First Weekly Assignment for the PMPH Course

This is the text of the first weekly assignment for the DIKU course "Programming Massively Parallel Hardware", 2023-2024.

Hand in your solution in the form of a short report in text or PDF format, along with whatever source-code files are mandated by specific subtasks. All code files should be zipped together into an archive named `wa1-code.zip`.

For task 3, you are supposed to write yourself a CUDA program in a file named `wa1-task3.cu` and a `Makefile` for it. For the other (`Futhark`) programming tasks (i.e., tasks 2 and 4), you are supposed to fill in the missing parts in the code such that all the tests are valid (and to report some performance results for larger datasets).

With the exception of task 3, please send back the same files under the same structure that was handed in, i.e., please implement the missing parts directly in the provided files. There are comments in the source files that are supposed to guide you (together with the text of this assignment).

Unziping the handed in archive `wa1-code-handin.tar.gz` will create the `w1-code-handin` folder, which contains (i) files `spMVmult-flat.fut` and `spMVmult-seq.fut`---which correspond to sparse-matrix vector multiplication problem, and (ii) subfolder `lssp`, which contain the files corresponding to the longest-satisfying segment problem.

Task 1 (2 pts):

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

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

```
1. h []  = e
2. h [a] = f a
3. h (x ++ y) = (h x) o (h y)
```

where `o` 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 []) o (h [a]) = (h []) o (f a)`.

Your task is to prove that `( Img(h), o )` is a monoid with neutral element `e`, i.e., prove that `o` is associative and that `e` is neutral element:

- prove that for any `a, b, c` in `Img(h)`, `(a o b) o c = a o (b o c)` (**associativity**)
- prove that for all `b` in `Img(h)`, `b o e = e o b = b` (**neutral element**)

Notation and Hint: If `h : A → B`, then `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 `Img(h)`, there exists an `a` in `A` such that `h(a) = x`. The solution is short, about 6-to-8 lines.

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".

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

```
(reduce (+) 0) . (map f)

            ==

(reduce (+) 0) . (map ( (reduce (+) 0) . (map f) ) ) . 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 `.` denotes the operator for function composition. Include the solution in the written (text) report.

Big Hint: Please observe that `(reduce (++) []) . distr_p = id` where `id` is the identity function:

```
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) . (map f) .  (reduce (++) []) . 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!

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`.
  - 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=opencl lssp-sorted.fut` and so on.
  - compile the program with acceleration `futhark opencl lssp-sorted.fut` and without acceleration (sequential C) `futhark c lssp-sorted.fut` and report the speedup of the accelerated version by running both on a large dataset, for example with the command: `futhark dataset --i32-bounds=-10:10 -b -g [10000000]i32 | lssp-sorted -t /dev/stderr -r 10`
  - 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 (text) report the speedups obtained in comparison with the sequential implementation and the validation tests you added, together with your solution, i.e., the missing five lines in Section 2.5.3 of lecture notes.

Observation: I have included also 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 (`opencl` will take forever because you are running a sequential program).

Task 3: CUDA exercise, see lab 1 slides: Lab1-CudaIntro (3pts)

Write a CUDA program with two functions that both map the function (x/(x-2.3))^3 to the array [1,...,753411], i.e., of size 753411. The first function should implement a serial map performed on the CPU; the second function should implement a parallel map in CUDA performed on the GPU. Check that the result on CPU is equal to the result on GPU (modulo an epsilon error, e.g., `fabs(cpu_res[i] - gpu-res[i]) < 0.0001` for all `i`), and print a VALID or INVALID message. Also print the runtime (excluding the time for CPU-to-GPU transfer and GPU memory allocation) taken by the sequential and CUDA implementations.

