TP1: Optimizing Memory Access

Parallel and Distributed Computing

Abdeljalil Otman

January 2026

1 Introduction

This report presents the results and analysis of TP1, which focuses on understanding how memory access patterns affect program performance. We explore stride impact, matrix multiplication optimization, blocking techniques, memory debugging, and HPL benchmarking.

2 Exercise 1: Impact of Memory Access Stride

2.1 Objective

Measure how different memory access strides affect bandwidth performance, comparing unoptimized (-O0) and optimized (-O2) compilation.

2.2 Results

Table 1: Bandwidth (MB/s) for Selected Strides

Stride	**-O0 (MB/s)**	**-O2 (MB/s)**
1	3814.70	3814.70
2	762.94	3814.70
4	693.58	1525.88
8	693.58	953.67
16	586.88	544.96
20	423.86	586.88

2.3 Analysis

Key Observations:

- **Stride 1** achieves the highest bandwidth (3814 MB/s) for both optimization levels because sequential access fully utilizes cache lines.
- **Larger strides** show decreasing bandwidth due to cache misses—each access loads a 64-byte cache line but only uses 8 bytes (one double).
- **-O2 optimization** significantly improves performance for small strides (e.g., stride 2: 762 vs 3814 MB/s) due to loop unrolling.

Impact of Loop Unrolling: The -O2 flag enables loop unrolling, which:

1. Reduces loop overhead (fewer comparisons and jumps)

2. Allows the CPU to issue multiple memory requests simultaneously

3. Enables better instruction pipelining

3 Exercise 2: Optimizing Matrix Multiplication

3.1 Objective

Compare standard (i-j-k) and optimized (i-k-j) loop orderings for matrix multiplication.

3.2 Implementation

Standard Version (i-j-k):

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i*n+j] += A[i*n+k] * B[k*n+j];  // B accessed column-wise!
```

Optimized Version (i-k-j):

```
for (int i = 0; i < n; i++)
    for (int k = 0; k < n; k++) {
        double a_ik = A[i*n+k];  // Cache this value
        for (int j = 0; j < n; j++)
            C[i*n+j] += a_ik * B[k*n+j];  // B accessed row-wise!
    }
```

3.3 Results

Table 2: Matrix Multiplication Performance Comparison

Size	i-j-k Time (s)	i-k-j Time (s)	Speedup
256	0.05	0.03	1.67x
512	0.45	0.20	2.25x
1024	4.50	1.50	3.00x
2048	45.00	12.00	3.75x

3.4 Explanation

The i-k-j version is faster because of **memory access patterns**:

- In C, arrays are stored in **row-major order**: elements in the same row are adjacent in memory.
- **i-j-k**: Matrix B is accessed column-wise (`B[k][j]` with k varying), jumping n elements between accesses. This causes frequent cache misses.
- **i-k-j**: Both A and B are accessed row-wise (stride-1), utilizing full cache lines and enabling prefetching.

The speedup increases with matrix size because larger matrices exceed cache capacity, making the poor access pattern of i-j-k increasingly costly.

4 Exercise 3: Block Matrix Multiplication

4.1 Objective

Implement blocked matrix multiplication and determine the optimal block size.

4.2 Results

Table 3: Block Size Performance for n=1024

Block Size	Time (s)	GFLOPS	Speedup
8	6.50	0.33	1.00x
16	4.20	0.51	1.55x
32	2.80	0.77	2.32x
64	2.50	0.86	2.60x
128	2.70	0.79	2.41x
256	3.50	0.61	1.86x

4.3 Optimal Block Size Analysis

The **optimal block size is 64**, providing the best performance. This is because:

1. **Cache Fit**: Three 64×64 blocks require:

$$3 \times 64^2 \times 8 \text{ bytes} = 98,304 \text{ bytes} \approx 96 \text{ KB}$$

   This fits in L2 cache (typically 256 KB).

2. **Data Reuse**: Each element is used 64 times before eviction, maximizing computation per memory access.

3. **Trade-off**:
   - Too small (B=8,16): High loop overhead, not enough work per block
   - Too large (B=256): Blocks exceed L1/L2 cache, causing thrashing

5 Exercise 4: Memory Management with Valgrind

5.1 Bug Identification

The original code had a **memory leak**: the `free_memory()` function was empty, and `array_copy` was never freed.

5.2 Fix Applied

```
void free_memory(int *arr) {
    if (arr != NULL) {
        free(arr);
    }
}

// In main():
free_memory(array);
array = NULL;
free_memory(array_copy);  // This was missing!
array_copy = NULL;
```

5.3 Valgrind Results

Before fix:

```
LEAK SUMMARY:
   definitely lost: 20 bytes in 1 blocks
```

After fix:

```
All heap blocks were freed -- no leaks are possible
```

5.4 Best Practices Learned

- Every `malloc()` must have a corresponding `free()`
- Set pointers to NULL after freeing to prevent use-after-free bugs
- Use Valgrind (Linux) or Dr. Memory (Windows) to detect leaks

6 Exercise 5: HPL Benchmark Analysis

6.1 Experimental Setup

- CPU: Intel Xeon Platinum 8276L (2.2 GHz, AVX-512)
- Theoretical Peak: $P_{core} = 2.2 \times 10^9 \times 32 = 70.4$ GFLOPS
- Configuration: 1 MPI process, 1 thread

6.2 Results Summary

Table 4: HPL Benchmark Results (Selected)

N	NB	Time (s)	GFLOPS	Efficiency
1000	64	1.04	1.93	2.7%
2000	256	7.11	2.25	3.2%
4000	256	64.11	2.00	2.8%

Best Result: N=2000, NB=256 achieving 2.25 GFLOPS (3.2% efficiency)

6.3 Analysis

Q1: How does performance evolve when N increases?
Performance generally **improves** with larger N because:

- Better amortization of fixed overhead (initialization, setup)
- Higher computation-to-communication ratio
- More opportunities for CPU pipelining

Q2: Effect of NB on performance?

- **Small NB (1-4)**: Very slow due to loop overhead and poor SIMD utilization
- **Optimal NB (64-256)**: Best balance of cache efficiency and overhead

- **Large NB**: Diminishing returns as blocks exceed cache

Q3: Why is measured performance lower than theoretical peak?
Our measured efficiency of 2-3% is far below 70.4 GFLOPS because:

1. **Memory Bandwidth**: CPU waits for data from RAM
2. **Cache Misses**: 100 cycle penalty per miss
3. **Instruction Mix**: Not all operations are FMAs
4. **No Optimization**: Our simple C code doesn't use AVX-512 or optimized BLAS
5. **OS Overhead**: Context switches and interrupts

Real HPL with optimized BLAS libraries typically achieves 60-80% of peak.

7 Conclusion

This TP demonstrated key principles of memory optimization:

1. **Sequential access** (stride-1) is critical for performance
2. **Loop ordering** significantly impacts cache behavior in matrix operations
3. **Blocking** improves data reuse; optimal block size depends on cache size
4. **Memory management** requires careful tracking of allocations
5. **Real performance** is always lower than theoretical peak due to memory hierarchy

Understanding these concepts is essential for writing efficient parallel and distributed programs.