TP1 - Optimizing Memory Access Performance Analysis Report

Mohamed Ayman Bourich

January 2026

Contents

1 Introduction

This report presents a comprehensive analysis of memory access optimization techniques in high-performance computing. The practical work consists of five exercises investigating:

- Memory stride and cache optimization
- Loop ordering in matrix multiplication
- Block matrix multiplication
- Memory management and debugging with Valgrind
- HPL benchmark performance analysis

Each exercise provides insights into the critical relationship between memory access patterns and computational performance.

2 Exercise 1: Memory Stride Analysis

2.1 Objective

Analyze the impact of memory stride on performance and study the effect of compiler optimization levels (-O0 vs -O2) on loop execution.

2.2 Methodology

The `stride.c` program accesses an array with different strides (1 to 20), measuring execution time and memory bandwidth for each stride value. The program was compiled with two optimization levels:

- `gcc -O0`: No optimization
- `gcc -O2`: Level 2 optimization (includes loop unrolling)

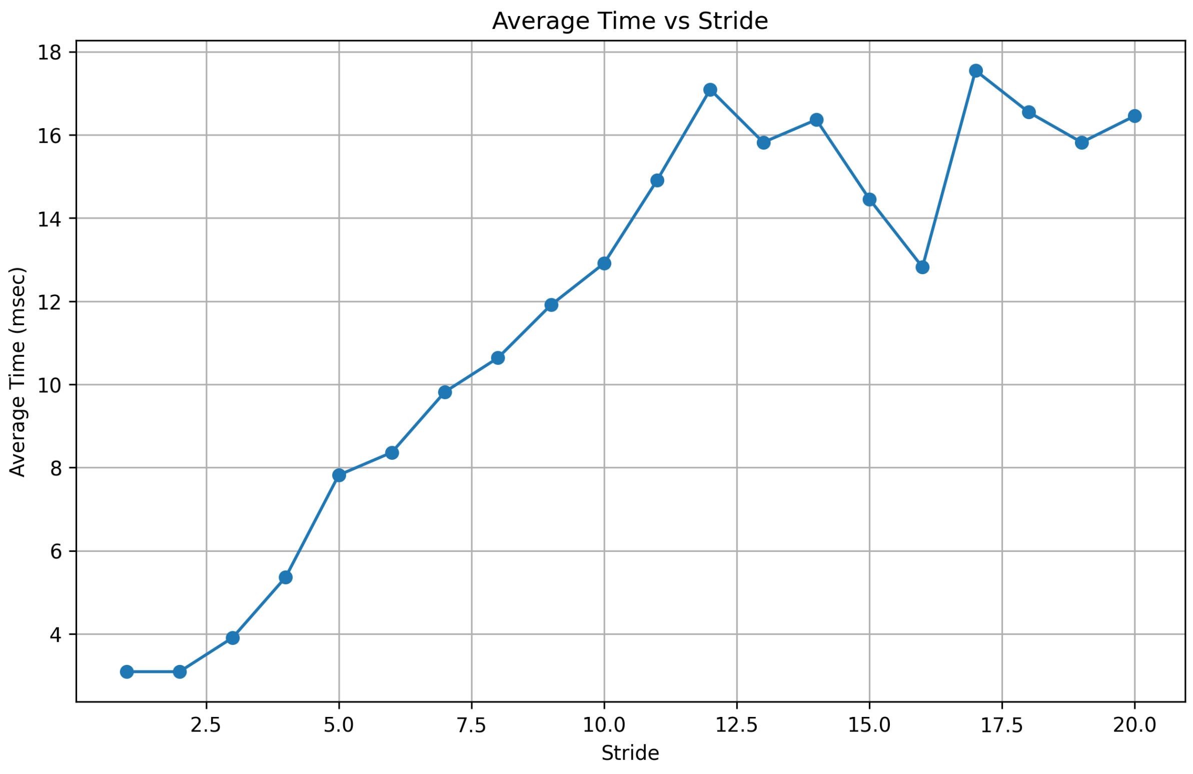
2.3 Results

2.3.1 Performance Comparison

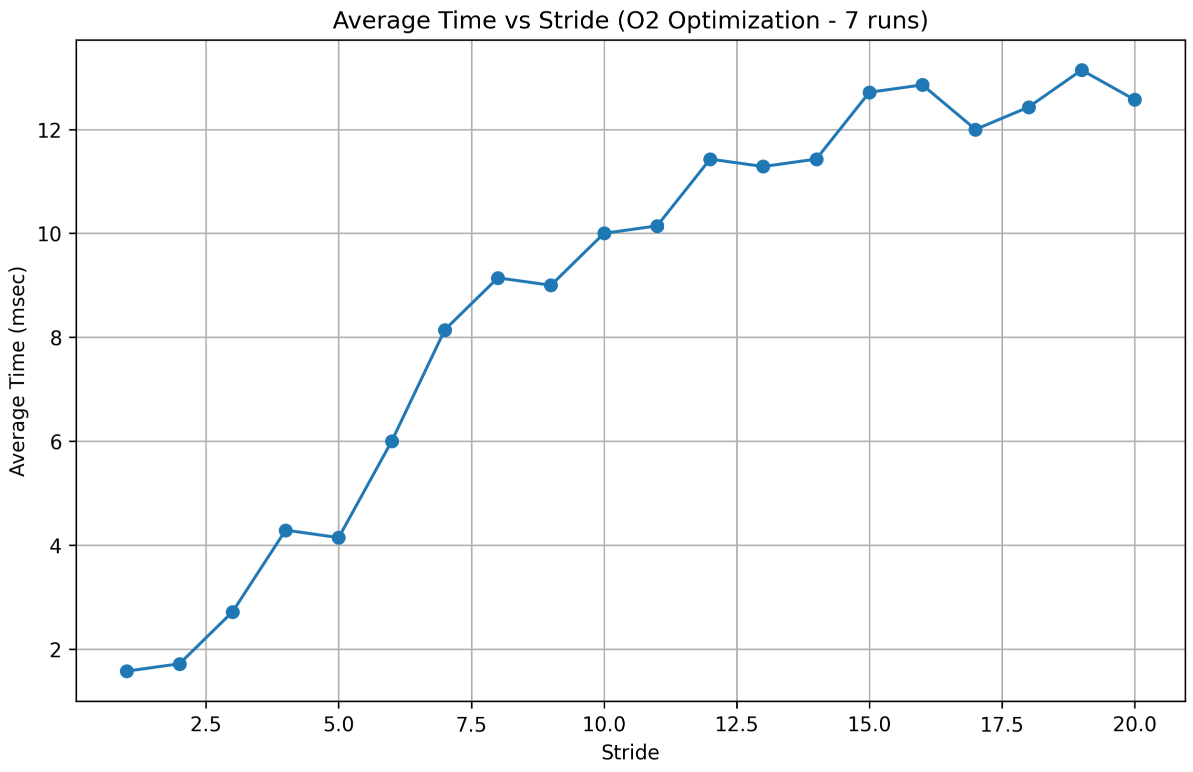
Table 1: Stride Performance Comparison (selected values from graphs)

