PMPH Weekly 1

Simon Lykke Andersen {sxd682}

September 11, 2024

Technical disclaimer

All tests were run on the hendrixfut03fl server.

1 Task

1.1 a

To prove that $(\text{Img}(h), \odot)$ is a monoid with neutral element e given a well-defined list-homomorphic program $h : A \to B$ we start by proving associativity.

1.1.1 Associativity

We need to prove:

$$\forall x, y, z \in \text{Img}(h) : (x \odot y) \odot z = x \odot (y \odot z)$$

To which we introduce lists a, b and c which are defined as follows $x = h\ a$, $y = h\ b$ and $z = h\ c$ since we know that x, y and z belong to the image of h. Now we can simply start from the LHS of the equation and deduce the RHS by the different cases of the list-homomorphic program, the specific case used (and direction of rewriting) is indicated in the subscript to =.

$$
\begin{aligned}
(x \odot y) \odot z &= \\
((h\ a) \odot (h\ b)) \odot (h\ c) &=_{3.\leftarrow} \\
(h\ (a +\!\!+ b)) \odot (h\ c) &=_{3.\leftarrow} \\
h\ (a +\!\!+ b +\!\!+ c) &=_{3.\rightarrow} \\
(h\ a) \odot (h\ (b +\!\!+ c)) &=_{3.\rightarrow} \\
(h\ a) \odot ((h\ b) \odot (h\ c)) &= \\
x \odot (y \odot z)\Box
\end{aligned}
$$

1.1.2 Neutral element

We need to prove:

$$\forall z \in \mathrm{Img}(h) : x \odot e = e \odot x = x$$

Once again we introduce a list a defined by $x = h\ a$. We split up the equality proving $x \odot e = x$ and $e \odot x = x$ separately which will then of course also imply $x \odot e = e \odot x$. For the proof we need to remind ourselves that the empty list is the neutral element for concatenation, i.e. $a +\!\!+ [] = [] +\!\!+ a = a$.

Firstly $x \odot e = x$:

$$
\begin{aligned}
x \odot e &= \\
(h\ a) \odot e &=_{1.\leftarrow} \\
(h\ a) \odot (h\ []) &=_{3.\leftarrow} \\
h\ (a +\!\!+ []) &= \\
h\ a &= \\
x\Box
\end{aligned}
$$

Secondly $e \odot x = x$ in the exact same manner except for the empty list being to the left of a:

$$
\begin{aligned}
e \odot x &= \\
e \odot (h\ a) &=_{1.\leftarrow} \\
(h\ []) \odot (h\ a) &=_{3.\leftarrow} \\
h\ ([] +\!\!+ a) &= \\
h\ a &= \\
x\Box
\end{aligned}
$$

Now we have proven that $(\mathrm{Img}(h), \odot)$ is both associative and has a neutral element e and we can therefore conclude that a list-homomorphic program induces a monoid structure.

1.2 b

We need to prove:

$$\begin{aligned}(\texttt{reduce}\ (+)\ 0) \circ (\texttt{map f}) \equiv \\ (\texttt{reduce}\ (+)\ 0) \circ (\texttt{map}\ ((\texttt{reduce}\ (+)\ 0) \circ (\texttt{map f})) \circ \mathrm{distr}_p\end{aligned}$$

To prove it we need the promotion lemmas from the lecture notes which will be referred to when used as well as the following list identity:

$$(\texttt{reduce}\ (++)\ [\,]) \circ \mathrm{distr}_p = \texttt{id}$$

We start with the LHS of the equivalence:

