Low-Level Parallel Programming

Assignment 2
Deadline: February 14 23:59

Uppsala University, Spring 2025

1 Deadlines, Demonstrations, and Submissions

For each lab, a .zip file containing the lab report and your source code has to be submitted by its corresponding deadline. During the lab session on the day of the deadline, demonstrate your working solution to a lab assistant. Be prepared to answer questions about your solution.

For this lab, there is an opportunity to obtain a bonus point. Bonus points can be accumulated throughout the rest of the course. One bonus point will raise your final course grade from 3 to 4, while two bonus points will raise it to 5. There will be two more opportunities to get bonus points after this lab. Please remember to divide the work equally within your group!

2 Overview

The objective of this lab is to recognize vectorization possibilities within serial code, and to be aware of the possibilities and limitations for SIMD operations and CUDA kernels. If a part of the code isn't suitable for a particular parallelization, then keep it sequential and defend your decision during the demonstration and the report.

The CUDA part of this assignment is optional. If you complete this part, you will be awarded with a **bonus point**. The bonus point counts towards a higher final grade.

Note: This lab requires significantly more effort than the previous one. Make sure to start on time.

3 Lab Instructions

Make an additional parallelization of the code section you sped up during lab 1, using SIMD (vector) operations. On successful completion of the assignment, the calculation for the next position of in total **four** agents is computed **at the same time** by a single thread. This requires you to unroll the loop in focus by four iterations, and to rewrite the loop body such that these agents are handled in parallel. Note that it is **not allowed** to repeatedly gather data into aligned memory for every *tick* call. Instead, a consecutive vector containing the relevant data should already exist that you just need to access.

3.1 Before you start

If you make extensive changes to the project it may change the performance of the original sequential version from Assignment 1. Therefore, measure the performance of your current sequential version, and store the results (runtime in seconds) to compare with your data after making changes. Use the *three scenarios* available in the git repository for your measurements (hugeScenario.xml, scenario_box.xml, scenario.xml). It is probably a good idea to vary the tick count for the scenarios, in order to get a reasonably short test time while still getting accurate readings.

3.2 Refactor the Codebase

Vectorization is quite a different beast compared to multi-threading, often requiring a different mindset than the object-oriented one. Do not be afraid of doing extensive structural changes in the project for this lab. However, it is a requirement that there must still be a functioning sequential, C++ thread, and OpenMP version[1] when you are done. Below is a helpful line of questioning to get you started.

- How must data be placed in memory to be suitable for SIMD?
- Is this object-oriented code as it is now suitable for vectorization?
- What structural changes to the code do you want to do in order to expose the vectorization possibilities?

Do not be afraid of modifying given code and even refactoring out entire functions. The code will not bite back (unless you program it to). Additionally, do not be afraid of discussing your ideas with the lab assistants.

3.3 Parallelization with SIMD

You can use the Intel Intrinsics Guide/Arm Intrinsics to look up available functions. Crocodile supports up to SSE 4.2, as well as AVX and AVX-512. Note that systems do not necessarily support all variants of SIMD instructions. Some Arm based systems will have Neon instead! Thus it is critical to check the type of instructions supported on your machine.

Remember: a requirement is that you process 4 agents at the same time using SIMD (why only 4? think about it, maybe you can process 8!!) and that repeatedly gathering data into aligned memory for every *tick* call is disallowed.

3.4 Working on your own machine that has an Arm ISA

Low-level programming involves writing code that is - by definition! - closer to the hardware in the software abstraction hierarchy. This means that the code is often written with a particular architecture in mind, and SIMD instructions are a case in point. Here you will see however that with a few small alterations or mitigations, one can write code that can be compiled for various architectures.

It is therefore totally fine to work on your own machine, even if it has an Arm ISA. This could be Macs or even Windows machines with Arm processors. In this case, you can use Neon instructions for the SIMD part. However, because you will eventually need to move back to a x86 machine with a Nvidia GPU, here are some suggestions that I have.

For Labs 1, 2 (non-bonus part), 3 - you can use your own laptop. Make sure that you implement the refactoring of the data layout as instructed above. That will be crucial for the future labs. As long as OpenMP/threads works with your refactored code (which is a requirement for this lab), you can be graded on your laptop.