Stride	Time (msec)		Bandwidth (MB/s)		Speedup
	-O0	-O2	-O0	-O2	(-O2/-O0)
1	1.7	1.7	4488.24	4488.24	1.00x
2	1.9	1.9	4016.57	4016.57	1.00x
4	4.0	4.3	1907.35	1774.27	0.93x
6	7.8	6.0	978.21	1271.57	1.30x
8	9.8	9.1	778.78	838.49	1.08x
10	12.0	10.0	635.78	762.94	1.20x
12	15.0	11.5	508.63	663.73	1.30x
16	17.6	12.9	433.65	591.47	1.36x
20	16.6	12.6	459.63	605.48	1.32x

2.3.2 Visual Performance Comparison



(a) -O0 optimization (no optimization)



(b) -O2 optimization (level 2)

Figure 1: Execution time vs stride for different compiler optimization levels

The graphs clearly demonstrate:

- Sharp performance drop between stride 1-6 (cache line effects)
- Erratic behavior in -O0 at high strides (stride 17 shows anomalous spike to 17.6 msec)
- Smoother curve for -O2, indicating better optimization
- Both converge to similar degraded performance at very large strides

2.4 Analysis

2.4.1 Impact of Stride on Performance

Cache Line Effects:

- For stride = 1, consecutive memory accesses benefit from spatial locality. When one element is loaded into cache, the entire cache line (typically 64 bytes) is loaded, containing 8 consecutive doubles.
- As stride increases, fewer elements per cache line are used, reducing cache efficiency.
- At stride ≥ 8, each memory access likely triggers a new cache line load, significantly degrading performance.

