CS610 Assignment-4

Mahaarajan J - 220600

November 16, 2025

Workstations used: `csews8` and `gpu3` (sm_80 GPU)

Running Instructions

```
# Compile all CUDA/C++ files using the provided Makefile
make

# This produces the following executables in their respective directories:
./p1       # Problem 1
./p2a        # Problem 2 (part a)
./p2b        # Problem 2 (part b)
./p3_v1       # Problem 3 (naive)
./p3_v2       # Problem 3 (optimized multi-launch)
./p3_v3       # Problem 3 (UVM version)
./p3_v4       # Problem 3 (Thrust version)
./p4      # Problem 4

# To clean binaries:
make clean
```

All programs print CPU time, kernel-only GPU time, and end-to-end time. GPU outputs are compared with CPU reference results and report `No differences found` upon correctness.

1 Problem 1: CUDA Kernel and Memory Optimization

1.1 Objective

To optimize a simple 1D CUDA kernel over a large array and compare:

- CPU baseline.
- A naive global-memory kernel.
- A shared-memory tiled kernel.
- A further optimized kernel (loop unrolling, `__restrict__`, tuned grid/block).
- A version using pinned host memory and multiple CUDA streams to overlap host–device transfers with kernel execution.

Both kernel-only and end-to-end timings (including allocations and copies) were measured, and Nsight was used to profile runtime breakdown.

1.2 Approach

Starting from a straightforward element-wise kernel, the following steps were applied:

1. **Naive kernel:** Each thread processes one element using direct global-memory loads and stores. Grid size is chosen so that N elements are covered by `blockDim.x * gridDim.x` threads.
2. **Shared-memory kernel:** Input data for each block is cooperatively loaded into a shared-memory tile and then written back to global memory. This reduces redundant global loads and improves coalescing.

3. **Optimized kernel:**
   - Used `const` and `__restrict__` qualifiers.
   - Per-thread strip-mining so each thread processes multiple elements in a strided loop.
   - Chosen block sizes from {1, 2, 4, 8} and empirically tuned.
4. **Pinned & overlapped version (V4):**
   - Host arrays allocated with `cudaMallocHost()`.
   - Work split into chunks and processed using several CUDA streams.
   - For each stream: enqueue H2D copy, kernel launch, and D2H copy, allowing PCIe transfers to overlap with kernel execution.
5. **Correctness:** For every configuration the GPU result is compared against the CPU baseline; runs report "No differences found".

1.3 Performance Results

Block size experiments showed that very small blocks (1, 2 threads) greatly hurt performance: for block size 1, kernel times rose to ≈ 19–24 ms and end-to-end to 80+ ms. Block sizes 4 and 8 were stable; best performance is on size=4. Also for reference the average CPU time was 37ms (independent of block sizes)

Kernel-only Time (ms)

Block Size	Naive	Shared	Optimized
8	6.73075	1.42438	0.667648
4	4.81382	2.37158	0.653312
4	3.71814	2.36646	0.65040
2	6.73792	5.72109	3.13043
1	19.0648	24.448	21.5245

End-to-End Time (ms)

Block Size	Naive	Shared	Optimized	Pinned+Overlap
8	68	62	61	33
4	66	63	61	32
4	64	62	60	32
2	69	67	64	36
1	82	86	83	53

1.4 Nsight Profiling Observations

Nsight Systems was used on the optimized configuration:

- **GPU-side breakdown:**
  - `gpukernsum` showed three kernels (naive, shared, optimized) each taking only tens of microseconds; the optimized kernel contributed the smallest share.
  - `gpumemtimesum` showed that `cudaMemcpy` H2D and D2H together account for roughly $> 95\%$ of total GPU time, confirming that end-to-end performance is dominated by PCIe transfers rather than computation.
- **CPU-side breakdown:** System trace indicated most CPU time in `sem_wait`, `poll`, and `ioctl`, i.e., the CPU is largely waiting for GPU work to complete. Overheads from `cudaMalloc/cudaFree` were visible but small compared to data transfers.
- **Overlap effectiveness:** With multiple streams and pinned memory, the kernel execution bars in the Nsight timeline largely overlap with H2D/D2H memcpy bars, reducing the effective end-to-end time from about 60–68 ms to about 32–36 ms.

