CS451 Project 1
Design Document

Adrian Birylo
Daniel Hitchings

February 5, 2014

1 Overall Program Design

To tackle the development of this program, we decided that it would be easier to break the program into several smaller programs, each with their own challenges. Then, in order to test the full functionality, we would tie all the programs together with a script to run however many tests we needed. Each program would run the test many times to eliminate the overhead time.

1.1 CPU Benchmarking Program

The initial plan for this program was to simply perform a large number of floating point and integer operations and time them. From there it would be simple enough to calculate the exact number of operations done and the time taken, thus giving us the number of operations per second. We would also do the same thing for integer operations.

However, simply doing the operations in a large loop would cause issues with the pipelining in modern processors and would give us skewed results. To overcome this, each iteration of the loop would do a number of operations, 24 in our case. Having a separate function to do each of these numerous loops made threading simple, since we could just call the function for each thread. There are no issues with deadlock or synchronization, since each thread is working independently.

One possible improvement to this program would be to implement the operations being done in assembly. This would solve any problems that could arise from compiler optimization, however, the improvements might not even be noticable.

1.2 GPU Benchmarking Program

This program was going to be very similar to the CPU benchmarking program, but implemented using CUDA, to utilize the massively multi-threaded capabilities of the GPU. Since this was developed after the CPU benchmarking program, there were few complications with it. The biggest issue was to ensure that the concurrency was executing correctly, so that we could get the results that we expected.

1.3 Memory Benchmarking Program

To benchmark the memory, we needed to do a large number of memory read/writes. The best way to do this is to use the memcpy function in C. However, the amount of memory being copied had to be larger than the cache size of the processor, to ensure that we were measuring memory speed and not cache speed. To avoid this, the chunk of memory that would be copied was close to 1GB, much larger than the few MBs of most modern processors.

I'm not sure if there would be a way to avoid the issue with processor caching. However, once the memory being accessed is much greater than the size of the cache, the effect of caching will become minimal.

1.4 Disk Benchmarking Program

The disk benchmarking program would be very similar to the memory one. To implement it, we broke it down into more individual operations than the memory benchmarking program, which made it easier to implement and test. Just as with the memory, caching would be an issue when trying to test the disk speed. To overcome this, we found out that there was a way in linux to clear the disk cache, so we just had to do that between each operation.

1.5 Network Benchmarking Program

The networking benchmark program was the hardest on to make work since it required a server and client program. The server is the simpler of the two programs since it only listens for data on the socket and then returns the data back to the sender. The client code has the benchmarking code that loops through and sends data on the socket at the specified data size and then listens for the data back again. Thus the bandwidth and latency of the network could be measured. The way the test was done it only used the loopback device thus the speed is higher than standard networking speed would be since it is just copying data in memory.

2 Possible Improvements

To improve upon these benchmarking tests, the biggest issue would be to find a balance between doing a measurable number of operations and doing an excessive number. With more testing and tweaking, it would probably be possible to configure the size of the tests to get an accurate measurement without taking a long time to do it.

Another improvement would be to get the results as close to theoretical value by using assembly code to get the best performance. Assembly code would be the only way to get the closest to theoretical since higher level languages can make optimizations that are not necessarily the best for a particular device.

Lastly, a nicer frontend might be an improvement, as it would make it easier to perform the tests on various systems. Having a more complete front end would also make it easier for a variety of people to run the tests, giving us more information to analyze.