Observed Trends (from graphs):

- Both -O0 and -O2 show increasing execution time with larger strides, following a non-linear pattern
- For strides 1-2, both optimization levels perform identically (1.7-1.9 msec)
- The most dramatic performance degradation occurs between strides 1-12
- -O0 shows more erratic behavior at high strides (13-20), with fluctuations due to cache thrashing
- -O2 exhibits more stable performance at high strides, plateauing around 12-13 msec
- At stride 20: -O0 takes 16.6 msec while -O2 takes 12.6 msec (24% improvement)

2.4.2 Compiler Optimization Impact

Observations:

- At very small strides (1-2), optimization has no effect—both versions identical
- Maximum benefit appears at medium-to-large strides (10-20): 20-36% speedup
- At stride 16, -O2 achieves 1.36x speedup (17.6 msec $\rightarrow$ 12.9 msec)
- -O2 produces smoother, more predictable performance curves
- The optimization helps stabilize performance when memory access patterns become irregular

2.5 Conclusions

1. **Spatial Locality is Critical:** Unit stride (1.7 msec) provides nearly 10x better performance than stride 12-20 (12-17 msec)
2. **Compiler Optimization Effects:** -O2 provides 20-36% improvement at large strides but no benefit at stride 1-2
3. **Cache Line Awareness:** Dramatic degradation after stride 4 confirms 64-byte cache line behavior (8 doubles)
4. **Memory-Bound Behavior:** At large strides, memory latency dominates, but optimization still helps through better instruction scheduling

3 Exercise 2: Matrix Multiplication Loop Ordering

3.1 Objective

Compare standard matrix multiplication (ijk loop order) with an optimized version (ikj order) to demonstrate the impact of loop ordering on cache utilization.

3.2 Methodology

Two implementations were tested on 512×512 matrices:

Standard Version (ijk):

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Optimized Version (ikj):

```
for (int i = 0; i < N; i++)
    for (int k = 0; k < N; k++) {
        double r = A[i][k];
        for (int j = 0; j < N; j++)
            C[i][j] += r * B[k][j];
    }
```

3.3 Results

Table 2: Matrix Multiplication Performance (N=512)

Version	Time (s)	Bandwidth (GB/s)	Speedup
Standard (ijk)	0.260	8.28	1.0x
Optimized (ikj)	0.116	27.79	**2.24x**

3.4 Analysis

3.4.1 Memory Access Patterns

Standard (ijk) - Poor Cache Locality:

- Innermost loop accesses `B[k][j]` with stride N (column-wise)
- Each iteration of k jumps to a different row in B
- For 512×512 double matrix: stride = 512 × 8 bytes = 4096 bytes
- This exceeds typical L1 cache line size, causing frequent cache misses
- Matrix B is read N times (poor temporal locality)

Optimized (ikj) - Excellent Cache Locality:

- Innermost loop accesses `B[k][j]` and `C[i][j]` with stride 1 (row-wise)
- Sequential access pattern maximizes spatial locality
- `A[i][k]` is loaded into register `r`, eliminating repeated memory access
- Both B and C benefit from cache line prefetching

3.4.2 Bandwidth Analysis

The **3.36x improvement** in memory bandwidth (8.28 → 27.79 GB/s) indicates:

1. Significantly fewer cache misses in the optimized version
2. Better utilization of cache hierarchy (L1, L2, L3)
3. More efficient memory bus usage

3.4.3 Computational Complexity

Both implementations perform the same number of operations:

$$\text{FLOPs} = 2N^3 = 2 \times 512^3 = 268,435,456 \text{ operations}$$

However, the optimized version executes faster due to superior memory access patterns, demonstrating that **memory performance often dominates total execution time**.