Measure the time taken for both functions and write a max 5 line explanation why one is better than the other:

- Play with the size of the array and find out where is the sweet-point when the GPU starts being faster than the CPU
- Also increase the size of the array to determine what is roughly the maximal speedup.
- When you measure the GPU time:

- Call the CUDA kernel (repeatedly) inside a loop of some non-trivial count, say 300.
- After the loop, please place a `cudaDeviceSynchronize();` statement.
- Measure the time **before** entering the loop and **after** the `cudaDeviceSynchronize();` --- the latter ensures that all Cuda kernels have actually finished execution --- than report the average kernel time, i.e., divide by the loop count.

Please submit:

- your program named `wa1-task3.cu` together with a `Makefile` for it.
- write in your written report:
  - whether it validates (and what epsilon have you used for validating the CPU to GPU results)
  - the 5-line explanation of the speedups
  - the code of your CUDA kernel together with how it was called, including the code for the computation of the grid and block sizes.

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

This task refers to writing a flat-parallel version of sparse-matrix vector multiplication in Futhark. Take a look at Section 3.2.4 `Sparse-Matrix Vector Multiplication'' in lecture notes, page 40-41 (and potentially also at rewrite rule 5 in Section 4.1.6` Flattening a Reduce Directly Nested in a Map'' in lecture notes). The sequential version of the code is attached as `spMVmult-seq.fut`, can be compiled with `futhark c spMVmult-seq.fut` 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 --i64-bounds=100:100 -g [10000]i64 --f32-bounds=-10.0:10.0 -g [10000]f32
| ./spMVmult-seq -t /dev/stderr -r 10 > /dev/null
```

However, your task is to fill in a flat-parallel implementation in file `spMVmult-flat.fut`, function `spMatVctMult`, which currently contains a dummy implementation. Add at least one more standard reference input/output dataset to the source file and measure speedup with respect to the sequential version. The parallel version, once implemented can be tested with

```
$ futhark test --backend=cuda spMVmult-flat.fut
```

and bigger datasets can be generated and run with something like:

```
$ futhark cuda spMVmult-flat.fut

$ futhark dataset --i64-bounds=0:9999 -g [1000000]i64 --f32-bounds=-7.0:7.0 -g
[1000000]f32 --i64-bounds=100:100 -g [10000]i64 --f32-bounds=-10.0:10.0 -g [10000]f32
| ./spMVmult-flat -t /dev/stderr -r 10 > /dev/null
```

The former command will create (see also `main` function in file `spMVmult-flat.fut`):

- the sparse matrix corresponding to the `mat_inds` and `mat_vals` flat arrays of length one million elements consisting of indices in the `[0⋯9999]` range and float values in `[-7.0, 7.0]` range, respectively,
- the shape array `shp` of length ten thousands having all values equal to one hundred,
- the vector `vct` of length ten thousands --- which fits the indices stored in `mat_inds`.
- hence the dense array would have size `10000 x 10000` but it is sparse, so each row contains only `100` non-zero elements. Of course, your implementation should work with irregular matrices, i.e., in which rows have different length of non-zero elements.

One of the necessary steps for fulfilling the task is to compute the flag array for a given shape array. For simplicity you may assume that all the entries of the shape array have values greater than zero, i.e., no empty rows. If you cannot figure it out how to compute the flag array you may use the `mkFlagArray` function, which is shown in lecture notes, chapter 4 (page 48) and is also implemented in Futhark in `futhark-code/L2/mk-flag-array.fut`.

However, please keep in mind that Futhark is using sized types, hence you might need to (dynamically) cast the array obtained by `mkFlagArray` to the expected size/length with the `:>` operator. For example, if `xs0` is an array of single-precision floats (`f32`), and you know that its size should be `n` then writing something like `let xs = xs0 :> [n]f32` will create an aliased array `xs` which the compiler knows to be of type `[n]f32`.

Please submit:

- the `spMVmult-flat.fut` file once implemented and tested.
- In the written (text) report add:
  - the flat-parallel implementation of the `spMatVctMult` function and a short explanation of what each line is doing.
  - a short explanation about the speedup of your accelerated version in comparison with `spMVmult-seq.fut`