CS 557 — Programming Assignment 2

Xinchen Shen

February 21, 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++ 21.1.8 (LLVM)

Compilation and reproducibility.
The project uses CMake (Ninja generator) to build four executables from the same source files, differentiated by compile definitions: `lap_baseline` (unmerged, 12 threads), `lap_merged` (`-DUSE_MERGED`, 12 threads), `lap_baseline_serial` (unmerged, 1 thread), and `lap_merged_serial` (`-DUSE_MERGED`, 1 thread). All targets are compiled with `-O3 -march=native -std=c++23 -fopenmp=libomp` and linked against `libomp.lib`. Image output is disabled during timing runs by passing `writeIterations = false`.

```
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_COMPILER=clang++ -G Ninja -S
    . -B build
cmake --build build
ctest --test-dir build -V
```

OpenMP settings.
CTest sets `OMP_DYNAMIC=FALSE`, `OMP_PROC_BIND=close`, `OMP_PLACES=cores`, and `OMP_NUM_THREADS=12` (or 1 for serial runs). As observed in Programming Assignment 1, enabling SMT (24 threads) provides no benefit for memory-bandwidth-bound workloads and can degrade performance due to contention on shared resources within each core. Therefore, all parallel experiments use exactly 12 threads pinned one-per-core, effectively utilizing all physical cores while avoiding SMT interference.

2 Task 1: Instrumented Kernel Timing

Each kernel invocation in the Conjugate Gradients loop is wrapped with a dedicated `Timer` object that accumulates time across all 256 iterations using the `Restart()`/`Pause()` interface from `LaplaceSolver_0_3`. Timers are declared as globals in `main.cpp` and referenced via `extern` declarations in `ConjugateGradients.cpp`. Kernels inside the loop (lines 6–16) are each called 256 times; lines 2 and 4 are one-time setup.

		12 Threads			1 Thread		
Line	Kernel	Time (ms)	Avg/call (ms)	% CG	Time (ms)	Avg/call (ms)	% CG
2	`ComputeLaplacian(x, z)`	11.7	11.7	0.1	15.5	15.5	0.1
2	`Saxpy(z, f, r, -1)`	23.3	23.3	0.1	22.8	22.8	0.1
2	`Norm(r)`	3.3	3.3	0.0	7.8	7.8	0.0
4	`Copy(r, p)`	7.6	7.6	0.0	16.2	16.2	0.1
4	`InnerProduct(p, r)`	6.8	6.8	0.0	13.8	13.8	0.1
6	`ComputeLaplacian(p, z)`	2491.6	9.7	14.9	2835.7	11.1	13.0
6	`InnerProduct(p, z)`	1433.8	5.6	8.6	3673.2	14.3	16.8
8	Saxpy(z, r, r, $-\alpha$)	2769.2	10.8	16.6	2360.4	9.2	10.8
8	`Norm(r)`	830.0	3.2	5.0	1927.2	7.5	8.8
13	`Copy(r, z)`	1812.5	7.1	10.9	2373.8	9.3	10.8
13	`InnerProduct(z, r)`	1443.1	5.6	8.7	3696.7	14.4	16.9
16	Saxpy(p, x, x, α)	2956.5	11.5	17.7	2402.1	9.4	11.0
16	Saxpy(p, r, p, β)	2846.3	11.1	17.1	2507.3	9.8	11.5
	Sum of kernels	**16635.8**		**99.8**	**21852.2**		**99.9**
	Total CG	**16667.3**			**21876.5**		

Table 1: Kernel timing breakdown for the unmerged baseline. Lines 2 and 4 execute once; lines 6–16 execute 256 times each.

Observations.
In the parallel case, the four `Saxpy` calls dominate (∼51%), followed by `ComputeLaplacian` (15%) and the two `InnerProduct` calls (17%). The `Copy` at line 13 is surprisingly expensive (10.9%) for a pure memory copy — this is entirely memory bandwidth cost.
In the serial case, the relative cost of `InnerProduct` increases significantly (from 17% to 34%) because it uses `omp parallel for reduction` and scales well with threads, whereas the `Saxpy` calls lack OpenMP pragmas in the original code.
`ComputeLaplacian` is only ∼14% faster in parallel (2492 ms) than in serial (2836 ms), confirming that the stencil computation is memory-bandwidth-bound: adding more threads does not proportionally increase available bandwidth with a single memory channel.

3 Task 2: Kernel Merging

3.1 Merged Kernels Implemented

Four merged kernels were implemented in `MergedKernels.cpp`, consolidating five opportunities identified in the lecture notes:

1. `LaplacianSaxpyAndNorm` (Line 2).
Merges `ComputeLaplacian(x, z)`, `Saxpy(z, f, r, -1)`, and `Norm(r)` into a single pass. The intermediate array `z` is never written to memory; the Laplacian is computed, subtracted from `f`, and the infinity norm is accumulated — all in one loop nest. This eliminates two full array traversals.

2. `LaplacianAndDot` (Line 6).
Merges `ComputeLaplacian(p, z)` and `InnerProduct(p, z)`. The Laplacian $\mathcal{L}p$ is computed and stored into `z`, while simultaneously accumulating $\mathbf{p}^T\mathbf{z}$. This saves one full read of both `p` and `z`.

3. `SaxpyAndNorm` **(Line 8).**
Merges `Saxpy(z, r, r, -`α`)` and `Norm(r)` so that the updated residual and its norm are computed in a single pass over the `r` and `z` arrays.

4. `DoubleSaxpy` **(Line 16).**
Merges `Saxpy(p, x, x,` α`)` and `Saxpy(p, r, p,` β`)`. Both operations read `p`, so by caching `p[i][j][k]` in a register and computing both updates, we halve the reads of `p`. Since `p` is both input and output, the old value must be read before writing the new one; the merged kernel handles this by saving `p_val` before updating.