$$\begin{aligned}(\texttt{reduce}\ (+)\ 0) \circ (\texttt{map f}) &\equiv_{\text{id}} \\ (\texttt{reduce}\ (+)\ 0) \circ (\texttt{map f}) \circ (\texttt{reduce}\ (++)\ [\,]) \circ \mathrm{distr}_p &\equiv_{\text{2.lemma}} \\ (\texttt{reduce}\ (+)\ 0) \circ (\texttt{reduce}\ (++)\ [\,]) \circ (\texttt{map}\ (\texttt{map f})) \circ \mathrm{distr}_p &\equiv_{\text{3.lemma}} \\ (\texttt{reduce}\ (+)\ 0) \circ (\texttt{map}\ (\texttt{reduce}\ (+)\ 0)) \circ (\texttt{map}\ (\texttt{map f})) \circ \mathrm{distr}_p &\equiv_{\text{1.lemma}} \\ (\texttt{reduce}\ (+)\ 0) \circ (\texttt{map}\ ((\texttt{reduce}\ (+)\ 0) \circ (\texttt{map f}))) \circ \mathrm{distr}_p \square\end{aligned}$$

Therefore the optimized map-reduce lemma holds.

2 Task

2.1 Solution

```
let segments_connect = if (x_len == 0 || y_len == 0)
                       then true else (pred2 x_last y_first)

let new_lss = max x_lss (if segments_connect
                         then (max y_lss (x_lcs+y_lis))
                         else y_lss) -- Longest Satisfying Segment.
let new_lis = if x_lis == x_len && segments_connect
              then x_lis + y_lis
              else x_lis -- Longest Initial Segment
let new_lcs = if y_lcs == y_len && segments_connect
              then y_lcs + x_lcs
              else y_lcs -- Longest concluding Segment
let new_len = x_len+y_len -- New length
```

2.2 Tests

For each of the three predicates I added three additional inline tests (to the one given) in order to check that my implementation works as expected. They can be found in the test comments above the various functions for the different predicates. Below I give the test input and expected output in tabular form.

`lssp-zeros.fut`:

Input	Expected output
`[0, 0, -2, 1, 0, 1, 0, 0, -3, 0, 0, 0, 1]`	3
`[-3, -2, -1, 0, 1, 2, 3]`	1
`[-3, -2, -1, 1, 2, 3]`	0

lssp-sorted.fut:

Input	Expected output
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	10
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	12

lssp-same.fut:

Input	Expected output
[1, -2, -2, 4, -6, 1]	2
[1, 2, 3, 1, 2, 3]	1
[1, -1, 1, 1, -1, -1, 1]	2

When running the following three commands (having separated the larger datasets and therefore only testing the above listed test cases):

```
futhark test --backend=cuda lssp-sorted.fut
futhark test --backend=cuda lssp-same.fut
futhark test --backend=cuda lssp-zeros.fut
```

They all pass and I am therefore reasonably certain that my implementation of lssp is correct.

2.3 Parallel and Sequential comparison

To compare the parallel and sequential implementation and the speedups obtained I generate two sets of datasets. One for `lssp-zeros.fut` and one for both `lssp-same.fut` and `lssp-sorted.fut` since I did not see any good reasons to vary the ranges for the randomly generated benchmarking inputs for those. In a benchmarking shell script (named `benchmarking.sh`) I generate these two sets of data in three different sizes (and for each size two different ranges of random numbers):

```
echo "GENERATING: data for lssp-zeros.fut"
futhark dataset --i32-bounds=-1:1 -b -g '[64000]i32' > test_data/lssp-zeros-...
futhark dataset --i32-bounds=-2040:2040 -b -g '[64000]i32' > test_data/lssp-zeros-
futhark dataset --i32-bounds=-1:1 -b -g '[640000]i32' > test_data/lssp-zeros-...
futhark dataset --i32-bounds=-2040:2040 -b -g '[640000]i32' > test_data/lssp-zeros
```

```
futhark dataset --i32-bounds=-1:1 -b -g '[64000000]i32' > test_data/lssp-zeros-...
futhark dataset --i32-bounds=-2040:2040 -b -g '[64000000]i32' > test_data/lssp-zer

echo "GENERATING: data for lssp-sorted.fut and lssp-same.fut"
futhark dataset --i32-bounds=1:10 -b -g '[64000]i32' > test_data/lssp-sorted-...
futhark dataset --i32-bounds=-2040:2040 -b -g '[64000]i32' > test_data/lssp-sorted
futhark dataset --i32-bounds=1:10 -b -g '[640000]i32' > test_data/lssp-sorted-...
futhark dataset --i32-bounds=-2040:2040 -b -g '[640000]i32' > test_data/lssp-sorte
futhark dataset --i32-bounds=1:10 -b -g '[64000000]i32' > test_data/lssp-sorted-..
futhark dataset --i32-bounds=-2040:2040 -b -g '[64000000]i32' > test_data/lssp-sor
```

