Weekly assignment 2

October 2, 2020

Contents

1 Task 1: Flattened Prime computation 1

1.1 Speedups: 2

2 Task 2: MemCpy but faster 3

2.1 Performance speedup: 4

2.1.1 The slow version 4

2.1.2 The fast version 4

2.1.3 Conclusion 5

2.2 Resubmission 5

3 Task 3: Scanning reimagined 5

3.1 Performance speedup: 6

4 Task 4: Scanning 2: electric bugaloo 6

5 Task 5: Weekly 1 but in C++ 7

5.1 Speedup 9

5.2 Resubmission 9

5.2.1 Correct benchmarking 9

5.2.2 Validation 10

1 Task 1: Flattened Prime computation

I flattened the prime algorithm into the following futhark(only included is the parts we neede to write ourselves, the whole file can be seen in `primes-flat.fut` in the code handin)

```
-- I need to unwrap mm1s
let mm1s = map (\p -> (len / p) - 1) sq_primes
-- Create flag arr
let inds = scan (+) 0 mm1s
let mm1s_rot = map (\i -> if i == 0 then 0 else inds[i-1]) (indices sq_primes)
let flags = scatter (replicate flat_size 0) inds sq_primes
-- Finally unroll the iota in map
let mm1s_tmp = replicate flat_size 1
let mm1s_flat = sgmSumI32 flags mm1s_tmp
-- flat list of [2..m]
let mm1sp2 = map (+1) mm1s_flat
-- I need to replicate all of sq_prime inds times
-- I can use the flags array to get the indexes of "new" primes
let flags_ones = map (\x -> if x == 0 then 0 else 1) flags
let p_inds = scan (+) 0 flags_ones
let p_expanded = map (\i -> sq_primes[i]) p_inds

let not_primes = map2 (*) mm1sp2 p_expanded
```

1.1 Speedups:

On the GPU04 machine i ran:

```
[wbr220@a00333 primes-futhark]$ futhark c primes-naive.fut
[wbr220@a00333 primes-futhark]$ futhark c primes-seq.fut
[wbr220@a00333 primes-futhark]$ futhark opencl primes-naive.fut -o primes-naive-opencl
[wbr220@a00333 primes-futhark]$ futhark opencl primes-flat.fut
[wbr220@a00333 primes-futhark]$ echo "1000000" | ./primes-naive -t /dev/stderr -r 10 >
6909
...
6974
[wbr220@a00333 primes-futhark]$ echo "1000000" | ./primes-seq -t /dev/stderr -r 10 > /
17923
...
18639
[wbr220@a00333 primes-futhark]$ echo "1000000" | ./primes-naive-opencl -t /dev/stderr -
10669
...
10629
```

```
[wbr220@a00333 primes-futhark]$ echo "1000000" | ./primes-flat -t /dev/stderr -r 10 > /
1174
...
1139
```

And as expected, the flat version computes much faster than its counterparts. Noted as the slowest is the sequential version, calculating on the c backend, which is also expected.

2 Task 2: MemCpy but faster

In this task i am asked to improve a written "Copy from/to Global memory from/to Shared memory. The given function(and replacement) is shown below:

```
template<class T, uint32_t CHUNK>
__device__ inline void
copyFromGlb2ShrMem( const uint32_t glb_offs
  , const uint32_t N
  , const T& ne
  , T* d_inp
  , volatile T* shmem_inp
) {
    #pragma unroll
    for(uint32_t i=0; i<CHUNK; i++) {
// Was
// uint32_t loc_ind = threadIdx.x *chunk + i;
uint32_t loc_ind = threadIdx.x + (i*blockDim.x);
// Is now
uint32_t glb_ind = glb_offs + loc_ind;
T elm = ne;
if(glb_ind < N) {
    elm = d_inp[glb_ind];
}
shmem_inp[loc_ind] = elm;
    }
    __syncthreads(); // leave this here at the end!
}
```

Coalesced access requires each thread in a warp to execute sequential reads. To get this i had to find a way to index while adding idx: $c + idx$ instead of $idx * c1 + c2$, using `idx` as part of a multiplication. With the rewrite shown above, we have a constant($i * blockDim$)in the context of this single instruction) being added to the threadIds in every warp. Thus every thread in the warp will fetch a constant and it's id, resulting in 32 sequential integers being fetched from memory. Thus we get the added coalescence bonus.