If you want to go for the Lab 2 bonus, and work on the Lab 4 part, all you will need to do is comment out your SIMD instructions. (Keep the refactored data layout!). By commenting out your SIMD instructions, your code should be buildable on a x86 machine and you should be able to make use of the refactored code in your CUDA programs.

Please be aware that you will need to provide two demos for the lab 2 if you go for the bonus. One on your machine (using the Arm SIMD), and another on the crocodile machine (or any other machine that has a Nvidia GPU) that shows the bonus part.

Note that this is a workaround, rather than a solution. If you want an extra challenge, see if you can write your code so that it compiles on both architectures, without commenting out any source code. No bonus points here, but you'll get kudos from us!

Please post a question on the discussion board or ask the teaching staff if you have any other questions.

[1] It is ok to modify the assignment 1 code to accomplish this if you need to, just make sure that you have done all the timing measurements for the first report before you do.

3.5 BONUS POINT: Parallelization with CUDA

Make a final parallelization of the same code section with a CUDA kernel. You may also choose to use OpenCL for this, if you have decided to work on your own laptop and don't have an NVIDIA graphics card. Depending on your prior solution, it may be a good idea to use the vectorized code as a starting point.

Please submit and be graded **on time**! Late submissions/gradings will **not** award you bonus points. If you have scheduling conflicts for gradings, then you MUST contact the TA well in advance to reserve another slot to be graded. However the submission must be made on time regardless of the rescheduled grading session.

3.6 Evaluation

Obtain the execution time for all versions (Assignment 1 Serial[2], Serial, OpenMP, C++ Threads, SIMD, and CUDA for the bonus point), using the three scenarios given for this lab (available at Studium). When timing, be sure to use the `--timing-mode` command line argument. Use the number of threads that you have identified as optimal for OpenMP and C++ threads during Assignment 1. *Run each experiment several times and use the average value.* Visualize the data with a bar plot. You can change the scenario file by sending an additional launch argument, i.e., **demo/demo hugeScenario.xml** or to simply change the file name directly in the source code.

3.7 Questions

A. What kind of parallelism is exposed with your code refactorization?

B. Explain your solution. Include, amongst other details, an answer to the following questions:

- What kind of code transformations were required for this lab, and why were they needed?
- How is an agent represented in the new code?

C. Did the performance of the serial version get affected by the changes you made to allow SIMD operations and if so, how?

D. How can one determine which SIMD operations are supported on a given machine? List the instruction set families available on the machine you used (either your own or Crocodile), and how you obtained this information.

E. Which parallelization has been most beneficial in these two labs? Support your case using plotted timing measurements.

F. Compare the effort required for each of the three (four) parallelization techniques. Would it have made a difference if you had to build the program from scratch, instead of using given code?

G. (Bonus:) List your GPU specifications.

H. (Bonus:) Is this code section suitable to be offloaded to the GPU? What makes it suitable / not suitable?

I. (Bonus:) Give a short summary of similarities and differences between SIMD and GPU programming.

4 Submission Instructions

Submit the code and a **short** report including all answers to above questions in **PDF** format to studium.

1. Your code **has** to compile.
2. **Clean** your project (run `make submission` which will copy code files from demo and libpedsim and make a tarball called submission.tar.gz.

[2]Obtain this data before you make changes to the code.

3. Document how to choose different versions (serial, C++ Threads, and OpenMP implementation).
4. Include a specification of the machine you were running your experiments on.
5. State the question and task number.
6. State in the assignment whether all group members have contributed equally towards the solution - it will be important for your final grade.
7. Include all generated plots.
8. Put everything into a .zip file and name it team-n-lastname$_1$-lastname$_2$-lastname$_3$.zip, where n is your team number.
9. Upload the .zip file on Studium by the corresponding deadline.

5 Acknowledgment

This project includes software from the PEDSIM[3] simulator, a microscopic pedestrian crowd simulation system, that was originally adapted for the Low-Level Parallel Programming course 2015 at Uppsala University.

[3] http://pedsim.silmaril.org/