I test these datasets on the sequential implementation of each of the predicates (using the handed-out `lssp_seq` implementation) on the c backend and on the parallel implementation of each of the predicates on the cuda backend. Where I get the following runtimes (I have shortened the test filenames):

Test file	Zeros	Sorted	Same
`...-small-interval-small.in`	279	459	212
`...-large-interval-small.in`	179	452	131
`...-small-interval-large.in`	2882	4543	2156
`...-large-interval-large.in`	1782	4494	1303
`...-small-interval-very-large.in`	165794	258447	162949
`...-large-interval-very-large.in`	109274	255434	75211

Table 1: The table for futhark bench –backend=c lssp-....fut -e mainSeq

Test file	Zeros	Sorted	Same
`...-small-interval-small.in`	36	36	36
`...-large-interval-small.in`	37	43	35
`...-small-interval-large.in`	60	62	59
`...-large-interval-large.in`	60	62	60
`...-small-interval-very-large.in`	1454	1454	1454
`...-large-interval-very-large.in`	1452	1453	1461

Table 2: The table for futhark bench –backend=cuda lssp-....fut -e mainSeq

I calculated the speedups in a small python script where I got the following speedups per. data set.

Test file	Zeros	Sorted	Same
`...-small-interval-small.in`	7.75	12.75	5.89
`...-large-interval-small.in`	4.84	10.51	3.74
`...-small-interval-large.in`	48.03	73.27	36.54
`...-large-interval-large.in`	29.7	72.48	21.72
`...-small-interval-very-large.in`	114.03	177.75	112.07
`...-large-interval-very-large.in`	75.26	175.8	51.48

Table 3: The table of speedups between the sequential and parallel version per. dataset for each of the predicates

3 Task

3.1 Validation

Running with $N = 753411$ and `float epsilon_error = 0.000001;` I get two invalid results (though at different cut-off epsilon errors). One is for index 1 and one is for index 3:

```
Invalid result at index 1, actual: -296.296265, expected: -296.296478.
Invalid result at index 3, actual: 13.026666, expected: 13.026662.
```

Though if one sets `float epsilon_error = 0.0000001;` basically all of the cases fails.

3.2 Solution

The CUDA kernel:

```
__global__ void mapKernel(float* X, float *Y, int N) {
    const unsigned int gid = blockIdx.x*blockDim.x+threadIdx.x;
    if (gid < N) {
        float x = X[gid];
        float partial_res = (x/(x-2.3));
        Y[gid] = partial_res*partial_res*partial_res;
    }
}
```

This is how the block sizes and grid gets calculated.

```
unsigned int B = 256; // chose a suitable block size in dimension x
unsigned int numblocks = (N + B -1) / B; // number of blocks in dimension x
dim3 block(B,1,1), grid(numblocks,1,1);
```

This is how the kernel is called and how it is timed. (`GPU_RUNS = 300`)

```
double gpu_elapsed; struct timeval gpu_t_start, gpu_t_end, gpu_t_diff;
gettimeofday(&gpu_t_start, NULL);

for(int r = 0; r < GPU_RUNS; r++) {
    mapKernel<<<grid , block>>>(d_in , d_out, N);
}
```

```
cudaDeviceSynchronize();

// TIMING
gettimeofday(&gpu_t_end, NULL);
timeval_subtract(&gpu_t_diff, &gpu_t_end, &gpu_t_start);
gpu_elapsed = (1.0 * (gpu_t_diff.tv_sec*1e6+gpu_t_diff.tv_usec)) / GPU_RUNS;
double gpu_gigabytespersec = (2.0 * N * 4.0) / (gpu_elapsed * 1000.0);
printf("The kernel took on average %f microseconds. GB/sec: %f \n",
  \gpu_elapsed, gpu_gigabytespersec);
```

3.3 Maximal throughput

The largest `N` which I tried was 1073083647. Below is a small table of the correlation between size of `N` and the gotten throughput GB/sec.

