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

Mini-Project 2

Convolutional Kernel Explanation

```
__global__ void convolution_layer(VTYPE *synapse, VTYPE *neuron_i, VTYPE *neuron_n)
```

Figure 1: CUDA kernel function definition

The convolution_layer kernel performs a forward 2D convolution operation. The layouts of the weight, input, and output data structures respectively are as follows: synapse[Nn][Ni][Ky][Kx], neuron_i[Ni][NYPAD][NXPAD], neuron_n[Nn][NYSCL][NXSCL].

The data layout dimensions for the synapse and neuron_i arrays were rearranged in this project to improve memory access efficiency on the GPU. The change was made after discovering that our mini project 1 data dimension order was unideally organized for the kernel's memory access patterns. This change aligns the memory layout with the access patterns used inside the CUDA kernel, where threads loop over input channels and kernel dimensions in a nested fashion. By placing the most frequently iterated dimensions (Ni, Ky, and Kx for synapse; Ni for neuron_i) in the innermost positions, the kernel enables more coalesced global memory accesses.

Coalescing is crucial for performance because GPUs are optimized to fetch data in contiguous blocks when threads in a block/warp access adjacent memory locations. In the original layout, accesses along the input channel dimension were strided in memory by the output channel index (which never changed within a thread's execution), causing threads to load data from widely separated addresses, which results in uncoalesced memory transactions and relatively poor throughput. The rearranged layout ensures that threads accessing different input channels or kernel elements now access adjacent memory locations, thus reducing latency and improving bandwidth utilization.

```
// Thread & block indices
int tx = threadIdx.x;
int ty = threadIdx.y;
int bx = blockIdx.x * blockDim.x;
int by = blockIdx.y * blockDim.y;
int n = blockIdx.z;

int x = bx + tx;
int y = by + ty;
```

Figure 2: Per-thread contextualization code

At the beginning of the kernel, each thread computes the unique output coordinate within the overall output tensor that it will write to. The thread indices threadIdx.x and threadIdx.y define the local position of the thread within its CUDA block, while blockIdx.x and blockIdx.y, when scaled by the block dimensions, define the top-left corner of the output tile

being processed by the block. The output spatial coordinates (x, y) are derived by adding the local thread indices to the global block offsets. In addition, the third grid dimension, blockIdx.z, indexes the output channel n that this thread is responsible for computing a coordinate for. This setup ensures that each thread is mapped to a unique spatial location in a specific output feature map, allowing the kernel to compute many output activations in parallel.

The following two sections are executed over each of the input channels (as indicated by the outermost for loop in Figure 3, with iterator i going up to the bound Ni):

```
// Shared memory per input channel
  __shared__ float shmem_input[SHMEM_TILE_Y][SHMEM_TILE_X];

  VTYPE sum = 0;

  // Iterate over input channels
  for (int i = 0; i < Ni; i++) {
    // Each thread will cooperatively load the padded input tile into shared memory
    for (int dy = ty; dy < SHMEM_TILE_Y; dy += blockDim.y) {
      for (int dx = tx; dx < SHMEM_TILE_X; dx += blockDim.x) {
        int global_y = by + dy;
        int global_x = bx + dx;

        if (global_y < NYPAD && global_x < NXPAD) {
          int input_idx = (i * NYPAD + global_y) * NXPAD + global_x;
          shmem_input[dy][dx] = neuron_i[input_idx];
        }
      }
    }

    __syncthreads();  // Ensure all data is loaded
```

Figure 3: Shared Memory Load

To improve memory access efficiency, each CUDA block cooperatively loads a tile of the input tensor from global memory into shared memory. This tile corresponds to the region of the input required to compute the block's corresponding output region, including the necessary halo regions determined by the kernel size (Kx and Ky). The shared memory buffer shmem_input is sized to store the padded input tile. To optimally load the input array into memory, each thread within a block loads a contiguous block of elements per iteration of the innermost loop. Whereas this ensures coalesced accesses to the device memory for faster loading, the undesired approach partitions the input tensor into blocks handled by each thread, thus resulting in uncoalesced memory accesses and wasted data reads. Each element in shared memory is loaded from the global input array neuron_i, taking care to avoid out-of-bounds accesses. This stage is repeated for each input channel (Ni), and a synchronization barrier (__syncthreads()) is used to ensure that all threads complete the data loading before any computation begins. By staging the input in shared memory, the kernel significantly reduces redundant global memory accesses and

enables multiple threads to reuse the same input values, which is especially beneficial when kernel windows overlap.

```
// Compute convolution only if output thread is within bounds
    if (x < NXSCL && y < NYSCL && n < Nn) {
      for (int ky = 0; ky < Ky; ky++) {
        for (int kx = 0; kx < Kx; kx++) {
          int sh_y = ty + ky;
          int sh_x = tx + kx;
          VTYPE val = shmem_input[sh_y][sh_x];

          int synapse_idx = ((n * Ni + i) * Ky + ky) * Kx + kx;
          sum += synapse[synapse_idx] * val;
        }
      }
    }

    __syncthreads();  // Wait before loading next input channel
  }
```

Figure 4: Sum Compute

Once the required input tile is fully loaded into shared memory, each thread performs the convolution operation for its assigned output position. For every input channel, the thread iterates over the kernel window (Ky × Kx) and retrieves the corresponding input values from the shared memory tile. Simultaneously, it accesses the appropriate weights from the synapse array using a flattened index that reflects the [Nn][Ni][Ky][Kx] structure. The thread performs a multiply-accumulate (MAC) operation for each kernel element, accumulating the result in a local sum variable. Another synchronization barrier is placed at the end of each input-channel loop to ensure that the shared memory tile is not overwritten before all threads complete the current channel's computation.

```
// Write result if output index is valid
  if (x < NXSCL && y < NYSCL && n < Nn) {
    int output_idx = (n * NYSCL + y) * NXSCL + x;
    neuron_n[output_idx] = transfer_d(sum);
  }
```

Figure 5: Output Write

After the convolution result has been fully computed for the thread's output position and channel, the final step is to write the result back to global memory. The thread calculates the linear index into the output array neuron_n based on its assigned output channel and spatial location. Before storing the result, the value is optionally through an activation function/transformation. Because the threads are spatially aligned with the output tensor and are laid out contiguously in x, the global memory writes are mostly coalesced, which improves memory throughput and write efficiency.

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

Performance Model Explanation

Inputs

The following parameters directly influence the problem sizes and convolutional kernel efficiency/implementation:

- **Nx & Ny**: These define the spatial dimensions of the input feature map. These values determine the size of the input tensor that the kernel operates on. They also influence the output size, padded dimensions (NXPAD, NYPAD), and the shared memory footprint, since the input is convolved with a filter kernel and needs to be padded accordingly.
- **Kx & Ky**: These define the spatial dimensions of the filter. These parameters control how large the receptive field is for each output value. They directly affect: The number of multiply-accumulate operations per output pixel, the size of the halo added to the shared memory tile, and the total number of FLOPs performed.
- **Ni**: This defines the number of input channels. Each output value is computed as a sum of convolutions over all Ni input channels, each with its own kernel slice.
- **Nn**: This defines the number of output channels. Each filter produces a separate output feature map.
- **BLOCK_X & BLOCK_Y**: These define the CUDA thread block dimensions in the X and Y directions. Each block computes a tile of the output feature map, with dimensions BLOCK_X × BLOCK_Y. The total number of threads in a block (BLOCK_X × BLOCK_Y) controls the level of parallelism and directly impacts: the shared memory tile size, how many output pixels are computed per block, and the distribution of work across the GPU. Optimizing these values is critical to balance occupancy, shared memory usage, and memory coalescing.

GPU System Parameters

Before modeling the Titan V GPU, we first needed to gather information about its key hardware parameters that would directly link our performance model to the GPU's actual behavior. To accomplish this, we cloned the CUDA-samples GitHub repository [1] and ran the deviceQuery utility in order to gather information about our exact GPU specifications. The parameters shown in Figure 6 directly convey the hardware capabilities and constraints of the Titan V. To find the normal operating GPU frequency (which was not reported in the query), we used *nvidia-smi --query-gpu=clocks.sm --format=csv* to profile the GPU and found the frequency to be 1.2 GHz, meaning this device was not overclocked.

