CSS 600: Independent Study Contract - Final Report

Student: Piotr Warczak
Student ID: 99XXXXX
Quarter: Fall 2011
Credit: 2
Grading: Decimal

Independent Study Title

The GPU version of the MASS library.

Focus and Goals

The current version of the MASS library is written in java programming language and combines the power of every computing node on the network by utilizing both multithreading and multiprocessing. The new version will be implemented in C and C++. It will also support multithreading and multiprocessing and finally CUDA parallel computing architecture utilizing both single and multiple devices. This version will harness the power of the GPU a general purpose parallel processor.

My specific goals of the past quarter were:

- To create a baseline in C language using Wave2D application as an example.
- To implement single thread Wave2D
- To implement multithreading Wave2D
- To implement CUDA enabled Wave2D

Work Completed

During this quarter I have created Wave2D application in C language using a single thread, multithreaded and CUDA enabled application. The reason for this is that CUDA is an extension of C and thus we need to create baseline against which other versions of the program can be measured. This baseline illustrates how much faster the CUDA enabled program is versus single thread and multithreaded versions. Furthermore, once the GPU version of the MASS library is implemented, we can compare the results to identify if there is any difference in program's total execution time due to the potential MASS library overhead. And if any major delays in execution are found, the problem ares will be identified and corrected.

Wave2D Single Thread

This is program was there first step to implement Wave2D in C language. The results between Single thread version and multithreaded version using one thread are so insignificant that I didn't use them when analyzing the results.

Wave2D Mulithreaded

This version uses pthreads and barrier synchronization. To compile the code please use the following:

```
gcc -pthread Wave2DThread.c -o Wave2DThread
```

Multithreaded version uses pthreads and the following synchronization methods:

- Pthread_barrier_init – initializes the barrier with the specified number of threads
- pthread_create – spawns each thread and specifies the function to be executed
- pthread_join – waits for all the threads to finish their assigned work

And the barrier is implemented like this:

```
// Synchronization point
```

```
int rc = pthread_barrier_wait(&barr);
if(rc != 0 && rc != PTHREAD_BARRIER_SERIAL_THREAD)
```

Wave2D CUDA

This version as the name suggests uses CUDA. To compile the code please use the following:

```
nvcc Wave2D.cu -o Wave2D
```

The most important part of CUDA is that once you get the program executing correctly, you need to figure out the number of threads per block and blocks per grid for the program. If improperly configured, one might not gain all the power CUDA device provides. Here is a number of methods I used in this experiment.

500 threads per block (500, 1, 1)

```
dim3 dimBlock(MAX_THREADS_PER_BLOCK, 1, 1);
dim3 dimGrid((SIMULATION_TOTAL_SPACE + dimBlock.x - 1) / dimBlock.x, 1, 1);
```

256 threads per block (16, 16, 1)

```
dim3 dimBlock(MAX_THREADS_PER_BLOCK, 1, 1);
dim3 dimGrid((SIMULATION_SPACE + dimBlock.x - 1)/ dimBlock.x, (SIMULATION_SPACE + dimBlock.y - 1) / dimBlock.y, 1);
```

512 threads per block (32, 16, 1)

```
dim3 dimBlock(MAX_THREADS_PER_BLOCK, 1, 1);
dim3 dimGrid((SIMULATION_SPACE + dimBlock.x - 1)/ dimBlock.x, (SIMULATION_SPACE + dimBlock.y - 1) / dimBlock.y, 1);
```

MAX_THREADS_PER_BLOCK – that's usually the Simulation space; however , if the simulation space gets larger than the maximum number of threads allowed per block than we have to divide the total simulation space between number of blocks we want to use. The Tesla C1060 only supports 512 threads per block.

SIMULATION_SPACE is width and/or height (could be either since I'm only using even sides)
SIMULATION_TOTAL_SPACE is width * height

In order to configure threads per block and blocks per grid, one needs to identify what parameters installed CUDA device provides. There is a program available in the SDK, which I moved to the examples folder in MASS/GPU folder. The program returns multiple devices, since hydra has 3 nvidia cards installed; however, we are only interested in the one we have been using in our experiment. The other two could be used when we are ready to test distributed GPU programming.

