

DPP Weekly Assignment 1

Simon Vinding Brodersen(rzn875)

November 26, 2025

Note on benchmarking

Throughout this assignment all benchmarking was done using the Hendrix cluster with the NVIDIA GeForce GTX Titan X GPU. Further all testing was done using the builtin *futhark bench* command. I don't know the specifics of which CPU the batch jobs were run on, but if we assume these are the same as the default ssh machine, then it is: Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz. Each task is also run with 5 CPUS assigned and only a single GPU.

Further, whenever there is a speedup graph line, this is in reference to the baseline Sequential C runtime.

The programs can be run by entering the specific folders and running *futhark test <program_file>.fut* or for benchmarking running *futhark bench -backend=<desired backend> <program_file>.fut*. The batch scripts used to generate the graphs are also provided.

1 Getting started

1.1 Write a function

The program is a straight forward map-reduce composition.

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

And is tested via the following test cases:

```
-- Process tests
-- ==
-- entry: test_process
-- nobench input { [23 ,45 , -23 ,44 ,23 ,54 ,23 ,12 ,34 ,54 ,7 ,2 , 4 ,67]
--                  [ -2 , 3 , 4 ,57 ,34 , 2 , 5 ,56 ,56 , 3 ,3 ,5 ,77 ,89] }
-- output { 73 }
-- nobench input { empty([0] i32) empty([0] i32) }
-- output { 0 }
entry test_process = process
```

Which seems to work.

1.2 Benchmark your function

The benchmarking can be seen in Figure 1. The runtimes are reported via a log scale, where we can see that initially the GPU does not offer better performance. However, as the size of the input increases the GPU scales better and overtakes the sequential and multicore runtime.

Further, we can see that the multicore runtime is only slightly better when we increase the input size significantly and otherwise it has much the same performance as the baseline.