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
42.9	1,34,73,28,271	2	67,36,64,135.5	67,36,64,135.5	57,08,946	1,34,16,19,325	94,46,31,288.0	sem_wait
39.5	1,23,77,02,131	18	6,87,61,229.5	2,33,84,770.0	3,829	53,64,57,529	12,45,34,931.5	poll
16.1	50,61,11,284	514	9,84,652.3	15,117.0	1,031	17,60,29,824	91,21,141.8	ioctl
1.3	4,10,29,495	27	15,19,610.9	15,284.0	2,653	2,16,28,934	54,33,208.3	mmap
0.1	27,27,664	27	1,01,024.6	15,175.0	8,780	17,16,110	3,25,330.6	mmap64
0.0	4,08,220	9	45,357.8	43,228.0	34,184	62,769	9,708.4	sem_timedwait
0.0	3,89,358	45	8,652.4	7,627.0	4,541	20,591	3,219.6	open64
0.0	3,40,773	1	3,40,773.0	3,40,773.0	3,40,773	3,40,773	0.0	pthread_cond_wait
0.0	3,08,363	39	7,906.7	5,545.0	1,258	36,660	7,599.2	fopen
0.0	2,67,128	4	66,782.0	68,423.0	58,619	71,663	5,724.6	pthread_create
0.0	1,35,687	17	7,981.6	7,103.0	3,161	15,881	3,484.9	munmap
0.0	96,625	5	19,325.0	9,463.0	6,069	55,914	21,039.9	fgets
0.0	90,227	16	5,639.2	6,147.0	1,047	11,291	3,351.5	fwrite
0.0	66,498	29	2,293.0	1,852.0	1,113	7,495	1,425.9	fclose
0.0	41,170	6	6,861.7	6,700.5	1,937	12,932	3,668.5	open
0.0	34,637	2	17,318.5	17,318.5	12,989	21,648	6,122.8	socket
0.0	34,138	15	2,275.9	2,212.0	1,092	3,516	753.4	read
0.0	27,305	10	2,730.5	2,777.0	1,660	3,923	763.5	write
0.0	24,693	2	12,346.5	12,346.5	10,044	14,649	3,256.2	fread
0.0	21,706	3	7,235.3	9,211.0	2,742	9,753	3,900.8	pipe2
0.0	18,817	2	9,408.5	9,408.5	5,709	13,108	5,231.9	pthread_cond_broadcast
0.0	13,053	1	13,053.0	13,053.0	13,053	13,053	0.0	connect
0.0	9,162	1	9,162.0	9,162.0	9,162	9,162	0.0	putc
0.0	4,448	1	4,448.0	4,448.0	4,448	4,448	0.0	fopen64
0.0	4,352	1	4,352.0	4,352.0	4,352	4,352	0.0	bind
0.0	3,579	3	1,193.0	1,008.0	1,008	1,563	320.4	fcntl
0.0	1,206	1	1,206.0	1,206.0	1,206	1,206	0.0	listen

Figure 1: Nsight Profile 1

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
29.7	18,07,35,914	4	4,51,83,978.5	1,63,669.5	1,01,427	18,03,07,148	9,00,82,128.6	cudaMalloc
29.7	18,06,87,251	6	3,01,14,541.8	3,01,34,843.0	2,97,16,279	3,05,19,533	3,79,231.6	cudaMemcpy
24.9	15,16,21,560	2	7,58,10,780.0	7,58,10,780.0	7,50,31,441	7,65,90,119	11,02,151.8	cudaHostAlloc
9.0	5,47,32,736	2	2,73,66,368.0	2,73,66,368.0	2,72,19,569	2,75,13,167	2,07,605.1	cudaFreeHost
5.3	3,21,34,122	1	3,21,34,122.0	3,21,34,122.0	3,21,34,122	3,21,34,122	0.0	cudaStreamSynchronize
0.7	43,52,411	3	14,50,803.7	16,84,475.0	9,43,636	17,24,300	4,39,671.2	cudaEventSynchronize
0.5	29,24,778	4	7,31,194.5	7,47,427.5	2,59,496	11,70,427	4,97,248.3	cudaFree
0.1	5,80,437	4	1,45,109.3	63,401.5	30,384	4,23,250	1,86,084.8	cudaLaunchKernel
0.0	83,933	3	27,977.7	27,846.0	25,076	31,011	2,969.7	cudaMemset
0.0	51,692	1	51,692.0	51,692.0	51,692	51,692	0.0	cudaStreamCreate
0.0	34,731	2	17,365.5	17,365.5	9,221	25,510	11,518.1	cudaMemcpyAsync
0.0	30,669	6	5,111.5	5,212.0	2,846	7,137	1,996.1	cudaEventRecord
0.0	28,585	1	28,585.0	28,585.0	28,585	28,585	0.0	cudaStreamDestroy
0.0	18,389	2	9,194.5	9,194.5	1,268	17,121	11,209.8	cudaEventCreate
0.0	1,044	1	1,044.0	1,044.0	1,044	1,044	0.0	cuModuleGetLoadingMode

