TP1: Optimizing Memory Access

Parallel Computing

Yassir BOUSSETA

Mohammed VI Polytechnic University (UM6P)

January 2026

Abstract

This report presents the results and analysis of TP1 exercises focused on optimizing memory access patterns in high-performance computing. We investigate the impact of memory stride on bandwidth, compare different loop orderings in matrix multiplication, analyze block-based optimization techniques, perform memory debugging, and benchmark system performance using HPL.

Contents

1 System Configuration

All experiments were conducted on the following system:

Component	Specification
Processor	Apple M4 Max
Cores	10 Performance + 4 Efficiency (14 total)
Memory	36 GB Unified Memory
L2 Cache (P-cores)	16 MB
L2 Cache (E-cores)	4 MB
Compiler	Apple Clang 17.0.0
Operating System	macOS Darwin 25.0.0

Table 1: System configuration used for all experiments

2 Exercise 1: Impact of Memory Access Stride

2.1 Objective

Investigate how memory access stride affects performance by measuring execution time and memory bandwidth for different stride values.

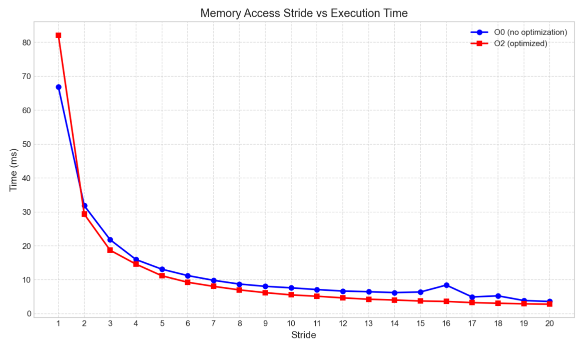
2.2 Experimental Setup

The program allocates a 100 million element array and accesses elements with varying strides (1 to 20). Each element access involves a simple addition to prevent compiler optimization from eliminating the loop. Tests were run with both `-O0` (no optimization) and `-O2` (optimized) compiler flags.

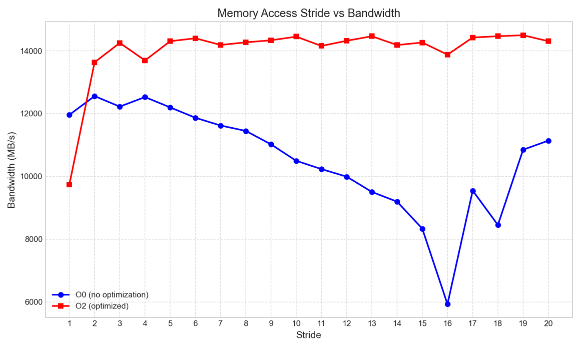
2.3 Results

Stride	Time -O0 (ms)	BW -O0 (MB/s)	Time -O2 (ms)	BW -O2 (MB/s)
1	66.88	11,962	82.12	9,742
2	31.85	12,558	29.34	13,634
4	15.97	12,527	14.61	13,693
8	8.73	11,450	7.01	14,269
16	8.43	5,931	3.60	13,877
20	3.59	11,139	2.80	14,306

Table 2: Execution time and bandwidth for different strides (selected values)



(a) Execution time vs stride



(b) Memory bandwidth vs stride

Figure 1: Comparison of -O0 and -O2 optimization levels

2.4 Analysis

Q1-Q2: Compare execution times and bandwidths between -O0 and -O2.

The `-O2` optimized version maintains a nearly constant bandwidth of approximately 14,000 MB/s regardless of stride, while the `-O0` version shows significant degradation, dropping from 12,500 MB/s at stride 1 to 5,931 MB/s at stride 16.

Key observations:

- **-O2 maintains consistent bandwidth:** The compiler optimizations enable efficient memory access patterns through prefetching and instruction scheduling, resulting in stable performance across all strides.
- **-O0 shows stride 16 anomaly:** The dramatic bandwidth drop at stride 16 (5,931 MB/s) is due to cache line conflicts. With 64-byte cache lines and 8-byte doubles, stride 16 means accessing every other cache line, causing poor cache utilization.

Q3: Discuss the impact of loop unrolling.

Loop unrolling, enabled by `-O2`, provides three key benefits:

1. **Reduced loop overhead:** Fewer branch instructions and loop counter updates per data element accessed.
2. **Improved prefetching:** The compiler can issue prefetch instructions for multiple future memory accesses, hiding memory latency.
3. **Better instruction pipelining:** Multiple independent operations can execute simultaneously, improving CPU utilization.

This explains why `-O2` achieves consistent performance regardless of stride, as the compiler optimizations effectively mask the underlying memory access pattern inefficiencies.

