KØBENHAVNS UNIVERSITET

PMPH - Assignment 2

Mikkel Willén, bmq419

29. september 2022

Indhold

Task 1

All tests in all the tasks are run on GPU04.

Sparse-Matrix Vector Multiplication with c			
runs	sequential	naive	parallel
1	334636	209575	421567
2	330112	205502	418266
3	330412	206217	418236
4	331150	205936	422485
5	330359	208181	426574
6	332731	207723	426919
7	330874	210625	427852
8	329498	206084	424498
9	330135	206958	427011
10	331606	209562	425682
avg	331151.3	207636,3	423909

Speeddown is 0.49 when compared to the naive version, when compiled and run with c.

Sparse-Matrix Vector Multiplication with opencl		
runs	naive	parallel
1	32098	11188
2	32323	11207
3	32128	11138
4	32587	11077
5	32161	11111
6	32046	11064
7	32299	11054
8	32278	11075
9	32093	11107
10	32158	11124
avg	32217.1	11114.5

The speedup is 2.90 when compared to the naive version, when compiled with opencl. The sequential version is not tested, since it would take a long long time, for it to run on the GPU.

Task 2

My one-line replacement implementation:

```
uint32_t loc_ind = blockDim.x * i + threadIdx.x;
```

This insures coalesced access to global memory since threadIdx.x denotes the local thread id in the current block, blockDim.x the size of the blocks, and i the current block we are looking at. We can see that, we start by iterating through each thread in the first block, and can therefore access memory in the same SIMD instruktion. Then we iterate through the second block, and so on.
Optimiseret:

```
Naive Memcpy GPU Kernel runs in: x microsecs, GB/sec: 540.58
```

```
Testing Naive Reduce with Int32 Addition Operator:
Reduce GPU Kernel runs in: 1371 microsecs, GB/sec: 145.89
Reduce CPU Sequential runs in: 31530 microsecs, GB/sec: 6.34
Reduce: VALID result!

Testing Optimized Reduce with Int32 Addition Operator:
Reduce GPU Kernel runs in: 410 microsecs, GB/sec: 487.84
Reduce CPU Sequential runs in: 31399 microsecs, GB/sec: 6.37
Reduce: VALID result!

Testing Naive Reduce with MSSP Operator:
Reduce GPU Kernel runs in: 4440 microsecs, GB/sec: 45.05
Reduce CPU Sequential runs in: 245763 microsecs, GB/sec: 0.81
Reduce: VALID result!

Testing Optimized Reduce with MSSP Operator:
Reduce GPU Kernel runs in: 1217 microsecs, GB/sec: 164.35
Reduce CPU Sequential runs in: 246839 microsecs, GB/sec: 0.81
Reduce: VALID result!

Scan Inclusive AddI32 GPU Kernel runs in: 1201 microsecs, GB/sec: 499.62
Scan Inclusive AddI32 CPU Sequential runs in: 54987 microsecs, GB/sec: 7.27
Scan Inclusive AddI32: VALID result!

SgmScan Inclusive AddI32 GPU Kernel runs in: 1602 microsecs, GB/sec: 436.98
SgmScan Inclusive AddI32 CPU Sequential runs in: 161472 microsecs, GB/sec: 2.48
SgmScan Inclusive AddI32: VALID result!
```