[6/8] Executing 'gpukernsum' stats report

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ	BlockXYZ	Name
38.7	17,25,132	1	17,25,132.0	17,25,132.0	17,25,132	17,25,132	0.0	32 32 32	8 8 8	naive_kernel(const double *, double *, unsigned long)
31.7	14,12,618	1	14,12,618.0	14,12,618.0	14,12,618	14,12,618	0.0	32 32 32	8 8 8	shmem_kernel(const double *, double *, unsigned long)
29.6	13,22,314	2	6,61,157.0	6,61,157.0	6,60,101	6,62,213	1,493.4	32 32 32	8 8 8	opt_kernel(const double *, double *, unsigned long)

[7/8] Executing 'gpumemtimesum' stats report

Time (%)	Total Time (ns)	Count	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Operation
50.0	10,57,02,642	4	2,64,25,660.5	2,96,05,020.0	1,67,75,869	2,97,16,733	64,33,425.9	[CUDA memcpy HtoD]
49.7	10,50,10,478	4	2,62,52,619.5	3,00,64,527.5	1,47,25,614	3,01,55,809	76,84,796.7	[CUDA memcpy DtoH]
0.3	7,00,934	3	2,33,644.7	2,32,962.0	2,32,322	2,35,650	1,765.9	[CUDA memset]

Figure 2: Nsight Profile 2

1.5 Summary

- Shared-memory tiling and some micro-optimizations like pre-computing reduced kernel-only time from $\approx$ 6–7 ms (naive) to $\approx$ 0.65 ms (optimized).
- However, end-to-end performance is constrained by host–device transfers; naive vs. optimized end-to-end times differ only modestly (68 ms $\rightarrow$ 60 ms).
- Using pinned memory and overlapping transfers with computation via multiple streams nearly halves the end-to-end runtime ($\approx$ 32 ms), giving the most practical speedup.

2 Problem 2: Inclusive Prefix Sum Using CUDA

2.1 Objective

To implement an inclusive prefix-sum (scan) over large arrays using two GPU strategies:

- **Version A (Copy Then Execute):** Standard device allocations using `cudaMalloc` with explicit host–device transfers.
- **Version B (UVM):** Using `cudaMallocManaged` to enable on-demand migration and allows oversubscription beyond GPU memory capacity.

Both versions were benchmarked for very large input sizes up to 2^{32} elements. (The first kernel gave **OOM** error at like 2^{32} due to over-subscription (uint_32 was used and GPU had a capacity of 4G)

2.2 Approach

1. **CPU Baseline:** Sequential single-thread prefix sum.
2. **GPU Kernel:** Shared-memory block-level scan + global block-prefix propagation.
3. **Version A:** Allocations via `cudaMalloc`, explicit `cudaMemcpy`, kernel launch, and device-to-host copy.
4. **Version B (UVM):** Allocations via `cudaMallocManaged` while using hints `cudaMemAdvise()` and `cudaMemPrefetchAsync()`, letting the driver handle migrations automatically.

All results were validated against the CPU baseline; each run reported: `No differences found` ensuring correctness of our implementation.

As part of optimisation , I also experimented with all the UVM–related advice hints recommended in the assignment, including `cudaMemAdviseSetPreferredLocation`, `cudaMemAdviseSetAccessedBy`, and `cudaMemAdviseSetReadMostly`, along with explicit `cudaMemPrefetchAsync` calls to the GPU before launching each kernel. These were tested both individually and in combination; however, for Part (i) (the copy-based implementation), these advice flags did not provide any benefit as there was no real pre-fetching, and the behaviour remained dominated by raw memory bandwidth but these mechanisms became important in the UVM version, where prefetching helped reduce on-demand page faults

2.3 Notes on Measurement

Prefix-sum is **memory-bandwidth bound**, and timings were highly **noisy**, especially for large N. Unified Memory additionally introduces page fault overhead and TLB pressure making it slower than Copy Then Execute for reasonable sizes. However it is the only implementation that can handle oversubscription to the GPU memory (even though it is slow due to the page faults)

