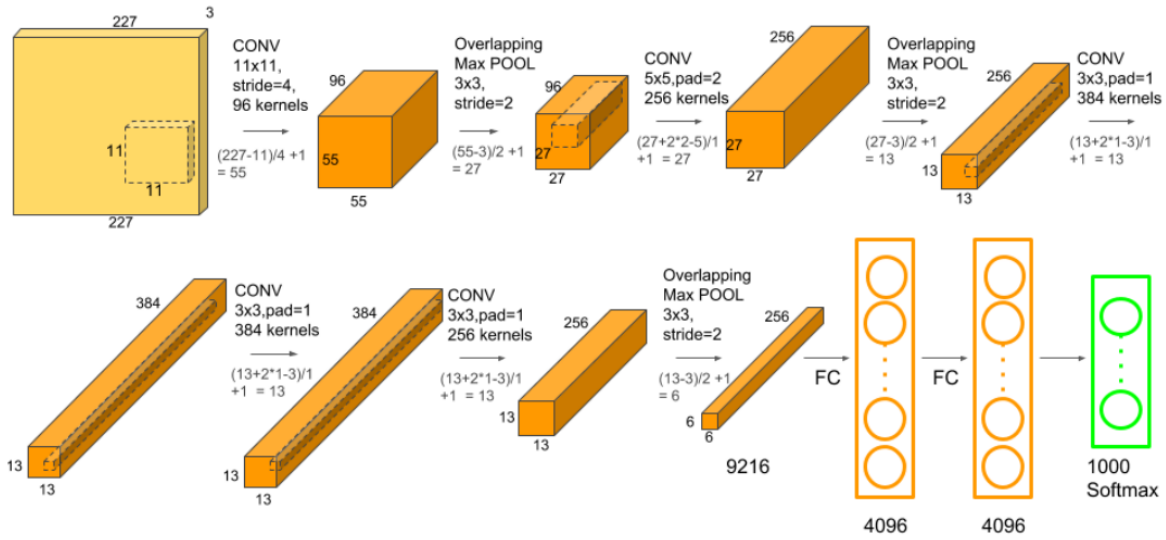
Shreya Gaur

ECE60827 CUDA ASSIGNMENT 3

Introduction

AlexNet is a popular convolutional neural network architecture designed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. AlexNet won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012 and significantly outperformed previous state-of-the-art methods. The architecture consists of 5 convolutional layers, 3 fully connected layers, and an output layer. In this report, I will compare the implementation of the convolutional layers and the matrix multiplication in the fully connected layers of AlexNet on a CPU and a GPU. The GPU implementation of Convolutional Layer achieved an average speed up of 99.458% compared to the CPU implementation. For the Matrix Multiplication implementation, the optimized GPU implementation gains an average speed up of 99.86% compared to CPU implementation.

