Lab Assignment 1 – CSE 591/CSE 392, Spring 2015

Due: Friday, April 24, 11:59pm

This lab will get you started with CUDA programming on NVIDIA GPUs. Upon completion of the assignment please submit the following to me via dropbox and send me the link

- the complete software (all that is needed to build the executable: source code, project files, etc)
- an executable (.exe file) of your work
- a comprehensive report that illustrates with (1) narrative text, (2) code snippets, (3) program output, and (4) run time of all aspects of your work

All of these components are equally important. Please carefully note the policies posted on the class webpage. Please also be aware that this assignment will take a good amount of time to do, so please start right away so you can ask questions if you have them.

Your CUDA program will have the following includes:

```
// C-headers
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// CUDA-headers
#include <cutil.h>

// your kernel, replace "???_kernel" by the name you choose
#include <???_kernel.cu>
```

You will likely also have a .h file with your own definitions, such as a structure for the Matrix data type and various constants. Finally, if you call a C-function to be run on the host from a .cu function, you need to export the C-interface, for example:

```
extern "C"
void C-function(args);
```

Since the key goal of GPU programming is time performance, you need to accurately measure run time for all parts of this lab assignment. To measure CPU time, you may use the following mechanism (taken from http://www.songho.ca/misc/timer/timer.html):

```
#include <iostream>
#include <windows.h>                // for Windows APIs
using namespace std;

int main()
{
    LARGE_INTEGER frequency;        // ticks per second
    LARGE_INTEGER t1, t2;           // ticks
    double elapsedTime;

    // get ticks per second
    QueryPerformanceFrequency(&frequency);

    // start timer
```

```
    QueryPerformanceCounter(&t1);

    // do something
    ...

    // stop timer
    QueryPerformanceCounter(&t2);

    // compute and print the elapsed time in millisec
    elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 /
frequency.QuadPart;
    cout << elapsedTime << " ms.\n";

    return 0;
}
```

To measure the elapsed kernel time on a GPU using CPU timers, consider the following (taken from the NVIDIA CUDA Best Practices Guide).

1. Using the CPU timers: Here it is critical to remember that many CUDA API functions are asynchronous; that is, they return control back to the calling CPU thread prior to completing their work. All kernel launches are asynchronous; so are all memory copy functions with the *Async* suffix on the name. Therefore, to accurately measure the elapsed time for a particular call or sequence of CUDA calls, it is necessary to synchronize the CPU thread with the GPU by calling cudaThreadSynchronize() immediately before starting and stopping the CPU timer. cudaThreadSynchronize() blocks the calling CPU thread until all CUDA calls previously issued by the thread are completed.
2. Using CUDA GPU timers: The CUDA event API provides calls that create and destroy events, record events (via timestamp), and convert timestamp differences into a floating-point value in milliseconds. The following code illustrates their use:

```
cudaEvent_t start, stop; float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 ); kernel<<<grid,threads>>> ( d_odata,
d_idata, size_x, size_y, NUM_REPS); cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop ); cudaEventElapsedTime( &time, start, stop
);
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

3. Using the CUDA Visual Profiler. It comes with the CUDA toolkit and it is probably the easiest way to measure the performance. Proceed as follows:

   (1) Build your executable file, keeping the path and name of ".exe" file in mind.
   (2) Open the NVIDIA (CUDA) Compute Visual Profiler
   (3) Create a Project
   (4) Create a session, fill the path and name of the ".exe" file into the dialog.
   (5) Click "Launch Applications"

   Then the profiler will call the exe file multiple times and generate a spreadsheet about the detailed statistics on GPU and CPU time. Various counts can be selected in the session dialog (step 3). It is easier because you will need to write any code on timing. It will collect all the performance numbers that timing functions calls can do. It is also more powerful than the timing-functions. It will record "cache miss" and "un-coaleased read and write" which is not reported by timing function calls.

1. CPU-based matrix multiplication

Write a program that for now computes the product of two matrices on the CPU. For this, write a function

```
MatMultCPU(float* C, const float* A, const float* B, unsigned int
hA, unsigned int wA, unsigned int wB)
```

that reads two floating point matrices A,B of dimensions (hA,wA) and (hB=wA,wB) and computes C=A*B on the CPU. Assume square matrices where hA=wA. The result will serve later as your gold standard to verify the GPU code. So it will be helpful later to precede your GPU code by this code and verify the result C. To make sure it works for all matrices, randomize values in a matrix initialization routine. For this, use the C-function rand()/(float)RAND_MAX for this and initialize it with srand(seed). Make sure you try hA and wA for small and large values. Allocate memory dynamically based on these values, do not allocate memory statically.

2. GPU-based matrix multiplication with only one thread block

Now develop two functions with extension .cu. One will be the CPU/host code (it can be the main program that calls all other functions, recall the C-interface export comment above) and the other the GPU/device code. The host code will allocate and initialize the two matrices, call the gold standard code, load the matrices into the GPU, and finally read back the results to the CPU. The device code will perform the matrix calculations in parallel. It will adhere to what we discussed in class for the matrix-multiplication approach that only used one thread block. So your matrix will only have a restricted size (what is it?).

3. GPU-based matrix multiplication with many thread blocks

Extend the functions developed above to multiple thread blocks, using the approach discussed in class that tiled the destination matrix into thread blocks. Now you can make the matrices much larger. Be sure to try large matrices so you can appreciate the differences in CPU and GPU timings. Finally, answer the following two questions which will prepare you for the next portion of this assignment:

1. How many times is each element of the input matrices loaded during the execution of the kernel?
2. What is the memory-access to floating-point computation ratio in each thread? Consider a multiply and addition as separate operations, and ignore the storing of the result. Only global memory loads should be counted towards your off-chip bandwidth.
3. If you use the visual profiler, also report uncoaleased reads/writes, occupancy and cache misses.

4. GPU-based matrix multiplication using shared memory

The former code only used global memory. Now develop the kernel code that makes use of shared memory via the approach discussed in class that uses tiles also for the input matrices. Answer the questions listed in part 2 for this configuration. Again, report the timings, differences, and answer the questions above.

5. Report results on comparative charts

Take all the results you obtained and compile a chart/plot that demonstrates the performance measured in the programs developed in the 4 sections above. Plots speed over matrix size for each of the 4 variations in common plot so they can be easily compared. You can use Excel for this. In addition, create plots for speedup = time_CPU / time_GPU for each of the GPU versions (in a single chart for ease of comparison).