TDT4260 Image Optimization

tl;dr: Make it go fast/efficient and **keep a log (not a report!)** of what you do and graph performance. You can work in pairs

Problem

"We have just made an amazing new image processing algorithm. This will change everything! Now we need you to optimize the code massively. The algorithm developers need a fast version on their workstations. In addition, the product department believes it will be too slow for our mainstream ARM based devices. I have already told them you can solve it. You know the presentation to the investors is on next Monday, so get to it." – Your boss

Your colleague tells you that the program implements a difference of gaussians in three sizes using a box blur.

Goal

The goal is to explore and measure as much as possible. Try to measure different variants using `make run` and Climbing Mont Blanc (CMB), and explore where the code spends most time using `make perf`. As you optimize portions of the code, other portions will grow in importance. Optimize the code. Try to parallelize it using both OpenMP and SIMD instructions.

The code has also been ported to OpenCL. Try to run and improve the OpenCL solution using some of the optmizations you previously discovered. We don't expect the same level of work here.

- Try to find both fast (low runtime) and efficient (low energy delay product (EDP)) solutions.
- **Keep a log of what you tried and how it performed (short notes for optimizations - no full report).**
- If you get stuck and need to debug your code then write down what you did.
- **Create a graph showing the runtimes of your different solutions on CMB and the VM, i.e., one line for each platform.**
- **Create a second graph showing the EDP of your different solutions on CMB.**

Climb & VM

Create an account on CMB (please use your NTNU email; requires being connected to the NTNU network or using NTNU's VPN) and select *TDT4260 Image Optimization*. Here you can upload your solution to measure how much time and power execution takes on an Odroid ARM board.

Once your solution has been uploaded and verified you can also profile it to find out which parts take longest. There is a leaderboard that can be sorted by runtime, energy usage, and energy delay/runtime product (EDP).

As CMB does not allow debugging you can develop and debug your improvements on the VMs (`ssh username@tdt4260-username.idi.ntnu.no`).

C & make commands

`image_processing_c.c` implements a naive approach to solving the problem. The code is bad and far too slow to be run on CMB. Assume that someone that hardly knows programming wrote the code. You may modify the `Makefile`, but the CMB server uses its own. Your submission should include `ppm.c`, `ppm.h` and (your improved version of) `image_processing_c.c`. A suitable zip file can be generated using `make zip`.

You can run the code using `make run`, profile it using `make perf` and check its output against the reference using `make check`.

OpenCL

Additionally the code was translated to OpenCL that can run on GPUs. This code already processes all three color channels at once but it is still quite inefficient. This code consists of two parts:

- The part running on the CPU in `image_processing_opencl.cpp`
- The part running on the GPU in `image_processing_opencl.cl`

The OpenCL version provides the same make commands but requires the addition of `VERSION=opencl` (e.g. `make run VERSION=opencl`)

As the VMs have no GPUs, both parts will run on the CPU and there will be no visible speedup. The OpenCl example should already be fast enough to run on CMB, as it can use the GPU there.

Accuracy & Checker

- When working on real problems the answer can be open for interpretation.
  - Can you solve the problem in a different way?
  - How much precision is needed?
  - Is pixel accuracy needed?
  - Reusing data/math.
- You are allowed some minor pixel errors in the final output(s). Each pixel is in the range 0-255.
  - A few thousand pixels with +/- 1 differences is fine.
  - A few hundred larger differences is fine.
  - The included `checker.c` program will inform you of the number of bad pixels. Use the `make check` rule to run a pixel error check.

  - The checker also outputs `flower_tiny_errors.ppm`, `flower_small_errors.ppm`, `flower_medium_errors.ppm`. These images have red pixels for wrong values and can help with debugging.

Optimizations (from easy to hard)

- There are several simple changes that will improve performance. You should consider the following:
  - Caches and access patterns.
  - Useless code.
  - Needed precision
  - Branches and data layout/usage.
  - The amount of operations performed by your program.
  - The Wikipedia article about box blur.
- OpenMP supports parallelization of loops and sections using compiler directives
  - This provides a short reference of OpenMP directives
  - As the VMs only provide one core the number of threads needs to be manually set to something greater than 1 to test OpenMP (`omp_set_num_threads(4)`). This will not provide any speedup
- SIMD code is not necessary for good performance but it can provide a further speedup. Use GCC Vector extensions, since they can run on the x86 VMs and the Odroid ARM board CMB uses.
- OpenCL can provide great speedups at the cost of complexity. Try this out once you can get no further speedup on the CPU
  - focus on the kernel in `image_processing_opencl.cl` first
  - The ARM documentation lists general and Mali-specific OpenCL optimizations