Investigation of CUDA Optimizations for Accelerating a Toy CNN Architecture

Holden Grissett[a]

[a]*UCLA*

Professor Jason Cong

1. Parallelization Strategy

We map the two spatial dimensions of the pooled output (112x112) onto blockDim.x and blockDim.y and the 256 output channels onto gridDim.z. Within each block, every thread cooperatively loads a small tile of the input feature map into shared memory—with each thread taking multiple elements to amortize the global-load cost—then performs the 5x5 convolution across all 256 input channels, accumulating four pooled outputs in registers (for the 2x2 MaxPool). By fusing convolution, ReLU, and MaxPool in one kernel we eliminate extra passes over the data; by tiling into shared memory we reuse each input element many times (once per filter row/column and per pooling location) rather than reloading it from DRAM; and by assigning one output channel per thread along blockDim.z we achieve fine-grain parallelism across channels. This scheme suits the T4's 640 GB/s of memory bandwidth vs 2.6 TFLOP/s compute balance, because it maximizes arithmetic intensity and minimizes global-memory traffic.

2. Applied Optimizations

Starting from a naïve fused kernel, our first major win came from shared-memory tiling: loading overlapping input windows into a block-local buffer so each of the 25 convolution weights per input channel sees the same data in fast on-chip memory, boosting performance from 350 GFlops to 1580 GFlops. Next, we unrolled the 2×2 MaxPool in-kernel, turning four separate reductions into straight-line code and saving loop overhead (another 100 GFlops). We then added const qualifiers on the small inline index helpers to help the compiler hoist and precompute address arithmetic. Finally, we experimented with "#pragma unroll" on the 5×5 loops to see basically no extra gain. In total these steps took us from 350 GFlops (baseline) to 1684 GFlops on a static 16x16xZ launch.

3. Dimensions and T4 Specs

Our sweep showed that (blockDim.x, blockDim.y, blockDim.z) = (16,8,4) with (gridDim.x, gridDim.y, gridDim.z) = (7,14,96) yielded the fastest measured throughput of 1692 GFlops. A T4 GPU has 40 streaming multiprocessors, each with 64 FP32 CUDA cores (total = 2560 cores) and can support up to 2048 resident threads per SM. A 16x8x4 block is 512 threads, so each SM can hold up to four such blocks (4×512 = 2048)—perfectly saturating its maximal warp capacity. In practice we see 86% of the purely static-tile's peak (1486 GFlops) because our dynamic-tile code incurs a few more integer multiplies and slightly more bank-conflict risk, but the chosen block shape still aligns exactly with the SM's thread-capacity. If one insisted on absolute top speed one could revert to a 16×16 static tile (1024 threads) and bake in #define constants, but for flexibility and near-peak performance this (16,8,4) tuning is fully supported by the T4's hardware limits.

Table 1. Final Best Configuration Performance

Configuration	Time (s)	Throughput (GFlops)
block=(16,8,4), grid=(7,14,96)	0.097117	1692.98

4. Optimizations Performed

I investigated a few major techniques to improve the performance of the CNN kernel. The first was simply fusing all the operations into a single kernel, which provided a speedup over the sequential operation. The first, and most impactful optimization I made was to reduce global memory accesses by using a shared memory tile for multiple kernels to use together within the same SM. This provided a huge boost. The second most impactful optimization was unrolling the MaxPool operation, which gave us a 6.5% boost in speed. The last impactful optimization was adding const to the helper functions, which probably helps by giving hints to the compiler that it can optimize more aggressively, which gave us a modest 5% boost.

```
#define SHARED_MEM_TILE_HEIGHT ((BLOCK_DIM_Y - 1) * kPoolStride + kPoolSize + kKernelSize - 1)
#define SHARED_MEM_TILE_WIDTH  ((BLOCK_DIM_X - 1) * kPoolStride + kPoolSize + kKernelSize - 1)
```

Figure 1. Macros for shared memory tile size

```
int num_threads_loading_2d = BLOCK_DIM_X * BLOCK_DIM_Y;
int elements_to_load = SHARED_MEM_TILE_HEIGHT * SHARED_MEM_TILE_WIDTH;
int loads_per_thread = (elements_to_load + num_threads_loading_2d - 1) / num_threads_loading_2d;

for (int i = 0; i < loads_per_thread; ++i) {
    int flat_s_mem_idx = (threadIdx.y * BLOCK_DIM_X + threadIdx.x) + i * num_threads_loading_2d;
    if (flat_s_mem_idx < elements_to_load) {
        int s_h = flat_s_mem_idx / SHARED_MEM_TILE_WIDTH;
        int s_w = flat_s_mem_idx % SHARED_MEM_TILE_WIDTH;

        int g_h = input_global_base_h + s_h;
        int g_w = input_global_base_w + s_w;

        if (g_h < kInputHeight && g_w < kInputWidth) {
            s_input[s_h][s_w] = input[feature_map_idx(in_c, g_h, g_w, kInputHeight, kInputWidth)];
        } else {
            s_input[s_h][s_w] = 0.0f;
        }
    }
}
__syncthreads();
```

