APMA 2822: High Performance Computing
Final Project Report

PERFORMANCE ANALYSIS OF 2D IMAGE CONVOLUTION: CPU, GPU, AND DISTRIBUTED PARALLELISM

January 6, 2026

Project Repository: `https://github.com/DonSteven/conv2d-project`

Sihan Long
School of Engineering
Brown University

Abstract

This project implements and evaluates high-performance solvers for 2D image convolution, a fundamental kernel in scientific computing. We explore four parallelism strategies: multi-threaded CPU (OpenMP), distributed memory (MPI), GPU Naive, and GPU Shared-Memory Tiling (CUDA). Experiments reveal that GPU implementations achieve up to ∼1000 GFLOP/s, providing a speedup of over 20x compared to the optimized CPU baseline. Profiling via Nsight Systems identifies memory allocation and PCIe transfer as the primary end-to-end bottlenecks. Roofline analysis demonstrates that the RTX 3090's L2 cache effectively mitigates the global memory bandwidth overhead in the Naive kernel, yielding performance comparable to the manually optimized Tiled kernel for this specific workload.

1 Introduction

Image convolution involves computing the weighted sum of pixel neighborhoods. For an input image $I \in \mathbb{R}^{H \times W}$ and a square kernel K of size $(2r+1) \times (2r+1)$, the output $F(x, y)$ is:

$$F(x, y) = \sum_{u=-r}^{r} \sum_{v=-r}^{r} K(u, v) I(x+u, y+v) \tag{1}$$

The computational intensity is determined by the total floating-point operations (FLOPs), approximated as $2k^2HW$, where $k = 2r + 1$. This project focuses on minimizing runtime and maximizing GFLOP/s through architecture-specific optimizations.

2 Implementation Details

2.1 CPU and OpenMP

The baseline implementation utilizes nested loops with clamped boundary conditions. OpenMP parallelism is applied to the outer loop (`#pragma omp parallel for`), distributing rows across CPU cores. This shared-memory approach is limited by the system's aggregate memory bandwidth.

2.2 MPI Domain Decomposition

We implement a 1D domain decomposition where the image is split into horizontal strips.

- **Halo Exchange:** To process boundary pixels, each rank exchanges r "ghost rows" with its neighbors using `MPI_Sendrecv`.
- **Scalability:** This enables processing images larger than a single node's memory capacity and leverages distributed compute resources.

2.3 GPU Acceleration (CUDA)

Naive Kernel: Maps one thread to one output pixel. It relies heavily on the GPU's L2 cache to handle redundant reads from global memory.

Tiled Kernel: Utilizes Shared Memory (scratchpad) to explicitly manage data locality. Threads cooperatively load a tile of pixels (including the halo) into on-chip memory, reducing global memory transactions.

3 Experimental Setup

- **CPU:** 2 × AMD EPYC 7413 (Milan), Theoretical Peak BW: 409.6 GB/s.
- **GPU:** NVIDIA GeForce RTX 3090, Theoretical Peak BW: 936.1 GB/s, Peak FP32: 35.58 TFLOP/s.
- **Parameters:** Kernel size $K = 7$ (7×7), Image sizes $N \in \{1024, 2048, 4096, 8192\}$.

4 Results and Analysis

4.1 OpenMP Strong Scaling

OpenMP Strong Scaling: As shown in Table 1, performance scales well up to 4 threads but saturates significantly by 8–16 threads. This indicates the application hits the **Memory Wall**, where CPU memory bandwidth becomes the limiting factor rather than compute core availability.

Table 1: OpenMP Strong Scaling ($N = 4096, K = 7$).

Threads	Time [ms]	Throughput [GFLOP/s]	Speedup
1	897.08	1.83	0.99×
2	455.13	3.61	1.95×
4	232.79	7.06	3.81×
8	241.35	6.81	3.68×
16	238.47	6.89	3.72×

These results establish a key baseline: even with multiple CPU threads, convolution becomes bandwidth-bound quickly, so additional cores do not translate into proportional speedup. We next examine whether distributing the working set across MPI ranks can improve scaling by reducing shared-memory contention, at the cost of explicit communication.

