Assignment 3

Rishikesh S

November 2025

1 Introduction

The goal of this assignment is to progressively optimize a 2D convolution kernel on Nvidia GPU's by exploring various optimization techniques such as : memory coalescing, on-chip memory, register tiling and concurrent execution.

2 Methodology

We describe the implementation of each variant here:

2.1 Baseline implementation

This was the given, inefficient implementation of a 2D convolution kernel. This kernel had uncoalesced global memory accesses, leading to low compute throughput and high execution time. Furthermore, the kernel maps threads within a warp to different rows instead of contiguous columns. This thread-to-data mapping is the direct cause of uncoalesced memory accesses, which serializes memory requests and reduces memory bandwidth.

2.2 Variant 1

In this variant, we change the thread-to-data mapping of the baseline implementation to take advantage of coalesced memory accesses. Furthermore, we change the block shape from (16, 16) to (32, 8) to ensure that every warp processes a single, contiguous segment of a row. This optimization improves memory bandwidth, compute throughput, and execution time.

2.3 Variant 2

In this variant, we implement **Shared** and **Constant** memory to further optimize our kernel. **The baseline implementation (and Variant 1) relied entirely on the hardware L1/L2 caches to mitigate the latency of redundant global memory accesses.** In this variant, we replace that implicit caching with explicit memory management.

- **Shared Memory:** This is a software-managed cache available for each thread block on NVIDIA GPUs. The latency of accessing shared memory is comparable to that of L1 cache, with the additional guarantee that data will not be evicted by the hardware. For our convolution kernel, we explicitly load the input image region required by the thread block into shared memory. This ensures that when different threads within the block access the same pixels for convolution, they retrieve them directly from high-speed **shared memory** instead of repeatedly fetching them from global memory.

- **Constant Memory:** This is a read-only memory region available on NVIDIA GPUs designed for high-speed access. Importantly, it is optimized for uniform access patterns through a **broadcasting** mechanism, which allows a single memory read to serve multiple threads simultaneously. In our convolution kernel, we store the convolution filter here, as the weights are never modified during execution and are accessed by all threads in the warp simultaneously.

We leverage these memory spaces to accelerate kernel execution. Each thread block (of dimension $m \times n$) is responsible for computing an output region of size $m \times n$. However, to perform the convolution, the thread block requires a larger input region of dimensions $(m + 2 \cdot \text{radius}) \times (n + 2 \cdot \text{radius})$ to account for the boundary pixels. We load this expanded region, corresponding to each thread block, into shared memory. Since this tile contains all input pixels necessary for the block's computation, threads can reuse this data efficiently. This drastically reduces global memory traffic, as threads retrieve input data directly from low-latency shared memory.

Finally, it is important to note that **constant memory** allocations require a fixed size defined at compile time. Even if we were to utilize dynamic shared memory to support arbitrary input tile sizes, the execution would still be limited by the static buffer size allocated for the filter weights. Therefore, **we have enforced a global maximum kernel radius of 7. Attempting to execute a kernel with a radius exceeding this limit will result in out-of-bounds memory accesses.**

2.4 Variant 3

In this variant, we implement **Thread Coarsening** and **Register Reuse** to further minimize Shared Memory traffic.

- **Background:** Registers represent the fastest level of the GPU memory hierarchy. Accessing data from registers is significantly faster than accessing Shared Memory or L1 cache. By maximizing register usage, we can reduce the instruction latency and pressure on the Shared Memory bandwidth.

Methodology: In standard convolution, adjacent threads compute adjacent output pixels. For example, Thread A computes output pixel i, requiring inputs $[j, j+1, j+2]$, while Thread B computes output pixel $i+1$, requiring inputs $[j+1, j+2, j+3]$. There is significant overlap in the input data required by these adjacent threads. To exploit this, we employ **thread coarsening**, where a single thread computes a 2×2 block of output pixels simultaneously. This allows us to perform **register reuse** via a sliding window mechanism:

1. In iteration k, the thread loads input pixel $j+1$ from shared memory to compute the contribution for the second output pixel.
2. For the next iteration $(k+1)$, the first output pixel *requires* input $j+1$.
3. Instead of reloading $j+1$ from shared memory, we simply shift the value stored in the register from the previous calculation steps.

By maintaining these values in local registers, we effectively halve the number of shared memory load instructions required for the inner loops. In our implementation, each thread computes 4 output pixels.

3 Observations and Discussion

All the results presented below are for a batch of 16 2048 x 2048 images, genrerated randomly, to which a convolution kernel of size 11 x 11 is applied.

