Contents

1 Task 1

1.1 Subtask 1

For subtask 1 we look at the following list homomorphism:

```
h []  = e
h [a] = f a
h (x ++ y) = (h x) o (h y)
```

1.1.1 Proving associativity

I aim to prove that for any `[a,b,c]` in `Img(h)` the following holds: `(a o b) o c = a o (b o c)`

I pick `(x,y,z)` and such that `(h x) = a, (h y) = b, (h z) = c` I look at the third statement and state:

```
h( (h (x .. y) .. h(z)) = (h(x) o (h y)) o (h z) = (a o b) o c
```

Then i calculate it in "reverse" and say

```
h x .. (h (y .. z)) = h(x) o (h(y) o h(z)) = a o (b o c)
```

Furthermore we know that however we split our input lists `(x .. (y .. x)` or `((x .. y) .. z`, our program gives the same result (e.g. it is a well-defined homomorphism). Therefore it is valid to say

```
h (h (x ++ y) ++ z) = h( x ++ (h y ++ h z))
=>
(a o b) o c = a o (b o c)
```

1.1.2 Proving the neutral element

I then want to prove that for all `b` in `Img(h)`, `b o e = e o b = b`

For every `b in Img(h)` there exists an a in `A` s.t. `h(a) = b` We have from the assignment text(and the definition of a LH) that

```
h([a]) = h([] ++ [a]) = h([a] ++ [])
=>
h (a) = h ([]) o h (a) = h(a) o h ([])
=>(knowing h [] = e)
b = b o e = e o b
```

1.2 Subtask 2

I want to prove the following invariant:

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

We know that for a list `x` it holds:

```
reduce (..) [] distr_p x == x
```

And we can use this on the lhs of the invariant, and draw the following conclusion

```
(reduce (+) 0) . (map f)
=>
(reduce (+) 0) . (map f) . (reduce (..) []) distr_p
=> (3rd promotion lemma)
(reduce (+) 0) . (map(reduce(+)0)) . map(map f)) . distr_p
=> (1st promotion lemma)
(reduce (+) 0) . (map ( (reduce (+) 0) . (map f) ) ) . distr_p
```

2 Task 2

2.1 Feedback:

2.1.1 Also, please explain and discuss how you execute your benchmarks: Which test

cases you include, which GPU you run them on, and so on. If you decide to resubmit, please include this.

For the benchmarks described later in this section, all tests were run on the `gpu04` machine, using the datasets outputted by futhark dataset. The applications are then run ten times each, printing the timestamps to `/dev/stderr`. For benchmarking purposes i only included the `lssp-sorted` problem, since all three problems should see approximately the same speedup(the only difference is the predicates being tested), however in this resubmission i include the tests for all three lssp problems.

```
$ lssp-zeros-c
17859
...
17824
-- Avg: 17835
$ lssp-zeros-opencl
736
...
742
-- Avg: 737

$ lssp-same-c
23793
..
23783
-- Avg: 23808
$ lssp-same-opencl
744
...
736
-- Avg: 739

$ lssp-sorted-c
23757
...
```

```
23823
-- Avg: 23934
$ lssp-sorted-opencl
735
...
736
-- Avg: 736
```

When calculating my speeup down below i made an error. The correct way to calculate the speedup would be $\frac{sequential}{parallel}$, and i do this for each of the three predicates:

```
17835/737 = 24x speedup
23808/739 = 32x speedup
23934/736 = 32x speedup
```

And as can be seen, this parallel version gives us around 32x speedup. One reason why the lssp-zeros runs faster than the other two, might be because comparing with zero is a machine-code instruction, while comparing two numbers to each other, requires a subtraction and a register read/write. This could explain why lssp-zeros is faster htan lssp-sorted and lssp-same.

2.2 Lssp

We write the parallel version of the longest segment by implementing according to the lecture notes. More in-depth; We calculate the LSS and the following extra baggage:

1. LSS: The current longest segment, which is defined as `max(max(lssx, lssy), connecting_length)` where the connecting length is (lcsx + lisy) if and only if there is a connection across boundaries

2. LIS: Longest initial segment, which differs from MIS by only being applicable if `lisx` spans the whole x segment. This is the expanded expression: `if connect && lisx ==tlx then tlx + lisy else lisx`

3. LCS: Longest Concluding segment, which differs in the same way, and is computed much the same (with the exception of checking against tly instead)

