INF3201 Parallel Programming

Assignment 3 - CUDA C

06 October - 27 October 2025

1 Finding the greatest number

In this assignment, the task is to find the greatest number in an array of integers. One serial and one parallel version must be implemented. The parallel version will be implemented in CUDA C/C++ and run on NVIDIA GPUs. Use *reduction trees* for your parallel design. The idea is similar to what is used on reduction trees in MPI collective operations (see Figure 3.6 in the book). Implementing this idea on a GPU will look different, but the core idea is still the same. Most of the needed background is found in chapter 6 of Pacheco and Malensek [**RN266**], but you may have to consult the CUDA Programming Guide for details (https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html).

1.1 Precode

There are no precodes for this assignment. Write the code in C/C++ and compile with *nvcc*. Describe how to compile and run the solution in the report.

2 Getting to know the GPU

The first part of the assignment is to explain general terms regarding the architecture of NVIDIA GPUs. Briefly explain these terms with regards to NVIDIA GPUs and CUDA in your report's introduction:

- GPU
- SIMT
- Threads, blocks, grids, and warps
- Core (SP) and streaming multiprocessor (SM)
- GPU global memory, shared memory, registers

3 Design and implementation

3.1 Serial version

For the serial version, implement a loop that finds the greatest number in the array.

3.2 Parallel versions

For the parallel version, use a reduction tree approach to find the greatest number in the array. You can implement reduction trees at several levels, e.g., at warp, block, and global level. A suggested solution is to utilize a feature of CUDA called warp shuffles for the warp-level reduction tree. The approach is outlined here:

1. For each warp of 32 threads, find the greatest number within a correspondingly short range of the array, e.g., by using the CUDA-function __shfl_down_sync().
2. For each block – consisting of multiple warps – do a reduction by using *shared memory* and *__syncthreads()*.
3. Finally, find the globally greatest value, and make this value available to the host. Here, one of CUDA's *atomic functions* may be useful if you use a kernel for your solution. Another approach is letting a loop on the host make a kernel-call per iteration, where one kernel-call computes a single level of the tree.

Make sure to provide a README file that explains how to run your sequential and parallel codes.

4 Experiments and results

4.1 Generating data

You should at least experiment with three array sizes initialized with pseudo-random integers. We suggest 134217728 (2^{27}), 268435456 (2^{28}), and 402653184 ($2^{27} + 2^{28}$) elements. These sizes correspond to 0.5 GiB, 1 GiB, and 1.5 GiB of memory given that we use 32-bit integers. Ideally, the sizes should be larger. If your GPU is equipped with enough global memory, you can experiment with larger arrays as well. You can also decide if you want to generate the arrays once, save them to disk, and load them for consecutive runs.

4.2 Taking timings

Measure the time spent executing the serial and the parallel version. Use a wall clock time function with sufficient resolution for the measurements. Repeat each experiment 30 times to account for timings variability.

For the parallel version, separately report the times spent copying data to/from the device and the time spent running the kernel, as well as the total time. You do not have to measure the time spent allocating and freeing memory in this assignment.

NOTE: To profile CUDA code, it may be safe to look into cudaEventRecord.

4.3 Include in your report

- The minimum elapsed time
- The mean of the elapsed times
- The percent of time the parallel versions spent in comparison to the serial version

Include relevant information about CPU and GPU model, clock speeds, and memory. For the GPU, include number of cores and SMs. The mean elapsed times should be presented in a table and easy to compare.

4.4 Comparing environments

Additionally, run and report your serial and parallel experiments with one of the array sizes in two different environments. Include relevant information about the environments, such as the OS, and the CPU and GPU model, clock speeds, and memory. For the GPU, include number of cores and SMs.

- If your local computer has a suitable GPU, run your experiments locally and on a node in the cluster equipped with an NVIDIA GPU.
- If you have two different usable operating systems or environments, you can run your experiments on them.
- Alternatively, you can run on two different nodes in the cluster if you do not have other options.

5 Report

The report should be between 4 and 6 pages. Remember to explain and answer all questions from the assignment. Make your figures and tables clear and understandable. Include captions for figures and tables. Also, include a reference list at the end of the report, if any. The report should have the following sections:

- Introduction
  - Give a short introduction to the assignment and a background explaining how GPUs are used for parallel computing, based on what you did in "Getting to know the GPU"
- Method
  - Explain how you used reduction trees to parallelize the problem and your particular solution's design
  - Consider including figures
- Implementation
  - Here you can give more information about various implementation details
  - How to compile and run your code
- Experiments and results
  - Detail the hardware you used for testing
  - Compare the serial and parallel versions of your program
  - Provide the information and tables requested in the assignment text
- Discussion
  - What was expected/unexpected in your work?
  - What is the major time-consuming part of the parallel version, and can you think of situations where the GPU would do even better?
  - Did you notice any important differences between the execution times for the environments, such as the ratios between the serial and parallel times?

  - What can be improved in your work?
- Conclusion
  - Conclude your work
- References
  - A list of references that you have cited, if any

6 Minimum expectations

At a minimum, you need to implement a serial version and a parallel version using the GPU. Your parallel implementation must use reduction trees somehow, but it is not required to be exactly as suggested in the assignment text. The code must compile and run. The experiments should be automated, i.e., in your program or a script. You must write a report according to the descriptions given in the assignment text. Explaining the design of your solution in the methods section is important. Use tables and figures to present results. If you can't run all experiments, focus on one array size and run on one system. If your solution does not fully work or was not finished in time, clearly state what was missing in a separate section in the report.

7 Hand-in

You will work in groups of two for this assignment. GitHub classroom is used as a hand-in platform for the course. You can and probably should commit and push as often as possible. Remember to push all your changes before **27 October 2025 23:59:59**. Any changes pushed to your repository after that deadline will not be evaluated.

8 General

The general rules for assignments apply. The assignment is a learning experience. Do not copy code from others, do not use any code generation tools or any third-party libraries. If you use smaller code snippets or inspiration from other places, please include links or references in the report and code. Do not give this assignment to others outside the course, and do not share your own solution with others.

9 Practical issues on the cluster

We have experienced some code generation issues on the cluster with at least some of the GPUs and the current CUDA installation. To avoid any issues, you may want to add -arch sm_50 to the compiler flags. That should ensure that the compiled code would work on the older GPUs that we have in the cluster.

If you are interested in more details and possible optimizations for newer GPUs, please check the compiler directives for the *nvcc* compiler.

There are other options, but this is a relatively simple workaround that works for now. It should also save you from spending time on mysterious bugs with code that appears to work but doesn't quite do the right thing.

The `/share/ifi/list-cuda-hosts.sh` script can be used to find target nodes for running your program on.

You can use the following Python code to check the compute capability of the current node:

```
#!/usr/bin/env python3

import pycuda
import pycuda.driver as cuda
import pycuda.autoinit
import socket

# Assume just one GPU
dev = cuda.Device(0)
attrs = dev.get_attributes()
major = attrs[pycuda._driver.device_attribute.COMPUTE_CAPABILITY_MAJOR]
minor = attrs[pycuda._driver.device_attribute.COMPUTE_CAPABILITY_MINOR]
print(f"{socket.gethostname()} : CC {major}.{minor}")
```