Parameter	Value
Total amount of global memory	12050 MBytes (12635406336 bytes)
Multiprocessors	80
GPU Max Clock rate	1455 MHz (1.46 GHz)
Memory Clock rate	850 Mhz
Memory Bus Width	3072-bit
L2 Cache Size	4718592 bytes
Total amount of constant memory	65536 bytes
Total amount of shared memory per block	49152 bytes
Total shared memory per multiprocessor	98304 bytes
Warp size	32
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Max dimension size of a thread block (x,y,z)	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z)	(2147483647, 65535, 65535)

Figure 6: Key parameters for an NVIDIA Titan V GPU

For our compute model, we particularly care about the number of multiprocessors, warp size, maximum threads per multiprocessor and block, and the maximum block and grid dimensions since these factors directly affect the way our kernel data is mapped to the physical hardware and ultimately, the compute time.

Similarly, we used parameters from Figure 6 in order to factor in the Titan V's memory constraints into our memory model. The sizes of each level of the memory hierarchy were very significant since they directly influence the amount of data that the GPU can store at each level when executing our convolution kernel. However, the physical storage limits only capture the amount of data that can be stored in each level, not the temporal information of memory accesses. Since this data was not published by NVIDIA and accessible by nvprof or nvidia-smi, we turned to a research study by Citadel [2] in order to get order-of-magnitude estimates for the Titan V's memory access latencies at each level of the hierarchy. Though the authors did not directly measure the latencies for the Titan V, they tested and reported an NVIDIA V100 GPU's memory latencies, a GPU that uses the same Volta architecture as the Titan V. As a result, we decided that the latencies would be similar enough and used the measured values provided in the report (Figure 7 and Figure 8a&b) in our memory model. By synthesizing the empirical temporal data with our GPU's physical memory constraints, we hoped that our memory model would accurately approximate the Titan V's actual memory behavior, which was almost entirely hidden from us.

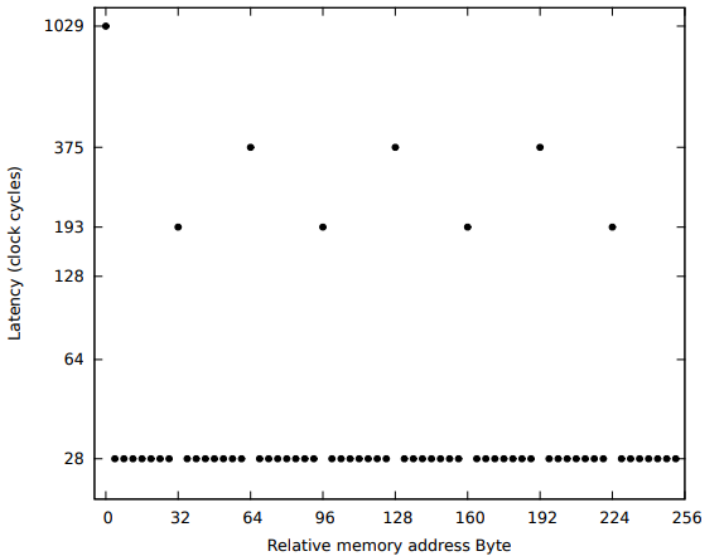

Figure 7: Memory latencies for a V100 GPU

		Volta V100 GV100	Pascal P100 GP100	Pascal P4 GP104	Maxwell M60 GM204	Kepler K80 GK210
Registers	Number of banks	2	4	4	4	4
	bank width	64 bit	32 bit	32 bit	32 bit	32 bit
L1 data	Size	32...128 KiB	24 KiB	24 KiB	24 KiB	16...48 KiB
	Line size	32 B	32 B	32 B	32 B	128 B
	Hit latency	28	82	82	82	35
	Number of sets	4	4	4	4	32 or 64*
	Load granularity	32 B	32 B	32 B	32 B	128 B
	Update granularity	128 B	128 B	128 B	128 B	128 B
	Update policy	non-LRU	LRU	LRU	LRU	non-LRU
	Physical address indexed	no	no	no	no	no
L2 data	Size	6,144 KiB	4,096 KiB	2,048 KiB	2,048 KiB	1,536 KiB
	Line size	64 B	32 B	32 B	32 B	32 B
	Hit latency	~193	~234	~216	~207	~200
	Populated by cudaMemcpy	yes	yes	yes	yes	yes
	Physical address indexed	yes	yes	yes	yes	yes
L1 constant	Broadcast latency	~27	~24	~25	~25	~30
	Cache size	2 KiB	2 KiB	2 KiB	2 KiB	2 KiB
	Line size	64 B	64 B	64 B	64 B	64 B
	Number of sets	8	8	8	8	8
	Associativity	4	4	4	4	4
L1.5 constant	Broadcast latency	~89	~96	~87	~81	~92
	Cache size	>=64 KiB	>=64 KiB	32 KiB	32 KiB	32 KiB
	Line size	256 B	256 B	256 B	256 B	256 B
L2 constant	Broadcast latency	~245	~236	~225	~221	~220
L0 instruction	Cache size	~12 KiB	-	-	-	-
L1 instruction	Cache size	128 KiB	8 KiB	8 KiB	8 KiB	8 KiB
L1.5 instruction	Cache size	-	128 KiB	32 KiB	32 KiB	32 KiB
	SMX private or shared	-	private	private	private	private
L2 instruction	Cache size	6,144 KiB	4,096 KiB	2,048 KiB	2,048 KiB	1,536 KiB
L1 TLB	Coverage	32 MiB	~32 MiB	~32 MiB	~2 MiB	~2 MiB
	Page entry	2 MiB	2 MiB	2 MiB	128 KiB	128 KiB
L2 TLB	Coverage	~8,192 MiB	~2,048 MiB	~2,048 MiB	~128 MiB	~128 MiB
	Page entry	32 MiB	32 MiB	32 MiB	2 MiB	2 MiB
L3 TLB	Coverage	-	-	-	~2,048 MiB	~2,048 MiB
	Page entry	-	-	-	2 MiB	2 MiB
Specifications	Processors per chip (P)	80	56	20	16	13
	Max graphics clock (f_g)	1,380 MHz	1,328 MHz	1,531 MHz	1,177 MHz	875 MHz
Shared memory	Size per SMX	up to 96 KiB	64 KiB	64 KiB	96 KiB	48 KiB
	Size per chip	up to 7,689 KiB	3,584 KiB	1,280 KiB	1,536 KiB	624 KiB
	Banks per processor (B_s)	32	32	32	32	32
	Bank width (w_s)	4 B	4 B	4 B	4 B	8 B
	No-conflict latency	19	24	23	23	26
	Theoretical bandwidth	13,800 GiB/s	9,519 GiB/s	3,919 GiB/s	2,410 GiB/s	2,912 GiB/s
	Measured bandwidth	12,080 GiB/s	7,763 GiB/s	3,555 GiB/s	2,122 GiB/s	2,540 GiB/s
Global memory	Memory bus	HBM2	HBM2	GDDR5	GDDR5	GDDR5
	Size	16,152 MiB	16,276 MiB	8,115 MiB	8,155 MiB	12,237 MiB
	Max clock rate (f_m)	877 MHz	715 MHz	3,003 MHz	2,505 MHz	2,505 MHz
	Theoretical bandwidth	900 GiB/s	732 GiB/s	192 GiB/s	160 GiB/s	240 GiB/s
	Measured bandwidth	750 GiB/s	510 GiB/s	162 GiB/s	127 GiB/s	191 GiB/s
	Measured/Theoretical Ratio	83.3%	69.6%	84.4%	79.3%	77.5%

Figure 8a: Measured memory access times for various GPUs

```
// -------------------- GPU constants --------------------
// ------ Hardware Compute Limits
const long NUM_CORES             = 5120;
const long NUM_SM                = 80;
const long CORES_PER_SM          = 64;
const long WARP_SIZE             = 32;
const long MAX_FLOPS             = 13800000000000;
const long MAX_INT_OPS           = 55200000000000;
const long MAX_BLOCKS_PER_SM     = 32;
const long MAX_WARPS_PER_SM      = 64;
const long MAX_THREADS_PER_SM    = 2048;
const long MAX_THREADS_PER_BLOCK = 1024;
const long MAX_THREAD_X          = 1024;
const long MAX_THREAD_Y          = 1024;
const long MAX_THREAD_Z          = 64;
const long MAX_GRID_X            = 2147483647;
const long MAX_GRID_Y            = 65535;
const long MAX_GRID_Z            = 65535;

// ------ Memory/Cache Limits
const long GLBL_MEM              = 12635406336; // Bytes
const long MEM_BUS_WIDTH         = 3072/8;      // Bytes: 384
const long L2_SIZE               = 4718592;     // Bytes
const long CONST_MEM             = 65536;       // Bytes
const long SHARED_MEM_PER_BLOCK  = 49152;       // Bytes
const long SHARED_MEM_PER_SM     = 98304;       // Bytes
const long MAX_REGISTERS_PER_SM  = 65536;
const long L1_UPDATE_GRANULARITY = 128;
const long L2_LINE_SIZE          = 64;
const long double DRAM_BW        = 750e9;       // Bytes/sec

// ------ Cycle Calculations Limits
const long double GPU_PERIOD     = (1.0 / GPU_FREQ);
const long double SHAREDMEM_HIT  = 19 * GPU_PERIOD;
const long double L1_HIT         = 28 * GPU_PERIOD;
const long double L2_HIT         = 193 * GPU_PERIOD;  // L1 miss, data in L2
const long double DRAM_TLB_HIT   = 375 * GPU_PERIOD;  // L1 and L2 miss, TLB miss data in DRAM
const long double DRAM_HIT       = 1029 * GPU_PERIOD; // L1 and L2 miss, TLB miss data in DRAM
```

