THE UNIVERSITY OF SYDNEY

COMP2129 Assignment 5

Due: 11:59pm Friday 3rd June

Task

This task will improve your parallel programming skills by implementing an algorithm on a GPGPU device using CUDA. This task will also help to improve performance engineering skills as the run-time performance of your program will need to be tuned in order to obtain top marks.

Your task is to implement the computation of the power of a n-by-n square matrix A. The power of a matrix is defined as:

$$\begin{aligned} A^0 &= I \\ A^k &= \prod_{i=1}^{k} A \quad k > 0 \end{aligned} \tag{1}$$

or alternatively as a recurrence relation:

$$\begin{aligned} A^0 &= I \\ A^k &= A^{k-1}A \end{aligned}$$

The computation is to multiply the n-by-n matrix k-times, each time multiplying it to the previous matrix, starting with the identity matrix. Use the naïve method given in Equation 1 to compute the power of the matrix.

Calculating the result C of matrix multiplication A times B is a simple process. The i, j'th entry in the resultant matrix is defined as follows:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j} \tag{2}$$

for all $1 \leq i, j \leq n$. All of the matrices we are dealing with in this task are square n-by-n matrices. A more thorough explanation of matrix multiplication can be found on the Wikipedia page[1] .

The identity matrix I of size n is defined as follows:

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \tag{3}$$

It is a n-by-n matrix of zeros, with the diagonal consisting of ones.

[1] http://en.wikipedia.org/wiki/Matrix_multiplication

Technical Details

Your program should be implemented in CUDA, in which you should write a kernel function to aid you in the computation of the kth power of the matrix A.

CUDA kernel functions do not support the **`double`** data type. As such, your program should store all floating point numbers using the **`float`** data type.

The **`Makefile`** provided will produce two binaries named **`power`**; they will be located in **`build/power`** and **`debug/power`**. The binary in the **`build`** folder needs a GPGPU to be executed and runs on the actual hardware. The binary in the **`debug`** folder is built for the emulator and does not require a GPGPU to be run.

Implementation

We strongly suggest that you implement first the most naïve version of this algorithm before making any optimisation attempts. The most naïve version may be implemented as given below:

- Allocate GPGPU memory for matrices used to calculate the power.
- Copy A^0 and A from RAM to GPGPU memory.
- Set up at least n^2 threads in a grid. Make sure that you choose the block size appropriately. The GPGPU target will be the graphics card used in the labs, i.e., GT210. Each thread is responsible for calculating the i, j'th entry of the resultant matrix for the current iteration.
- Each thread computes the inner project for the i, j'th entry.
- Between each invocation of the kernel, call **`cudaThreadSynchronize()`** to make the C program wait for the kernel function to finish executing.
- After k iterations, copy the resultant matrix back to RAM from GPGPU memory.
- Free your allocated GPGPU matrices.
- Print result.

To ensure that kernels are executed in sequential order, use the following code template which executes **`cudaThreadSynchronize`** between two kernel invocations.

```
...
// invoke the kernel K times to compute the K'th power
for (k = 0; k != K; k++) {
  ...
  kernel<<<grid_dim, block_dim>>>(...); // invoke the kernel
  cudaThreadSynchronize(); // block here until the kernel terminates
  ...
}
// copy data from GPU memory to RAM
...
```

Only if you have an initial version in working order, try to improve the performance of your code . You can obtain speedups by technical and/or algorithmic improvements.

Input/Output

The structure of the input is as follows:

```
<n>
<power>
<element>
...
```

The first line specifies the size of the square matrix, i.e., the number of rows and columns (n). The next line specifies the power to which the matrix needs to be computed (k). After the second line, the elements of the square matrix are defined. The elements are listed row-by-row. Both n and k are guaranteed to be non-negative. It is erroneous for n to be zero.

For example, for the matrix

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

and a power of $k = 3$, we have following input:

```
2
3
1.0
2.0
3.0
4.0
```

For scanning in the elements, you should use the format string **`"%f\n"`**.

Before your program terminates, it should print out the values of the entries of the resultant matrix A^k. For our A^3 example above, the output should be:

```
37.0000 54.0000
81.0000 118.0000
```

For printing the elements, use the format string **`"%.3f"`**. If there is an error in the input, print the string `error` and exit the program.

Writing your own test cases

Since your assignment will be automatically marked, it is crucial that you follow our instructions carefully. Your output will need to be in exactly the right format. To assist with this, we have made available some sample test cases. If you are logged in to the **`ucpu[01]`** machines, you can run these tests using the **`Makefile`** we provided by typing **`make test`**. If your output is incorrect, you will see both the expected output and your output. If you passed the test, then you will only see the name of the test.

The sample test cases are by no means exhaustive. You will need to test your code more thoroughly by thinking carefully about the specifications and writing your own tests.

As with the previous assignment, you can view the sample test cases manually. They are located in the folder **`~comp2129/assignment5/sample/question1`**.

Remember: If you run your code on the **`ucpu[01]`** machines, you need to run the **`debug/power`** binary, which will run your code in the *emulator*. The emulator cannot be used for performance testing as the kernel is not executed on a GPGPU. To performance test your code you will need to execute it on a machine with a GPGPU, such as the Carslaw lab machines. When you are on a machine with a GPGPU, you need to run the **`build/power`** binary. This binary is built for the hardware instead of the emulator.