Taylor Kawate, Arunan Elamaran
CS 259 – System Design/Architecture: Optimizing Hardware for Machine Learning

Mini-Project 1

Question 1: What was your basic parallelization strategy? Are there limitations in scaling this strategy, or consequences for throughput or latency?

General Motivation

Given that the convolution and classifier kernels were targeting a GPU, our main goal was to first maximize all of the parallel streaming multiprocessor (SM) units on the GPU. This was both to fully utilize the GPU's computational hardware and ensure that via high-throughput parallelization, the GPUs large memory latencies are hidden. We derived the following strategies from the original codes for convolution and classifier from the *fp-diannao* repository.

Convolution

DianNao Strategy

The *convolution_layer_blocked* function performs convolution by combining tiling, sliding windows, and partial sum reuse to improve memory efficiency/reuse. The spatial dimensions (Ny, Nx) are divided into tiles of size Ty and Tx, while output channels (Nn) are blocked into tiles of size Tnn. Within each tile, the function slides the convolutional kernel (Ky, Kx) over the input using a sliding window approach. For each output position, the input channels (Ni) are also processed in smaller tiles of size Ti, improving reuse and reducing memory bandwidth. Partial sums are accumulated inside a sum[n] array across the sliding window positions before any result is written back. This local accumulation avoids redundant memory stores and keeps intermediate results in registers which are fast to access.

CUDA Strategy

The parallelization strategy assigns one CUDA thread to compute the convolution for a single output position (y, x) and a single output channel n. The threads are organized into a 3D grid, where blockIdx.x and threadIdx.x determine the output column x, blockIdx.y and threadIdx.y determine the output row y, and blockIdx.z determines the output feature map n. This structure parallelizes computation across both the spatial dimensions and the output channels simultaneously, allowing the output neurons to be computed independently and concurrently. Each thread loads only the weights and input activations it needs for its assigned output neuron, leading to no thread synchronization overhead, and thus high independence. Additionally, since each thread writes to a unique output location in global memory, there are no write conflicts or need for explicit synchronization between threads.

The excessive tiling and sliding window method used in the original blocked CPU implementation is not necessary for the CUDA parallelization strategy because parallelism at the thread level already naturally breaks up the work into small, independent pieces. In the CPU version, tiling was crucial to improve cache reuse and manage limited working memory, since a

single core had to handle large portions of the output and repeatedly fetch data. In contrast, in CUDA, each thread computes only one output neuron (y, x, n), meaning the amount of data a thread processes is small and fits easily into fast-access memory without further tiling. CUDA's hardware scheduling, thread-level parallelism, and memory coalescing mechanisms take over the role of hiding memory latency and improving throughput without needing manual tiling within the kernel.

Overlapping memory accesses occur for the input activations (neuron_i) but not for the weights (synapse). Since each thread computes a single output neuron at (y, x, n) by reading a Ky × Kx patch of input pixels, and neighboring output threads (especially threads computing adjacent (y, x) output locations) will access overlapping regions of neuron_i, different threads will often read many of the same input activation values. However, for the weights (synapse), each output channel n uses a different set of filters, so threads with different n values access disjoint weight memory. Therefore, input activation memory reads are spatially overlapping across neighboring threads, while weight memory reads are isolated per output channel. This spatial overlap on the input side could potentially be exploited by using shared memory to load input tiles once and let threads reuse the data locally, but in the current implementation, all input loads are done independently by each thread from global memory.

The explicit loops over Ky, Kx, and Ni inside each thread are used to accumulate the convolution sum for the thread's assigned output position and channel. The looping over input channels and kernel positions ensures that the convolution operation remains consistent with the original implementation. The outer loops over spatial and channel dimensions are eliminated and replaced by thread indexing. As a result, each thread independently handles its own computation without requiring outer-level for-loops over (y, x, n).

CUDA Strategy Limitations

There are several limitations when scaling to larger problem sizes. First, each thread independently loads all the input activations it needs from global memory, even though neighboring threads often access overlapping input regions, leading to redundant memory traffic and poor exploitation of spatial reuse. Second, there is no use of scratchpad memory for either the input activations (neuron_i) or the synapse weights (synapse), which means the implementation heavily depends on global memory bandwidth, and memory accesses may not be fully coalesced, especially when accessing across input channels. Third, as the number of input channels (Ni) or kernel size (Ky × Kx) grows, the inner loops inside each thread become longer, increasing per-thread latency. Finally, although there is no explicit synchronization needed between threads, the overall memory system becomes a bottleneck as more threads compete for input data from global memory, particularly when scaling to larger feature maps or deeper layers.

Classifier

DianNao Strategy

classifier_layer_blocked uses hierarchical tiling over both the output neurons (Nn) and the input neurons (Ni) to improve computation and memory access efficiency. The outermost

loop tiles the output neurons into blocks of size Tnn, and within each output block, the input neurons are further tiled into smaller blocks of size Tii. Within these tiles, finer-grained loops over Tn output neurons and Ti input neurons are used to further partition the computation into smaller manageable chunks, allowing better use of fast memory. The innermost computation accumulates partial sums for a small group of input neurons before adding to the overall output sum (sum[n]), which improves reuse of synapse and input activations within the small tiles and reduces the overhead of repeatedly fetching the same data. Unlike a convolution layer that requires sliding windows over spatial input, this classifier is a simple matrix-vector multiply without spatial movement.

CUDA Strategy

The parallelization strategy assigns one CUDA thread to compute the full sum for one output neuron n, meaning each thread is responsible for completing all the multiply-accumulate operations associated with a specific output. The threads are organized in a 1D grid along the output neuron dimension (Nn), where blockIdx.x and threadIdx.x determine the output neuron index n. Threads independently accumulate their sums and write to unique output locations, so no synchronization is needed between threads for collaboration on a shared output element. However, to improve efficiency, all threads within a block collaboratively load a tile of input activations (neuron_i) into scratchpad memory to avoid repeatedly fetching the same input values from global memory. By using shared memory, the design reduces global memory traffic and improves reuse of the input neuron values across different output computations.

The input neurons (Ni) are processed in tiles of size Tii, where each tile is loaded into shared memory during each iteration of the outer loop. Inside the loop, each thread independently computes partial sums using the shared inputs, multiplying them with its corresponding synapse weights. The for loop over tile_i steps through all the input tiles, allowing the full sum across Ni inputs to be built piece by piece. Synchronization (__syncthreads()) is used to ensure that scratchpad memory is fully loaded before threads proceed to computation, and also to ensure that all threads in a block have finished using the values in that tile before moving on. After all tiles are processed, the final accumulated sum is passed through an activation function and stored to the output array.

