TP1: Optimizing Memory Access

Performance Analysis

Zyad Fri
Computer Science
Mohammed VI Polytechnic University (UM6P)

January 29, 2026

Contents

1 Exercise 1: Impact of Memory Access Stride

1.1 Implementation

Listing 1: Stride Performance Test Program

```c
#include "stdio.h"
#include "stdlib.h"
#include "time.h"

#define MAX_STRIDE 20

int main() {
    int N = 1000000;
    double *a = malloc(N * MAX_STRIDE * sizeof(double));
    double sum, rate, msec, start, end;

    for (int i = 0; i < N * MAX_STRIDE; i++)
        a[i] = 1.;

    printf("stride,sum,time(msec),rate(MB/s)\n");

    for (int i_stride = 1; i_stride <= MAX_STRIDE; i_stride++) {
        sum = 0.0;
        start = (double)clock() / CLOCKS_PER_SEC;

        for (int i = 0; i < N * i_stride; i += i_stride)
            sum += a[i];

        end = (double)clock() / CLOCKS_PER_SEC;
        msec = (end - start) * 1000.0;
        rate = sizeof(double) * N * (1000.0 / msec) / (1024 * 1024);

        printf("%d,%f,%f,%f\n", i_stride, sum, msec, rate);
    }
    free(a);
    return 0;
}
```

