Final Report

Fatime Houjaij, Safaa Diab, Sara Katerji

May 2020

Problem Description

Neural Networks is a supervised learning technique used in a lot of AI tasks like classifications, approximations, regression analysis, etc. It is defined as a network composed of several neurons inspired by the human brain. As depicted in Figure 1, a neural network is typically composed of an input layer, several hidden layers and an output layer.

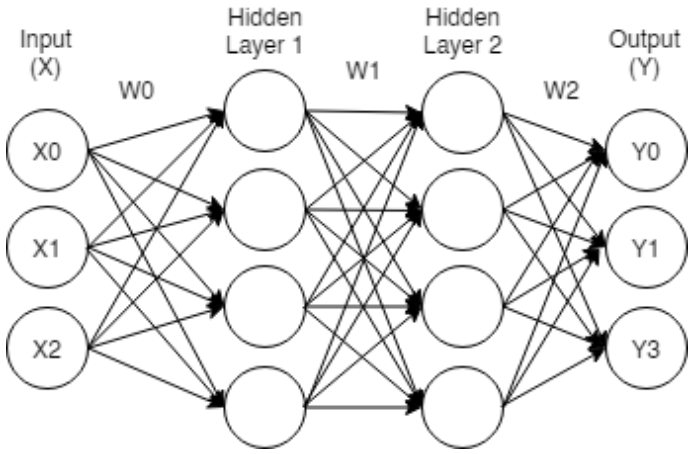

Figure 1: An example of Neural Network

The aim of this project is to implement a sparse multilayer perceptron with batching on GPUs and optimize it. In other words, given an input layer and weights matrices, we are going to calculate the output layer values on the GPU with batching. Since, our matrices are sparse and we are implementing batching, we will be using sparse matrix matrix multiplication. At the end of the project, we also aim to compare the results of different optimized GPU implementations against a sequential implementation.

Sequential Implementation

Since we are taking the default project, the sequential implementation and the neural network dataset was provided by the Instructor. The detailed explanation of the sequential coded is provided below. The sequential code starts by reading the dataset and it takes two types of data: input layer data and weights data represented as COO Matrices. These steps are presented in Figure 2.

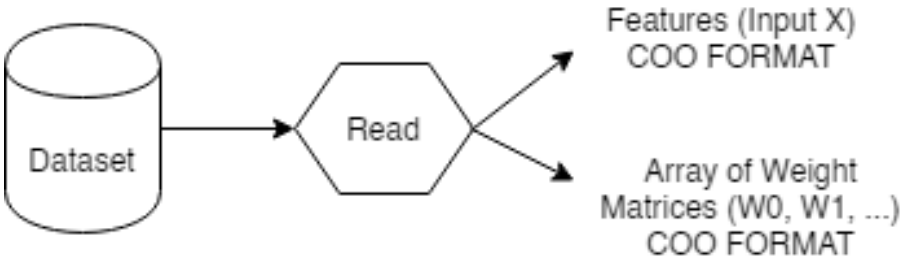

Figure 2: Preprocessing dataset

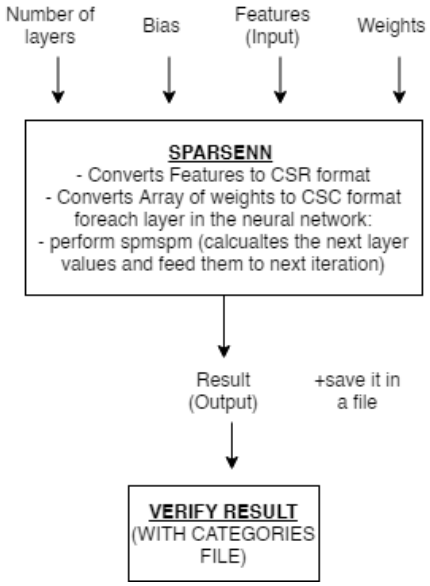

Figure 3: SPARSENN and VERIFY RESULT

After reading and preparing the data, we pass the number of layers, bias, features (input in COO format) and array of weights (in COO format) to the SPARSENN kernel. The SPARSENN kernel first converts both the input (features) and the array of weights from COO format to CSR and CSC respectively. Then, it loops through all layer of the neural network and performs sparse ma-