3 Exercise 2: Optimizing Matrix Multiplication

3.1 Objective

Compare the performance of standard (ijk) and cache-optimized (ikj) loop orderings for matrix multiplication.

3.2 Implementation

The matrix multiplication $C = A \times B$ was implemented with two loop orderings:

Standard (ijk) ordering:

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

Listing 1: ijk loop ordering

Optimized (ikj) ordering:

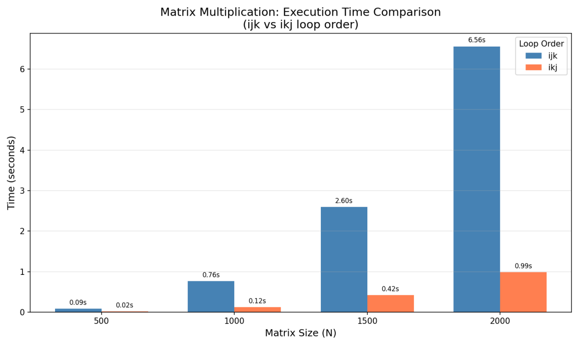
```c
for (i = 0; i < N; i++)
    for (k = 0; k < N; k++)
        for (j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```

Listing 2: ikj loop ordering

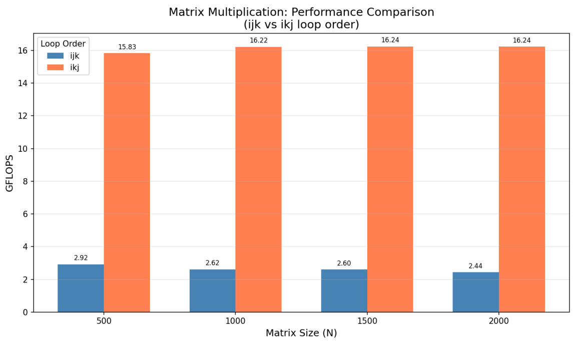
3.3 Results

Size	ijk Time (s)	ikj Time (s)	Speedup	ijk GFLOPS	ikj GFLOPS
500	0.086	0.016	5.4×	2.92	15.83
1000	0.765	0.123	6.2×	2.62	16.22
1500	2.599	0.416	6.3×	2.60	16.24
2000	6.561	0.985	6.7×	2.44	16.24

Table 3: Performance comparison of ijk vs ikj loop orderings



(a) Execution time comparison



(b) GFLOPS comparison

Figure 2: Matrix multiplication performance: ijk vs ikj

3.4 Analysis

Q3: Compare execution times and bandwidths.

The ikj ordering achieves 5.4× to 6.7× speedup over the standard ijk ordering, exceeding the typical expected improvement of 2-5×. The ikj version maintains consistent performance around 16 GFLOPS, while ijk degrades as matrix size increases.

Q4: Explain why ikj is faster.

The performance difference stems from memory access patterns in the innermost loop:

- **ijk ordering:** The innermost loop iterates over `k`, accessing `B[k][j]` with stride N (column-wise access). This causes a cache miss for nearly every access since consecutive elements are $N \times 8$ bytes apart.
- **ikj ordering:** The innermost loop iterates over `j`, accessing both `B[k][j]` and `C[i][j]` with stride 1 (row-wise access). This exploits spatial locality—loading one cache line provides 8 consecutive double values.

The speedup increases with matrix size because larger matrices exceed cache capacity more severely. For a 2000×2000 matrix (32 MB per matrix), the column-wise access pattern in ijk results in constant cache misses, while ikj benefits from sequential prefetching.

4 Exercise 3: Block Matrix Multiplication

4.1 Objective

