LabC Part2

Adam Olsson and Sebastian Frenzel Gabrielsson

April 2020

Getting Started (1)

0.1 Write a Function (1.1)

The results of calling the function on s1 and s2 is 73.

Run your Function (1.2)

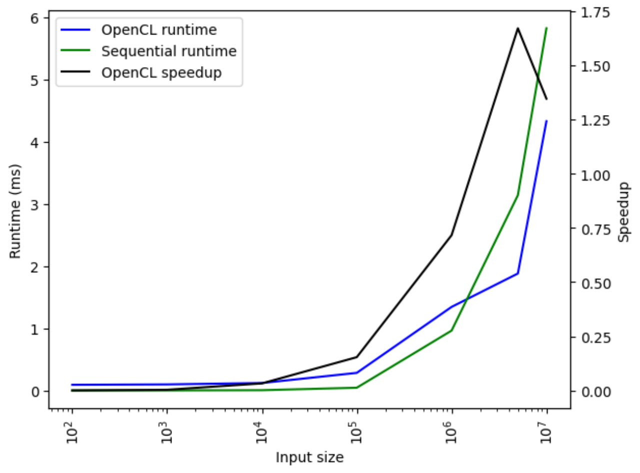

Figure 1: Runtimes for the benchmark of finding the absolute largest difference.

Extend your Function (1.3)

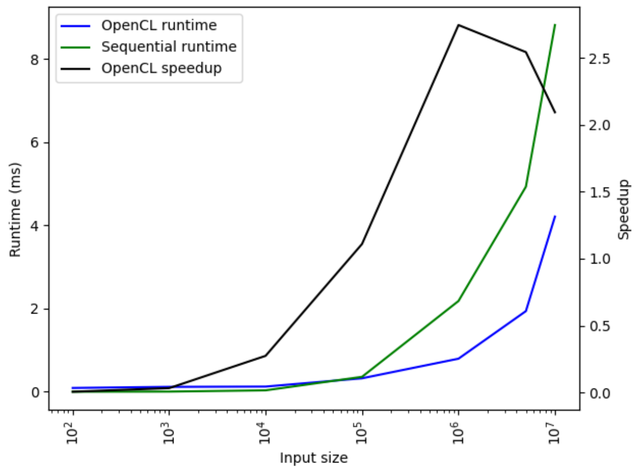

Figure 2: Runtimes for the benchmark of finding the absolute largest difference and returning its index.

Segmented Operations (2)

Proof of Associativity (2.1)

Addition is an associative operator with neutral element 0. Therefore,

$$(0,\ \text{false}) \oplus' (10,\ \text{true}) = (\text{if } f_2 \text{ then } v_2 \text{ else } v_1 + v_2, f_1 \vee f_2) = (10,\ \text{true}) \quad (1)$$

is enough to show that (0, false) is left-neutral. Additionally, whenever the left term has false the body of the function will always evaluate to the right term.

```
let oper 't (op: t -> t -> t)
            ((v1,f1): (t, bool))
            ((v2,f2): (t, bool)) : (t, bool) =
    if f2 then (v2    , f1 || f2)
    else       (op v1 v2, f1 || f2)
```

Figure 3: Source code for the function.

```
oper' (0, false) (10, true) = (10, true)
oper' (0, false) (9, false) = (9, false)
```

Figure 4: Example runs of the implementation.

Segmented Scans and Reductions (2.2)

So the main problem here is not necessarily that the implementation is bad. Instead, to match the given definition of segscan and secreduce, both functions needs to take an array of type (t, bool). However, the array for the sequential scan and reduce should be of type t. To make this go with the dataset generated, I first had to zip the two arrays generated which is included in the benchmark runtime. Therefore, a reason for why the segmented versions performs much worse could be because of this zip. To solve this, I tried to generate tuples using futhark dataset but found that it is not supported. See futhark dataset doc (link).

```
let main [n] (k:[n]i32) (d:[n]bool) = segscan (+) 0 (zip k d)
let main [n] (k:[n]i32) (d:[n]bool) = scan (+) 0 k

let main [n] (k:[n]i32) (d:[n]bool) = segreduce (+) 0 (zip k d)
let main [n] (k:[n]i32) (d:[n]bool) = reduce (+) 0 k
```

Figure 5: Entry functions for the benchmarks.

