1 Task 1 (2 pts)

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

Assume you have a well-defined list-homomorphic program $h : A \to B$, i.e., all list splitting into $x + +y$ give the same result:

1. $h[] = e$
2. $h[a] = fa$
3. $h(x + +y) = (hx) \circ (hy)$

where $\circ$ is the binary operator of the homomorphism (and does **not** denote function composition). Assume also that you may apply the third definition as well as the second one for the case when the input is a one-element list. For example, the following derivation is legal:
$h[a] = h([] + +[a]) = (h[]) \circ (h[a]) = (h[]) \circ (fa)$.

Your task is to prove that $(\text{Img}(h), \circ)$ is a monoid with neutral element e, i.e., prove that $\circ$ is associative and that e is neutral element:

- prove that for any x, y, z in $\text{Img}(h)$, $(x \circ y) \circ z = x \circ (y \circ z)$ (**associativity**)
- prove that for all x in $\text{Img}(h)$, $x \circ e = e \circ x = x$ (**neutral element**)

Notation and Hint: If $h : A \to B$, then $\text{Img}(h) = \{h(a) | \forall a \in A\}$ is the subset of B formed by applying h to all elements of A. It follows that for any x in $\text{Img}(h)$, there exists an a in A such that $h(a) = x$. The solution is short, about 6-to-8 lines.

For example, for associativity, you can start by stating that, by definition of Img, for any x, y, z in $\text{Img}(h)$, there exist a, b and c such that $x = ha$, $y = hb$ and $z = hc$. Then write a suite of equalities that takes you from the left-hand side of the equality (ultimately) to its right-hand side:
$(x \circ y) \circ z = ((ha) \circ (hb)) \circ (hc) = \ldots$ continue equality-based rewriting $\ldots = x \circ (y \circ z)$

Solution:

1. associativity

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

   we now have

   $$(x \circ y) \circ z = ((ha) \circ (hb)) \circ (hc)$$

   we know from the definition of list homomorphism that

$$(ha) \circ (hb) = h(a + +b)$$

so we can rewrite the expression as

$$(h(a + +b)) \circ (hc) = h((a + +b) + +c)$$

as ++ is associative, we can further rewrite the expression as

$$h((a + +(b + +c)) = h(a + +(b + +c))$$

we can now reverse out computation

$$h(a + +(b + +c)) = h(a) \circ h(b + +c) = h(a) \circ (h(b) \circ h(c)) = x \circ (y \circ z)$$

2. neutral element we have the following:
   $h[a] = h([] + +[a]) = (h[]) \circ (h[a]) = (h[]) \circ (fa)$.
   let $e = h[]$ be the neutral element. for any x in Img(h), there exists an a in A such that $h(a) = x$.
   $e \circ x = h[] \circ h(a) = h([] + +[a]) = h([a]) = x$
   $x \circ e = h(a) \circ h[] = h([a] + +[]) = h([a]) = x$
   we need to also show that exactly one such identity element exists
   assume there exists two neutral elements e and e'.
   $e \circ e' = e$ by the definition of the neutral element.
   $e \circ e' = e'$ by the definition of the neutral element.
   $e = e'$ by the previous equalities.
   therefore, there is at most one neutral element.

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

This task refers to the List Homomorphism Promotion Lemmas, which were presented in the first lecture and can be found in the lecture notes at page 17-19 inside Section 2.4.2 entitled "Other List-Homomorphism Lemmas". **In the following $\circ$ denotes function composition**.

Your task is to use the three List Homomorphism Promotion Lemmas to prove the following invariant (Theorem 4 in lecture notes):

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

where distr_p distributes the original list into a list of p sublists, each sublist having about the same number of elements, and where $\circ$ denotes the operator for function composition. Include the solution in the written (text) report.