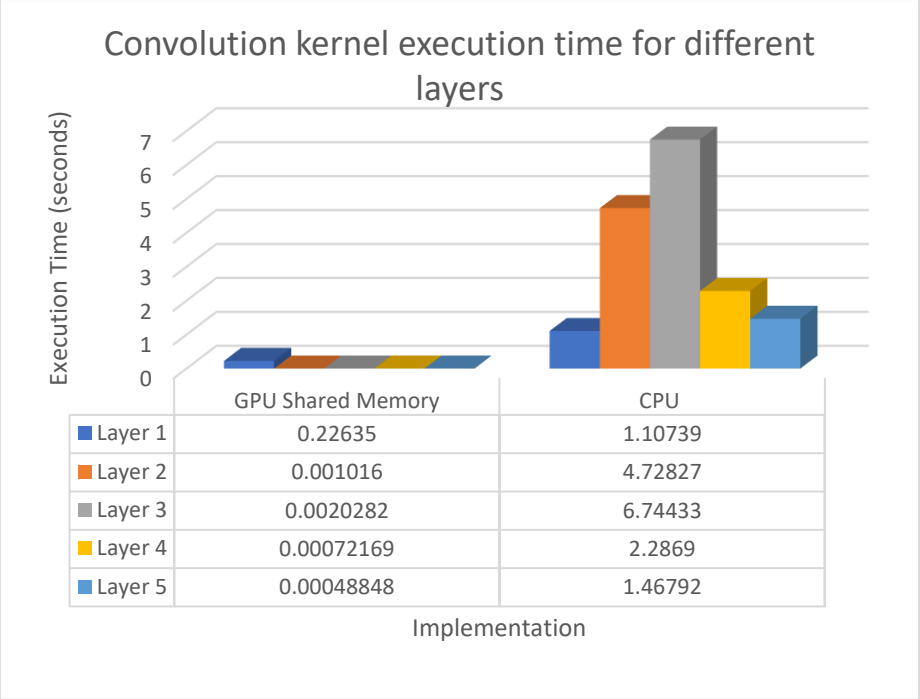


Convolutional Layer

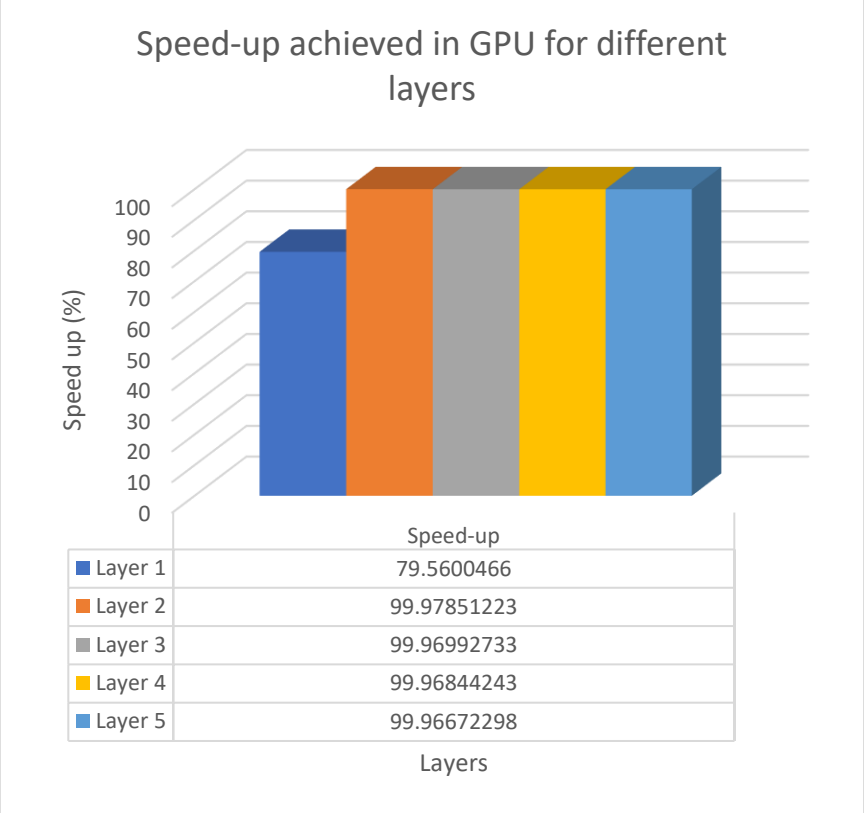
Comparison between CPU and GPU implementations

I implemented the convolutional layers of AlexNet on both CPU and GPU. The GPU used for the implementation is Nvidia Volta V100. An important feature of Volta V100 GPU includes configurable 96KB shared memory. I have used this shared memory and constant memory features of CUDA to optimize the GPU implementations of both Matrix Multiplication and Convolution. With the use of shared memory, the implementation of Convolutional Layer attains an average occupancy of 96% per SM.

The naïve CPU implementation is taken in which the computation of each element of the output is carried out sequentially for all the different channels of the filters. This takes on average 3.266 seconds to complete for each layer. Compared to this, the optimized GPU implementation on average takes only 1ms to complete with 0 errors. The graph below depicts the execution time taken by GPU and CPU for different layers in seconds.



The speedups achieved by the GPU implementation of different layers on AlexNet can be seen in the graph below.



Basic GPU Implementation

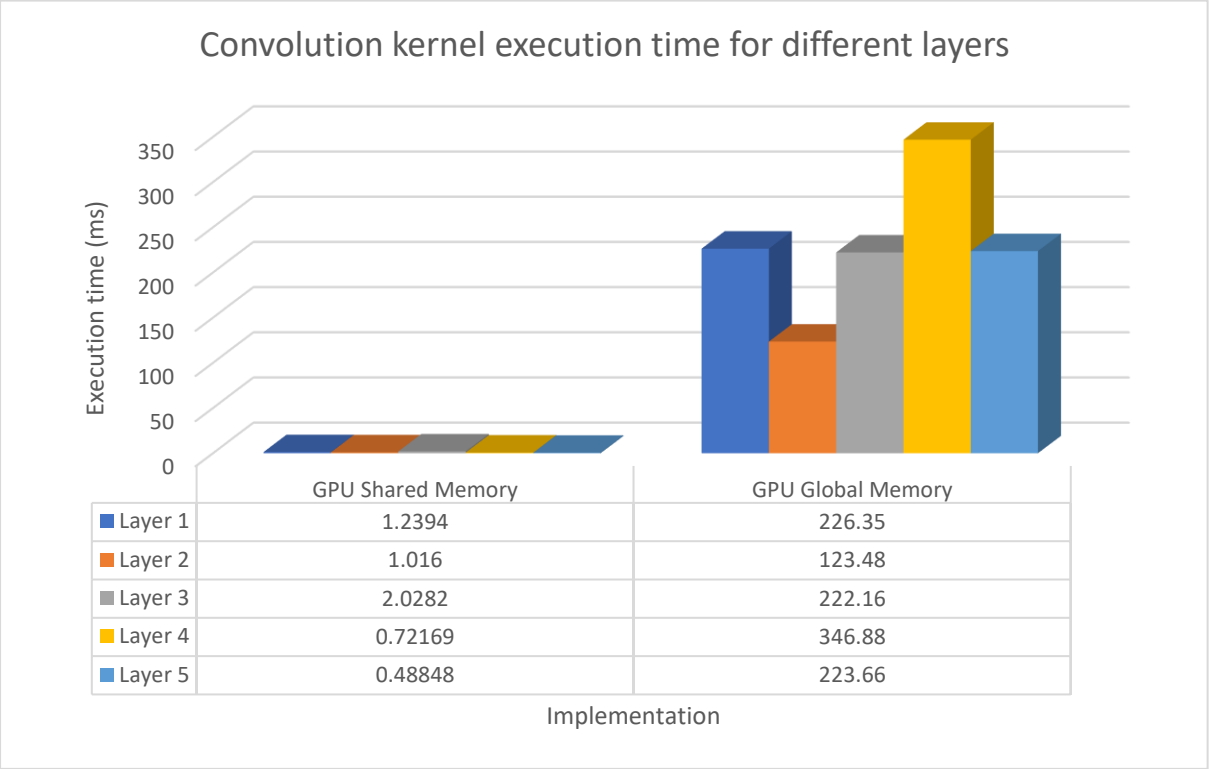
My initial implementation of the Convolutional layers on the GPU utilized the global memory to store the inputs and filters for the convolution operation. The implementation was not parallelized across channels and the number of threads was decided based upon the number of output elements to be computed. The basic idea was to assign the task of computation of one output pixel to a single thread for all the channels of input. This was a very naïve approach was taken basically taken from the CPU implementation of the convolutional layers.