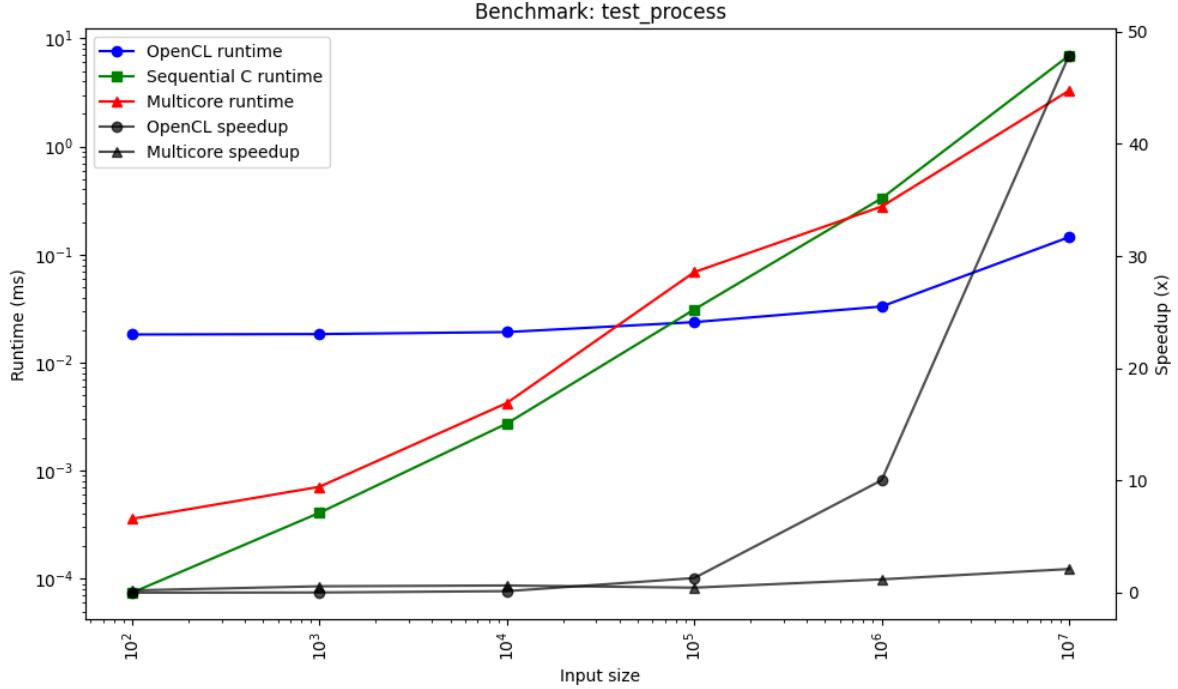


Figure 1: Runtime and speedup of the program written in Section 1.1

1.3 Extend your function

The definition of the test_process_idx is provided below

```
def process_idx [n] (xs: [n] i32) (ys: [n] i32) : (i32, i64) =
  let max (d1, i1) (d2, i2) =
    if d1 > d2 then (d1, i1)
    else if d1 < d2 then (d2, i2)
    else if i1 > i2 then (d1, i1)
    else (d2, i2)
  in reduce_comm max (0, -1)
    (zip
      (map i32.abs (map2 (-) xs ys))
      (iota n)
    )
```

The function implements a max function which is cumulative by using the indices for tie-breaking. This lets us use the faster reduce_comm command. The default reduce function optimises this already when we use a built in operator which is cumulative.

The function is then tested and benchmarked as follows:

```
-- Process idx tests
-- ==
-- entry: test_process_idx
-- nobench input { [23, 45, -23, 44, 23, 54, 23, 12, 34, 54, 7, 2, 4, 67]
--                  [-2, 3, 4, 57, 34, 2, 5, 56, 56, 3, 3, 5, 77, 89] }
-- output { 73 12i64 }
-- nobench input { empty([0] i32) empty([0] i32) }
-- output { 0 -1i64 }
-- notest random input { [100] i32 [100] i32 }
```

