Lab C

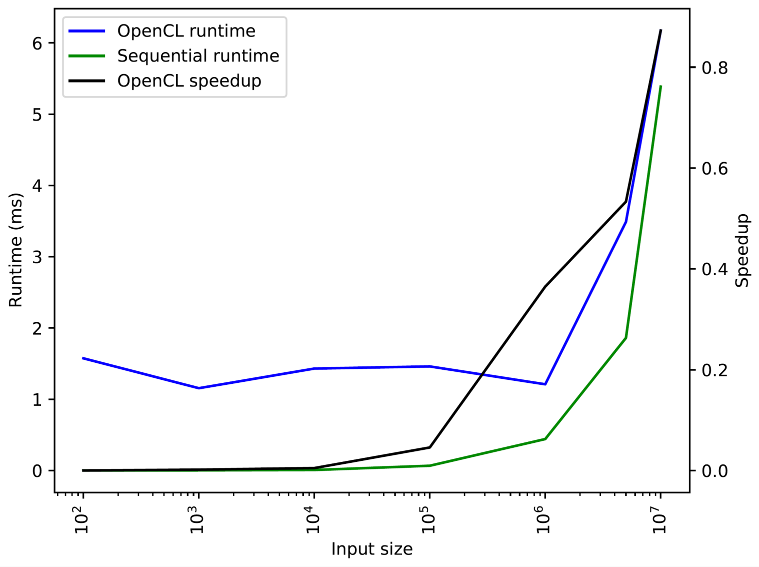
Erik Sievers, Predrag Bozhovikj

Due to issues with Mac OS and NVIDIA cards, we were able to run OpenCL benchmarks only on a single machine, namely a Macbook with an integrated GPU. The results are presented below.

Exercise 1

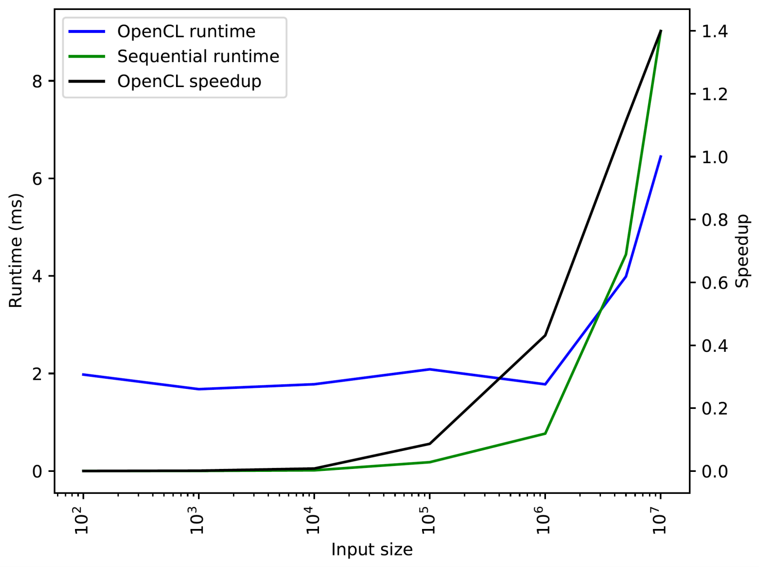
Exercise 1.1

No speedup is achieved over the sequential run for this exercise with the OpenCL runtime nearing sequential runtime for bigger input sizes.



Exercise 1.3

A speedup is achieved over the sequential runtime given $\text{input size} \geq 3 \times 10^6$.



Exercise 2.1

$$\oplus \text{ is an associative operator with neutral element } 0$$

(1)

$$(v_1, f_1) \oplus' (v_2, f_2) = (\text{if } f_2 \text{ then } v_2 \text{ else } v_1 \oplus v_2, f_1 \vee f_2)$$

(2)

$$(2) \;\; \therefore \;\; (0, false) \oplus' (v_2, true) = (v_2, f_1 \vee true) = (v_2, true)$$

(3)