CUDA Strategy Limitations

There are important limitations in scaling this CUDA classifier strategy, which affect both throughput and latency. First, because all threads in a block share a small buffer (shared_neuron_i[Tii]), the maximum tile size Tii is constrained by the available scratchpad memory on each Streaming Multiprocessor (SM). Second, if the number of output neurons (Nn) is not large enough relative to the number of threads per block, the GPU may experience underutilization, leading to low occupancy and reduced overall throughput. Third, each tile load requires two synchronization points (__syncthreads()), and as the number of input tiles grows, the cumulative synchronization overhead can become non-negligible and hurt performance, especially when the computation per tile is small compared to synchronization costs. Another limitation is that while loading neuron inputs into scratchpad memory improves locality, each thread still independently fetches its corresponding synapse weights from global memory.

Finally, the strict serial accumulation across many tiles within each thread becomes a bottleneck at large scale.

Invocation Strategy (Calling from *main*)

This strategy gave us one main parameter to tweak: the threads per block (blocks per grid were calculated to scale properly with the chosen threads per block). Through some preliminary testing for convolution, it became apparent that uneven dimensions (32 x 8, 8 x 32) did not give any significant performance boost compared to symmetric dimensions, so we focused on varying the dimensions evenly. We tested 8 x 8, 16 x 16, and 32 x 32 threads per block to see the impact on performance, with the expectation that more threads per block would simply perform better. Increasing the number of threads per block can improve thread scheduling efficiency and memory access patterns. Larger thread blocks allow for better occupancy, meaning more warps are available per SM, enabling the hardware scheduler to hide memory latency by switching between active warps when some are waiting for memory. In addition, when more threads within a block access neighboring memory addresses, memory accesses become more coalesced, resulting in fewer global memory transactions and higher effective memory bandwidth utilization.

This scaling was limited by the NVIDIA Titan V GPU being used, which had a maximum of 1024 threads per block (32 x 32 threads). There were no significant consequences for parallelizing this way, as it just more efficiently utilizes the vector lanes available to us; there are significant performance consequences for not parallelizing in this way, however, or for under-utilizing the vector lanes.

The classifier kernel was similar, except we experimented with 1D threads per block: 1, 32, 256, and 1024. Given that 1024 is the upper limit of threads per CUDA block, and since threads are scheduled in warps of 32, we decided that multiples of 32 threads should be used to fully utilize the hardware available to us, but also wanted to check below that threshold as well (1 thread).

Though not a central part of our parallelization strategy, we did also do some experimentation with varying the data types used. We compared FP-16 32 x 32 with FP-32 8 x 8, 16 x 16, and 32 x 32. These tests were mainly to verify our intuition that lower precision data types would generally execute faster due to their smaller size (and therefore requiring lower bandwidth). In earlier iterations of our kernel, we also did some testing and profiling with INT8 to compare and contrast an even smaller data type with the FP-16 and FP-32 results. However, we did not include correctness checks at this early stage in the project, and since we cannot guarantee the profiling metrics are for a computationally correct kernel, we voided these results and re-profiled our correct kernels with just the FP-16 and FP-32 data types for simplicity and clarity. However, our early profiling results with INT-8 can be seen in the Appendix, where we can broadly assess that moving to smaller data types generally decreased the memory bandwidth, though at the cost of extra overhead due to casting and quantization.

We also did some testing with shared memory for the classifier kernel to see the impact of storing the neuron in shared memory. The same parallelization strategy was applied for this version of the classifier kernel, and the same thread patterns were tested for an even comparison. We compared the shared memory version against the non-shared version of the classifier for the 1024 thread per block test case.

Taylor Kawate, Arunan Elamaran
CS 259 – System Design/Architecture: Optimizing Hardware for Machine Learning

Question 2: What is the execution time of each kernel? (and throughput if you use batching)

Convolution

Test	Data Type	FP-32	FP-32	FP-32	FP-16
	Thread Dimensions per Block	8 x 8	16 x 16	32 x 32	32 x 32
GPU activities	convolution_layer	128.10ms	122.08ms	255.14ms	75.733ms
	[CUDA memcpy DtoH]	18.352ms	17.431ms	17.554ms	8.3134ms
	[CUDA memcpy HtoD]	6.7995ms	6.7769ms	6.8067ms	3.1578ms
	SUM	153.252ms	146.288ms	279.501ms	87.204ms
API calls	cudaMalloc	367.04ms	368.95ms	364.36ms	374.54ms
	cudaDeviceSynchronize	130.60ms	124.44ms	257.53ms	78.098ms
	cudaMemcpy	27.666ms	27.294ms	27.436ms	15.903ms
	cudaLaunchKernel	2.8554ms	935.53us	912.11us	2.8513ms
	cudaFree	1.8846ms	1.6645ms	1.6786ms	2.0473ms
	cuDeviceGetAttribute	538.58us	547.15us	504.82us	505.95us
	cuDeviceGetName	68.376us	60.050us	47.245us	43.643us
	cuDeviceGetPCIBusId	41.655us	38.306us	39.362us	33.311us
	cudaGetLastError	6.7580us	6.5530us	6.7280us	7.6900us
	cuDeviceGetCount	4.3740us	4.9580us	4.0410us	4.1750us
	cuDeviceGet	2.1650us	3.1990us	2.3250us	2.3330us
	cuModuleGetLoadingMode	996ns	1.1300us	913ns	2.0790us
	cuDeviceTotalMem	1.3310us	2.8950us	1.1510us	1.0500us
	cuDeviceGetUuid	979ns	808ns	812ns	760ns
	SUM	530.711ms	523.949ms	652.524ms	474.041ms
	TOTAL SUM	683.963ms	670.237ms	932.025ms	561.245ms

Figure 1: CONV1 Execution Times

Test	Data Type	FP-32	FP-32	FP-32	FP-16
	Thread Dimensions per Block	8 x 8	16 x 16	32 x 32	32 x 32
Metrics	Floating Point Operations(Single Precision)	3702587392	3702587392	3702587392	3700980686
	Floating Point Operations(Half Precision)	0	0	0	3712218875
	Integer Instructions	565182464	565182464	565182464	1377629924
	Executed IPC	0.042422	0.040827	0.018926	0.114846
	Multiprocessor Activity	99.48%	99.17%	99.32%	98.30%
	Achieved Occupancy	0.946287	0.973672	0.982331	0.966252
	Device Memory Read Throughput	549.43GB/s	559.68GB/s	591.56GB/s	509.43GB/s
	Device Memory Write Throughput	2.2065GB/s	2.1040GB/s	0.9968GB/s	3.3004GB/s
	Device Memory Utilization	Max (10)	Max (10)	Max (10)	High (9)
	Global Memory Load Efficiency	12.58%	12.56%	12.56%	6.27%
	Issue Stall Reasons (Instructions Fetch)	0.18%	0.18%	0.00%	0.01%
	Issue Stall Reasons (Data Request)	11.93%	10.05%	1.11%	19.40%
	Issue Stall Reasons (Memory Throttle)	87.20%	89.07%	98.45%	79.52%
	Issue Stall Reasons (Not Selected)	0.53%	0.53%	0.30%	0.77%
	Issue Stall Reasons (Other)	0.01%	0.01%	0.00%	0.01%
	Issue Stall Reasons (Pipe Busy)	0.01%	0.03%	0.06%	0.02%
	Issue Stall Reasons (Synchronization)	0.00%	0.00%	0.00%	0.00%

