# GPU 2D Convolution Optimization Report

Anmol Gupta (27257)

November 23, 2025

## 1 Experimental Setup

### 1.1 Hardware Configuration

- **GPU:** NVIDIA A100-SXM4-40GB
- **Memory:** 40 GB HBM2
- **Compute Capability:** 8.0

### 1.2 Test Configuration

- **Batch Size:** 16
- **Image Dimensions:** 2048×2048 pixels
- **Kernel Size:** 11×11

## 2 Overall Performance Summary

| Implementation | Time (ms) | Speedup | Compute Throughput (%) | Memory Throughput |
|---|---|---|---|---|
| Baseline | 37.70 | 1.00× | 16.81 | 97.88 |
| Variant 1 | 7.87 | 4.80× | 80.52 | 57.51 |
| Variant 2 | 4.92 | 7.66× | 89.97 | 49.70 |
| Variant 3 | 4.75 | 7.94× | 93.19 | 51.19 |
| **Variant 4** | **2.38** | **15.84×** | **93.03** | **51.09** |

Table 1: Overall performance comparison across all variants

# 3 Variant 1: Global Memory Access Pattern Optimization

## 3.1 Implementation Strategy

- **Approach:** Restructured thread mapping to enable memory coalescing
- **Key Design:** Aligned threadIdx.x with contiguous memory dimension (columns)
- **Block Configuration:** Changed from 16×16 to 32×8 for optimal warp alignment
- **Tradeoffs:** Pure access pattern optimization without additional memory usage

```
// BEFORE: Poor coalescing
int row = blockIdx.y * blockDim.y + threadIdx.x;
int col = blockIdx.x * blockDim.x + threadIdx.y;

// AFTER: Optimal coalescing
int col = blockIdx.x * blockDim.x + threadIdx.x; // x for columns
int row = blockIdx.y * blockDim.y + threadIdx.y; // y for rows
```

Listing 1: Key Optimization in Variant 1

## 3.2 Performance Analysis

| Metric | Baseline | Variant 1 |
|---|---|---|
| Execution Time | 37.70 ms | 7.87 ms |
| Speedup | 1.00× | 4.80× |
| Compute Throughput | 16.81% | 80.52% |
| Memory Throughput | 97.88% | 57.51% |
| Global Load Efficency | 4% | 1.35% |

Table 2: Variant 1 performance metrics from Nsight Compute

## 3.3 Discussion

- **Why it works:** Enables hardware memory coalescing where 32 threads access contiguous addresses
- **Observed Phenomenon:** Dramatic compute throughput increase indicates transition from memory-bound to compute-bound
- **GPU Architecture:** Warp-based execution benefits from contiguous memory access patterns
- **Profiling Insight:** Memory throughput decreased but overall performance improved significantly

# 4 Variant 2: On-Chip Memory Optimization

## 4.1 Implementation Strategy

- **Approach:** Leveraged shared memory for data reuse and constant memory for kernel weights
- **Key Design:** Cooperative loading of input tiles with halo regions for boundary handling
- **Memory Usage:** 26×26 shared memory tiles for 11×11 kernel with 32×8 block size
- **Tradeoffs:** Increased complexity for significant data reuse benefits

```
__constant__ float constant_kernel[256]; // Constant memory
extern __shared__ float shared_tile[];   // Shared memory

// Cooperative loading with boundary handling
for (int idx = thread_id; idx < total_elements; idx +=
   total_threads) {
    if (input_row >= 0 && input_row < height && input_col >= 0 &&
       input_col < width) {
        shared_tile[load_y * shared_x + load_x] = pixel_val;
    }
}
__syncthreads();
```

Listing 2: Shared Memory Implementation in Variant 2

## 4.2 Performance Analysis

| Metric | Variant 1 | Variant 2 | Change |
|---|---|---|---|
| Execution Time | 7.87 ms | 4.92 ms | -37.5% |
| Compute Throughput | 80.52% | 89.97% | +11.7% |
| Memory Throughput | 57.51% | 49.70% | -13.6% |
| Shared Memory Usage | None | 26×26 tiles | New capability |

Table 3: Variant 2 performance improvements

## 4.3 Discussion

- **Why it works:** Shared memory provides 100× higher bandwidth than global memory
- **Observed Phenomenon:** Lower memory throughput but better performance due to data reuse

- **GPU Architecture:** On-chip memories reduce pressure on global memory bandwidth
- **Profiling Insight:** Each input pixel reused 121× (11×11 kernel), justifying shared memory overhead

