Lab 1

Part 1: Introduction to CUDA

- Code tarball: lab1.tgz

In this hands-on lab, you will learn to use CUDA to program a GPU. The lab can be conducted on the SSSU Fermi Blade (M2050) or NCSA Forge using NVIDA M2070 GPUs.

To prevent overloading the cluster, please work in teams of two or three.

This assignment has to be completed individually. (Never show or share your code with others, please follow this strictly. If someone needs help, you can always ask your peers or me for oral explanation.)

What to turn in and when

This lab has a part due at the end of class as well as a part that will be due next (Tue June 12th). Strictly avoid late submissions or requests for extentions.

Part 1: Running a CUDA Program on Forge

The Forge has 44 nodes, 32 of which connected to 6 Fermi processors and the other 12 nodes connected to 8 Fermi processors. In total, it has 288 accelerator units.

To request a interactive job

```
qsub -I -l nodes=1
```

Instead of compiling with `gcc` or `icc`, CUDA programs are compiled with `nvcc` (the NVIDIA C Compiler). Also, CUDA source code files typically use the `.cu` file extension instead of the `.c` extension. Other than those minor differences, building and running CUDA applications are similar to normal C applications.

```
nvcc vecAdd.cu timer.c -o vecAdd
./vecAdd
```

The file `vecAdd.cu` contains the Vector Addition program, which uses a very simple kernel to compute $C_i = A_i + B_i$. Try compiling and running the program. Remember, you can compile and even run it on the head node of Forge, but you will need to submit a job or open an interactive node to run it on a GPU node when required to have a dedicated access for accurate timings.

Start by answering these questions about this example program.

1. What is the execution time for transferring the data (A and B) from the host (CPU) to the device (GPU)?

2. What is the execution time for transferring the result (C) from the device (GPU) to the host (CPU)?

3. What is the execution time of the kernel?

4. Using the timing measurements, compute the effective performance (GFLOP/s) and the effective memory bandwidth (GB/s) for this kernel.

5. Recall that the GPU peak performance is 1030 GFLOP/s (single precision) and that its peak memory bandwidth is 144 GB/s. What fraction of peak does this kernel achieve?

6. Try increasing the size of the vectors (the variable `N`), recompile, and then run the program again. How do the effective performance and bandwidth numbers look for various sizes of `N`?
   ```
   [ ./vecAdd 2 > & tmp     ---  to redirect even stderr output to tmp.
      cat tmp | grep CPU | awk '{print %6}'     ---  to get timing for CPU run.
    cat tmp | grep "host to device" | awk '{print $9}'
                                          --- to the host to device data transfer time.
   ```

   Make N as a command line argument to run the code with different input sizes.

   ```
   for N in 1024 4096 8192
   do
   ./vecAdd $N  2 >& tmp
   #process tmp using grep and awk for the required values.
   # you can then use GNUPLOT to generate a plot.
   done
   ```

   Shell command "paste tmp1 tmp2 > tmp" to have the values in tmp1 and tmp2 copied row wise in tmp.

7. The program at the end prints out the number of 'errors'. Currently this number is not 0 because we have no CPU program results to compare against. Complete the function 'compute_vec_add' so that the number of errors become 0. Compare the performance of the CPU program against that of the GPU kernel.

Note: `CUDA_CHECK_ERROR` is a macro for a function that checks to see if a CUDA kernel threw any errors. Note: The file `timer.c` contains timing related functions. It is a useful tool for measuring the execution time of your functions. It is not required that you know how it works but you may find it useful if you were to study it.