Figure 2: CONV1 Profiling Results

Test	Data Type	FP-32	FP-32	FP-32	FP-16
	Thread Dimensions per Block	8 x 8	16 x 16	32 x 32	32 x 32
GPU activities	convolution_layer	13.878ms	10.696ms	10.656ms	10.624ms
	[CUDA memcpy HtoD]	5.0928ms	4.8422ms	4.7542ms	2.1243ms
	[CUDA memcpy DtoH]	55.296us	55.647us	55.584us	28.832us
	SUM	19.026ms	15.594ms	15.466ms	12.777ms
API calls	cudaMalloc	371.12ms	367.86ms	369.90ms	374.95ms
	cudaLaunchKernel	62.100ms	2.7268ms	1.5712ms	73.600ms
	cudaDeviceSynchronize	16.355ms	13.074ms	13.183ms	12.951ms
	cudaMemcpy	7.3517ms	6.9881ms	912.11us	7.0826ms
	cudaFree	1.7185ms	1.5066ms	1.6786ms	2.3475ms
	cuDeviceGetAttribute	558.34us	505.47us	504.82us	554.62us
	cuDeviceGetName	80.091us	43.326us	47.245us	68.130us
	cuDeviceGetPCIBusId	40.165us	38.666us	39.362us	41.159us
	cudaGetLastError	17.331us	7.0910us	6.7280us	16.500us
	cuDeviceGetCount	4.6860us	3.5930us	4.0410us	4.1230us
	cuDeviceGet	1.7590us	2.0150us	2.3250us	2.0530us
	cuDeviceTotalMem	1.1840us	1.1230us	913ns	1.4140us
	cuModuleGetLoadingMode	1.0910us	1.0650us	1.1510us	1.2420us
	cuDeviceGetUuid	693ns	732ns	812ns	794ns
	SUM	459.351ms	392.759ms	387.852ms	471.621ms
	TOTAL SUM	478.377ms	408.353ms	403.318ms	484.398ms

Figure 3: CONV2 Execution Times

Test	Data Type	FP-32	FP-32	FP-32	FP-16
	Thread Dimensions per Block	8 x 8	16 x 16	32 x 32	32 x 32
Metrics	Floating Point Operations(Single Precision)	924944384	924944384	924944384	924894149
	Floating Point Operations(Half Precision)	0	0	0	925245322
	Integer Instructions	271003648	271003648	272969728	272768906
	Executed IPC	0.147923	0.126813	0.249877	0.382886
	Multiprocessor Activity	75.65%	91.24%	91.28%	91.32%
	Achieved Occupancy	0.743604	0.646858	0.480812	0.482057
	Device Memory Read Throughput	.3045GB/s	1.8558GB/s	1.8166GB/s	933.04MB/s
	Device Memory Write Throughput	15.47MB/s	235.78MB/s	238.82MB/s	230.22MB/s
	Device Memory Utilization	Low (1)	Low (1)	Low (1)	Low (1)
	Global Memory Load Efficiency	12.50%	12.51%	12.50%	6.25%
	Issue Stall Reasons (Instructions Fetch)	0.04%	0.04%	0.08%	0.10%
	Issue Stall Reasons (Data Request)	2.34%	2.16%	2.83%	6.73%
	Issue Stall Reasons (Memory Throttle)	95.18%	95.67%	92.53%	88.19%
	Issue Stall Reasons (Not Selected)	1.86%	1.56%	3.01%	3.00%
	Issue Stall Reasons (Other)	0.02%	0.01%	0.05%	0.09%
	Issue Stall Reasons (Pipe Busy)	0.00%	0.00%	0.02%	0.03%
	Issue Stall Reasons (Synchronization)	0.00%	0.00%	0.00%	0.00%

Figure 4: CONV2 Profiling Results

As seen in Figure 1, CONV1 execution times were dominated by host-side API calls, as expected. The actual kernel execution time on the GPU was often about 4-5x faster than the API calls. FP-16 had the fastest kernel execution time of ~75ms, which was expected as it was a smaller data type (half the precision of FP-32) and required less memory bandwidth. This can be seen in Figure 2, where the memory utilization was lower for FP-16 32 x 32 compared to FP-32 32 x 32. It notably took 3x more time to launch the kernel for FP-16 than the comparable FP-32 kernel, though the tradeoff was minimal (~2ms) when accounting for the drastically reduced kernel execution time (~180ms), making FP-16 still have the fastest overall execution time.

At full precision, the 16 x 16 thread blocking was actually significantly faster than the 32 x 32 block when executing the convolution kernel for CONV1 (with a negligible difference in CONV2). From Figure 2, it is evidently not a result of memory efficiency, but likely a result of stalls due to memory throttling when dealing with larger thread blocks.

CONV2 had somewhat similar results, with FP-16 32 x 32 still having the fastest kernel execution time, but only by ~30us due to the drastically reduced kernel execution times seen across the board (~15ms). However, this was at the cost of more overhead on the host-side, as FP-16 required extra data conversions in multiple parts of the code, resulting in the slower API calls seen in Figure 1. As a result, the overhead penalty actually outweighed the benefits, and FP-32 32 x 32 had the fastest overall execution time.

Classifier

Test	Data Type	FP-32	FP-32	FP-32	FP-32	FP-32	FP-32	FP-32
	Thread Dimensions per Block	1	32	32	256	256	1024	1024
	Shared/ Not Shared	Not Shared	Not Shared	Shared	Not Shared	Shared	Not Shared	Shared
GPU activities	[CUDA memcpy HtoD]	222.92ms	262.63ms	8.8377ms	314.82ms	269.82ms	317.31ms	276.40ms
	simple_classifier_layer_cuda	6.4615ms	4.8089ms	177.44us	9.9392ms	19.375ms	41.269ms	19.420ms
	[CUDA memcpy DtoH]	3.5840us	3.9040us	2.1120us	3.3920us	3.3920us	2.9120us	2.8800us
	SUM	229.385ms	267.443ms	9.017ms	324.763ms	289.198ms	358.582ms	295.823ms
API calls	cudaMalloc	415.03ms	365.67ms	384.32ms	363.44ms	388.14ms	361.84ms	371.18ms
	cudaMemcpy	223.70ms	268.54ms	309.28ms	320.54ms	274.26ms	321.81ms	278.36ms
	cudaDeviceSynchronize	8.6264ms	6.9448ms	21.583ms	12.055ms	21.531ms	43.440ms	21.583ms
	cudaFree	2.3875ms	1.6726ms	1.6571ms	1.6915ms	1.6864ms	2.7435ms	1.6350ms
	cudaLaunchKernel	1.3610ms	2.0048ms	1.9993ms	2.3781ms	1.8535ms	2.2106ms	2.9347ms
	cuDeviceGetAttribute	519.72us	536.85us	543.78us	516.31us	542.10us	548.29us	500.04us
	cuDeviceGetPCIBusId	52.606us	34.935us	36.502us	36.222us	36.233us	34.213us	36.262us
	cuDeviceGetName	51.130us	68.438us	72.929us	47.064us	71.354us	159.97us	44.426us
	cudaGetLastError	31.483us	39.523us	11.329us	10.637us	10.221us	50.398us	10.527us
	cuDeviceGetCount	3.2920us	3.6190us	3.9430us	3.5150us	3.7810us	4.0600us	3.9010us
	cuDeviceGet	1.9920us	2.1570us	2.5520us	2.2230us	1.8970us	2.9440us	1.8600us
	cuDeviceTotalMem	1.3880us	1.1510us	2.0930us	1.0260us	927ns	1.2270us	1.2510us
	cuModuleGetLoadingMode	948ns	995ns	1.0450us	989ns	1.3450us	1.2120us	1.0150us
	cuDeviceGetUuid	699ns	708ns	706ns	806ns	711ns	751ns	665ns
	SUM	651.768ms	645.521ms	719.514ms	700.723ms	688.139ms	732.847ms	676.293ms
	TOTAL SUM	881.153ms	912.964ms	728.531ms	1025.486ms	977.337ms	1091.429ms	972.116ms