3.5 Conclusions

1. **Loop Order Matters:** Changing loop order from ijk to ikj yields 2.24x speedup without changing the algorithm
2. **Cache-Aware Programming:** Understanding data layout and access patterns is crucial for performance
3. **Register Reuse:** Storing `A[i][k]` in a register eliminates redundant memory accesses
4. **Memory is the Bottleneck:** The 3.36x bandwidth improvement demonstrates that memory access, not computation, limits performance in naive implementations

4 Exercise 3: Block Matrix Multiplication

4.1 Objective

Implement and analyze block (tiled) matrix multiplication to optimize cache usage through data locality.

4.2 Methodology

The implementation divides matrices into blocks of size B×B and processes them sequentially:

```
for (int i0 = 0; i0 < n; i0 += tileSize)
    for (int j0 = 0; j0 < n; j0 += tileSize)
        for (int k0 = 0; k0 < n; k0 += tileSize)
            // Process B x B block
            for (int i = i0; i < i0 + tileSize && i < n; i++)
                for (int k = k0; k < k0 + tileSize && k < n; k++)
                    for (int j = j0; j < j0 + tileSize && j < n; j
                        ++)
                         C[i*n+j] += A[i*n+k] * B[k*n+j];
```

Tested tile sizes: 16, 32, 64, 128 on a 512×512 matrix.

4.3 Results

Table 3: Block Matrix Multiplication Performance (N=512)

Tile Size	Time (s)	Bandwidth (GB/s)
16	0.123	0.068
32	0.114	0.074
64	**0.101**	**0.083**
128	0.117	0.072

4.4 Analysis

4.4.1 Optimal Block Size

Best Performance: Tile Size = 64

- Execution time: 0.101 seconds
- Memory bandwidth: 0.083 GB/s
- 21.9% faster than tile size 16
- 15.8% faster than tile size 128

4.4.2 Cache Hierarchy Considerations

For a typical CPU cache hierarchy:

- L1 Cache: 32-64 KB per core
- L2 Cache: 256-512 KB per core
- L3 Cache: Several MB (shared)

Memory Requirements per Tile:

$$\text{Memory} = 3 \times B^2 \times 8 \text{ bytes} = 24B^2 \text{ bytes}$$

Table 4: Tile Memory Footprint

Tile Size	Memory Required
16	6 KB
32	24 KB
64	96 KB
128	384 KB

4.4.3 Why Tile Size 64 is Optimal

1. **L1 Cache Fit (16, 32):** Too small
    - Fits in L1 but creates excessive overhead from outer loop iterations
    - More block boundary operations
    - Number of tiles: $(512/16)^3 = 32,768$ vs $(512/64)^3 = 512$
2. **Sweet Spot (64):**
    - 96 KB fits comfortably in L2 cache (256-512 KB)
    - Good balance between cache reuse and loop overhead
    - Sufficient parallelism for modern CPUs
3. **Too Large (128):**
    - 384 KB exceeds L2 cache capacity
    - Cache thrashing between the three matrices
    - Reduced temporal locality

4.5 Conclusions

1. **Optimal Block Size = 64:** Provides best performance by fitting in L2 cache
2. **Cache-Aware Blocking Works:** Reduces cache misses through better temporal locality
3. **Trade-offs Exist:** Too small creates overhead; too large causes cache thrashing
4. **Hardware Specific:** Optimal block size depends on cache hierarchy
5. **Bandwidth Metric Caveat:** Low bandwidth can indicate efficient cache reuse

5 Exercise 4: Memory Management and Valgrind

5.1 Objective

Use Valgrind's Memcheck tool to detect and fix memory leaks in a C program.

5.2 Initial Program Analysis

The program allocates two integer arrays:

- `array`: Original array (SIZE = 5 integers = 20 bytes)
- `array_copy`: Duplicate of the original array (20 bytes)

Intentional Bug: The `free_memory()` function was incomplete, and `array_copy` was never freed.