Here is the program's output:

```
[dslab@hydra examples]$ QueryCudaDevices
```

CSS 600: Independent Study Contract - Final Report

Student: Piotr Warczak
Student ID: 99XXXXX
Quarter: Fall 2011
Credit: 2
Grading: Decimal

```
CUDA Device Query...
There are 3 CUDA devices.
CUDA Device #0
Major revision number:         1
Minor revision number:         3
Name:                          Tesla C1060
Total global memory:           4294770688
Total shared memory per block: 16384
Total registers per block:     16384
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     512
Maximum dimension 0 of block:  512
Maximum dimension 1 of block:  512
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   65535
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   1
Clock rate:                    1296000
Total constant memory:         65536
Texture alignment:             256
Concurrent copy and execution: Yes
Number of multiprocessors:     30
Kernel execution timeout:      No
```

To summarize, the maximum number of threads are the following:
Maximum number of threads per block : 512
Block dimension = 512 x 512 x 64 (threadIdx.x, threadIdx.y, threadIdx.z)
Possible options:

- 512, 1, 1
- 16, 16, 1 = 256
- 32, 16, 1 = 512
- 32, 32, 1 = 1024 (incorrect since the cuda device only supports 512 threads per block)

Grid dimension = 65535 x 65535 x 1 (blockIdx.x, blockIdx.y, blockIdx.z)
Possible options:

- 65535, 1, 1
- 256, 256, 1
- 64, 64, 16

Based on this number, the largest Wave2D calculation with even sides is 5792 by 5792. In this experiment, I've used 5000 as the largest width and height.

The cards available for testing have 1.3 or lower compute capability and are restricted to the numbers above; however; the newest NVIDIA cards has 2.0 or higher compute capability which supports 1024 threads per block and a third grid dimension. This could significantly improve the results shown here. The numbers for the newest card:

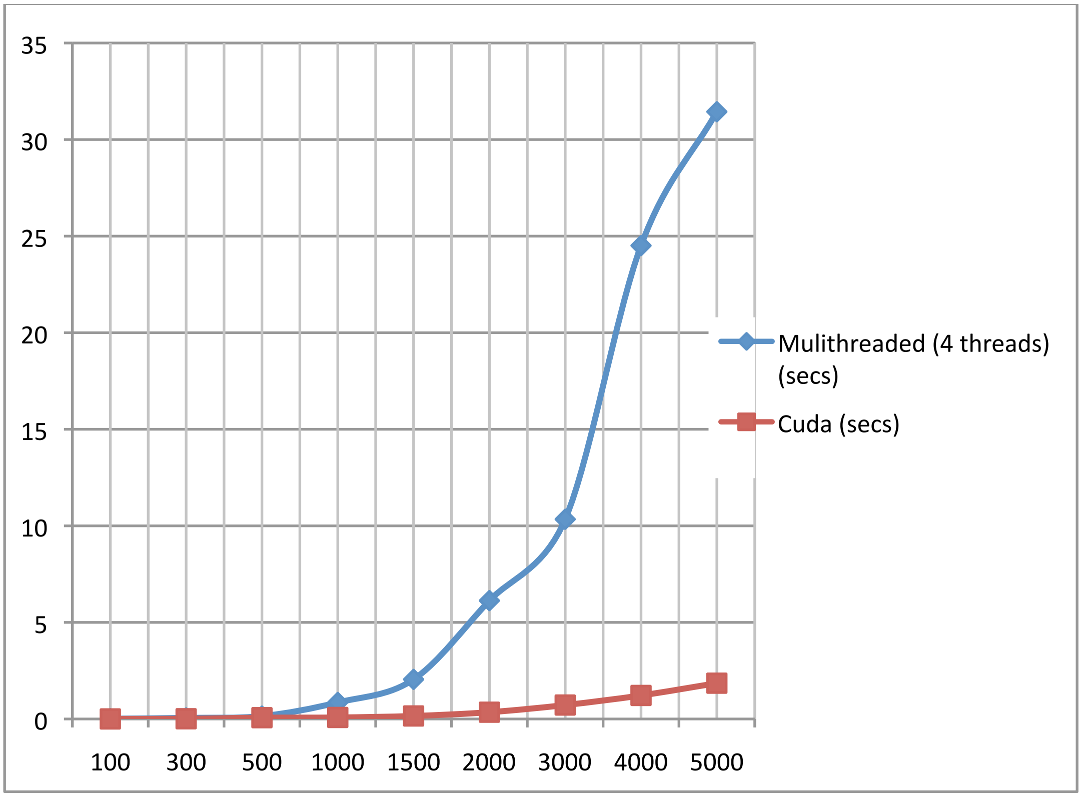
- Threads per Block: 1024
- Block Dimensions: 1024 x 1024 x 64
- Grid Dimensions: 65535 x 65535 x 65535