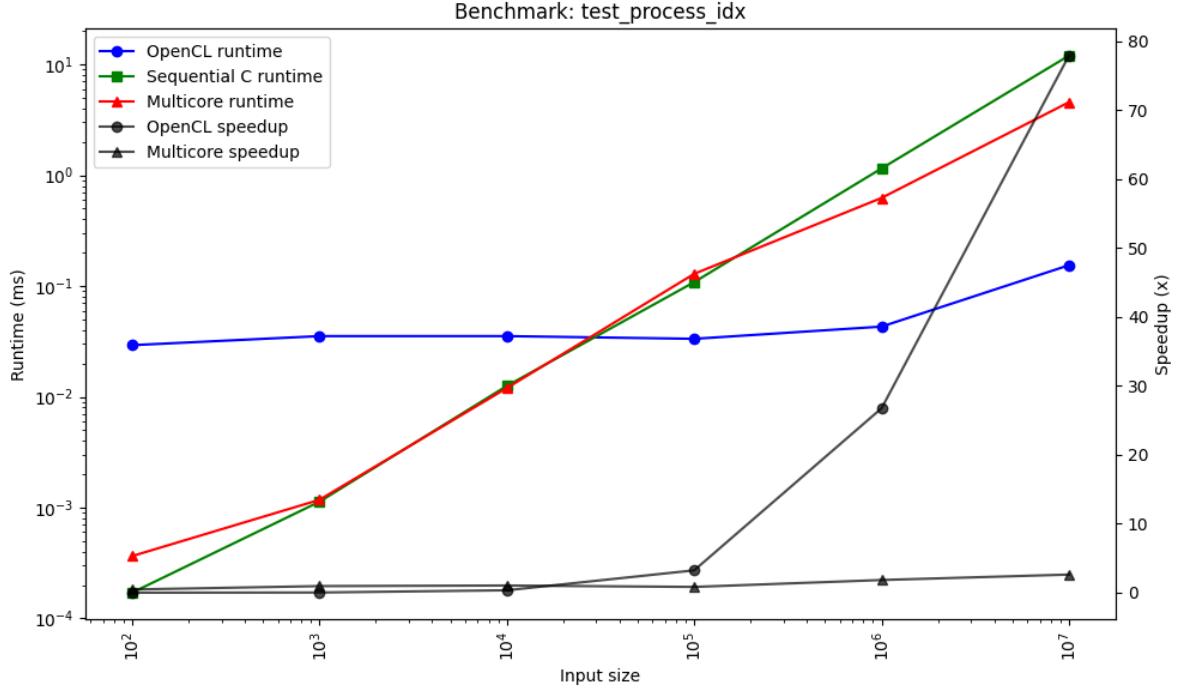


Figure 2: Runtime and speedup of the program written for Section 1.3

```
— notest random input { [1000]i32 [1000]i32 }
— notest random input { [10000]i32 [10000]i32 }
— notest random input { [100000]i32 [100000]i32 }
— notest random input { [1000000]i32 [1000000]i32 }
— notest random input { [10000000]i32 [10000000]i32 }
entry test_process_idx = process_idx
```

And this seems to work. The benchmarking can be seen in Figure 2, which provides the same story as in the previous section.

2 Implementing prefix sum

2.1 Hillis Steele

The hillis steele implementation can be seen below:

```
def hillis_steele [n] (xs: [n]i32) : [n]i32 =
  let m = ilog2 n
  in loop xs = copy xs for d in 0...m-1 do
    let pow_2 = 1 << d
    in map (\i ->
      if i - pow_2 >= 0 then
        xs[i-(pow_2)] + xs[i]
      else
        xs[i]) (iota n)
```

This works quite intuitively as we are assuming our input is some power of 2. First, we compute m via the provided ilog2 function. Then, in a loop given m we map over all the indices. If an index is less than the current 2^d , then we simply copy the previous iteration result and otherwise we add $xs[i - pow_2] + xs[i]$.

This was tested as follows:

```

-- Hillis test
-- ==
-- entry: test_hillis
-- nobench input { [0, 0, 1, 0, 0, 0, 0, 0] }
-- output { [0, 0, 1, 1, 1, 1, 1] }
-- nobench input { [0, 1, 1, 0, 0, 0, 0, 0] }
-- output { [0, 1, 2, 2, 2, 2, 2] }
-- nobench input { [3, 1, 7, 0, 4, 1, 6, 3] }
-- output { [3, 4, 11, 11, 15, 16, 22, 25] }
entry test_hillis = hillis_steele

```

And passes the provided tests.

2.2 Work efficient

The work efficient algorithm is provided below:

```

def work_efficient [n] (xs: [n]i32) : [n]i32 =
let m = ilog2 n
let upswept =
loop xs = copy xs for d in 0...m-1 do
    let stride = 1 << (d + 1)
    let offset = 1 << d
    let idxs = map (\i -> i + stride - 1) (0..stride...n-1)
    -- j = i + 2^(d + 1) - 1 => j - 2^d = i + 2^d - 1, from the Guy paper
    let vals = map (\j -> xs[j - offset] + xs[j]) idxs
    in scatter xs idxs vals
let upswept[n-1] = 0
let downswept =
loop xs = upswept for d in m-1..m-2...0 do
    let stride = 1 << (d + 1)
    let offset = 1 << d
    let (left_idxs, right_idxs) = map (\i ->
        let left_idx = i + offset - 1
        in (left_idx, left_idx + offset)) (0..stride...n-1) |> unzip
    let left_vals = map (\r -> xs[r]) right_idxs
    -- First item is t from paper
    let right_vals = map2 (\l r -> xs[l] + xs[r]) left_idxs right_idxs
    let all_idxs = left_idxs ++ right_idxs
    let all_vals = left_vals ++ right_vals
    in scatter xs all_idxs all_vals
in downswept

```

This algorithm is significantly more complex and attempts to closely follow the pseudocode provided in the provided material "Prefix Sums and Their Applications by Guy E. Blelloch".

The up-sweep first calculates the stride for the parallel computation along with the offset. First, we calculate the indices that we are updating via a map. Then given these indices we are able to calculate the values at that location and at last we can do a scatter into the original array.

