Assignment 3

Objectives

In this assignment, we will explore CUDA programming by optimizing the Reducing Matrix Multiplication (RMM) algorithm on a GPU.

Assignment

We will consider the same RMM algorithm explored in Assignments 1 and 2. Your task is to write an optimized parallel program in CUDA that performs the RMM algorithm on a GPU. The same specifications and assumptions on the RMM algorithm as before apply to this assignment too. Specifically:

1. Input: Matrix A of dimension M * N and Matrix B of dimension N * K
2. Output: Matrix C of dimension (M/2) * (K/2)
3. You may assume M, N, and K are even numbers

You will use CUDA to write an optimized RMM kernel targeting the GPUs on the SCITAS GPU cluster Izar. Borrow ideas from A1 and A2 and adapt them to the GPU architecture. Your program must operate as follows:

1. Read the values of M, N, and K from the command line arguments
2. Generate and initialize matrices A and B with random values
3. Compute matrix C as the RMM of matrices A and B on the GPU
4. Write the computed matrix C into a CSV file matC.csv and exit

Step 3 is the main operation and will be timed. Note that step 3 includes any operations required to copy data between the GPU and CPU.

GPUs contain thousands of cores to accelerate graphics applications as well as scientific computation. To get the best out of the massive parallelism available in GPUs, you need to consider the characteristics of the underlying hardware architecture. We will be using the SCITAS GPU cluster Izar for this assignment. You can find the specifications of the machines in the Izar cluster here.

You can access the Izar cluster similarly as the Helvetios cluster using the command `ssh <username>@izar.hpc.epfl.ch`. It also has the same directory structure as the Helvetios cluster. You can run your Slurm scripts in this cluster in the same way as the previous assignments, but please make sure to use the template script provided with this assignment as it contains extra arguments necessary for using the GPU machines.

Development Infrastructure

We provide a file A3.zip which contains the following files:

1. `rmm.cu` – contains two functions with the same inputs and outputs: `rmm_cpu` and `rmm_gpu`. In both functions, the first three arguments are pointers to matA, matB and matC respectively. The next three arguments are M, N and K respectively.
    a. `rmm_cpu` – contains the unoptimized single-threaded CPU implementation. Use this function as your baseline and reference implementation.
    b. `rmm_gpu` – function that you need to complete. The function must copy the input matrices to the GPU, compute the RMM on the GPU and then copy the result matrix back to the CPU. The result matrix generated by this function must exactly match the result matrix generated by `rmm_cpu`.
2. `assignment3.cu` – contains the `main` function which performs steps 1, 2 and 4 and calls `rmm_cpu` or `rmm_gpu` for computing the RMM as needed.
3. `utility.h` – contains a set of helper functions.
4. `Makefile` – to compile the code using `nvcc` (Nvidia's CUDA compiler).
5. `execute.sh` – a sample script to submit jobs to the Izar cluster.

Note: In contrast to A1 and A2 where the matrices were stored as 2D-arrays, the matrices matA, matB and matC are stored in row-major form as 1D-arrays for A3.

The execution environment will operate as follows:

1. `rmm.cu` and `assignment3.cu` must be present in the same directory as the `Makefile` and `utility.h`. Don't change any file names and function signatures.
2. You can compile the program using the `Makefile` with the command `make all`. The compilation generates a binary executable file `assignment3`.
3. The program takes four arguments – the values of M, N, and K respectively in that order, followed by the debug flag. Set the debug flag to 1 to print matrices into the terminal.
4. On finishing execution, the program must save the result matrix into the file matC.csv and display the execution time in the following format (execution times in this example are non-representative):

```
# /bin/bash
$ ./assignment3 1024 1024 1024 0
Host to Device MemCpy takes 15s
RMM operation takes 70s
Device to Host MemCpy takes 15s
Total time taken: 100s

$ ls
matC.csv ...
```

Please note that it might be more practical for you to install CUDA on your machine (if it has an NVIDIA GPU) and start developing your code there before moving to the cluster to avoid the cluster's queueing time. **Adhering to the execution environment will guarantee that your program will comply with all specifications during grading.**

Deliverables

You need to turn in the following things for this assignment:

1. `rmm.cu` containing a completed, parallelized and optimized version of `rmm_gpu`
2. A report that answers the questions listed below. The report name should be a3_GROUPID.pdf

Remember to check if your code complies to the format expected by the automated tester; otherwise, you will receive no points for correctness!

To submit your code and your report, create a tarball archive (**this is the only accepted format!**) called `a3_GROUPID.tgz` and upload it on Moodle[1].

Report

In the report, we expect you to answer the following questions:

1. Explain how you implemented and optimized `rmm_gpu`.
    a. Identify the different ways of splitting the work among threads and thread blocks, and discuss their advantages and disadvantages.
    b. List all optimizations you applied and the reasons why you chose to apply them.
2. Report how much each optimization improves performance and explain the result obtained. Design and run experiments that isolate the effect of each optimization/thread organization you tried out and then report the results.
3. For each of the following <M, N, K> combinations {<16, 16, 16>, <64, 64, 64>, <256, 256, 256>, <1024, 1024, 1024>, <4096, 4096, 4096>}:
    a. In a table, present the execution times you measured for running the baseline CPU version and your optimized GPU version.
    b. Present graphically the breakdown of the GPU runtime for copying data from host to device, computation and copying data from device to host.
4. Analyze and explain your results from the previous question.

The report should be as precise as possible, addressing the four questions above. Keep the descriptions of the algorithm implemented and of your reasoning in parallelizing the program short. A regular report should not be much longer than 4 pages.

[1] If you don't know how, look online for a guide like this one: https://www.howtogeek.com/248780/how-to-compress-and-extract-files-using-the-tar-command-on-linux/

Grading

The code will be graded automatically by a script, and checked for plagiarism. The script will check the running time of your program for different configurations and if the results returned are consistent with what was expected. Plagiarized code will receive 0. The reports will be read and graded by humans. The grade breakdown is as follows:

• Correctness:	50%
• Report:	50%