2.4 Performance Results

Kernel-only Time (ms)

Input Size	Copy-based Kernel	UVM Kernel	CPU time
2^{25}	1.84	61	175
2^{27}	7.30	175.9	510
2^{30}	-	1656.26	4547

End-to-End Timing Results (ms)

Input Size	Copy-based Kernel	UVM Kernel	CPU time
2^{25}	139	395	175
2^{27}	526	2199	510
2^{30}	-	12701	4547

2.5 Memory Considerations

The prefix sum problem is memory bound hence we do not get much speedup wrt the CPU on either of our methods on end-end time. The copying of memory from CPU to GPU is where most of the time is spent, I tried reducing this using the hints but they did not give as much speedup

2.6 Unified Memory Oversubscription

- With UVM, arrays up to 2^{30} elements ran successfully despite exceeding physical VRAM.
- Performance degraded dramatically due to continuous page migration the time for 2^{30} doubles was 12710ms and the process was getting killed by the cluster when attempted on even larger values
- Attempting 2^{33} elements resulted in `GPUassert: out of memory` on running the `CTE` version indicating the oversubscription limit had been surpassed.

2.7 Summary

- Copy-based version is significantly faster and more stable.
- UVM supports oversubscription but introduces large overheads.
- Prefix-sum is dominated by memory bandwidth, hence measurement noise and not very effective on GPU
- All outputs matched CPU reference outputs for correctness.

3 Problem 3: GPU Acceleration of a 10-Dimensional Constraint Grid Search

3.1 Objective

To accelerate the exhaustive 10-dimensional grid search where each grid point must satisfy ten linear constraints

$$|C_i x - d_i| \leq e_i.$$

The GPU implementation must:

- produce the exact same **11608** satisfying points as the CPU version, and
- preserve the CPU's strict lexicographic (r1 $\rightarrow$ r10) output order.

3.2 Implemented Versions

The given .c file took **432s** for refernce on `csews 8`. We use its result.txt here for correctness. Four progressively refined GPU versions were implemented. Many additional ideas (warp-aggregation, strip-mining, deeper unrolling, UVM migration, alignment strategies, register-pressure reductions) were tested, but these did not provide consistent speedups and are not treated as separate versions.

Version 1 (V1): Naive Single-Kernel Implementation

- The entire 10-dimensional grid is processed using a single monolithic kernel, with each thread responsible for evaluating exactly one grid point against the ten linear constraints.
- Satisfying points are recorded using a global atomic counter that provides a unique write position for each successful thread. Although atomics introduce serialization, the number of satisfying points (11608) is extremely small relative to the total grid size, so contention remains limited.
- The lexicographic ordering requirement is preserved explicitly by mapping thread indices to the r1–r10 loop nest in the same linearized order as the CPU reference code. This ensures that the output produced by V1 is bit-identical and order-identical to the sequential implementation.
- This version serves as the baseline for all subsequent optimizations. Despite being simple, it already achieves a substantial speedup over the CPU because the constraint evaluation is embarrassingly parallel across grid points.

Version 2 (V2): Multi-Launch r1-Sliced Version with Shared/Constant Memory, Tiling, and Loop Unrolling

- The full 10-dimensional grid was partitioned into 13 independent slices, one for each value of the outermost coordinate r_1.Each slice was processed using a separate kernel launch to allow for easier parallelism given the large iteration space
- The read-only coefficient arrays (C, d, e) were moved into `__constant__` memory so that all threads in a warp could access them through the constant cache. This eliminates repeated global-memory loads during constraint evaluation.
- Frequently reused parts of (C, d) were also cooperatively staged/ precomputed then passed on as `__shared__` memory at the start of each kernel. Each thread block loads the required values exactly once, after which all threads reuse the same shared-memory tiles. This significantly reduces memory traffic.
- A small tiling strategy was introduced along the inner grid dimensions so that each thread processes a few adjacent points (i.e., a small tile) before writing outputs. This improves locality and amortizes index-computation overhead across multiple constraint evaluations.
- A light strip-mining strategy was also incorporated so that each thread evaluates a small batch of adjacent grid points, improving index-arithmetic reuse while keeping register pressure manageable.
- The loop over the 10 constraints was partially unrolled to extract some instruction-level parallelism and reduce loop-control overhead.
- These modifications together create a cleaner memory-access pattern, reduce repeated arithmetic inside the nested loops, and improve warp utilization. While the performance improvement over the naive baseline was modest and somewhat noisy, Version2 consistently outperformed Version1 in both kernel-only and end-to-end timings.

