CS 6023 GPU Programming
Assignment 2

Handout date: 28/02/2019
Due date: 10/03/2019 (end of day)

General instructions:

1. The goal of this assignment is to continue your understanding of CUDA C with application to matrix multiplication. You will start to use shared memory, particularly for the tiling pattern.
2. This is an individual assignment. Collaborations and discussions with others are not encouraged. However, if you do collaborate with anyone, you are requested to explicitly state the same in your submitted report.
3. You have to use C with CUDA for your implementation.
4. You have to turn in three components: (a) well documented code (.cu files for each programming question), (b) output files (.txt for each relevant programming question), and (c) a detailed report (single .pdf file) of the results of the experiment. All these are to be submitted on Moodle. No hard copy is required.
5. Code will be checked for functional correctness, quality of code, and performance. Report will be verified for completeness and to-the-point answers to the subjective questions posed.
6. If you have doubts please raise them in the Moodle Q&A forum, and follow the same regularly.
7. Codes generating incorrect output will be awarded zero marks.
8. We provide you with boilerplate code for populating the input matrices, and for printing the output matrix to a file. Please include these code snippets as it is in your source code. Modifying the code snippets may result in an incorrect output and lead to consequences mentioned in 7.

Submission instructions:

1. Make a report PDF as *<username>_report.pdf*, where <username> is the username used by you to login to the GPU cluster. For many it may be your roll number.
2. Put all files inside a folder named your username and compress the folder into a tarball in the following format : *username_A02.tar.gz*
   [example directory structure here]
3. Upload the tarball in Moodle. For instructions on how to do this contact the TAs.

Boilerplate code:

1. Use the following function to populate a matrix:

```
void fill_matrix(double *mat, unsigned numRows, unsigned numCols)
{
    for(unsigned i=0; i < numRows; i++)
       for(unsigned j=0; j < numCols; j++)
       {
          mat[i*numCols + j] = i*2.1f + j*3.2f;
       }
}
```

2. Use the following function for printing the output:

```
void print_matrix_to_file(double *mat, unsigned numRows, unsigned numCols)
{
  const char *fname = "assignment2_out";
  FILE *f = fopen(fname, "w");

  for(unsigned i=0; i < numRows; i++)
  {
     for(unsigned j=0; j < numCols; j++)
     fprintf(f,"%4.4f ", mat[i*numCols + j]);
     fprintf(f,"\n");
  }
  fclose(f);
}
```

Include the following header file: `#include <stdio.h>`

Questions:

Q1. (**1 point**) Write CUDA code for performing matrix multiplication with 2D block and 2D thread configuration. Populate each input matrix with numbers of type "`double`", on the host (CPU), using the function "`fill_matrix()`" and perform matrix multiplication on the GPU. Assume that one thread is assigned to output one element of the output matrix. Assume the input and the output matrices are of dimension *N x N* each with N = 8192.

Use blocks having the following dimensions:

```
dim3 threads(16,16);
```

Implement the above code for the following two variations:

(a) Keep the fastest varying index as '.x' inside the kernel. Please see the below pseudo code for example.

```
kernel1(...) {
    row = threadIdx.x;
    col = threadIdx.y;
    A[row][col] = …;
}
```

(b) Keep the fastest varying index as '.y' inside the kernel. Please see the below pseudo code for example.

```
kernel2(...) {
  row = threadIdx.y;
  col = threadIdx.x;
  A[row][col] = …;
}
```

Compare the kernel execution times in the two cases (a) and (b) and report the factor of speedup or slow down observed in (b) wrt (a). Also, reason about the performance difference between (a) and (b).

Submit the source code as <username>_1.cu
Include the comparison between (a) and (b) and your comments in the report titled <username>_report.pdf

~~~

Q2. (**2 points**) Vary the block dimensions and thread dimensions in the code for Q1 and observe its effect on the runtime of the kernel. Plot the runtime values for different block and thread dimension pairs (test powers of two) (data points). You should plot for at least eight data points.

Submit the source code as <username>_2.cu
Include the plot in the report titled <username>_report.pdf

~~~

Q3. (**3 points**) Modify the code in Q1 to use shared memory (preloading from global memory to shared memory and then reading from the shared memory to perform the matrix multiplication operation) without using tiling.

