width: Up to 128bit
Memory Bandwidth (GB/sec): Up to 32.0
Bus Support: PCI Express 2.0
Maximum Digital Resolution: Up to 2560x1600

CPU : Intel Core i7-3610QM Processor
Number of Cores: 4
Number of Threads : 8
Clock Speed: 2.3 GHz
Max Turbo Frequency: 3.3 GHz
Intel Smart Cache: 6 MB
DMI: 5 GT/s
Instruction Set: 64-bit
Instruction Set Extensions: AVX
Lithography: 22 nm
Max TDP: 45 W

Contents

Chapter 1

Introduction

1.1 Introduction to GPU

A **Graphics Processing Unit (GPU)**, also occasionally called **visual processing unit (VPU)**, is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the building of images in a frame buffer intended for output to a display. GPUs are used in embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. In a personal computer, a GPU can be present on a video card, or it can be on the motherboard orin certain CPUson the CPU die.

1.2 Introduction to CUDA

CUDA also known as **Compute Unified Device Architecture** is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the latest Nvidia GPUs become accessible for computation like CPUs. Unlike CPUs, however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly. This approach of solving general-purpose (i.e., not exclusively graphics) problems on GPUs is known as GPGPU.

Chapter 2

Initial Findings

2.1 How CPUs and GPUs differ ?

1. Latency Intolerance versus Latency Tolerance
2. Task Parallelism versus Data Parallelism
3. Multi-threaded Cores versus SIMT (Single Instruction Multiple Thread) Cores
4. 10s of Threads versus 10,000s of Threads

2.2 Latency and Throughput

1. Latency is a time delaybetween the moment something is initiated, and the moment one of its effects begins or becomes detectable
2. For example, the time delay between a request for texture reading and texture data returns
3. Throughput is the amount of work done in a given amount of time
4. For example, how many triangles processed per second
5. CPUs are low latency low throughput processors
6. GPUs are high latency high throughput processors

2.3 Latency

GPUs are designed for tasks that can tolerate latency

1. Example: Graphics in a game (simplified scenario):
2. To be efficient, GPUs must have high throughput, i.e. processing millions of pixels in a single frame

2.4 CPUs are designed to minimize latency

1. Example: Mouse or keyboard input
2. Caches are needed to minimize latency
3. CPUs are designed to maximize running operations out of cache
4. Instruction pre-fetch
5. Out-of-order execution, flow control
6. CPUs needa large cache, GPUs do not
7. GPUs can dedicate more of the transistor area to computation horsepower

2.5 Parallism in GPU v. GPU

2.5.1 CPU

1. CPUs use task parallelism.
2. Multiple tasks map to multiple threads.
3. Tasks run different instructions.
4. 10s of relatively heavyweight threads run on 10s of cores.
5. Each thread managed and scheduled explicitly.
6. Each thread has to be individually programmed.

2.5.2 GPU

1. GPUs use data parallelism.
2. SIMD model (Single Instruction Multiple Data).
3. Same instruction on different data.
4. 10,000s of lightweight threads on 100s of cores.
5. Threads are managed and scheduled by hardware.
6. Programming done for batches of threads (e.g. one pixel shader per group of pixels, or draw call).

2.6 Why are we still using CPUs instead of GPUs?

GPUs have far more processor cores than CPUs, but because each GPU core runs significantly slower than a CPU core and do not have the features needed for modern operating systems, they are not appropriate for performing most of the processing in everyday computing.Features missing from GPUs include interrupts and virtual memory, which are required to implement a modern operating system. Furthermore, GPUs use a fundamentally different architecture; one would have to program an application specifically for a GPU for it to work, and significantly different techniques are required to program GPUs. The GPU is not faster than the CPU. The CPU excels at doing complex manipulations to a small set of data, the GPU excels at doing simple manipulations to a large set of data.

The reason why we are still using CPU is not because x86 is the king of CPU architecture and Windows is written for x86, the reason why we are still using CPU is because the kind of tasks that an OS needs to do, i.e. making decisions, is run more efficiently on a CPU architecture. An OS needs to look at a number of different types of data and make various decisions which all depends on each other; this kind of job does not easily parallelizes, at least not into an SIMD architecture.

Chapter 3

Problems faced while Installation

3.1 How to install NVidia driver in Fedora 18

Fedora 18 comes with open source **NOUVEAU driver** for NVIDIA graphics card. To install the NVIDIA propriteray driver follow the below steps.

1. Blacklist the nouveau driver: Add below line to /etc/modprobe.d/blacklist.conf file blacklist nouveau.
2. Rebuild the initramfs image file using dracut:
   * Backup the initramfs file

   ```
   $ sudo mv /boot/initramfs-(uname -r).img /boot/initramfs-(uname -r).img.bak
   ```

   * Rebuild the initramfs file

   ```
   $ sudo dracut -v /boot/initramfs-(uname -r).img (uname -r)
   ```

3. Reboot the system to runlevel 3 (without graphics)
4. Check that nouveau driver is not loaded

   ```
   $ lsmod | grep nouveau
   ```

5. Run the NVidia driver package

   ```
   $ sudo ./NVIDIA-Linux-x86\_64-195.36.15-pkg2.run
   ```

   Above command will create xorg.conf file in /etc/X11 directory which is responsible to use NVidia driver in X.

6. Restart the system and NVidia driver will be used now.

3.2 How to install NVidia driver in Fedora 18

---

1. Get the latest NVIDIA driver from here. Reboot with run-level 3 and install the driver. This should be pretty much straight-forward. Reboot after install.