CSS 600: Independent Study Contract - Final Report

Student: Piotr Warczak
Student ID: 99XXXXX
Quarter: Fall 2011
Credit: 2
Grading: Decimal

The results below compare multithread version running with 4 threads and the CUDA version. In CUDA version I made the thread number equal to Simulation Space up to 500; thereafter, it was also 500 since 512 is the limit of the threads per block. For example, with dimensions 100 by 100, I used 100 threads per block, with 300 I used 300 threads per block and with dimensions 500 by 500 and greater, I used 500 threads per block. I have also tried to use 256 threads configured 16 by 16, and the numbers were obviously slightly slower, by not by much.

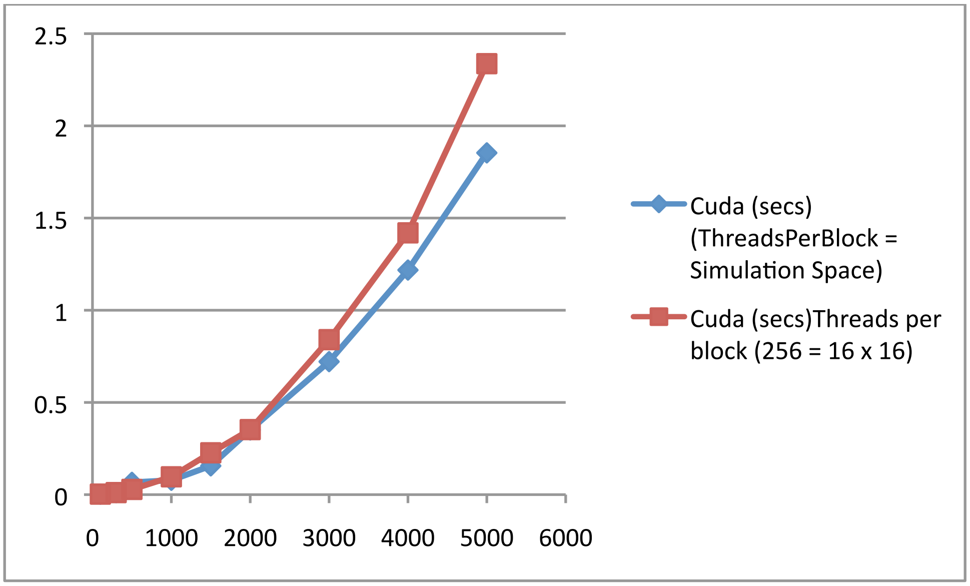
Simulation Space	Simulation Time	Multithread (secs)	CUDA (secs)	Improvement (%)
100	100	0.012218	0.002756	443.3236575
100	500	0.078713	0.011086	710.0216489
100	1000	0.155534	0.021247	732.028051
300	100	0.055218	0.011018	501.1617353
300	500	0.398841	0.044054	905.3457121
300	1000	0.660387	0.085429	773.0243828
500	100	0.149133	0.066598	223.9301481
500	500	1.172416	0.283811	413.0974487
500	1000	2.519719	0.554915	454.0729661
1000	100	0.85561	0.077878	1098.654305
1000	500	4.770346	0.286193	1666.828329
1000	1000	9.97026	0.546464	1824.50445



The graph above shows how Wave2D CUDA version outperforms the multithread version. When the Wave2D dimensions are small, the difference is insignificant; however, as we make the dimensions larger the multithreaded version performance exponentially deteriorates while the CUDA version consistently generates great results.

This test showed that although small there is a difference in how you configure threads allocation. Initially, the difference is invisible, but as we get into larger simulation space that small difference starts to add up and eventually becomes a somewhat significant. However, even the improperly configured CUDA version still outperforms multithreaded version by a huge margin.