# 5 Variant 3: Register-Level Optimization

## 5.1 Implementation Strategy

- **Approach:** Compiler-assisted optimizations with loop unrolling
- **Key Design:** Used #pragma unroll directives for both kernel dimensions
- **Optimization:** Enabled compiler to optimize register allocation and instruction scheduling
- **Tradeoffs:** Slight increase in register usage (32 to 34) for better ILP

```
// Loop unrolling for better register utilization
#pragma unroll
for (int ky = 0; ky < kernel_size; ky++) {
    #pragma unroll
    for (int kx = 0; kx < kernel_size; kx++) {
        const int filter_idx = ky * kernel_size + kx;
        const float weight = constant_kernel[filter_idx];
        result += pixel * weight;
    }
}
```

Listing 3: Register Optimization in Variant 3

## 5.2 Performance Analysis

| Metric | Variant 2 | Variant 3 | Change |
|---|---|---|---|
| Execution Time | 4.92 ms | 4.75 ms | -3.5% |
| Compute Throughput | 89.97% | 93.19% | +3.6% |
| Memory Throughput | 49.70% | 51.19% | +3.0% |
| Register Usage | 32 | 34 | +6.3% |

Table 4: Variant 3 performance improvements

## 5.3 Discussion

- **Why it works:** Eliminates loop overhead and enables better instruction-level parallelism

- **Observed Phenomenon:** Small but consistent improvement despite register pressure increase
- **GPU Architecture:** Register file provides fastest storage for thread-local computations
- **Profiling Insight:** 93.19% compute throughput indicates near-optimal compute utilization

# 6 Variant 4: Multi-Stream Concurrent Execution

## 6.1 Implementation Strategy

- **Approach:** Concurrent execution using multiple CUDA streams
- **Key Design:** Split batch into two halves processed in separate streams
- **Stream Management:** Created and synchronized two independent streams
- **Tradeoffs:** Increased complexity for significant parallelization benefits

```
// Use 2 streams for concurrent execution
const int NUM_STREAMS = 2;
cudaStream_t streams[NUM_STREAMS];

// Split batch and process concurrently
conv2d_variant3(input1, kernel, output1, batch1, height, width,
    kernel_size, streams[0]);
conv2d_variant3(input2, kernel, output2, batch2, height, width,
    kernel_size, streams[1]);

// Synchronize both streams
for (int i = 0; i < NUM_STREAMS; i++) {
    cudaStreamSynchronize(streams[i]);
    cudaStreamDestroy(streams[i]);
}
```

Listing 4: Multi-Stream Implementation in Variant 4

### 6.2 Performance Analysis

| Metric | Variant 3 | Variant 4 | Change |
|---|---|---|---|
| Execution Time | 4.75 ms | 2.38 ms | -49.9% |
| Speedup vs Baseline | 7.94× | 15.84× | +7.90× |
| Compute Throughput | 93.19% | 93.03% | -0.2% |
| Memory Throughput | 51.19% | 51.09% | -0.2% |

Table 5: Variant 4 performance through multi-stream execution

### 6.3 Discussion

- **Why it works:** Enables overlapping of independent computations across streams
- **Observed Phenomenon:** Nearly 2× performance improvement while maintaining efficiency
- **GPU Architecture:** Multiple streams utilize parallel execution engines on GPU
- **Profiling Insight:** Nsight Systems timeline shows concurrent kernel execution
- **Architectural Benefit:** Better utilization of GPU compute resources through parallelism

## 7 Architectural Insights

### 7.1 Memory Hierarchy Progression

The optimization journey systematically targeted different levels of GPU memory hierarchy:

1. **Variant 1:** Global memory access patterns (coalescing)
2. **Variant 2:** Shared memory (data reuse) + Constant memory (kernel weights)
3. **Variant 3:** Register file (thread-level optimization)
4. **Variant 4:** Stream-level parallelism (concurrent execution)

## 7.2 Performance Bottleneck Evolution

| Variant | Primary Bottleneck Addressed | Speedup |
|---|---|---|
| Baseline | Memory access patterns and coalescing | 1.00× |
| Variant 1 | Global memory transaction efficiency | 4.80× |
| Variant 2 | Data reuse and memory bandwidth | 7.66× |
| Variant 3 | Instruction-level parallelism | 7.94× |
| Variant 4 | Serial execution limitation | 15.84× |

Table 6: Evolution of performance bottlenecks through optimization

## 7.3 Nsight Compute Profiling Insights

Key metrics collected for each variant:

- **Compute Throughput:** Improved from 16.81% to 93.03%
- **Memory Throughput:** Evolved from inefficient high usage to efficient moderate usage
- **Memory Coalescing:** Dramatic improvement in Variant 1 maintained throughout
- **Register Usage:** Moderate increase in Variant 3 with performance benefit

# 8 Conclusion

## 8.1 Key Achievements

- **15.84× total speedup** over baseline implementation
- **93.03% compute throughput** achieved in final variant
- **Systematic optimization** through memory hierarchy and concurrency
- **Numerical correctness** maintained throughout all optimizations