2. Get the latest CUDA Toolkit.

3. Install the toolkit. This is a little bit tricky as the installer will claim that you have the wrong GCC version. In order to overcome this problem, run the installer with

   ```
   >sh cuda_5.0.35_linux_64_fedora16-1.run -override compiler
   ```

   Skip the driver installation and select [y] for the toolkit and sample files.

4. Although in step (3) the GCC version check has been skipped, it is still hard-coded in one of the CUDA header files. Thus, go to line 80 of

   ```
   /usr/local/cuda-5.0/include/host_config.h
   ```

   (or wherever the installer has put this header file) and change

   ```
   #if __GNUC__ > 4 || (__GNUC__ == 4 && __GNUC_MINOR__ > 6)
       to
   #if __GNUC__ > 4 || (__GNUC__ == 4 && __GNUC_MINOR__ > 7)
   ```

   Now NVCC wont complain about this.

5. When you try to compile your project NVCC still claims about this

   ```
   ... atomicity.h(48): error: identifier "__atomic_fetch_add" is undefined
   ... atomicity.h(52): error: identifier "__atomic_fetch_add" is undefined
   ```

   We found a nice workaround for this problem here. All you need to do is

   ```
   > echo "#undef _GLIBCXX_ATOMIC_BUILTINS"
     > /usr/local/include/undef_atomics_int128.h
   > echo "#undef _GLIBCXX_USE_INT128"
     >> /usr/local/include/undef_atomics_int128.h
   ```

and include this file when compiling with NVCC, i.e.

```
> nvcc.bin --pre-include /usr/local/include/undef_atomics_int128.h
  [other commands]
```

Done. That worked for me, and I could compile the examples (after modifying the Makefiles with the –pre-include trick from (5) without any troubles. Results from the NVIDIA samples look reasonable, what makes me think that CUDA 5.0 should work without (large) troubles on F18.

```
After following the steps above you may need to do the following
to build the examples.

A) The Makefiles don't pay attention to LD_LIBRARY_PATH or
the ldconfig.
Assuming you installed to /usr/local, they use the path
/usr/local/cuda/lib64
but disregard the /usr/lib64/nvidia/ directory in which the libcuda*so
is located. To fix ad a symlink in the /usr/local/cuda/lib64 directory:
$ cd /usr/local/cuda/lib64
$ ln -s /usr/lib64/nvidia/libcuda.so libcuda.so

B) Run make from the /usr/local/cuda/samples directory as follows:
$ cd /usr/local/cuda/samples
$ make -k EXTRA_NVCCFLAGS=
"--pre-include /usr/local/include/undef_atomics_int128.h"
where /usr/local/include/undef_atomics_int128.h is the header file
constructed in the previous post.
The "-k" tells make to continue even if a particular example fails to build,
e.g. for examples requiring something you don't have installed such as mpi.
```

Chapter 4

Benchmark-I

4.1 The Task

The objective of the program is simple. It generates an array up to a certain length of integers. It then computes the square roots of these, saving them to another array of the same length. This operation it performs on both the CPU and the GPU, and reports the process times taken on the two and their ratio. These three numbers are measured for a range of process size (total number of operations required for the process). Therefore, with the array size fixed, the task is instead repeated, i.e. square roots of the same first array are computed and repeatedly saved to the second array, replacing the identical values before. In terms of utility this operation is an unproductive repetition, but in terms of analysing optimization, which is our goal, this serves equally well as any other, more utilitarian method of lengthening the process. In fact, increasing the array size would have meant computing square roots of increasingly larger numbers, which is not a proportional increase in task size. However, repeating the same computation is a linear way to increase the task size, and thus the results may conveniently be plotted on a linear scale. The repetition loop is lengthened in steps of 100. For each length of the loop, process times on the GPU, CPU and their ratio are measured for 10 samples, and the average of each is written to a file.

4.2 Code Walkthrough

Code Walkthrough mentioned in file codewalkthrough.

4.3 Results

4.3.1 Output

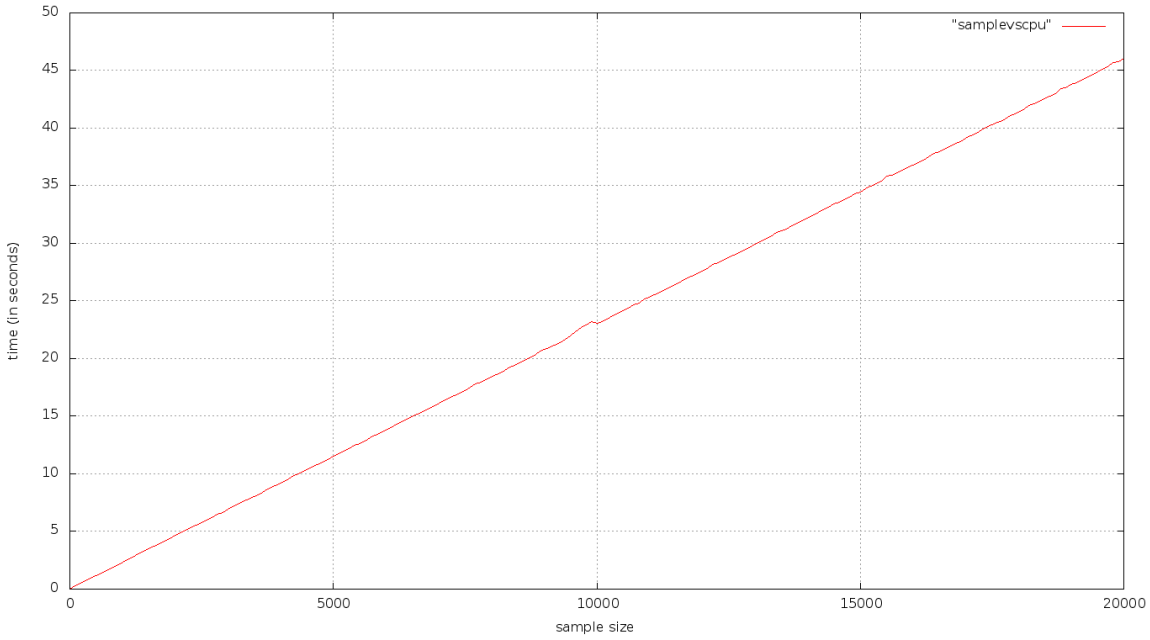
The program displays information from each sample on screen, but writes only the statistics from each sampling to file, without information from all the samples. The first allows easy debugging, while the second allows easy analysis and plotting of the final data.