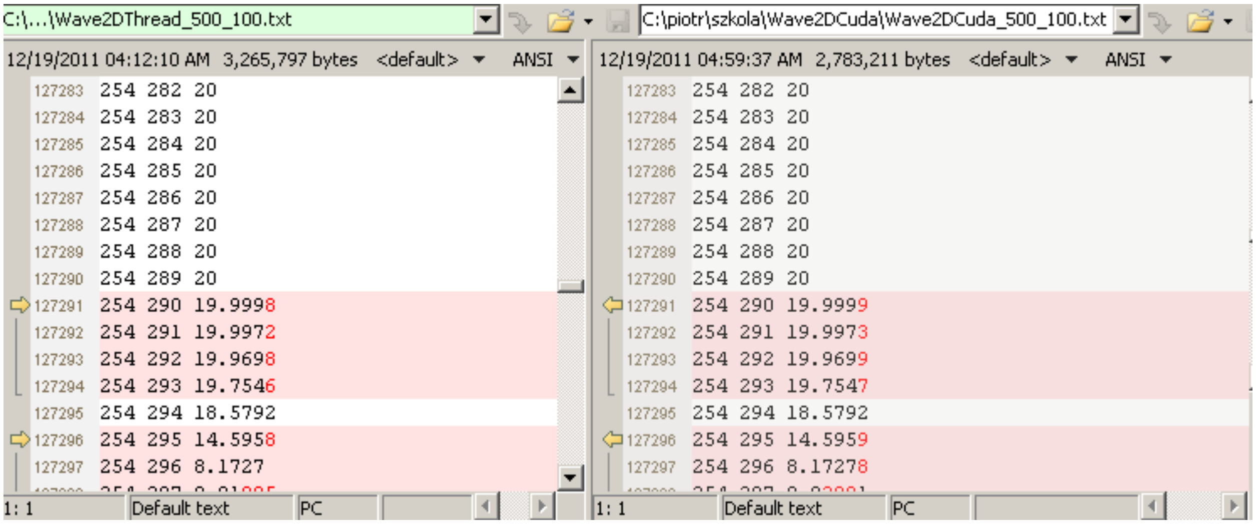
Simulation Space	Cuda (secs)(ThreadsPerBlock = Simulation Space)	Cuda (secs)Threads per block (256 = 16 x 16)
100	0.002756	0.002874
300	0.011018	0.011044
500	0.066598	0.028317
1000	0.077878	0.096686
1500	0.156104	0.226899
2000	0.350629	0.35269
3000	0.721331	0.840495
4000	1.217904	1.419647
5000	1.853456	2.337157



CSS 600: Independent Study Contract - Final Report

Student: Piotr Warczak
Student ID: 99XXXXX
Quarter: Fall 2011
Credit: 2
Grading: Decimal

