Notes on the first weekly assignment in Parallel Functional Programming

Troels Henriksen
DIKU, University of Copenhagen

November 2019

1 Introduction

This note has been written to address flaws I commonly notice in student reports on Parallel Functional Programming. It is not intended as a primer on Futhark programming, nor does it constitute a reference solution—in fact, I shall only discuss a subset of Exercise 1. The focus is on how to present the result of your work, including your experiments, in a convincing manner. The code discussed is available in the accompaying archive `futhark-lab-notes.tar.gz`.

2 The Exercises

Exercise 1.1

The program here is a very straightforward map-reduce composition. The only wrinkle is the need to create a wrapper `main` function that calls the actual process function:

```
let process (xs: []i32) (ys: []i32): i32 =
  reduce i32.max 0 (map i32.abs (map2 (-) xs ys))

let main (xs: []i32) (ys: []i32) = process xs ys
```

Exercise 1.2

This exercise asks us to compile the program we wrote in the previous section with two different compilers (`futhark c` and `futhark opencl`), and characterise the resulting performance. While the a course is not deeply concerned with benchmarking, it is still important for a computer scientist to understand how to report benchmark results.

Perhaps the most important thing is to report up front which system you are measuring on. This is not something that belongs after the results, or squirreled away in an appendix. Performance measurements are useless to the reader until they know what system is being used.

Second, make your experimental methodology clear. Make it clear what or how you measure (for example, do you count GPU initialisation time, or time taken to load data from disk?). If you are using an existing/standard benchmarking tool, it is acceptable to simply state that you are using that tool—you do not have to investigate exactly how it operates; it is probably doing something sensible.

Third—and this is mostly for your own sake—make your experiments repeatable. Ideally, everything, from compiling the benchmark programs to generating the performance graphs, should be fully automated and re-runnable with a single command. For some extremely complicated or poorly designed systems, this is not feasible, but Futhark is not among these. Automating the experiments will usually involve writing various scripts, Makefiles, or utility programs to glue together various other tools and convert between data for- mats. This is fine—it does not have to be pretty, it just has to work. You are computer scientists, so do not be afraid to write code!

For this exercise, we will use `futhark dataset` to generate random data sets, `futhark bench` can be used to perform the benchmarking, and a simple Makefile to glue it all together.

We start by adding a header to the Futhark program that describes where to find the data sets:

```
-- ==
-- input @ two_100_i32s
-- input @ two_1000_i32s
-- input @ two_10000_i32s
-- input @ two_100000_i32s
-- input @ two_1000000_i32s
-- input @ two_5000000_i32s
-- input @ two_10000000_i32s
```

This header will be read by `futhark bench` and must be the first thing in the source file. It is also possible to specify expected outputs for every input. These are elided here for brevity, but usually a good idea: benchmark results are worthless if the program produces the wrong result. (Or alternatively: it is very easy to make a program run fast if it does not have to be correct.) We write a generic Makefile rule for generating a data file:

```
two_%_i32s:
  futhark dataset -b --i32-bounds=-10000:10000 \
                  -g [$*]i32 -g [$*]i32 > $@
```

The `futhark bench` tool can generate JSON files containing the results (while also printing them to the screen). This is useful for further processing.

We can write the following Makefile rules to specify how to benchmark our program:

```
SIZES = 100 1000 10000 100000 1000000 5000000 10000000

exercise_1_1-opencl.json: $(SIZES:%=two_%_i32s) exercise_1_1.fut
  futhark bench --backend=opencl \
                --json exercise_1_1-opencl.json \
                exercise_1_1.fut

exercise_1_1-c.json: $(SIZES:%=two_%_i32s) exercise_1_1.fut
  futhark bench --backend=c \
                --json exercise_1_1-c.json \
                exercise_1_1.fut
```

We can now run `make exercise11-opencl.json`

or `make exercise11-c.json` to generate the corresponding JSON files containing run-time measurements for every dataset.

It is straightforward to write a simple Python program to parse these JSON files and use the Matplotlib library to generate graphs of the results. I am certainly no skilled Python programmer, but I usually manage to cobble together a crude script (using Google and Stackoverflow liberally throughout the process). If you are more comfortable in another language with good plotting facilities, feel free to use that. The important thing is that the graphs can be generated automatically based on data files.

I have run the benchmarks and generated the graphs on two different systems:

- My laptop, which contains an Intel Core i7-7700HQ CPU and an Intel HD Graphics 630 integrated GPU.
- Some rack server, which contains an Intel Xeon E5-2550 CPU, and an NVIDIA RTX 2080 Ti GPU.

