**Programming Massively Parallel Hardware** **Name:** *Gilles Souton*
**Assignment 1**
**15-09-2022**

**Note** all the test were done on the machine gpu04.

# Task 1

## 1.a Prove that a list-homomorphism induces a monoid structure

**Associativity** For any $a, b, c\ (a \circ b) \circ c = a \circ (b \circ a)$

Knowing: $\exists\ x, y, z$ where $h\ x = a, h\ y = b, h\ z = c$
we can write:

$$
\begin{aligned}
(h\ x \circ h\ y) \circ h\ z \equiv && (1)\\
h(x +\!\!+\ y) \circ h\ z \equiv && (2)\\
h(x +\!\!+\ y +\!\!+\ z) \equiv && (3)\\
h(x +\!\!+\ (y +\!\!+\ z)) \equiv && (4)\\
h\ x \circ (h\ (y +\!\!+\ z))) \equiv && (5)\\
h\ x \circ (h\ y \circ h\ z) \equiv && (6)\\
&& (7)
\end{aligned}
$$

(4) Since the operator ++ is associative

**Neutral element** For all $b$ in $Img(h)\ b \circ e = e \circ b = b$

Knowing: $\exists\ x$ where $h\ x = b$
we can write:

$$
\begin{aligned}
b\ \circ e \equiv && (8)\\
h\ x \circ e \equiv && (9)\\
h\ (x +\!\!+\ [\,]) \equiv && (10)\\
h\ ([\,] +\!\!+\ x) \equiv && (11)\\
e \circ hx \equiv && (12)\\
e \circ b \equiv && (13)\\
&& (14)
\end{aligned}
$$

The transition $(9-> 10)$ is explained with the rule $h[\,] = e$

Finally the concatenation with the emtpy list (10, 11)

$$h\ (x +\!\!+ [\,]) \equiv \tag{15}$$
$$h\ [x] \equiv \tag{16}$$
$$f\ x \equiv \tag{17}$$
$$\tag{18}$$

## 1.b Prove the Optimized Map-Reduce Lemma

### Task

$$(reduce\ (+)\ 0) \circ (map\ f) \equiv \tag{19}$$
$$(reduce\ (+)\ 0) \circ (map\ ((reduce\ (+)\ 0) \circ (map\ f))) \circ distr_p \tag{20}$$

Using the hint: $(reduce\ (++)\ [\,])\ \circ distr_p = id$ we can deduce

$$(reduce\ (+)\ 0) \circ (map\ f) \equiv \tag{21}$$
$$(reduce\ (+)\ 0) \circ (map\ f) \circ (reduce\ (++)[\,]) \circ distr_p \equiv \tag{22}$$
$$(reduce\ (+)\ 0) \circ (map\ (reduce\ (+)\ 0)) \circ (\ mapf) \circ distr_p \equiv \tag{23}$$
$$(reduce\ (+)\ 0) \circ (map\ ((reduce\ (+)\ 0) \circ (\ mapf))) \circ distr_p \tag{24}$$

(23) Use of the rule $(\ reduce(\odot)e) \circ (\ reduce\ (++)\ [\,]) \equiv (reduce\ (\odot)e) \circ (\ map\ (reduce(\odot)e))$

(24) Use of the rule $(\ map\ f) \circ (\ map\ g) \equiv map(\ f \circ g)$

# Task 2: Longest Satisfying Segment (LSS) Problem

| Benchmark | | | |
|---|---|---|---|
| source | openCL | CUDA | C |
| *lssp-sorted* | 2875 | 3432 | 23877 |
| *lssp-same* | 2872 | 3438 | 23866 |
| *lssp-zeros* | 3020 | 3411 | 12940 |

The benchmark was done with the following command:

```
futhark dataset --i32-bounds=-10:10 -b -g [10000000]i32 |
↪  lssp-sorted -t /dev/stderr -r 10
```

We can observe a speedup that is overall aroud 8 using openCL as a backend.