3.1 Variant 1

Table 1: Performance Comparison: Baseline vs. Variant 1

Metric	**Baseline**	**Variant 1**	**Change (%)**
Sectors/Req	9.58	2.95	**-69.21%**
Compute Throughput (%)	23.08	98.31	**+325.95%**
Memory Throughput (GB/s)	14.23	60.51	**+325.23%**
Kernel Time (ms)	37.59	8.82	**-76.54%**
Throughput (MPix/s)	2396.50	8877.54	**+270.44%**

3.1.1 Discussion

The experimental data confirms that correcting the thread-to-data mapping successfully mitigates the memory bottleneck. The sharp decline in the Sectors/Req metric indicates that the hardware now merges per-thread loads into fewer, fuller transactions, validating that spatial locality has been achieved.

Consequently, the effective memory bandwidth increases substantially, as the bus is no longer saturated by fragmented requests. This efficient data delivery

allows the compute units to remain active without stalling for memory, leading to higher compute utilization and a significant improvement in overall execution speed.

3.2 Variant 2

Table 2: Performance Comparison: Baseline vs. Variant 2

Metric	**Baseline**	**Variant 2**	**Change (%)**
Sectors loaded from global mem	4,854,582,783	23,676,700	**-99.51%**
L1 cache hit rate (%)	99.32	10.58	**-89.35%**
Memory Throughput (GB/s)	14.23	108.22	**+660.51%**
Shared Memory load util (% Peak)	0.00	43.61	–
Kernel Time (ms)	37.59	4.93	**-86.88%**
Throughput (MPix/s)	2396.50	15463.02	**+545.23%**

3.2.1 Discussion

The experimental results for Variant 2 demonstrate the efficacy of explicit memory hierarchy management. By utilizing shared memory for data reuse and constant memory for kernel weights, we achieved a significant reduction in global memory traffic, improving the overall performance of our kernel.

Global Memory Traffic Reduction: The impact of this optimization is observed in the *sectors loaded from global memory* metric, which plummeted from 4.85 billion to 23.6 million. In the baseline, every thread independently requested its required pixels, forcing the memory subsystem to handle billions of redundant requests for overlapping regions. In Variant 2, threads collaboratively load a unique input tile into shared memory. This ensures that each pixel is fetched from global memory only once per thread block, effectively eliminating redundant global loads.

Shared Memory Utilization: The shift in data locality is confirmed by the *shared memory load utilization*, which reached **43.61% of the hardware peak**. This metric shows that the compute units are now fed primarily by the low-latency on-chip scratchpad rather than the high-latency global memory.

The L1 Hit Rate Paradox: An interesting phenomenon observed in Table 2 is the sharp decline in L1 Cache Hit Rate from 99.32% to 10.58%. Counterintuitively, this lower hit rate correlates with higher performance.

- In the **Baseline**, the L1 cache was inundated with redundant requests from neighboring threads accessing the same pixels. The high hit rate (99%) masked the inefficiency of the access pattern.

- In **Variant 2**, this data reuse is explicitly managed in shared memory. The requests that *do* reach the L1/Texture cache are primarily "compulsory misses" (loading the tile for the first time). Since shared memory filters out all the "easy" hits, the L1 hit rate naturally drops. Thus, the low hit rate is not a sign of failure, but a confirmation that shared memory has successfully taken over the responsibility of data reuse.

3.3 Variant 3

Table 3: Performance Comparison: Baseline vs. Variant 3 (Final Optimization)

Metric	**Baseline**	**Variant 3**	**Change (%)**
Compute Throughput (%)	23.08	92.94	**+302.69%**
Memory Throughput (GB/s)	14.23	254.65	**+1689.53%**
Kernel Time (ms)	37.59	2.10	**-94.41%**
Throughput (MPix/s)	2396.50	38493.02	**+1506.22%**
Registers Per Thread	40	40	–

3.3.1 Discussion

The performance data highlights the critical role of register-level data reuse. By enabling each thread to compute a 2×2 output patch (Thread Coarsening), we achieved a drastic increase in pixel throughput compared to the baseline.

Exploiting Instruction-Level Parallelism: The significant rise in throughput is a result of thread coarsening. Unlike single-pixel threads where pipeline stalls occur due to instruction dependencies, making each thread process four pixels exposes independent arithmetic instructions. This increased Instruction-Level Parallelism (ILP) allows the hardware scheduler to effectively hide latency, keeping the CUDA cores saturated with productive work.

Memory Bandwidth Saturation: The increase in **memory throughput** indicates a shift from a latency-bound to a bandwidth-bound state. This allows the SMs to consume data at a much higher rate, leading to the significant decrease in execution time.

Register Efficiency vs Performance: Despite the complexity of maintaining a sliding window and multiple accumulators, the *Registers Per Thread* does not show any increase. This indicates highly efficient compiler scheduling. By avoiding excessive register pressure, the kernel achieves a perfect balance: it maintains high occupancy to hide global latency, while utilizing ILP to hide arithmetic latency.