4.2 MPI Strong Scaling

MPI Strong Scaling: MPI demonstrates superior scaling efficiency (Table 2), achieving strong speedup at higher rank counts. This suggests that domain decomposition can mitigate shared-memory bandwidth contention by distributing working sets and memory traffic across ranks, at the cost of explicit communication and synchronization.

Table 2: MPI Strong Scaling ($N = 4096, K = 7$).

Ranks	CPU Base [ms]	MPI Total [ms]	GFLOP/s	Speedup
1	883.17	922.24	1.78	0.96×
2	883.50	467.70	3.52	1.89×
4	883.89	244.03	6.74	3.62×
8	892.90	130.43	12.61	6.85×

Compared to OpenMP, MPI improves scaling at higher parallelism levels by trading implicit shared-memory contention for explicit halo exchange and collectives. With the CPU-side scaling picture in place, we now turn to GPU acceleration, where the performance regime shifts again: the kernel can approach device bandwidth limits, while end-to-end time is strongly influenced by runtime overheads and PCIe transfers.

4.3 GPU Throughput and End-to-End Performance

We measured both the pure kernel throughput (Table 3) and the end-to-end application runtime (Table 4), which includes memory allocation and host-device data transfer.

Table 3: Kernel Throughput comparison ($K = 7$).

N	CPU [GFLOP/s]	GPU Naive [GFLOP/s]	GPU Tiled [GFLOP/s]
1024	1.83	818.78	646.78
2048	1.83	964.13	947.63
4096	1.84	998.52	980.23
8192	1.85	1006.98	984.76

Table 4: End-to-end runtime (including H2D/D2H copy and allocation) and speedup vs. CPU.

N	CPU [ms]	GPU Naive		GPU Tiled	
		Time [ms]	Speedup	Time [ms]	Speedup
1024	55.71	3.75	14.86×	3.63	15.34×
2048	222.30	11.14	19.96×	11.17	19.91×
4096	884.74	38.90	22.75×	38.82	22.79×
8192	3559.76	150.06	23.72×	149.04	23.88×

Analysis:

1. **Naive vs. Tiled:** Surprisingly, the Naive kernel performs slightly *better* or equal to the Tiled kernel in GFLOP/s for large N. This suggests that the RTX 3090's L2 cache captures spatial locality as effectively as manual tiling for this stencil size, without the overhead of thread synchronization (`__syncthreads()`) and halo thread divergence inherent in the Tiled approach.

2. **Warm-up Effect:** Early runs were dominated by CUDA runtime initialization (context creation, module loading, and allocator setup). Adding an explicit warm-up isolates steady-state behavior, ensuring that reported kernel and end-to-end timings reflect the actual algorithmic cost rather than one-time startup penalties.

4.4 Roofline Analysis

Figure 1 illustrates the performance relative to hardware limits. The Naive implementation (red dot) operates slightly *above* the theoretical memory bandwidth bound (based on Global Memory traffic). This confirms that the **Effective Arithmetic Intensity** is significantly higher than the theoretical value (0.49) due to high L2 cache hit rates.

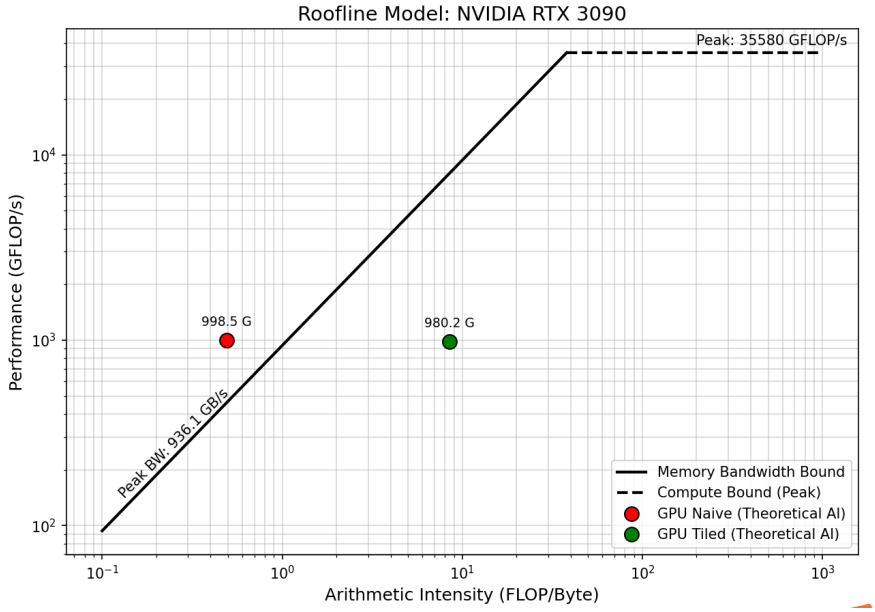

Figure 1: Roofline Model for RTX 3090 ($N = 4096$). The proximity of the Naive kernel to the bandwidth ceiling indicates the application is memory-bound.