Figure 2. Code utilizing shared memory tiles

```
int local_conv_h_0, local_conv_h_1;
int local_conv_w_0, local_conv_w_1;

local_conv_h_0 = threadIdx.y * kPoolStride + 0;
local_conv_w_0 = threadIdx.x * kPoolStride + 0;
for (int kh = 0; kh < kKernelSize; ++kh) {
    for (int kw = 0; kw < kKernelSize; ++kw) {
        conv_sum_00 += weight[weight_idx(channel_idx, in_c, kh, kw)] *
                       s_input[local_conv_h_0 + kh][local_conv_w_0 + kw];
    }
}

local_conv_w_1 = threadIdx.x * kPoolStride + 1;
for (int kh = 0; kh < kKernelSize; ++kh) {
    for (int kw = 0; kw < kKernelSize; ++kw) {
        conv_sum_01 += weight[weight_idx(channel_idx, in_c, kh, kw)] *
                       s_input[local_conv_h_0 + kh][local_conv_w_1 + kw];
    }
}

local_conv_h_1 = threadIdx.y * kPoolStride + 1;
for (int kh = 0; kh < kKernelSize; ++kh) {
    for (int kw = 0; kw < kKernelSize; ++kw) {
        conv_sum_10 += weight[weight_idx(channel_idx, in_c, kh, kw)] *
                       s_input[local_conv_h_1 + kh][local_conv_w_0 + kw];
    }
}

for (int kh = 0; kh < kKernelSize; ++kh) {
    for (int kw = 0; kw < kKernelSize; ++kw) {
        conv_sum_11 += weight[weight_idx(channel_idx, in_c, kh, kw)] *
                       s_input[local_conv_h_1 + kh][local_conv_w_1 + kw];
    }
}
```

Figure 3. Max Pool Unrolling

```
relu_val = fmaxf(0.f, conv_sum_00);
max_pool_val = fmaxf(max_pool_val, relu_val);

relu_val = fmaxf(0.f, conv_sum_01);
max_pool_val = fmaxf(max_pool_val, relu_val);

relu_val = fmaxf(0.f, conv_sum_10);
max_pool_val = fmaxf(max_pool_val, relu_val);

relu_val = fmaxf(0.f, conv_sum_11);
max_pool_val = fmaxf(max_pool_val, relu_val);
```

Figure 4. Max Pool Unrolling pt. 2

Table 2. Execution time and throughput of successive CNN GPU kernels

CNN version	Time (s)	Perf. (GFlops)
GPU kernel fusion (baseline)	0.468	351.33
Shared-memory tiling (V1)	0.104	1577.44
V1 + MaxPool unrolling (V2)	0.098	1679.26
V2 + `const` helpers (V3)	0.0976	1684.91
V3 + `#pragma unroll` (V4)	0.0977	1682.32

5. Comparison of Grid and Block Sizes

Table 3. CNN Performance at various threads per block and blocks per grid

Grid (X,Y,Z)	Block (X,Y,Z)	Blk.Z	Grid.Z	Time (s)	GFlops
(7,7,96)	(16,16,4)	4	96	0.098102	1675.98
(7,7,128)	(16,16,4)	4	128	0.09838	1671.24
(7,7,64)	(16,16,4)	4	64	0.098417	1670.61
(7,7,256)	(16,16,2)	2	256	0.100585	1634.6
(7,7,128)	(16,16,2)	2	128	0.100996	1627.95
(7,7,192)	(16,16,2)	2	192	0.101129	1625.81
(7,7,384)	(16,16,1)	1	384	0.110262	1491.15
(7,7,256)	(16,16,1)	1	256	0.110603	1486.55
(7,7,512)	(16,16,1)	1	512	0.110737	1484.75

6. Analysis and Discussion

One problem I faced was in attempting to make a GEMM-version of the CNN problem, where we recast the kernel multiplication into a GEMM format. This became difficult to do with the current problem because it required us to allocate a much larger device memory from the host device. I considered using a preprocessing technique to metaprogram the main.cu program to allocate more memory before compile time, but this proved complicated and I ran out of time trying it out.