Figure 8b: Parameters embedded into and used in our code

Outputs

For our performance model structure, we decided to use a roofline model that would analyze the convolution kernel's memory-bound execution time and compute-bound execution time separately. The model would then use the maximum of the two as the limiting execution time factor, and divide the total number of operations by that maximum. Though the GPU does not operate with memory and compute completely decoupled, this core model serves as a baseline for analyzing the main bottleneck of our kernel.

Calculating Tera Operations per second (TOps)

The floating-point operation count (FLOPs) and the integer operation count should be calculated separately in GPU kernel performance analysis because they represent fundamentally different types of computation. FLOPs occur during arithmetic-heavy workloads such as multiply-accumulate (MAC) operations inside convolution loops. However, integer operations originate from index calculations, loop counters, bounds checking, and memory address computation. While both contribute to total instruction throughput, their hardware execution paths differ. Thus it is necessary to independently derive equations for both types of instruction counts. Furthermore, summing them without distinguishing between them would prevent ease of determining whether a kernel is compute-bound (FLOPs-heavy) or control/memory-bound (integer-heavy), which is essential for optimization and architectural insights.

As seen from Figures 1 through 5, only Figure 4 contains the code for the sum variable's multiply-accumulate, which contributes to the total FLOP count. Because this code is only performed by threads that directly write to a given output tensor coordinate, the formula in Figure 9 is easily derived, due to each output coordinate requiring one floating point addition and one floating point multiply per iteration.

```
long flop = 2*long(Nx)*long(Ny)*long(Kx)*long(Ky)*long(Ni)*long(Nn);
```

Figure 9: FLOP formula

In the CUDA convolution kernel, integer operations refer to all arithmetic and indexing calculations involving integers, excluding floating-point computations. These include additions, multiplications, and divisions used for loop control, array indexing, block and thread ID calculations, shared memory coordinate computations, and synapse and input address generation. The formula for estimating integer operations was derived by analyzing the behavior of each thread and block during kernel execution.

The main approach that we used to calculate the INT operation count is as below:

- **Thread/Block Indexing** from Figure 2: There are two integer additions and two integer multiplications per thread
  - $Thread/Block\ Indexing = 4$
- **Input Loading Loop** from Figure 3: Over the Ni input layers, the middle loop (just below the Ni loop) executes SHMEM_TILE_Y/threads in a block's Y-dimension (BLOCK_Y) times. Similarly the inner-most loop executes SHMEM_TILE_X/BLOCK_X

times. Unravelling all three of these loops into one larger loop exposes that each iteration requires one increment (one integer addition). The total number of iterations is derived by simply multiplying the total amount size of the bounds of each nested for loop, thus yielding the following equation for Memory Load Loop Increment:

  - $LoopIncr_{inputLoad} = 1 * Ni * \frac{SHMEM\ TILE_X}{BLOCKS_X} * \frac{SHMEM\ TILE_Y}{BLOCKS_Y}$

- **Convolution Loop** from Figure 4: Over the Ni input layers, the MAC operation is performed Ni*Ky*Kx times. By similarly unrolling, there is one increment per overall iteration:
  - $LoopIncr_{conv} = 1 * Ni * Ky * Kx$
- **Input Indexing** from Figure 3: There are six integer operations per element that is read into the shared memory array.
  - $Input\ Indexing = 6 * Ni * (SHMEM\ TILE_X)(SHMEM\ TILE_Y)$
- **Synapse Indexing** from Figure 4: There are 8 integer operations per convolution MAC operation, thus yielding the following equation:
  - $Synapse\ Indexing = 8 * Ni * Ky * Kx$
- **Output Indexing** from Figure 5: There are four integer operations.
  - $Output\ Indexing = 4$

The above equations can be coalesced into one final equation per thread:

$$Per\ thread\ INT\ OP = 4 + 4 + Ni * [6 * SHMEM\ TILE_X * SHMEM\ TILE_Y + 8 * Ky * Kx + Ky * Kx + \frac{SHMEM\ TILE_X}{BLOCKS_X} * \frac{SHMEM\ TILE_Y}{BLOCKS_Y}]$$

Equation 1: per thread integer operations in the convolutional kernel

This per thread integer operation count is then multiplied by the number of threads that write to the output tensor (which is simply the number of elements in the output tensor):

$$Total\ INT\ OP = (per\ thread\ INT\ OP)(NXSCL)(NYSCL)(Nn)$$

Equation 2: Total integer operations in the convolutional kernel

```
long int_ops;
// Method 1:
long long int_ops = static_cast<long long>(NXSCL) * NYSCL * Nn * (
    8 + Ni * (
        6 * SHMEM_TILE_Y * SHMEM_TILE_X +
        9 * Ky * Kx +
        1 * (SHMEM_TILE_Y / BLOCK_Y) * (SHMEM_TILE_X / BLOCK_X)
    )
);
int_ops /= 60 * 1.69*(32/BLOCK_X)
```

Figure 10: Method 1 for calculating INT Ops

The ending division by the trailing constant was necessary to match the above theoretical

equation with the actual output. The most probable cause of this is likely the occurrence of forwarding of the integer operation results, depending on the block size.

Calculating Compute Bound Time

To calculate the compute-bound time of our kernel, we began by dissecting the architecture of the GPU and identifying how resources are allocated, divided, and how they operate on data.

Given our GPU specifications, we analyzed our kernel to identify the parameters needed to properly model our hardware constraints. GPUs use the Single-Instruction Multiple Thread (SIMT) model and are built around streaming multiprocessors (SMs), the main compute cores for a GPU. Each SM could operate on and execute multiple blocks of threads, with a grid of blocks being mapped to multiple SMs [3]. The mapping of threads to blocks, blocks to a grid, grid to SMs, and the hardware constraints regarding threads per blocks and threads per SM all are significant factors in accurately modeling the GPU's compute resources, and are hence factored into our performance model.

Our model first calculates the grid dimensions used by our convolution kernel (Figure 11). We then use the product of these dimensions to calculate the total number of blocks in our grid. We similarly calculate the threads per block in our kernel by simply taking the product of the threading dimensions (Figure 11).

```
long grid_dim_x = (Nx + BLOCK_X - 1) / BLOCK_X;
long grid_dim_y = (Ny + BLOCK_Y - 1) / BLOCK_Y;
long grid_dim_z = Nn;
long num_blocks = grid_dim_x * grid_dim_y * grid_dim_z;
long threads_per_block = BLOCK_X * BLOCK_Y;
```

Figure 11: Calculating grid dimensions, blocks, and threads per block

GPU's can assign multiple blocks to SMs, up to the thread limit of each SM, though blocks cannot be split across SMs. GPU's also assign blocks evenly across SMs before doubling up on the same SM [3]. Given these policies, we calculated the ceiling of the blocks assigned to each SM and the ceiling of the number of iterations each SM needed to execute all of the assigned blocks. Our main simplifying assumptions were that each iteration must be run sequentially and could not be overlapped due to the hardware resources available, and that each iteration took the same amount of time, regardless of whether it was at max utilization or not (i.e. not thread-count dependent). The ceiling was taken for both blocks per SM and the SM iterations, in order to be conservative yet accurate in our modeling [Figure 12].

```
blocks_per_sm = (num_blocks + active_sms - 1) / active_sms; // ignores thread max
sm_iterations = ((blocks_per_sm * threads_per_block) + MAX_THREADS_PER_SM - 1) / MAX_THREADS_PER_SM;
```

Figure 12: Ceiling functions to calculate blocks per SM and total SM iterations required

To add temporal information, we modeled the compute time using estimated throughput, as opposed to calculating the cycle time. This was done for both simplicity and

accuracy, as calculating a cycle-accurate computation time directly would be challenging with its natural coupling to memory operations, which we were purposely avoiding. As a result, we estimated computation time by appropriately scaling the max theoretical FLOP and INT Op throughput rates [4]. These were scaled using a ratio of active SMs (SMs with blocks assigned to them) to total SMs, in order to only count the SMs doing work [Figure 13]. Though using max throughput values is optimistic, we figured this would still be accurate as we did see very high SM utilization and efficiency from our profiling tests, and did not therefore see a significant need to add a heuristic efficiency/scaling factor to the throughput values.