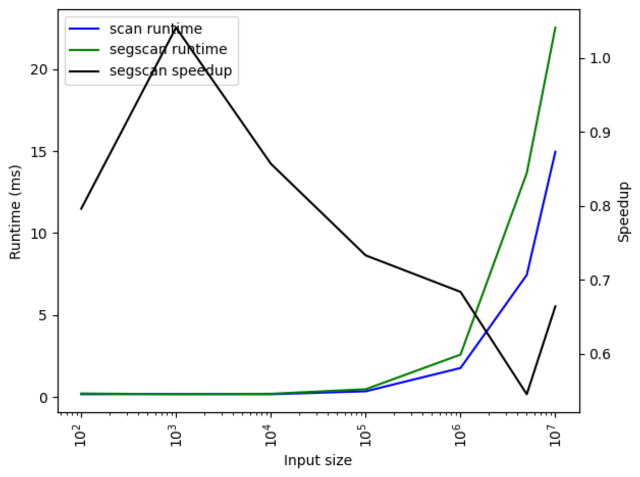

Figure 6: Runtimes for scan and segmented scan from the benchmark.

Implementing reduce_by_index (2.3)

We were unable to implement the given definition of reduce_by_index with the destination argument. We simply didnt manage to write to the destination array. So instead we implemented the same function but without the destination array and are aware of that this can affect performance marginally to the better because of the avoided writing.

We compiled both our and the built in implementation using opencl but did not see any speedups. In fact, our implementation is significantly slower than the built-in version. This becomes especially evident on larger input sizes. The built-in version is most likely faster because the original coder has significant more knowledge about the language than we do.

Looking at the graph for our implementation, the complexity seems to be somewhere around O(N logN).

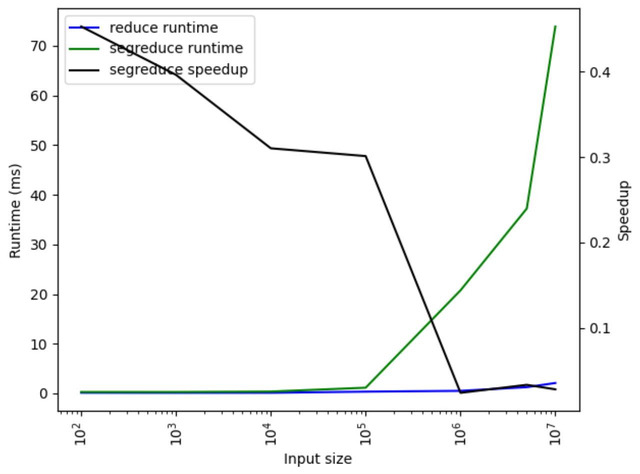

Figure 7: Runtimes for reduce and segmented reduce from the benchmark.

Monte Carlo Simulation (3)

Monte Carlo Estimation of π (3.1 & 3.2)

On larger input sizes, the opencl implementation performs better in terms of runtime. However, on smaller input sizes, the overhead of using opencl is to large, and the sequential version performs better.

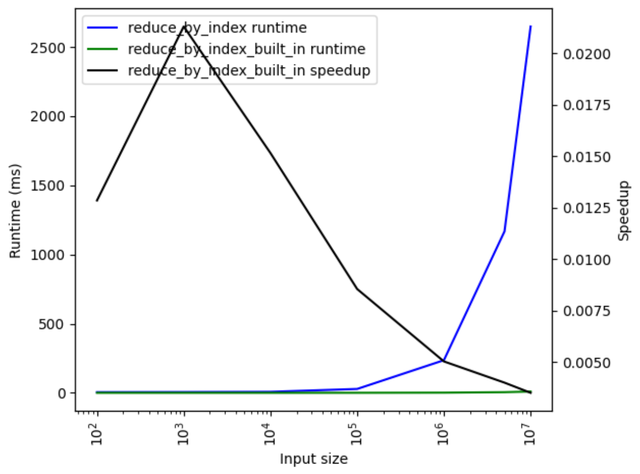

Figure 8: Runtimes for reduce_by_index built-in and our implementation.

Monte Carlo Integration (3.3)

Just as the estimation of π the opencl implementation performs better on larger input sizes and worse on smaller.

