Exercises Week 1

Anders Wind Steffensen (qrj742@alumni.ku.dk) 18-09-2017

Exercise 1

Theorems: Theorem 1: (map f) . (map g) = map(f . g) Theorem 2: (map f) . (reduce (++) []) = (redude(++) []) . (map(map f)) Theorem 3: (reduce op id) . (reduce (++) []) = (reduce op id) . (map(reduce op id))

distr_p :: [a]->[[a]]

Redomap: redomap op f id = (reduce op id) . (map f) Hint: (reduce (++) []) . distrp = id

Proof: We want to prove:

```
redomap op f id = (reduce op id) . (map(redomap op f id)) . distrp


redomap op f id = (reduce op id) . (map f)
                = (reduce op id) . (map f) . id
                = (reduce op id) . (map f) . (reduce (++) []) . distrp
: by hint
                = (reduce op id) . (reduce (++) []) . (map(map f)) . distrp
: by theorem 2
                = (reduce op id) . (map(reduce op id)) . (map(map f)) . distrp
: by theorem 3
                = (reduce op id) . (map((reduce op id) . (map f)) . distrp
: by theorem 1
```

Proof done.

Exercise 2

For implementation see file "lssp.fut"

To meassure the performance of lssp, I generated the following dataset: $ futhark-dataset --i32-bounds=-50:50 -g [1000000]i32 > data

I then ran(multiple times) both the futhark-c compiled and the futhark-opencl compiled versions and get the following results: GPU (opencl) $ ./lssp -t /dev/stderr 1720 10i32

```
CPU (c)
2082
10i32
```

Where the 10i32 is the result of lssp.

We can see here that the running time of the opencl compiled version is faster than the cpu version. The speedup is not just a factor of adding the cores of the machine, which means that it is probably limited by the accesses to global memory. Had we increased the size of the array we would probably see a larger speedup in the GPU version.

Exercise 3

Program

```
sqrt_primes = primesOpt (sqrt (fromIntegral n))
nested = map (\p ->
    let m = (n 'div' p)
    in map (\j -> j*p) [2..m])
not_primes = reduce (++) [] nested
```

Normalized

```
sqrt_primes = primesOpt (sqrt (fromIntegral n))
nested = map (\p ->
    let m       = n ‘div‘ p in                -- distribute map   // 1
    let mm1     = m - 1 in                    -- distribute map   // 2
    let iot     = iota mm1 in                 -- F rule 4         // 3
    let twom    = map (+2) iot in             -- F rule 2         // 4
    let rp      = replicate mm1 p in          -- F rule 5         // 5
    in map (\(j,p) -> j*p) (zip twom rp)      -- F rule 2         // 6
) sqrt_primes
```

Flattened

```
sqrt_primes = primesOpt (sqrt (fromIntegral n))
F( map (\p -> let m = (n 'div' p) in map (\j -> j*p) [2..m]) ) =
    1. let ms        = map(\p -> n ‘div‘ p) sqrt_primes
    2. let mm1s      = map(\m -> m - 1) ms
    3. let iots      = F( map(\mm1 -> (iota mm1) mm1s) )
    4. let twoms     = F( map(\iot -> map (+2) iot) iots )
    5. let rps       = F( map (\(mm1, p) -> replicate mm1 p) mm1s sqrt_primes )
    6. let nested    = F(map(\(js,ps) -> map (*) js ps)) twoms rps -- assuming
automatic zipping of elements in twoms and rps.

3: using rule 4
F( map(\mm1 -> (iota mm1) mm1s )
    inds = scanexc (+) 0 mmis
    size = reduce (+) 0 mmis
    flag = scatter (replicate size 0) inds mm1s
    tmp = replicate size 1
    iots = sgmScanExc (+) 0 flag tmp

4: using rule 2
F( map(\iot -> map (+2) iot) iots )
    twoms = map(\i -> i +2) iots

5: using rule 5
F( map (\(mm1, p) -> replicate mm1 p) mm1s sqrt_primes )
    vals = scatter (replicate size 0) inds sqrt_primes
    rps = sgmScanInc (+) flag vals

6: using rule 2
F(map(\(js,ps) -> map (*) js ps)) twoms rps
    nested = map (*) twoms rps
```

For the implementation, see "primes-flat.fut"

Discussion

```
primes-naive    (CPU):
    $ echo 10000000 | ./primes-naive -t /dev/stderr > /dev/null
    593387
primes-naive    (GPU):
    echo 10000000 | ./primes-naive -t /dev/stderr > /dev/null
    38996
primes-opt              (CPU):
    $ echo 10000000 | ./primes-opt -t /dev/stderr > /dev/null
    229836
primes-opt              (GPU):
     $ echo 10000000 | ./primes-opt -t /dev/stderr > /dev/null
     **Does not finish within 1 min**
primes-flat             (CPU):
    $ echo 10000000 | ./primes-flat -t /dev/stderr > /dev/null
    329925
primes-flat             (GPU):
    echo 10000000 | ./primes-flat -t /dev/stderr > /dev/null
    98041
```

I am generally seeing some weird results. First of all to compare the CPU versions; *Naive* is the slowest version which makes sense since the depth is "sqrt(n)" and not "lg lg n" like *opt* and *flat*. Naturally we also see an increase in runnning time for *flat* since the flattening requires more operations. The GPU results are difficult to explain. *Naive* is the fastest of the three with an order of magnitude better running time than its own CPU version. The *flat* GPU version is also an improvement over the CPU version but not to the same extend. In *opt* an "unsafe"-keyword had to be added around the last filter to make it compile with opencl. The program did not finish within a reasonable time limit so I omitted the results. I would have expected *flat* to be the fastest of the three given the lower depth and since the flattened form would allow for higher parallelism.

Exercise 4

For implementation, see file "simple.cu"

I typically see such results:

```
CPU Took 132 microseconds (0.13ms)
GPU Took 46 microseconds (0.05ms)
```

The GPU is a factor of ~3 faster than the CPU. Even though the CPU is running sequentially, the faster clock speed and cache control makes it relatively fast at computing the result. The GPU however while faster, does not have a speedup by a factor of the number of cores. The clock speed of each core is of course much lower than the CPU's cores but the explanation is probably that the GPU spends too much time on retrieving/writing to global memory. This means that the math/memory ratio is too low.