Figure 5: CLASS1 Execution Times

Test	Data Type	FP-32	FP-32	FP-32	FP-32	FP-32	FP-32	FP-32
	Thread Dimensions per Block	1	32	32	256	256	1024	1024
	Shared/ Not Shared	Not Shared	Not Shared	Shared	Not Shared	Shared	Not Shared	Shared
Metrics	Floating Point Operations(Single Precision)	205524992	205524992	205524992	205524992	205524992	205524992	205524992
	Floating Point Operations(Half Precision)	0	0	0	0	0	0	0
	Integer Instructions	38563840	38563840	1437696	38563840	1437696	38563840	1437696
	Executed IPC	0.995056	0.047485	0.067926	0.098424	0.067934	0.102046	0.067942
	Multiprocessor Activity	86.23%	89.98%	9.92%	19.89%	9.94%	4.98%	9.96%
	Achieved Occupancy	0.421615	0.025858	0.249999	0.123335	0.249999	0.492519	0.249999
	Device Memory Read Throughput	98.647GB/s	159.02GB/s	42.391GB/s	72.743GB/s	42.445GB/s	18.879GB/s	42.576GB/s
	Device Memory Write Throughput	276.25MB/s	712.10MB/s	114.37MB/s	219.47MB/s	116.89MB/s	56.279MB/s	115.47MB/s
	Device Memory Utilization	Low (2)	Low (3)	Low (1)	Low (2)	Low (1)	Low (1)	Low (1)
	Global Memory Load Efficiency	12.50%	12.50%	12.68%	12.51%	12.68%	12.65%	12.68%
	Issue Stall Reasons (Instructions Fetch)	0.37%	0.30%	0.06%	0.14%	0.03%	0.03%	0.03%
	Issue Stall Reasons (Data Request)	24.60%	89.39%	0.50%	13.84%	0.50%	4.88%	0.50%
	Issue Stall Reasons (Memory Throttle)	57.36%	3.95%	91.58%	82.82%	91.61%	93.15%	91.63%
	Issue Stall Reasons (Not Selected)	9.09%	0.00%	0.51%	0.48%	0.51%	1.24%	0.51%
	Issue Stall Reasons (Other)	0.19%	0.00%	0.01%	0.01%	0.00%	0.00%	0.00%
	Issue Stall Reasons (Pipe Busy)	0.00%	0.00%	0.02%	0.00%	0.02%	0.00%	0.02%
	Issue Stall Reasons (Synchronization)	0.00%	0.00%	3.92%	0.00%	3.92%	0.00%	3.90%

Figure 6: CLASS1 Profiling Results

Test	Data Type	FP-32	FP-32	FP-32	FP-32	FP-32	FP-32	FP-32
	Thread Dimensions per Block	1	32	32	256	256	1024	1024
	Shared/ Not Shared	Not Shared	Not Shared	Shared	Not Shared	Shared	Not Shared	Shared
GPU activities	[CUDA memcpy HtoD]	9.2012ms	8.7928ms	8.8377ms	8.7014ms	9.2159ms	8.3954ms	8.6200ms
	simple_classifier_layer_cuda	193.25us	226.30us	177.44us	843.00us	177.12us	7.9043ms	177.25us
	[CUDA memcpy DtoH]	2.0480us	2.0480us	2.1120us	2.0800us	2.1120us	2.5600us	2.0800us
	SUM	9.396ms	9.021ms	9.017ms	9.546ms	9.395ms	16.302ms	8.799ms
API calls	cudaMalloc	364.86ms	371.19ms	383.35ms	366.12ms	381.61ms	423.62ms	378.46ms
	cudaMemcpy	9.9530ms	9.5285ms	9.5700ms	9.5072ms	9.9778ms	11.182ms	9.4760ms
	cudaLaunchKernel	2.8294ms	2.9383ms	1.7177ms	2.0443ms	2.3227ms	5.0852ms	2.1257ms
	cudaDeviceSynchronize	2.3518ms	2.3817ms	2.3367ms	3.0041ms	2.4968ms	12.271ms	2.3400ms
	cudaFree	963.64us	959.38us	731.76us	700.87us	774.67us	775.46us	715.04us
	cuDeviceGetAttribute	530.18us	583.02us	527.10us	505.73us	550.32us	533.68us	545.80us
	cuDeviceGetName	55.828us	73.135us	66.549us	43.163us	86.638us	60.386us	76.508us
	cuDeviceGetPCIBusId	36.908us	35.974us	36.611us	33.844us	36.633us	36.518us	34.858us
	cudaGetLastError	4.8110us	6.2920us	2.5580us	3.7710us	1.8230us	2.5980us	1.5050us
	cuDeviceGetCount	3.7990us	3.8690us	4.5740us	2.3210us	4.2370us	4.1310us	4.1680us
	cuDeviceGet	1.7960us	2.2330us	1.8590us	1.9910us	1.8090us	2.0380us	2.0740us
	cuDeviceTotalMem	1.0730us	1.5400us	1.0920us	952ns	1.1350us	1.3110us	1.2370us
	cuModuleGetLoadingMode	973ns	1.1210us	864ns	931ns	1.0470us	1.1450us	1.0190us
	cuDeviceGetUuid	785ns	737ns	673ns	686ns	662ns	840ns	723ns
	SUM	381.594ms	387.706ms	398.348ms	381.970ms	397.866ms	453.576ms	393.785ms
	TOTAL SUM	390.990ms	396.727ms	407.365ms	391.516ms	407.261ms	469.878ms	402.584ms