The Outputs have been mentioned in the following files :

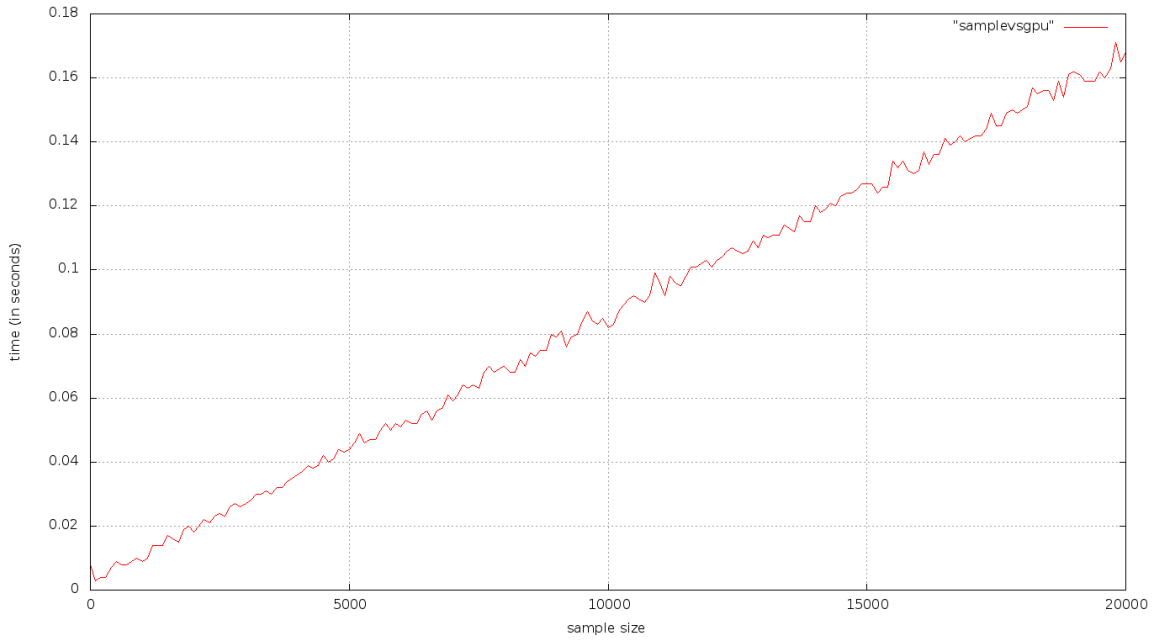
1. samplevsgpu for GPU Result
2. samplevscpu for CPU Result
3. samplevsratio for GPU-CPU Ratio
4. time.txt for Time

