CS 557 — Programming Assignment 1

Xinchen Shen

February 6, 2026

1 System Configuration

Component	Specification
CPU	AMD Ryzen 9 9900X (12 cores / 24 threads, Zen 5)
Memory	1×64 GB DDR5-5600 (single-channel)
OS	Windows 11 x64
Compiler	Clang++ (version 21.1.8) (LLVM) via MSVC Build Tools

Compilation. All code was compiled using the `build-windows` target in the provided Makefile, which invokes Clang++ with the flags `-O3 -march=native -std=c++23 -fopenmp=libomp` and links against `libomp.lib` from the LLVM distribution. Three separate executables were built: `lap.exe` (baseline i-j-k), `lap_kji.exe` (k-j-i), and `lap_ikj.exe` (i-k-j).

OpenMP settings. During formal experiments, the following environment variables were set to pin threads to physical cores and prevent the Windows scheduler from migrating threads:

```
OMP_DYNAMIC=FALSE  OMP_PROC_BIND=close  OMP_PLACES=cores
```

This was necessary because preliminary runs without thread pinning showed noticeable timing jitter (see Section 2).

2 Verifying OpenMP Functionality

To confirm that OpenMP is working, the baseline code was run with thread counts of 1, 2, 4, 8, 12, 16, and 24 both with and without thread pinning (`make test` and `make test-dynamic-close`). Figure 1 shows the average runtime (excluding the first cold-start iteration) for each configuration.

Based on these observations, all formal loop-ordering experiments (Section 4) use pinned threads to ensure stable, reproducible measurements.

3 Effective Bandwidth Estimation

Memory footprint. The grid is 512^3 with two `float` arrays (`u` and `Lu`):

$$\text{Total data} = 2 \times 512^3 \times 4 \text{ bytes} = 2^{30} \text{ bytes} = 1.00 \text{ GiB}$$

Each invocation reads the entire `u` array and writes the entire `Lu` array, so exactly 1.00 GiB of data is transferred through the memory subsystem per iteration.

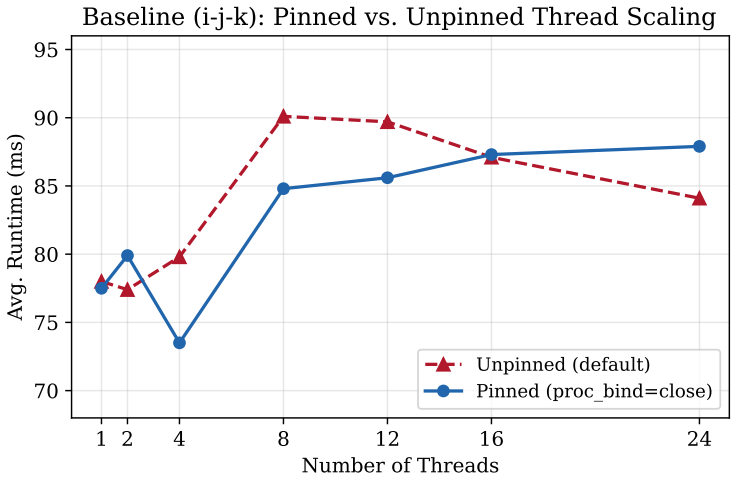

Figure 1: Thread scaling with and without thread pinning. Both curves confirm OpenMP is active: runtime varies with thread count. Without pinning, the Windows scheduler migrates threads between cores, introducing jitter and degraded performance at higher thread counts. With pinning (`OMP_PROC_BIND=close`, `OMP_PLACES=cores`), behavior stabilizes, with the best runtime at 4 threads (∼73 ms). Both configurations plateau beyond 4 threads, indicating the benchmark is memory-bandwidth-bound.

Theoretical peak bandwidth. With a single DDR5-5600 DIMM, the system operates in single-channel mode:

$$\text{Peak BW} = 5600\ \text{MT/s} \times 8\ \text{bytes} = 44.8\ \text{GB/s} \approx 41.7\ \text{GiB/s}$$

STREAM benchmark. The STREAM benchmark was compiled and run to provide a practical upper bound. Results:

Kernel	Best Rate (GiB/s)	% of Peak
Copy	21.29	51.0%
Scale	19.84	47.6%
Add	22.36	53.6%
Triad	21.28	51.0%

Table 1: STREAM benchmark results (converted to GiB/s). Achieving ∼50% of peak is reasonable for single-channel DDR5.

Laplacian effective bandwidth. Using the average runtimes from the formal experiments (iterations 2–10, with pinning), the effective bandwidth for the baseline (i-j-k) configuration with 12 threads is:

$$\text{Effective BW} = \frac{1.00\ \text{GiB}}{82.0\ \text{ms}} \approx 12.2\ \text{GiB/s}$$

This is about 29% of the theoretical peak and roughly 57% of the STREAM Triad bandwidth. The gap between STREAM and Laplacian performance is expected: the stencil's irregular access

pattern (reading 7 neighbors per output point) reduces cache efficiency compared to STREAM's purely sequential accesses.

4 Loop Order Experiments

The formal experiments (`make exp-windows`) tested three loop orderings at 12 and 24 threads, with thread pinning enabled. The first iteration of each run is excluded to avoid cold-start effects. Table 2 and Figures 2, 3 summarize the results.

