

1 Task 1 (2 pts)

1.1 Task 1.a) Prove that a list-homomorphism induces a monoid structure (1pt)

Solution:

1. Associativity:

By definition of Img , for any x, y, z in $\text{Img}(h)$, there exist a, b , and c such that $x = h(a)$, $y = h(b)$, and $z = h(c)$.

We now have:

$$\begin{aligned}(x \circ y) \circ z &= ((h(a) \circ h(b)) \circ h(c)) \\ &= (h(a ++ b)) \circ h(c) \quad (\text{by definition of list homomorphism}) \\ &= h((a ++ b) ++ c) \\ &= h(a ++ (b ++ c)) \quad (\text{as } ++ \text{ is associative}) \\ &= h(a) \circ h(b ++ c) \\ &= h(a) \circ (h(b) \circ h(c)) \\ &= x \circ (y \circ z)\end{aligned}$$

2. Neutral element: Let $e = h([])$ be the neutral element. For any x in $\text{Img}(h)$, there exists an a in A such that $h(a) = x$.

$$\begin{aligned}e \circ x &= h([]) \circ h(a) = h([] ++ [a]) = h([a]) = x \\ x \circ e &= h(a) \circ h([]) = h([a] ++ []) = h([a]) = x\end{aligned}$$

We also need to show that exactly one such identity element exists. Assume there exist two neutral elements e and e' .

$$\begin{aligned}e \circ e' &= e \quad (\text{by definition of neutral element}) \\ e \circ e' &= e' \quad (\text{by definition of neutral element}) \\ e &= e' \quad (\text{by the previous equalities})\end{aligned}$$

Therefore, there is at most one neutral element.

1.2 Task 1.b) Prove the Optimized Map-Reduce Lemma (1pt)

Solution:

Let $id = (\text{reduce } (++) []) \circ \text{distr}_p$

We start by applying the identity:

$$\begin{aligned} & (\text{reduce } (+) 0) \circ (\text{map } f) \\ &= (\text{reduce } (+) 0) \circ (\text{map } f) \circ (\text{reduce } (++) []) \circ \text{distr}_p \end{aligned}$$

Now, we apply the three lemmas in sequence:

1. Apply lemma 2: $(\text{map } f) \circ (\text{reduce } (++) []) \equiv (\text{reduce } (++) []) \circ (\text{map } (\text{map } f))$
 $= (\text{reduce } (+) 0) \circ (\text{reduce } (++) []) \circ (\text{map } (\text{map } f)) \circ \text{distr}_p$
2. Apply lemma 3: $(\text{reduce } \odot e_\odot) \circ (\text{reduce } (++) []) \equiv (\text{reduce } \odot e_\odot) \circ (\text{map } (\text{reduce } \odot e_\odot))$

Here, we apply the lemma with $\odot = +$ and $e_\odot = 0$. The lemma transforms:

$$(\text{reduce } (+) 0) \circ (\text{reduce } (++) [])$$

into:

$$(\text{reduce } (+) 0) \circ (\text{map } (\text{reduce } (+) 0))$$

Thus, we get:

$$= (\text{reduce } (+) 0) \circ (\text{map } (\text{reduce } (+) 0)) \circ (\text{map } (\text{map } f)) \circ \text{distr}_p$$

3. Apply lemma 1: $(\text{map } f) \circ (\text{map } g) \equiv \text{map}(f \circ g)$

$$= (\text{reduce } (+) 0) \circ (\text{map } ((\text{reduce } (+) 0) \circ (\text{map } f))) \circ \text{distr}_p$$

This completes the proof of the Optimized Map-Reduce Lemma.

2 Task 2: Longest Satisfying Segment (LSS) Problem (3pts)

Solution:

1. LSSP Operator Implementation

```
let segments_connect =
  x_len == 0
  || y_len == 0
  || pred2 x_last y_first

let new_lss = max
  (max x_lss y_lss)
  (if segments_connect then x_lcs + y_lis else 0)
```

```

let new_lis = if segments_connect &&
  x_lis == x_len then x_lis + y_lis else x_lis
let new_lcs = if segments_connect &&
  y_lcs == y_len then x_lcs + y_lcs else y_lcs

let new_len = x_len + y_len

let new_first = if x_len == 0 then y_first else x_first
let new_last  = if y_len == 0 then x_last  else y_last

```

2. Inline Tests

The following inline tests were added to validate the program:

```

— Small dataset for sorted
— ==
— compiled input {
—   [1, -2, -2, 0, 0, 0, 0, 3, 4, -6, 1]
— }
— output {
—   9
— }
— This test checks for a mixed sequence with a long segment of zeros.
— compiled input {
—   [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
— }
— output {
—   10
— }
— This test checks for a completely sorted sequence.
—   [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
— }
— output {
—   1
— }

— Small dataset for Same
— ==
— compiled input {
—   [1, -2i32, -2i32, 2i32, 0i32, 0i32, 0i32, 3i32, 4i32, -6i32, 1i32]
— }
— output {
—   3i32
— }
— This test for the "Same" predicate checks a mixed sequence with repeated

```

```

—  compiled input {
—    [1i32, 0i32, 3i32, 3i32, 3i32, 3i32, 6i32, 7i32, 8i32, 9i32, 10i32]
—  }
—  output {
—    4i32
—  }
—  This test checks a sequence with a longer segment of identical elements.
—  compiled input {
—    [1i32, 0i32, 1i32, 0i32, 1i32, 0i32, 1i32, 0i32, 1i32, 1i32]
—  }
—  output {
—    2i32
—  }
—  This test checks an alternating sequence ending with two identical elements.

—  Small dataset for zeros
—  ==
—  entry: main
—
—  input { [0i32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1] }
—  output { 10i32 }
—  This test for the "zeros" predicate checks a sequence with a long run of zeros.
—  input { [1i32, -2, -2, 0, 0, 0, 3, 4, -6] }
—  output { 3i32 }
—  This test checks a mixed sequence with a shorter run of zeros.
—  input { [0i32, 1, 0, 0, 3, 0, 0, 4, 0] }
—  output { 2i32 }
—  This test checks a sequence with multiple short runs of zeros.
—  output { 2i32 }
—  This test checks a short sequence with a run of zeros in the middle.