For the optimization of this naïve approach, I decided to exploit the input and feature reuse with the 3D convolution operation. The input is reused for each batch of the filter and the filter is reused by each batch of the input. Here since the computation only takes place for a single batch, it was more effective to put the input in the shared memory as it will be used by all the filters. For further update, one can even put each layer of filter in the shared memory.

I also exploited the fact that the bias is necessarily constant and does not change throughout the runtime. Therefore, it is a perfect candidate to be put in the constant memory. The filter cannot be put in the constant memory due to restricted space in the constant memory.

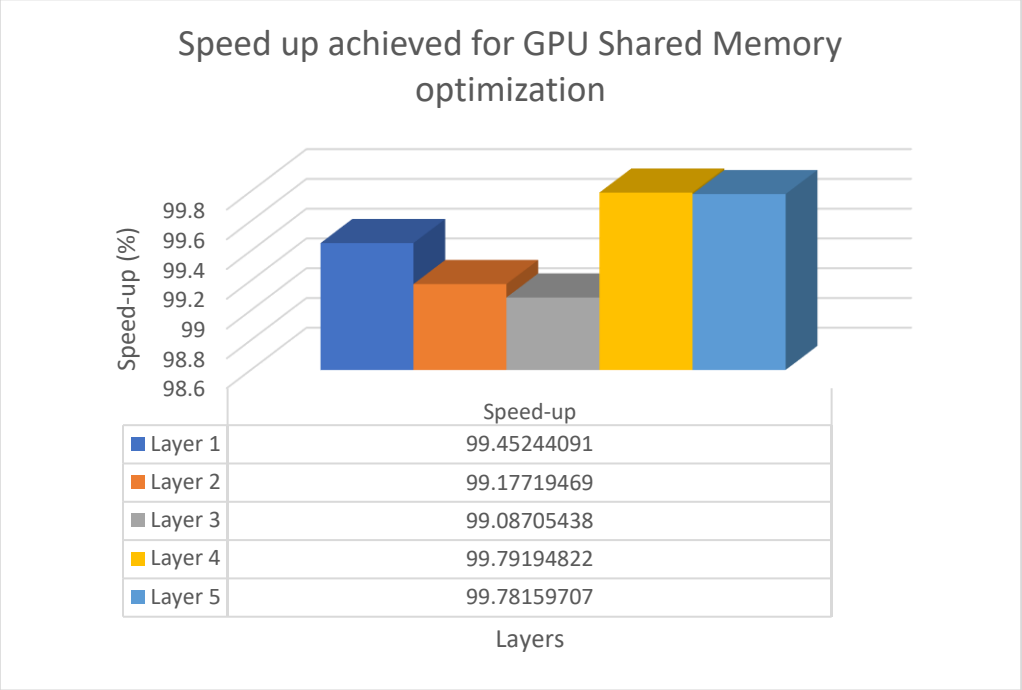
Optimized GPU implementation with Shared Memory

The shared memory implementation also parallelises across the channels of the input. Convolution between each channel of the filter and the corresponding filter is stored in the thread registers and eventually an atomicAdd() operation is used in order to compute the sum of each of the thread's output for an input channel. The atomicAdd() operation add additional overhead but the gain in speedup is considerable as compared to the naïve global memory-based implementation in GPU as can be seen from the graph below.

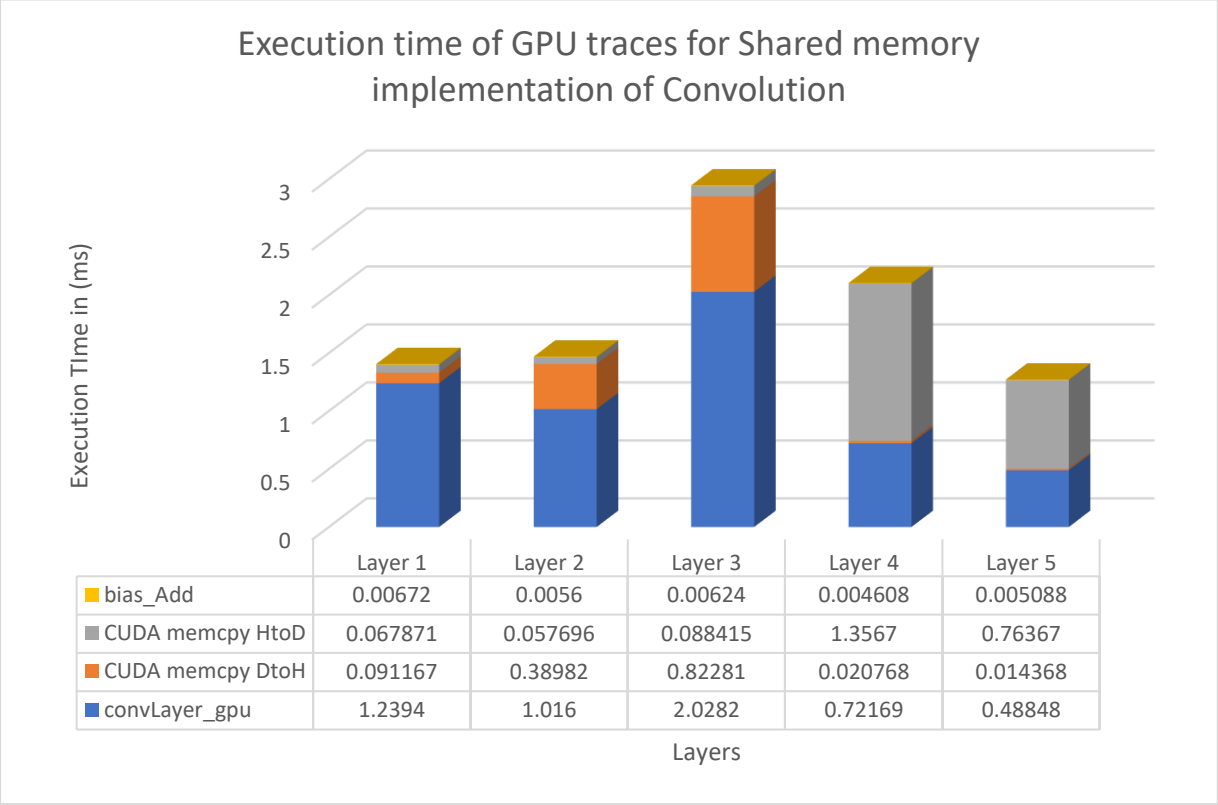


	GPU Shared Memory	GPU Global Memory
Layer 1	1.2394	226.35
Layer 2	1.016	123.48
Layer 3	2.0282	222.16
Layer 4	0.72169	346.88
Layer 5	0.48848	223.66

