CUD(A)lexnet

ECE695 Programming Assignment | Raghav Venkatasubramanian

My best performing GPU Implementation takes **0.196** seconds to complete all 5 layers for a batch size of 128.

1 Introduction

In this assignment, the convolutional layers of the famed AlexNet are implemented on a GPU by exploiting the innate parallel nature of the computations. The objective is to create a GPU kernel for Convolution (Cross-Correlation) operation, and then perform successive optimizations and evaluate the performance obtained against a naïve CPU implementation.

2 Methodology

2.1 CPU

We start off with a basic seven loop CPU Implementation for the Convolution layers. The code contains the implementation for one layer which can be configured based on the parameter passed to the function. All the inputs are initialized randomly and after execution the data can be dumped to a file for validation.

2.2 GPU Naïve Implementation

The naïve implementation parallelizes the algorithm by launching thread blocks that are organized into 3D grids as follows:
Grid: X= # of tiles along Width, Y= # of tiles along Height, Z= # of tiles along Channels of convolution output respectively.
Block: X= TileWidth, Y= TileHeight, Z= TileDepth
Each thread then calculates the result of one output element for each batch. Although intuitive and easy to implement, this approach generates a large amount of memory traffic since multiple threads fetch redundant data. This redundancy comes from the fact that the sliding kernel often has a lot of overlapping data between successive strides.

3 Optimizations

3.1 Shared Memory

In this approach, we try to exploit the redundancy between the input data being fetched to find the result for successive output elements. As shown in Fig1 the kernel being swept across the input has a lot of overlap for different elements of the output matrix. The output matrix is divided into square tiles. For the elements within a tile, there is a lot of spatial locality in the input. This input data can be fetched into the shared memory and then the convolution can be done at a faster rate as the compute to memory access ratio is increased

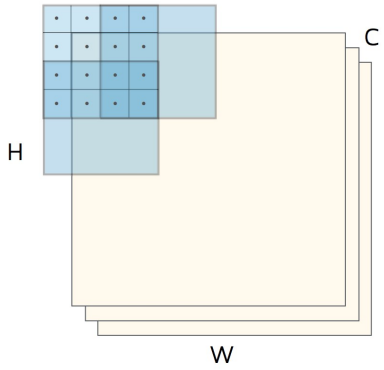

Figure 1: Overlap in kernel Computations

Grid: X= (# of tiles along Output Width) * (# of tiles along Output Height), Y= total output channels, Z= batch size.
Block: X= TileWidth, Y= TileHeight, Z= 1
The shared memory stores the data required by an output tile for each input channel. We iterate over the input maps to compute the result for one output tile. Hence, we parallelize the computations along the output channels as well as the batch size.

3.2 Toeplitz Transform and GEMM

In this optimization, the input data is converted into a Toeplitz representation that converts the convolution problem into effectively a matrix multiplication computation. The kernel can be directly multiplied by the Toeplitz representation to obtain the output. Matrix multiplication is extremely fast on GPUs as it is a highly parallel operation which can be implemented by tilling and shared memory.
For the Toeplitz kernel, for each batch we use the same shared memory tiling in the previous section and transform the input maps into a matrix of dimension.

$$[\text{filterwidth} * \text{filterheight} * \text{input channels}] \times [\text{output Height} * \text{output Width}]$$

This matrix is multiplied on the left by the filter matrix to obtain the final output map. We use the GEMM kernel with shared memory tiling to compute the matrix multiplication.

4 Results

The kernels are run for batch sizes of 1, 32, 128 for each layer. In this section, results for batch size equal to 128 are discussed. The results are listed below.

Unified Virtual Memory (Time in ms)							
Layers	CPU	Naïve GPU		Shared Memory Tiling		Toeplitz + GEMM	
		Value	Speedup	Value	Speedup	Value	Speedup
Layer1	109410.89	54.27	2016.12	54.98	1989.90	41.47	2638.28
Layer2	468241.66	168.87	2772.86	136.14	3439.44	65.59	7139.21
Layer3	154987.68	110.84	1398.31	63.32	2447.61	47.08	3291.90
Layer4	235062.75	98.07	2396.94	102.62	2290.66	38.81	6056.82
Layer5	159502.45	79.79	1999.10	67.66	2357.48	30.55	5221.45

Figure 2: Results for Unified Virtual Memory

CudaMalloc+ CudaMemCopy (Time in ms)							
Layers	CPU	Naïve GPU		Shared Memory Tiling		Toeplitz + GEMM	
		Value	Speedup	Value	Speedup	Value	Speedup
Layer1	109410.89	51.65	2118.23	48.25	2267.40	35.18	3109.66
Layer2	468241.66	123.41	3794.32	146.57	3194.60	72.94	6419.71
Layer3	154987.68	117.26	1321.79	89.30	1735.64	23.99	6459.51
Layer4	235062.75	102.82	2286.11	136.66	1720.03	36.06	6518.76
Layer5	159502.45	78.48	2032.41	94.24	1692.50	28.42	5611.45

