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++ using NVIDIA’s CUDA parallel computing architecture. This version will harness the power of the GPU a general purpose parallel processor.

My specific goals of the past quarter were:
- Read about CUDA and its architecture, CUDA C/C++ compiler and familiarize myself with the CUDA Toolkit which has all the development tools, libraries, environment configuration and documentation.
- Implement a number of prototypes which might help to define an outline for methods/functions to be used in the CUDA version of the MASS library.

Work Completed
During the first portion of the quarter I have read provided material by the professor as well as became a member of the CUDA community. Besides assigned reading material I’ve been extensively reading CUDA C Best Practice Guide and CUDA C Programming Guide which are part of the CUDA SDK. After reading material as well as watching CUDA Courses Online I have started working on a number of prototypes. Although now the basics seem trivial, initially I had problems compiling and executing simple programs. However, as the quarter went on I became more familiar with the library and the CUDA programming. Below is the list of work completed and hopefully it’s a great start for anyone who had not been exposed to CUDA programming and wants to learn CUDA programming. As in any programming language, my first program was to create a very simple “Hello, World” version in CUDA. However, before you can start doing this you need to configure and install the following:

How to check if your computer is CUDA enabled
- Open Run Dialog (Win + R) and enter dxdiag command
- Once DirectX program is opened, select Display tab which will tell you about the graphics card in your system.
- Next go to NVIDIA’s list of CUDA-enabled products and search for your graphics card
- In earlier versions of CUDA, it was possible to compile code without having CUDA-enabled environment available, however, this feature was deprecated as now there are more readily available tools to debug and profile CUDA code using tools such as
  - On windows you can download Parallel NSight for Visual Studio integration. This download is free of charge.
  - On linux you can use cuda-gdb and cuda-memcheck

What to install
- NVIDIA CUDA Toolkit v4.0 or latest (32 or 64 bit) depending on your machine
CSS 600: Independent Study Contract - Final Report

Student: Piotr Warczak
Quarter: Summer 2011
Student ID: 9936333
Credit: 3
Grading: Decimal

- Contains C/C++ compiler, Visual Profiler and other libraries, tools, and documentation
  - NVIDIA SDK 4.0
    - Includes all code samples
  - Developer Drivers (based on operating system)
    - Windows for WinXP, WinVista, Win7
    - Notes: Linux and Mac OS also available (more info available in the link below)
  - Visual Studio 2008 or 2010


**How to compile programs in execution mode (with CUDA-enabled system)**
Currently, I implement code on my laptop. Then I copy the files over to dslab server. Once the files are there all you have to do is use the CUDA compiler called nvcc to compile necessary files.

```bash
nvcc myHelloWorld.cu --o myHelloWorld
```

If you don’t specify --o output a.exe will be created on Windows or a.out in Linux

---

**KERNELS**
Kernels in CUDA are essentially extension of C functions which execute code in parallel by N different CUDA threads as opposed regular C functions, which execute just once when called.
To define a kernel function one must prefix the function signature with \texttt{__global__} keyword and the number of threads to execute that kernel using the following notation \texttt{<<<blocksPerGrid, threadsPerBlock,>>>} called execution configuration syntax. Each thread can be easily identified by using the built-in variable \texttt{threadIdx}.

Example:

```c
// kernel definition
__global__ void Add(int* a, int b, int* c)
{
    int tID = threadIdx.x;
    if (tID < N)
    {
        c[tID] = a[tID] + b[tID];
    }
}

int main()
{
    // this is how to call kernel function from the host
    Add<<<1, N>>>(a,b,c);
}
```

Threads
The built-in threadIdx is a 3-component vector so that threads can be easily identified by one, two or three dimensional thread index
- threadIdx.x
- threadIdx.y
- threadIdx.z

\textbf{How thread ID can be identified by the index of a thread}

One dimensional block
It’s quite simple because they are the same. Let’s say we have an array with 10 elements and we execute kernel using this syntax functionName\texttt{<<<1,10>>>(var1, var2, var3)}. Each element var1[0], var1[1], var1[2]..., is going to be accessed by a separate thread. If we look at the example from above we can notice that function Add will be called N (=10) times and each time the function call is executed by separate thread.

Two dimensional block
If the size of the block is \(Dx, Dy\), then the thread ID of a thread index \((x, y)\) is \(x + y \times Dx\)

Three dimensional block
If the size of the block is \(Dx, Dy\), then the thread ID of a thread index \((x, y, z)\) is \((x + y \times Dx + z \times Dx \times Dy)\)

Example (Kernel invocation with single block of \(N \times N \times 1\) threads) :
// Kernel definition
__global__ void Add(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    int blocksPerGrid = 1;
    dim3 threadsPerBlock(N, N);
    Add<<<blocksPerGrid, threadsPerBlock>>>(A, B, C);
}

In today’s GPUs the limit of threads per block is about 1024 or less depending on the NVidia card. This is due to the fact that all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core.

However, each kernel can be executed by multiple equally shaped thread blocks, so that the total number of threads is:

\[
\text{Total thread count} = \text{Threads per Block} \times \text{Number of blocks (thread blocks)}
\]

As in threads, thread blocks can also be one, two or even three dimensional grid of thread blocks. The number of thread blocks in a grid is usually dictated by the size of the data to process or number of processors in the system.

The number of threads per block as well as thread blocks per grid can defined in the execution configuration by int or dim3 data type. As in threads, each thread block can be identified by built-in variable called blockIdx and the size of the block can be identified by also a built-in variable called blockDim.

Example with N = 32 (Kernel invocation with multiple blocks 16 * 16 * 4 = 1024):
// Kernel definition
__global__ void Add(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    //Kernel invocation
    dim3 threadsPerBlock(16, 16);
dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
Add<<<numBlocks, threadsPerBlock>>>(A, B, C);

In this example 256 threads per block * 4 blocks, which gives us total of 1024 threads.

Here is the illustration how threads per block and thread blocks per grid are maintained. This picture is available in the Cuda C Programming Guide, which is available after installing CUDA SDK.

In terms of MASS library, to process data in the shortest possible time, we need to determine the size of the data to processed and the limitations of GPU card or multiple cards. This way we can correctly
calculate how we need to divide the data so each block executes optimal number of threads and thread blocks per grid.

**NVCC**

The NVIDIA nvcc compiler driver converts .cu files into C for the host system and CUDA assembly or binary instructions for the device. It supports a number of switches, of which the following is especially useful for optimization:

- `arch=sm_13` or higher is required for double precision.

**Wave2D**

After creating a few of these prototypes, I have decided to convert Wave2D written in Java to CUDA/C++. The first step was to convert java version of the program to stripped version in C++ (without graphics). By doing this, I would be able to compare the data between my basic version and CUDA version of Wave2D in C++. As of now, the Wave2D converted project only compiles with Visual Studio and not the g++ compiler.

I’ve also started to convert Wave2D program to CUDA, but will continue this work during the Fall quarter of 2011.

All code snippets and examples are available at the following directory on hydra machine:

`~dslab/SensorGrid/MASS/GPU/examples/`