1 Task 1

In your report:

- Please state whether your implementation validates on both datasets.
- Present the code that you have added and briefly explain it, i.e., its correspondence to the flattening rules.
- 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?

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 1: 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
scanIncBlock(volatile typename OP::RedElTp* ptr, const unsigned int idx) {
    const unsigned int lane     = idx & (WARP - 1);
```

```
    const unsigned int warpid  = idx >> lgWARP;
    const unsigned int n_warps = (blockDim.x + WARP - 1) >> lgWARP; // Total num

    // 1. Perform scan at warp level.
    typename OP::RedElTp res = scanIncWarp<OP>(ptr, idx);
    __syncthreads();

    // 2. Place the end-of-warp results into a separate location in shared memor
    if (lane == (WARP - 1)) {
        ptr[blockDim.x + warpid] = OP::remVolatile(ptr[idx]);
    }
    __syncthreads();

    // 3. Let the first warp scan the per-warp sums.
    if (warpid == 0 && idx < n_warps) {
        scanIncWarp<OP>(ptr + blockDim.x, idx);
    }
    __syncthreads();

    // 4. Accumulate results from the previous step.
    if (warpid > 0) {
        res = OP::apply(ptr[blockDim.x + warpid - 1], res);
    }

    return res;
}
```

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 2: Comparison of Unoptimized and Optimized GPU Kernel Implementations

we compare the two kernels for various array sizes:

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.

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

Table 3: Performance comparison (GB/s) between optimized and unoptimized versions for different array sizes

4 Task 4

Please explain in the report:

- The nature of the bug.

  the nature of the bug is a race condition that occurs between thread number 1024 and this is because within this code block we are both reading and writing to ptr.

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

  the bug appears only w

- How you fixed it.

```
if (lane == (WARP - 1)) {
    // we offset the index by blockDim.x to avoid race conditions
    ptr[blockDim.x + warpid] = OP::remVolatile(ptr[idx]);
}
```

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, flo
    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 fr
    }
}

#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