2.1 Performance speedup:

2.1.1 The slow version

```
[wbr220@a00333 cuda]$ make
...
Reduce GPU Kernel runs in: 3214 microsecs, GB/sec: 62.23
Reduce CPU Sequential runs in: 233958 microsecs, GB/sec: 0.85
Reduce: VALID result!

Scan Inclusive AddI32 GPU Kernel runs in: 2531 microsecs, GB/sec: 237.08
Scan Inclusive AddI32 CPU Sequential runs in: 42051 microsecs, GB/sec: 9.51
Scan Inclusive AddI32: VALID result!

SgmScan Inclusive AddI32 GPU Kernel runs in: 1772 microsecs, GB/sec: 395.06
SgmScan Inclusive AddI32 CPU Sequential runs in: 147676 microsecs, GB/sec: 2.71
SgmScan Inclusive AddI32: VALID result!
```

2.1.2 The fast version

```
[wbr220@a00333 cuda]$ make
...
Reduce GPU Kernel runs in: 2835 microsecs, GB/sec: 70.55
Reduce CPU Sequential runs in: 232041 microsecs, GB/sec: 0.86
Reduce: VALID result!

Scan Inclusive AddI32 GPU Kernel runs in: 1791 microsecs, GB/sec: 335.03
Scan Inclusive AddI32 CPU Sequential runs in: 40986 microsecs, GB/sec: 9.76
Scan Inclusive AddI32: VALID result!

SgmScan Inclusive AddI32 GPU Kernel runs in: 1693 microsecs, GB/sec: 413.50
SgmScan Inclusive AddI32 CPU Sequential runs in: 145349 microsecs, GB/sec: 2.75
SgmScan Inclusive AddI32: VALID result!
```

2.1.3 Conclusion

The speedup is easily viewable in tests above, while no significant speedup is seen in the Int32 operators. I assume this is because only those functions make efficient use of the Global vs Shared memory

2.2 Resubmission

For the resubmission of this task i want to describe the methodology of these benchmarks. All of the tests described above have been run on the `gpu04` machine with the following stats:

```
Device name: GeForce RTX 2080 Ti
Number of hardware threads: 69632
Max block size: 1024
Shared memory size: 49152
```

To run the benchmarks, i use the `test-pbb` executable with an array length of 50003565 and a block size of 128. Later on when testing the race condition that occurs, i set the block size to 1024, however to avoid exhausting gpu resources all other benchmarks run with a blocksize of 128.

3 Task 3: Scanning reimagined

In this task i implement a parallel inclusive scan at WARP level. The algorithm to be implemented is explained in detail in the lecture notes for the second lab class, so the actual implementation is pretty trivial

```
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++){ // Run 5 times deep
int h = 1 << d; // h = 2^d
if (lane >= h){// Only execute on the last 2^d threads in warp
    ptr[idx] = OP::apply(ptr[idx-h], ptr[idx]);
}

    }
    return OP::remVolatile(ptr[idx]);
}
```

3.1 Performance speedup:

After implementing the previous two optimizations, i present the following table: Where `C = coalesced reads` and `W = warp-level optimization`, and all results in microsecs

	!C !W	C !W	!C W	C W
Naive Reduce<int32> add	1371	1371	1371	1371
Optimized Reduce<int32 add	401	401	410	410
Naive Reduce<mssp>	4439	4439	4439	4439
Optimized Reduce<mssp>	3189	2833	2086	1732
Scan Inclusive AddI32	2515	1791	1763	1278
SgmScan Inclusive AddI32	1768	1694	1766	1695

4 Task 4: Scanning 2: electric bugaloo

In the function `scanIncBlock` we spot a nasty race condition when running with the max size of blocks(1024), the code in question looks like this initially:

```
template<class OP>
__device__ inline typename OP::RedElTp
scanIncBlock(volatile typename OP::RedElTp* ptr, const unsigned int idx) {
    const unsigned int lane   = idx & (WARP-1);
    const unsigned int warpid = idx >> lgWARP;
    // 1. perform scan at warp level
    typename OP::RedElTp res = scanIncWarp<OP>(ptr,idx);
    __syncthreads();
    // 2. place the end-of-warp results in the first warp.
    if (lane == (WARP-1)) {
ptr[warpid] = OP::remVolatile(ptr[idx]);
    }
    __syncthreads();
    // 3. scan again the first warp
    if (warpid == 0) scanIncWarp<OP>(ptr, idx);
    __syncthreads();
    // 4. accumulate results from previous step;
    if (warpid > 0) {
res = OP::apply(ptr[warpid-1], res);
    }
    return res;
}
```

This being a race condition i start looking at step 2, since it contains both a read and a write in the same parallel instruction. Intuitively this can lead to some nasty bugs. Looking at the code, and knowing that the condition only occurs when `bs=1024` i look at which array indices are calculated at this specific warp. I discover that when we are in the very last warp, the following holds: $idx = 1023$ and $warpid = \frac{idx+1}{32} - 1 = 31$. The trouble occurs because the last thread of the first warp has the values $idx = 31, \quad warpid = 0$. When executing the code, the following two instructions are executed at the same time(although not in lockstep):

```
ptr[0]  = ptr[31]
ptr[31] = ptr[1023]
```