4. TL: The total length is the sum of the subsegments total length, e.g. `tlx + tly`

5. First: First is firstx(unless x is the neutral element, in which case we get firsty)

6. Last: Much like first, last is `lasty` unless y is the neutral element.

2.3 Speedups

I now test the runtime difference between a sequential c compilation, and an opencl compilation:

```
[wbr220@a00333 lssp]$ futhark dataset --i32-bounds=-10:10 -b -g [10000000]i32 > data.in
[wbr220@a00333 lssp]$ futhark c -o lssp-sorted-c lssp-sorted.fut
[wbr220@a00333 lssp]$ futhark opencl -o lssp-sorted-opencl lssp-sorted.fut
[wbr220@a00333 lssp]$ ./lssp-sorted-c -t /dev/stderr -r 10 <data.in > /dev/null
26101
25264
24472
24481
24607
24504
24471
24475
24472
25291
-- Avg: 24813
[wbr220@a00333 lssp]$ ./lssp-sorted-opencl -t /dev/stderr -r 10 <data.in > /dev/null
741
739
735
737
738
737
735
736
743
737
-- Avg: 737
```

And we see a 97% speedup on average

3 Task 3

3.1 Resubmission

For the resubmission, i split the kernel into two lines:

```
__global__ void kernel(float *d_in, float *d_out, int N){
  const unsigned int lid = threadIdx.x; // Local id inside a block
  const unsigned int gid = blockIdx.x*blockDim.x + lid; // global id
  if (gid < N){
    float x = d_in[gid]/(d_in[gid]-2.3);
    d_out[gid] = x*x*x;
  }
}
```

And the same for the sequential code. This achieves the optimizations mentioned. Furthermore i added a macro `BENCH_RUNS = 200`, and use this to run the gpu kernel 200 times, averaging the final runtime. Furthermore, i add a call to `cudaDeviceSynchronize()` after each kernel call to ensure the kernel has run completely. The final testing code looks like this:

```
gettimeofday(&t_start, NULL);
for(int i = 0; i < BENCH_RUNS; i++){
  kernel<<<num_blocks, block_size>>>(d_in, d_out, N);
}
cudaDeviceSynchronize();// Ensure kernel has finished
cudaAssert(cudaPeekAtLastError());
gettimeofday(&t_end, NULL);
```

Also included is an assertion that ensures no errors have been thrown

3.2 Assignment

I wrote the code, included in `wa1-task3.cu`, in Cuda c++. The code includes the requested GPU parallel in the `gpu_run(float* inp, float* out, int N)` function, that takes two allocated arrays of memory, designed for the input and output. It's sequential equivalent `seq_run(float* inp, float* out, int N)` has the same signature.

When running, the functions output their runtime in microseconds. Compiling the program with `nvcc -O3 -DN_ELEMS=753411 wa1-task3.cu` i get the following output:

```
[wbr220@a00333 t3]$ ./a.out
CPU Run took 53609 microseconds (53.61ms)
GPU Run took 33 microseconds (0.03ms)
Passed: 753411, Invalid: 000000
```

Which clearly demonstrates the effectiveness of parallel programming. The GPU Runs 99.9% faster than the CPU, outside of the time it takes to move the data to and from the device. This speedup is mostly explained by the GPU computing the results in blocks of 256 at a time.

We are interested in locating the spot where the GPU computes faster than the CPU. To help us, the compiler takes a directive: N_{ELEMS}, which defines amount of elements. To find the point i continually recompile the program while changing this amount, and log the time values

```
[wbr220@a00333 t3]$ ./test.sh
Compiling test 1
TEST 1
CPU Run took 7 microseconds (0.01ms)
GPU Run took 29 microseconds (0.03ms)
Passed: 000001, Invalid: 000000
Compiling test 10
TEST 10
CPU Run took 8 microseconds (0.01ms)
GPU Run took 31 microseconds (0.03ms)
Passed: 000010, Invalid: 000000
Compiling test 100
TEST 100
CPU Run took 16 microseconds (0.02ms)
GPU Run took 30 microseconds (0.03ms)
Passed: 000100, Invalid: 000000
Compiling test 250
TEST 250
CPU Run took 26 microseconds (0.03ms)
GPU Run took 28 microseconds (0.03ms)
Passed: 000250, Invalid: 000000
Compiling test 500
TEST 500
CPU Run took 49 microseconds (0.05ms)
GPU Run took 30 microseconds (0.03ms)
Passed: 000500, Invalid: 000000
Compiling test 1000
```

```
TEST 1000
CPU Run took 93 microseconds (0.09ms)
GPU Run took 30 microseconds (0.03ms)
Passed: 001000, Invalid: 000000
```