The Optimized GPU implementation gains average speed up of 99% compared to the GPU global memory-based implementation. The speed up gained in case of each of the layers can be seem in the graph below.



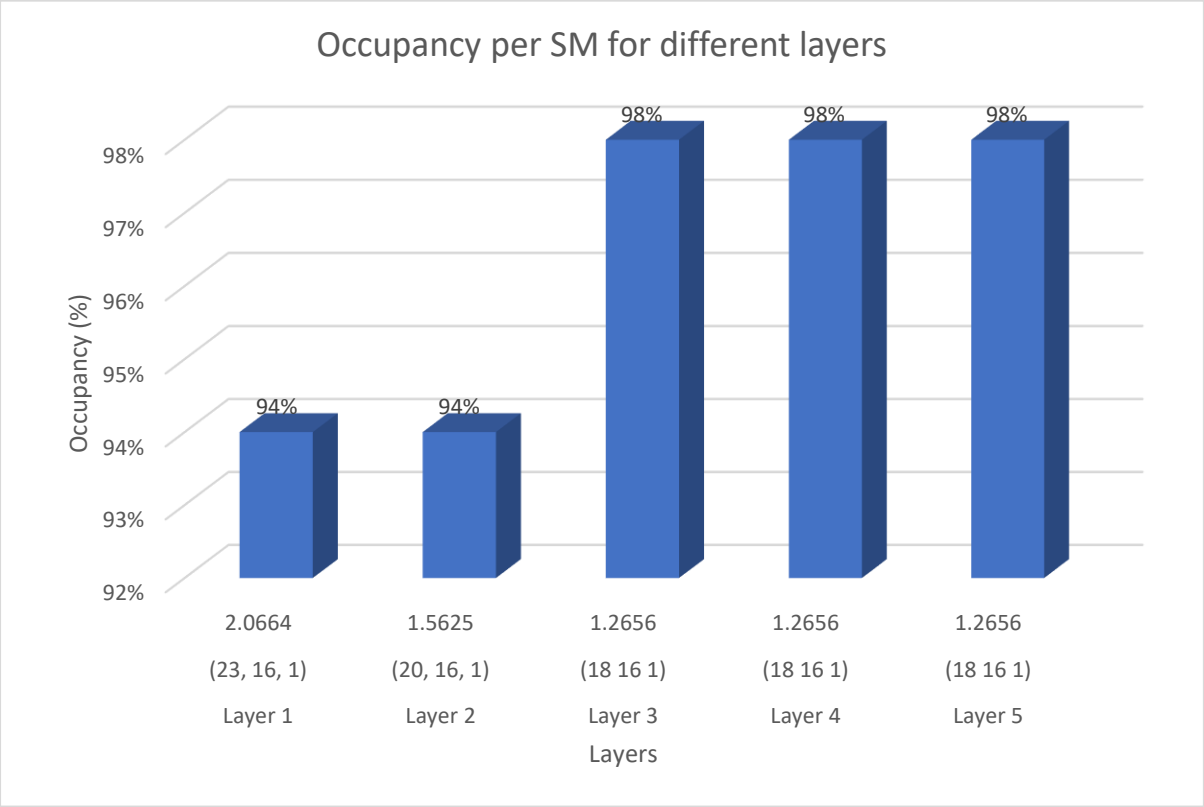
The GPU traces for different kernels and cudaMemcpy() in case of optimized GPU shared memory-based implementation is seen in the graph below.



The bias_Add kernel is used to add the bias value to each of the output elements after the convolution results are outputted. The reason to employ a separate bias_Add kernel are specific to the loading and

accessing of input data in the shared memory. The input data is loaded in form of strips of tiles of size **tileW = BLOCK_SIZE + fShape.width – args.strideW** and **tileH = BLOCK_SIZE + fShape.width – args.strideH**. The dimensions of the block, therefore, are **(tileW, threadsPerSubblock)**, where threadsperSubblock decides the number of rows which are to be loaded in one strip of the tile. The thread blocks load strips of their tiles of size **tileH*tileW** one by one and put that into their shared memory. This process for a tile is sequential. To access the elements of shared input one must do this sequential retrieving which makes addition of the bias complex. With the experiments I conducted, the bias was getting added multiple times to a single element of output. To avoid this confusion altogether, I assigned the task of adding bias to a different kernel.

The occupancy of the optimized GPU implementation calculated by CUDA-Occupancy-Calculator is shown in the graph below for different layers with the initial respective Block Size and shared memory usage. The current implementation has a very good occupancy of 96% on average for all the layers.