trix matrix multiplication (spmspm) on each layer. The goal of SPMSPM is to calculate the next layer values and feed them to the next iteration. Moreover, to make the process easier, we make use of double buffers (inbuffer, outbuffer). Thus, at the end of each iteration, the outbuffer and inbuffer values are swapped (output becomes input in the next iteration) until reaching the last iteration. Finally, SPARSENN returns a result vector with all the non zero rows. This vector is saved to a file and then checked with the categories file provided in the dataset. These steps are depicted in Figure 3.

As mentioned previously, the SPARSENN method uses SPMSPM in each iteration. This function will do the multiplication of two given matrices A and B in CSR and CSC formats respectively and save in matrix Scores in CSR format. The routine takes each row of A and multiply with the column of B making sure that all 0 rows/columns are not included in computation. Once we get the output of a row (from A) multiplied by a column (from B) we execute the RELU activation function and save the output entry in the output matrix scores. Once done, SPARSENN will move to the next iteration (layer).

Description of naive parallel implementation

To solve the problem in parallel code, we took the following approach. We started by making the main input layer (A) as a CSRMatrix, the weight layer(B) as a CSCMatrix and the output (result) as a COOMatrix. The main kernel spmspm launches a 2D grid of threads where each thread handles one element of the output. For example, the element from the COOMatrix with row = 1 and column = 3 is handled by thread (x=3,y=1). In addition, each thread is responsible for multiplying one full row from input layer (A) with one full column from weight layer (B).
As in the sequential code, the program first starts by loading the dataset then it launches the method sparseNN which contains our naive parallel implementation. Each layer is still calculated in one iteration of our for loop that goes for numLayers times. The result matrix is set in global memory as an empty COOMatrix (outBuffer) and the input layer is set as inBuffer. After that, we call spmspm kernel that takes the inBuffer (CSRMatrix) and W (CSCMatrix) and writes to the outbuffer (COOMatrix).
At the end of each iteration call, the outBuffer is set to be the next layer's inBuffer just like what we had in the sequential code. The full process of our naive parallel implementation is depicted in Figure 4. The code is uploaded and up to date on github `https://github.com/FatimaHojeij/Team7GPU_Project`.

So far, the outbuffer has values that are not sorted which is why we pass it again to some helper kernels. This will ensure that the output buffer (outbuffer) is well sorted according to the COOMatrix convention. These helper kernels are called in the following order:

- histogram: Count how many elements there is in each row
- scan: Get the row pointers of CSR from histogram

- binning: Place columns and values in the right ranges (depending on scan)
- sorting: Sort the columns and values within the ranges (parallel bubble sort)