4.3.2 Plots

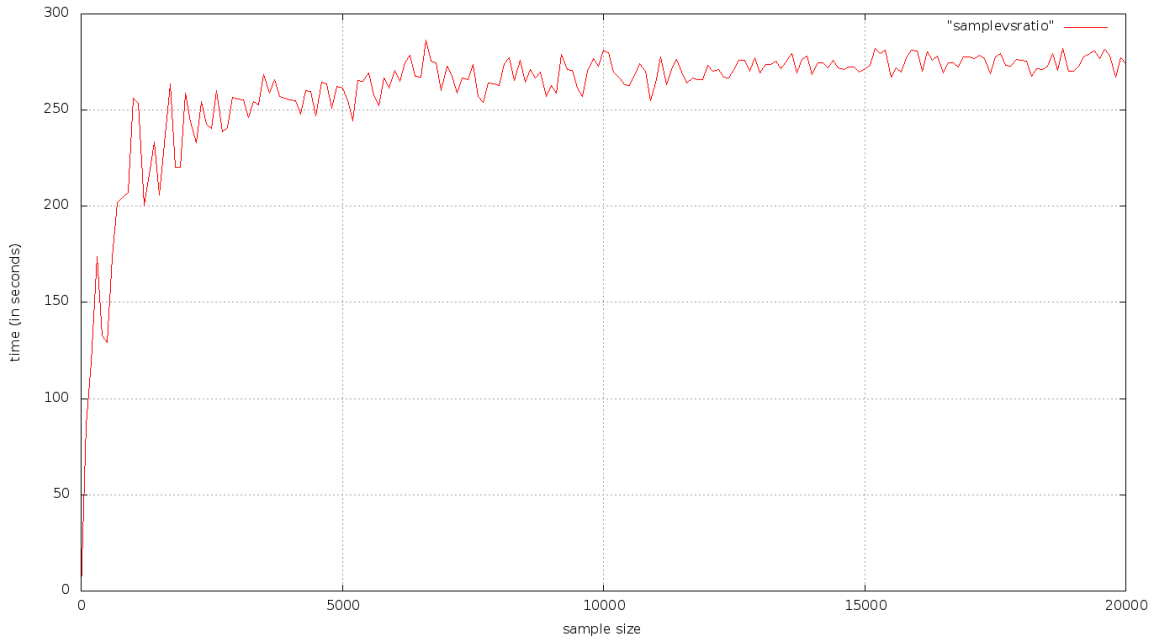
CPU Process Times



GPU Process Times



CPU / GPU Ratio



4.4 Inferences

1. For smaller tasks, the GPU is not much faster than the CPU as the data transfer overhead between host and device costs more than time saved by parallelization.

2. The ratio between GPU and CPU speeds, however, does not keep rising with increasing task size. It reaches an asymptote of about a 280-fold efficiency. This occurs when the transfer overheads take negligible time compared to that taken by the actual arithmetical computations on the device.

3. The above code follows Amdahl's Law The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 as shown in the diagram, no matter how many processors are used.

Chapter 5

Benchmark-II

5.1 The Task

The objective of the program is simple. The function takes an input array, A, and multiplies it by a constant here **2** to create a new array. In this operation, array multiplication is performed on both the CPU and the GPU, and reports the process times taken on the two. These two numbers are measured for a range of process size (total number of operations required for the process).

5.2 Code Walktrough

Code Walkthrough mentioned in file codewalkthrough.

5.3 Results

The program depicits the capabilities of CPU and GPU indivudially. Elements in array v/s time taken in CPU has a constant positive slope whereas the slope for the same graph plotted is zero. No matter how many elements are in array, the time taken by GPU remains constant (with very slight variation). There are few points where the graph shoots up for both CPU and GPU.

5.3.1 Output

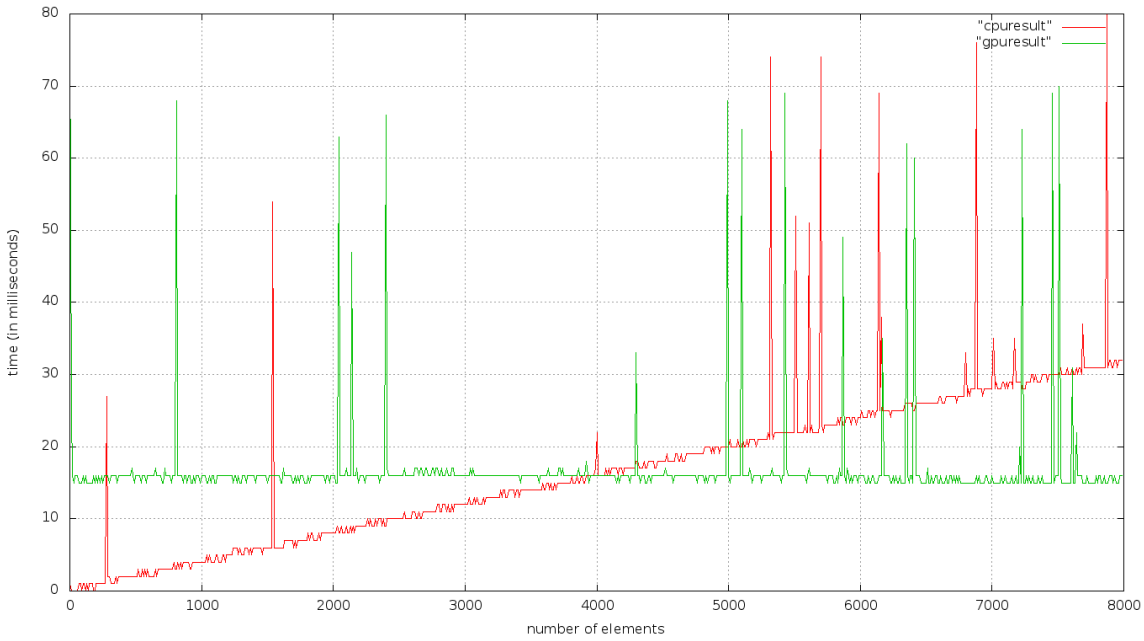
The output of the program in which the array is multiplied with a constant **'2'** have been mentioned in the following files :

1. gpuresult for GPU Result

2. cpuresult for CPU Result

5.3.2 Plots

CPU vs GPU



5.4 Inferences

1. When this code is run, you will find that the CPU is faster than the GPU for an array of 3,900 elements. If you increase the size to 3,900 to about 4,000 elements in the array, the two will essentially tie. Above 4,000 elements in the array, and the GPU will win.

2. The threads should be running in groups of at least 32 (since the kernel issues commands equal to the warp size [32 in our case]) for best performance, with total number of threads numbering in the thousands. Branches in the program code do not impact performance significantly, provided that each of 32 threads takes the same execution path; the SIMD execution model becomes a significant limitation for any inherently divergent task.

3. The overhead in creating threads on GPU kernel remains same, it does not depends on the number of elements.

Chapter 6

Benchmark-III

6.1 The Task

The objective of the program is simple. Its the brute force euclidian distance transform. Basically, in a binary image, for each pixel in the foreground we verify what is the 2D euclidian distance to the nearest pixel in the background.