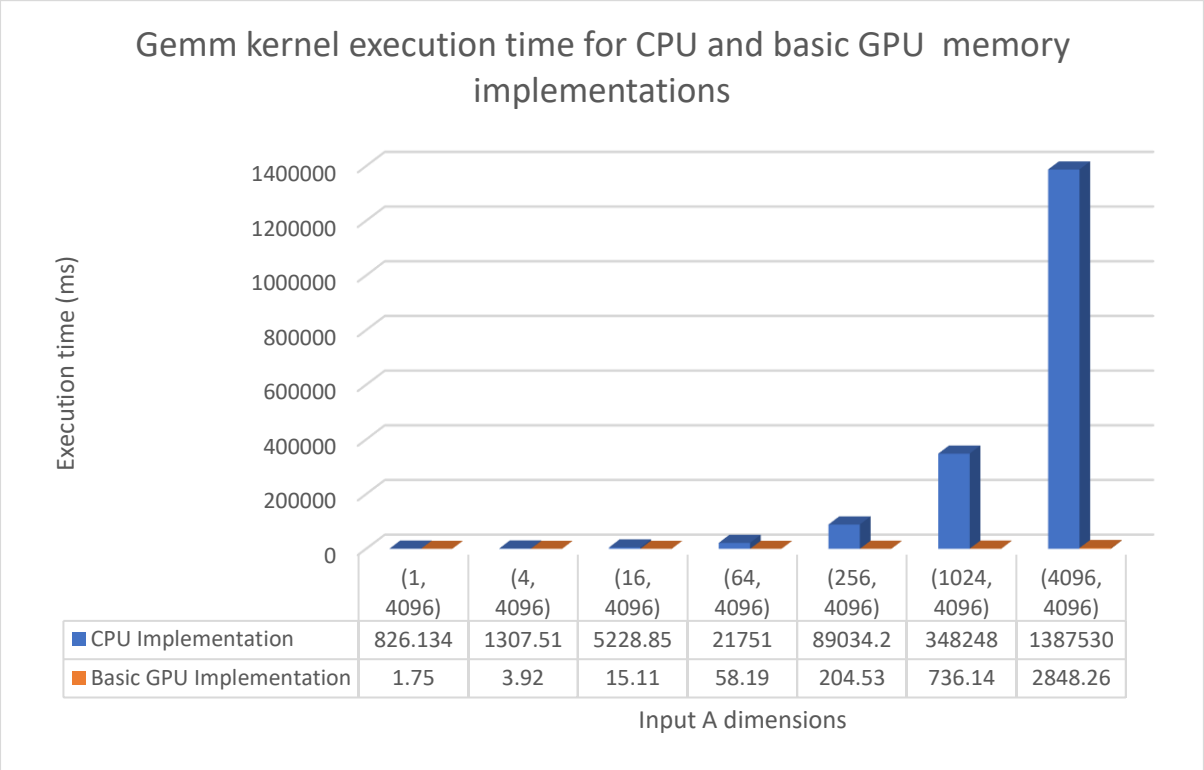


Matrix Multiplication

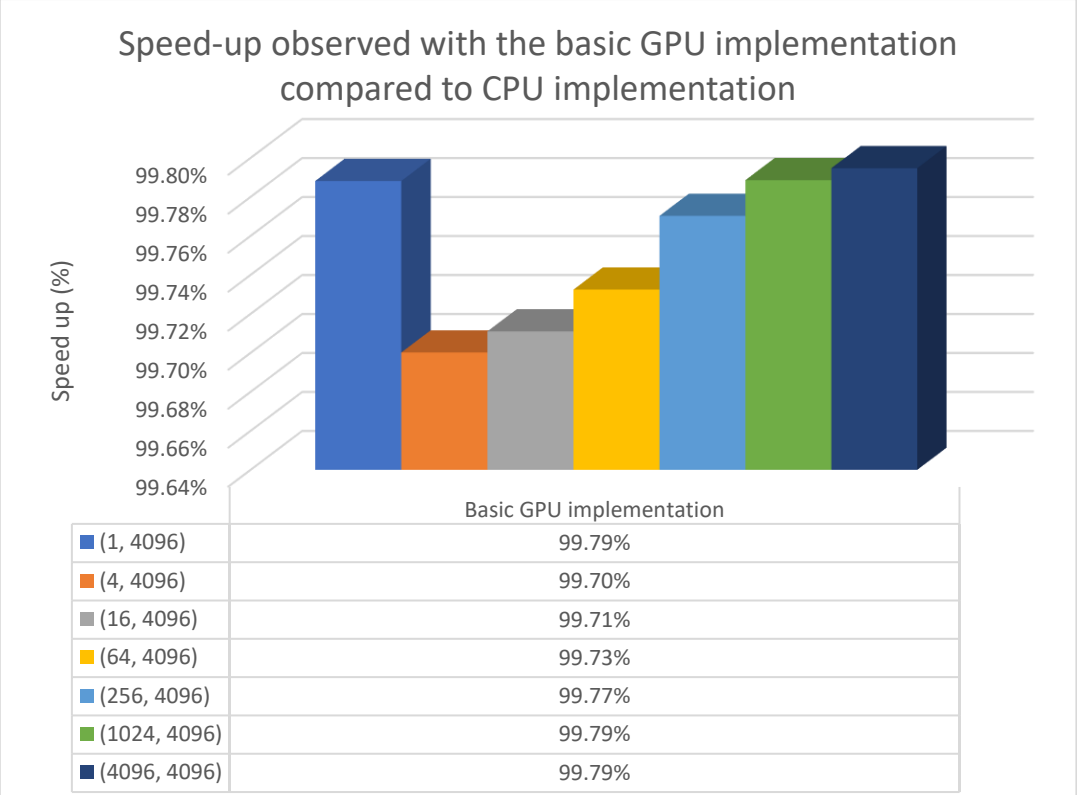
Matrix multiplication is a fundamental operation in deep learning, and it is used extensively in convolutional neural networks (CNNs). AlexNet uses matrix multiplication in its fully connected layers. In this report, we will compare the implementation of matrix multiplication for the fully connected layers of AlexNet on a CPU and a GPU. We implemented three different versions of the GPU implementation: an unoptimized global memory-based implementation, an unoptimized unified virtual memory-based implementation, and a shared memory-based optimized implementation.