**Code:** The code added was the following:

```
let connect = if (tlx != 0) && (tly != 0) then pred2 lastx
↪  firsty else true -- think about empty list

let newlss = if connect then max(lssy, max(lssx, (lcsx +
↪  lisy))) else  max(lssx, lssy) -- if connected then
↪  either the sum of both segment or the maximum between
↪  the left and right segment

let newlis = if connect && lisx == tlx then lisx +  lisy
↪  else lisx -- if connected and the left segment is
↪  reaching the connection

let newlcs = if (lcsy == tly) && connect && lcsy == tly
↪  then lcsy +  lcsxl else lcsy -- same condition as
↪  before but for the right segment

let newtl  = tlx + tly
```

## Task 3: CUDA exercise, see lab 1 slides: Lab1-CudaIntro

We can observe that the task run on the GPU is way faster than the one running on the CPU with an average speedup of 36.

**Explanation** Mapping the function on each element of the array becomes inherently faster when distributing the computation with multiple threads. Generally the bigger the array the bigger the speedup will be. Since that the computation will be distributed.

**Code** below some part of the code showing how the block size is computed and the kernel code.

```
#define SIZE 753411 //default size for the array
#define CPU_RUN 100 // number of time the task will be
↪  executed on CPU
#define GPU_RUN 100 // number of time the task will be
↪  executed on GPU
#define BLOCK_SIZE 256
#define MAX_BLOCK_SIZE 1024
#define EPSILON 0.001

__global__ void cubeKernel(float *d_in, float *d_out,
↪  unsigned int size ) {
    const unsigned int local_id = threadIdx.x; // local id
     ↪  inside a block
```

```
    const unsigned int global_id = blockIdx.x * blockDim.x + local_id; // global id
    if(global_id < size){
        float x = d_in[global_id]/(d_in[global_id]-2.3);
        d_out[global_id] = x*x*x;
    }
}

unsigned long int execute_task_on_gpu(float* host_in, float* host_out, env_t *env){
    unsigned int block_size = env->block_size;
    unsigned int gpu_run = env->gpu_run;
    assert(block_size <= MAX_BLOCK_SIZE);

    unsigned long int elapsed; struct timeval t_start, t_end, t_diff;

    //Compute number of block needed
    unsigned int num_blocks = (env->array_size + (block_size - 1)) / block_size;
    fprintf(stderr, "{Block_size: %d, num_blocks: %d}\n", block_size, num_blocks);

    //// allocate device memory
    //// copy host memory to device
    //// execute the kernel
    //// copy result from device to host
    //// clean-up memory
}
```

**Benchmark** On the benchmark below we can see that the data is validated, however it is possible that the data invalidate on smaller datasets.