```

3. Performance Comparison

Runtimes and speedups for the LSSP benchmarks on arrays of size 10^8 :

we generate the data as follows:

```

futhark dataset --i32-bounds=-10:10 -b -g [100000000]i32 > data.in
futhark c gen-lssp-same.fut && ./gen-lssp-same < data.in > data_same.out
futhark c gen-lssp-sorted.fut && ./gen-lssp-sorted < data.in > data_sorted.out
futhark c gen-lssp-zeros.fut && ./gen-lssp-zeros < data.in > data_zeros.out

```

3 Task 3: CUDA Exercise (3pts)

3.0.1 Kernel Code

```

__global__ void cuda_map(float* X, float* Y, int n) {

```

Benchmark	Parallel Runtime (us)	Speedup	Backend
lssp-zeros	135340	1.0x	C
lssp-zeros	2259	59.9x	CUDA
lssp-sorted	416340	1.0x	C
lssp-sorted	2226	187.0x	CUDA
lssp-same	155244	1.0x	C
lssp-same	2225.3	70.0x	CUDA

Table 1: Runtime comparison and speedup for LSSP benchmarks

```

const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i < n) {
    float x = X[i]; // Load the input element
    float temp = _fdividef(x, x - 2.3f);
    Y[i] = temp * temp * temp; // Avoid using pow()
}

```

3.0.2 Grid and Block Size Computation

```

int BlockSize = 1024;
int blocksPerGrid = (N + BlockSize - 1) / BlockSize;

```

3.0.3 Kernel Invocation

```

cuda_map<<<blocksPerGrid, BlockSize>>>(d_in, d_out, N);

```

3.0.4 Validation

```

validate<float>(h_out, h_out_seq, N, 0.000001);

```

3.1 Memory Throughput Analysis

- Peak bandwidth at initial length (753,411): 151 GB/s
- Maximum bandwidth achieved: 1,097.57 GB/s
- Array length for maximal throughput: $2^{25} = 33,554,432$
- Peak memory bandwidth of GPU hardware (A100 40GB): 1,555 GB/s

3.2 Conclusion

The implementation fulfills all criteria of the task. The CUDA version achieves a significant speedup over the CPU version and approaches the theoretical peak memory bandwidth of the GPU hardware when using larger array sizes.

4 Task 4: Flat Sparse-Matrix Vector Multiplication in Futhark (2pts)

4.1 Solution

4.1.1 Implementation

```
— entry: main
— input {
—   [0i64, 1i64, 0i64, 1i64, 2i64, 1i64, 2i64, 3i64, 2i64, 3i64, 3i64]
—   [2.0f32, -1.0f32, -1.0f32, 2.0f32, -1.0f32, -1.0f32, 2.0f32, -1.0f32, -1.0f32, 2.0f32, 2.0f32]
—   [2i64, 3i64, 3i64, 2i64, 1i64]
—   [2.0f32, 1.0f32, 0.0f32, 3.0f32]
— }
— output { [3.0f32, 0.0f32, -4.0f32, 6.0f32, 9.0f32] }
— input @ data.in
— output @ data.out

let spMatVctMult [num_elms][vct_len][num_rows]
  (mat_val: [num_elms](i64, f32))
  (mat_shp: [num_rows]i64)
  (vct: [vct_len]f32)
  : [num_rows]f32 =

  — Compute the flag array using the mkFlagArray function from the lecture notes
  — The flag array gives us a boolean array where true means that the element is non-zero
  let flag_arr = mkFlagArray mat_shp false (replicate num_rows true)

  — Cast the flag array to the type of the products array
  let typed_flag_arr = flag_arr :> [num_elms]bool

  — Map across the list of tuples, index into vec with
  — the first tuple element and multiply by second tuple element
  — This gives us the product of the matrix and the vector
  let products = map (\(ind, value) =>
    value * vct[ind]
  ) mat_val

  — Use the segmented scan to sum over the products within each row
  let scan_res = sgmSumF32 typed_flag_arr products

  — Get the indices of the last element of each row by doing a scan over the products
  — By summing we can compute the global index of the last element of each row
  let last_indices = scan (+) 0 mat_shp

  — Extract the last element of each segmented sum
```

```

let row_sums = map (
  \i -> if i == 0 then
    scan_res[i]
    else
      scan_res[i - 1]
) last_indices

in row_sums

```

4.1.2 Benchmarking

We benchmark on a dataset generated with:

```
futhark dataset --i64-bounds=0:9999 -g [1000000]i64 --f32-bounds=-7.0:7.0 -g [1000000]f32 --
```

The sequential version takes 1676 μ s while the flat version takes 202 μ s, which is a speedup of approximately 8x.