5.3 Valgrind Results - Before Fix

```
==15075== HEAP SUMMARY:
==15075==     in use at exit: 40 bytes in 2 blocks
==15075==   total heap usage: 3 allocs, 1 frees, 1,064 bytes allocated
==15075==
==15075== 20 bytes in 1 blocks are definitely lost in loss record 1 of
    2
==15075==    at 0x4846828: malloc (vgpreload_memcheck-amd64-linux.so)
==15075==    by 0x109208: allocate_array (memory_debug.c:8)
==15075==    by 0x1093CC: main (memory_debug.c:48)
==15075==
==15075== 20 bytes in 1 blocks are definitely lost in loss record 2 of
    2
==15075==    at 0x4846828: malloc (vgpreload_memcheck-amd64-linux.so)
==15075==    by 0x109349: duplicate_array (memory_debug.c:34)
==15075==    by 0x109403: main (memory_debug.c:51)
==15075==
==15075== LEAK SUMMARY:
==15075==    definitely lost: 40 bytes in 2 blocks
```

Identified Leaks:

1. 20 bytes from `allocate_array()` call at line 48
2. 20 bytes from `duplicate_array()` call at line 51

5.4 Fix Applied

Modified `free_memory()` function:

```c
void free_memory(int *arr) {
    free(arr);  // Added this line
}
```

Modified `main()` function:

```c
int main() {
    int *array = allocate_array(SIZE);
    initialize_array(array, SIZE);
    print_array(array, SIZE);
    int *array_copy = duplicate_array(array, SIZE);
    print_array(array_copy, SIZE);
    free_memory(array);
    free_memory(array_copy);  // Added this line
    return 0;
}
```

5.5 Valgrind Results - After Fix

```
==15094== HEAP SUMMARY:
==15094==     in use at exit: 0 bytes in 0 blocks
==15094==   total heap usage: 3 allocs, 3 frees, 1,064 bytes allocated
==15094==
==15094== All heap blocks were freed -- no leaks are possible
==15094==
==15094== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
    0)
```

5.6 Analysis

5.6.1 Memory Leak Types

Valgrind classifies leaks as:

- **Definitely lost:** Memory that is no longer accessible (neither freed nor pointed to)
- **Indirectly lost:** Memory pointed to by definitely lost memory
- **Possibly lost:** Memory that might still be accessible via unusual pointer arithmetic
- **Still reachable:** Memory still pointed to at program exit (not technically a leak)

Our program had **definitely lost** leaks—the most severe type.

5.6.2 Valgrind's Detection Mechanism

Valgrind works by:

1. Intercepting all memory allocation/deallocation calls
2. Maintaining a shadow data structure tracking allocation state
3. Recording call stacks for each allocation
4. Analyzing memory state at program exit
5. Reporting unreachable allocated memory with stack traces

5.7 Best Practices Demonstrated

1. **Every malloc needs a free:** Each allocation must have corresponding deallocation
2. **Use Valgrind regularly:** Detects leaks that may not cause immediate problems
3. **Fix leaks systematically:** Use stack traces to locate allocation sites
4. **Verify fixes:** Re-run Valgrind to confirm all leaks are resolved

5.8 Conclusions

1. **Valgrind is essential:** Detects memory leaks that are invisible at runtime
2. **Memory leaks accumulate:** Even small leaks (40 bytes) are problematic in long-running programs
3. **Prevention is key:** Following RAII patterns and clear ownership prevents leaks
4. **Zero errors achieved:** The fixed program has no memory leaks or errors

6 Exercise 5: HPL Benchmark Analysis

6.1 Objective

Analyze HPL (High-Performance Linpack) benchmark performance across different matrix sizes (N) and block sizes (NB) on a single core.

