TP1: Optimizing Memory Access

Lab Report

Khaoula Jellal

Parallel And Distributed Programming

January 2026

Exercise 1: Impact of Memory Access Stride

Objective

The goal is to measure how memory access stride (the gap between consecutive memory reads) affects execution time and memory bandwidth in a simple summation program. This effect comes mainly from CPU cache behavior: how well data fits in cache, how the hardware predicts what data to load next, and potential cache conflicts. We also compare two compiler optimization levels (-O2 vs. -O0) to see how they change performance, sometimes making programs much faster but occasionally causing problems.

The program sums 1 000 000 doubles (8 MB of data) from a large array, using constant strides from 1 to 20. The array is intentionally large (160 MB) so that large-stride accesses stay within bounds and exceed typical L3 cache sizes.

Methodology

- Array size: `malloc(1 000 000 × 20 × sizeof(double))` (160 MB)
- Array values: all set to 1.0 (expected sum = 1 000 000.0 for any stride)
- Access pattern: `for (i = 0; i < N × stride; i += stride) sum += a[i];` (always exactly 1 000 000 reads)
- Timing: wall-clock time using `clock()`, converted to milliseconds
- Bandwidth calculation: 8 MB × (1000 / msec) / 1024
- Compiler versions: `gcc -O0` (basic, no optimization) vs. `gcc -O2` (loop unrolling, vectorization, better instruction order)

Results

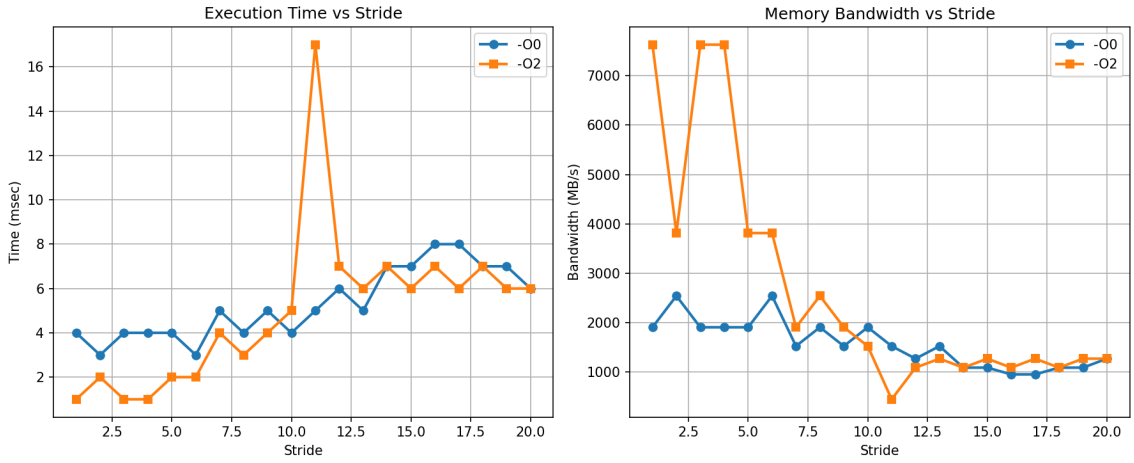

Figure 1: Execution time (left) and effective memory bandwidth (right) vs. stride for -O0 and -O2 compilations.

Table 1: Selected results showing key performance patterns.

Stride	-O0		-O2	
	Time (ms)	BW (MB/s)	Time (ms)	BW (MB/s)
1	4	1907	**1**	**7629**
4	4	1907	**1**	**7629**
8	4	1907	3	2543
11	5	1525	**17**	**449**
20	6	1271	6	1271

Key observations:

- Best performance (-O2, stride 1/3/4): about 1 ms, giving 7629 MB/s — 4 times faster than -O0 at stride 1.
- Strides 1–6 generally perform well with -O2; bandwidth often drops by half or more at certain steps (e.g., stride 2/5/6 get about 3815 MB/s).
- Severe slowdown with -O2 at stride 11: 17 ms / 449 MB/s (17 times slower than best case, 3.4 times worse than -O0 at same stride).
- For strides above 12, both -O0 and -O2 get similar results: 1000–1300 MB/s — limited by main memory speed.