Version 3 (V3): Unified Memory Version with Memory Hints and Prefetching

- In this version, all data structures-including the grid parameters, coefficient matrices (C, d, e), and the output buffers-were allocated using `cudaMallocManaged`. Unified Memory (UVM) allows the GPU driver to migrate pages automatically between CPU and GPU memory, enabling the program to run even when the working set exceeds the available device memory.
- Several UVM-related performance hints were applied, including `cudaMemAdviseSetPreferredLocation`, `cudaMemAdviseSetAccessedBy`, and `cudaMemAdviseSetReadMostly`, with the goal of reducing on-demand page migration during kernel execution. These hints inform the driver where the data is likely to reside and how it will be accessed.
- Explicit `cudaMemPrefetchAsync` calls were also issued before each kernel launch. Prefetching the managed pages to the GPU helps reduce page faults inside the kernel and ensures that most of the active working set is available on device memory at launch time.
- UVM made experimentation with very large grid sizes feasible, and the implementation remained completely functional even when the managed allocation exceeded device memory capacity. However, this flexibility came at the cost of substantial runtime variability due to page thrashing and PCIe traffic.
- Although V3 incorporates the same logic as the optimized multi-launch implementation, its performance was noticeably more noisy and generally slower. Kernel times fluctuated significantly (e.g., 24–27 seconds) depending on the amount of page migration triggered at runtime. This behavior reflects the inherently unpredictable nature of UVM-driven memory movement in a branch-heavy, low-locality workload such as this constraint evaluation kernel.
- Overall, V3 demonstrates the ease-of-use benefits of Unified Memory and provides a correct, order-preserving implementation, but it does not deliver stable speedups compared to the explicit-memory versions.

Version 4 (V4): Thrust-Based Compaction and Lexicographic Sorting

- In this version, the kernel does not directly write satisfying points into a global output buffer using atomics. Instead, each thread writes a simple binary flag indicating whether its assigned grid point satisfies all ten constraints. This eliminates atomic contention entirely and converts the problem into a parallel filtering step.
- After kernel execution, we use `thrust::transform` to convert the boolean flags into index values for all valid grid points. These indices are then compacted using `thrust::copy_if`, producing a dense array of exactly **11608** satisfying point locations.
- Because the CPU reference output must be in strict lexicographic order, the compacted index list is passed through a `thrust::sort` step. Sorting cost was found to be negligible (typically < 1 ms) relative to the total runtime, owing to the small number of satisfying points.
- Once the sorted list of valid indices is available, the final coordinates are generated in a post-processing stage. This ensures that the final output matches the CPU reference *exactly*, both in content and ordering.
- This approach cleanly separates computation from output generation and removes the need for atomic writes entirely. However, the additional transformation, compaction, and sorting stages add overhead to the end-to-end runtime. As a result, V4 performed slightly slower overall compared to the optimized multi-launch versions (V2 and V3), even though its kernel-only time was competitive.
- Despite the marginally higher end-to-end time, V4 offers a conceptually clean and highly maintainable implementation, ensuring correctness without relying on atomics or complex ordering logic.

3.3 Performance Results

All versions match the CPU output exactly and produce **11608** satisfying points. All optimized kernels are more or less similar in their performance and do significantly beat the naive kernel. Also the kernel, end to end times were almost similar due to the task being compute heavy and memory does not play a huge role here

Timings were consistently **noisy** due to:

- Branch divergence,
- high register pressure (up to 255 registers/thread in early versions),
- occupancy variability,

- large instruction footprint,
- and variable amounts of active work (only 11608 outputs out of millions of grid points).

Kernel-only Times

Version	Kernel Time (ms)
V1: Naive single kernel	27315–29025
V2: Optimised multi-launch (shared + step + unroll)	23600–23660
V3: Optimized hints + UVM	23480–23740
V4: Thrust-based compaction	24535–24540

End-to-End Times

Version	End-to-End Time (ms)
V1: Naive single kernel	27316–29026
V2: Optimised multi-launch (shared + step + unroll)	23610–23660
V3: Optimized hints + UVM	23481–23745
V4: Thrust-based compaction	24541–24735

3.4 Summary

- Four GPU versions were implemented: Naive (V1), Multi-Launch (V2), Optimized Multi-Launch (V3), and Thrust Compaction (V4).
- All versions produced the correct output and preserved lexicographic ordering.
- Although optimizations reduced kernel workload, the problem remains dominated by branching, register pressure, and sparse arithmetic intensity.
- Resulting performance remained clustered around 23–29 seconds across versions (astronomical speed up over the 432s in reference file), with no single method providing a clear consistent advantage.