Comparison between CPU and GPU implementations:

The CPU implementation of Matrix Multiplication does use the concept of tiling, i.e, a tile of the output element is computed by adding the partial sums of matrix multiplication of tiles of the inputs A and B. This however done one tile of the output at a time hence there is a great scope of parallelization. Each of the individual tiles of the output can be computed by a block of size **(args.tileW, args.tileH)**. This is implemented in a basic GPU implementation of the algorithm. The graph comparing the time taken by GEMM kernel for both the approaches can be seen below:



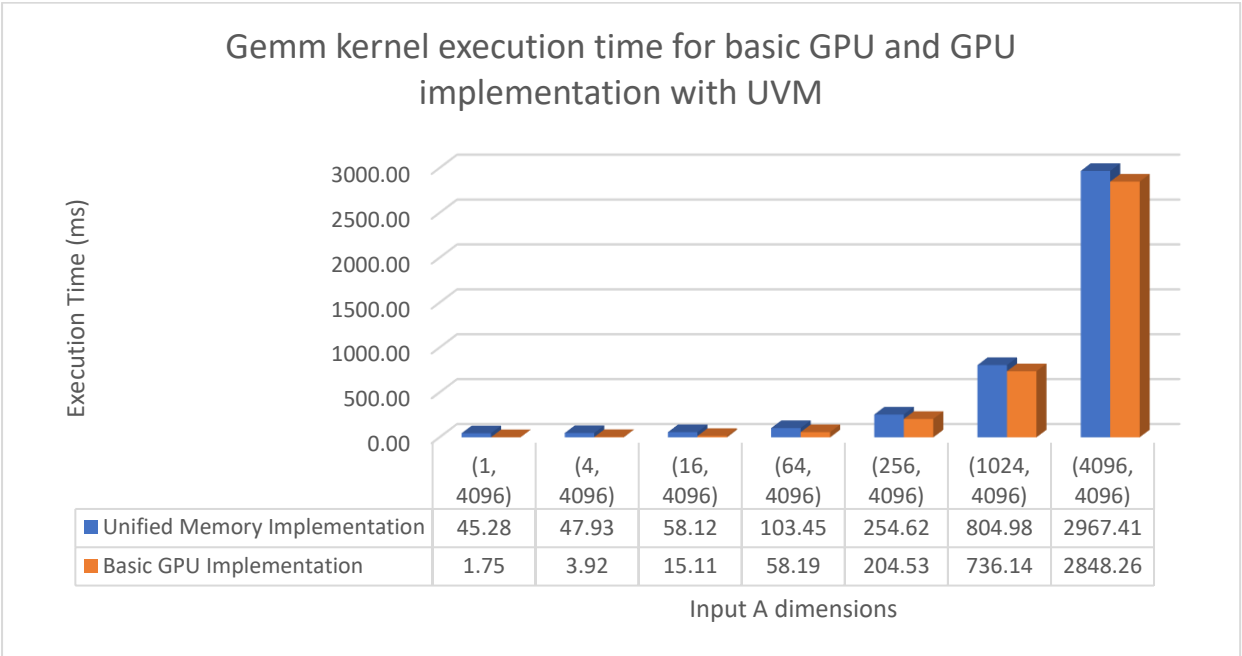
It can be observed that even the unoptimized implementation on the GPU significantly outperforms the CPU. The speed ups observed can be seen the graph below.



	Basic GPU implementation
(1, 4096)	99.79%
(4, 4096)	99.70%
(16, 4096)	99.71%
(64, 4096)	99.73%
(256, 4096)	99.77%
(1024, 4096)	99.79%
(4096, 4096)	99.79%

Basic GPU implementation with Unified Virtual Memory

Unified Memory is a single memory address space accessible from any processor in a system. The unified memory alleviates the usage of cudaMemcpy() completely the GPU can get the data directly from the CPU through page faults. For the next part of my implementation, I used Unified Virtual Memory in order to transfer data from the host to the device. The graph depicting the performance of GEMM kernel of the UVM based GPU implementation and the basic GPU implementation with the usual cudaMemcpy() is given below.

