Coursework: Performance Modelling, Vectorisation and GPU Programming

Module: Performance Modelling, Vectorisation and GPU Programming (COMP 52315)
Term: Epiphany term, 2024
Lecturer: Anne Reinarz[1] and Laura Morgenstern[2]

Submission Please submit a zip-directory containing all files required for this assignment via the Gradescope submission point provided via Blackboard Learn Ultra at https://blackboard.durham.ac.uk/ultra/courses/_54359_1/outline.

Deadlines Consult the MISCADA learning and teaching handbook for submission deadlines.

Plagiarism and collusion Students suspected of plagiarism, either of published work or work from unpublished sources, including the work of other students, or of collusion will be dealt with according to Computer Science and University guidelines.

Preface You inherited the number crunching code contained in `number_crunching.cpp`. Your goal is to conduct a performance analysis of the code and develop a parallel implementation for GPUs to minimize its execution time while maintaining correctness.

Question 1

The objective of this question is to examine the performance of `number_crunching.cpp` on CPUs. Please use Hamilton 8 for all measurements in this question. Use the slurm partition `shared` (add the `#SBATCH -p shared` directive to your job script) for development, and the partition `test` (add the `#SBATCH -p test` directive) for your final performance measurement. The `test` partition will allow you to reserve a whole node, and exclusive access will help you ensure reliability of the measurement and reproducibility of the result.

(1.1) In `report_<CIS_username>.pdf`, report the number of data accesses (loads and stores), the number of floating point operations, and the best case operational intensity of the functions `function_a()`, `function_b()`, `function_c()`, and `function_d()`. The latter should be determined by hand—this task requires looking at the source code, but not running it. For clarity, you may want to use a table.

Marking scheme. Each function is worth 3 marks: one for the correct data access count, one for the correct floating point count, and one for the correct operational intensity. Marks will be awarded independently: if your data access and/or floating point operation count are incorrect, but you use them correctly in the formula for the operational intensity, you will still get the mark for operational intensity. Be sure to explain your process clearly in the report.

[12 marks]

(1.2) Use the profiling tool `gprof` to determine the overall execution time and the hotspot function(s) that requires the most execution time. In order to determine the influence of the problem size, carry out the profiling for $N = k \cdot 10^4$, where $k = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$. Visualize the execution times graphically in `report_<CIS_username>.pdf`. Submit a script `number_crunching_gprof.sh` containing the commands you used to profile the code, and the file `number_crunching_gprof.out` with the output produced by `gprof` for the largest input data set.

Marking scheme. In your report the graph is worth 4 marks, the reported output is worth 5 marks and the explanation of the output using your results from Task (1.1) is worth 5 marks. If the script

[1] anne.reinarz@durham.ac.uk
[2] laura.morgenstern@durham.ac.uk

and output of the script are not provided a mark of zero will be given.

[14 marks]

(1.3) Use `likwid` to profile the memory and floating point performance of the hotspot function as determined in Task (1.2) for $N = 10^4$. Submit the instrumented code `number_crunching_likwid.cpp`, a shell script `number_crunching_likwid.sh` with the commands you used to run the benchmark, and a file `number_crunching_likwid.out` containing the output of those commands. In `report_<CIS_username>.pdf`, report the observed operational intensity and discuss the results.

Marking scheme. The correct instrumented code and shell script are worth up to 3 marks each, the correct output is worth 3 marks if correct and 0 otherwise. The write-up is worth 6 marks.

[15 marks]

(1.4) Visualise the results obtained for the hotspot function as determined in Task (1.2) in the form of a roofline model. In `report_<CIS_username>.pdf`, outline what techniques could be applied to optimize the execution time of the code on CPUs based on this. Your discussion should be no longer than 150 words.

Marking scheme. Your discussion must take into account the roofline model and the results of Tasks (1.1)-(1.3). No marks will be given for a general discussion, which doesn't take the measurements into account. The roofline model is worth 3 marks and the discussion is worth 6 marks.

[9 marks]

Question 2

Please use NCC to develop and test all subsequent implementation tasks.

(2.1) Use either CUDA, OpenMP or SYCL to develop a GPU implementation of `number_crunching.cpp` that exploits the loop-parallelism contained in the compute functions `function_a()`, `function_b()`, `function_c()` and `function_d()`. Save your source code as `number_crunching_loop.cu` or `number_crunching_loop.cpp` (dependent on your chosen programming model) and adjust the `Makefile` to build your GPU code by adding a target `loopgpu`.

Marking scheme. Only implementations that maintain correctness under concurrent execution will receive marks. An implementation is correct if it corresponds to the reference solution as computed by the sequential implementation for the same problem size N.

[18 marks]

(2.2) Draw the task graph of the source code in `number_crunching.cpp` at function level and include it in the `report_<CIS_username>.pdf`.

Marking scheme. 1 mark is awarded for each function node with the correct data dependencies.

[6 marks]

(2.3) Extend your GPU-implementation of `number_crunching.cpp` to also make use of task parallelism by executing independent compute functions simultaneously. Your implementation should allow to execute all data-independent functions, as determined in Task (2.2), concurrently. Submit your source code as `number_crunching_task.cu` or `number_crunching_task.cpp` (dependent on your chosen programming model) and adjust the `Makefile` to build your GPU code by adding a target `taskgpu`.

Marking scheme. Only implementations that maintain correctness under concurrent execution will receive marks. An implementation is correct if it corresponds to the reference solution as computed by the sequential implementation for the same problem size N.

[6 marks]

(2.4) In `report_<CIS_username>.pdf`, write a short essay (500 words at most) that describes your approach to porting the code to GPUs and justifies the techniques used. Use plots as you see fit. In your report, you should:

- Explain why you chose the programming model you applied,
- Outline the memory management approach,
- Justify any code restructuring and performance optimization techniques,
- Outline how your approach allows for the overlapping execution of kernels in a task-parallel manner,

- Determine the runtime for a problem size N of your choice and compare the corresponding runtime with the sequential runtime on CPUs. If applicable to your programming model, determine the optimal kernel configuration.

Marking scheme. A maximum of 4 marks is awarded for each of the points listed above.

[20 marks]