6.2 Code Walkthrough

Code Walkthrough mentioned in file codewalkthrough.

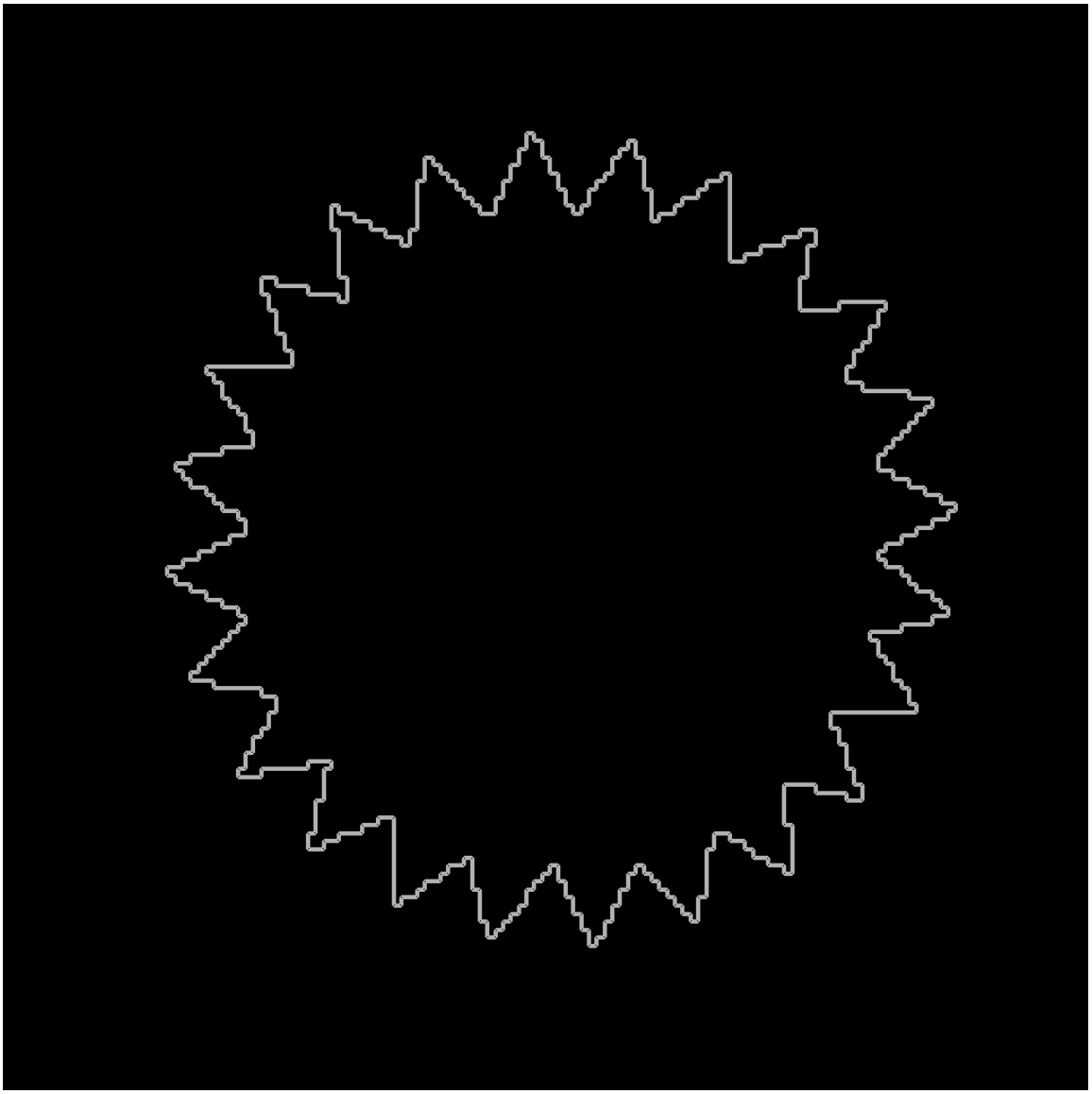
6.3 Result

6.3.1 Output

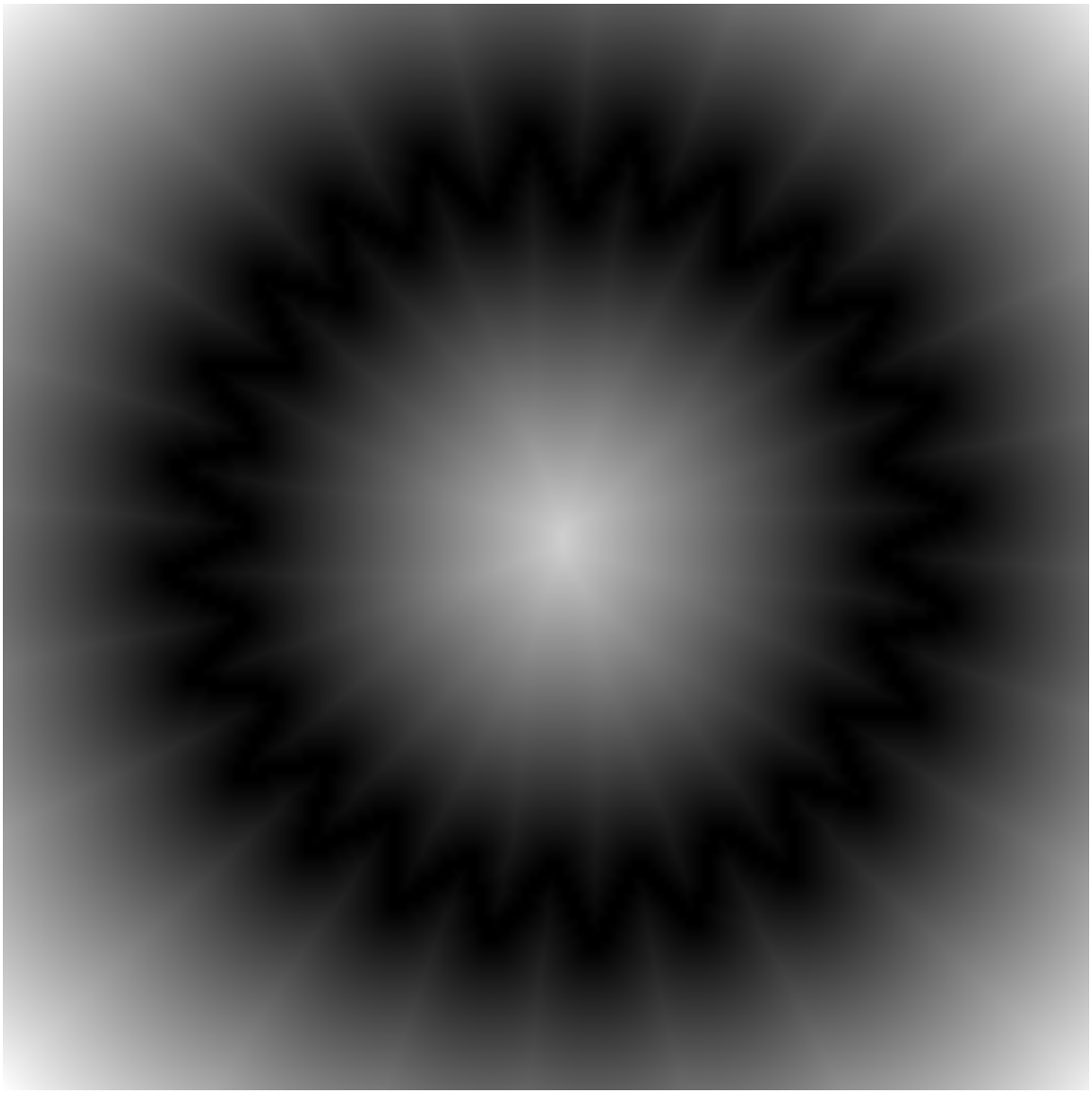
It took about 35.8 seconds for a 1 megapixel image in a Tesla C2050 GPU and about 1 hour on normal CPU.

6.3.2 Images

Input Image



Output Image



6.4 Inferences

- From this it could be inferred that brute force ran fast on GPU rather than CPU.

Chapter 7

Benchmark-IV

7.1 The Task

What is Video Rendering

Rendering is the computer's way of taking effects, such as color correction or transitions, and adding them to the actual video in order to make the video playback smooth. Without the rendering process, the video playback would be choppy because the computer would have to compute the video and the effects separately.

Benchmarking GPU Acceleration in Vegas Pro 12

Vegas Pro 12 leverages the processing capabilities of modern GPUs (Graphic Processing Units) using the industry-standard OpenCL framework. Rather than being tied to a single manufacturer or technology, this hardware-agnostic approach enables Vegas Pro 12 users to enjoy remarkable performance improvements across a broad range of popularly-priced, widely available GPU devices. By utilizing the amazing parallel computing resources of the GPU for video processing, the main CPU is freed up for other tasks, such as video decoding and user interface display.
Vegas Pro 12 accelerates both video playback and rendering, providing improved performance results from start to finish. Significant portions of the application were entirely reworked, resulting in an enhanced and more creative editing experience. Over 45 effects, transitions, generators and compositors are GPU-accelerated in Vegas Pro 12, as well as a substantial amount of built-in video processing such as crossfades, fades, alpha compositing, framerate resampling, interlace processing, pan/crop, track motion, opacity, fade-to-color, and multicamera display.

7.2 Results

Video Details:
Video Resolution: 1920 x 1080
fps: 29
filetype: NTSC-mp4
Playback Time: 6:47 min

Intel Xeon with Nvidia Tesla C2050
Rendering Time: 5:40 min
Output format: Mainconcept Internet HD 1080p (mp4)
CPU Usage : 70-75%
GPU Usage : 60-70%