The down-sweep works in somewhat the same fashion. We calculate the left- and right child indices in an initial map. Then, given these left and right child pairs, we calculate the values for the given positions. Here we are able to optimise some of the computation. For example instead of recomputing $i + 2^{d+1} - 1$ when finding the value for the left child, we instead realise this is simply the same index as the right child itself. This now gives us all the left child indices along with values and the same for the right child, so appending the two together we can do a final scatter into the original array.

The code is tested as shown below:

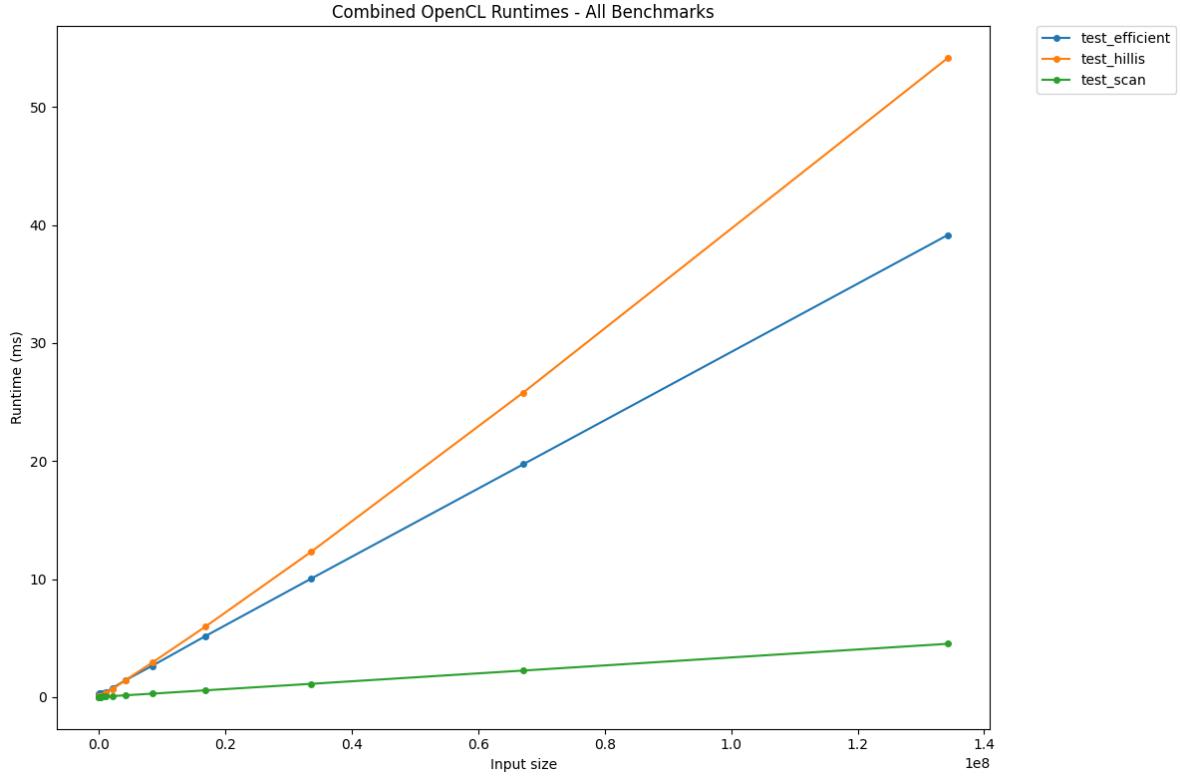


Figure 3: Benchmark comparison of Opencl runtimes.

```
-- Work efficient test
-- ==
-- entry: test_efficient
-- nobench input { [0, 0, 1, 0, 0, 0, 0, 0] }
-- output { [0, 0, 1, 1, 1, 1, 1, 1] }
-- nobench input { [0, 1, 1, 0, 0, 0, 0, 0] }
-- output { [0, 0, 1, 2, 2, 2, 2, 2] }
-- nobench input { [3, 1, 7, 0, 4, 1, 6, 3] }
-- output { [0, 3, 4, 11, 11, 15, 16, 22] }
entry test_efficient = work_efficient
```

And it passes these tests.