1. The time measured is in miliseconds (ms).
2. The column validity indicate the validity valid = 1, invalid = 0.

| array_size | block_size | epsilon | cpu_time | gpu_time | speedup | validity |
|---|---|---|---|---|---|---|
| 500000 | 256 | 0.00001 | 1.08 | 0.03 | 35 | 0 |
| 500000 | 256 | 0.00010 | 1.07 | 0.03 | 34 | 0 |
| 500000 | 256 | 0.00100 | 1.07 | 0.03 | 35 | 0 |
| 500000 | 256 | 0.01000 | 1.07 | 0.03 | 35 | 0 |
| 500000 | 256 | 0.10000 | 1.09 | 0.03 | 36 | 0 |
| 500000 | 512 | 0.00001 | 1.10 | 0.03 | 36 | 0 |
| 500000 | 512 | 0.00010 | 1.07 | 0.03 | 35 | 0 |
| 500000 | 512 | 0.00100 | 1.07 | 0.03 | 35 | 0 |
| 500000 | 512 | 0.01000 | 1.07 | 0.03 | 35 | 0 |
| 500000 | 512 | 0.10000 | 1.09 | 0.03 | 36 | 0 |
| 500000 | 1024 | 0.00001 | 1.08 | 0.04 | 30 | 0 |
| 500000 | 1024 | 0.00010 | 1.08 | 0.04 | 30 | 0 |
| 500000 | 1024 | 0.00100 | 1.07 | 0.04 | 30 | 0 |
| 500000 | 1024 | 0.01000 | 1.07 | 0.04 | 30 | 0 |
| 500000 | 1024 | 0.10000 | 1.07 | 0.04 | 30 | 0 |
| 600000 | 256 | 0.00001 | 1.32 | 0.04 | 36 | 0 |
| 600000 | 256 | 0.00010 | 1.31 | 0.04 | 36 | 0 |
| 600000 | 256 | 0.00100 | 1.28 | 0.04 | 35 | 0 |
| 600000 | 256 | 0.01000 | 1.29 | 0.04 | 35 | 0 |
| 600000 | 256 | 0.10000 | 1.29 | 0.04 | 35 | 0 |
| 600000 | 512 | 0.00001 | 1.30 | 0.04 | 37 | 0 |
| 600000 | 512 | 0.00010 | 1.28 | 0.04 | 36 | 0 |
| 600000 | 512 | 0.00100 | 1.29 | 0.04 | 36 | 0 |
| 600000 | 512 | 0.01000 | 1.28 | 0.04 | 36 | 0 |
| 600000 | 512 | 0.10000 | 1.31 | 0.04 | 37 | 0 |
| 600000 | 1024 | 0.00001 | 1.30 | 0.04 | 33 | 0 |
| 600000 | 1024 | 0.00010 | 1.29 | 0.04 | 33 | 0 |
| 600000 | 1024 | 0.00100 | 1.30 | 0.04 | 33 | 0 |
| 600000 | 1024 | 0.01000 | 1.30 | 0.04 | 33 | 0 |
| 600000 | 1024 | 0.10000 | 1.28 | 0.04 | 32 | 0 |
| 750000 | 256 | 0.00001 | 1.60 | 0.04 | 35 | 0 |
| 750000 | 256 | 0.00010 | 1.60 | 0.04 | 35 | 0 |
| 750000 | 256 | 0.00100 | 1.59 | 0.04 | 35 | 0 |
| 750000 | 256 | 0.01000 | 1.60 | 0.04 | 35 | 0 |
| 750000 | 256 | 0.10000 | 1.62 | 0.04 | 35 | 0 |
| 750000 | 512 | 0.00001 | 1.59 | 0.04 | 36 | 0 |
| 750000 | 512 | 0.00010 | 1.60 | 0.04 | 36 | 0 |
| 750000 | 512 | 0.00100 | 1.64 | 0.04 | 37 | 0 |
| 750000 | 512 | 0.01000 | 1.67 | 0.04 | 37 | 0 |
| 750000 | 512 | 0.10000 | 1.59 | 0.04 | 36 | 0 |
| 750000 | 1024 | 0.00001 | 1.66 | 0.05 | 33 | 0 |
| 750000 | 1024 | 0.00010 | 1.64 | 0.05 | 33 | 0 |
| 750000 | 1024 | 0.00100 | 1.61 | 0.05 | 32 | 0 |