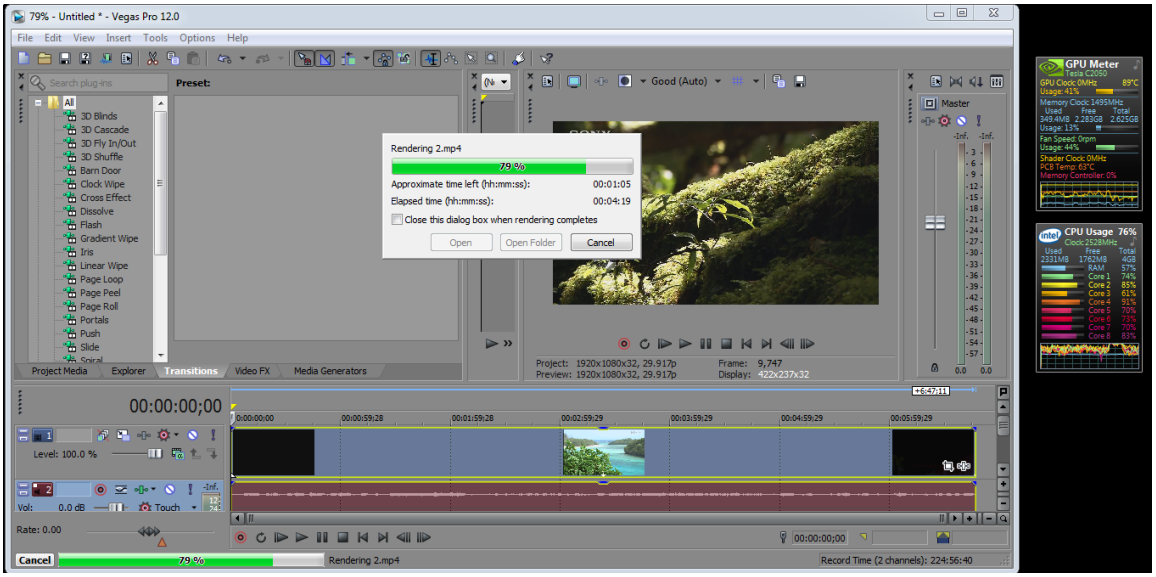
Intel Xeon without Nvidia Tesla C2050
Rendering Time: 19:34 min
Output format: Mainconcept Internet HD 1080p (mp4)
CPU Usage : 90-95%
GPU Usage : None

Intel i5-2450M with Nvidia Geforce GT 525m
Rendering Time: 16:18 min
Output format: Mainconcept Internet HD 1080p (mp4)
CPU Usage : 70-75%
GPU Usage : 60-70%

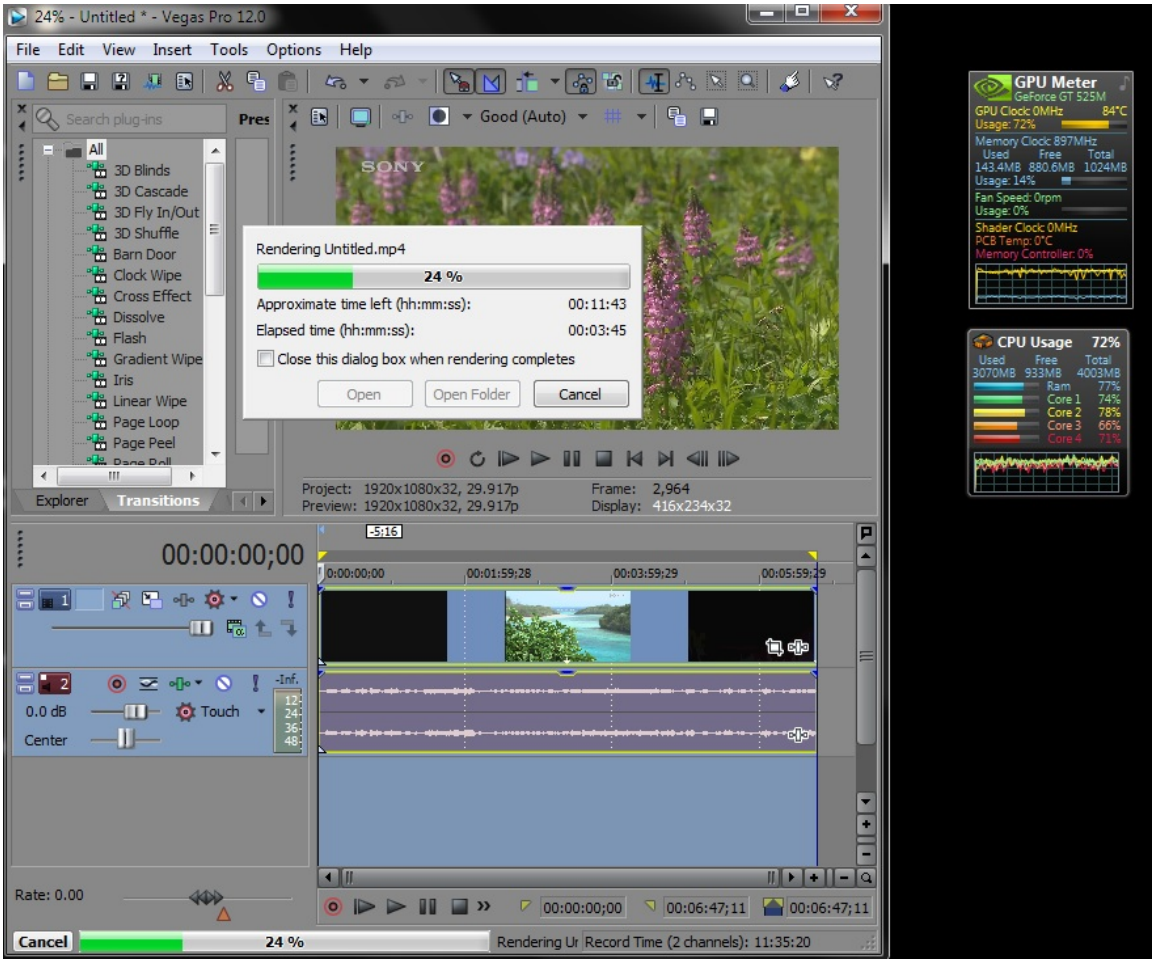
Intel i7-3610QM with Nvidia Geforce GT 630m
Rendering Time: 9:01 min
Output format: Mainconcept Internet HD 1080p (mp4)
CPU Usage : 25-35%
GPU Usage : 85-90%