And the race condition is now obvious. To fix this issue, i looked at step 4 where we pull out the values again, and realize that after the scan `ptr[31]` is never actually read out again, since it reads `ptr[warpid-1]` and there will never be a `warpid = 32`. A simple but working solution is then presented: just ignore the last warp when scanning in step three, since that value will never be used. The solution is then presented:

```
template<class OP>
__device__ inline typename OP::RedElTp
scanIncBlock(volatile typename OP::RedElTp* ptr, const unsigned int idx) {
    ...
    // 2. place the end-of-warp results in the first warp.
    if (lane == (WARP-1) && warpid < 31) {
ptr[warpid] = OP::remVolatile(ptr[idx]);
    }
    ...
    return res;
}
```

Another more robust fix could be to read the result into a temporary variable, sync the threads and the write it back in a seperate call. This does however add a possibly expensive call to "syncthreads" and an extraneous if statement.

5 Task 5: Weekly 1 but in C++

In the first weekly assignment, we wrote a flat version of sparse-matrix vector multiplication in Futhark. In this assignment we write the same flat algorithm in Cuda, using four kernels:

```
__global__ void
replicate0(int tot_size, char* flags_d) {
    // Fill the flags d [tot_size]char array with 0
    uint32_t gid = blockIdx.x * blockDim.x + threadIdx.x;
    if (gid < tot_size){
flags_d[gid] = 0;
    }
}

__global__ void
mkFlags(int mat_rows, int* mat_shp_sc_d, char* flags_d) {
// Essentially a cpp version of
/// let row_flg  = scatter (replicate num_elms 0) shp_rot (replicate num_rows 1)
// Creating an array of flags corresponding to the scanned matrix shape array
    uint32_t gid = blockIdx.x * blockDim.x + threadIdx.x;
    if (gid < mat_rows){
if(gid == 0) // Base case
    flags_d[gid] = 0;
else{
    // Since we really want the right-shifted array, use
    // gid-1 instead of gid
    int tmp = mat_shp_sc_d[gid-1];
    flags_d[tmp] = 1;
}
    }
}

__global__ void
mult_pairs(int* mat_inds, float* mat_vals, float* vct, int tot_size, float* tmp_pairs)
    // Perform the actual matrix multiplication
    uint32_t gid = blockIdx.x * blockDim.x + threadIdx.x;
    if (gid < tot_size){
// mat_inds holds the column of the value
// Multiply $val * vct[column]
tmp_pairs[gid] = mat_vals[gid] * vct[mat_inds[gid]];
    }
}

__global__ void
select_last_in_sgm(int mat_rows, int* mat_shp_sc_d, float* tmp_scan, float* res_vct_d)
```

```
    uint32_t gid = blockIdx.x * blockDim.x + threadIdx.x;
    // Extract the last value of each row
    if (gid < mat_rows){
// mat_shp_sc_d[gid] holds the index of the start of a row
// mat_shp_sc_d[gid]-1 is then be the index of the end of a row
res_vct_d[gid] = tmp_scan[mat_shp_sc_d[gid] - 1];
    }
}
```

We also have to calculate the amount of blocks for our kernel calls:

```
// Replicate0 and mult_pais
unsigned int num_blocks     = ((tot_size + (block_size - 1)) / block_size);
// mkflags and select_last...
unsigned int num_blocks_shp = ((mat_rows + (block_size - 1)) / block_size);
```

The code above is taken from equivalent calculations in weekly 1

5.1 Speedup

Running `make run-spmv` we see a very obvious speedup:

```
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: 11499 microsecs
GPU Sparse Matrix-Vector Multiplication runs in: 16 microsecs
Sparse Mat-Vect Mult VALID   RESULT
```

5.2 Resubmission

5.2.1 Correct benchmarking

For the resubmission, my TA correctly pointed out a flaw in how the benchmarks are run. I accidentally a for loop, so only one run would be completed, yet the final time still divided by the GPU_{RUNS} variable. The Correct benchmark now looks like this:

```
./test-spmv 11033 2076 256
Device name: GeForce RTX 2080 Ti
Number of hardware threads: 69632
Max block size: 1024
Shared memory size: 49152
```

```
====
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: 12309 microsecs
GPU Sparse Matrix-Vector Multiplication runs in: 840 microsecs
Sparse Mat-Vect Mult VALID   RESULT
```

This benchmark is run on the `gpu04` machine, with gpu specs visible in the benchmark above. The inputs to the program is also visible as `num-rows: 11033, vct-size: 2076, block size: 256`

5.2.2 Validation

To ensure the program is valid, aside from a visual inspection of the code, we run the `test-spmv` program. This program completes both a CPU-sequential run and a GPU run, and validates that both versions compute the same result, using the CPU resulting vector as the guiding answer.