Lastly, these throughput values were used to divide the FLOP or INT Op counts calculated earlier. Under the assumption that integer and floating point operations would require different hardware and could run in parallel (hence their different max throughputs), we took the max of both quotients, and multiplied it by the SM iteration factor to account for the amount of data required to be processed [Figure 13]. This resulting product was our computation-bound execution time estimate. This value was entirely built upon our understanding of thread and block mapping to GPU SMs, taking into account key parameters such as active SMs, data type throughput variations, and the varying grid and block dimensions in our various kernel tests.

```
flop_throughput = MAX_FLOPS * (float(active_sms) / NUM_SM);
int_throughput = MAX_INT_OPS * (float(active_sms) / NUM_SM);
comp_time = sm_iterations * max((int_ops / int_throughput), (flops / flop_throughput));
```

Figure 13: Throughput and computation time calculations

Calculating Memory Bound Time

We took a similar approach to the memory-bound execution time model, analyzing the GPU's hardware and policies alongside our kernel workload. From our profiling, we clearly saw the memory hierarchy of the Titan V: L1/Scratchpad, L2 cache, and L3/DRAM. We also noted the different clock domains in the GPU, with Scratchpad and L2 operating on the GPU clock, while the DRAM used a much slower memory clock.

Memory coalescing is also a significant factor at play. Given the 80 SMs and floating point data type, even if all SMs initiated a read or write at the same time, these could all be coalesced and fed across the DRAM bus as they would still be under the maximum bus-width, reducing stalls.

Our kernel used shared memory for fast input neuron access. Using the *__shared__* directive in CUDA meant that the GPU's on-chip SRAM would be partially allocated for each block, up to the maximum limit of 48kB per block and 96kB per SM [5]. If the input neuron size exceeded this limit, the CUDA code would run into a compilation error and not execute, so we added this same check into our performance model (though we only print a warning, and execute anyways).

On the other hand, our output neurons and synapses were not shared, and therefore get stored into DRAM when the CUDA memcpy executes. As our kernel executes and the synapses are read, there would be compulsory misses to DRAM before the data would get pulled into L2 until evicted. The output neuron writes would always be to DRAM, but this latency would be

largely hidden by the parallelism and other memory accesses.

To assist with our memory model, we created helper functions to calculate the approximate access times to each level in the Titan V's memory hierarchy. We initially estimated L1 access times to be approximately 20 cycles (GPU clock), L2 access times to be around 215 cycles (GPU clock), and DRAM access times to be around 615 cycles (GPU clock) + 125 cycles (Memory clock). These times were meant to be order of magnitude rough estimates, mainly representing the increasing access penalties. However, upon further research we discovered the Citadel report [2] which gave exact cycle time measurements for a V100 GPU as mentioned in our parameter section above. We decided that using these values would have more merit than arbitrarily choosing approximate values.

Overall Approach Explanation:

The overall approach that we took was to analyze the memory accesses and the memory reuse patterns in our kernel per section, and within or across blocks. Our critical assumption was that the GPU runs the blocks sequentially in order of block-level xy layers. We then derived the following equations, which summed together, give the final theoretical memory-bound time.

There are three main points in our code where memory operations are performed: input neuron reads from DRAM into L1/shared memory (Figure 3), input neuron reads from shared memory and synapse reads from L2 for convolution (Figure 4), and output neuron writes (Figure 5). However, the last group of memory operations (the memory writes), are written directly to DRAM without passing through L2. Hence, from these conclusions, we made the following reasonable assumptions; L1 only is composed of input neuron elements and L2 is composed of only synapse tensor elements. From here, we calculated the memory operation times for each section, across all blocks and iterations of Ni:

- **Memory read for input neurons (DRAM → L1)**:
  - $DRAM\ access\ necessary\ (misses)\ per\ block\ =\ (DRAM\ read\ time)\ *\ roundUp(SHMEM\ Tile_X\ *\ SHMEM\ Tile_Y\ *\ sizeof(float)/(Memory\ read\ in\ per\ DRAM\ read))$
  - Memory read in per DRAM read = L1 update granularity = 128 bytes
    - Source: Table 3.1, "L1 update granularity: 128 bytes", page 19
  - DRAM read time = 375 cycles
    - Source: Page 20, Figure 3.2: "L2 miss + TLB hit latency ≈ 375 cycles"
  - To estimate the time it takes to load a shared memory tile from DRAM for a single input channel (Ni) and a single block, we use a model based on the transaction granularity and access latency of the memory hierarchy. Although the threads in the kernel cooperatively load a contiguous block of data into shared memory and the accesses are aligned and coalesced, the GPU memory system still fetches data from DRAM in fixed-size transactions. The L1 update granularity is 128 bytes, which means that any DRAM read into L1, and then subsequently into shared memory, pulls in at least 128 bytes, regardless of the exact request size. This defines the unit of effective DRAM access granularity. While the Volta memory interface theoretically supports up to 16 KB of data transfer per burst across its 512-bit (64-byte × 8 channel) memory bus and burst length of 32, this full capacity is only achieved in streaming scenarios or very

large, aligned memory transactions. In this case, since the shared tile is loaded piecewise within a loop over tile coordinates, the GPU issues many smaller loads rather than a single DMA-style transfer. As a result, the correct model for DRAM accesses is to divide the total tile size (in bytes) by the 128-byte L1 update granularity and multiply the number of resulting transactions by the average DRAM latency. The average DRAM read latency is around 375 cycles for an L1 miss with an L2 miss and TLB hit, which is a realistic non-worst-case estimate.

- Thus, across all the blocks and across Ni channels, the following equation is derived (noting that each block reads in a distinct part of the input neuron array:

$$Shared\ Mem\ Read\ Time_{input\ neurons} = (per\ block)(block\ Dim_x)(block\ Dim_y)(Ni)(block\ Dim_z) / (BLOCKS\_PROCESSED\_AT\_A\_TIME)$$

- **Compute related memory reads**:
  - Input Neurons reads:
    - $per\ thread\ input\ neuron\ read\ time = Ky \times Kx \times Ni \times shared\_mem\_read\_latency$
      - shared_memory_read_latency = 19; → cycles per access under no contention (Table 3.1)
      - The above equation simply multiplies the amount of time necessary to read an element from shared memory with the total number reads done to shared memory for a given thread
    - $total\ input\ neuron\ read\ time = (Nx * Ny * Nn * per\ thread) / (BLOCKS\_PROCESSED\_AT\_A\_TIME * THREADS\_PER\_BLOCK)$
      - The per thread equation is then multiplied by the number of output neurons (only threads that write to an output neuron perform these memory accesses)
  - Synapse tensor reads:
    - $weights\ bytes = Ky * Kx * sizeof(float)$
      - Given any block, the iterators that iterate over Ni, Kx, and Ky are completely independent of the block or thread index. However, n, which is the output layer any thread works on, is dependent on its block's z-dimension. Hence, for any given xy layer of blocks (and thus x*y*block.x*block.y threads), as seen in Figure 4, the exact same synapse chunk of Kx*Ky*Ni values will be accessed. Only across different z-layers of blocks are different synapse chunks accessed. Furthermore, the memory layout of the 4D synapse tensor was strategically chosen such that all chunks are contiguously laid out in memory.
    - $l2\ lines = roundUp(weights\ bytes\ /\ 64)$
      - This describes the amount of times that reads to DRAM are necessary. All the synapse data originally resides on DRAM.
    - $latency\ per\ line = L2\ miss + DRAM\ latency\ (with\ TLB\ hit) = 193 + 375$
    - $total\ unique\ layers = Ni * Nn$
    - $Nn\ active\ at\ once = roundDown(BLOCKS\ AT\ A\ TIME / (block\ Dim_X * blockDim_Y))$
    - $total\ synapse\ fetch\ time\ to\ L2 = (l2\ lines)(latency\ per\ line)(total\ unique\ layers)$

$/\ max(1,\ Nn\ active\ at\ once)$

- $Synapse\ read\ time\ =\ [(Nx)(Ny)(Ni)(Nn)\ -\ (l2\ lines)(total\ unique\ layers)]\ *\ (L2\ read\ time)$
  - L2 read time = 193 cycles

- **Output Writes**
  - $Output\ write\ time\ =\ (Nx)(Ny)(Nn)(DRAM\ write\ time)$
  - The above can be IGNORED as it is hidden.
- **Solving for Active Blocks Processed at a Time**
  - The number of concurrently active thread blocks on a GPU is restricted by the resource constraints of each Streaming Multiprocessor (SM). Each SM has a fixed amount of computational resources, such as registers, shared memory, the number of threads it can support, and the maximum number of warps and blocks it can hold.
  - To determine the total number of active blocks on the GPU at any point in time, we first calculated how many blocks fit on a single SM. This involved calculating how many blocks can fit within each constraint, and then taking the minimum of those values. We considered the architectural upper bound on blocks per SM, which may be stricter than the resource-derived limits.
  - Once we determined the per-SM block capacity, we scaled that value by the total number of SMs on the GPU to compute the total number of blocks that can be active concurrently across the entire device. This final output number reflects the degree of hardware-level parallelism available for the given kernel.