7.3 Output

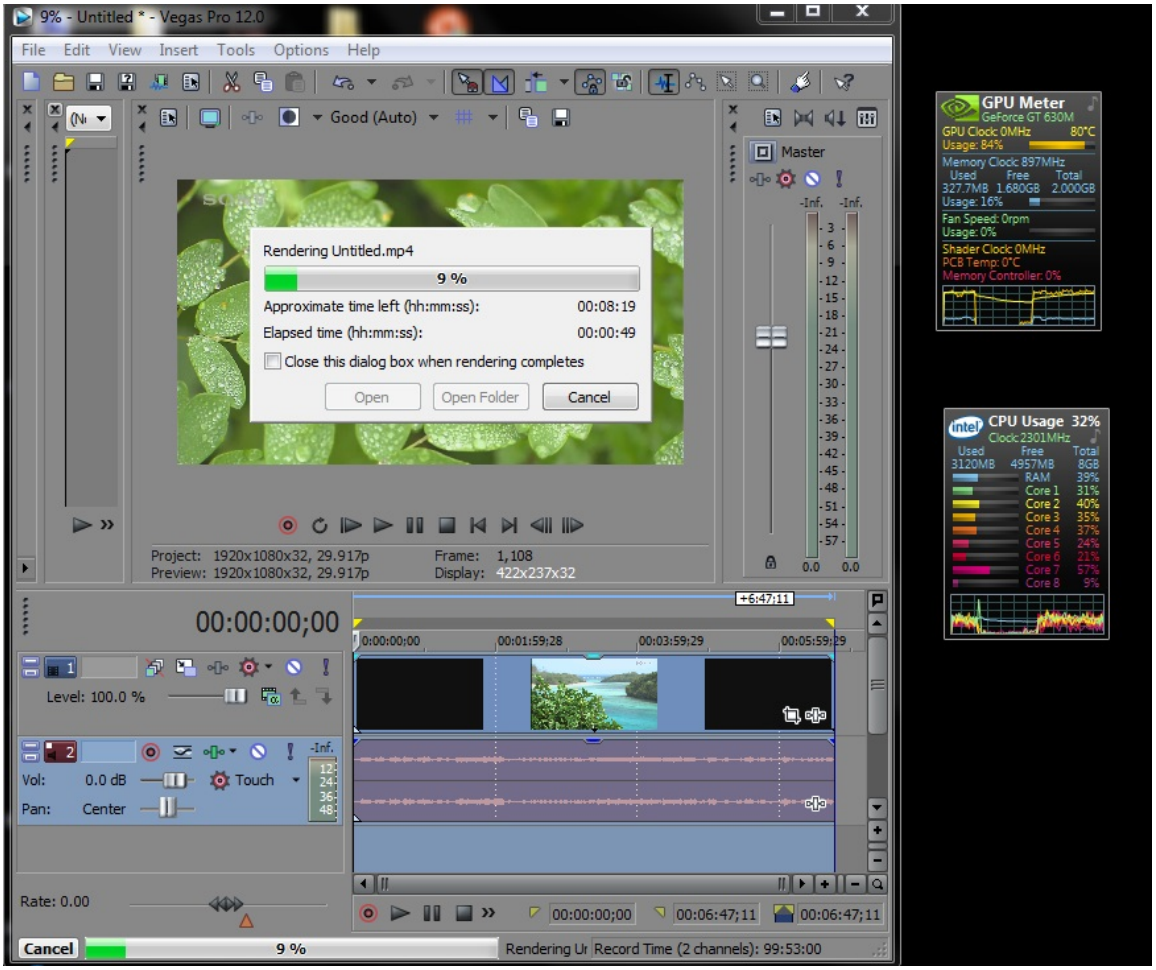
Intel Xeon with Nvidia Tesla C2050



Intel i5-2450M with Nvidia Geforce GT 525m



Intel i7-3610QM with Nvidia Geforce GT 630m



Chapter 8

Bottlenecks

1. Copying between host and device memory may incur a performance hit due to system bus bandwidth and latency (this can be partly alleviated with asynchronous memory transfers, handled by the GPU's DMA engine)

2. Threads should be running in groups of at least 32 for best performance, with total number of threads numbering in the thousands. Branches in the program code do not impact performance significantly, provided that each of 32 threads takes the same execution path; the SIMD execution model becomes a significant limitation for any inherently divergent task

3. Consider we have an application in which we need to create an array that has 20,000 integers. So the amount of memory we will need to fit 20,000 integers = 20,000 * 4B = 80KB. But the maximum amount of shared memory per multiprocessor is 48 KB.

4. For a large number of threads, we can't assign more than 1024 threads per block.

5. Local memory being limited to 512 KB could hurt the performance in two ways-

   (a) Increased pressure on the memory bus

   (b) Increased instruction count

Chapter 9

Conclusion

9.1 Project Status

- The project is not complete in its entireity as a lot more modules can be implemented. We tried to implement the Brute Force technique but due to time boundation we were not able to implement it.
- Some of the benchmarking tools could not be implemented due to insufficient resources.

9.2 Future Intentions with the project

- It has been quite a learning process and the project made can surely undergo a lot of modications to become a well equipped parallelizer.
- We would continue this to make sure that Brute Force (MD5 Hashing) is implemented and a bit of more study of how can this be improved would be done.