Weekly assignment 2

Christian P.

September 18, 2020

Contents

1 TODO Task 1

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 TODO Performance speedup:

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 TODO Performance speedup:

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]);
    }
    // 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