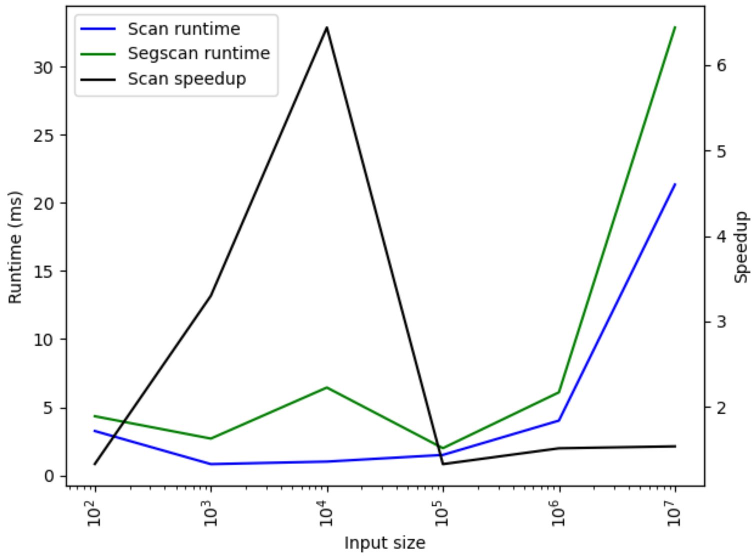
$$(1), (2) \;\; \therefore \;\; (0, false) \oplus' (v_2, false) = (0 \oplus v_2, false \vee false) = (v_2, false)$$

(4)

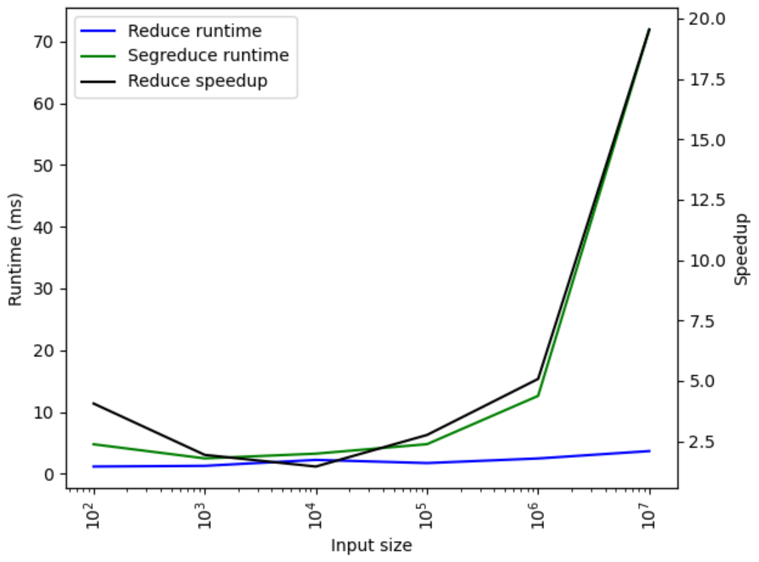
$$(3), (4) \;\; \therefore \;\; (0, false) \oplus' (v_2, f_2) = (v_2, f_2) \quad \blacksquare$$

(5)

Exercise 2.2



The standard scan performs better than the segmented scan, but the speedup seems to vary. Looking at the chart, it's unclear if they scale evenly or not. However, since the segmented scan only performs a small amount of extra work for each element it'd be expected that the speedup is consistent across problem sizes.



The standard reduce performs better than the segmented reduce, and the speedup increases with input size. Comparing the previous chart with this one, we can see that scan actually is quite a bit slower than reduce, which makes sense since reduce only needs to produce the output of the last element produced by scan. In our implementation however, we're using scan, which already is going to bring the performance closer to scan than reduce. Furthermore, we perform a couple of unzip and zip operations, a rotate and a filter operation which also adds to the execution time.