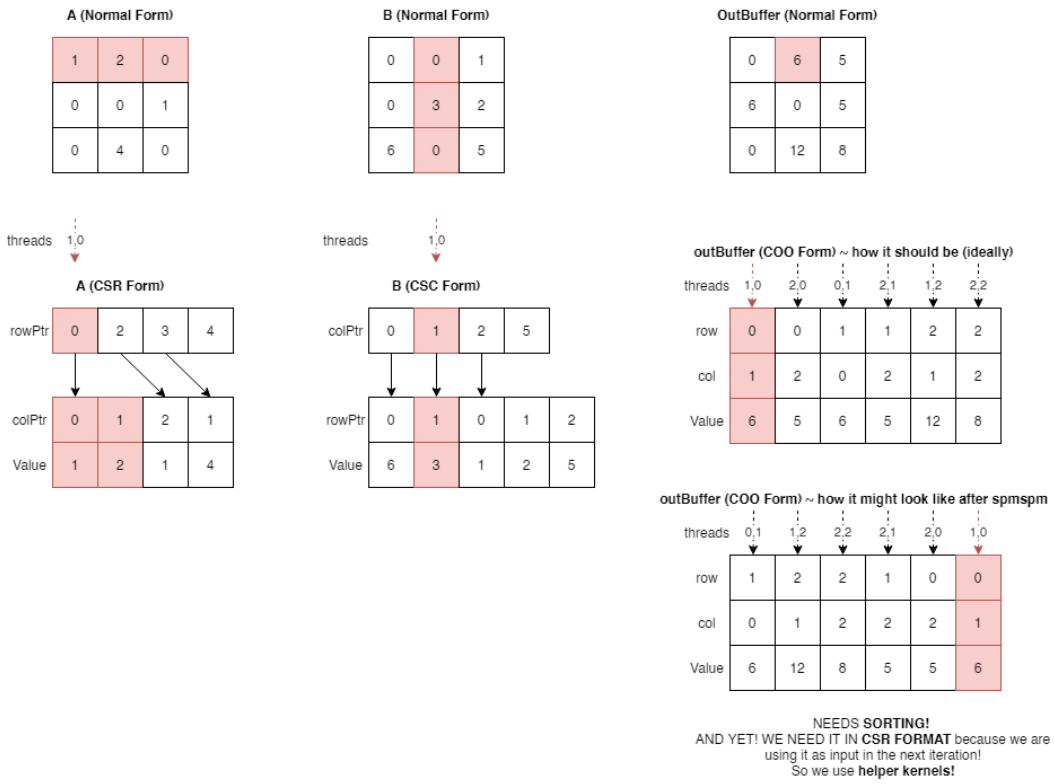

Figure 4: An example of spmspm on thread(1,0)

The histogram, scan, and binning kernels use the resulted COO matrix in order to build it's corresponding row pointers array and fill in the column indices and values within their correct ranges. This will be then passed to the sorting kernel which will sort the column indices and values.

sorting: We launch a thread for every row of the CSR matrix, we make that thread to sort it's row's corresponding values and columns sequentially. The process of the 1st iteration of the parallel code is depicted in Figure 5.

Profiling and Evaluation of Naive and Sequential Code

Name	Total Inference Time	Avg. spmspm kernel time per layer
Sequential	350 s	2.887 s
Naive	668 s	2.443 s

- The naive implementation uses a parallel approach to convert from COO to CSR.
- The timing of the naive implementation using the updated skeleton code Dr. Izzat provided was 774 seconds which is worse than the timing we got from the old implementation.
- The total inference time of the sequential code is less than the total inference time of the naive code. This is reasonable because in the naive implementation we are launching multiple kernels to convert from COO to CSR. However, it is worth to note that the timing of the SPMSPM kernel was better than that of the sequential. Refer to Figure 5 showing the timeline of the naive implementation proving that converting consumes more time than spmspm.
- As shown in Figure 7 our algorithm is memory bound i.e. the kernel performance is bound by instruction and memory latency.
- Occupancy: As shown in Figure 8, the occupancy achieved is only 61.94%. We know that occupancy is usually constrained by 3 factors: number of threads per block, number of register usage per threads, and shared memory size per SM.
  - Number of threads: We are using 32 by 32 threads per block which is equal to 1024 threads which is less than the maximum number of threads per block (2048). Hence we are using 2 blocks per SM less than 32 which is the maximum number of block per SM. So, our occupancy is not limited by the number of threads per block.
  - Number of registers per thread: If we run the command to get the number of registers per thread it will states that the kernel uses 17 registers/thread. $32 * 32$ threads uses $17 * 1024 = 17,408$ registers so in case of full occupancy we will get $17*2084 = 34,816$ registers which is less than the maximum number of registers/SM (64K). Then, it's not limited by the max shared registers/SM.
  - Shared Memory: Not Applicable.
- We concluded that our occupancy is either limited because we are not utilizing the shared memory, or because each thread is using a lot of memory. And we know that if we have too many threads and each is using a lot of memory then our kernel performance will be negatively affected.