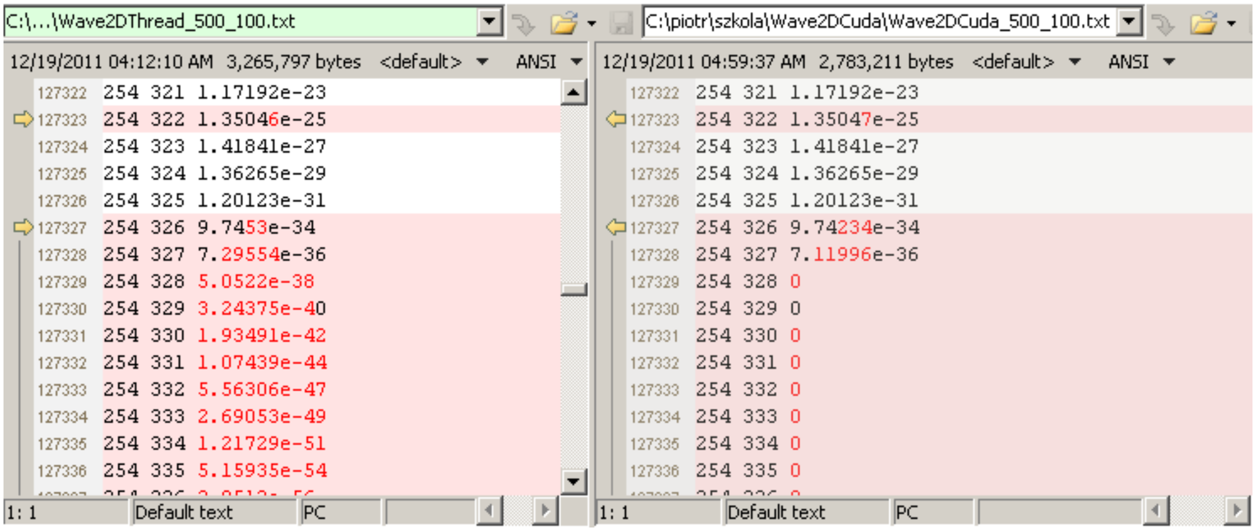
The images below show the difference in floating point precision between host (CPU) and device (GPU) results. These results come from 500 by 500 simulation space and 100 simulation time comparing both multithreaded and CUDA results.



```
C:\...\Wave2DThread_500_100.txt
12/19/2011 04:12:10 AM  3,265,797 bytes  <default>  ANSI
127283 254 282 20
127284 254 283 20
127285 254 284 20
127286 254 285 20
127287 254 286 20
127288 254 287 20
127289 254 288 20
127290 254 289 20
127291 254 290 19.9998
127292 254 291 19.9972
127293 254 292 19.9698
127294 254 293 19.7546
127295 254 294 18.5792
127296 254 295 14.5958
127297 254 296 8.1727
1: 1  Default text  PC
```

```
C:\piotr\szkola\Wave2DCuda\Wave2DCuda_500_100.txt
12/19/2011 04:59:37 AM  2,783,211 bytes  <default>  ANSI
127283 254 282 20
127284 254 283 20
127285 254 284 20
127286 254 285 20
127287 254 286 20
127288 254 287 20
127289 254 288 20
127290 254 289 20
127291 254 290 19.9999
127292 254 291 19.9973
127293 254 292 19.9699
127294 254 293 19.7547
127295 254 294 18.5792
127296 254 295 14.5959
127297 254 296 8.17278
1: 1  Default text  PC
```

As can be seen the 4 decimal point is different due to different rounding methods used by GPU.



```
C:\...\Wave2DThread_500_100.txt
12/19/2011 04:12:10 AM  3,265,797 bytes  <default>  ANSI
127322 254 321 1.17192e-23
127323 254 322 1.35046e-25
127324 254 323 1.41841e-27
127325 254 324 1.36265e-29
127326 254 325 1.20123e-31
127327 254 326 9.7453e-34
127328 254 327 7.29554e-36
127329 254 328 5.0522e-38
127330 254 329 3.24375e-40
127331 254 330 1.93491e-42
127332 254 331 1.07439e-44
127333 254 332 5.56306e-47
127334 254 333 2.69053e-49
127335 254 334 1.21729e-51
127336 254 335 5.15935e-54
1: 1  Default text  PC
```

```
C:\piotr\szkola\Wave2DCuda\Wave2DCuda_500_100.txt
12/19/2011 04:59:37 AM  2,783,211 bytes  <default>  ANSI
127322 254 321 1.17192e-23
127323 254 322 1.35047e-25
127324 254 323 1.41841e-27
127325 254 324 1.36265e-29
127326 254 325 1.20123e-31
127327 254 326 9.74234e-34
127328 254 327 7.11996e-36
127329 254 328 0
127330 254 329 0
127331 254 330 0
127332 254 331 0
127333 254 332 0
127334 254 333 0
127335 254 334 0
127336 254 335 0
1: 1  Default text  PC
```

In this example, it shows that numbers with greater than 36 decimal points are rounded. In this case is zero. The difference is so small that it most cases it doesn’t affect the programs final results.

All code snippets and examples are available at the following directory on hydra machine:
~ dslab/SensorGrid/MASS/GPU/Wave2D
~ dslab/SensorGrid/MASS/GPU/examples

CSS 600: Independent Study Contract - Final Report

Student:	Piotr Warczak	**Quarter:**	Fall 2011
Student ID:	99XXXXX	**Credit:**	2
		Grading:	Decimal

Next quarter

During winter quarter, I plan on converting multithreaded MASS library to C language and then create Wave2D version using the MASS-C library. Once completed, I will analyze the results to identify the overhead of the MASS-C library. Once the MASS-C Library is created the next step is to create a C++ version as well as multiprocessing support.