Ikke optimiseret:

```
Naive Memcpy GPU Kernel runs in: 740 microsecs, GB/sec: 540.58


Testing Naive Reduce with Int32 Addition Operator:
Reduce GPU Kernel runs in: 1371 microsecs, GB/sec: 145.89
Reduce CPU Sequential runs in: 31563 microsecs, GB/sec: 6.34
Reduce: VALID result!

Testing Optimized Reduce with Int32 Addition Operator:
Reduce GPU Kernel runs in: 410 microsecs, GB/sec: 487.84
Reduce CPU Sequential runs in: 31591 microsecs, GB/sec: 6.33
Reduce: VALID result!

Testing Naive Reduce with MSSP Operator:
Reduce GPU Kernel runs in: 4440 microsecs, GB/sec: 45.05
Reduce CPU Sequential runs in: 245795 microsecs, GB/sec: 0.81
Reduce: VALID result!

Testing Optimized Reduce with MSSP Operator:
Reduce GPU Kernel runs in: 1437 microsecs, GB/sec: 139.19
Reduce CPU Sequential runs in: 238048 microsecs, GB/sec: 0.84
Reduce: VALID result!
```

```
Scan Inclusive AddI32 GPU Kernel runs in: 1595 microsecs, GB/sec: 376.20
Scan Inclusive AddI32 CPU Sequential runs in: 55039 microsecs, GB/sec: 7.27
Scan Inclusive AddI32: VALID result!

SgmScan Inclusive AddI32 GPU Kernel runs in: 1577 microsecs, GB/sec: 443.91
SgmScan Inclusive AddI32 CPU Sequential runs in: 171398 microsecs, GB/sec: 2.33
SgmScan Inclusive AddI32: VALID result!
```

Adding this up we get an average speedup of 1.04.

Task 3

```
template<class OP>
__device__ inline typename OP::RedElTp
scanIncWarp( volatile typename OP::RedElTp* ptr, const unsigned int idx) {
    const unsigned int lane = idx & (WARP - 1);

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

Optimiseret:

```
Naive Memcpy GPU Kernel runs in: 740 microsecs, GB/sec: 540.58


Testing Naive Reduce with Int32 Addition Operator:
Reduce GPU Kernel runs in: 1371 microsecs, GB/sec: 145.89
Reduce CPU Sequential runs in: 31520 microsecs, GB/sec: 6.35
Reduce: VALID result!

Testing Optimized Reduce with Int32 Addition Operator:
Reduce GPU Kernel runs in: 410 microsecs, GB/sec: 487.84
Reduce CPU Sequential runs in: 31441 microsecs, GB/sec: 6.36
Reduce: VALID result!

Testing Naive Reduce with MSSP Operator:
Reduce GPU Kernel runs in: 4440 microsecs, GB/sec: 45.05
Reduce CPU Sequential runs in: 246155 microsecs, GB/sec: 0.81
Reduce: VALID result!

Testing Optimized Reduce with MSSP Operator:
Reduce GPU Kernel runs in: 1227 microsecs, GB/sec: 163.01
Reduce CPU Sequential runs in: 236647 microsecs, GB/sec: 0.85
Reduce: VALID result!

Scan Inclusive AddI32 GPU Kernel runs in: 1201 microsecs, GB/sec: 499.62
Scan Inclusive AddI32 CPU Sequential runs in: 55059 microsecs, GB/sec: 7.27
Scan Inclusive AddI32: VALID result!

SgmScan Inclusive AddI32 GPU Kernel runs in: 1640 microsecs, GB/sec: 426.86
SgmScan Inclusive AddI32 CPU Sequential runs in: 173311 microsecs, GB/sec: 2.31
SgmScan Inclusive AddI32: VALID result!
```

Ikke optimiseret:

```
Naive Memcpy GPU Kernel runs in: 740 microsecs, GB/sec: 540.58


Testing Naive Reduce with Int32 Addition Operator:
Reduce GPU Kernel runs in: 1370 microsecs, GB/sec: 146.00
Reduce CPU Sequential runs in: 31539 microsecs, GB/sec: 6.34
Reduce: VALID result!

Testing Optimized Reduce with Int32 Addition Operator:
Reduce GPU Kernel runs in: 402 microsecs, GB/sec: 497.55
Reduce CPU Sequential runs in: 31504 microsecs, GB/sec: 6.35
Reduce: VALID result!

Testing Naive Reduce with MSSP Operator:
Reduce GPU Kernel runs in: 4441 microsecs, GB/sec: 45.04
Reduce CPU Sequential runs in: 237422 microsecs, GB/sec: 0.84
Reduce: VALID result!