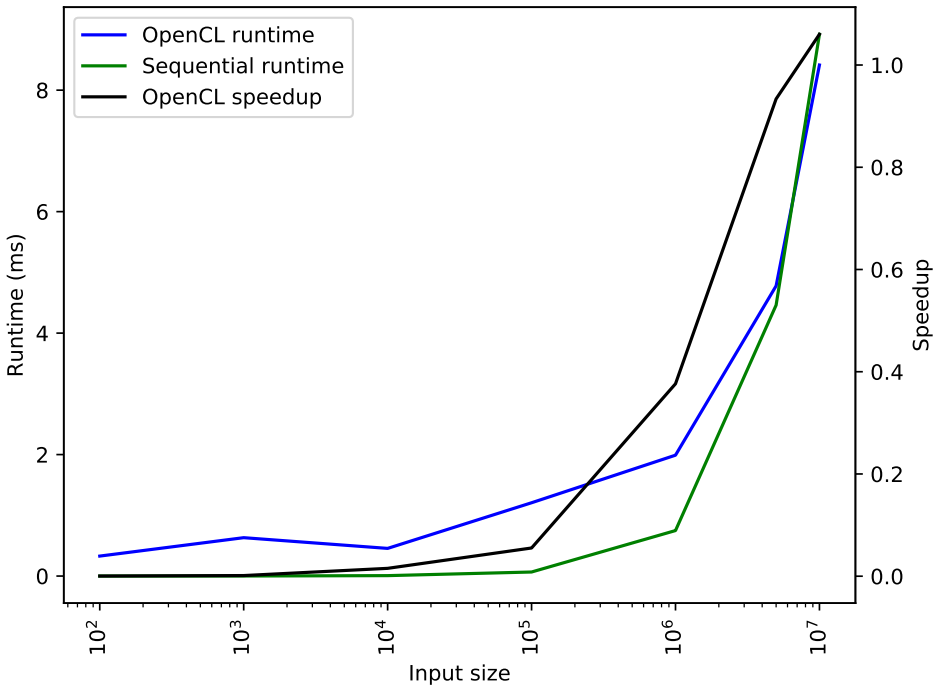
On fig. 1 I report both absolute runtimes and OpenCL *speedup*. The latter is simply the sequential runtime divided by the OpenCL runtime, which quantifies how much faster the OpenCL version is compared to the sequential version. When we benchmark, we are often (but not always) comparing one implementation to some other reference, and in these cases it is very useful to provide a relative comparison. Speedup graphs are typically much easier to read, and more informative, than pure runtime graphs, although they contain the same information.

There are *many* decisions to make when reporting benchmark results. Logarithmic axes? Labeling specific runtimes? Error bars to show variance in measurements? But often there is no reason to go crazy. The above graphs, simple as they are, tell the core story: on my laptop, the OpenCL version cannot match the performance of the sequential version. On the server, it is more than an order of magnitude faster on the larger data sets. On the smaller data sets, sequential execution wins handily on both machines, due to the overhead of GPU execution. Clearly, using the OpenCL version is only worth it for larger problem sizes, and if you have a good GPU.

As stated previously, PFP is not a course about hard-core benchmarking or low-level GPU architecture. Hence, you are not expected to be able to necessarily explain results as the one above in detail. But they do show, that when you are benchmarking parallel programs, it is a probably more interesting to run them on a system that supports a lot of parallelism. Your personal laptop just isnt that much fun.

For the curious, the results above arise from the fact that this program is massively memory-bound. The amount of computation is fairly miniscule compared to the amount of memory we are accessing, and so we are mostly just measuring overhead plus memory performance. On mobile systems, the integrated GPU typically uses the same memory as the CPU, and so there is no advantage to GPU execution. On the server, however, the NVIDIA GPU has its own much faster (and smaller) on-board memory, which explains the order-of-magnitude speedup. For more compute-bound problems, which are not dominated by the speed of memory, we can expect speedups even on mobile GPUs.

On my laptop:



On the server:

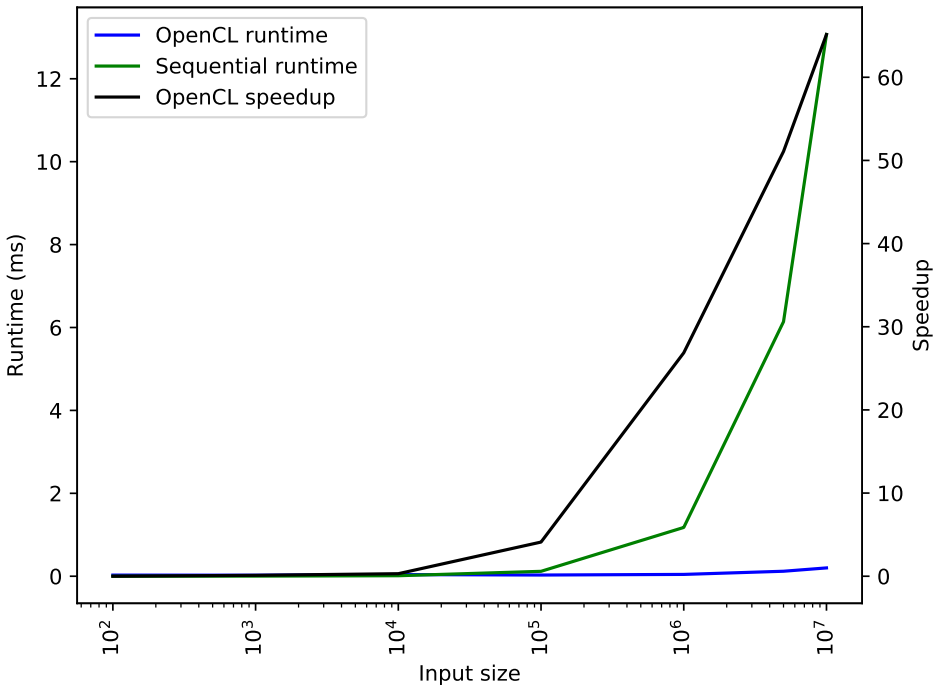


Figure 1: Runtime and speedup of the program written for Exercise 1.1.

Exercise 1.3

This problem is solvable as follows:

```
let process_idx [n] (xs: [n]i32) (ys: [n]i32)
                  : (i32,i32) =
  let max ((d1,i1): (i32,i32)) ((d2,i2): (i32,i32)) =
        if d1 > d2 then (d1,i1)
    else if d2 > d1 then (d2,i2)
    else if i1 > i2 then (d1,i1)
    else                 (d2,d2)
  in reduce_comm max (0, -1)
     (zip (map i32.abs (map2 (-) xs ys))
     (iota n))

let main (xs: []i32) (ys: []i32) = process_idx xs ys
```

The only trick here is a careful formulation of the max function to ensure it is commutative, by using indices as a tie-breaker in case the values are equal. This allows us to use the faster `vreduce_comm` SOAC instead of plain reduce. We did not explicitly use `reduce_comm` in Exercise 1.1, because the compiler automatically recognises reductions with built-in operators known to be commutative, and translates them into commutative reductions. However, when using compicated user-defined reduction operators, we must explicitly tell the compiler whether they are commutative or not.

As we see on fig. 2, the performance story is about the same as for Exercise 1.1. The differences are likely due to the additional complexity of the reduction operator. We note that speedup on my laptop is (slightly) better than for Exercise 1.1; likely because this operator requires more computational work compared to the amount of memory traffic (although the program is ultimately still heavily memory-bound).

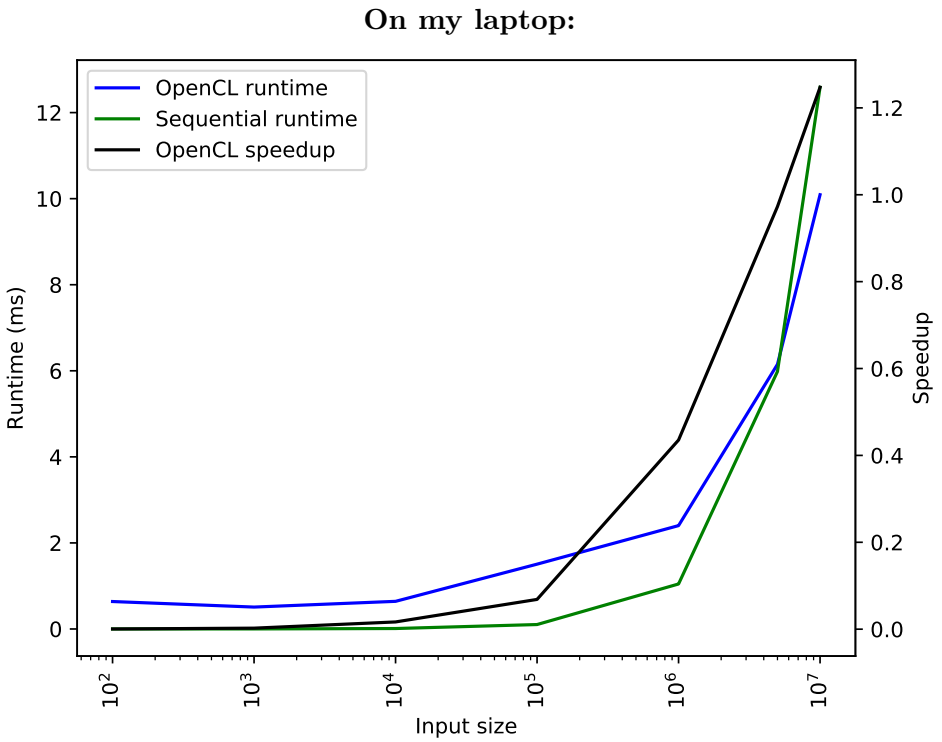


Figure 2: Runtime and speedup of the program written for Exercise 1.3.