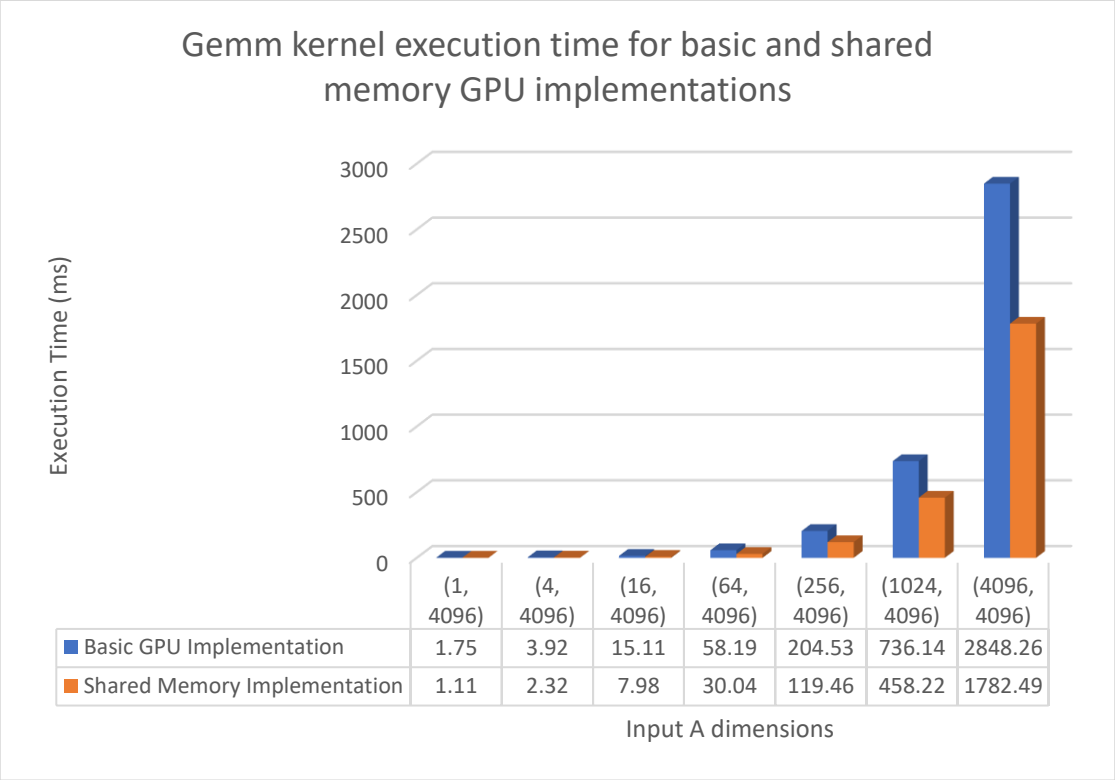


	(1, 4096)	(4, 4096)	(16, 4096)	(64, 4096)	(256, 4096)	(1024, 4096)	(4096, 4096)
Unified Memory Implementation	45.28	47.93	58.12	103.45	254.62	804.98	2967.41
Basic GPU Implementation	1.75	3.92	15.11	58.19	204.53	736.14	2848.26

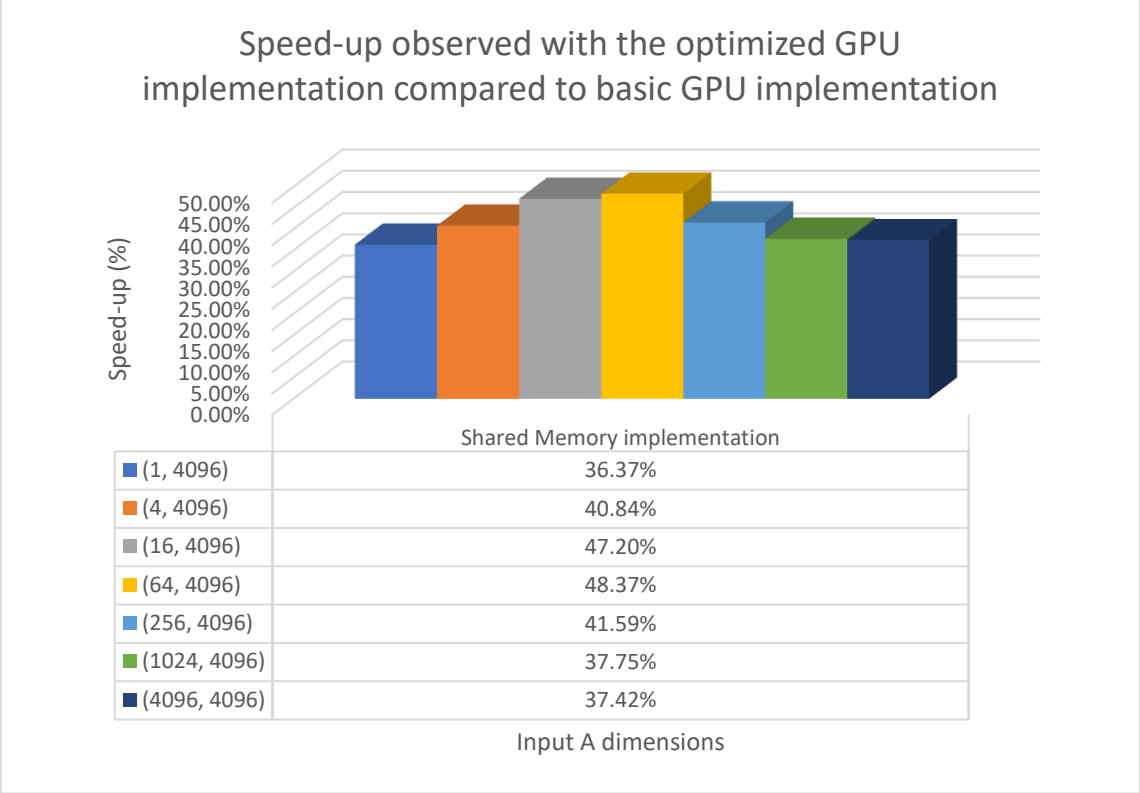
It can be observed that the use of UVM adds on additional overheads in the migration of the data to the device from the host. This can be observed in the graph above. In case of (1, 4096) Input A dimension, the additional overhead increases the execution time a lot. The migration overhead in this simple code is caused by the fact that the CPU initializes the data, and the GPU only uses it once. The overhead can reduce if the data is initialized on the GPU itself.

