CSS 600: Independent Study Contract - Final Report

Student:  Piotr Warczak  Quarter:  Fall 2011
Student ID:  99XXXX  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 areas 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:

```bash
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)

```c
    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)

```c
    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)

```c
    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
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
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.

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

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

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

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
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.