1 Task 1

In your report:

- Please state whether your implementation validates on both datasets.

  my implementation validates on both datasets.

- Present the code that you have added and briefly explain it, i.e., its correspondence to the flattening rules.

```
-- we can implement a flattened iota version with
-- a composition of a segmented scan
let flattened_iota [n] (mult_lens: [n]i64) : []i64 =
  let rp = replicate n 1i64
  let flag = mkFlagArray mult_lens 0i64 rp
  let vals = map (
    \ f -> if f!=0 then 0 else 1
  ) flag
  in sgmScan (+) 0 flag vals

-- we can implement a flattened replicate version with a segmented scan
let flattened_replicate [n] (mult_lens: [n]i64, values : [n]i64) : []i64 =
  let (flag_n,flag_v) = zip mult_lens values
    |> mkFlagArray mult_lens (0i64, 0i64)
    |> unzip
  in sgmScan (+) 0 flag_n flag_v

....

-- the flattened implementation
let iots = flattened_iota mult_lens :> [flat_size]i64
let twoms = map (+2) iots -- we add 2 to each element

-- replicate the primes array
let replicate_primes = flattened_replicate (mult_lens, sq_primes) :> [flat_size]i64

-- we use a map2 to multiply each element of twoms by the corresponding
--element of replicate_primes
let not_primes = map2 (\j p ->
  j*p
) twoms replicate_primes :> [flat_size]i64
```

  flattening rules applied:

  much of the code for flattened iota and replicate are taken from lecture-notes Line 53-55

  flattened iota(rule 4): can be rewritten as a composition composition of scan and scatter: flattened replicate(rule 3): can be rewritten as a composition of a composition of scan and scatter

  However note that our implementation is using the segmented scan instead of scan and scatter. We can do this as a scan + scatter can be rewritten as a segmented scan.

- Report the runtimes of all four code versions for the large dataset and try to briefly explain them, i.e., do they match what you would expect from their work-depth complexity?

Filename	Backend	mean Runtime (us)
primes-flat.fut	cuda	3102
primes-adhoq.fut	cuda	182061.6
primes-naive.fut	cuda	50080.4
primes-seq.fut	c	187043.9

Table 1: Runtime for large dataset (10000000i64)

we note that the flat implementation although it theoretically has worse work and depth complexity compared to adhoq implementation is much more performant.

2 Task 2

Please write in your report:

we do the following replacement:

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

With loc_ind = i * blockDim.x + threadIdx.x, consecutive threads within a warp (which have consecutive threadIdx.x values) will access consecutive memory locations. So the memory access pattern happens with a stride of 1.

.. Explain to what extent your one-line replacement has affected the performance, i.e., which tests and by what factor.

from the test results we see the following performance improvements:

Operation	Non-Coalesced (GB/s)	Coalesced (GB/s)	Improvement
Naive Reduce (AddI32)	323.90	309.13	-4.56%
Optimized Reduce (AddI32)	1020.44	1023.05	+0.26%
Naive Reduce (MSSP)	110.65	110.68	+0.03%
Optimized Reduce (MSSP)	180.27	276.06	+53.14%
Scan Inclusive (AddI32)	287.85	643.80	+123.66%
SgmScan Inclusive (AddI32)	457.53	1032.49	+125.67%

Table 2: Performance impact of coalesced memory access (Task 2)

we note that the scan computations were most affected by the change.