2.3 Benchmarking

The benchmarking can be seen in Figure 3. Here we can see that the built-in scan function is significantly faster than both implementations. However, as expected the work efficient algorithm does scale better than the hillis Steele implementation. This scaling does look like what we would expect given the asymptotic guarantees.

3 Segmented operations

3.1 Proof

We have:

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

$$= (0 \oplus v_2, f_1) \quad (2)$$

$$= (v_2, f_1) \quad (3)$$

As such, $(0, \text{false})$ is indeed the neutral element as 0 is the neutral element of \oplus

3.2 Segmented scans and reductions

The implementation of both the segmented scan and reduction can be seen below:

```
def segscan [n] 't (op: t -> t -> t) (ne: t)
    (arr: [n](t, bool)): [n]t =
let (res, _) = scan (\(v1, f1) (v2, f2) ->
    (if f2 then v2 else op v1 v2, f1 || f2)
    ) (ne, false) arr
|> unzip
in res

def segreduce [n] 't (op: t -> t -> t) (ne: t)
    (arr: [n](t, bool)): []t =
let res = segscan op ne arr
let (_, flags) = unzip arr
-- exclusive scan
let keep = map (\i ->
    if i == n-1 then 1
    else if flags[i + 1] then 1
    else 0
) (iota n)
let offsets1 = scan (+) 0 keep
in if (length offsets1) == 0
    then []
    else scatter (replicate (last offsets1) ne)
        (map2 (\i k -> if k == 1 then i-1 else -1) offsets1 keep) res
```

The segmented scan works by simply calling scan on our new found neutral element.

The segreduce works by calling the segscan function, and then finding all the items that we wish to keep. These are the items for which the next item is a start of an array. These are all the elements at the end of our segments. Then we can scan the keep array, which contains a 0 if we do not want to keep the item and a 1 if we wish to keep it, which gives us the offset into the final array of each of the keep elements. At last, we have to check if there are no elements we wish to keep in which case we simply return the empty array and otherwise we scatter into any array where we substitute all 0s with -1 (in this case it is instantiated with the neutral element, but as we overwrite each element this is not technically necessary.). This works because the builtin scatter function simply ignores all the -1 elements.

3.3 Benchmarking

The benchmarking can be seen in Figure 4. We can see that the extra work we do in the segreduce significantly worsens our runtime. The overhead of the segscan is not as large, but still noticeable. They all seem to be performing in linear time.