The total memory cycles required by all the above steps are accounted for by adding the highlighted equations together. The parameters in our code utilize time (cycle count * cycle period) instead of just cycle count, thus not necessitating a final conversion in our code. The code for the above equations is featured below:

```
// --- Memory Read for Input Neurons (DRAM -> L1)
// Method 1:
long double MR_DRAM_access_per_block = DRAM_TLB_HIT*ceil((SHMEM_TILE_X*SHMEM_TILE_Y*sizeof(float))/
(L1_UPDATE_GRANULARITY));
long double MR_shared_mem_read_time = MR_DRAM_access_per_block  * grid_dim_x * grid_dim_y * grid_dim_z * Ni /
blocks_at_a_time;
```

Figure 14: Calculation for Memory read for input neurons (DRAM → L1)

```
// Synapse Reads
// Method 1:
long long weights_bytes = Ky*Kx*sizeof(float);
long double l2_lines = max(1.0,ceil(weights_bytes/64));
long double latency_per_line = L2_HIT + DRAM_TLB_HIT;
long total_unique_layers = Ni*Nn;
long Nn_active_at_once = floor(blocks_at_a_time / (grid_dim_x * grid_dim_y));
long double synapse_read_time = l2_lines * latency_per_line * total_unique_layers / max(1L, Nn_active_at_once);
synapse_read_time += (Nx*Ny*Ni*Nn - l2_lines*total_unique_layers) * L2_HIT / (blocks_at_a_time*threads_per_block);
```

Figure 15: Calculation for Compute related memory reads

```
// Calculate Active Number of Blocks
// Derived values
long shared_mem_per_block_bytes = SHMEM_TILE_X * SHMEM_TILE_Y * sizeof(float);
long warps_per_block = ceil(threads_per_block / 32.0);

// Per-SM resource limits
long max_blocks_by_threads = MAX_THREADS_PER_SM / threads_per_block;
long max_blocks_by_shared_mem = SHARED_MEM_PER_SM / shared_mem_per_block_bytes;
long max_blocks_by_warps = MAX_WARPS_PER_SM / warps_per_block;

blocks_per_sm = min({max_blocks_by_threads,
                     max_blocks_by_shared_mem,
                     max_blocks_by_warps,
                     MAX_BLOCKS_PER_SM});

long blocks_at_a_time = blocks_per_sm * NUM_SM;
```

Figure 16: Calculation for solving for active blocks processed at a time

However after further testing, we realized that this was unfortunately inaccurate and thus then derived the following method as a replacement for solving for "MR_shared_mem_read_time" as seen in Figure 14 and "synapse_read_time" in Figure 15. The most probable issue with the above implementation is improperly calculating or factoring in the number of active blocks, and naively assuming that they can all share the bus bandwidths fully and evenly, thus resulting in times much faster than expected and actually result.

```
long double input_bytes_total =
    grid_dim_x * grid_dim_y * grid_dim_z * Ni * SHMEM_TILE_Y * SHMEM_TILE_X * sizeof(float);

long double MR_shared_mem_read_time = input_bytes_total / DRAM_BW; // in seconds
```

Figure 17: Bandwidth-limited calculation for shared memory loading

This section first calculates the total number of bytes transferred from DRAM into shared memory to preload the input neurons used by each thread block. It assumes that every block loads a shared memory tile of dimensions SHMEM_TILE_Y × SHMEM_TILE_X for each input channel Ni. This process occurs across all thread blocks in the grid, corresponding to the spatial tile indices and output channels (Nn). The total number of bytes read is computed by multiplying the number of tiles with the tile size and the size of each float element. Once the total data volume is known, the memory read time is estimated by dividing this volume by the measured DRAM bandwidth, DRAM_BW. This models the time taken to transfer data under a bandwidth-bound assumption, which reflects the high-throughput, coalesced access pattern typically seen in GPU shared memory loading.

```
// Method 2: bandwidth-limited
long double bytes_per_thread = Ky * Kx * Ni * sizeof(float);
long double total_threads = NXSCL * NYSCL * Nn;
long double total_synapse_bytes = bytes_per_thread * total_threads;
long double synapse_read_time = total_synapse_bytes / DRAM_BW;
```

Figure 16: Bandwidth-limited calculation for loading synapse elements

This final section models the time required for each thread to read synapse weights from DRAM during the convolution operation. Each thread processes one output neuron and requires Ky × Kx × Ni weights, where Ni is the number of input channels, and each weight is a float. By multiplying the number of threads with the number of bytes each thread must read, the code calculates the total volume of synapse data accessed. The memory read time is then estimated by dividing this total byte count by the available DRAM bandwidth, modeling a situation where synapse weights are streamed from global memory with little cache reuse.

Calculating Achieved Operation Intensity

Operation intensity (operations per byte) is a metric used to measure the level of memory reuse we get at each stage of the memory hierarchy. Due to the kernel being highly regular and the GPU data movement being approximated as mentioned above, we determined the following:

- **L1/Scratchpad**: the only data we explicitly write to shared memory are our input neurons. These values originate from the CPU side and are randomly generated before being copied over to GPU DRAM and ultimately transferred over to the L1/Scratchpad shared memory. Since all of the input neurons are eventually stored in shared memory, and we don't use shared memory for any other data, the total number of bytes at this level is equal to the total number of bytes of input neurons we have.
- **L2 Cache**: similar to the previous case, the synapses are initially randomized on the CPU side then transferred over to the GPU's DRAM. These values are then all eventually pulled to the L2 cache at some point in the kernel execution. Since no other data gets loaded into this memory level, the total number of bytes at this level is equal to the total number of bytes of synapse weights we have.
- **L3/DRAM**: the input neurons, synapses, and output neurons are all initially copied over to the GPU's DRAM. However, the input neurons are copied over to shared memory and never reused directly from DRAM. The synapses are pulled from DRAM and often stored and accessed from the L2 cache, and the output neurons are only ever written directly to DRAM. Therefore, the number of bytes at this level is equal to the sum of the number of synapse bytes and the number of output neuron bytes, since only the synapse and output neuron data are stored or used in DRAM, as opposed to the input neurons which primarily reside in the shared memory.

We use the above observations to calculate the operational intensity at each level of the memory hierarchy, signifying how much work is done per byte of data and how efficient our kernel is regarding compute and memory resources (Figure 14).

```
// Calculate operational intensity (same whether comp or mem bound)
outputs.L1_OpIntensity = ops / neuron_i_size;
outputs.L2_OpIntensity = ops / synapse_size;
outputs.DRAM_OpIntensity = ops / (synapse_size + neuron_n_size);
```

Figure 14: Calculations for operational intensity

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

Performance Model Results

In order to analyze the effectiveness of our performance model across various parameters, we set default parameters for each test, then varied one parameter at a time to analyze our model's accuracy. The default parameters used were all set to relatively small and regular dimensions in order to just stress one dimension at a time. Our default parameters were as follows: 32x32 input neuron size, 3x3 kernel size, 32 input and output channels, and 32x32 warp/block size. Our four key tests were varying block dimensions, channel dimensions, kernel dimensions, and size dimensions. The varied parameter values and results of our tests are shown in table form in Figures 15-18 and the error rates are shown in Figures 19-22.

