KØBENHAVNS UNIVERSITET

PMPH - Assignment 1

Mikkel Willén, bmq419

23. september 2022

Indhold

Task 1

1a

Associativity:

$$\begin{aligned}
(h\ a \circ h\ b) \circ h\ c &= h(a++b) \circ h\ c && \text{(Def 3.)}\\
&= h(a++b++c) && \text{(Def 3.)}\\
&= h\ a \circ h(b++c) && \text{(Def 3.)}\\
&= h\ a \circ (h\ b \circ h\ c) && \text{(Def 3.)}
\end{aligned}$$

Neutral element:

$$\begin{aligned}
b \circ e &= h\ [b] \circ h\ [\,] && \text{(Def 1. and Def 2.)}\\
&= h\ [b] && \text{(Def. in assignment text)}\\
&= h([\,]++[b]) && \text{(Def. in assignment text)}\\
&= h\ [\,] \circ h\ [b] && \text{(Def. in assignment text)}\\
&= e \circ b && \text{(Def 2. and Def 1.)}
\end{aligned}$$

1b

$$\begin{aligned}
& (reduce\ (+)\ 0) \circ (map\ f)\\
={}& (reduce\ (+)\ 0) \circ (map\ f) \circ (reduce\ (++)\ [\,]) \circ distr_p && \text{(Assignment text)}\\
={}& (reduce\ (+)\ 0) \circ (reduce\ (++)\ [\,]) \circ (map\ (map\ f)) \circ distr_p && \text{Rule 2.}\\
={}& (reduce\ (+)\ 0) \circ (map(reduce\ (+)\ 0)) \circ (map\ (map\ f)) \circ distr_p && \text{Rule 3.}\\
={}& (reduce\ (+)\ 0) \circ (map\ ((reduce\ (+)\ 0) \circ (map\ f)) \circ distr_p && \text{Rule 1.}
\end{aligned}$$

Task 2

The five added lines to the lssp.fut file:

```
let connect= tlx == 0 || tly == 0 || pred2 lastx firsty
let newlss = if connect then max(lssx, max(lssy, lcsx + lisy))
  else max(lssx, lssy)
let newlis = if connect && lisx == tlx then tlx + lisy else lisx
let newlcs = if connect && lcsy == tly then tly + lcsx else lcsy
let newtl  = tlx + tly
```

Below is the testdata for lssp-sorted with and without acceleration.

lssp-sorted		
runs	with acceleration	without acceleration
1	732	24403
2	726	24524
N 3	725	24318
4	725	24396
5	737	24572
6	722	24302
7	722	24405
8	723	24320
9	723	24282
10	737	24233
avg	727,2	24375,5

Which gives a speedup of 33.5.

Below is the testdata for lssp-same with and without acceleration.

lssp-same		
runs	with acceleration	without acceleration
1	724	24499
2	724	24421
3	723	24462
4	722	24447
5	722	24460
6	725	24534
7	725	24559
8	721	24511
9	720	24483
10	726	24645
avg	723.2	24502.1

Which gives a speedup of 33.7.

Below is the testdata for lssp-zeros with and without acceleration.

lssp-zeros		
runs	with acceleration	without acceleration
1	750	13293
2	760	13280
3	761	13264
4	757	13395
5	753	13224
6	757	13242
7	749	13209
8	752	13248
9	755	13208
10	757	13241
avg	755.1	13260.4

Which gives a speedup of 17.6.

The tests for lssp-sorted.fut:

```
-- compiled input {
--     [1i32, -2, -2, 0, 0, 0, 0, 0, 3, 4, -6, 1]
-- }
-- output {
--     9
-- }
-- compiled input {
--     [1i32, -2, -2, 0, 0, 0, 0, 0, 3, 4, -6, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 2, 1, 3, 0, 2, 1, 0]
-- }
-- output {
--     12
-- }
-- compiled input {
--     [1i32, -2, -2, 3, 4, -6, 1]
-- }
-- output {
--     4
-- }
-- compiled input {
--     [1i32, -2, -2, 3, 4, 6, 1, 0]
-- }
-- output {
--     5
-- }
-- compiled input {
--     [0i32, -2, -2, 3, 4, 6, 10, 0]
-- }
-- output {
--     6
-- }
```

The tests for lssp-same.fut:

```
-- compiled input {
--     [1i32, -2, -2, 0, 0, 0, 0, 0, 3, 4, -6, 1]
-- }
-- output {
--     5
-- }
-- compiled input {
--     [1i32, -2, -2, 0, 0, 0, 0, 0, 3, 4, -6, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 2, 1, 3, 0, 2, 1, 0]
-- }
-- output {
--     10
-- }
-- compiled input {
--     [1i32, 2, 2, 3, 4, -6, 1]
-- }
-- output {
--     2
-- }
```

```
-- compiled input {
--    [1i32, -2, -2, -3, 4, -6, 1, 0]
-- }
-- output {
--    2
-- }
-- compiled input {
--    [0i32, 2, 2, 2, 3, 4, -6, 1, 0]
-- }
-- output {
--    3
-- }
```