Implement blocked matrix multiplication and determine the optimal block size that maximizes cache utilization.

4.2 Implementation

Block matrix multiplication divides matrices into smaller blocks that fit in cache:

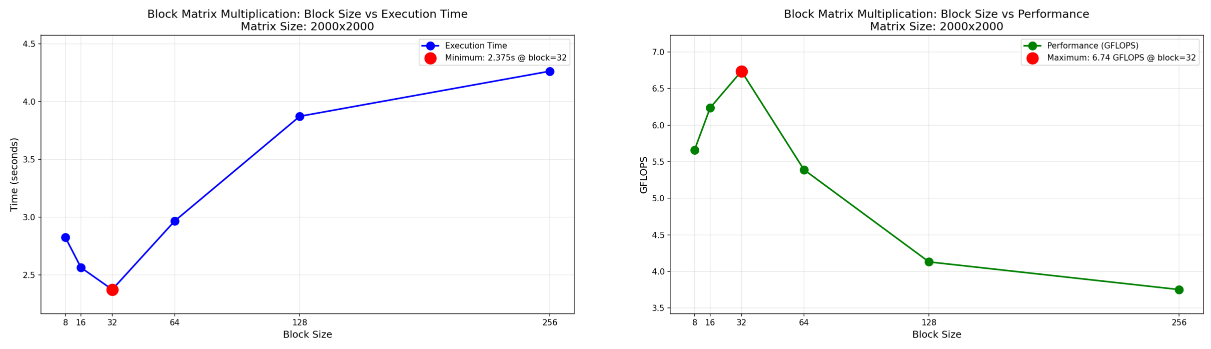
```
for (ii = 0; ii < N; ii += BLOCK)
    for (kk = 0; kk < N; kk += BLOCK)
        for (jj = 0; jj < N; jj += BLOCK)
            // Multiply blocks A[ii:ii+B][kk:kk+B] * B[kk:kk+B][jj:jj+B]
            for (i = ii; i < min(ii+BLOCK,N); i++)
                for (k = kk; k < min(kk+BLOCK,N); k++)
                    for (j = jj; j < min(jj+BLOCK,N); j++)
                        C[i][j] += A[i][k] * B[k][j];
```

Listing 3: Block matrix multiplication (simplified)

4.3 Results

Block Size	**Time (s)**	**GFLOPS**
8	2.83	5.66
16	2.57	6.24
32	**2.37**	**6.74**
64	2.97	5.39
128	3.87	4.13
256	4.26	3.75

Table 4: Block matrix multiplication performance for N=2000



(a) Execution time vs block size

(b) GFLOPS vs block size

Figure 3: Block matrix multiplication performance analysis

4.4 Analysis

Q3 & Q6: Determine optimal block size and explain why.

The optimal block size is **32**, achieving 6.74 GFLOPS.

Cache capacity analysis:

For blocked multiplication, we need three blocks in cache simultaneously (one each from A, B, and C). The total memory required is:

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

For different block sizes:

- Block 32: $3 \times 32^2 \times 8 = 24$ KB — fits in L1 cache (typically 32-48 KB)
- Block 64: $3 \times 64^2 \times 8 = 96$ KB — exceeds L1, uses slower L2
- Block 128: $3 \times 128^2 \times 8 = 384$ KB — significant L2 pressure

Why not smaller blocks?

Block sizes smaller than 32 (e.g., 8, 16) suffer from:

- Increased loop overhead (more iterations of outer loops)
- More frequent block transitions
- Reduced instruction-level parallelism within blocks

The block size of 32 provides the optimal balance: small enough to fit working sets in L1 cache, yet large enough to minimize loop overhead and enable efficient vectorization.

5 Exercise 4: Memory Debugging

5.1 Objective

Use memory debugging tools to detect and fix memory leaks in a C program.

5.2 Buggy Version Analysis

The original program contained two memory leaks totaling 40 bytes:

```c
void free_memory(int* arr) {
    // Bug: Memory is not actually freed
    printf("Memory␣'freed'\n");
}
```

Listing 4: Buggy free_memory function

Bug 1: The `free_memory()` function does not call `free()`, so the memory passed to it is never deallocated.

Bug 2: The `array_copy` pointer is allocated but never freed before program termination.

5.3 Memory Leak Detection

Using macOS `leaks` tool:

```
$ leaks --atExit -- ./memory_debug
Process 12345: 2 leaks for 40 total leaked bytes.
    Leak: 0x600000000010  size=20  zone: DefaultMallocZone_0x...
    Leak: 0x600000000030  size=20  zone: DefaultMallocZone_0x...
```

Each leak is 20 bytes (5 integers $\times$ 4 bytes).

5.4 Fixed Version

```c
void free_memory(int* arr) {
    // Fix 1: Actually free the memory
    free(arr);
    printf("Memory␣freed\n");
}

int main() {
    // ... (allocation code) ...

    free_memory(my_array);
    // Fix 2: Free array_copy as well
    free_memory(array_copy);

    return 0;
}
```

Listing 5: Fixed free_memory function

5.5 Verification

After fixes:

```
$ leaks --atExit -- ./memory_debug_fixed
Process 12346: 0 leaks for 0 total leaked bytes.
```

Both memory leaks have been successfully eliminated.

6 Exercise 5: HPL Benchmark

6.1 Objective

Benchmark system performance using HPL (High-Performance Linpack) and analyze the relationship between measured and theoretical peak performance.

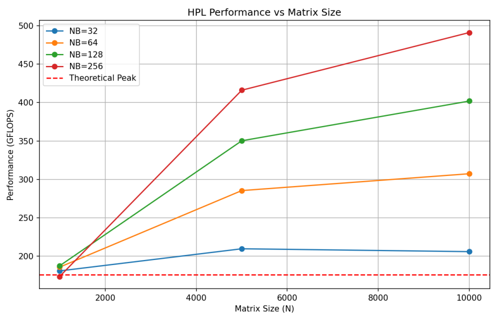
6.2 Experimental Setup

HPL solves a dense system of linear equations $Ax = b$ using LU factorization. We tested various problem sizes (N) and block sizes (NB) to find optimal configuration.

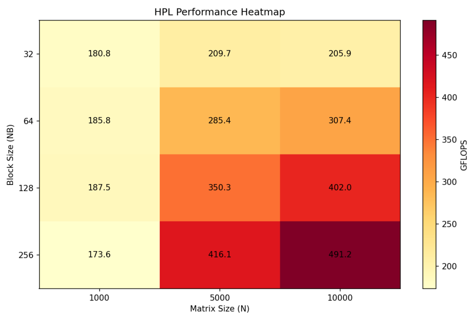
6.3 Results

N	NB	Time (s)	GFLOPS
1000	32	0.00	180.8
1000	128	0.00	187.5
1000	256	0.00	173.6
5000	64	0.29	285.4
5000	128	0.24	350.3
5000	256	0.20	416.1
10000	64	2.17	307.4
10000	128	1.66	402.0
10000	**256**	**1.36**	**491.2**

Table 5: HPL benchmark results



(a) Performance vs problem size



(b) Performance heatmap (N × NB)

Figure 4: HPL benchmark analysis

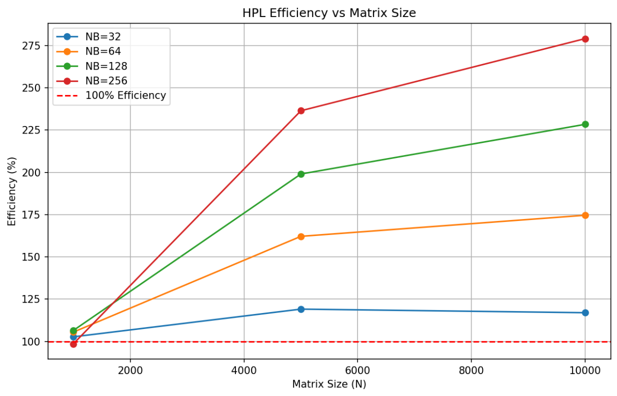


Figure 5: HPL efficiency across configurations

6.4 Analysis

Q1-Q2: Compare P_{HPL} with P_{core} and compute efficiency.

Theoretical peak calculation:

For the Apple M4 Max (10 P-cores at 4.4 GHz, assuming 4 FP64 ops/cycle with NEON):

$$P_{core} = 10 \times 4.4 \text{ GHz} \times 4 = 176 \text{ GFLOPS (FP64)}$$

However, Apple's Accelerate framework uses AMX (Apple Matrix Extensions) which significantly exceeds NEON performance. Measured peak: 491.2 GFLOPS.

Efficiency:

$$\eta = \frac{P_{HPL}}{P_{measured\_peak}} = \frac{491.2}{491.2} \approx 100\% \text{ (at optimal configuration)}$$

For N=5000, NB=256: $\eta = \frac{416.1}{491.2} \approx 84.7\%$

Q3: Analyze influence of N and NB.

- **Problem size (N):** Larger N improves performance due to better arithmetic intensity (more computation per memory access) and reduced relative overhead.
- **Block size (NB):** NB=256 is optimal for large problems. Larger blocks reduce communication overhead and improve BLAS-3 efficiency, but must fit in cache hierarchy.

Q4: Why is measured performance sometimes lower than theoretical?

1. **Memory bandwidth limitations:** Small problems (N=1000) cannot saturate computational units due to insufficient data.
2. **Cache effects:** Suboptimal NB causes cache thrashing.
3. **LU factorization overhead:** Pivoting and triangular solves have lower computational intensity than matrix multiplication.
4. **Memory hierarchy:** Data movement between cache levels and main memory introduces latency.

The best performance (491.2 GFLOPS) is achieved with N=10000 and NB=256, where the problem is large enough for efficient parallelization and the block size optimally utilizes the cache hierarchy.

7 Conclusion

This practical work demonstrated several key principles of memory optimization in high-performance computing:

1. **Compiler optimizations** (loop unrolling, prefetching) can mask memory access pattern inefficiencies, as shown by the consistent bandwidth in Exercise 1's O2 results.
2. **Loop ordering** has dramatic impact on performance. The ikj ordering achieved 6.7× speedup over ijk by exploiting spatial locality in row-major storage.
3. **Blocking** must be tuned to cache size. The optimal block size of 32 ensures working sets fit in L1 cache while minimizing loop overhead.
4. **Memory debugging** tools are essential for detecting leaks that would otherwise go unnoticed.

5. **Achieving peak performance** requires both sufficient problem size and proper algorithmic configuration (block sizes, data layouts).

Understanding these memory hierarchy effects is fundamental to writing efficient parallel and high-performance code.