		Block Dimension			
		4	8	16	32
Actual	FLOPs	18,907,136	18,907,136	18,907,136	18,907,136
	INT OPs	41,025,536	32,702,464	28,983,296	27,234,304
	Total OPs	59,932,672	51,609,600	47,890,432	46,141,440
	Exec. Time (ms)	0.050335	0.049631	0.05136	0.06464
	TOPs	1.191	1.040	0.932	0.714
Predicted	FLOPs	18,874,368	18,874,368	18,874,368	18,874,368
	INT OPs	385,524	1,763,784	10,476,711	72,575,626
	Total OPs	19,259,892	20,638,152	29,351,079	91,449,994
	Comp. Time (ms)	0.001	0.001	0.001	0.003
	Mem. Time (ms)	0.067	0.060	0.058	0.058
	Exec. Time (ms)	0.068	0.061	0.060	0.061
	TOPs	0.28935	0.344074	0.503263	1.58886
	L1 Op. Intensity	122	131	187	583
	L2 Op. Intensity	522	559	796	2480
	DRAM Op. Intensity	114	122	174	544
Error	FLOPs	0.17%	0.17%	0.17%	0.17%
	INT OPs	99.06%	94.61%	63.85%	166.49%
	Total OPs	67.86%	60.01%	38.71%	98.19%
	Exec. Time (ms)	34.96%	23.61%	16.22%	5.67%
	TOPs	75.70%	66.91%	46.03%	122.58%

Figure 15: Block dimension test results

		Channel Dimension								
		2	4	8	16	32	64	128	256	512
Actual	FLOPs	74,753	299,008	1,187,840	4,734,976	18,907,136	75,563,008	302,120,960	1,208,221,696	4,832,362,496
	INT OPs	89,744	468,544	1,775,872	6,906,880	27,234,304	108,150,784	431,030,272	1,720,975,360	6,877,609,984
	Total OPs	164,497	767,552	2,963,712	11,641,856	46,141,440	183,713,792	733,151,232	2,929,197,056	11,709,972,480
	Exec. Time (ms)	0.006015	0.009568	0.017856	0.03312	0.064384	0.12886	0.26726	1.0613	9.9316
	TOPs	0.027	0.080	0.166	0.352	0.717	1.426	2.743	2.760	1.179
Predicted	FLOPs	73,728	294,912	1,179,648	4,718,592	18,874,368	75,497,472	301,989,888	1,207,959,552	4,831,838,208
	INT OPs	283,650	1,134,276	4,536,461	18,144,552	72,575,626	290,297,334	1,161,178,998	4,644,695,312	18,578,739,886
	Total OPs	357,378	1,429,188	5,716,109	22,863,144	91,449,994	365,794,806	1,463,168,886	5,852,654,864	23,410,578,094
	Comp. Time (ms)	0.000	0.000	0.001	0.002	0.003	0.007	0.022	0.175	1.401
	Mem. Time (ms)	0.000	0.001	0.004	0.014	0.058	0.230	0.921	3.684	14.735
	Exec. Time (ms)	0.000	0.001	0.004	0.016	0.061	0.237	0.943	3.859	16.135
	TOPs	1.58953	1.58917	1.589	1.58891	1.58886	1.58884	1.58883	1.58882	1.58882
	L1 Op. Intensity	36	72	145	291	583	1166	2332	4665	9331
	L2 Op. Intensity	2481	2481	2480	2480	2480	2480	2480	2480	2480
	DRAM Op. Intensity	42	84	162	305	544	893	1313	1717	2029
Error	FLOPs	1.37%	1.37%	0.69%	0.35%	0.17%	0.09%	0.04%	0.02%	0.01%
	INT OPs	216.07%	142.09%	155.45%	162.70%	166.49%	168.42%	169.40%	169.89%	170.13%
	Total OPs	117.26%	86.20%	92.87%	96.39%	98.19%	99.11%	99.57%	99.80%	99.92%
	Exec. Time (ms)	92.70%	86.14%	75.07%	51.39%	5.29%	83.97%	252.76%	263.58%	62.46%
	TOPs	5712.28%	1881.00%	857.35%	352.03%	121.70%	11.44%	42.08%	42.43%	34.75%

Figure 16: Channel dimension test results

		Kernel Dimension				
		3	5	7	9	11
Actual	FLOPs	18907136	52461568	102793216	169902080	253788160
	INT OPs	27234304	26574848	35553280	60358656	75792384
	Total OPs	46,141,440	79,036,416	138,346,496	230,260,736	329,580,544
	Exec. Time (ms)	0.065631	0.098847	0.14477	0.20624	0.27897
	TOPs	0.703	0.800	0.956	1.116	1.181
Predicted	FLOPs	18,874,368	52,428,800	102,760,448	169,869,312	253,755,392
	INT OPs	72,575,626	82,751,156	94,167,605	106,824,972	120,723,258
	Total OPs	91,449,994	135,179,956	196,928,053	276,694,284	374,478,650
	Comp. Time (ms)	0.003	0.009	0.019	0.031	0.046
	Mem. Time (ms)	0.058	0.149	0.287	0.470	0.699
	Exec. Time (ms)	0.061	0.159	0.305	0.501	0.745
	TOPs	1.58886	0.90469	0.686449	0.588798	0.53606
	L1 Op. Intensity	583	771	1011	1285	1582
	L2 Op. Intensity	2480	1320	981	833	755
	DRAM Op. Intensity	544	578	593	597	597
Error	FLOPs	0.17%	0.06%	0.03%	0.02%	0.01%
	INT OPs	166.49%	211.39%	164.86%	76.98%	59.28%
	Total OPs	98.19%	71.04%	42.34%	20.17%	13.62%
	Exec. Time (ms)	7.09%	60.77%	111.02%	142.78%	166.89%
	TOPs	126.00%	13.15%	28.17%	47.26%	54.63%

Figure 17: Kernel dimension test results

		Size Dimensions				
		14	28	56	112	256
Actual	FLOPs	3,618,944	14,475,776	57,903,104	231,612,416	1,210,056,704
	INT OPs	23,620,992	26,171,904	104,561,664	417,997,824	1,742,995,456
	Total OPs	27,239,936	40,647,680	162,464,768	649,610,240	2,953,052,160
	Exec. Time (ms)	0.044095	0.052447	0.072768	0.24189	0.98479
	TOPs	0.618	0.775	2.233	2.686	2.999
Predicted	FLOPs	3,612,672	14,450,688	57,802,752	231,211,008	1,207,959,552
	INT OPs	13,891,428	55,565,713	222,262,855	889,051,422	4,644,840,086
	Total OPs	17,504,100	70,016,401	280,065,607	1,120,262,430	5,852,799,638
	Comp. Time (ms)	0.001	0.003	0.004	0.067	1.138
	Mem. Time (ms)	0.016	0.046	0.182	0.729	3.684
	Exec. Time (ms)	0.017	0.048	0.186	0.796	4.822
	TOPs	1.08575	1.53724	1.53724	1.53724	1.58886
	L1 Op. Intensity	473	569	628	661	681
	L2 Op. Intensity	474	1899	7597	30389	158767
	DRAM Op. Intensity	282	510	639	682	694
Error	FLOPs	0.17%	0.17%	0.17%	0.17%	0.17%
	INT OPs	41.19%	112.31%	112.57%	112.69%	166.49%
	Total OPs	35.74%	72.25%	72.39%	72.45%	98.19%
	Exec. Time (ms)	61.95%	8.16%	156.12%	228.98%	389.60%
	TOPs	75.76%	98.35%	31.15%	42.76%	47.01%