Figure 3: Results for CudaMemCopy

5 Analysis

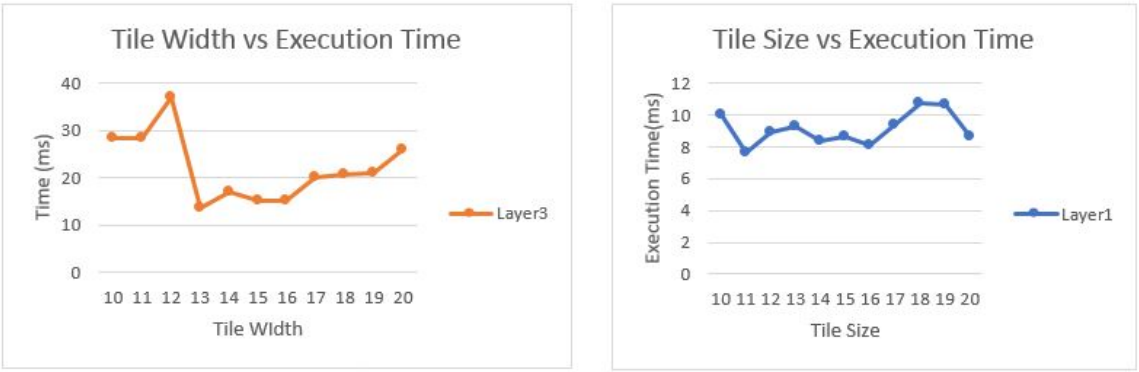

Figure 4: Effect of Tile Size on Execution Time of Layers

5.1 Execution Time vs Tile Width

It is observed that the performance is optimized when the width of the output matrix is a multiple of the Tile Width. This is because since thread blocks are launched with a granularity of the tile width, if the width of the output is not a multiple then additional threads would be launched which would cause divergence and suboptimal usage of resources. This can be seen in the case of layer 3 where, a Tile Width of 13 performs the best.

5.2 Size of thread blocks in SM

We also observe that lower values of Tile Width perform much worse than bigger values. This is can be attributed to higher number of thread blocks required to calculate the result and since there is a limit on the number of thread blocks launched on the SM, the resources of the SM are underutilized.