Figure 7: CLASS2 Execution Times

Test	Data Type	FP-32	FP-32	FP-32	FP-32	FP-32	FP-32	FP-32
	Thread Dimensions per Block	1	32	32	256	256	1024	1024
	Shared/ Not Shared	Not Shared	Not Shared	Shared	Not Shared	Shared	Not Shared	Shared
Metrics	Floating Point Operations(Single Precision)	8389632	8389632	8389632	8389632	8389632	8389632	8389632
	Floating Point Operations(Half Precision)	0	0	0	0	0	0	0
	Integer Instructions	1580032	1580032	1059840	1580032	1059840	1580032	1059840
	Executed IPC	0.726889	0.047596	0.049398	0.101246	0.0494	0.099687	0.049487
	Multiprocessor Activity	94.19%	38.72%	38.39%	4.96%	38.07%	1.25%	38.43%
	Achieved Occupancy	0.199112	0.015625	0.015625	0.123661	0.015625	0.492632	0.015625
	Device Memory Read Throughput	80.159GB/s	68.950GB/s	87.515GB/s	18.536GB/s	87.255GB/s	4.6257GB/s	87.458GB/s
	Device Memory Write Throughput	5.9218GB/s	6.9140GB/s	6.3258GB/s	1.3270GB/s	5.9299GB/s	338.83MB/s	5.9199GB/s
	Device Memory Utilization	Low (2)	Low (2)	Low (2)	Low (1)	Low (2)	Low (1)	Low (2)
	Global Memory Load Efficiency	12.50%	12.50%	12.84%	12.50%	12.84%	12.53%	12.84%
	Issue Stall Reasons (Instructions Fetch)	0.61%	0.61%	0.50%	0.16%	0.50%	0.06%	0.51%
	Issue Stall Reasons (Data Request)	55.97%	87.97%	77.47%	9.24%	77.49%	2.88%	77.45%
	Issue Stall Reasons (Memory Throttle)	27.11%	0.46%	6.42%	87.25%	6.42%	95.11%	6.43%
	Issue Stall Reasons (Not Selected)	2.52%	0.00%	0.00%	0.52%	0.00%	1.25%	0.00%
	Issue Stall Reasons (Other)	0.28%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	Issue Stall Reasons (Pipe Busy)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	Issue Stall Reasons (Synchronization)	0.00%	0.00%	1.92%	0.00%	1.92%	0.00%	1.93%

Figure 8: CLASS2 Profiling Results

The classifier kernel itself executed the fastest for CLASS1 in the 32 thread per block case when using scratchpad memory, with the kernel execution time at ~177.44 us, which is 27 times faster than the 32 thread per block case without scratchpad memory. This thus resulted in a ~9ms overall GPU time in the former case compared to 267 ms in the latter scenario. As can be seen in Figure 6, the scratchpad memory use cases exhibit low device memory utilization and low issue stalls for most reasons compared to the non-scratchpad implementations. Further the former case exhibited lower multiprocessor activity. The massive speedup observed when running the classifier with 32 threads per block and scratchpad memory occurs because this configuration likely balances thread parallelism, memory efficiency, and hardware utilization. The choice of 32 threads matches exactly one warp, ensuring full warp utilization without divergence or wasted hardware lanes (warp underutilization). This combination of scratchpad use, warp-efficient execution, and reduced memory bottlenecks enables significantly higher execution efficiency, as seen by the faster execution time.

For CLASS2 the fastest kernel execution time was ~177.12 us for the 256 thread per block case when using scratchpad memory. However, all of the kernels utilizing scratchpad memory executed in about the same time, all around 177 us. This is in stark contrast to the equivalent cases without scratchpad memory, averaging around 2291.1us, or about 13x slower. Unlike the convolution kernels, GPU time was dominated by the initial memory copying, while the host-side API calls still dominated the total execution time, though they were much more constant across the various tests. This makes sense as the weight matrix's size is a product of the size of the input vector multiplied by the size of the output vector, hence causing the input loading cost to be significantly greater than the output loading cost. The kernel execution times with scratchpad memory are almost identical across different thread block sizes because the computation becomes compute-bound rather than memory-bound. When scratchpad memory is used, redundant global memory reads are eliminated, meaning that memory bandwidth is no longer the primary bottleneck. Regardless of block size, each output neuron still must perform the same number of floating-point operations. Since the amount of computation per output neuron is fixed and the memory access cost is minimized, the overall kernel runtime is determined mainly by the total floating-point workload, which remains constant across different thread block sizes.

Taylor Kawate, Arunan Elamaran
CS 259 – System Design/Architecture: Optimizing Hardware for Machine Learning

Question 3: What do you suspect is the limiting factor for performance of each kernel (compute,dram,scratchpad)? Where does the implementation fall on the roofline model for your particular GPU? Please graph the results. For this, you can/should measure different bandwidths using nvprof.

Convolution

For CONV1, we can clearly see from Figure 2 and Figure 9 that we are fully utilizing the SMs in the Titan V GPU, which was intended as we wanted to max out the SM capabilities of the GPU so the compute bottleneck would be tied to the GPU, not our code. The high efficiency of the SMs (Figure 9) proves that we are not compute bound given this GPU as we are fully utilizing it and efficient with the usage as well.

We also had very high memory utilization (for CONV1, not CONV2) which seemed good initially, but upon further inspection, the memory efficiency was quite poor (~12%) as we did not explicitly thread our code (or use scratchpad) in a way to optimize for memory access reuse. Each thread in the convolution kernel reads a large number of input activations (neuron_i) and filter weights (synapse) from global memory without using scratchpad memory, and there is heavy overlapping of input accesses among neighboring threads that is not exploited. The memory efficiency remains extremely low, with less than 20% of theoretical efficiency achieved, suggesting that the memory accesses are poorly coalesced. As a result, even though the GPU's compute units are busy and memory bandwidth is heavily used, the convolution kernel is unable to reach high overall performance because it wastes memory bandwidth on inefficient access patterns. For CONV2, the memory utilization was low but so was the memory efficiency. As a result, the convolution kernel is memory bound for CONV1, and could be improved with a different threading/tiling structure. This is because even with infinite compute resources, we would still be memory (DRAM) bound due to our access patterns. CONV2 is more inconclusive, but likely still memory (DRAM) bound due to the low DRAM efficiency and high memory throttle stall percentage.

Classifier

For the classifier kernel, when shared memory is not used, the kernel is similarly limited by global memory bandwidth because each thread independently fetches input activations, resulting in high redundant memory traffic. The profiling data (Figure 6 & 8) shows high data request stalls, indicating the kernel is limited by DRAM throughput rather than computation. However, when shared memory is used, the classifier kernel transitions to being compute-bound, because scratchpad memory drastically reduces the pressure on DRAM bandwidth by keeping the inputs locally. Once the memory bottleneck is alleviated, the limiting factor becomes the total number of floating-point operations needed to compute each output neuron.