Big Hint: Please observe that (reduce (++) []) $\circ \text{distr}_p$ = id where id is the identity function, meaning:

reduce (++) [] (distr_p x) = x, for any list x

So you should probably start by composing the identity at the end of the first (left) term and then apply the rewrite rules that match until you get the second (right) term of the equality:

(reduce (+) 0)$\circ$(map f) = (reduce (+) 0)$\circ$(map ((reduce (+) 0)$\circ$(map f)))$\circ$ distr_p

You should be done deducing the second term of the identity after three steps, each applying a different lemma. Should be a short solution!

Solution:

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

(reduce (+) 0)$\circ$(map f) = (reduce (+) 0)$\circ$(map ((reduce (+) 0)$\circ$(map f)))$\circ$ 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: (map f)$\circ$(reduce (++) []) $\equiv$ (reduce (++) [])$\circ$(map (map f))

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

2. Apply lemma 3: (reduce $\odot$ $e_\odot$) $\circ$ (reduce (++) []) $\equiv$ (reduce $\odot$ $e_\odot$) $\circ$ (map (reduce $\odot$ $e_\odot$))

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

   (reduce (+) 0) $\circ$ (reduce (++) [])

   into:

   (reduce (+) 0) $\circ$ (map (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: (map f) $\circ$ (map g) $\equiv$ map($f \circ g$)

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

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

Your task is to fill in the dots in the implementation of the LSS problem. Please see lecture slides or/and Sections 2.5.2 and 2.5.3 in lecture notes, pages 20-21. The handed-in code provides three programs (`lssp-same.fut`, `lssp-zeros.fut`, `lssp-sorted.fut`), for the cases in which the predicate is `same`, `zeros`, and `sorted` (in subfolder `lssp`). They all call the `lssp` function with its own predicate; the `lssp` (generic) function is for you to implement in the `lssp.fut` file. A generic sequential implementation is also provided in file `lssp-seq.fut`; you may test it by calling `lssp_seq` instead of `lssp` (and by commenting out the `import "lssp"`) in each of the three files—but then remember to compile with `futhark c`, otherwise it will be very slow. Your task is to

- implement the LSS problem in Futhark by filing in the missing lines in file `lssp.fut`.
  - code solution

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

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

  - add one or more small datasets—reference input and output directly in the main files `lssp-same.fut`, `lssp-zeros.fut`, `lssp-sorted.fut`—for each predicate (zeros, sorted, same) and make sure that your program validates on all three predicates by running `futhark test --backend=cuda lssp-sorted.fut` and so on.

    we write the test and confirm all 3 predicates work. This can be tested by running "bash validate.bash"
  - add a couple of larger datasets and automatically benchmark the sequential and parallel version of the code, e.g., by using `futhark bench --backend=c ...` and `futhark bench --backend=cuda ...`, respectively. (Improved sequential runtime `--backend=c` can be achieved when using the function `lssp_seq` instead of `lssp`, but it is not mandatory.) Report the runtimes and the speedup achieved by GPU acceleration. Several ways of integrating datasets directly in the Futhark program are demonstrated in github file `HelperCode/Lect-1-LH/mssp.fut`

  - submit your code `lssp.fut` (together with all the other provided code, so that your TAs do not have to move files around).

- include in your written report (1) your solution, i.e., the five missing lines in Section 2.5.3 of lecture notes, (2) the validation tests you added and (3) the runtimes and speedups obtained in comparison with the sequential implementation.

Solution:

1

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

let new_lss = max (max x_lss y_lss) (if segments_connect then x_lcs

let new_lis = if segments_connect then x_lis + y_lis else x_lis
let new_lcs = if segments_connect 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
 in (new_lss, new_lis, new_lcs, new_len, new_first, new_last)
```

2

the following inline tests were added to validate the program:

```
-- Small dataset for sorted
-- ==
-- compiled input {
--     [1, -2, -2, 0, 0, 0, 0, 0, 3, 4, -6, 1]
-- }
-- output {
--     9
-- }
-- compiled input {
--     [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
-- }
-- output {
--     10
-- }
-- compiled input {
--     [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
-- }
-- compiled input {
--     [1i32, 0i32, 3i32, 3i32, 3i32, 3i32, 6i32, 7i32, 8i32, 9i32, 10i3
-- }
-- output {
--     4i32
-- }
-- compiled input {
--     [1i32, 0i32, 1i32, 0i32, 1i32, 0i32, 1i32, 0i32, 1i32, 1i32]
-- }
-- output {
--     2i32
-- }

-- Small dataset for zeros


-- ==
-- entry: main
--
-- input { [0i32,0, 0, 0, 0, 0, 0, 0, 0, 0, 1] }
-- output { 10i32 }
--
-- input { [1i32, -2, -2, 0, 0, 0, 3, 4, -6] }
-- output { 3i32 }
--
-- input { [0i32, 1, 0, 0 ,3, 0, 0, 4, 0] }
-- output { 2i32 }
--
-- input { [0i32, 1, 0, 0, 1]}
-- output { 2i32 }
```

3

Runtimes and speedups:

Benchmark	Sequential Runtime	Parallel Runtime	Speedup
lssp-zeros	1676us	202us	8.3x
lssp-sorted	–	–	–
lssp-same	–	–	–

Table 1: Runtime comparison and speedup for LSSP benchmarks

Note: The values shown are placeholder values. Please replace them with the actual measured values for each benchmark.

Redundant Observation: We have also included a sequential generic version of LSSP, its implementation is in file `lssp-seq.fut`. You may enable it from any instances, for example by uncommenting in file `lssp-same.fut` the line `lssp_seq pred1 pred2 xs` (and commenting the other one). However, if you do that, then compile with the `c` backend (`cuda` will take forever because you are running a sequential program).

3 Task 3: CUDA exercise (3pts)

This task involves writing a CUDA program that implements two functions to map the function $f(x) = (\frac{x}{x-2.3})^3$ to an array of size 753411. The program should include both a serial CPU implementation and a parallel GPU implementation.

3.1 Program Requirements

- Implement two functions:
  1. A serial map performed on the CPU
  2. A parallel map in CUDA performed on the GPU
- Use single precision float
- Apply the function to the array [1, ..., 753411]
- Validate results: Check that CPU and GPU results are equal (within an epsilon error)
- Print "VALID" or "INVALID" based on the validation
- Report:
  1. Runtimes of CUDA vs CPU-sequential implementation
  2. Acceleration speedup

3. Memory throughput (GB/sec) of the CUDA implementation

- Determine the maximal memory throughput by increasing array size

3.2 Measurement Guidelines

- For CUDA runtime, exclude CPU-to-GPU transfer and GPU memory allocation time
- Call the CUDA kernel in a loop (e.g., 300 iterations)
- Use `cudaDeviceSynchronize()` after the loop
- Measure time before the loop and after `cudaDeviceSynchronize()`
- Report average kernel time (total time divided by loop count)

3.3 Submission Requirements

- Submit:
  1. Program file: `wa1-task3.cu`
  2. Makefile for the program
- Report should include:
  1. Validation result and epsilon used
  2. CUDA kernel code and how it was called
  3. Computation of grid and block sizes
  4. Array length for maximal throughput
  5. Memory throughput for maximal length and initial length (753411)
  6. Peak memory bandwidth of GPU hardware (if not using dedicated servers)

1. Runtimes of CUDA vs CPU-sequential implementation
2. Acceleration speedup
3. Memory throughput (GB/sec) of the CUDA implementation
4. Determine the maximal memory throughput by increasing array size
5. Validation result and epsilon used
6. CUDA kernel code and how it was called
7. Computation of grid and block sizes
8. Array length for maximal throughput
9. Memory throughput for maximal length and initial length (753411)
10. Peak memory bandwidth of GPU hardware (if not using dedicated servers)

Solution

Performance comparison

we benchmark the cuda kernel 300 times and take the average performance.

1. Runtime: serial map: 316.00 microseconds cuda map: 3.97 microseconds 2. speedup: 316.00 microseconds / 3.97 microseconds 80.00x 3. memory throughput of cuda kernel: 151 GB/s

Kernel code

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

        // we do this to avoid pow
        Y[i] = temp * temp * temp;
    }
}
```