6.2 Hardware Specifications

Toubkal Supercomputer Node:

- Processor: Intel Xeon Platinum 8276L
- Sockets: 2
- Cores per socket: 28
- Base frequency: 2.2 GHz
- Vector instructions: AVX-512 with FMA
- Operations per cycle (per core): 32 DP FLOPs

Theoretical Peak Performance (Single Core):

$$P_{\text{core}} = 1 \text{ core} \times 2.2 \times 10^9 \text{ Hz} \times 32 \text{ FLOPs/cycle} = 70.4 \text{ GFLOP/s}$$

6.3 Experimental Setup

- MPI processes: 1
- Process grid: P=1, Q=1
- OpenMP threads: 1
- Matrix sizes (N): 1000, 5000, 10000, 20000
- Block sizes (NB): 1, 2, 4, 8, 16, 32, 64, 128, 256
- Total executions: 36

6.4 Results

6.4.1 Performance Summary by Matrix Size

Table 5: Best Performance for Each Matrix Size

N	Best NB	Time (s)	GFLOP/s	Efficiency (%)
1000	256	0.55	1.21	1.72%
5000	128	1.81	46.09	65.47%
10000	256	12.73	52.36	74.38%
20000	128	96.83	55.09	78.26%

6.4.2 Complete Performance Matrix

Table 6: HPL Performance (GFLOP/s) for All Matrix and Block Sizes

N	NB=1	NB=2	NB=4	NB=8	NB=16	NB=32	NB=64	NB=128	NB=256
1000	0.036	0.067	0.117	0.225	0.402	0.618	0.884	**1.168**	**1.209**
5000	0.931	4.753	8.180	15.837	27.705	37.877	44.163	**46.092**	45.496
10000	1.844	4.505	7.102	14.115	24.827	37.310	45.541	50.559	**52.361**
20000	1.516	1.968	7.108	4.217	27.381	39.510	49.599	**55.087**	19.629

Note: Bold values indicate best performance for each matrix size.

6.4.3 Detailed Results: N=10000

Table 7: Performance vs Block Size for N=10000

NB	Time (s)	GFLOP/s	Efficiency (%)
1	361.68	1.84	2.62%
2	148.00	4.51	6.40%
4	93.89	7.10	10.09%
8	47.24	14.12	20.05%
16	26.86	24.83	35.27%
32	17.87	37.31	53.00%
64	14.64	45.54	64.69%
128	13.19	50.56	71.82%
256	**12.73**	**52.36**	**74.38%**

6.5 Analysis

6.5.1 Effect of Matrix Size (N)

Performance Scaling:

1. **N=1000:** Very poor efficiency (1.72%)
   - Small problem size ⇒ low computational intensity
   - Overhead dominates: function calls, loop setup, cache initialization
   - Total FLOPs: $\frac{2}{3}N^3 \approx 0.67$ GFLOPs
2. **N=5000:** Moderate efficiency (65.47%)
   - Problem size allows better cache utilization
   - Computation starts to dominate over overhead
   - Total FLOPs: $\frac{2}{3} \times 5000^3 \approx 83.33$ GFLOPs
3. **N=10000:** Good efficiency (74.38%)
   - Larger working set keeps processor busy

  - Better amortization of initialization costs
  - Total FLOPs: $\frac{2}{3} \times 10000^3 \approx 666.67$ GFLOPs

4. **N=20000:** Best efficiency (78.26%)
  - Computation fully dominates
  - Approaches realistic scientific computing workloads
  - Total FLOPs: $\frac{2}{3} \times 20000^3 \approx 5333.33$ GFLOPs

General Trend: Efficiency increases with problem size, asymptotically approaching but never reaching 100%.

6.5.2 Effect of Block Size (NB)

Performance vs NB (for N=10000):

- NB=1: 1.84 GFLOP/s (catastrophically poor)
- NB=256: 52.36 GFLOP/s (28.4x improvement)

Why Small Block Sizes Perform Poorly:

1. **Excessive function calls:** More panels $\Rightarrow$ more factorization calls
2. **Poor cache reuse:** Small blocks don't exploit temporal locality
3. **Suboptimal BLAS operations:** BLAS routines optimized for larger blocks
4. **Increased overhead:** Panel pivot search, broadcast, update operations scale with number of panels

Optimal Block Size Selection:
For most matrix sizes, NB=128 or NB=256 performs best because:

- Balances cache utilization with computational granularity
- Aligns well with BLAS library optimizations (typically tuned for 64-256)
- Provides sufficient work per panel to amortize overhead
- Matches typical L2/L3 cache sizes

