Assignment done by :- Gilles Souton fpz747

- Axel Schwarzenbach wgb451

ALL RUNS FOR THIS ASSIGNMENT WERE ON GPU2

1)

Program	Runtime
seq c	~ 325'000 µs
naive c	~ 200'000 µs
flat c	~ 400'000 µs
naive opencl	~ 38'000 µs
flat opencl	~ 30'000 µs

The flat implementation is slower than seq and naive when compiled with c, as we compute more than needed (flag arrays and similar things).

The flat computation is quicker than the naive implementation as it allows for better work distribution on the different threads.

The work depth complexity matches theoretically, but the overheads created by spawning threads and creating the flag arrays, etc. heavily increase the execution time.

```
1  let nn = length sq_primes
2  let flag = map (\i -> if i == 0 then false else true)
   | | | | | | | | ((mkFlagArray mult_lens 0 sq_primes) :> [flat_size]i64)
3  let vals = map (\f -> if f then 0 else 1) flag
4  let iots = sgmSumI64 flag vals
5  let arr = map (+2) iots
   --let flag_mm1 = mkFlagArray mm1s 0 mm1s
   --let flag_sqrn = mkFlagArray mm1s 0 sq_primes
6  let tmp = mkFlagArrayTuple mult_lens (0,0) (zip mult_lens sq_primes) :> [flat_size](i64,i64)
7  let (flag_mm1, flag_sqrn) = unzip tmp
8  let ps = sgmSumI64 (map (\i -> if i == 0 then false else true) flag_mm1) flag_sqrn
9  let composite = map2 (*) ps arr
10 let not_primes = composite
```

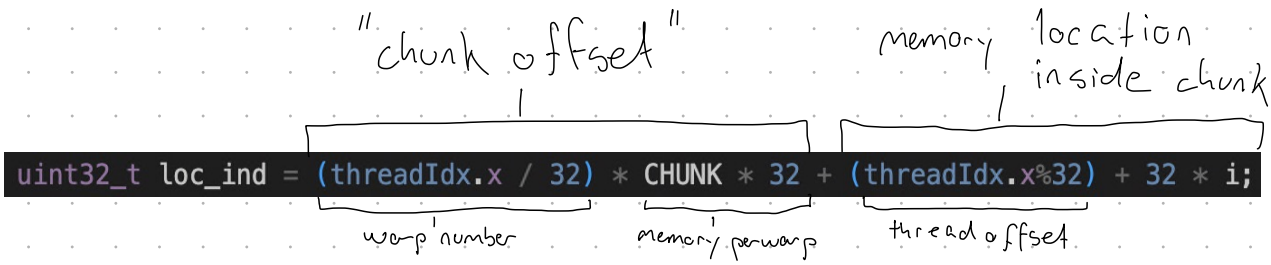
The first for lines is the application of the Map(Iota) flattening rule.

The fifth line is just mapping +2 on the iot array

Lines 6-8 are the application of the Map(Replicate) rule.

Line 9 is applying the Map(Map) rule.

2)



The idea here is for consecutive threads to access consecutive elements in memory, independently of which array it is. The index in global memory is computed by first adding an "offset", which corresponds to the warp number multiplied by the memory used by the arrays accessed in a single warp. To that we add the index inside the warp, which is simply the consecutive indices computed by threadIdx %32 + 32i.

	Naive Reduce +	Opt. Reduce +	Naive Reduce MSSP	Opt. Reduce MSSP	Scan^{Inc}	SgmScan^{Inc}
Non coalesced access	2808 µs	787 µs	20238 µs	7196 µs	11091µs	12528µs
Coalesced Access	2813 µs	787 µs	20220µs	4376µs	3862µs	6405µs
Speedup ⇒	~1	~1	~1	~1.6	~2.9	~2

The coalesed access only affected the last 3 tests with non-negligible speed ups.

3)

```
#pragma unroll
for (int d = 0; d < 5; ++d) {
    int h = 1 << d;
    if ((idx % 32) >= h) {
        ptr[idx] = OP::apply(ptr[idx-h], ptr[idx]);
    }
}
```

This is basically the implementation of the code we saw during Lab 2.

	Naive Reduce +	Opt. Reduce +	Naive Reduce MSSP	Opt. Reduce MSSP	Scan^{Inc}	SgmScan^{Inc}
dummy scan	7835 µs	804 µs	20246 µs	8384 µs	6245 µs	6413 µs
optimized scan	2813 µs	787 µs	20220 µs	4376 µs	3862 µs	6405 µs
speed ~ gps	~1	~1	~1	~2	~1.6	~1

Only the optimized MSSP and Inclusive Scan tests where affected by the optimized implementation. I guess this is because they are the only functions to call up a "standard" inclusive scan. I would guess that the factor would become less for larger arrays.

4) The bug appears in the second step, when placing end of warp results in the first warp. The problem only occurs with block size 1024, because in such a block we would have 32 warps. This implies 32 end-of-warp results and we hence fill the first warp fully. The race condition specifically appears in the ptr array at index 31, because we read from that location (end-of-warp result from first warp) and write a value to it (end-of-warp value of the last warp).

The fix:

```
if (lane == (WARP-1) && idx != 1023) { ptr[warpid] = OP::remVolatile(ptr[idx]); }
__syncthreads();

if (idx == 1023) {ptr[warpid] = OP::remVolatile(ptr[idx]);}
__syncthreads();
```

We ensure that first the value is read from ptr[31] and then written to ptr[31] by placing a _syncthreads() barrier!

5)

```
__global__ void
replicate0(int tot_size, char* flags_d) {
    // ... fill in your implementation here ...
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    if (id < tot_size) {
        flags_d[id] = 0;
    }
}

__global__ void
mkFlags(int mat_rows, int* mat_shp_sc_d, char* flags_d) {
    // ... fill in your implementation here ...
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    if (id < mat_rows) {
        flags_d[mat_shp_sc_d[id]] = 1;
    }
}

__global__ void
mult_pairs(int* mat_inds, float* mat_vals, float* vct, int tot_size, float* tmp_pairs) {
    // ... fill in your implementation here ...
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    if (id < tot_size) {
        tmp_pairs[id] = mat_vals[id] * vct[mat_inds[id]];
    }
}

__global__ void
select_last_in_sgm(int mat_rows, int* mat_shp_sc_d, float* tmp_scan, float* res_vct_d) {
    // ... fill in your implementation here ...
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    if (id < mat_rows) {
        res_vct_d[id] = tmp_scan[mat_shp_sc_d[id]-1];
    }
}
```

```
unsigned int num_blocks     = tot_size / block_size + 1;
unsigned int num_blocks_shp = mat_rows / block_size + 1;
```

CPU-time: 11'773 µs
GPU-time: 2416 µs

⇒ speedup: ~ 4.9