Figure 2 depicts the Roofline model for the dual AMD EPYC 7413 node. The baseline single-threaded implementation (blue dot) achieves $\sim$1.84 GFLOP/s.

Unlike the GPU results, this point lies far below both the memory bandwidth and compute ceilings. This large gap quantifies the efficiency loss due to the lack of parallelism (utilizing only 1 of 48 available cores) and the lack of explicit SIMD vectorization in the scalar baseline code. The vertical distance to the roof represents the potential speedup unlockable via OpenMP (to hit the memory bandwidth roof) and cache blocking (to move right towards the compute roof).

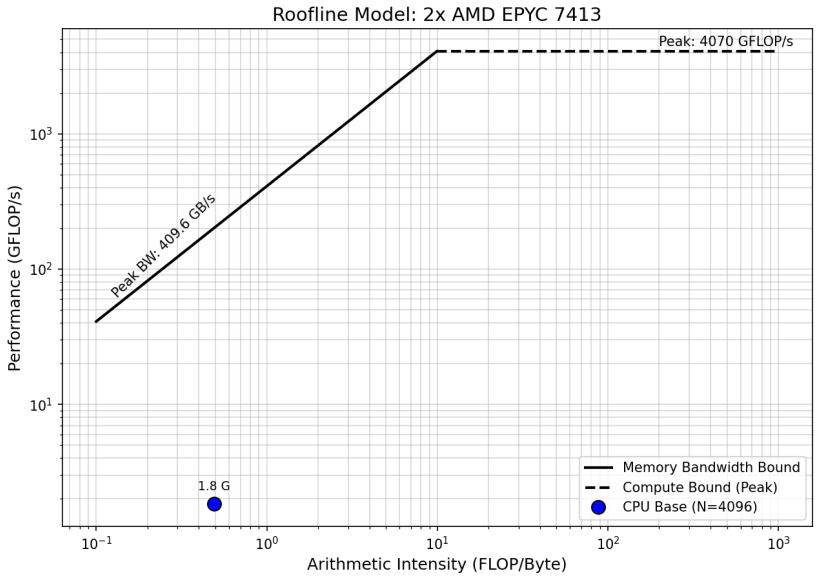

Figure 2: Roofline Model for 2x AMD EPYC 7413 ($N = 4096$). The single-threaded baseline (blue) is far below the hardware limits, indicating massive room for parallel optimization.

4.5 Profiling Deep Dive

We utilized NVIDIA Nsight Systems to analyze the execution timeline. Two complementary instrumentation mechanisms are used:

- **NVTX ranges (semantic annotation):** The application inserts labeled ranges (e.g., `GPU Naive Total`, `Malloc`, `H2D Copy`, `Naive Kernel Compute`, `D2H Copy`). These appear as a dedicated **NVTX track** and provide ground-truth boundaries for end-to-end phases from the application perspective.

- **CUDA events (numerical kernel timing):** Events bound the kernel launch and are synchronized to report precise device-side elapsed time for CSV and GFLOP/s calculations. In contrast to NVTX (which is primarily structural), events are optimized for accurate timing.