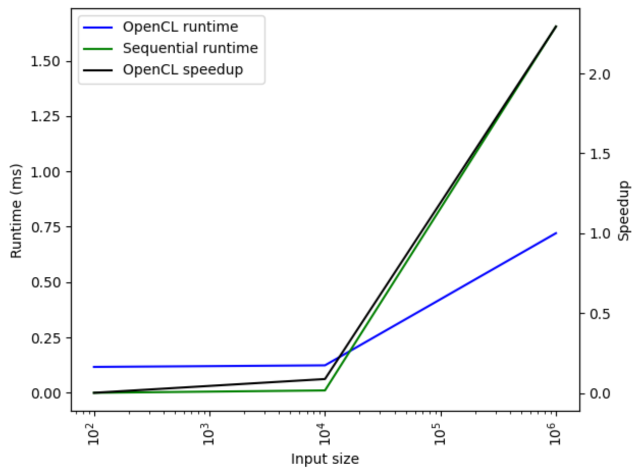

Figure 9: Runtimes for estimation of π from the benchmark.

2D Ising Model (4)

Due to heavy uses of the map function, opencl manages to parallelize these operations quite well. Therefore, we see significant speed ups using the version compiled for open cl. This becomes especially evident in figure 11 where the size of the window grow logarithmic and the duration of the simulation held constant. In figure 12, the size of the window is held constant and the simulation time is increased. Here we can also see that the version compiled for opencl performs much better because of the parallel maps.

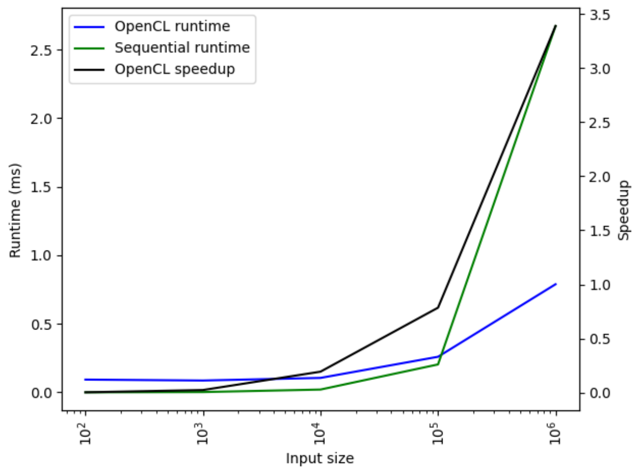

Figure 10: Runtimes for the integration from the benchmark.

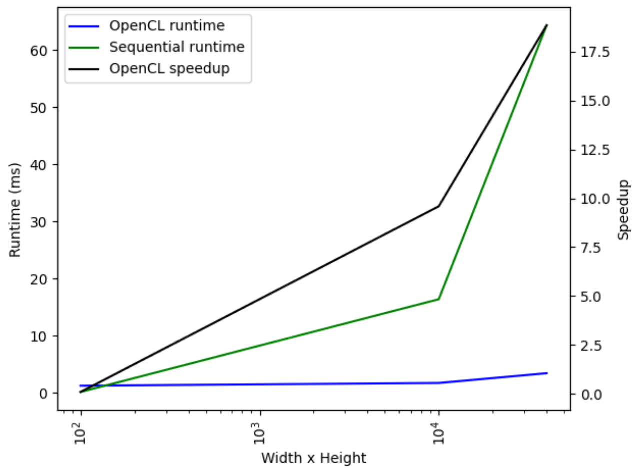

Figure 11: Runtimes for the Ising model benchmark using a fixed n=60.

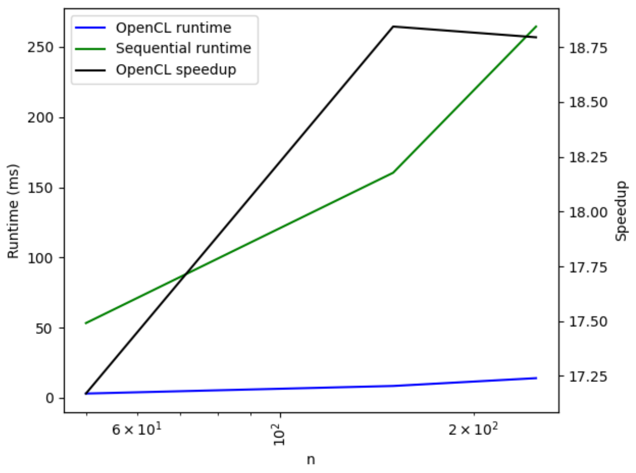

Figure 12: Runtimes for the Ising model benchmark using a fixed size=200×200