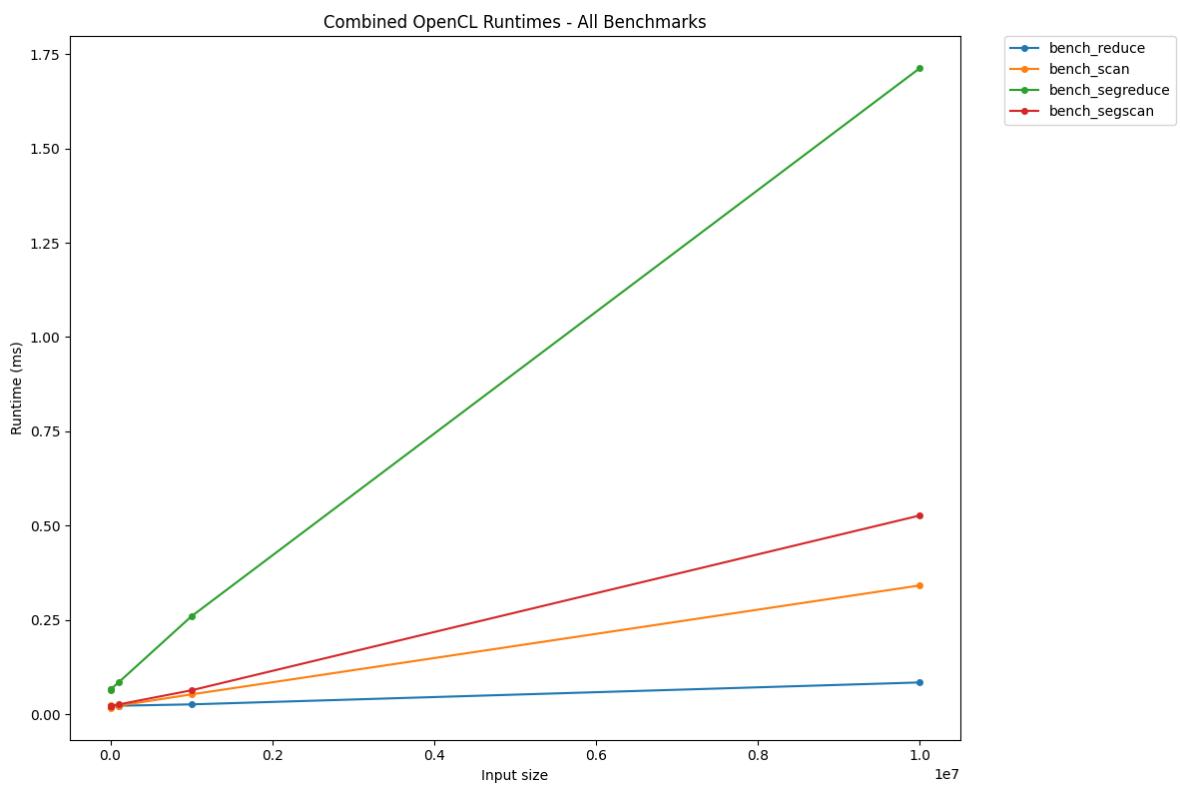


Figure 4: Benchmarking of the Segmented scan and reduce functions.

3.4 Reduce by index

The implementation for the reduce by index can be seen below:

```
def custom_reduce_by_index 'a [m] [n]  (dest: *[m] a)
    (f: a -> a -> a) (ne: a)
    (is: [n] i64) (as: [n] a) : *[m] a =
let (sorted_is, sorted_as) = radix_sort_int_by_key
    (\(i, _) -> i)
    i64.num_bits
    i64.get_bit
    (zip is as)
    |> unzip
let prev_is = rotate (-1) sorted_is
let flag_array = map2 (\i_curr i_prev -> i_curr != i_prev) sorted_is prev_is
let seg_res = zip sorted_as flag_array |> segreduce f ne
in if length seg_res >= m then map2 (\a b -> f a b) (take m seg_res) dest
else
let pad = replicate (m - length seg_res) ne
let inp = seg_res ++ pad
in map2 (\a b -> f a b) (inp :> [m] a) dest
```

The function works by first sorting the indices and values by the indices. Then we create the flag array by simply checking if the current index is equal to the previous index. We can then use the aforementioned segreduce function to compute the reduced sum of each of the segments. At last, we have to check if we have enough values to fill the entire dest array. If we do not, then we pad with the neutral elements and map this into the destination. One could argue whether or not we should add the neutral element or simply leave the dest value as is. However, I chose this solution as there is "no" sum when there is no elements that correspond to it.

The implementation was tested as follows:

```
-- ==
-- entry: test_reduce
-- nobench input { [0,0,0] [0i64, 1i64, 1i64, 2i64] [10, 10, 10, 10] }
-- output { true }
-- nobench input { [0,0,0]
--                 [0i64, 1i64, 1i64, 2i64, 2i64] [10, 10, 10, 10, 10] }
-- output { true }
-- nobench input { [0,0,0]
--                 [0i64, 1i64, 1i64, 2i64, 2i64] [10, 20, 30, 40, 50] }
-- output { true }
entry test_reduce dest is vs =
    let builtin = reduce_by_index (copy dest) (+) 0i32 is vs
    let custom = custom_reduce_by_index (copy dest) (+) 0i32 is vs
in foldl (&&) true <|
    map2 (\a b -> a == b) (builtin :> [3] i32) (custom :> [3] i32)
```

And we seem to get the same result as the default implementation in these cases.