Figure 18: Size dimension test results

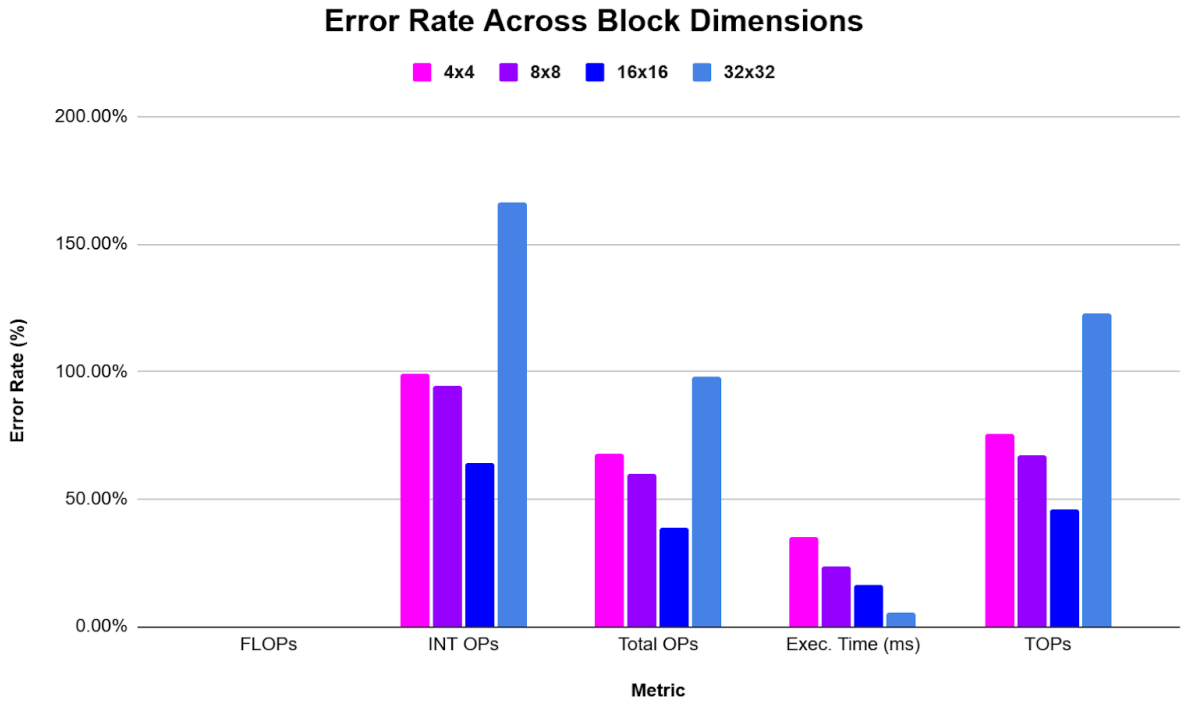

Figure 19: Block dimension test results

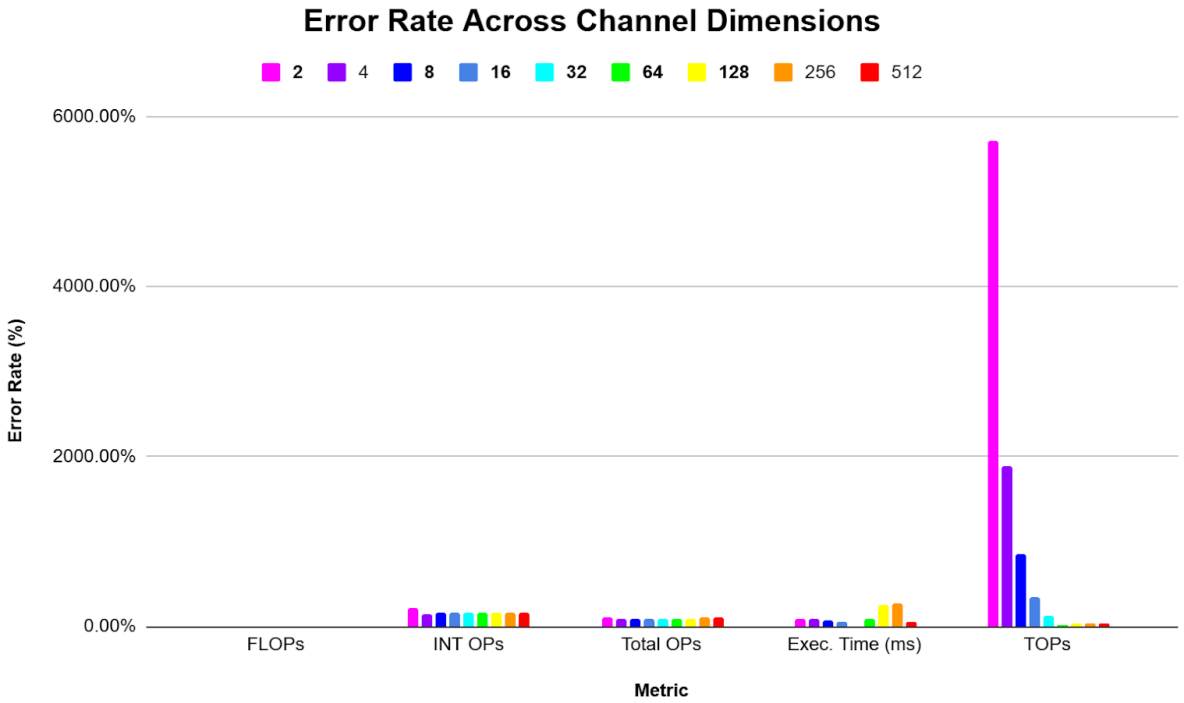

Figure 20: Channel dimension test results

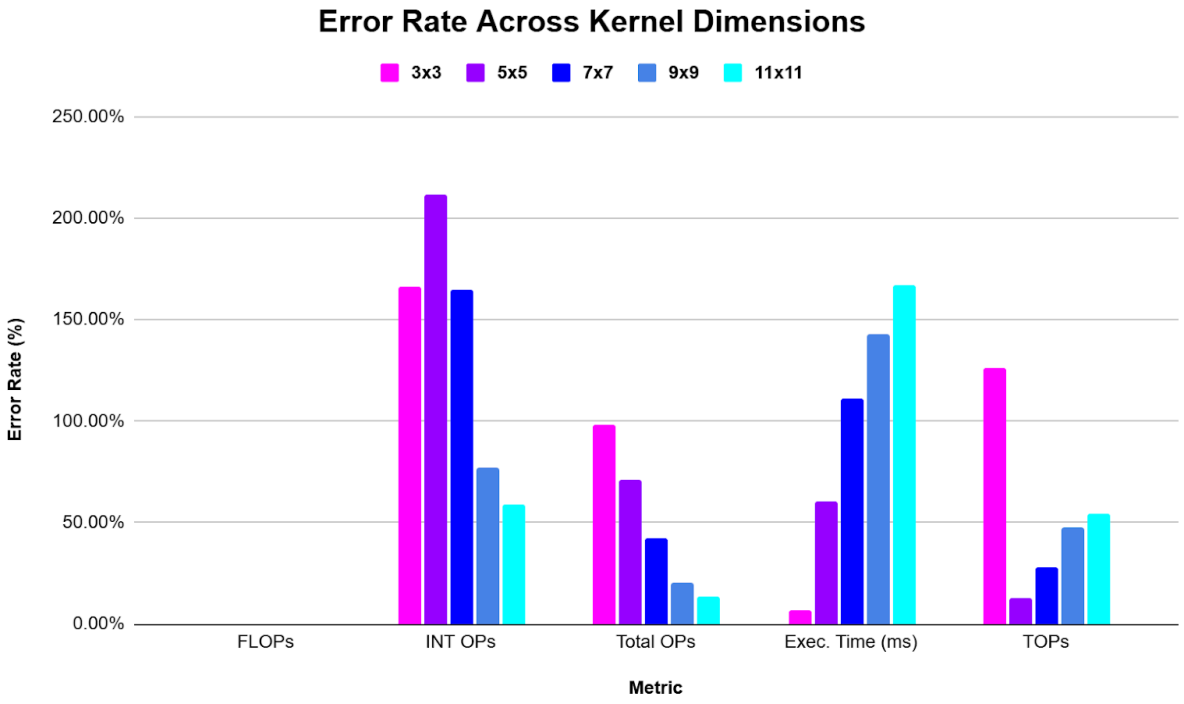

Figure 21: Kernel dimension test results

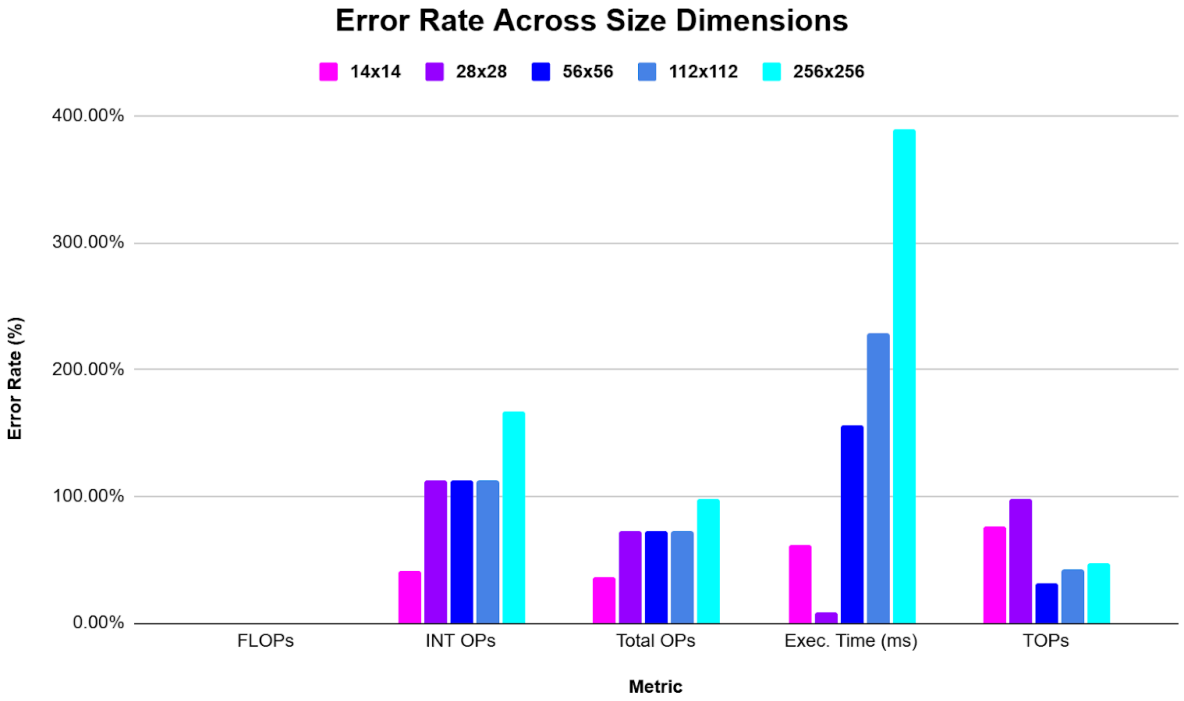

Figure 22: Size dimension test results

When varying thread block dimensions, our model's accuracy across the board generally scaled with thread size, though its error spiked at the max size of 32x32. This is likely because 32x32 is the max block size which maxes out the compute resources of the GPU, leading to compute-bound throttling as seen with the longer than expected execution times compared to our model in Figure 15.

Across channel dimensions, all accuracies were on the same order of magnitude except for the TOPs calculation, which was drastically off for very small channel sizes before quickly returning to normal error rates. This significant accuracy drop is likely due to non-linear scaling of execution time due to overheads we did not account for. It makes logical sense that our predicted near-zero execution time for super small channel dimensions (Figure 16) is unrealistic, and there is some constant overhead that we did not account for that resulted in our overly optimistic TOPs calculations for small channel dimensions.

Our model performed decently when varying the kernel dimensions, with all metrics having a relatively similar error rate. Our OP counts got more accurate as the dimensions grew larger, while our execution time estimates saw the opposite trend. These trends are due to kernel size dependent miscalculations in our integer operation count code, influencing our total operations and therefore our execution time and TOPs. We also saw an error spike for TOPs at the smallest kernel dimension tested, however in this case the culprit is likely the (accurate) low execution time being significantly exacerbated by the high integer operation inaccuracy (Figure 17).

When varying input neuron size dimensions, our results were somewhat inaccurate, yet the culprit appeared to be a combination of integer operation inaccuracy again, along with very inaccurate execution time estimates as size increased. Our model seemed to be drastically overestimating the execution time necessary for large sizes, specifically in the memory portion (Figure 18). Though both our compute and memory bound times overestimate the actual execution time, the memory time is nearly 4x the actual execution time. As opposed to a drastic misinterpretation of either memory or compute behavior, what could be happening is a snowballing of error in this case. For very large size dimensions, the data being processed increases quadratically, meaning over many computations and iterations, small errors in our memory latency functions or compute loops would snowball and result in large discrepancies over time.

Overall, it is worth noting that the FLOP error rate was essentially zero for every single test case, showing that our model counted the FLOPs extremely accurately. This was less true for integer operation counts, which used a much more complex formula and as a result, was prone to much more error. This integer error rate was then reflected in the total operations for every case, snowballing into our execution time and TOPs estimates. Our computation time and memory time estimates were generally quite accurate however, barring very specific cases as explained above.

The main strength of our model appears to lay in our simpler sections of code that simply utilize the known hardware parameters of the GPU, whereas our more complex sections of code with significant assumptions and heuristic approaches fall short. In general, our model seems to handle a decent amount of cases with sufficient accuracy, and can likely be improved by adding more checks for extreme cases, such as adding overhead to cover small cases, and testing for more precise constants to handle larger cases where errors can snowball.

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

Architecture Insight

Based on your model, which hardware parameters should future GPUs change if they want to perform better on your implementation of convolution? To answer this question, perform a sensitivity analysis on some parameters.

We found that certain hardware parameters play a more critical role in determining convolution kernel efficiency than others. The most significant takeaway is that memory performance, not just raw compute power, is the dominant factor in execution time, especially as input sizes and channel counts increase. Hence, optimizing memory bandwidth and memory hierarchy latency should be a key focus for future GPU architectures aimed at accelerating convolution workloads.