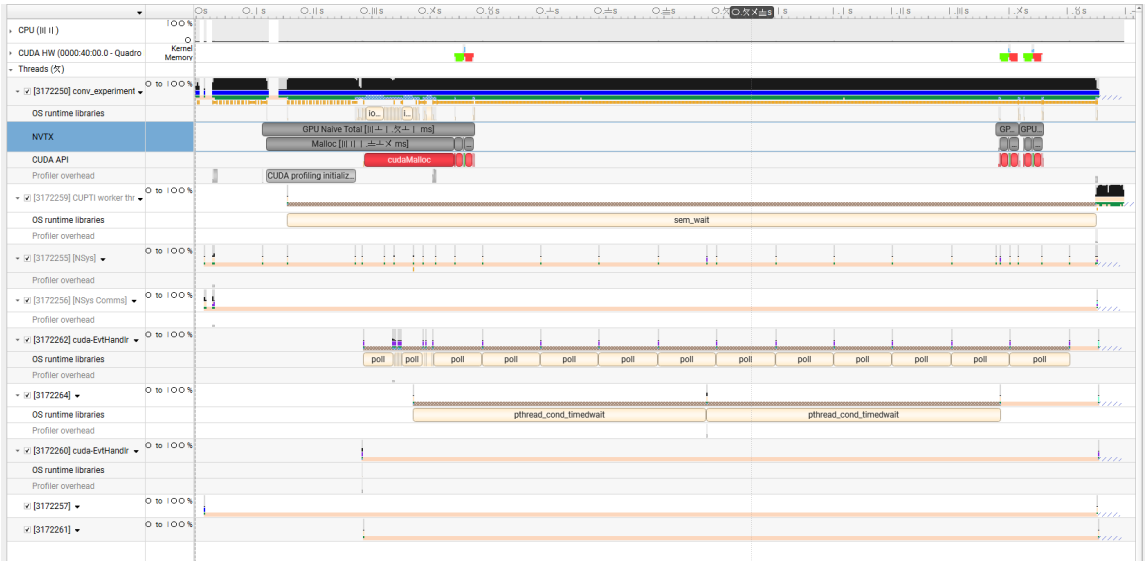

Figure 3: Nsight Systems overall timeline. CUDA runtime calls (e.g., `cudaMalloc`/`cudaMemcpy`) dominate wall time relative to kernel execution, especially during the warm-up segment that triggers one-time runtime initialization.

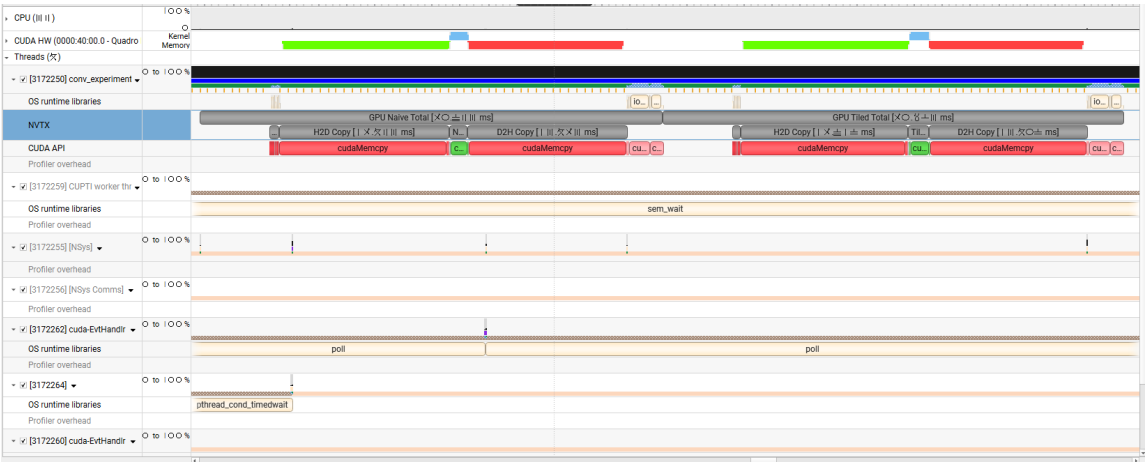

Figure 4: Detailed view of steady-state GPU phases (corresponding to the second GPU activity region). H2D copy → kernel → D2H copy is serialized on the default stream.