The tests for lssp-zeros.fut:

```
-- compiled input {
--    [1i32, -2, -2, 0, 0, 0, 0, 0, 3, 4, -6, 1]
-- }
-- output {
--    5
-- }
-- compiled input {
--    [1i32, -2, -2, 0, 0, 0, 0, 0, 3, 4, -6, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 2, 1, 3, 0, 2, 1, 0]
-- }
-- output {
--    10
-- }
-- compiled input {
--    [1i32, -2, -2, 3, 4, -6, 1]
-- }
-- output {
--    0
-- }
-- compiled input {
--    [1i32, -2, -2, 3, 4, -6, 1, 0]
-- }
-- output {
--    1
-- }
-- compiled input {
--    [0i32, -2, -2, 3, 4, -6, 1, 0]
-- }
-- output {
--    1
-- }
```

Task 3

The program validates (det virker sgu = valid) the outputs of the functions with an $\epsilon = 0.0000001$. Below is the call, where the functions map to the array $[1, \ldots, 753411]$

```
[bmq419@a00333 CUDA]$ ./a.out 753411
```

```
det virker sgu
Parallel took:   73 microseconds (0.00ms)
Sequential took: 11625 microseconds (11.62ms)
```

The parallel function is way faster at an N of 753411. The speedup of the parallel function is around 150x faster.

Below this is a test of when the parallel function becomes faster than the sequential function.

```
[bmq419@a00333 CUDA]$ ./a.out 128
det virker sgu
Parallel took:   2 microseconds (0.00ms)
Sequentiel took: 10 microseconds (0.01ms)

[bmq419@a00333 CUDA]$ ./a.out 64
det virker sgu
Parallel took:   3 microseconds (0.00ms)
Sequentiel took: 5 microseconds (0.01ms)

[bmq419@a00333 CUDA]$ ./a.out 32
det virker sgu
Parallel took:   2 microseconds (0.00ms)
Sequentiel took: 2 microseconds (0.00ms)

[bmq419@a00333 CUDA]$ ./a.out 16
det virker sgu
Parallel took:   3 microseconds (0.00ms)
Sequentiel took: 1 microseconds (0.00ms)

[bmq419@a00333 CUDA]$ ./a.out 24
det virker sgu
Parallel took:   2 microseconds (0.00ms)
Sequentiel took: 2 microseconds (0.00ms)

[bmq419@a00333 CUDA]$ ./a.out 20
det virker sgu
Parallel took:   3 microseconds (0.00ms)
Sequentiel took: 1 microseconds (0.00ms)
```

It seems like it is between $N = 16$ and $N = 32$. If speed is the only requirement for the program, then it is pretty much always benefitial to use the parallel version of the function.

Code for the CUDA kernel:

```
__global__ void squareKernel(float* d_in, float* gpu_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 temp = d_in[gid]; // access memory once
        float inner = temp / (temp - 2.3); // do computation of inner function
        gpu_out[gid] = inner * inner * inner; // computation of the power of 3
```

```
    }
}
```

Code for the call to the CUDA kernel:

```
squareKernel<<< (blocksize - 1 + N) / blocksize, blocksize >>>
    (d_in, gpu_out, N);
```

The blocksize is by default set to 256, and can be set further up in the code.

Task 4

The code for the flat-parallel implementation:

```
1 let spMatVctMult [num_elms] [vct_len] [num_rows]
2                  (mat_val : [num_elms](i64,f32))
3                  (mat_shp : [num_rows]i64)
4                  (vct : [vct_len]f32) : [num_rows]f32 =
5   -- makes a kind of index array
6   let shp_sc = scan (+) 0 mat_shp
7   -- scan made exlusive - index array
8   let inds = map(\i -> if i == 0
9                           then 0
10                       else
11                          shp_sc[i - 1]) (iota num_rows)
12  -- makes an array of 0 and a number
13  let tmp = scatter (replicate num_elms 0) inds mat_shp
14  -- converts to boolean
15  let flag = map (\i -> i != 0) tmp
16  -- multiply the vector on to the matrix
17  let vTr = map (\(i, x) -> x * vct[i]) mat_val
18  -- calls the segmentScan function
19  let segSum = sgmSumF32 flag vTr
20  -- get the result
21  in map (\i -> segSum[i - 1]) shp_sc
```

Line 6 makes an array with the index of the last element of each subarray. Line 8-11 makes the index array by changing the result from line 6 to inclusive scan. Line 13 makes the flag array, but with 0 or a non-zero number. Line 15 then converts that to boolean values, since sgmSumF32 only works with boolean arrays. Line 17 multiply the vector on to the matrix. Line 19 calls the segmentScan function provided. Line 21 then maps the result from segScan with the last element of each subarray, and returns that value.

Below is given the data from running a few test sets on the code for both the sequential and the parallel programs.

Sparse-Matrix Vector Multiplication		
runs	sequential	parallel
1	1855	119
2	1631	120
3	1668	118
4	1735	128
5	1629	120
6	1624	131
7	1684	118
8	1627	120
9	1677	118
10	1690	118
avg	1682	121

This gives a speedup of 13.9.