Part 2 : Matrix Addition

Understanding Thread IDs

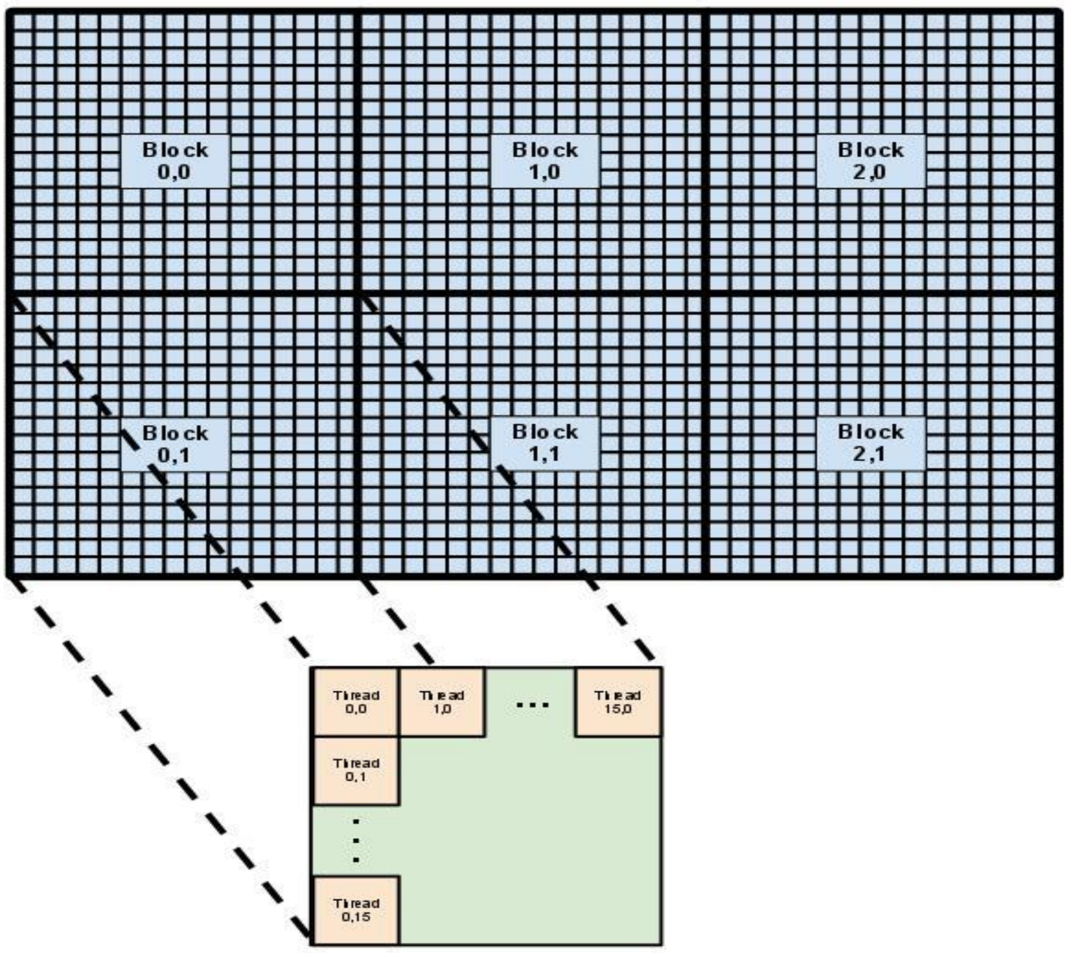
The vector add example used three pre-defined variables, `blockIdx.x`, `blockDim.x`, and `threadIdx.x`, in order to calculate the array index of each thread.

```
/* Calculate array index for this thread */
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

In this exercise, we will implement a GPU kernel that adds two *matrices* together. This exercise accesses elements in a 2-D matrix instead of an array. To assist us, we will take advantage of CUDA's 2-D thread layouts. For example, consider a kernel invocation of some GPU kernel function, `myKernel`:

```
dim3 blocks(3,2);
dim3 threads(16,16);
myKernel<<<blocks,threads>>(A,B);
```

The diagram below illustrates the organization of threads. (The figure is borrowed from "CUDA By Example" by Sanders and Kandrot).



In a 2D thread layout, `blockDim.y` and `threadIdx.y` are used to calculate indices.

```
/* Calculate array index for this thread */
int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;
```

Now, compile and run the program.

```
nvcc matAdd.cu timer.c -o matAdd
./matAdd
```

The program will print out information regarding the execution times and errors similar to those of the vecAdd kernel. However, in this case, the GPU kernel has not been implemented. Assume that the matrix is stored in row-major format.

8. Look at the following lines of code that can be found in matAdd.cu

```
dim3 GS (N/16, N/16, 1);
dim3 BS (16, 16, 1);
```

Determine how many threads are in a single thread block and describe how the work is being distributed among the thread blocks.

9. Complete the function matAdd. You'll know whether you have implemented it correctly when the program prints 'Success'.

Part 3: Shared Memory and Coalesced Reads

Matrix transpose performs the following operation: $B[j][i] = A[i][j]$.

```
__global__ void matTranspose_naive(float *A, float *B) {

   /* Calculate global index for this thread */
   int i = blockIdx.y * blockDim.y + threadIdx.y;
   int j = blockIdx.x * blockDim.x + threadIdx.x;