Loop Order	Threads	Avg. Time (ms)	Eff. BW (GiB/s)	% of STREAM	% of Peak	Slowdown
i-j-k (baseline)	12	82.0	12.2	57.3%	29.3%	1.0×
i-j-k (baseline)	24	85.7	11.7	55.0%	28.1%	1.0×
i-k-j	12	135.4	7.4	34.8%	17.8%	1.7×
i-k-j	24	131.3	7.6	35.7%	18.2%	1.5×
k-j-i	12	1199	0.83	3.9%	2.0%	14.6×
k-j-i	24	1109	0.90	4.2%	2.2%	12.9×

Table 2: Performance of different loop orderings. "% of STREAM" uses STREAM Triad (21.28 GiB/s) as reference. "Slowdown" is relative to the baseline at the same thread count.

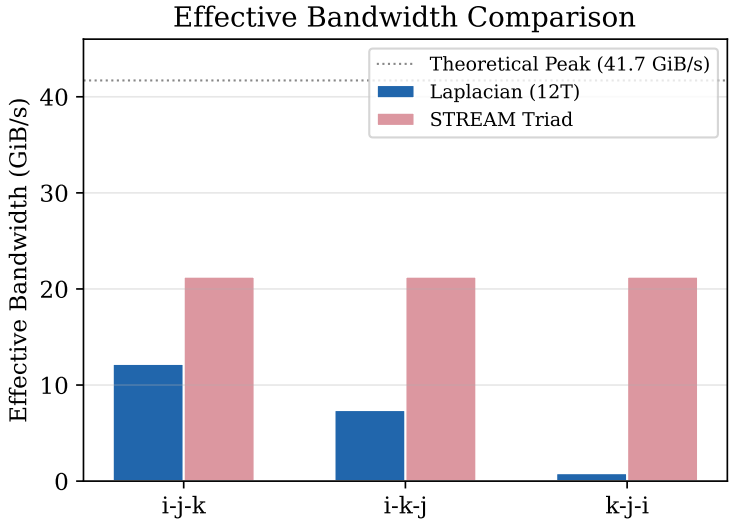

Figure 2: Effective bandwidth vs. STREAM Triad reference. The dashed line indicates the theoretical peak (41.7 GiB/s).

5 Discussion

Why i-j-k is fastest. In C/C++, multidimensional arrays are stored in row-major order, meaning the last index (`k`) varies fastest in memory. The i-j-k ordering places `k` as the innermost loop, so consecutive iterations access contiguous memory addresses. This maximizes spatial locality and allows the hardware prefetcher to work effectively, resulting in the best cache utilization.

Why k-j-i is catastrophically slow (∼15× slower). Reversing all three loops makes `i` the innermost loop. Since `u[i][j][k]` has stride $512 \times 512 \times 4 = 1{,}048{,}576$ bytes between consecutive `i` values, every inner-loop iteration touches a completely different cache line roughly 1 MB apart. This causes massive cache misses on every access and defeats the prefetcher entirely, explaining the order-of-magnitude slowdown.

Why i-k-j is moderately slower (∼1.6×). Here the outermost loop (`i`) is preserved, so the OpenMP parallelization granularity is unchanged. However, swapping the inner two loops means the innermost loop iterates over `j` with stride $512 \times 4 = 2{,}048$ bytes, which is much worse than the stride-4 accesses of the baseline but far better than the stride-1 MiB of k-j-i. The moderate slowdown reflects the intermediate level of cache inefficiency.

Thread count observations. Increasing from 12 to 24 threads (i.e., using hyperthreads) provides no improvement for the baseline, which is expected for a memory-bandwidth-bound workload: there is no additional memory bandwidth to exploit, and hyperthreads share the same physical resources. For k-j-i, the 24-thread run is slightly faster than 12 threads, likely because hyperthreading can partially hide the extreme memory latency caused by the cache misses.

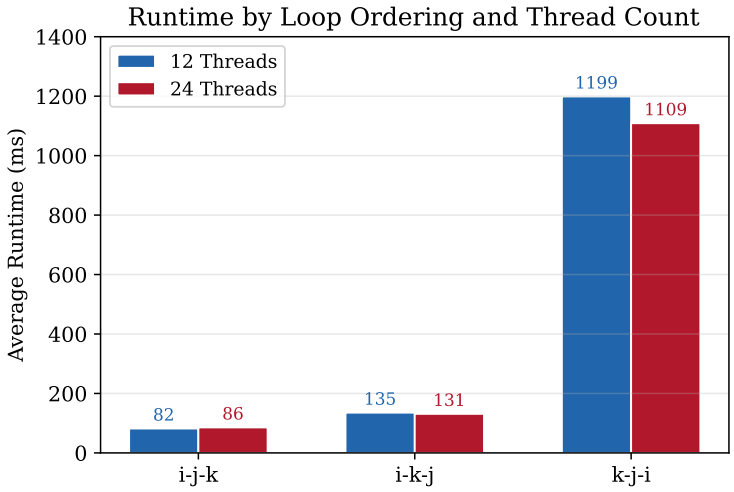

Figure 3: Runtime comparison across loop orderings.