As the number of threads per block increases to 32, 256, and 1024, SM utilization improves slightly for Class1, particularly when shared memory is used, because more concurrent work is scheduled and redundant memory accesses are reduced. For CLASS2, a similar pattern

is observed. As the thread block size increases to 256 and 1024 threads per block, SM utilization improves, especially when shared memory is applied. DRAM utilization was generally low, and efficiency stayed relatively constant across the board like in the convolution case, regardless of sharing. Similarly, across all cases, memory efficiency remains low, suggesting that the global memory access patterns are still scattered and not well coalesced, regardless of the block size. However, this was expected as a notable flaw of the profiling was the inability to see the scratchpad utilization and efficiency (which is likely the main reason for the drastic execution time speedups in contrast to the non-shared memory cases). Overall, for CLASS1, increasing threads per block improves parallelism, and using shared memory helps move the kernel closer toward being compute-bound. Though it is harder to pinpoint the bottleneck in the classifier as opposed to the convolution kernel, the culprit for the classifier becomes more apparent when looking at the stalls from profiling. From Figure 6 and 8, both CLASS1 and CLASS2 issue stalls are dominated by memory throttle and dependency stalls, like with the convolution kernel.

Figures 9 and 10 showcase our convolution kernel's SM and DRAM utilization and efficiency more visually for the Titan V GPU. Figures 11 and 12 showcase the same metrics but for the classifier kernel.