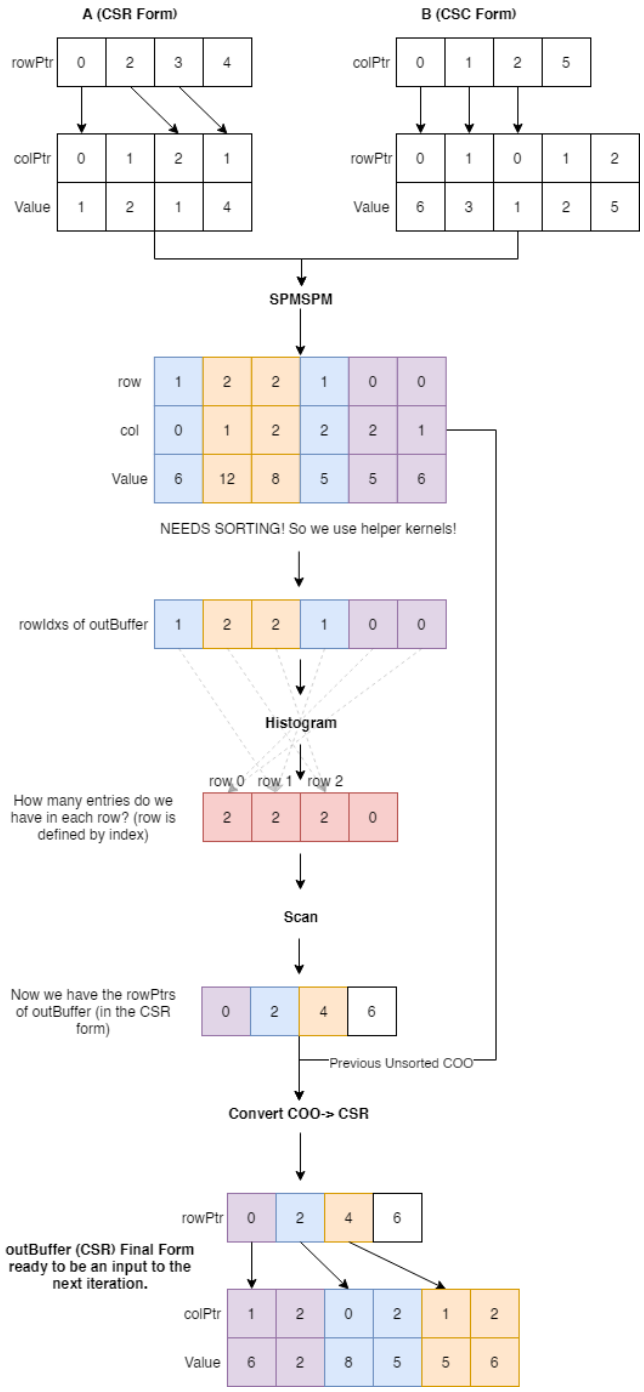

Figure 5: Process of the 1st iteration of the parallel code (From unsorted COO to sorted CSR)

Figure 6: Timeline of different kernels

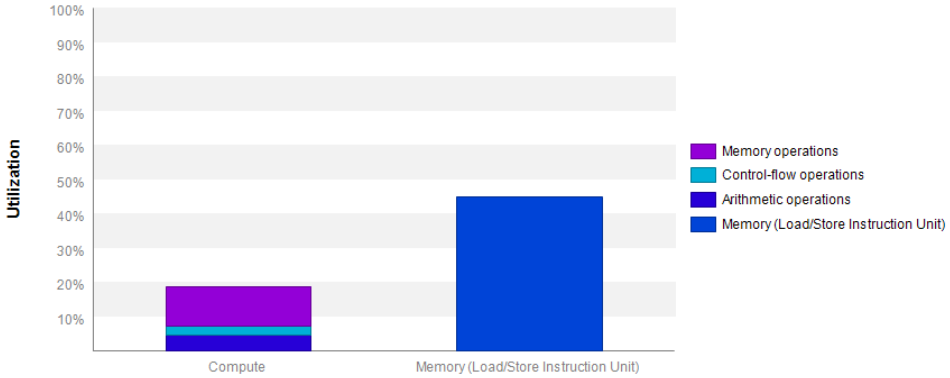

Figure 7: Memory vs Compute Boundedness

Theoretical Occupancy [%]	100
Theoretical Active Warps per SM [warp/cycle]	64
Achieved Occupancy [%]	61.94
Achieved Active Warps Per SM [warp/cycle]	39.64