| array_size | block_size | epsilon | cpu_time | gpu_time | speedup | validity |
|---|---|---|---|---|---|---|
| 800000 | 256 | 0.00001 | 1.73 | 0.05 | 36 | 1 |
| 800000 | 256 | 0.00010 | 1.71 | 0.05 | 35 | 1 |
| 800000 | 256 | 0.00100 | 1.70 | 0.05 | 35 | 1 |
| 800000 | 256 | 0.01000 | 1.72 | 0.05 | 35 | 1 |
| 800000 | 256 | 0.10000 | 1.74 | 0.05 | 36 | 1 |
| 800000 | 512 | 0.00001 | 1.70 | 0.05 | 36 | 1 |
| 800000 | 512 | 0.00010 | 1.70 | 0.05 | 36 | 1 |
| 800000 | 512 | 0.00100 | 1.77 | 0.05 | 37 | 1 |
| 800000 | 512 | 0.01000 | 1.74 | 0.05 | 36 | 1 |
| 800000 | 512 | 0.10000 | 1.70 | 0.05 | 36 | 1 |
| 800000 | 1024 | 0.00001 | 1.70 | 0.06 | 30 | 1 |
| 800000 | 1024 | 0.00010 | 1.75 | 0.06 | 31 | 1 |
| 800000 | 1024 | 0.00100 | 1.71 | 0.06 | 31 | 1 |
| 800000 | 1024 | 0.01000 | 1.71 | 0.06 | 31 | 1 |
| 800000 | 1024 | 0.10000 | 1.74 | 0.06 | 31 | 1 |
| 900000 | 256 | 0.00001 | 1.97 | 0.05 | 37 | 1 |
| 900000 | 256 | 0.00010 | 1.91 | 0.05 | 36 | 1 |
| 900000 | 256 | 0.00100 | 1.98 | 0.05 | 37 | 1 |
| 900000 | 256 | 0.01000 | 1.93 | 0.05 | 36 | 1 |
| 900000 | 256 | 0.10000 | 1.92 | 0.05 | 36 | 1 |
| 900000 | 512 | 0.00001 | 2.02 | 0.05 | 38 | 1 |
| 900000 | 512 | 0.00010 | 1.93 | 0.05 | 37 | 1 |
| 900000 | 512 | 0.00100 | 1.93 | 0.05 | 37 | 1 |
| 900000 | 512 | 0.01000 | 1.96 | 0.05 | 37 | 1 |
| 900000 | 512 | 0.10000 | 1.91 | 0.05 | 36 | 1 |
| 900000 | 1024 | 0.00001 | 1.93 | 0.06 | 32 | 1 |
| 900000 | 1024 | 0.00010 | 1.96 | 0.06 | 32 | 1 |
| 900000 | 1024 | 0.00100 | 1.93 | 0.06 | 32 | 1 |
| 900000 | 1024 | 0.01000 | 1.91 | 0.06 | 31 | 1 |
| 900000 | 1024 | 0.10000 | 1.98 | 0.06 | 33 | 1 |
| 1000000 | 256 | 0.00001 | 2.12 | 0.06 | 35 | 1 |
| 1000000 | 256 | 0.00010 | 2.14 | 0.06 | 36 | 1 |
| 1000000 | 256 | 0.00100 | 2.13 | 0.06 | 36 | 1 |
| 1000000 | 256 | 0.01000 | 2.13 | 0.06 | 36 | 1 |
| 1000000 | 256 | 0.10000 | 2.13 | 0.06 | 36 | 1 |
| 1000000 | 512 | 0.00001 | 2.12 | 0.06 | 36 | 1 |
| 1000000 | 512 | 0.00010 | 2.12 | 0.06 | 36 | 1 |
| 1000000 | 512 | 0.00100 | 2.13 | 0.06 | 36 | 1 |
| 1000000 | 512 | 0.01000 | 2.14 | 0.06 | 36 | 1 |
| 1000000 | 512 | 0.10000 | 2.13 | 0.06 | 36 | 1 |
| 1000000 | 1024 | 0.00001 | 2.16 | 0.07 | 31 | 1 |
| 1000000 | 1024 | 0.00010 | 2.13 | 0.07 | 31 | 1 |
| 1000000 | 1024 | 0.00100 | 2.13 | 0.07 | 31 | 1 |

## Task 4: Flat Sparse-Matrix Vector Multiplication in Futhark

**Explanation** If we look at the benchmark below we can see a speedup that is around 7.5. With the flatened version the gpu cores can take advantage of parallelism and have good loadbalance to outrun the cpu. Where as the sequential implementation make everything crowded and result in slower time than the *C* backend.

**Code** here's the code with explanation of each line

```
-- 1. compute inclusive scan
 let shp_sc   = scan (+) 0 mat_shp

 -- compute the exclusive scan from the inclusive:
 --   - rotate the array to have the last element at the begining
 --   - set the first element to be 0
 let shp_exc = map(\x -> if x == num_elms then 0 else x) (rotate (-1) shp_sc) -- rotating the array so num_elms at the begining

 -- 2. prepare array for scatter to create flags arra
 let input_vec = replicate num_elms false -- list of 0
 let data_vec = replicate num_rows true -- value to use to replace in the data vector

 -- 3. create the flags array
 let flags = scatter input_vec shp_exc data_vec -- modify the 0 to 1 in the input vector according to the index_vector

 -- 4. make the product across each row
 let prods = map (\(i,x) -> x*vct[i]) mat_val

 -- 5. sum up the products across each row of the matrix
 let segmented_sum = sgmSumF32 flags prods
 -- 6. Get the last element of each sub array (flat array)
 in map(\x -> segmented_sum[x-1]) shp_sc
```

| Benchmark | | | |
|---|---|---|---|
| source | openCL | CUDA | C |
| *seq* | ∅ | ∅ | 1730 |
| *flat* | 260 | 100 | 1825 |

```
[fpz747@a00333 spMatVct]$ ./bench
--- C ---
futhark c spMVmult-seq.fut
1694
1673
1688
1618
1772
1635
1727
1727
1725
1762
futhark c spMVmult-flat.fut
3355
1825
1847
1820
1884
1924
1952
2000
1931
1868
--- openCL ---
futhark opencl spMVmult-flat.fut
223
232
276
227
222
276
228
```

```
--- cuda ---
futhark cuda spMVmult-flat.fut
100
110
100
99
100
110
98
98
99
100
```