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 (++) []) $\circ$ 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 (++)[] $\circ$ 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_las

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_li
let new_lcs = if segments_connect then x_lcs + y_lcs else y_lc
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.

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)

3.4 Implementation Notes

A similar task is discussed in the first Lab slides. The github folder `HelperCode/Lab-1-Cuda` contains a naive CUDA implementation for multiplying each element of an array by two, which works for arrays smaller than 1025 elements. The code in the slides generalizes this implementation to work for arbitrary sizes and includes time instrumentation and validation.