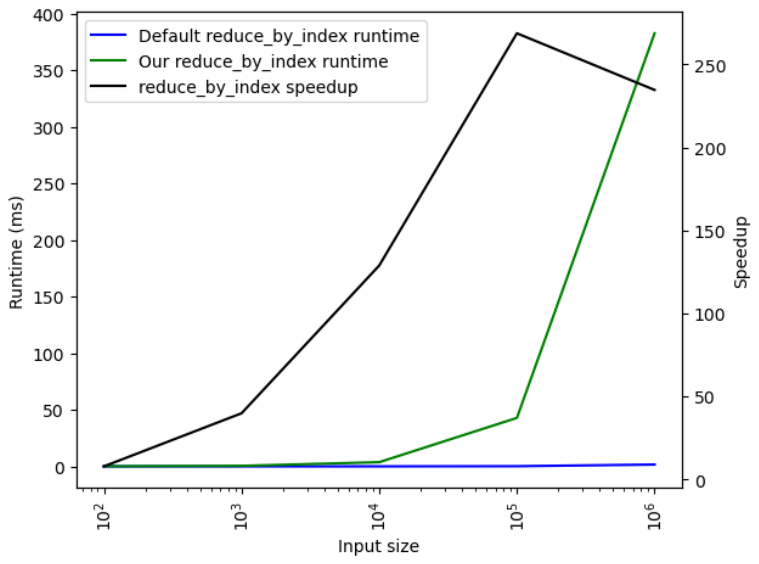
Exercise 2.3

When determining the asymptotic complexity in terms of work and span of the `reduce_by_index` implementation, we need to consider the complexity of its components, namely:

Function	Work	Span
`(un)zip`	"very cheap"	"very cheap"
`filter fst/snd`	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$
`scan`	$\mathcal{O}(n \times W(\text{operation}))$	$\mathcal{O}(\log(n) \times W(\text{operation}))$
`rotate`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`length`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`scatter`	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`mapN`	$\mathcal{O}(n \times W(\text{function}))$	$\mathcal{O}(S(\text{function}))$
`radix_sort_by_key`	$\mathcal{O}(k \times n)$	$\mathcal{O}(k \times \log(n))$

Source: futhark prelude github soacs, array, zip.

Given these complexities, we argue that the complexity of the implementation of `segreduce` is work complexity $\mathcal{O}(n \times W(\text{op}))$ and span complexity $\mathcal{O}(\log(n) \times W(\text{op}))$. From this we argue that the work complexity of the implemented `reduce_by_index` is $\mathcal{O}(k \times n \times W(\text{op}))$ and span complexity $\mathcal{O}(\max(\log(n) \times W(\text{op}),\ S(\text{op}))) = \mathcal{O}(\log(n) \times W(\text{op}) + S(\text{op}))$, where k is the number of bits in each element in the array sorted by `radix_sort_by_key`.



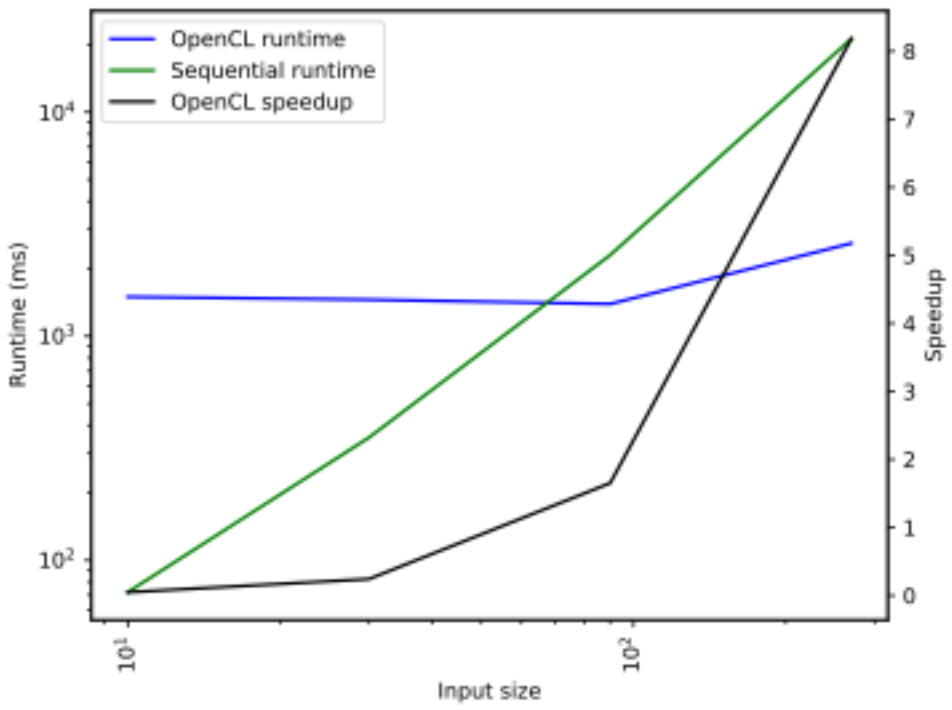
Our reduce*by*index is significantly worse than the built-in version. The built-in version's complexity for work is $O(n \times W(op))$ and for span it is $O(n \times W(op))$ according to the documentation.