4 Problem 4: GPU Optimization of 2D and 3D Convolution

4.1 Objective

The goal of this problem was to accelerate a small-radius 2D and 3D convolution (filter sizes 3×3 and 3×3×3) on large grids:

$$N_{2D} = 256, \qquad N_{3D} = 1024,$$

and compare:

- CPU baseline implementation,
- Basic CUDA kernels (global memory),
- Optimized CUDA kernels using shared memory tiling, constant memory, and loop unrolling.

All GPU outputs were checked against CPU reference results, and all runs reported `No differences found`.

4.2 Implemented Optimizations

The optimized kernels contain several GPU-specific optimizations:

- **Constant memory filters:** The 3×3 and 3×3×3 convolution filters are stored in `__constant__` memory, enabling warp-wide broadcast and reducing global memory traffic.
- **Shared-memory tiling (2D and 3D):** The input tiles are loaded cooperatively into a shared-memory buffer:

  $$\text{Tile size}_{2D} = (BX{+}2R) \times (BY{+}2R) = 18 \times 18,$$

  $$\text{Tile size}_{3D} = (4{+}2R)^3 = 6 \times 6 \times 6.$$

  This eliminates redundant global loads and improves locality. I tried to optimize it further using padding to avoid bank conflicts but it did not give me significant speedup
- **Thread-block tiling:** Blocks of size $(4, 4)$ for 2D and $(4, 4, 4)$ for 3D were used. These were empirically tuned but still yielded highly noisy results.
- **Loop unrolling:** The three nested loops over the filter window were unrolled using `OMP` in both 2D and 3D kernels, reducing loop-control overhead

Many additional optimizations were attempted (larger block sizes, padding for coalescing, 3D tiling variants, alternative loop permutations), but they introduced either higher register pressure or greater timing noise and did not result in consistent significant speedups.

4.3 Performance Results

All results below are taken directly from the executed run:

Table 1: Performance of 2D and 3D Convolution (CPU vs GPU Basic vs GPU Optimized)

Case	CPU Time (ms)	GPU Kernel(ms)	GPU E-E(ms)
2D Basic	3749.43	4.00496	1067.97
2D Optimized	—	5.43308	2736.95
3D Basic	7956.04	10.391	703.195
3D Optimized	—	5.93495	541.091

4.4 Discussion of Results

- **2D kernel noise:** 2D kernels exhibited extreme variability. In some runs, the optimized kernel performed worse than the basic global-memory version because:
  - the shared-memory tile (18×18) is small, yielding little reuse,
  - overhead from explicit tile setup dominates,
  - register pressure reduces occupancy for small block sizes.

- **3D kernel stability:** The 3D kernels showed much more consistent behavior. The optimized version halved the kernel-only runtime (10.39 ms $\rightarrow$ 5.93 ms), though end-to-end time remained dominated by large memory transfers of a 1024^3 volume.
- **End-to-end dominated by transfers:** For both 2D and 3D, PCIe transfers of large arrays (256×256 and 1024^3 floats) dominated the total runtime. As a result, even large improvements in kernel-only time translated to modest or noisy improvements in total time.
- The speedup of using a GPU can be seen from the 3D example where there is a 10x speedup even in the naive case over CPU
- **All outputs correct:** Every implementation matched the CPU output (zero mismatches).

Note: All the results were noisy and varied across runs, sometimes even the naive kernel performed better. This could be due to the convolution operation being small (3x3 and 3x3x3)

4.5 Summary

- Shared-memory tiling, constant memory filters, and loop unrolling were successfully implemented for both 2D and 3D convolution.
- The 3D optimized kernel gave a clear improvement in kernel-only runtime (almost $2\times$).
- The 2D optimized kernel was more unstable and often slower end-to-end, due to setup overhead and low arithmetic intensity.
- End-to-end timings were dominated by host-device transfers, limiting practical speedups.
- Correctness was preserved across all kernel variants.