Figure 8: Occupancy

Optimizations

It was important to try out some optimizations to make some of the more time-consuming aspects of the naive implementation, such as redundant read/write operations on the global memory and all threads performing atomic operations, more efficient. Our optimizations were focused on the main spmspm kernel. The optimizations are listed and detailed below.

- Optimization 1: Intra-wrap synchronization (Warp Vote), applied on the number of non-zero elements of the result.
  Since every thread is responsible for one output element, then in the worst case each one of them is going to increment atomically the number of non zero elements (nnz). For that reason, we limited the number of threads incrementing this global variable to one leader thread per warp. This way we are limiting the number of threads waiting for the resource to be available.

- Optimization 2: Thread coarsening (factor 100).
  In general, the idea behind thread coarsening is to make every thread responsible for multiple output units. In our case, instead of making each thread responsible for only one output element, this thread will be responsible for COARSEFACTOR output elements. Each thread now is multiplying one column with COARSEFACTOR number of rows, thus producing COARSEFACTOR output elements. In this way, we increase the number of cache hits and decrease the number of redundant memory loads (refer to Figure 9). We tried several combinations and we find that coarsening on the number of rows only gave the best result. After tuning the COARSEFACTOR, we found that 100 is the best one.

- Optimization 3: Privatization Each thread block will have a private copy of the results.
  Since threads are constantly accessing global memory to place their results and to increment nnz, we used shared memory and privatization in order to decrease the number of accesses to global memory. Now, each thread block will write to a private copy of the result matrix. This private copy is going to be in shared memory and is of size the maximum number of output elements which is equal to the block dimension. At the end, each block will commit their private copy of the result matrix to global memory (refer to Figure 10).