Analysis

Impact of Stride on Cache Behavior

Modern CPUs load 64-byte cache lines (8 doubles). Stride determines how efficiently we use cache:

- **Stride 1:** Reading consecutive memory → perfect cache use, hardware prefetcher works very well → peak bandwidth.
- **Small strides (2–6):** Partial cache use; prefetcher still helps somewhat, especially for strides 3 and 4 with -O2.
- **Stride 8:** Only one double per cache line (12.5% efficiency) → bandwidth drops, but prefetching helps reduce the loss.
- **Large/irregular strides:** Almost random access → prefetcher can't help, many cache misses, limited by memory latency.

This explains why bandwidth decreases as stride increases.

Compiler Optimization Effects

The -O0 version produces a simple loop (one load and add per iteration). The -O2 version uses loop unrolling (doing multiple iterations at once), SIMD vectorization (processing 4 doubles per instruction with AVX), and better instruction scheduling. These optimizations give large improvements when memory is *not* the limiting factor (small strides):

- Up to 4 times faster at stride 1.
- Clear benefit even at moderate strides (3–6).

When memory speed *is* the limiting factor (large strides), optimizations don't help much; both versions get similar low bandwidth.

The Stride-11 Performance Problem with -O2

The most surprising result is the severe performance drop at stride 11 with -O2 (17 times slower than best case). This happens because of conflict misses in the cache. The stride of 11 (a prime number, not a power of 2) combined with the array's memory address and the optimized access pattern (unrolled/vectorized code changes memory offsets and timing) causes many cache lines to compete for the same cache locations. This creates thrashing (constantly replacing cache data) and very high miss rates.

This problem is absent or much smaller with -O0, where the simple loop doesn't trigger the same conflicts. This shows an important lesson: aggressive optimizations can sometimes expose hardware-specific problems that don't appear in unoptimized code.

Exercise 2: Optimizing Matrix Multiplication

Objective

The goal is to demonstrate how loop ordering in matrix multiplication dramatically affects cache performance due to spatial and temporal locality. We compare the naive (i-j-k) loop order which exhibits poor locality against an optimized (i-k-j) order that reuses data more effectively in cache. Performance is measured via execution time, GFLOPS, and effective memory bandwidth on square matrices of size $n = 400$ (each matrix $\approx$1.28 MB, total working set $\approx$3.8 MB).

Methodology

- Matrices: three $n \times n$ double arrays (A, B, C) allocated dynamically and initialized with random values in [0,1).
- Standard version (i-j-k): outer loops over i and j, inner over k computes $c_{ij} = \sum_k a_{ik} \cdot b_{kj}$.
- Optimized version (i-k-j): outer loops over i and k, inner over j computes $c_{ij} += a_{ik} \cdot b_{kj}$ with a_{ik} hoisted outside the inner loop.
- Timing: wall-clock time via `clock()` in seconds.
- Performance metrics:
  - FLOPs: $2n^3$ (multiply-add per element).
  - GFLOPS: $(2n^3/10^9)/t$.
  - Bandwidth: rough estimate $(3 \times n^2 \times 8\,\text{bytes})/t$ in GB/s (three matrices read/written).

- Compilation: `gcc -O2` (to enable reasonable optimizations without changing loop semantics too aggressively).
- Verification: both versions produce identical results (within 1e-10 tolerance).

Results

Table 2: Performance comparison for $n = 400$ matrices.

Version	Time (s)	GFLOPS	Bandwidth (GB/s)
Standard (i-j-k)	0.1390	0.92	0.03
Optimized (i-k-j)	0.0650	1.97	0.06
Speedup		2.14×	
Performance gain		+113.8%	
Bandwidth gain		+113.8%	

Both versions produce identical results (verified within floating-point tolerance).