5. Elimination of `Copy(r, z)` **(Line 13).**
In the merged version, `Copy(r, z)` is eliminated entirely. Since $\mathcal{M} = I$ (no preconditioner), line 13 reduces to $\mathbf{z} \leftarrow \mathbf{r}$, so `InnerProduct(z, r)` becomes `InnerProduct(r, r)`. This removes one full array copy per iteration.

3.2 Merged Timing Results

Algorithm Line	Merged Kernel	12 Threads		1 Thread	
		Time (ms)	Avg/call (ms)	Time (ms)	Avg/call (ms)
Line 2	`LaplacianSaxpyAndNorm`	13.8	13.8	20.4	20.4
Line 4	`Copy(r, p)`	8.2	8.2	12.1	12.1
Line 4	`InnerProduct(p, r)`	6.8	6.8	14.2	14.2
Line 6	`LaplacianAndDot`	3023.5	11.8	4851.3	19.0
Line 8	`SaxpyAndNorm`	2319.4	9.1	2638.2	10.3
Line 13	`InnerProduct(r, r)`	1140.8	4.5	3548.4	13.9
Line 16	`DoubleSaxpy`	3550.5	13.9	4456.0	17.4
	Sum of kernels	**10062.9**	–	**15540.6**	–
	Total CG	**10105.4**	–	**15567.6**	–

Table 2: Merged kernel timing for both parallel and serial runs.

3.3 Comparison: Unmerged vs. Merged

Threads	Kernel Group	Unmerged (ms)	Merged (ms)	Speedup
12	Line 2 (Lap+Saxpy+Norm)	38.3	13.8	2.78×
12	Line 6 (Lap+Dot)	3925.4	3023.5	1.30×
12	Line 8 (Saxpy+Norm)	3599.2	2319.4	1.55×
12	Line 13 (Copy+Dot)	3255.6	1140.8	2.85×
12	Line 16 (2×Saxpy)	5802.8	3550.5	1.63×
12	**Total CG**	**16667.3**	**10105.4**	**1.65×**
1	Line 2 (Lap+Saxpy+Norm)	46.1	20.4	2.26×
1	Line 6 (Lap+Dot)	6508.9	4851.3	1.34×
1	Line 8 (Saxpy+Norm)	4287.6	2638.2	1.63×
1	Line 13 (Copy+Dot)	6070.5	3548.4	1.71×
1	Line 16 (2×Saxpy)	4909.4	4456.0	1.10×
1	**Total CG**	**21876.5**	**15567.6**	**1.41×**

Table 3: Comparison of unmerged kernel groups and their merged replacements.

4 Discussion

Overall speedup.
Kernel merging delivers a **1.65**× speedup with 12 threads and **1.41**× in the serial case. The larger parallel benefit is expected: with multiple threads saturating memory bandwidth, each unnecessary array traversal becomes more costly, so eliminating them has a proportionally larger impact.

Biggest winners.
The most dramatic improvement comes from eliminating `Copy(r, z)` at line 13. Each array is 256^3 floats = 64 MiB. The baseline performs a full copy (128 MiB read+write) followed by an inner product re-reading both arrays (another 128 MiB). The merged `InnerProduct(r, r)` reads only `r` once (64 MiB), saving ∼192 MiB per iteration, or ∼48 GiB over 256 iterations — explaining the 2.85× parallel speedup.

`LaplacianAndDot` (line 6).
This merge shows a modest 1.30× improvement in parallel. The 7-point stencil has higher arithmetic intensity than the other kernels, partially masking the bandwidth savings from eliminating a second pass over `p` and `z`.

`DoubleSaxpy` (line 16).
The 1.63× parallel speedup comes from reading `p` once instead of twice. The serial improvement is smaller (1.10×), likely because the single thread is not bandwidth-limited and the compiler may keep `p` in cache across the two sequential loops.

Correctness verification.
All four configurations produce identical results (Table 4).

Configuration	Iterations	Final Residual Norm (ν)
Unmerged, 12 threads	256	0.000975891
Merged, 12 threads	256	0.000975891
Unmerged, 1 thread	256	0.000975891
Merged, 1 thread	256	0.000975891

Table 4: Correctness verification: all configurations converge identically.

5 Selected Code Snippets

Timer instrumentation with compile-time dispatch.
The `USE_MERGED` flag (set via CMake) selects between baseline and merged paths:

```
        // Algorithm : Line 6
#ifdef USE_MERGED
        timerLaplacianAndDot_line6.Restart();
        float sigma = LaplacianAndDot(p, z);
        timerLaplacianAndDot_line6.Pause();
#else
        timerLaplacian_line6.Restart();
        ComputeLaplacian(p, z);
        timerLaplacian_line6.Pause();

        timerInnerProduct_line6.Restart();
        float sigma = InnerProduct(p, z);
        timerInnerProduct_line6.Pause();
#endif
```

`LaplacianAndDot` — **fusing stencil with inner product.**

```
float LaplacianAndDot(const float (&p)[XDIM][YDIM][ZDIM],
                      float (&z)[XDIM][YDIM][ZDIM]) {
    double result = 0.0;
#pragma omp parallel for reduction(+:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++) {
        const float lp = -6*p[i][j][k]
            + p[i+1][j][k] + p[i-1][j][k]
            + p[i][j+1][k] + p[i][j-1][k]
            + p[i][j][k+1] + p[i][j][k-1];
        z[i][j][k] = lp;
        result += (double)p[i][j][k] * (double)lp;
    }
    return (float)result;
}
```