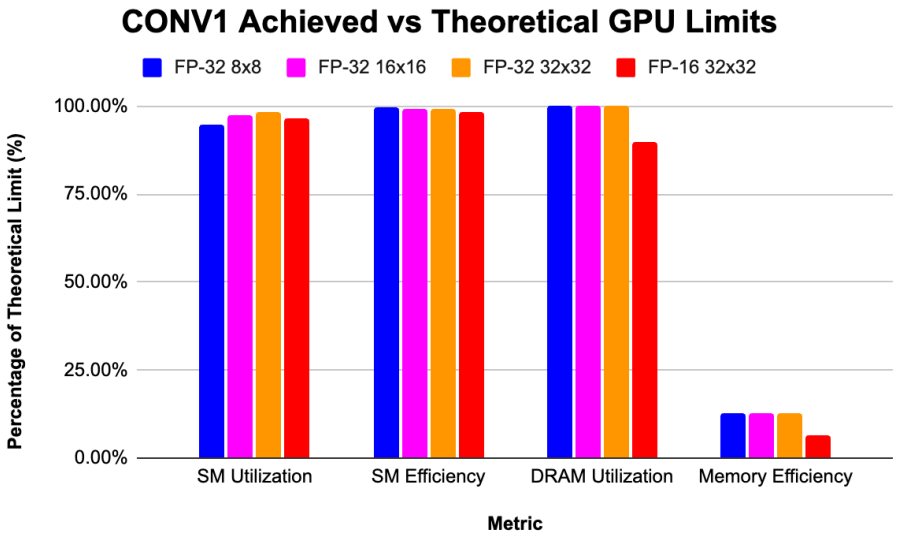

Figure 9: CONV1 Kernel Limitations

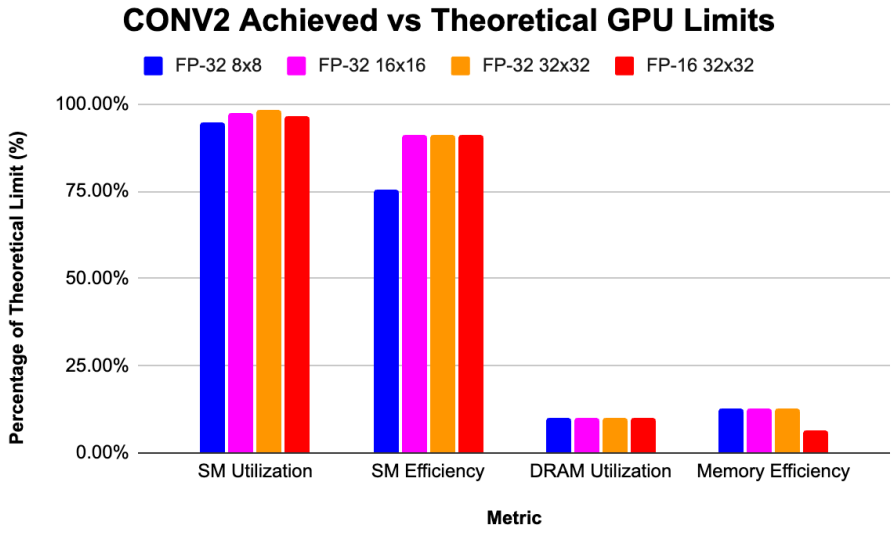

Figure 10: CONV2 Kernel Limitations

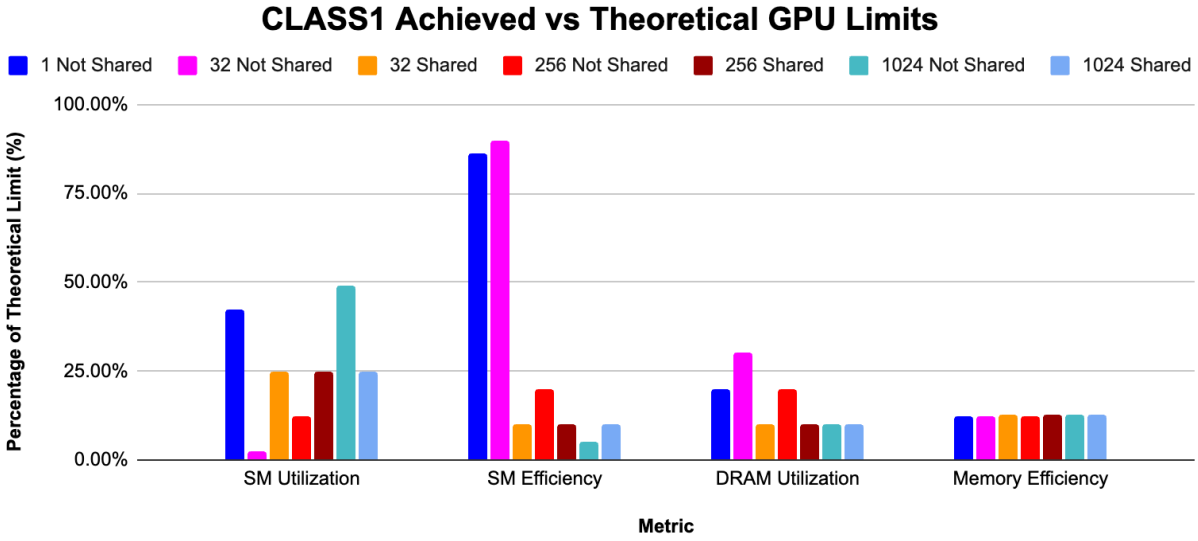

Figure 11: CLASS1 Kernel Limitations

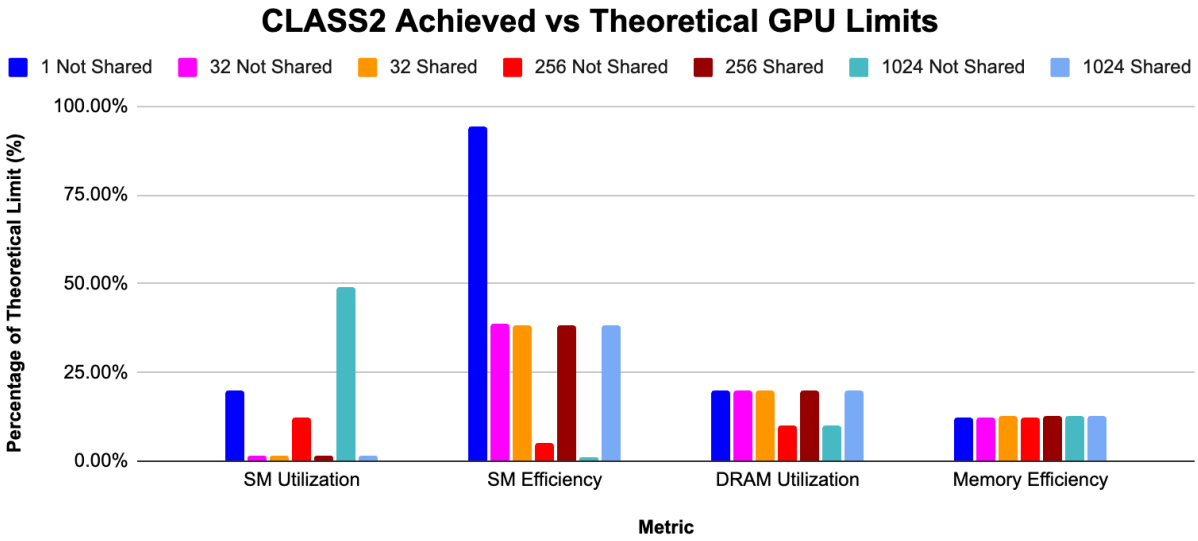

Figure 12: CLASS2 Kernel Limitations

Question 4: How does the implementation compare with CUDNN? (use same batch size in CUDNN) Please graph the results.