Exercise 3: 2D Ising Model

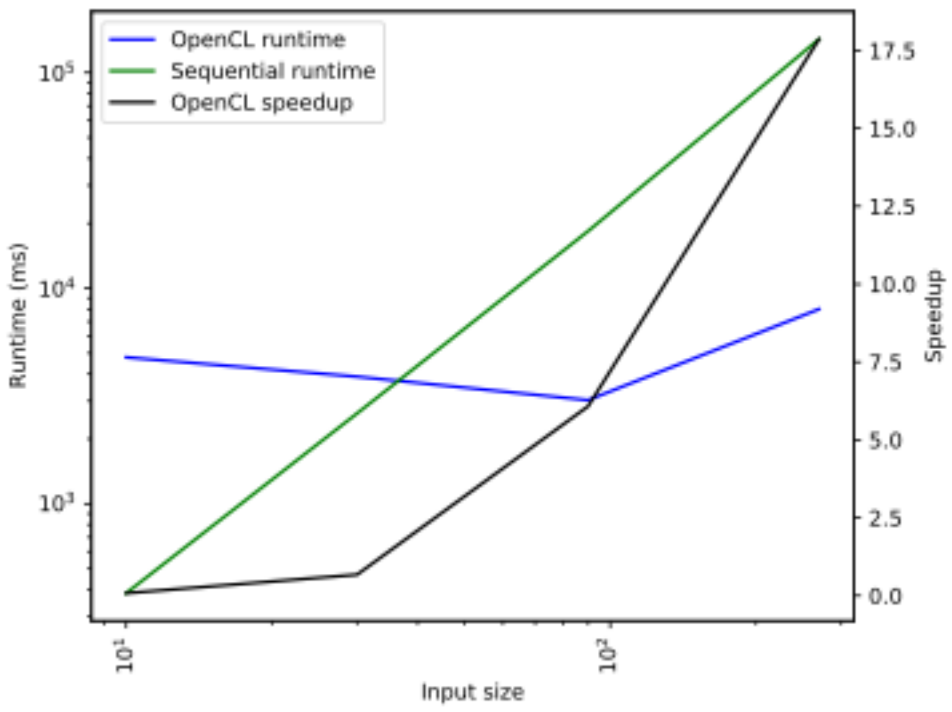
We ran the ising model benchmarks with four different sizes and three different number of iterations:

Size	Iterations	Time OpenCL [*μs*]	Time Sequential [*μs*]
10x10	2	1499	72
30x30	2	1456	353
90x90	2	1393	2303
270x270	2	2600	21281
10x10	20	4780	385
30x30	20	3888	2627
90x90	20	3026	18394
270x270	20	8002	142951
10x10	200	26983	2554
30x30	200	21323	22602
90x90	200	22088	155655
270x270	200	48805	1510192

