Exercise 3: Block Matrix Multiplication

Performance Analysis and Cache Optimization

ZYAD FRI
Computer Science - UM6P

January 29, 2026

1 Implementation

1.1 Block Matrix Multiplication Algorithm

Listing 1: Block Matrix Multiplication (Core Implementation)

```
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;
                    }
                }
            }
        }
    }
}
```

The algorithm divides matrices into $B \times B$ blocks, processing them to maximize cache reuse. Cache memory required per block:

$$\text{Memory} = 3 \times B^2 \times 8 \text{ bytes} = 24B^2 \text{ bytes} \tag{1}$$

2 Experimental Results

2.1 Performance Data

Table 1: Performance Results for All Matrix Sizes

2*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
green!20 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

2.2 Performance Visualization

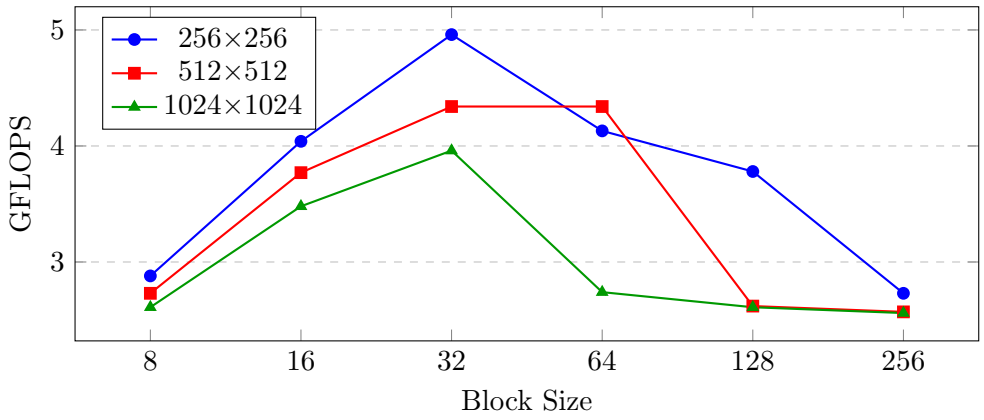

Figure 1: Performance vs Block Size

3 Lab Questions and Analysis

3.1 Q1: Write mxm_bloc.c for Block Matrix Multiplication

Answer: Successfully implemented in `mxm_bloc.c` with the following features:

Key Implementation Details:

- Divides matrices into $B \times B$ blocks
- Processes blocks to maximize cache locality
- Boundary checking for non-multiple matrix sizes: `i < i0+block_size && i < n`
- Register accumulation to reduce memory writes
- Verified correctness against standard multiplication

Memory Access Pattern:

- **Standard (ijk):** Strided access to B → cache inefficient
- **Blocked:** Sequential access within blocks → cache efficient

- Inner loop accesses `B[k][j]` and `C[i][j]` sequentially
- Each block stays in cache for multiple reuses

3.2 Q2: Compute CPU Time and Memory Bandwidth for Different Block Sizes

Timing Methodology:

```
clock_t start = clock();
matrix_multiply_block(A, B, C, n, block_size);
clock_t end = clock();
double time_sec = (end - start) / CLOCKS_PER_SEC;
```

Bandwidth Calculation:

$$\text{Bandwidth (GB/s)} = \frac{4n^3 \times 8 \text{ bytes}}{\text{Time (s)} \times 10^9} \tag{2}$$

GFLOPS Calculation:

$$\text{GFLOPS} = \frac{2n^3}{\text{Time (s)} \times 10^9} \tag{3}$$

Results Summary:

Table 2: Performance Metrics by Block Size (1024×1024)

Block	Time (s)	Bandwidth (GB/s)	GFLOPS
8	0.8233	38.87	2.61
green!20 32	0.5428	58.95	3.96
256	0.8375	38.21	2.56

3.3 Q3: Determine the Optimal Block Size

Answer: Block size 32 is optimal across all tested matrix sizes.

Optimal Performance:

- **256×256:** 32 → 4.96 GFLOPS, 73.87 GB/s
- **512×512:** 64 → 4.34 GFLOPS, 64.70 GB/s (32 also performs well)
- **1024×1024:** 32 → 3.96 GFLOPS, 58.95 GB/s

Speedup Analysis:

Table 3: Speedup vs Baseline (Block=8)

Block Size	256×256	512×512	1024×1024
8 (baseline)	1.00×	1.00×	1.00×
green!20 32 (optimal)	1.72×	1.59×	1.52×
256 (poor)	0.95×	0.94×	0.98×

3.4 Q4: Explain Why Block Size 32 is the Best Choice

Answer: Block size 32 is optimal due to cache hierarchy alignment:

3.4.1 1. Cache Capacity Match

Working Set Size:

$$\text{Block 32 Memory} = 24 \times 32^2 = 24{,}576 \text{ bytes} = 24 \text{ KB} \tag{4}$$

Cache Analysis:

- **L1d cache:** 48 KB per core
- **Block 32:** 24 KB fits comfortably in L1
- **Benefit:** L1 access = 1-2 ns (vs 60-100 ns for memory)
- **Block 64:** 96 KB exceeds L1, uses slower L2 (3-10 ns)

3.4.2 2. Temporal Locality (Reuse)

Cache Reuse Pattern:

- Each block element reused B times during computation
- **Block 32:** Each element reused 32× while in L1 cache
- Data stays "hot" in fastest memory level
- Minimizes expensive memory fetches

3.4.3 4. TLB Efficiency

Page Table Impact:

- Page size: 4 KB (typical)
- Block 32: 24 KB ÷ 4 KB = 6 pages
- L1 TLB capacity: 64 entries
- **Result:** TLB can easily hold working set, no page table walks

3.5 Q5: Run the Program with Different Block Sizes

Answer: Executed tests for block sizes: 8, 16, 32, 64, 128, 256

Execution Command:

```
gcc -O2 -o mxm_block mxm_bloc.c -lm
./mxm_block 1024
```

Results show clear pattern:

1. Performance increases: 8 → 16 → 32 (improving cache fit)
2. Peak performance at block 32
3. Performance degrades: 32 → 64 → 128 → 256 (cache thrashing)

3.6 Q6: Compare CPU Time and Bandwidth for Each Block Size

Answer: Comprehensive comparison reveals clear trends:

Time Comparison (1024×1024):

- **Best (32):** 0.5428 s
- **Worst (256):** 0.8375 s
- **Difference:** 1.54× slower for poor choice

Bandwidth Comparison:

- **Best (32):** 58.95 GB/s
- **Worst (256):** 38.21 GB/s
- **Difference:** 1.54× lower bandwidth

Pattern Across Matrix Sizes:

- Optimal block size stable (32-64) regardless of matrix size
- Hardware-determined (cache capacity), not problem-determined
- Larger matrices show bigger speedup from optimal blocking