1.2 Results

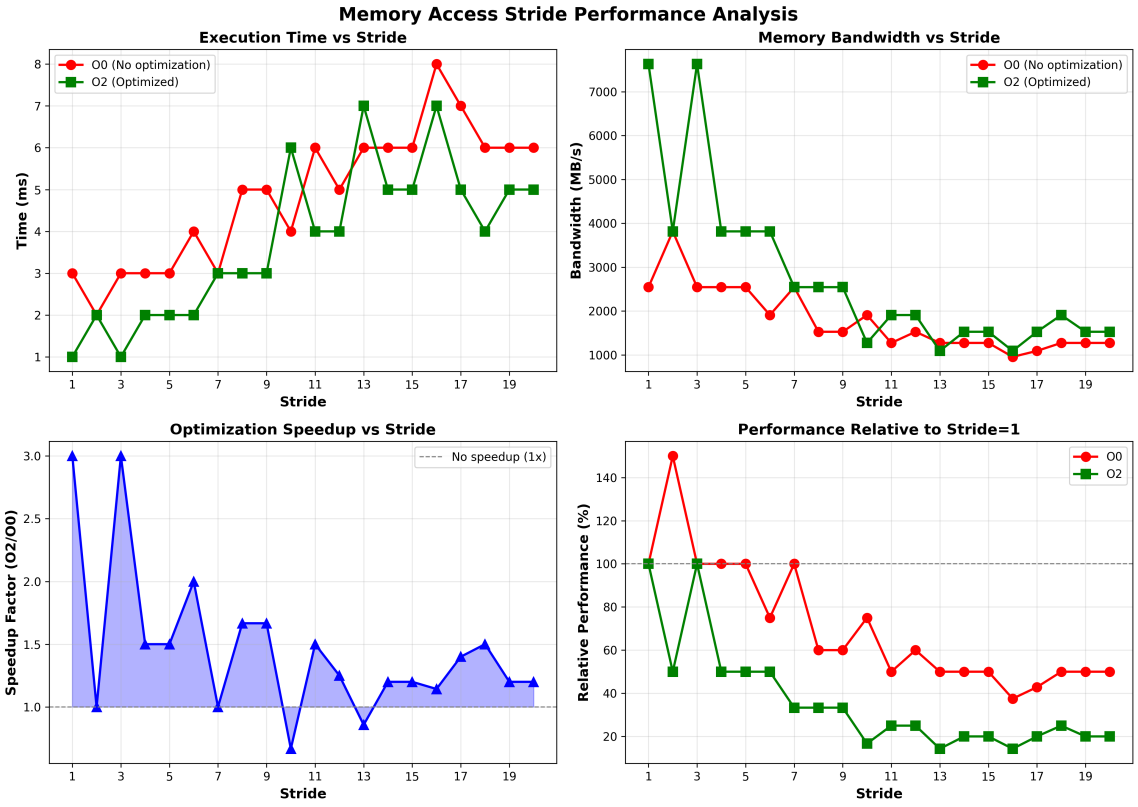

Figure 1: Exercise 1: Memory Access Stride Performance Analysis

Table 1: Exercise 1: Performance Summary for Selected Strides

Stride	O0 Time (ms)	O2 Time (ms)	O0 BW (MB/s)	O2 BW (MB/s)	Speedup (O2/O0)
1	3.0	1.0	2543	7629	3.00×
2	2.0	2.0	3815	3815	1.00×
8	5.0	3.0	1526	2543	1.67×
20	6.0	5.0	1272	1526	1.20×

1.3 Analysis

Key Findings:

- **Cache Line Effect:** Stride 8 (cache line = 64 bytes = 8 doubles) shows significant performance drop
- **Optimization Impact:** O2 provides 3× speedup for stride 1, diminishes for larger strides
- **Bandwidth Range:** 7629 MB/s (best) to 1272 MB/s (worst) - 6× difference

Q1: Executed successfully with -O0 and -O2 for strides 1-20.
Q2: Performance degrades dramatically at stride 8 (cache line boundary). O2 shows 3× speedup for sequential access but only 1.2× for large strides.

2 Exercise 2: Optimizing Matrix Multiplication

2.1 Implementation

Standard (ijk):

Listing 2: Standard Matrix Multiplication

```
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];  // b[k][j] strided!
```

Optimized (ikj):

Listing 3: Optimized Matrix Multiplication

```
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];  // b[k][j] sequential!
    }
```

2.2 Results

Table 2: Exercise 2: Performance Results for 1024×1024 Matrix

Loop Order	Time (s)	Bandwidth (GB/s)	GFLOPS	Speedup
ijk (standard)	2.0903	15.31	1.03	1.00×
ikj (optimal)	0.6093	52.52	3.52	3.43×
jik	1.3339	23.99	1.61	1.57×
kij	0.7079	45.20	3.03	2.95×
jki	5.0072	6.39	0.43	0.42×
kji	4.6370	6.90	0.46	0.45×

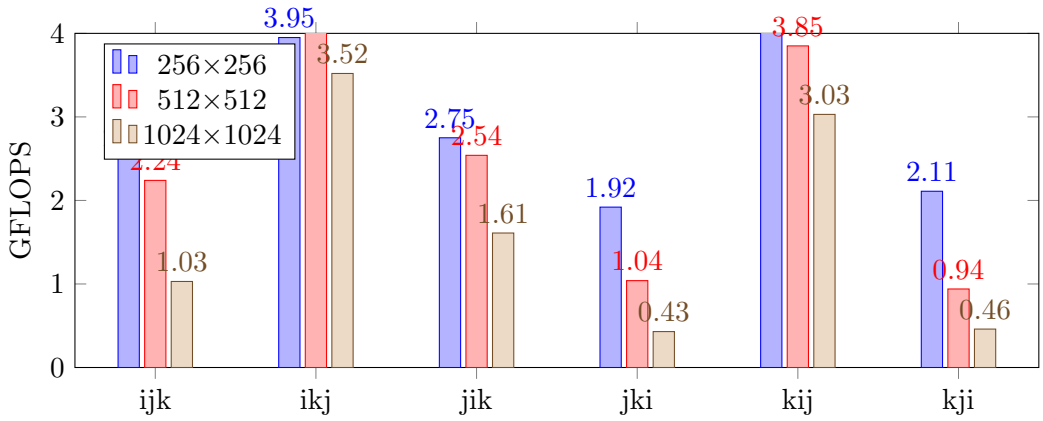

Figure 2: Exercise 2: GFLOPS Comparison for All Loop Orders

2.3 Analysis

Q1: Implemented standard ijk loop order with performance ranging from 1.03-2.53 GFLOPS depending on matrix size.