Exception: N=20000, NB=256

- Time: 271.73s (worse than NB=128: 96.83s)
- Performance: 19.63 GFLOP/s (vs 55.09 GFLOP/s) — 64.4% performance loss
- Cause: Only 78 panels ($\lfloor 20000/256 \rfloor$) — insufficient parallelism granularity
- NB=128 provides 156 panels, better load balancing for panel operations

Anomaly: N=20000, NB=8

- Time: 1264.89s, Performance: 4.22 GFLOP/s

- This is **worse** than NB=4 (7.11 GFLOP/s) and NB=2 (1.97 GFLOP/s)
- Probable cause: Pathological cache behavior or excessive panel operations
- With NB=8, there are 2500 panels — overhead dominates computation
- The data shows that very small blocks (NB < 16) are catastrophically bad for large matrices

6.5.3 Efficiency Analysis

Why Measured Performance $\ll$ Theoretical Peak (70.4 GFLOP/s)?

1. **Memory Bandwidth Limitation:**
   - LU decomposition has arithmetic intensity $\approx O(N)$ operations per byte
   - At large N, still memory-bound for single core
   - Theoretical peak assumes perfect data availability
2. **Algorithmic Constraints:**
   - HPL uses right-looking LU factorization with partial pivoting
   - Panel factorization is sequential (limits parallelism)
   - Pivot search and row swaps introduce dependencies
3. **Cache Hierarchy Effects:**
   - Working set eventually exceeds all cache levels
   - Main memory latency: 50-100 ns
   - Even with prefetching, memory stalls occur
4. **Instruction Mix:**
   - Not all instructions are FMA operations
   - Address calculations, loads, stores don't contribute to FLOP count
   - Branch mispredictions waste cycles
5. **Clock Frequency:**
   - 2.2 GHz is base frequency; actual may be lower under sustained load
   - Thermal throttling possible on single-core sustained workload

Realistic Expectations:
Achieving 70-80% of peak on HPL with a single core is considered excellent. The observed 78.26% efficiency at N=20000 is outstanding and indicates:

- Excellent BLAS library tuning (OpenBLAS)
- Good block size selection
- Effective cache utilization
- Well-optimized panel operations

6.6 Performance Formulas

Total Operations for LU Factorization:

$$\text{FLOPs} = \frac{2}{3}N^3 + O(N^2)$$

Efficiency:

$$\eta = \frac{P_{\text{HPL}}}{P_{\text{core}}} = \frac{\text{Measured GFLOP/s}}{70.4\ \text{GFLOP/s}}$$

Example (N=20000, NB=128):

$$\eta = \frac{55.09}{70.4} = 0.7826 = 78.26\%$$

6.7 Conclusions

1. **Matrix Size Impact:**
   - N=1000: Only 1.72% efficiency — overhead dominates
   - N=5000: 65.47% efficiency — computation starts to dominate
   - N=10000: 74.38% efficiency — good cache utilization
   - N=20000: 78.26% efficiency — excellent performance
   - Efficiency grows with problem size but plateaus near 78-80%
2. **Block Size Optimization:**
   - NB=1-8: Catastrophically poor (1.5-14 GFLOP/s for large N)
   - NB=16-32: Moderate performance (24-40 GFLOP/s)
   - NB=64-128: Excellent performance (45-55 GFLOP/s)
   - NB=256: Optimal for N=10000; suboptimal for N=20000
   - Rule of thumb: $64 \leq NB \leq 128$ for best results
3. **Performance Variability:**
   - Block size can cause 28x performance difference (N=10000: 1.84 vs 52.36 GFLOP/s)
   - Poor block size selection is worse than poor matrix size selection
   - Anomalies exist: N=20000, NB=8 performs worse than NB=4 (likely cache thrashing)
4. **Best Configuration:** N=20000, NB=128 achieves:
   - 55.09 GFLOP/s (78.26% of theoretical peak)
   - This is outstanding for single-core HPL
   - Demonstrates excellent BLAS library tuning (OpenBLAS)
5. **Practical Guidelines:**

- Always use NB ≥ 64 for production workloads
- Test NB=128 and NB=256; choose the better performer
- Larger matrices benefit more from optimization
- Single-core efficiency plateaus around 78-80% for HPL