3.4.1 Discussion and benchmarking

My algorithm first runs a radix sort algorithm, which runs in $O(k n)$ work and $O(k \log(n))$ span, where k is the number of bits in each element. Then it runs a rotate and map which both run in $O(n)$ work and $O(1)$ span before running a segreduce. The segreduce itself runs with work $O(n)$ and span $O(\log n)$ as it is dominated by the scan function. This means that the work and span is dominated by the sorting of $O(k n)$ and $O(k \log n)$.

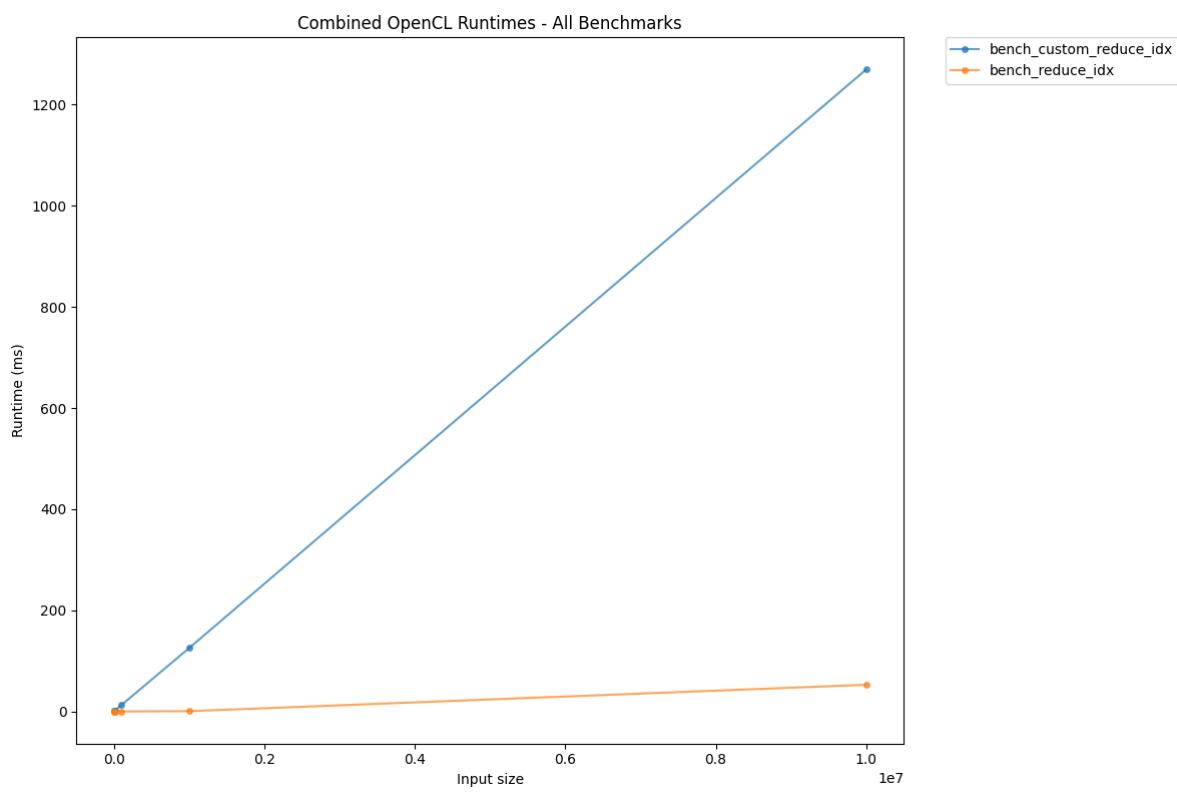


Figure 5: Benchmarking results for the reduce function

The benchmarking can be seen in Figure 5. Here we can see that my implementation is significantly slower than the built in version. While in other cases my implementation seemed to have the same time complexity this certainly does not seem to be the case. As such, I do not think this is optimal.