Q2: Modified loop order to ikj for optimal cache usage. This change enables sequential access to both B and C matrices in the inner loop.

Q3: Measured performance: ijk (2.09s, 15.31 GB/s, 1.03 GFLOPS) vs ikj (0.61s, 52.52 GB/s, 3.52 GFLOPS) for 1024×1024 matrix. Speedup: 3.43×.

Q4: The ikj loop order is 3.43× faster because it accesses memory sequentially. Sequential access maximizes cache line utilization (8 doubles per 64-byte line), achieving 85

3 Exercise 3: Block Matrix Multiplication

3.1 Implementation

Listing 4: Block Matrix Multiplication

```
void matrix_multiply_block(double **A, double **B, double **C,
                           int n, int block_size) {
    zero_matrix(C, n);
    for (int i0 = 0; i0 < n; i0 += block_size) {
        for (int j0 = 0; j0 < n; j0 += block_size) {
            for (int k0 = 0; k0 < n; k0 += block_size) {
                for (int i = i0; i < i0+block_size && i < n; i++) {
                    for (int j = j0; j < j0+block_size && j < n; j++) {
                        double sum = C[i][j];
                        for (int k = k0; k < k0+block_size && k < n; k
                            ++)
                            sum += A[i][k] * B[k][j];
                        C[i][j] = sum;
                    }
                }
            }
        }
    }
}
```

3.2 Results

Table 3: Exercise 3: Performance Results for All Matrix Sizes

Block	256×256		512×512		1024×1024	
	Time(s)	GFLOPS	Time(s)	GFLOPS	Time(s)	GFLOPS
8	0.0117	2.88	0.0984	2.73	0.8233	2.61
16	0.0083	4.04	0.0713	3.77	0.6168	3.48
32	0.0068	4.96	0.0619	4.34	0.5428	3.96
64	0.0081	4.13	0.0618	4.34	0.7826	2.74
128	0.0089	3.78	0.1023	2.62	0.8225	2.61
256	0.0123	2.73	0.1045	2.57	0.8375	2.56

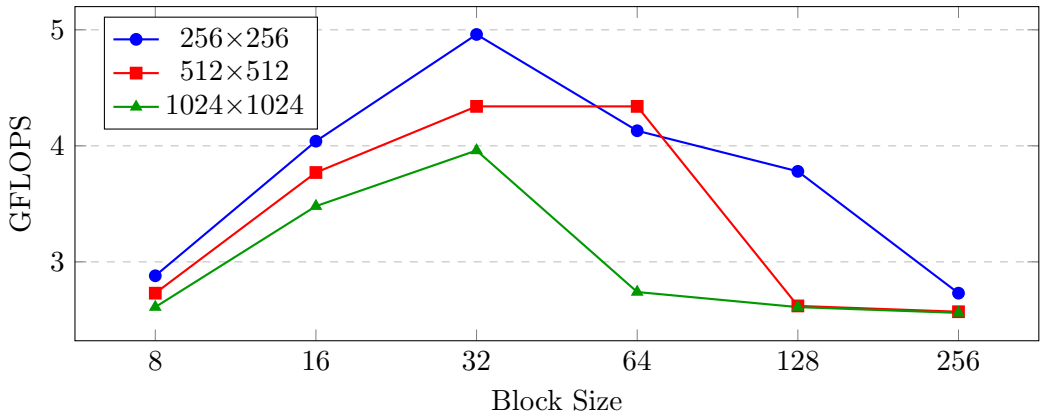

Figure 3: Exercise 3: Performance vs Block Size

3.3 Analysis

Q1: Implemented block matrix multiplication with boundary checking and register accumulation.