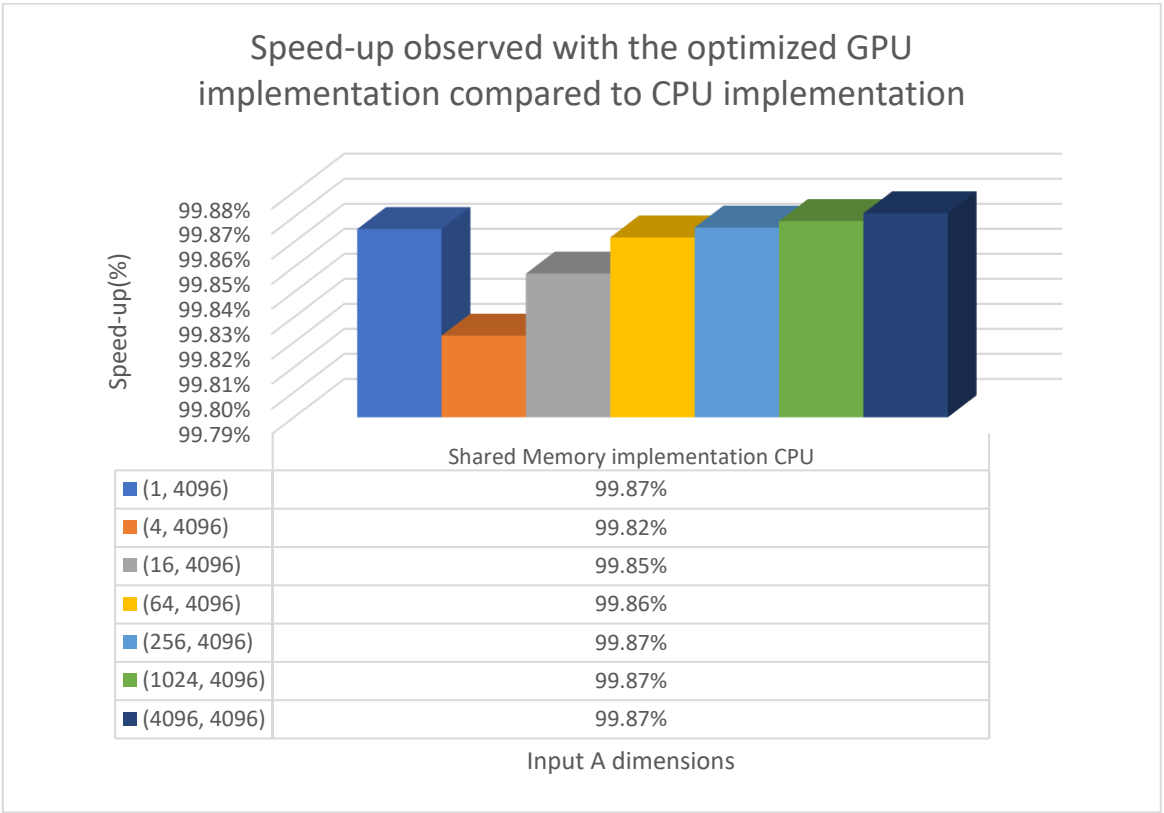
Optimized GPU implementation with Shared Memory

In the Optimized GPU implementation of matrix multiplication, the tiles of input A and input B are stored in the shared memory. The threads of each thread block co-ordinate to store the elements of their individual tiles in the shared memory. This approach offers significant performance improvement compared to the basic implementation where the inputs are stored in the global memory. The graph below shows the performance of the GEMM kernel for the global memory based and shared memory-based implementations.

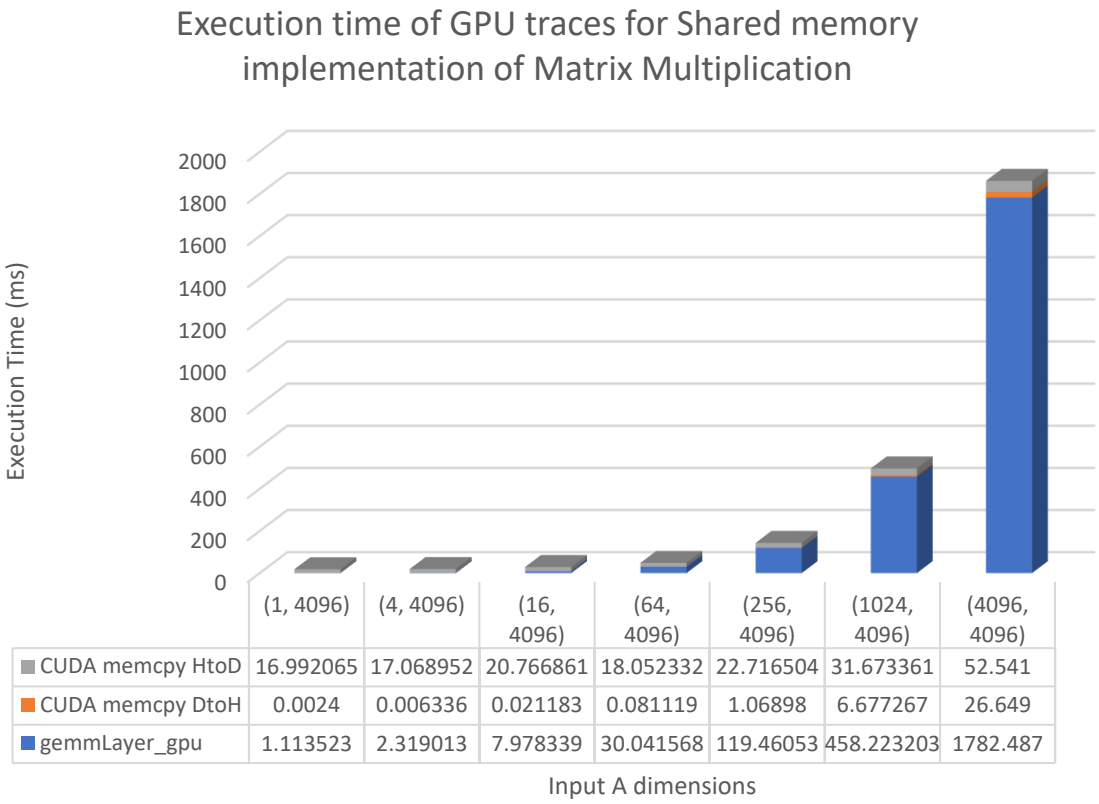


As the batch size scales, the shared memory approach shows considerable improvements over the basic global memory-based approach. The speedup gained by implementing the matrix multiplication in the shared memory can be seen in the graph below.





The stacked bar graph depicting the performance of the optimized GPU implementation is shown below.



Conclusion

The convolutional layers and matrix multiplication layers are implemented in GPU with optimization include use of shared/constant memory. The implementations gain considerable speed-up compared to the CPU implementations.