Weekly Assignment 2 - Report

Francesco Done'
qwg586@alumni.ku.dk

September 17, 2020

Contents

1 Task 1: Flat Implementation of Prime-Numbers Computation in Futhark

Here below there is the code implemented in `primes-flat.fut`:

```
-- primes-flat.fut
-- ...
--  [BEGIN A]: distribute map
   let ms = map (\ p->len/p) sq_primes
--  [END A]
--  [BEGIN B]: distribute map
   let mm1s = map (\ m->m-1) ms
--  [END B]
--  [BEGIN C]: F(Map(Iota))
   -- unflattened version: let iots = map (\ mm->(iota mm)) mm1s
   let len1 = length mm1s
   let flag1 = mkFlagArray mm1s 0 mm1s
   let vals1 = map (\ f-> if f != 0
   then 0
   else 1
   ) flag1
   let iots = segmented_scan (+) 0i32 flag1 vals1
--  [END C]
--  [BEGIN D]: F(Map(Map))
   -- unflattened version: let arrs = map (\ i->(map (+2) i) iots)
   let arrs = map (+2) iots
--  [END D]
--  [BEGIN E]: F(Map(Replicate))
   -- unflattened version: let pss = map (\ n p ->(replicate n p)) mm1s sq_primes
   -- flattened version 1
   -- let (flag_n, flag_p) =
   --                unzip <|
   --                mkFlagArray mm1s (0,0) <|
   --                zip mm1s sq_primes
   -- flattened version 2
   let flag_p = mkFlagArray mm1s 0 sq_primes
   let pss = segmented_scan (+) 0i32 flag1 flag_p
--  [END E]
--  [BEGIN F]: F(Map(Map))
   let composite = map (*) pss arrs
--  [END F]
let not_primes = flatten composite
-- let not_primes = replicate flat_size 0
-- ...
```

Unfortunately it doesn't work, even trying to cast the `flag_p` array.
By the way, the initial code was:

```
let composite = -- uses nested parallelism
map (\ p ->  let m = len / p -- distribute map [A]
             let mm1 = m - 1 -- distribute map [B]
             let iot = iota mm1 -- F(Map(Iota)) [C]
             let arr = map (+2) iot -- F(Map(Map)) [D]
             let ps = replicate mm1 p -- F(Map(Replicate)) [E]
             in map2 (*) ps arr -- F(Map(Map)) [F]
) sqrn_primes
let not_primes = reduce (++) [] composite -- noop because composite
-- is in flat - data form
```

As you can see, in the initial code there were labeled some lines of code and in the implemented code there are the references.

2 Task 2: Copying from/to Global to/from Shared Memory in Coalesced Fashion

2.1 Code

The replaced line in `copyFromGlb2ShrMem()` and `copyFromShr2GlbMem()` is:

```
// pbbKernels.cu.h
// ...
uint32_t loc_ind = (blockDim.x * i) + threadIdx.x;
//uint32_t loc_ind = threadIdx.x * CHUNK + i;
// ...
```

The code above ensures coalesced access to global memory as demonstrate in the table below:

threadId	**CHUNK**	**i**	**memory_index**
0	3	[0,1,2]	[0,1,2]
1	3	[0,1,2]	[3,4,5]
2	3	[0,1,2]	[6,7,8]
3	3	[0,1,2]	[9,10,11]
4	3	[0,1,2]	[12,13,14]

Table 1: Uncoalesced access

In Table 1, two consecutive threads (e.g. `threadId==0` and `threadId==1`) are accessing the memory in positions `memory_index==0` and `memory_index==3` at the same iteration `i==0`. Therefore it is an uncoalesced access.

threadId	**CHUNK**	**i**	**memory_index**
0	3	[0,1,2]	[0,5,10]
1	3	[0,1,2]	[1,6,11]
2	3	[0,1,2]	[2,7,12]
3	3	[0,1,2]	[3,8,13]
4	3	[0,1,2]	[4,9,14]

Table 2: Coalesced access

In Table 2, the same consecutive threads (`threadId==0` and `threadId==1`) are accessing the memory in positions `memory_index==0` and `memory_index==1` at the same iteration `i==0`, so it is a coalesced access. This happens because in the new formula we are considering the block dimension.

2.2 Performance

In table below there are shown the performances with and without Task 2 implementation. As you can see, the implementation of task 2 permits a better performance and where its presence is useless, the result is the same.

Test	μs w/	μs w/o	GB/s w/	GB/s w/o
Naive Memcpy GPU Kernel	1535	1535	260.60	260.60
Reduce GPU Kernel	2831	2830	70.65	70.68
Reduce CPU Sequential	30860	40940	6.48	4.89
Reduce GPU Kernel	822	822	243.33	243.33
Reduce CPU Sequential	30644	31391	6.53	6.37
Reduce GPU Kernel	19796	19812	10.1	10.1
Reduce CPU Sequential	232298	232005	0.86	0.86
Reduce GPU Kernel	28052	29395	7.13	6.8
Reduce CPU Sequential	232382	232461	0.86	0.86
Scan Inclusive AddI32 GPU Kernel	15715	22424	38.18	26.76
Scan Inclusive AddI32 CPU Sequential	41068	42114	9.74	9.5
SgmScan Inclusive AddI32 GPU Kernel	6066	12290	115.41	56.96
SgmScan Inclusive AddI32 CPU Sequential	144471	145535	2.77	2.75

Table 3: Performance differences with (w/) and without (w/o) implementation of task 2

3 Task 3: Implement Inclusive Scan at WARP Level

3.1 Code

The idea is that our function should start to apply the OP from the end of the array rather than the beginning, because it writes in the same positions where it will then read: there aren't two array, one for reading and one for writing data.

```
//pbbKernels.cu.h
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);
   if(lane==0) {
      int h=0;
      int i=WARP-1;
      #pragma unroll
      for(int d=0; d<lgWARP; d++){
         h = (int) powf(2,d);
         if(i>=h){
            while(i>=h){
               ptr[idx+i] = OP::apply(ptr[idx+i-h], ptr[idx+i]);
               i--;
            }
         }
         i=WARP-1;
      }
      /*for(int i=0; i<WARP; i++) {
      ptr[idx+i] = OP::apply(ptr[idx+i-1], ptr[idx+i]);
      }*/
   }
   return OP::remVolatile(ptr[idx]);
}
```

3.2 Performance

In table below there are shown the performances with and without Task 3 implementation.

Test	**μs w/**	**μs w/o**	**GB/s w/**	**GB/s w/o**
Naive Memcpy GPU Kernel	1535	1534	260.6	260.77
Reduce GPU Kernel	2831	2797	70.65	71.51
Reduce CPU Sequential	30860	35856	6.48	5.58
Reduce GPU Kernel	822	799	243.33	250.33
Reduce CPU Sequential	30644	35989	6.53	5.56
Reduce GPU Kernel	19796	19820	10.1	10.09
Reduce CPU Sequential	232298	234302	0.86	0.85
Reduce GPU Kernel	28052	8248	7.13	24.25
Reduce CPU Sequential	232382	240837	0.86	0.83
Scan Inclusive AddI32 GPU Kernel	15715	6130	38.18	97.89
Scan Inclusive AddI32 CPU Sequential	41068	44714	9.74	8.95
SgmScan Inclusive AddI32 GPU Kernel	6066	6074	115.41	115.25
SgmScan Inclusive AddI32 CPU Sequential	144471	144897	2.77	2.76

Table 4: Performance differences with (w/) and without (w/o) implementation of task 3