   /* Copy A[j][i] to B[i][j] */
   B[j * N + i] = A[i * N + j];
}
```

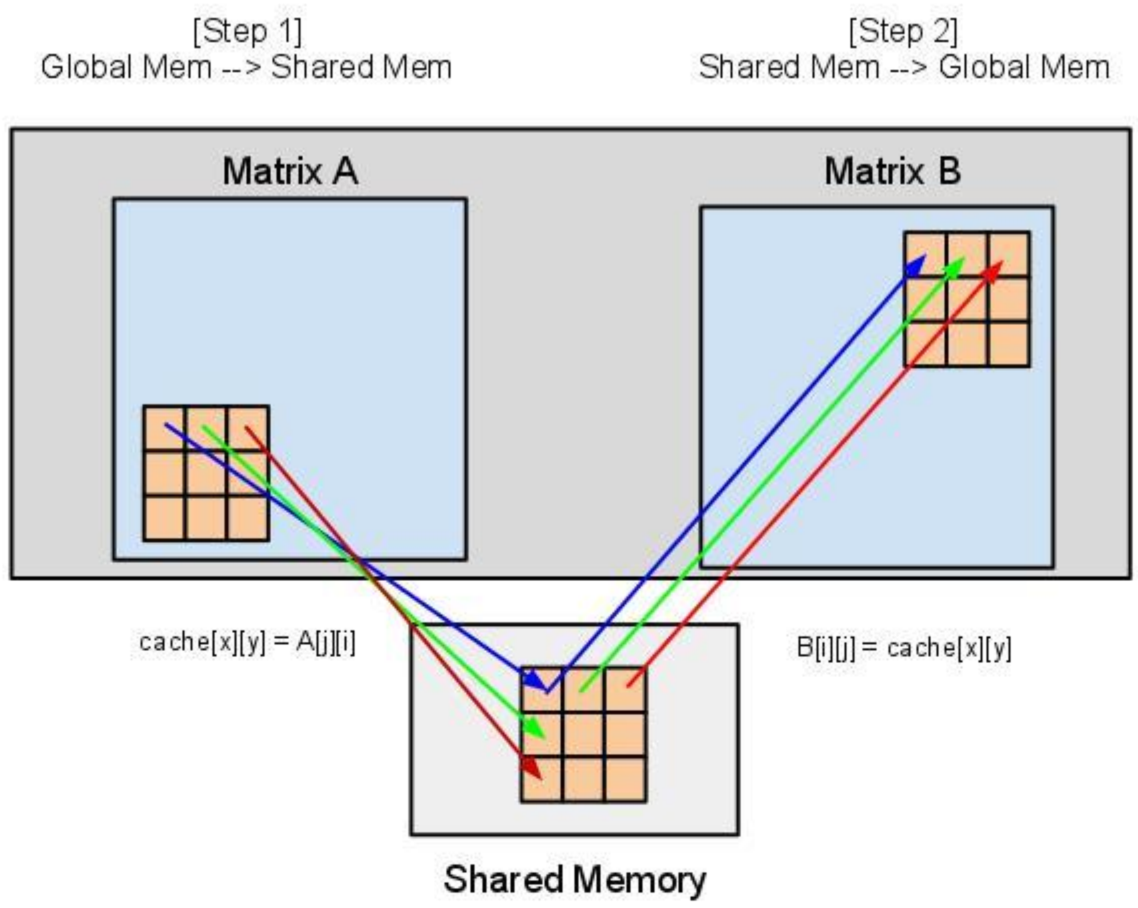
Note: Again, assume all matrices have row-major ordering

Unfortunately, this kernel performs very poorly due to $uncoalesced$ writes into the main memory. Whenever a thread requests a read (or a write) into the main memory, the GPU fetches (or writes) 128 consecutive bytes at once. Therefore, in order to fully utilize the GPU's bandwidth, a warp of threads should utilize all 128 bytes of data, and when this happens, we say that that there is a coalesced access to the memory.

Using Shared Memory

In this section, you will learn to use Shared Memory to improve the performance of a Matrix Transpose kernel. Shared Memory is a type of scratchpad memory that can be used to share data amongst threads in a thread block.

In the `matTranspose_naive` kernel, there are coalesced reads from matrix A. However, when each thread is writing its data into the transposed position in matrix B, data access is not coalesced.
In order to prevent this from happening, we need to swap data between threads such that consecutive threads write into consecutive positions in matrix B. This can be done using $shared\ memory$. The diagram below shows how this can be done.



Code snippet below shows the use of Shared Memory for matrix transpose.

```
__global__
void matTranspose_sm(float* B, float* A)  {
        float val;
        __shared__ float cache[BLOCK_SIZE][BLOCK_SIZE];

        /* get global index for this thread */
        int j = blockIdx.x * blockDim.x + threadIdx.x;
        int i = blockIdx.y * blockDim.y + threadIdx.y;


        /* load data into shared memory so that it is transposed */
        cache[FIX_ME][FIX_ME] = A[i * N + j];
        __syncthreads();

        /* fetch the right data back from the shared memory */
        val = cache[FIX_ME][FIX_ME];

        /* Write the data back to main memory */
        i = blockIdx.x * blockDim.y + threadIdx.y;
        j = blockIdx.y * blockDim.x + threadIdx.x;
        B[i * N + j] = val;
}
```

Note that after writing in to the shared memory, the threads need to synchronize in order to make sure that all threads have had a chance to write in to the shared memory before any thread reads data from it.

10. Complete the `matTranspose_sm` kernel and compare the performance between the two different implementations.