6 Appendix A: Speedups

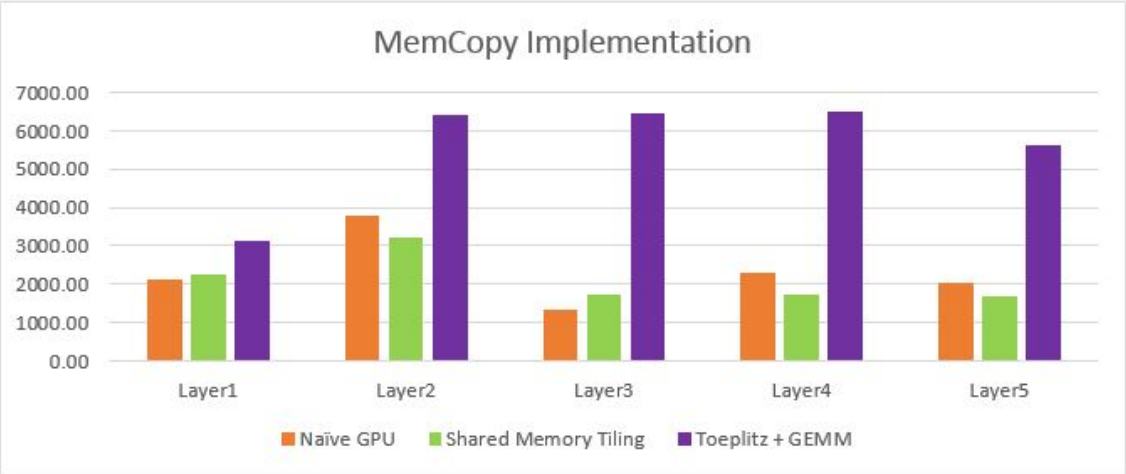

Figure 5: Speedup for Memcpy

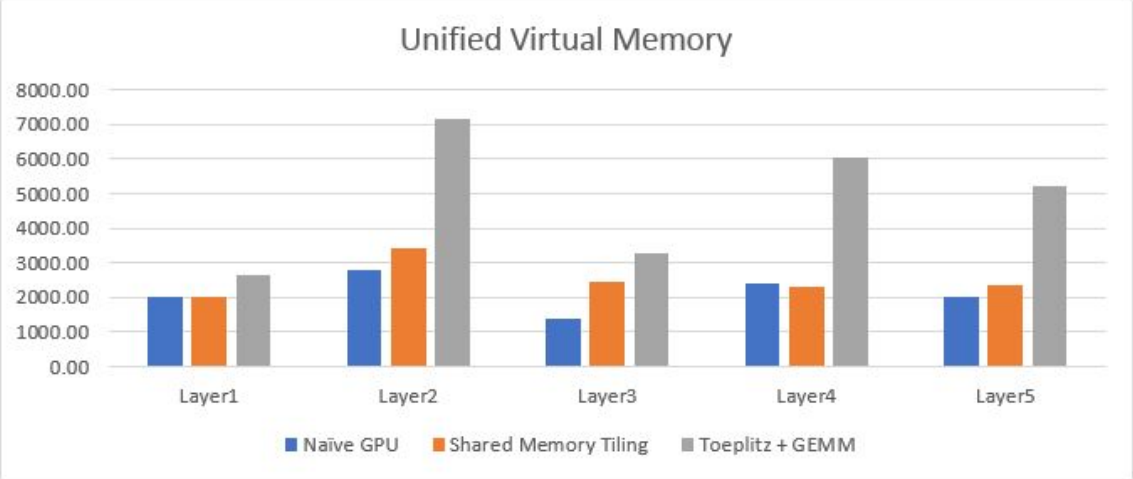

Figure 6: Speedup for UVM

7 Appendix B

Note: All data in milliseconds(ms)

7.1 Unified Virtual Memory

Naïve GPU			ShMem		Toeplitz		
Layers	Kernel	Memcpy	Kernel	Memcpy	Matrix Multiply	Toeplitz Conversion	MemCpy
Layer1	0.25	0.21	0.27	0.20	0.10	0.02	0.20
Layer2	1.18	0.77	1.21	0.86	0.44	0.02	0.81
Layer3	0.76	1.10	0.66	1.10	0.19	0.01	1.15
Layer4	0.73	1.62	0.98	3.01	0.26	0.01	1.66
Layer5	0.46	1.14	0.96	1.13	0.24	0.01	1.08

Figure 7: Batch Size : 1

Naïve GPU			ShMem		Toeplitz		
Layers	Kernel	Memcpy	Kernel	Memcpy	Matrix Multiply	Toeplitz Conversion	MemCpy
Layer1	8.39	6.25	7.74	6.11	3.13	1.25	6.45
Layer2	38.38	3.76	29.18	3.72	11.97	0.78	3.67
Layer3	25.35	2.66	13.80	2.40	4.56	0.10	2.40
Layer4	21.22	3.45	20.70	3.63	16.43	0.41	9.29
Layer5	16.60	3.47	13.82	3.48	4.58	0.15	4.57

Figure 8: Batch Size : 32

Naïve GPU			ShMem		Toeplitz		
Layers	Kernel	Memcpy	Kernel	Memcpy	Matrix Multiply	Toeplitz Conversion	MemCpy
Layer1	29.57	24.70	30.50	24.48	11.32	5.29	24.87
Layer2	153.51	15.36	116.26	19.88	47.76	3.13	14.70
Layer3	101.45	9.39	52.96	10.37	37.93	0.41	8.75
Layer4	84.25	13.82	80.13	22.49	24.59	0.61	13.61
Layer5	65.92	13.87	53.61	14.05	16.48	0.61	13.46

Figure 9: Batch Size : 128

7.2 CudaMalloc and CudaMemCopy

Naïve GPU			ShMem		Toeplitz		
Layers	Kernel	Memcpy	Kernel	Memcpy	Matrix Multiply	Toeplitz Conversion	MemCpy
Layer1	0.26	0.05	0.29	0.05	0.12	0.03	0.05
Layer2	0.96	0.39	1.34	0.32	0.51	0.02	0.36
Layer3	0.76	0.59	1.16	0.59	0.20	0.01	0.67
Layer4	0.74	1.03	1.78	1.11	0.31	0.01	0.83
Layer5	0.51	0.58	0.89	0.59	0.26	0.01	0.66

Figure 10: Batch Size : 1

Naïve GPU			ShMem		Toeplitz		
Layer	Kernel	Memcpy	Kernel	Memcpy	Matrix Multiply	Toeplitz Conversion	MemCpy
Layer1	8.50	4.51	8.27	4.24	3.04	1.24	4.36
Layer2	31.51	1.72	35.62	2.66	12.78	0.77	2.42
Layer3	27.36	1.64	23.18	1.06	4.80	0.10	1.63
Layer4	23.31	2.33	74.65	0.00	6.57	0.15	1.85
Layer5	17.97	2.26	21.04	2.29	4.47	0.15	2.26

Figure 11: Batch Size : 32

Naïve GPU			ShMem		Toeplitz		
Layers	Kernel	Memcpy	Kernel	Memcpy	Matrix Multiply	Toeplitz Conversion	MemCpy
Layer1	32.57	19.08	29.84	18.42	11.90	5.27	18.02
Layer2	112.01	11.40	136.40	10.17	31.38	3.12	38.44
Layer3	110.28	6.98	82.90	6.40	17.32	0.41	6.27
Layer4	93.26	9.56	126.01	10.65	25.95	0.61	9.50
Layer5	71.78	6.70	83.06	11.18	17.35	0.61	10.47

Figure 12: Batch Size : 128