Shared-memory usage requirements: A block of threads should read into the shared memory the rows and columns required for computing the entries of the output matrix that the block is assigned to. Threads of a block should read in the required data to the shared memory in a cooperative manner and each thread of a block must read in distinct element(s).

Test your implementation for the following two cases:

(a) Both input matrices having dimensions 32 x 32. (Note use the same `fill_matrix` command to initialize these matrices)

Use the following thread block configuration:

```
blockDim.x = 16
blockDim.y = 16
```

(b) Input matrices used in Q1.
Use the following thread block configuration:

```
blockDim.x = 16
blockDim.y = 16
```

Please report if your code fails for any of the above inputs. If so, reason about it.

Submit the source code as <username>_3.cu
Include your comments in the report titled <username>_report.pdf

~~~

Q4. (**1 point**) Modify the code in Q3 to use tiling, as discussed in the lecture, where the tile size is equal to the block size. Compare the runtime of this implementation with the implementation of Q1 and reason about the difference in the performance you observed.

Submit the source code as <username>_4.cu
Include your comments in the report titled <username>_report.pdf

~~~

Q5. (**1 points**) Vary the tile sizes for the code in Q4 and observe the runtime. Plot the runtime for different tile sizes (listed below) and report the optimal tile size. Keep the block size equal to the tile size.

i) 4 x 4
ii) 8 x 8
iii) 16 x 16
iv) 32 x 32

Submit the source code as <username>_5.cu
Include the plot in the report titled <username>_report.pdf

Q6. (**2 points**) Modify your code in Q1 such that it works for rectangular matrices as well, i.e., for the case when M, N, K are not all equal. Test your code on the input matrices having sizes 4096 x 8192 and 8192 x 16384. For both cases use the "`fill_matrix()`" function to initialize the matrices.

Submit the source code as <username>_6.cu
Include the runtime of the kernel in the report titled <username>_report.pdf

~~~

Q7. (**3 points**) The '3 idiots' are making mischief once again and the administration would like to carry out some disciplinary action. However, the identity of the 3 idiots is not known. But they know that the 3 idiots are friends with each other on Facebook. Your task is to help the disciplinary committee by finding out the total number of such possible groups of three, given the social network graph of Facebook. Use the matrix multiplication functions above to compute this. (HINT: If *A* is the adjacency matrix of a graph *G*, then *A^3*[*i, j*] gives the number of walks of length 3 from *i* to *j*.)

Input format:
The first line contains an integer: the total number of vertices N
The second contains an integer: the total number of edges M
The next 'M' lines contain two space separated integers each: u v, which denotes that u and v are friends

Constraints:
N <= 3008
1 <= u, v <= N

Output format:
Print a single line containing one integer (the answer)

Submit the source code as <username>_7.cu

You can use the template given below:

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

const int MAXSIZE = 3008;


// Computes the product of two square matrices m1, m2 and stores in result
__global__ void matrix_multiplication(int *m1, int *m2, int *result, int N) {
    // Your code here
}
```
~~~

```
int main() {

    // Variable for adjacency matrix
    int *h_adj;

    int memsize = MAXSIZE * MAXSIZE * sizeof(int);

    // Allocating host memory
    h_adj = (int *)malloc(memsize);

    // Initiliasing adjacency matrix with zeroes
    memset(h_adj, memsize, 0);

    // Taking input graph
    int N, M;
    scanf("%d\n", &N);
    scanf("%d\n", &M);
    for (int i = 0; i < M; i++) {
       int u, v;
       scanf("%d %d\n", &u, &v);
       h_adj[(u-1)*N + v-1] = h_adj[(v-1)*N + u-1] = 1;
    }


    // Your code here
    // Use the kernel to do matrix multiplications on the GPU
    // Copy the result back to host to compute the answer on the CPU

    int answer = 0;    // Set the correct value of answer
    printf("%d\n", answer);

    return 0;
}
```

~~~
~~~