3 Task 3

Solution:

```
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 < 5; d++) {
        int h = 1 << d; // 2^d double the stride
        if (lane >= h) {
            ptr[idx] = OP::apply(ptr[idx - h], ptr[idx]);
        }
    }

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

Explain the performance impact of your implementation: which tests were affected and by what factor. Does the impact become higher for smaller array lengths?

GPU Kernel	GB/sec		Improvement
	Unoptimized	Optimized	
Reduce (Int32 Addition)	325.74	304.42	-6.54%
Optimized Reduce (Int32 Addition)	990.13	1033.63	+4.39%
Reduce (MSSP)	107.70	108.70	+0.93%
Optimized Reduce (MSSP)	110.72	183.83	+66.03%
Scan Inclusive AddI32	540.32	798.43	+47.77%
SgmScan Inclusive AddI32	837.85	837.35	-0.06%

Table 3: Comparison of Unoptimized and Optimized GPU Kernel Implementations

we compare the two kernels for various array sizes: in the below table coalesced refers to the performance when including optimizations from task 2. Naive and Opt refers to the warp level scan implementation vs the thread level scan implementation.

we observe that the performance increase is most prominent when the array size is large. and the relative performance gain decreases as the array size decreases.

4 Task 4

Please explain in the report:

Operation	13,565		103,565		1,003,565		10,003,565	
	GB/s	% Diff	GB/s	% Diff	GB/s	% Diff	GB/s	% Diff
Naive Reduce AddI32 (UnCoalesced)	0.94	-8.5%	5.92	-9.1%	40.14	+3.1%	165.35	+9.0%
Naive Reduce AddI32 (Coalesced)	0.86		5.38		41.38		180.24	
Opt Reduce AddI32 (UnCoalesced)	3.88	+27.1%	27.62	+36.4%	160.57	+78.4%	625.22	+25.5%
Opt Reduce AddI32 (Coalesced)	4.93		37.66		286.73		784.59	
Naive Reduce MSSP (UnCoalesced)	0.92	-23.9%	4.87	-15.8%	25.25	+1.9%	57.82	+8.3%
Naive Reduce MSSP (Coalesced)	0.70		4.10		25.73		62.62	
Opt Reduce MSSP (UnCoalesced)	1.55	+29.7%	10.90	+40.7%	51.46	+62.5%	79.08	+65.9%
Opt Reduce MSSP (Coalesced)	2.01		15.34		83.63		131.19	
Scan Inc AddI32 (UnCoalesced)	6.03	+17.4%	42.85	+26.1%	236.13	+50.0%	412.52	+55.6%
Scan Inc AddI32 (Coalesced)	7.08		54.03		354.20		641.94	
SgmScan Inc AddI32 (UnCoalesced)	8.26	-20.7%	60.41	-17.2%	226.61	+1.6%	434.94	-0.3%
SgmScan Inc AddI32 (Coalesced)	6.55		50.00		230.33		433.59	

Table 4: Performance comparison (GB/s) between optimized and unoptimized versions for different array sizes

- The nature of the bug.

  The race condition occurs because:

  1. For the last thread in each warp (lane 31), ‘warpid‘ is equal to the warp number (0 to 31). 2. For these threads, ‘idx‘ points to the last element of their respective warp’s section in ‘ptr‘. 3. When warp 31 executes this code: - It reads from ‘ptr[1023]‘ (its last thread’s ‘idx‘) - It writes to ‘ptr[31]‘ (its ‘warpid‘) 4. However, ‘ptr[31]‘ is also where the last thread of warp 0 should store its result.

  This creates a race condition between the last thread of warp 31 and the last thread of warp 0. The final value in ‘ptr[31]‘ becomes unpredictable, as it depends on which thread writes last.

  ```
  // Place the end-of-warp results into a separate location in memory.
  typename OP::RedElTp end = OP::remVolatile(ptr[idx]);
  // synchronize the threads so that every thread has stored
  // the value in the memory location before we write
  __syncthreads();
  if (lane == (WARP - 1)) {
       ptr[warpid] = end;
  }
  __syncthreads();
  ```

  the bug appears only when warpid is 31 and we are in the 32nd warp ie idx = 1024. this is because in that instance we will have a race condition between thread 1024 in the block and warp 32.

- How you fixed it.

We fix the issue by using a temporary value to store the value then synchronizing the threads. and writing to the memory location only after the synchronization. In this way we avoid the race condition.

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

When compiling/running with block size 1024, remember to set the value of...

5 Task 5

- Implement the four kernels of file `spmv_mul_kernels.cuh` and two lines in file `spmv_mul_main.cu` (at lines 155-156).
- Add your implementation in the report (it is short enough) and report speedup/slowdown vs sequential CPU execution.

  code for determining number of blocks:

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

#ifndef SPMV_MUL_KERNELS
#define SPMV_MUL_KERNELS

__global__ void replicate0(int tot_size, char* flags_d) {
    uint32_t gid = blockIdx.x * blockDim.x + threadIdx.x;
    if (gid < tot_size) {
        flags_d[gid] = 0;
    }
}

__global__ void
mkFlags(
    int mat_rows, // number of rows
    int* mat_shp_sc_d, // the scanned shape array
    char* flags_d // flags array
) {
    uint32_t gid = blockIdx.x * blockDim.x + threadIdx.x;
    if (gid < mat_rows) {
        int start_idx = (gid == 0) ? 0 : mat_shp_sc_d[gid - 1];
        flags_d[start_idx] = 1;
    }
}

__global__ void
mult_pairs(
    int* mat_inds,
    float* mat_vals,
```

```
    float* vct,
    int tot_size,
    float* tmp_pairs
) {
    uint32_t gid = blockIdx.x * blockDim.x + threadIdx.x;
    if (gid < tot_size) {
        tmp_pairs[gid] = mat_vals[gid] * vct[mat_inds[gid]];
    }
}

__global__ void
select_last_in_sgm(
    int mat_rows,
    int* mat_shp_sc_d, // the shape array segmented scan
    float* tmp_scan, // the scan
    float* res_vct_d // store the result
) {
    uint32_t gid = blockIdx.x * blockDim.x + threadIdx.x;
    if (gid < mat_rows) {
        res_vct_d[gid] = tmp_scan[mat_shp_sc_d[gid] - 1];
        // the read from temp_scan the last element of each segment
    }
}

#endif // SPMV_MUL_KERNELS
```

On an nvidia a100 pcie 40gb, the following results were obtained: Testing Sparse-MatVec Mul with num-rows-matrix: 11033, vct-size: 2076, block size: 256

CPU Sparse Matrix-Vector Multiplication runs in: 19584 microsecs GPU Sparse Matrix-Vector Multiplication runs in: 447 microsecs speedup: 19584/447 = 43.86