Analysis

Loop Order and Locality

In row-major storage (C convention):

- **i-j-k (standard)**: For fixed i, j, inner loop over k accesses $a[i][k]$ (contiguous, good) but $b[k][j]$ (strided by row size, poor spatial locality) and writes single $c[i][j]$ elements repeatedly (poor temporal locality for C).
- **i-k-j (optimized)**: For fixed i, k, inner loop over j accesses full rows of $b[k][\cdot]$ and $c[i][\cdot]$ contiguously (excellent spatial locality). $a[i][k]$ is reused n times (good temporal locality) and hoisted to reduce loads.

This reordering transforms the access pattern from strided/column-major on B to row-major on B and C, significantly reducing cache misses and improving effective bandwidth.

Performance Impact

The optimized version achieves ≈2.14× speedup (113% gain) and doubles the effective bandwidth. This reflects better cache-line utilization and fewer memory stalls, even though the arithmetic intensity remains the same. The absolute GFLOPS (∼2) is low because:

- Single-threaded execution.
- No SIMD vectorization in the inner loop (compiler may not auto-vectorize perfectly).
- $n = 400$ matrices fit comfortably in modern L3 cache (∼8–32 MB), so gains come mostly from spatial locality and reduced loads rather than capacity misses.

Larger matrices ($n \gtrsim 1000$) or -O3 with explicit vectorization would amplify the difference.

Exercise 3: Blocked Matrix Multiplication

Objective

The goal is to implement blocked matrix multiplication to improve cache usage. By processing submatrices (blocks) of size $B \times B$, we keep data in fast cache memory longer, reducing slow main-memory accesses. We compare execution time, GFLOPS, and bandwidth across different block sizes to find the optimal B.

Methodology

- Matrix size: $n = 512$ (each matrix about 2 MB, total about 6 MB)
- Matrices: dynamically allocated arrays (row-major storage)
- Reference: non-blocked i-k-j loop order (from Exercise 2)
- Blocked version: six nested loops (outer loops over blocks; inner loops over elements within blocks)
- Block sizes tested: 8, 16, 32, 64, 128, 256
- Timing: wall-clock time using `clock()`
- Performance metrics:
  - FLOPs: $2n^3 \approx 268 \times 10^6$ operations
  - GFLOPS: $(2n^3/10^9)/t$ (billions of operations per second)
- Compilation: `gcc -O2`

Results

Table 3: Performance for $n = 512$ with different block sizes.

Block Size	Time (s)	Speedup vs Reference	GFLOPS
Reference (no blocking)	0.284	1.00×	0.94
8	0.062	4.58×	4.33
16	0.056	5.07×	4.79
32	0.047	6.04×	5.71
64	0.048	5.92×	5.59
128	0.046	6.17×	5.83
256	**0.043**	**6.60**×	**6.24**

The optimal block size is $B = 256$, achieving the fastest time (0.043 s) and best speedup (6.60 times faster than the reference).

Analysis

Blocking dramatically improves performance (up to 6.6 times faster) by keeping data in cache longer. Each block from matrices A and B is reused multiple times while computing the result block in C, all while staying in fast cache memory. The inner loops read consecutive memory locations, which helps the hardware prefetcher load data efficiently.

Why $B = 256$ is optimal:

The block size of 256 is the largest that fits comfortably in L2 or L3 cache (each block is about 0.5 MB, total active data about 1.5 MB for A/B/C blocks). This maximizes data reuse without exceeding cache capacity.

- **Smaller blocks (8–32):** Too much loop overhead compared to actual computation
- **Optimal blocks (128–256):** Best balance between cache reuse and overhead
- **Larger blocks (> 256):** Don't fit in cache, causing performance to drop

Note on larger matrices: For $n = 800$–1000, blocking showed less improvement (0.7–0.9×) because the reference i-k-j version already has good cache behavior for these sizes, and the blocked version adds extra loop overhead.

Exercise 4: Memory Management and Debugging with Valgrind

Objective

The goal is to find and fix memory leaks in a C program using Valgrind Memcheck. The program allocates, initializes, prints, and duplicates integer arrays. We use Valgrind to detect where memory leaks occur, understand why they happen, and verify that our fixes work. This exercise shows why proper memory management is important and how debugging tools can find errors that compilers miss.