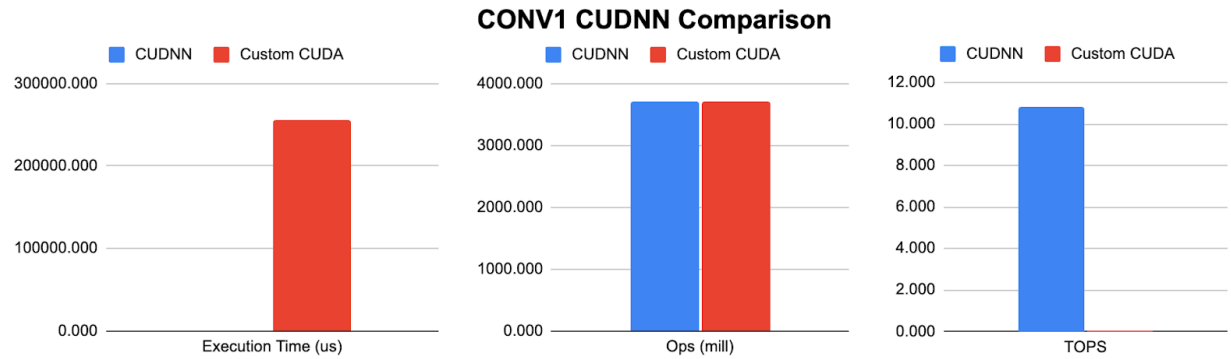

Figure 13: CONV1 CUDNN vs Custom CUDA Comparison

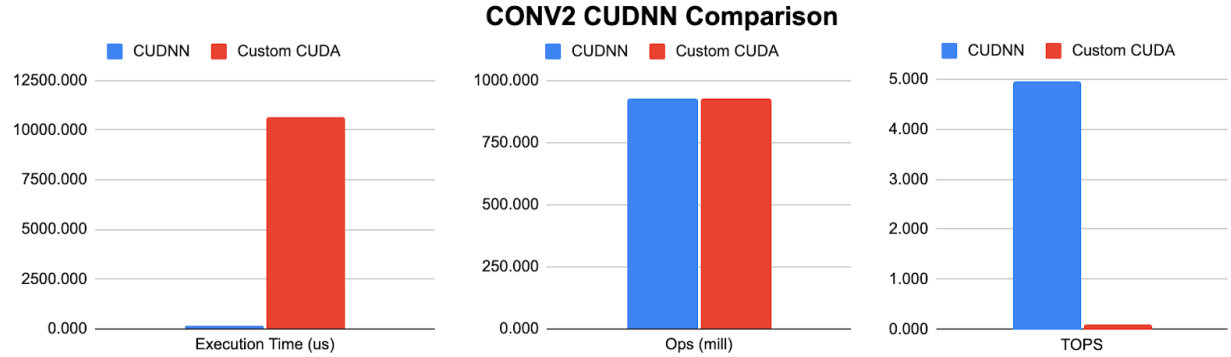

Figure 14: CONV2 CUDNN vs Custom CUDA Comparison

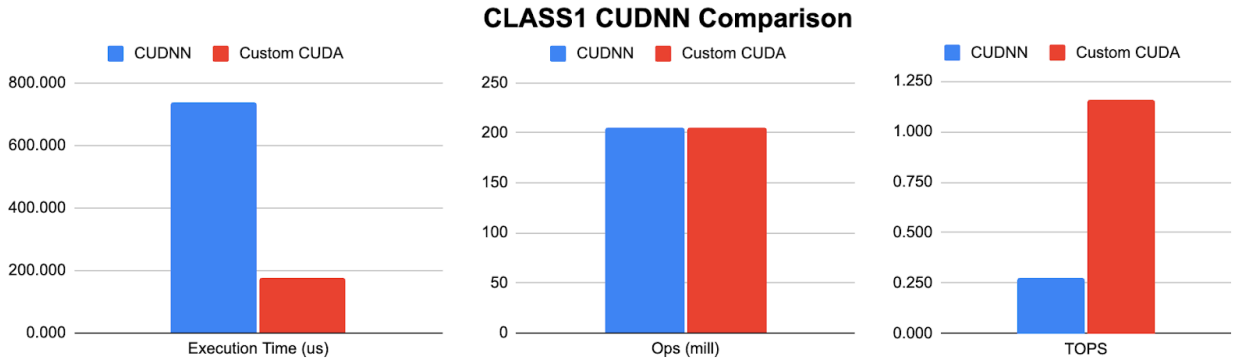

Figure 15: CLASS1 CUDNN vs Custom CUDA Comparison

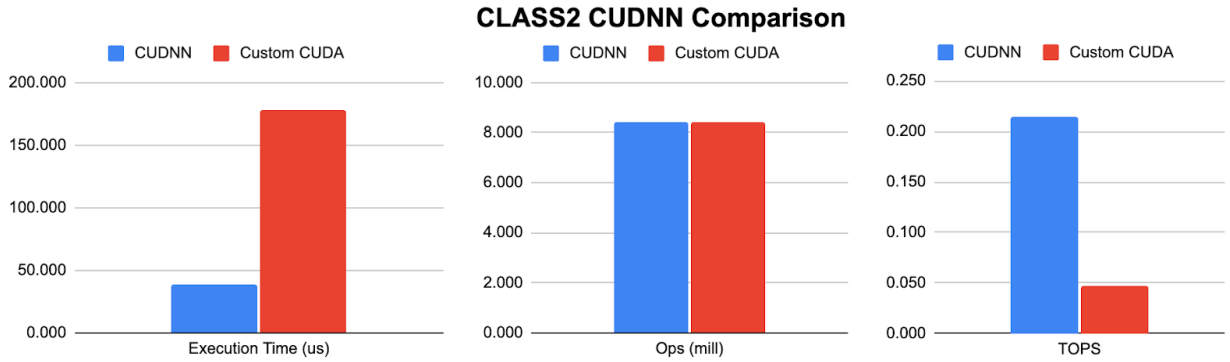

Figure 16: CLASS2 CUDNN vs Custom CUDA Comparison

Both our convolution and our classifier kernels had approximately the same amount of operations to be computed as the equivalent CUDNN versions.

Convolution

For CONV1 and CONV2, the CUDNN had drastically faster kernel execution times making the TOPS significantly higher for the CUDNN implementations. This is likely because our convolution kernel was more memory limited in our implementation compared to CUDNN. We saw a high percentage of stalls due to memory dependencies and memory throttling, which was likely not the case for CUDNN, hence the large performance differences seen in Figures 13 and 14.

Classifier

However, we had more varied results for our classifier kernel, as seen in Figures 15 and 16. Our classifier kernel outperformed CUDNN in execution time for CLASS1 for the shared case, which saw a drastically shorter execution time compared to the non-shared case (which was much slower than CUDNN). For CLASS2 we underperformed against CUDNN but not significantly, and this slowdown was once again likely due to our memory throttling and dependency stalls seen from our performance profiling.

Taylor Kawate, Arunan Elamaran
CS 259 – System Design/Architecture: Optimizing Hardware for Machine Learning

Question 5: What optimizations did you find most useful/least useful?

Most Useful:

1. Scratchpad (Shared Memory)
2. Loop Unrolling into Block & Thread Dimensions
3. Block Dimensions (as long as using shared memory to address global access patterns)

Least Useful:

1. Block Dimensions (as seen in convolution when not using shared memory)

Convolution

For the convolution kernel, simply spreading out the work across all SMs in the GPU proved to be one of the easiest yet most impactful optimizations, quickly achieving near 100% compute utilization and efficiency. However, while this massively improved the occupancy, it did not address the underlying memory bottlenecks inherent in convolution operations. Implementing memory optimizations, similar to what we successfully used in the classifier kernel, would likely have provided an even greater performance boost for convolution. In particular, using scratchpad (shared) memory to locally load and reuse overlapping regions of the input activations would have significantly reduced redundant global memory traffic, minimized bandwidth pressure, and improved memory efficiency. This would mirror the massive gains we observed for the classifier when using shared memory. Additionally, unrolling the larger tiled loops over output spatial dimensions and kernel windows into explicit block and thread indexing (rather than nested for-loops) turned out to be a simple but highly effective approach. This loop unrolling allowed more efficient mapping of work to hardware.

For the convolution kernel, the least useful optimization was tuning thread block size alone without addressing memory behavior. While varying the number of threads per block (e.g., 8×8, 16×16, 32×32) slightly affected SM utilization, it had minimal impact on overall execution time because the kernel remained fundamentally limited by inefficient global memory access patterns. Without incorporating shared memory or reorganizing memory access for better coalescing, block size tuning alone could not overcome the memory bottlenecks.

Classifier

For the classifier kernel, the introduction of scratchpad (shared) memory was by far the most useful and significant optimization. Since the classifier operation is fundamentally memory-bound, requiring each output neuron to access a large number of input neurons, moving input data into shared memory dramatically reduced the number of global memory transactions and allowed input activations to be reused efficiently across threads. This optimization alone made the classifier performance much more competitive compared to a highly optimized library like cuDNN, while without it, the kernel lagged substantially behind. On the other hand, our experiments with adjusting the number of threads per block produced mixed and inconclusive results for the classifier, with some minor differences in efficiency but no consistent trend or major improvements. Given the memory-bound nature of the classifier

without shared memory, changes in thread block size could not compensate for the lack of memory locality, making block size tuning the least useful optimization in this particular context and configuration. However, the combination of shared memory usage and proper block size tuning resulted in the most effective performance optimization, transforming the kernel from being memory-bound to compute-bound. Choosing an appropriate block size ensured that memory accesses into shared memory were perfectly coalesced, and that all threads could work together efficiently in loading and consuming data. The shared memory addressed the bandwidth bottleneck, while correct block sizing ensured full warp utilization and efficient resource scheduling.

Taylor Kawate, Arunan Elamaran
CS 259 – System Design/Architecture: Optimizing Hardware for Machine Learning

Appendix

Non-Shared CONV1 Data
(Data Type | Threads per Block | Total Execution Time)
FP32 8,8: 1136.123 ms
FP32 16,16: 1016.282 ms
FP32 32,32: 519.831 ms
FP16 8,8: 849.892 ms
FP16 16,16: 752.332 ms
FP16 32,32: 658.096 ms
INT8 8,8: 648.415 ms
INT8 16,16: 443.694 ms
INT8 32,32: 443.550 ms

Shared CONV1 Data
(Data Type | Threads per Block | Total Execution Time)
FP32 8,8: 535.369 ms
FP32 16,16: 524.248 ms
FP32 32,32: 527.259 ms
FP16 8,8: 493.455 ms
FP16 16,16: 471.146 ms
FP16 32,32: 468.641 ms
INT8 8,8: 400.944 ms
INT8 16,16: 407.190 ms
INT8 32,32: 398.665 ms

Non-Shared CONV2 Data
(Data Type | Threads per Block | Total Execution Time)
FP32 8,8: 414.218 ms
FP32 16,16: 407.248 ms
FP32 32,32: 403.190 ms
FP16 8,8: 406.792 ms
FP16 16,16: 399.378 ms
FP16 32,32: 406.361 ms
INT8 8,8: 404.467 ms
INT8 16,16: 392.333 ms
INT8 32,32: 413.647 ms

Shared CONV2 Data
(Data Type | Threads per Block | Total Execution Time)
FP32 8,8: 484.616 ms
FP32 16,16: 384.730 ms
FP32 32,32: 401.057 ms
FP16 8,8: 489.301 ms
FP16 16,16: 384.670 ms

Taylor Kawate, Arunan Elamaran
CS 259 – System Design/Architecture: Optimizing Hardware for Machine Learning

FP16 32,32: 400.871 ms
INT8 8,8: 496.914 ms
INT8 16,16: 382.040 ms
INT8 32,32: 393.098 ms