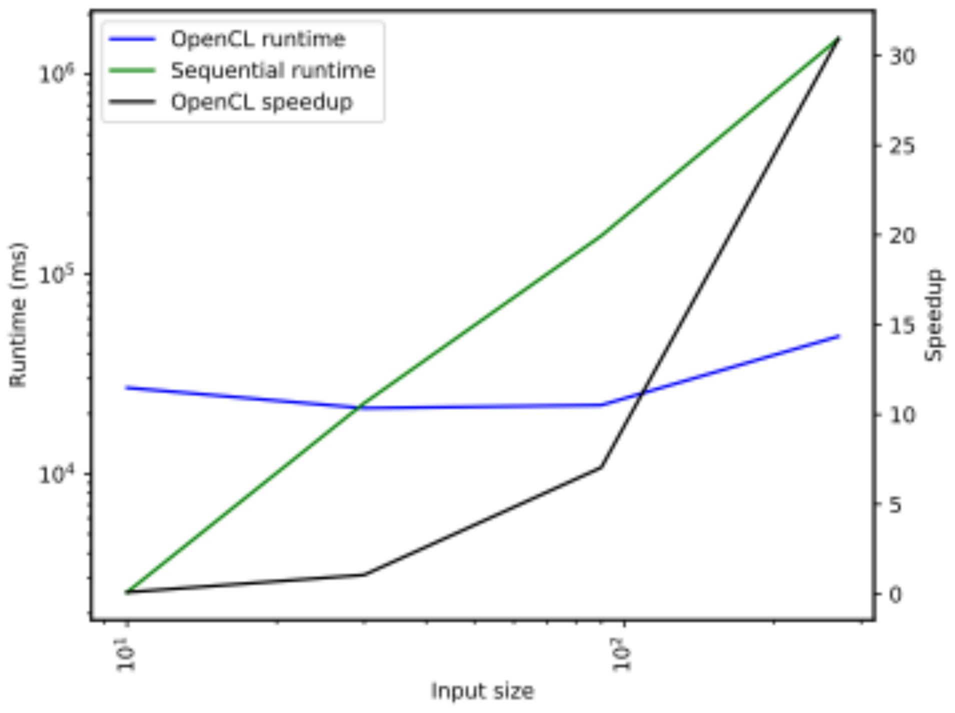
The computer used was a Macbook with an integrated GPU.



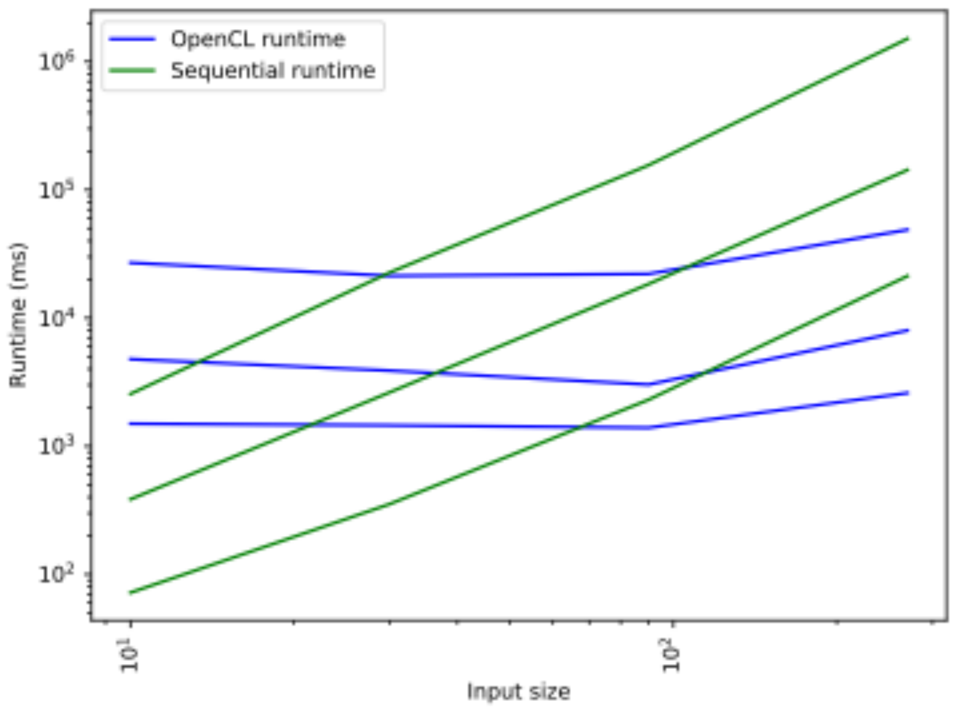
The first chart is for two iterations. As we can see, even with an integrated GPU the scaling for large problem sizes is impressive. There is a large overhead for using the GPU however.



The second chart is for 20 iterations. The pattern is similar to the one we saw with two iterations.



The third chart is for 200 iterations. The pattern is yet again similar to the one we saw with two iterations.



The final chart is showing the runtime across all number of iterations for all sizes. Interestingly, across all number of iterations the 10x10 model took longer to compute compared to both the 30x30 and 90x90 model when using OpenCL. OpenCL scales significantly better with regards to problem size for all number of iterations, but it also scales *slightly* better with regards to the number of iterations compared to the sequential implementation.