Our memory model revealed two major bottlenecks in our convolution kernel's performance, both caused by memory hierarchy limitations. The first arises from the initial loading of input neuron tiles from DRAM into shared memory. Although shared memory access is extremely fast, the latency involved in fetching the data from DRAM to L1, before it can be staged in shared memory, causes significant delays. This step is primarily constrained by DRAM bandwidth and the L1 update granularity, which is fixed at 128 bytes. Increasing the DRAM bandwidth or reducing this granularity would allow more efficient data movement and better utilization of shared memory, particularly in kernels like ours that rely on frequent and structured shared memory reuse.

However, an even larger bottleneck in our kernel arises from reading synapse weights directly from DRAM and the L2 cache during the convolution loop. Unlike input neurons, synapse values are not staged in shared memory due to their size and per-thread variability. As a result, each MAC operation triggers multiple global memory reads for the corresponding weights, leading to a high volume of high latency memory loads. Even when the data hits in L2, the access latency is significantly higher than shared memory, and DRAM accesses take much longer. Given the large number of MAC operations performed across all output neurons and channels, synapse reads dominate the overall memory-bound execution time.

To address this, future GPU architectures targeting convolution-heavy workloads should focus on reducing the latency of synapse accesses. Potential improvements include enlarging L2 cache size and associativity, or creating dedicated on-chip weight buffers that streamline in the weights (similar to preloading). While increasing DRAM bandwidth benefits both input and synapse reads, the non-reusable nature of synapse weights across threads makes it even more critical to reduce access latency through architectural enhancements.

Another key insight relates to shared memory capacity per SM. The current limit of 96 KB per SM restricts how large a shared memory tile can be. For kernels like ours that rely on tiling input neurons into shared memory to enable reuse, larger shared memory would allow us to process bigger tiles at once, thus reducing DRAM accesses and improving locality. Expanding

per-SM shared memory would relax our current tiling constraints and further enable architectural optimizations.

In terms of compute resources, we noted that increased thread concurrency per SM and more SIMD (warp) lanes could accelerate the MAC-intensive convolution operations, particularly when the kernel is compute-bound. For instance, increasing the number of threads per SM beyond the current 2048 limit could allow more blocks to be scheduled concurrently, enhancing throughput and improving pipeline utilization. However, our tests showed that compute-bound limitations only appeared in corner cases, thus indicating that memory bandwidth is the largest bottleneck.

Lastly, we recommend that future GPUs improve L2 cache latency or provide higher associativity. In our kernel, synapse weights are frequently reused within the same block layer (z-dimension), so having lower L2 access latencies (currently modeled at 193 cycles) would reduce the total synapse fetch time. Since our memory access patterns are highly regular, a better L2 reuse mechanism, such as manual cache hints or explicit software-managed caching policies, could significantly enhance performance.

In summary, if future GPUs aim to optimize convolution performance for workloads like ours, they should prioritize (1) increasing DRAM bandwidth, (2) expanding shared memory size, and (3) improving L2 cache latency and reuse mechanisms. These changes target the dominant bottlenecks identified in our model and would allow for more efficient execution across a wide range of convolutional kernel configurations.

Works Cited

[1] "CUDA Samples." GitHub, 23 Sept. 2023, github.com/nvidia/cuda-samples. Accessed 19 May 2025.

[2] Jia, Zhe, et al. *Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking*. Citadel Enterprise Americas LLC, 18 Apr. 2018.

[3] "GPU Glossary." *Modal*, 2025, modal.com/gpu-glossary. Accessed 19 May 2025.

[4] Oh, Nate. "The NVIDIA Titan v Deep Learning Deep Dive: It's All about the Tensor Cores." *Www.anandtech.com*, www.anandtech.com/show/12673/titan-v-deep-learning-deep-dive. Accessed 19 May 2025.

[5] Harris, Mark. "Using Shared Memory in CUDA C/C++." *NVIDIA Developer Blog*, 29 Jan. 2013, developer.nvidia.com/blog/using-shared-memory-cuda-cc/. Accessed 19 May 2025.