Interpreting key tracks. Nsight Systems exposes multiple layers of the execution stack:

- **NVTX track:** User-defined timeline ranges. In this project, NVTX cleanly separates allocation, PCIe copies, and compute. This is essential because the CPU baseline and post-processing (correctness checks, printing, and FLOP calculations) otherwise appear as "gaps" between GPU phases.
- **CUDA API track:** Host-side runtime calls such as `cudaMalloc`, `cudaMemcpy`, and `cudaEventSynchronize`. These calls execute on the CPU but often block to coordinate with the device, explaining why the program can appear CPU-serialized even though the kernel is massively parallel.
- **OS Runtime Libraries track:** Kernel-driver interactions triggered by CUDA runtime operations. Frequent `ioctl` activity is expected: CUDA uses `ioctl` to communicate with the NVIDIA kernel driver for memory management, launches, synchronization, and telemetry. This track is a strong indicator that overhead is in *runtime/driver control paths*, not in arithmetic.
- **Profiler Overhead track:** Instrumentation costs incurred by tracing itself (buffer flushes, bookkeeping, and sampling). This track should be interpreted as measurement overhead and not attributed to the application; it becomes more visible when collecting fine-grained traces or sampling CPU stacks.

Why the first GPU segment is slower. Profiling shows that the *first* `GPU Naive Total` range is substantially longer than subsequent invocations, despite executing the same kernel. This is consistent with cold-start behavior: CUDA context initialization, just-in-time loading of device code, and allocator setup are typically triggered by the first CUDA API calls (often `cudaMalloc` or the first kernel launch). The explicit warm-up call converts this cost into a one-time overhead, improving repeatability of steady-state measurements.

Why the timeline has long "blank" regions between GPU ranges. The program structure intentionally interleaves CPU work with GPU benchmarks. In particular, the CPU baseline convolution, correctness checks, and result formatting occur between GPU phases and are not wrapped in NVTX ranges. As a result, Nsight shows GPU inactivity (and fewer CUDA API events) between `GPU Naive Total` and the next GPU region. This is not wasted GPU time inside a kernel; it is simply time spent on CPU-only sections that are outside the instrumented NVTX scope.

Core bottlenecks and what they imply. The timeline confirms an important end-to-end performance reality:

- **Allocation dominates for short runs:** Per-call `cudaMalloc`/`cudaFree` introduces substantial overhead and drives many driver round-trips (`ioctl`). This cost is algorithm-independent and therefore a prime optimization target.

- **Transfers are serialized with compute:** The current code uses synchronous copies and the default stream, yielding a strict H2D → kernel → D2H dependency chain. Consequently, the kernel cannot hide transfer latency; the reported end-to-end speedup is limited by PCIe and runtime overhead even when the kernel itself approaches the bandwidth roof.

5 Conclusion

This project implemented and analyzed CPU (serial/OpenMP), MPI, and CUDA convolution solvers on modern HPC hardware. The results highlight a recurring theme in performance engineering: optimizing the *kernel* is necessary but not sufficient for optimizing the *application*. While both CUDA kernels achieve high steady-state throughput and sit near the bandwidth roof for large images, Nsight Systems shows that end-to-end execution is dominated by runtime overheads (`cudaMalloc`/`cudaFree`) and PCIe transfers that are currently serialized with compute.

Outlook and next steps. The profiling insights directly motivate a prioritized optimization roadmap:

- **Eliminate per-invocation allocation:** Persist device buffers across runs (or across image sizes where feasible) to remove repeated allocator and driver overhead. A production-grade approach would use CUDA memory pools and asynchronous allocation APIs to reduce allocator synchronization and improve temporal locality.
- **Overlap transfer and compute:** Replace blocking `cudaMemcpy` with `cudaMemcpyAsync` on non-default streams, and use pinned host memory to enable true asynchronous PCIe transfers. For batched workloads (multiple images or multiple channels), a double-buffered pipeline can hide most transfer latency behind kernel execution.
- **Reduce launch/synchronization overhead:** For fixed stencil parameters, CUDA Graph capture can amortize CPU-side launch costs and reduce runtime jitter, which matters when kernels are short or when profiling fine-grained phases.
- **Strengthen CPU-side parallel efficiency:** For OpenMP, future work should focus on NUMA-aware placement, first-touch initialization, and vectorization-friendly loop structure. For MPI, nonblocking halo exchange (e.g., `MPI_Isend`/`MPI_Irecv`) combined with interior-region compute can overlap communication with computation when scaling beyond a single node.

Overall, the combination of Roofline modeling and timeline profiling provides a coherent explanation of the observed behavior: the GPU kernels are bandwidth-bound but efficient, whereas the application is constrained by data motion and runtime control overheads.

Addressing these system-level bottlenecks is the most promising direction for further speedups.