We now clearly see that for an array of a size arond 250 the CPU starts being slower than the GPU, and by $n = 1000$ the GPU is more than 3 times as fast as the CPU

4 Task 4

4.1 Resubmission

Your benchmarks of 'spMVmult-flat' are good! However, 'spMVmult-seq' was **not** intended for compilation to the OpenCL backend, but to the C backend, so your measurement comparisons are unfortunately not very representative. If you decide to resubmit, please re-run the exact same benchmarks, but compiling 'spMVmult-seq.fut' to the C backend

To Fix This I reran the spMVmult-seq and spMVmult-flat again with the following commands:

```
[wbr220@a00333 w1-code-handin]$ futhark bench --backend=opencl --runs=20 spMVmult-flat
Compiling spMVmult-flat.fut...
Reporting average runtime of 20 runs for each dataset.

Results for spMVmult-flat.fut:
#0 ("[0i32, 1i32, 0i32, 1i32, 2i32, 1i32, 2i32, 3i32, 2..."):         95 (RSD: 0.117; r
data.in:                                                             287 (RSD: 0.021; r
[wbr220@a00333 w1-code-handin]$ futhark bench --backend=c --runs=20 spMVmult-seq.fut
Compiling spMVmult-seq.fut...
Reporting average runtime of 20 runs for each dataset.

Results for spMVmult-seq.fut:
#0 ("[0i32, 1i32, 0i32, 1i32, 2i32, 1i32, 2i32, 3i32, 2..."):          0 (RSD: 2.380; r
data.in:                                                            1630 (RSD: 0.011; r
```

The sequential function now runs at a much more respectable speed, and the speedup can be calculated:

$$\frac{1630\mu - 287\mu}{1630\mu} \cdot 100 \approx 82.4$$

Which still leaves us with a respectable 82.4% speedup.

4.2 Task

We need to flatten the following function:

```
map (\ row ->
let prods =
      map (\(i,x) -> x*vct[i]) row
in  reduce (+) 0 prods
    ) mat
```

I wrote the following futhark function:

```
let spMatVctMult [num_elms] [vct_len] [num_rows]
 (mat_val : [num_elms](i32,f32))
 (mat_shp : [num_rows]i32)
 (vct : [vct_len]f32) : [num_rows]f32 =
  -- Calculates the index of the start of each sub-array(1-indexed)
  let shp_sc   = scan (+) 0 mat_shp
  -- Shift the shape array right
  let shp_rot  = map (\i -> if i == 0 then 0 else shp_sc[i-1]) (iota num_rows)
  -- Distribute the flags to their calculated places
  let row_flg  = scatter (replicate num_elms 0) shp_rot (replicate num_rows 1)
  -- Perform the actual matrix-vector multiplication
  let muls = map (\(i, x) -> x*vct[i]) mat_val
  -- Sum up each row using the flags calculated above
  let row_sums = sgmSumF32 row_flg muls
  -- Extract the last element of each row
  in map (\i -> row_sums[i-1]) shp_sc
```

I test with the following information:

```
$ futhark dataset --i32-bounds=0:9999 -g [1000000]i32 --f32-bounds=-7.0:7.0 -g [100000
```

And the test cases:

```
-- compiled input @ data.in auto output
```

in both `spMVmult-flat` and `spMVmult-seq`, and let the futhark bench run
20 rounds each and i get the following output:

```
[wbr220@a00333 w1-code-handin]$ futhark bench --backend=opencl --runs=20 spMVmult-seq.f
t.fut < data.in
Compiling spMVmult-seq.fut...
Compiling spMVmult-flat.fut...
```

```
Reporting average runtime of 20 runs for each dataset.

Results for spMVmult-flat.fut:
#0 ("[0i32, 1i32, 0i32, 1i32, 2i32, 1i32, 2i32, 3i32, 2..."):         93s (RSD: 0.076;
data.in:                                                             255s (RSD: 0.027;

Results for spMVmult-seq.fut:
#0 ("[0i32, 1i32, 0i32, 1i32, 2i32, 1i32, 2i32, 3i32, 2..."):        430s (RSD: 0.037;
data.in:                                                          32370819s (RSD: 0.182;
```

And we see an average speedup of 99.999%.