Methodology

- Program tasks: allocate two arrays (original + copy), fill them with values (0, 10, 20, 30, 40), print both arrays, then free both
- Intentional bug: missing `free()` call for the copied array in `memory_debug_leaky.c`
- Compilation: `gcc -g -o memory_debug memory_debug.c` (the `-g` flag adds debug information for better Valgrind reports)
- Valgrind command: `valgrind -leak-check=full -show-leak-kinds=all -track-origins=ye ./executable`
- Testing steps:
  1. Run Valgrind on the fixed version (`memory_debug`)
  2. Run Valgrind on the leaky version (`memory_debug_leaky`)
  3. Fix the leak by adding `free_memory(array_copy);`
  4. Re-run Valgrind to confirm no leaks remain

Results

Fixed version (`memory_debug`) — No memory leaks:

```
Array elements: 0 10 20 30 40
Array elements: 0 10 20 30 40
==1678== HEAP SUMMARY:
==1678==     in use at exit: 0 bytes in 0 blocks
==1678==   total heap usage: 3 allocs, 3 frees, 1,064 bytes allocated
==1678== All heap blocks were freed -- no leaks are possible
==1678== ERROR SUMMARY: 0 errors from 0 contexts
```

Leaky version (`memory_debug_leaky`) — Two memory leaks detected:

```
Array elements: 0 10 20 30 40
Array elements: 0 10 20 30 40
==2238== HEAP SUMMARY:
==2238==     in use at exit: 40 bytes in 2 blocks
==2238==   total heap usage: 3 allocs, 1 frees, 1,064 bytes allocated
==2238== 20 bytes in 1 blocks are definitely lost in loss record 1 of 2
==2238==    at 0x...: malloc (...)
==2238==    by 0x...: allocate_array (memory_debug_leaky.c:8)
==2238==    by 0x...: main (memory_debug_leaky.c:48)
==2238== 20 bytes in 1 blocks are definitely lost in loss record 2 of 2
==2238==    at 0x...: malloc (...)
==2238==    by 0x...: duplicate_array (memory_debug_leaky.c:34)
==2238==    by 0x...: main (memory_debug_leaky.c:52)
==2238== LEAK SUMMARY:
==2238==    definitely lost: 40 bytes in 2 blocks
```

The fix: Adding `free_memory(array_copy);` before `return 0;` eliminates both leaks. After this fix, Valgrind reports no leaks (same as the fixed version).

Analysis

Valgrind found exactly where the memory leaks occurred:

- **First leak:** 20 bytes (one `int[5]` array) allocated in `allocate_array()` at line 8, never freed
- **Second leak:** 20 bytes allocated in `duplicate_array()` at line 34, never freed

Valgrind shows the complete path: which function allocated the memory (`malloc`), which function called it (`allocate_array` or `duplicate_array`), and where in `main()` it was called. The tool marks these as "definitely lost," meaning no pointer to this memory exists anymore.

Key Lessons

- **Every `malloc()` needs a `free()`:** All dynamically allocated memory must be freed, even in small programs

- **Functions that return pointers are risky:** It's easy to forget to free memory returned by functions like `duplicate_array()`
- **Debug symbols are essential:** Compiling with `-g` makes Valgrind show exact line numbers and function names
- **Valgrind is reliable:** It accurately detects memory leaks, invalid memory access, and use of uninitialized values without modifying your program

The fix in code:

```
// In main(), before return 0:
free_memory(array);        // Free original array
free_memory(array_copy);   // Fix: Free the copied array too
return 0;
```

Exercise 5: HPL Benchmark (High-Performance Linpack)

Objective

The goal is to run the HPL benchmark to measure performance when solving large systems of linear equations ($Ax = b$) with different matrix sizes N and block sizes NB. We record execution time, performance (GFLOPS), and verify correctness. Then we analyze how performance changes with problem size and blocking strategy.