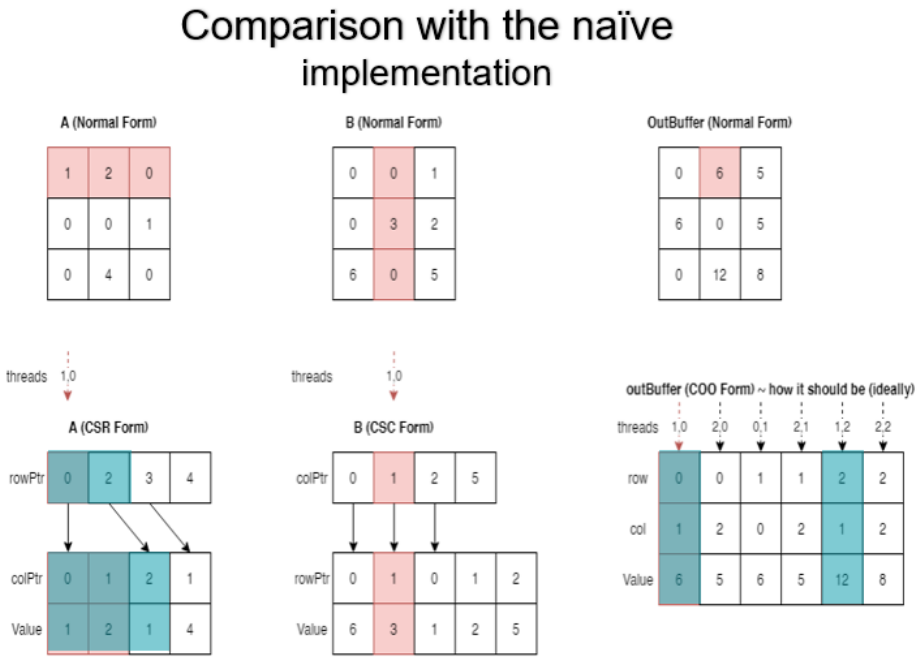

Figure 9: Process of thread executing in the coarsening kernel

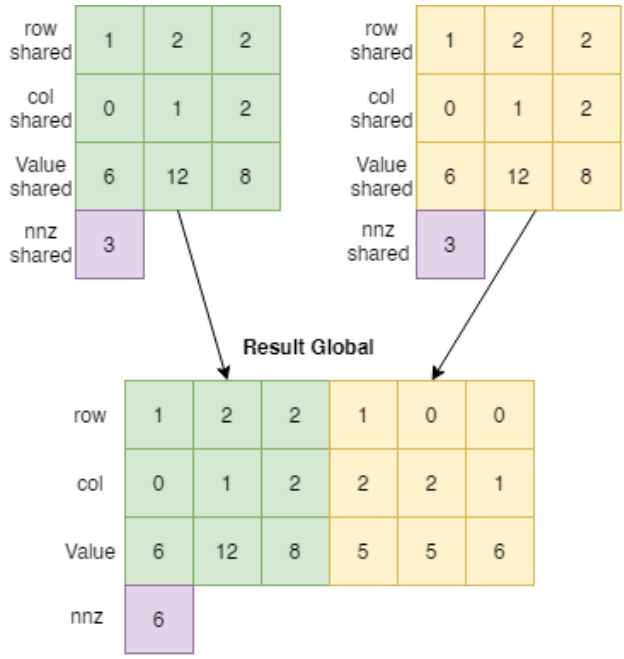

Figure 10: Example of Privatization

Profiling and Evaluation of the Optimizations

Name	Total Inference Time	Avg. spmspm kernel time per layer
Naive	668 s	2.443 s
opt 1 (intra warp)	663 s	2.565 s
opt 2 (coarsening)	450 s	0.746 s
opt 3 (privatization)	517 s	2.025 s
opt 4 (1 and 2)	453 s	0.751 s
opt 5 (1 and 3)	543 s	2.357 s

- Total inference time is not a reflection of our spmspm kernel time as most of the time is being taken up by the sorting helper kernel as seen in the timeline Figure 6. So, we used the average spmspm kernel time per layer as a more accurate performance measure.
- Optimization 1: The average kernel time per layer is worse than the naive one. This can be explained as this is a sparse sparse matrix multiplication and the number of active threads is probably low per warp. So, we are not fully utilizing the advantages of having a leader thread doing the nnz incrementations.

- Optimization 2: This achieved the best kernel time. As explained above, we reasoned that this performance is due to the cache hits.
- Optimization 3: This performed better than the optimization 1 but not better than optimization 2 (coarsening). The better performance can be reasoned from using shared memory and decreasing global memory direct access.
- Optimizations 4 and 5 : Merging Intra Warp with another optimizations made the performance worse.
- The kernels timeline and memory boundedness did not change from the naive implementations to the optimizations so we will not show them again.
- Below are some profiling results for the optimizations.

NAIVE

L1/Shared Memory		
Local Loads	0	0 B/s
Local Stores	0	396.266 kB/s
Shared Loads	0	0 B/s
Shared Stores	0	0 B/s
Global Loads	8425496480	137.063 GB/s
Global Stores	6663354	176.637 MB/s
Atomic	1943685	1.944 MB/s
L1/Shared Total	8434103519	137.242 GB/s
L2 Cache		
L1 Reads	8589091684	137.064 GB/s
L1 Writes	11093764	177.033 MB/s
Texture Reads	0	0 B/s
Noncoherent Reads	0	0 B/s
Atomic	3462304	27.626 MB/s
Total	8600185448	137.241 GB/s
Texture Cache		
Reads	0	0 B/s
Device Memory		
Reads	3769376	60.151 MB/s
Writes	14328740	228.657 MB/s
Total	18098116	288.808 MB/s
ECC Overhead	7919127	126.373 MB/s