N	GB/sec
753411	722.70
7534110	994.99
75341100	1152.77
500000000	1304.33
1073083647	1334.48

Table 4: Table over input size and throughput.

This indicates that the maximal throughput the program could attain would be around $\approx$ 1335 GB/sec. And for the initial input size 722.70 GB/sec.

4 Task

4.1 Solution

```
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 =
  let mat_flg' = mkFlagArray mat_shp (false) (replicate num_rows true)
  let mat_flg  = mat_flg' :> [num_elms]bool
  let prods    = map (\(i,x) -> x*vct[i]) mat_val
  let sc_mat   = sgmSumF32 mat_flg prods
  let indsp1   = scan (+) 0 mat_shp
  let res = map2 (\shp ip1 -> if shp==0 then 0
                              else sc_mat[ip1-1]
                ) mat_shp indsp1
  in res
```

I have used the implementation of `mkFlagArray` from the lecture notes and followed rewrite rule 5 also rather closely (though replacing the zero element argument to `mkFlagArray` with `false` since it is a boolean flag array).
Going through it line by line:

- `let mat_flg' = ...` and `let mat_flg  = ...` simply creates the flag array needed for the segmented scan. (The first calling `mkFlagArray` function and the second size-casting the result). I have also since implementing the code realized that the flag array needed here could also simply be obtained by:
  ```
  scatter (replicate num_elms false)
          (map (-1) mat_shp) (replicate num_rows true)
  ```
  possibly with a more complicated map-function if `mat_shp` possibly contains zeros (though only consisting of an if-statement checking if zero) and then that would also have consequences for the length of the true-value array to be scattered.

- `let prods    = ...` simply takes the needed map previously inside a let-binding in the map (which was then used in the reduce) and performs it directly (since the flattening of a nested map just becomes a map). This map also multiplies the matrix elements in column `i` with the corresponding vector entry.

- `let sc_mat   = ...` adds up each row (through a segmented scan) where the actually needed value is the last one in the row. (That which corresponds to the result of the reduce of the nested equivalent).
- `let indsp1   = ...` finds the indices (+1) of the previously mentioned "reduces" of each row (i.e. the sum of the products).
- `let res = map2(...` collects the final resulting vector of the matrix-vector multiplication. If a row of the matrix was not empty (in which case the value is of course 0 because an empty row would mean a zero-row in the sparse matrix representation) it uses the previously mentioned index of the "reduce" to look up the actual resulting value in the segmented scan.

4.2 Testing

For testing I added a slightly larger dataset with a "trusted" solution from the given sequential implementation. This I generate by a scaled down version of the given larger data sets via the following commands:

```
futhark dataset --i64-bounds=0:99 -g '[1000]i64' --f32-bounds=-7.0:7.0
  \-g '[1000]f32' --i64-bounds=10:10 -g '[100]i64' --f32-bounds=-10.0:10.0
  \-g '[100]f32' > spMVtest.in
futhark c spMVmult-seq.fut
./spMVmult-seq < spMVtest.in > spMVtest.out
```

And then through an added test case from a file source. Both the given test and the test with correct results via the trusted given implementation pass when testing with:

```
futhark test --backend=cuda spMVmult-flat.fu
```

4.3 Speedup

Testing on the given dataset generation specification (50 times instead of 10):

```
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 50 > /dev/null
...
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 50 > /dev/null
...
```

I get a speedup of 9.75 on the flattened version vs. the sequential version.