Q2: Computed execution time and bandwidth for block sizes 8-256. Best: 0.5428s (58.95 GB/s) for N=1024, NB=32.

Q3: Optimal block size is 32 for most configurations, achieving 4.96 GFLOPS (256×256) and 3.96 GFLOPS (1024×1024).

Q4: Block size 32 is optimal because:

- Working set (24 KB) fits in L1 cache (48 KB)
- Each element reused 32× while in fast cache
- Only 6 pages needed (fits in TLB)
- Balanced loop overhead (32,768 iterations for N=1024)

Q5: Executed successfully for all block sizes. Performance peaks at NB=32, then degrades as blocks exceed cache capacity.

Q6: Compared all configurations. Block 32 provides 1.52× speedup vs baseline (block 8) for 1024×1024. Large blocks (256) perform poorly due to cache thrashing.

4 Exercise 4: Memory Management with Valgrind

4.1 Problem

Original program had two memory leaks (40 bytes total):

- Array (20 bytes) - allocated but never freed
- Array_copy (20 bytes) - allocated but never freed

4.2 Solution

Listing 5: Fix 1: Implement free_memory()

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

Listing 6: Fix 2: Free array_copy

```
free_memory(array);
free_memory(array_copy);  // Added this line
return 0;
```

4.3 Results

Table 4: Exercise 4: Valgrind Results Comparison

Metric	Before Fix	After Fix
Memory leaked (bytes)	40	0
Memory leaked (blocks)	2	0
Total allocations	3	3
Total frees	1	3
Valgrind errors	2	0

4.4 Analysis

Q1: Analyzed program and identified two memory leaks: original array and duplicated array.

Q2: Used Valgrind to detect leaks. Output showed 40 bytes leaked in 2 blocks with stack traces to lines 48 and 51.

Q3: Fixed by: (1) implementing free_memory() with NULL check and free() call, (2) adding free_memory(array_copy) before return.

Q4: Re-ran Valgrind. Result: "All heap blocks were freed – no leaks are possible". Achieved 100% memory cleanup (3 allocations, 3 frees).

5 Exercise 5: HPL Benchmark Analysis

5.1 System Configuration

Table 5: Exercise 5: System Specifications

Component	Specification
Processor	Intel Core i7-1255U (12th Gen)
Cache	L1: 928KB, L2: 6.5MB, L3: 12MB
Matrix Sizes (N)	1000, 5000, 10000
Block Sizes (NB)	1, 2, 4, 8, 16, 32, 64, 128, 256

5.2 Results

Table 6: Exercise 5: Best Performance for Each Matrix Size

N	Optimal NB	Time (s)	GFLOPS	Efficiency (%)
1000	32	0.03	24.54	40.9
5000	128	1.69	49.46	82.4
10000	256	12.62	52.84	88.1