COARSENING

L1/Shared Memory		
Local Loads	0	0 B/s
Local Stores	0	1.391 MB/s
Shared Loads	0	0 B/s
Shared Stores	0	0 B/s
Global Loads	8425496480	142.698 GB/s
Global Stores	6663486	183.949 MB/s
Atomic	2004548	2.005 MB/s
L1/Shared Total	8434164514	142.886 GB/s
L2 Cache		
L1 Reads	8589206888	142.701 GB/s
L1 Writes	11155880	185.344 MB/s
Texture Reads	0	0 B/s
Noncoherent Reads	0	0 B/s
Atomic	3462304	28.761 MB/s
Total	8600362768	142.886 GB/s
Texture Cache		
Reads	0	0 B/s
Device Memory		
Reads	20337618	337.889 MB/s
Writes	14357111	238.529 MB/s
Total	34694729	576.418 MB/s
ECC Overhead	15988233	265.628 MB/s

Figure 11: Memory Utilization of Naive and Coarsening Optimization

The only noticeable difference between the two implementations is in the number of local stores this is perhaps due to the L1 Cache hits.

NAIVE

L1/Shared Memory		
Local Loads	0	0 B/s
Local Stores	0	396.266 kB/s
Shared Loads	0	0 B/s
Shared Stores	0	0 B/s
Global Loads	8425496480	137.063 GB/s
Global Stores	6663354	176.637 MB/s
Atomic	1943685	1.944 MB/s
L1/Shared Total	8434103519	137.242 GB/s
L2 Cache		
L1 Reads	8589091684	137.064 GB/s
L1 Writes	11093764	177.033 MB/s
Texture Reads	0	0 B/s
Noncoherent Reads	0	0 B/s
Atomic	3462304	27.626 MB/s
Total	8600185448	137.241 GB/s
Texture Cache		
Reads	0	0 B/s
Device Memory		
Reads	3769376	60.151 MB/s
Writes	14328740	228.657 MB/s
Total	18098116	288.808 MB/s
ECC Overhead	7919127	126.373 MB/s

Privatization

L1/Shared Memory		
Local Loads	0	0 B/s
Local Stores	0	1.324 MB/s
Shared Loads	31276690	3.956 GB/s
Shared Stores	8895571	1.125 GB/s
Global Loads	8426063024	135.807 GB/s
Global Stores	7205511	182.525 MB/s
Atomic	1930651	1.931 MB/s
L1/Shared Total	8475371447	141.074 GB/s
L2 Cache		
L1 Reads	8589777044	135.81 GB/s
L1 Writes	11633747	183.937 MB/s
Texture Reads	0	0 B/s
Noncoherent Reads	0	0 B/s
Atomic	3840000	30.356 MB/s
Total	8601410791	135.994 GB/s
Texture Cache		
Reads	0	0 B/s
Device Memory		
Reads	3840183	60.716 MB/s
Writes	10962986	173.331 MB/s
Total	14803169	234.047 MB/s
ECC Overhead	5725064	90.517 MB/s

Figure 12: Memory Utilization of Naive and Privatization Optimization

The difference here is that in the implementation with privatization we are using shared memory.

Intra-Warp

Theoretical Occupancy [%]	50
Theoretical Active Warps per SM [warp/cycle]	32
Achieved Occupancy [%]	45.07
Achieved Active Warps Per SM [warp/cycle]	28.85

Coarsening

Theoretical Occupancy [%]	50
Theoretical Active Warps per SM [warp/cycle]	32
Achieved Occupancy [%]	45.07
Achieved Active Warps Per SM [warp/cycle]	28.84

Privatization

Theoretical Occupancy [%]	50
Theoretical Active Warps per SM [warp/cycle]	32
Achieved Occupancy [%]	45.07
Achieved Active Warps Per SM [warp/cycle]	28.84

Figure 13: Optimizations Occupancy