we compute the grid size and block size as follows:

```
// Calculate the grid and block dimensions
int BlockSize = 1024;
int blocksPerGrid = (N + BlockSize - 1) / BlockSize;
```

we call the kernel

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

Validation

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

we set $\epsilon = 10^{-6}$

the validate function compares the output of the cuda kernel and the serial map and prints "VALID" if they are equal within the epsilon error.

Exploring maximal memory throughput

peak bandwidth at Length = 753411: 151 GB/s

to find the peak bandwidth we do a grid search over large array sizes that are powers of 2 as kernels work best when inputs are powers of 2.

we found the peak bandwidth at Length = 2**25 = 33554432, the peak bandwidth is 1097.57 GB/s which is close to the peak memory bandwidth for an a100 40gb gpu which is 1555 GB/s.

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

This task involves writing a flat-parallel version of sparse-matrix vector multiplication in Futhark. Refer to Section 3.2.4 "Sparse-Matrix Vector Multiplication" in the lecture notes (pages 40-41) and potentially rewrite rule 5 in Section 4.1.6 "Flattening a Reduce Directly Nested in a Map".

The sequential version of the code is provided as `spMVmult-seq.fut`. It can be compiled and run with:

```
$ futhark test --backend=c spMVmult-seq.fut
$ futhark c spMVmult-seq.fut
$ futhark dataset --i64-bounds=0:9999 -g [1000000]i64 --f32-bounds=-7.0:7.0 -g [1000000]f32
```

Your task is to implement a flat-parallel version in the file `spMVmult-flat.fut`, specifically in the function `spMatVctMult`. The current implementation is a dummy and needs to be replaced.

4.1 Requirements

1. Implement the flat-parallel version in `spMVmult-flat.fut`.
2. Add at least one more standard reference input/output dataset to the source file.
3. Measure and report the speedup compared to the sequential version.

The parallel version can be tested and run with:

```
$ futhark test --backend=cuda spMVmult-flat.fut
$ futhark cuda spMVmult-flat.fut
$ futhark dataset --i64-bounds=0:9999 -g [1000000]i64 --f32-bounds=-7.0:7.0 -g [1000000]f32
```

4.2 Implementation Notes

- You need to compute the flag array for a given shape array. Assume all entries in the shape array are greater than zero (no empty rows).
- You may use the `mkFlagArray` function if needed (see lecture notes, chapter 4, page 48, or `HelperCode/Lect-2-Flat/mk-flag-array.fut`).
- Be aware of Futhark's sized types. You may need to cast arrays using the `:>` operator.

4.3 Submission Requirements

Submit the following:

1. The implemented and tested `spMVmult-flat.fut` file.
2. In the written report, include:
   - The flat-parallel implementation of the `spMatVctMult` function with a brief explanation of each line's purpose.
   - The speedup of your accelerated version compared to `spMVmult-seq.fut` on a sufficiently large dataset.

Solution

implementation

```
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 =

    -- we compute the flag array using the function mkFlagArray from the lecture
    -- the flag array gives us a boolean array where true means that the element
    let flag_arr = mkFlagArray mat_shp false (replicate num_rows true)

    -- we cast the flag array to the type of the products array
    let typed_flag_arr = flag_arr :> [num_elms]bool

    -- we 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

    -- we 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 by doing a scan over the row
    -- by summing we can compute the global index of the last element of each ro
    let last_indices = scan (+) 0 mat_shp

    -- We 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
```

benchmarking

we benchmark on a dataset

```
futhark dataset --i64-bounds=0:9999 -g [1000000]i64 --f32-bounds=-7.0:7.0 -g [10
```

the sequential version takes 1676us the flat version takes 202us. which is a speedup of 8x.