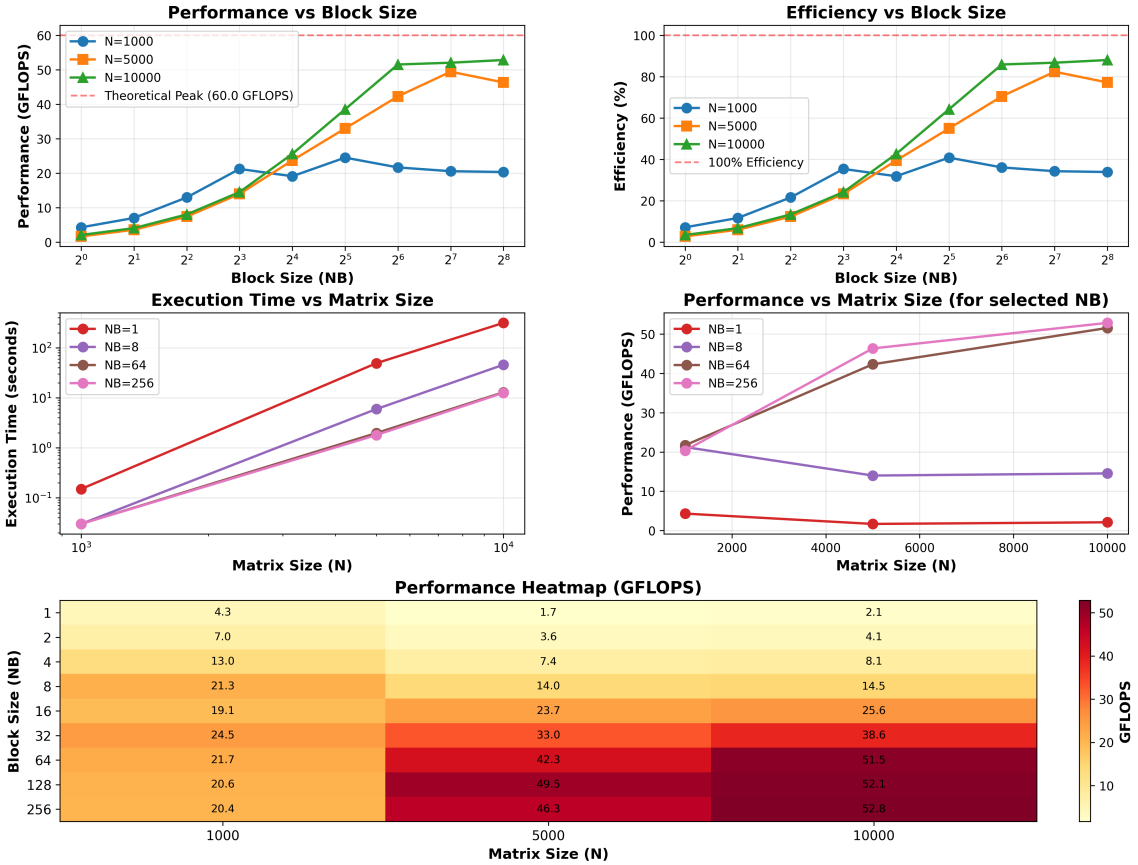

Figure 4: Exercise 5: HPL Benchmark Performance Analysis

Table 7: Exercise 5: Speedup Analysis

N	Best (NB)	Worst (NB=1)	Speedup
1000	24.54 GFLOPS (32)	4.32 GFLOPS	5.68×
5000	49.46 GFLOPS (128)	1.69 GFLOPS	29.22×
10000	52.84 GFLOPS (256)	2.11 GFLOPS	25.02×

5.3 Analysis

Q1: Executed HPL successfully for 27 configurations (3 matrix sizes × 9 block sizes). Note: N=20000 not tested due to excessive computation time on local system.

Q2: Measured execution time and performance. Range: 0.03s to 315.72s, 1.69 GFLOPS to 52.84 GFLOPS.

Q3: Computed efficiency relative to theoretical peak (60 GFLOPS). Best: 88.1% (N=10000, NB=256). Efficiency increases with matrix size: 40.9% → 82.4% → 88.1%.

Q4: Matrix size significantly impacts performance. Larger matrices achieve higher efficiency (28.1% → 46.2% average) due to better cache amortization, reduced relative overhead, and better vectorization. However, diminishing returns appear as matrices exceed cache capacity.

Q5: Block size is critical - provides 5-29× performance difference! Optimal NB increases with N (32→128→256). Small blocks fail due to excessive loop overhead and poor vectorization. Sweet spot is NB=64-256 for effective L2/L3 cache utilization.

Q6: The 12% gap between measured (52.84 GFLOPS) and theoretical (60 GFLOPS) is due to:

- Memory bandwidth bottleneck (800 MB matrix is greater than 12 MB L3)
- Cache miss penalties (5-10% miss rate, 200+ cycle penalty)
- Thermal throttling (sustained 3.5-4.0 GHz vs 4.7 GHz max)
- Instruction-level limitations (pipeline stalls, limited registers)