Note on Hardware: Due to unavailability of Toubkal HPC access, this exercise was performed on a personal laptop with 4 MPI processes. Results differ from what would be obtained on the target server (Intel Xeon Platinum 8276L).

Theoretical Performance

For the target hardware (single core):

$$P_{\text{core}} = 2.2\,\text{GHz} \times 32\,\text{operations/cycle} = 70.4\,\text{GFLOPS}$$

For the actual hardware used (4 cores on laptop):

$$P_{\text{4cores}} = 4 \times 2.2\,\text{GHz} \times 32\,\text{operations/cycle} = 281.6\,\text{GFLOPS}$$

Methodology

- HPL version: 2.3
- Hardware: personal laptop
- Parallel processing: 4 MPI processes in a 2×2 grid ($P = 2$, $Q = 2$)
- Matrix sizes tested: $N = 1000, 2000, 4000, 8000$
- Block sizes tested: $NB = 32, 64, 128$
- Metrics: execution time (seconds), GFLOPS, correctness check
- Analysis: Python script (`plot_results.py`) for data processing and visualization

Results

Table 4: HPL performance summary (4 MPI processes on laptop). All tests passed correctness check.

N	NB	Time (s)	GFLOPS	Efficiency (%)
1000	32	5.19	0.129	0.05
1000	64	3.67	0.182	0.06
1000	128	2.59	0.258	0.09
2000	32	12.43	0.430	0.15
2000	64	8.08	0.661	0.23
2000	128	5.17	1.033	0.37
4000	32	25.19	1.695	0.60
4000	64	21.04	2.029	0.72
4000	128	15.23	2.803	1.00
8000	32	68.21	5.006	1.78
8000	64	55.29	6.175	2.19
8000	128	40.66	8.398	2.98

Automated Performance Analysis

A Python script analyzed all test results:

Peak Performance:

- Best result: 8.40 GFLOPS at $N = 8000$, $NB = 128$
- Theoretical maximum: 281.6 GFLOPS
- Efficiency: 2.98%

Performance Scaling with Matrix Size:

- $N = 1000 \rightarrow 2000$: 4.01× faster
- $N = 2000 \rightarrow 4000$: 2.71× faster
- $N = 4000 \rightarrow 8000$: 3.00× faster

Block Size Impact: $NB = 128$ consistently performed best:

- At $N = 1000$: 2.00× faster than $NB = 32$
- At $N = 2000$: 2.40× faster than $NB = 32$
- At $N = 4000$: 1.65× faster than $NB = 32$
- At $N = 8000$: 1.68× faster than $NB = 32$

Analysis

How Performance Changes with Matrix Size

Performance improves dramatically as N increases: from about 0.13–0.26 GFLOPS at $N = 1000$ to 5–8.4 GFLOPS at $N = 8000$ (a 30–40× improvement).

This happens because larger matrices use cache memory more efficiently. The computer does more calculations per unit of data loaded from memory, and setup/communication overhead becomes less significant.

Effect of Block Size

Larger block sizes consistently give better performance. At $N = 8000$: $NB = 128$ achieves 8.40 GFLOPS while $NB = 32$ achieves only 5.01 GFLOPS (1.68× speedup).

Larger blocks help because:

- Data stays in cache longer and gets reused multiple times
- Less frequent synchronization between processes
- Better CPU vectorization (processing multiple numbers at once)

Efficiency vs Theoretical Peak

Best result: 8.4 GFLOPS vs 281.6 GFLOPS theoretical → 2.98% efficiency.

Why is actual performance much lower?

- **Hardware limitations:** Laptop CPU lacks AVX-512, has lower frequency, and smaller cache than server processors
- **Insufficient parallelism:** Only 4 MPI processes don't fully utilize available cores
- **Communication overhead:** Time spent coordinating between processes
- **Memory bottleneck:** Some operations wait for data from memory
- **No tuning:** Default settings not optimized for this hardware

On the target server (Xeon Platinum 8276L with 56 cores), properly tuned HPL typically achieves 40–60% efficiency.