4 2D Ising Model

4.1 Generate initial state

The function for the random grid can be seen below

```
def random_grid (seed: i32) (h: i64) (w: i64)
    : ([h][w]rng_engine.rng, [h][w]spin) =
let initial = rng_engine.rng_from_seed [seed]
let states_flat = rng_engine.split_rng (h * w) initial
let (new_states, spins) = map (\r ->
    let (r, i) = rand_i8.rand (0, 1i8) r
    in if i == 0 then (r, -1) else (r, i)
) states_flat |> unzip
in (unflatten new_states, unflatten spins)
```

We generate the random state by splitting an initial state into $h * w$ new states. Then we can randomly generate -1 or 1, and return the new random states that are also generated by rand_i8.

4.2 Computing Delta

The code for computing the deltas can be seen below:

```
def deltas [h][w] (spins: [h][w]spin): [h][w]i8 =
let rot_down = rotate 1 spins |> flatten
let rot_up = rotate (-1) spins |> flatten
let rot_left = map (rotate (-1)) spins |> flatten
let rot_right = map (rotate (1)) spins |> flatten
-- for spins[i, j] then l is rot_right[i, j]
let res = map5 (\c u d l r -> 2 * c * (u + d + l + r))
    (flatten spins)
    rot_down
    rot_up
    rot_right
    rot_left
in unflatten res
```

Here we make use of the rotate to rotate in each of the different directions. Then using a single map with the 5 inputs we simply calculate the deltas via. the formula provided.

4.3 The step function

The step function is provided below:

```
def step [h][w] (abs_temp: f32) (samplerate: f32)
    (rngs: [h][w]rng_engine.rng) (spins: [h][w]spin)
    : ([h][w]rng_engine.rng, [h][w]spin) =
let delta_es = deltas spins |> flatten |> map (f32.i8)
let (new_states, new_spins) = map3 (\r1 c delta_e ->
    let neg_delta_e = - delta_e
    let (r2, a) = rand_f32.rand (0f32, 1f32) r1
    let (new_r, b) = rand_f32.rand (0f32, 1f32) r2
    let pow_e = f32.exp (neg_delta_e / abs_temp)
```

```

in
  if (a < samplerate) && ((delta_e < neg_delta_e) || (b < pow_e)) then
    (new_r, -c)
  else
    (new_r, c)
) (flatten rngs) (flatten spins) delta_es |> unzip
in (unflatten new_states, unflatten new_spins)

```

First we compute the deltas using the aforementioned function. Inside a map we then follow directly the calculations provided by generating random values a and b between 0 and 1. Here it is important to note, that b is created using the new random state returned after creating b and also the random state we are returning is an unused new random state returned from generating b.

4.4 Benchmarking

Table 1: Performance comparison of Ising model implementations with speedup relative to sequential baseline

Parameters			Runtime (μ s)			Speedup	
h	w	n	OpenCL	Sequential	Multicore	OpenCL	Multicore
100	10	2	64	113	2,354	1.77×	0.05×
1,000	10	2	67	1,284	2,603	19.16×	0.49×
10,000	10	2	92	13,624	6,671	148.09×	2.04×
100,000	10	2	158	127,852	50,458	809.19×	2.53×
10	10	2	67	11	1,187	0.16×	0.01×
10	100	2	63	106	2,333	1.68×	0.05×
10	1,000	2	66	1,173	1,795	17.77×	0.65×
10	10,000	2	85	11,843	8,752	139.33×	1.35×
10	100,000	2	150	121,386	71,967	809.24×	1.69×
10	10	10	229	56	1,379	0.24×	0.04×
10	10	100	2,099	549	1,042	0.26×	0.53×
10	10	1,000	19,803	5,103	8,057	0.26×	0.63×
10	10	10,000	199,459	50,514	77,982	0.25×	0.65×
10	10	100,000	1,979,167	508,566	739,818	0.26×	0.69×
10	10	1,000,000	19,547,227	4,968,729	7,307,114	0.25×	0.68×

The benchmarks can be seen in Table 1. Here we can see that when h and w increase, then the GPU parallelisation significantly increases the performance. However, when n increases we do not see the same speedup. This makes sense as we simply have a sequential loop over n.