Testing Optimized Reduce with MSSP Operator:
Reduce GPU Kernel runs in: 1939 microsecs, GB/sec: 103.15
Reduce CPU Sequential runs in: 245677 microsecs, GB/sec: 0.81
Reduce: VALID result!

Scan Inclusive AddI32 GPU Kernel runs in: 1360 microsecs, GB/sec: 441.21
Scan Inclusive AddI32 CPU Sequential runs in: 54913 microsecs, GB/sec: 7.28
Scan Inclusive AddI32: VALID result!

SgmScan Inclusive AddI32 GPU Kernel runs in: 1546 microsecs, GB/sec: 452.81
SgmScan Inclusive AddI32 CPU Sequential runs in: 159559 microsecs, GB/sec: 2.51
SgmScan Inclusive AddI32: VALID result!
```

Looking at this data, we can see that some of the tests runs faster on the optimized code, where some run faster on the not-optimized code. We have an average speedup of 1.04, with Testing Optimized Reduce with MSSP Operator and Scan Inclusive AddI32 running faster, and the other generally running slower.

Task 4

The racecondition is on the following line.

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

The problem arises, we have 31 WARPs. In this case, the last warpid can be the same index as the idx. So we don't know if it reads before it writes. We already have a variable with the result value, so we can fix the problem, with the following line:

```
if (lane == (WARP-1)) { ptr[warpid] = res; }
```

where res is the defined further up in the code, and holds the value, we want to right to the specific index.

Task 5

The num_blocks and num_blocks_shp values:

```
unsigned int num_blocks     = (mat_rows * vct_size) / block_size;
unsigned int num_blocks_shp = (mat_rows + block_size - 1) / block_size;
```

Replicate kernel:

```
__global__ void
replicate0(int tot_size, char* flags_d) {
    const unsigned int lid = threadIdx.x; // local id inside a block
    const unsigned int gid = blockIdx.x * blockDim.x + lid; // global id
    if (gid < tot_size) {
        flags_d[gid] = 0;
    }
}
```

Flagarray kernel:

```
__global__ void
mkFlags(int mat_rows, int* mat_shp_sc_d, char* flags_d) {
    const unsigned int lid = threadIdx.x; // local id inside a block
    const unsigned int gid = blockIdx.x * blockDim.x + lid; // global id
    if (gid < mat_rows) {
        flags_d[mat_shp_sc_d[gid]] = 1;
    }
}
```

The kernel for multiplication of the matrix and the vector:

```
__global__ void
mult_pairs(int* mat_inds, float* mat_vals, float* vct, int tot_size, float*
    tmp_pairs) {
    const unsigned int lid = threadIdx.x; // local id inside a block
    const unsigned int gid = blockIdx.x * blockDim.x + lid; // global id
    if (gid < tot_size) {
        tmp_pairs[gid] = mat_vals[gid] * vct[mat_inds[gid]];
    }
}
```

Kernel for selecting the last value of each row.

```
__global__ void
select_last_in_sgm(int mat_rows, int* mat_shp_sc_d, float* tmp_scan, float*
    res_vct_d) {
    const unsigned int lid = threadIdx.x; // local id inside a block
    const unsigned int gid = blockIdx.x * blockDim.x + lid; // global id
    if (gid < mat_rows) {
        res_vct_d[gid] = tmp_scan[mat_shp_sc_d[gid] - 1];
    }
}
```

I ran the following test:

```
Testing Sparse-MatVec Mul with num-rows-matrix: 11033, vct-size: 2076, block size: 256

Vect_size: 2076, tot_size: 11511300 mat_rows: 11033
CPU Sparse Matrix-Vector Multiplication runs in: 12197 microsecs
GPU Sparse Matrix-Vector Multiplication runs in: 830 microsecs
Sparse Mat-Vect Mult VALID   RESULT
```

From this, we can calculate the speedup, which is: 14.7x.