University of Amsterdam
BSc Informatica

Distributed and Parallel Programming

Assignment 2: CUDA

GPU Programming in CUDA

dr. Robert Belleman, dr. Ana-Lucia Varbanescu

November 2024

Introduction

In this assignment, we focus on GPU programming using CUDA.

Please revisit the DAS5 instructions to make sure you can run the code on the GPUs on the DAS5. We recommend using the clusters in the VU cluster (`fs0.das5.cs.vu.nl`).

Note that we recommend you to use the TitanX GPUs on DAS5, but you can choose to use other/ more cards (such as your own) for your experiments. If you choose to do that, please make sure to adjust the compute capability in the Makefile. In any case: make sure you clearly state in your report which card(s) and flag(s) you have used for your experiments.

Assignment 2.1: Wave equation simulation with CUDA

Reconsider the 1-dimensional wave equation studied in the pthreads and OpenMP assignments. Please design a parallel version of the application suitable for the GPU. Furthermore, implement this application in CUDA, and plug that in the provided framework. Make sure you document both the design and the implementation of your solution.

You will start programming in `simulate.cu`, which already handles initialization and writing to file. Please note that no CUDA code is currently included in this file. You can use the code in `vector-add.cu` for inspiration, such that you don't have to start from scratch. Do not change the code already present in the framework unless absolutely necessary; should such changes be required, please explain all such changes and the reasons behind them in the report.

Next, please empirically evaluate the performance of your solution. We recommend the following experiments:

1. **Scalability** Run experiments with different problem sizes - we recommend using at least 10^3, 10^4, 10^5, 10^6, 10^7 amplitude points, and measure the execution time. Make sure you use a number of time steps that allows you to compare the performance against the previous versions

of the wave equation. For this first evaluation, please fix the number of threads per block to 512. Report your results, and compare them against the performance your implementations with pthreads and OpenMP.

2. **Kernel configuration impact** Experiment with different thread block sizes, using 32, 64, 128, 256, 512 and 1024 threads per block (on the same card). Report your results and explain why some sizes perform better than others.

3. **Accuracy** Compare the accuracy of the results generated by your CUDA implementation with those generated by your sequential implementation. Report your findings and explain the expected behavior, as well as any unexpected results.

Finally, please comment on any optimizations (i.e., ways to change your design/implementation) that could lead to performance improvement. Please discuss and/or apply (some of) them; pseudocode-based solutions are recommended, implementing such optimizations in the code, and evaluating the performance gain/loss is highly appreciated. If none of the optimizations listed on the slides would pay off, please explain why these are unlikely to provide any improvements.

Assignment 2.2: Parallel cryptography in CUDA

There are many cryptography algorithms that can be accelerated using parallel processing. The simplest one of them is Caesar's code. In this symmetric encryption/decryption algorithm, one needs to set a numerical key (1 number, typically between 1 and 255) that will be added to every character in the text to be encoded. For example, `Caesar('ABCDE', 1) = 'BCDEF'`.

In this assignment you are requested to build a parallel encryption/decryption application for a given text file with several features. The starting code for this example is already provided to you in the framework that is supplied with this assignment; refer to the README to see how to use it.

Specifically, please design, implement, and evaluate the algorithms (and their parallel implementations), and for the following features:

1. **Basic Caesar's cipher** Design and implement the sequential encryption and decryption, as well as the CUDA kernels for the basic Caesar's code - filling in the already defined functions and kernels. Please specify how you parallelized the algorithm for the GPU, and what are the implementation decisions you made (if any). The code must work correctly on any text input file.

   Input files Please test your code on at least 5 different files of different sizes from small (a couple of kilobytes) to very large (many megabytes). You can use any text file as an input for the algorithm, and you can get very large text files online. Note that the file names are **hardcoded**: `original.data` is the file to be encrypted, while `sequential.data` will be the reference result for the CPU encryption, and `cuda.data` will be the result of the GPU encryption. `recovered.data` is used as result for the GPU decryption, which should ultimately be identical with `original.data`. Finally, `sequential_recovered.data` is the decrypted file for the CPU algorithm. To test whether two files are identical (that is, to check whether the recovered file is the same as the original one), use the `diff` command. If this command produces non-empty output, the input files are not identical, so you must debug your solution.

Do **not** *submit any input files with your solution! Instead, please make sure you state, in your report, the size (in bytes) of each input file.*

Evaluation Report speed-up per file per operation (i.e., encryption and decryption separately), and compare these speed-ups for the different file sizes (we appreciate a visual comparisons, i.e., consider graphs rather than tables). Is there a correlation between the size of the files and the performance of the application for the sequential and the GPU versions? Explain.

2. **Multi-key extension** An extension of this encryption algorithm is to use a larger key - i.e., a set of values, applied to consecutive characters. For example, `Caesar('ABCDE', [1, 2]) = 'BDDFF'`. Implement this encryption/decryption algorithm, known as the Vigenère cipher[1], as an extension to the original version. You can assume the key is already known (fixed, constant), and choose its length (or, better, test with multiple lengths). Optimize your code as you see necessary for this extended version, and test the extended version the same way you for the basic algorithm (i.e., using the same files and checking the two operations separately).

   Finally, evaluate the performance of the new version for the same files, and compare the performance results against the single-value key operations. We would appreciate a visual form of this comparison (again, graphs instead of tables), and comment on your findings. How is the performance changing?

3. **Checksum** Implement a kernel to *efficiently* calculate the checksum of a file in CUDA (use a simple, additive checksum).

   Apply this checksum calculator to both the original and encrypted files, and compare the output. Comment on your findings.

   Finally, please evaluate the performance of the checksum calculation for all the encrypted files you used in the previous tests, and report the performance of the kernel for each file. Comment on the observed performance, specifically focusing on the question "is there a correlation between performance and the file size?"

Additional Information

Grade weights

Table 1: Weights of different parts of the assignments used in calculating the final grades.

Assignment	Name	Code	Report
2.1	Wave equation simulation with CUDA	25%	25%
2.2	Cryptography	25%	25%
		50%	50%

[1]`https://en.wikipedia.org/wiki/Vigenere_cipher`

Instructions for submission

For this assignment we expect two deliverables:

1. Source code in the form of a tar-archive of all relevant files that make up the solution of the programming exercise with an adequate amount of comments. Additionally, please include any Makefiles (in case they were modified) to showcase different settings needed for additional experiments you might have run and analysed.

2. A report in the form of a pdf file that explains the developed solution at a higher level of abstraction, illustrates and discusses the outcomes of experiments and draws conclusions from the observations made. For indications on how to write the report, check the report writing guidelines on Canvas[2].

[2] These are the same as the indications from assignment 1.