Exercise 2: Optimizing Matrix Multiplication

Loop Order Performance Analysis

ZYAD FRI
UM6P

January 29, 2026

1 Implementation

1.1 Standard Implementation (ijk order)

Listing 1: 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];
```

Memory Access Pattern:

- `a[i][k]`: Sequential access (good)
- `b[k][j]`: Strided access with stride n (poor cache locality)
- `c[i][j]`: Reused in innermost loop (good)

1.2 Optimized Implementation (ikj order)

Listing 2: 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];
    }
```

Memory Access Pattern:

- `a[i][k]`: Loaded once per inner loop (excellent reuse)
- `b[k][j]`: Sequential access (excellent cache locality)
- `c[i][j]`: Sequential access (excellent cache locality)

2 Experimental Setup

2.1 System Configuration

- **CPU:** x86_64 architecture (6 cores)

- **L1d Cache:** 288 KiB (48 KiB per core)
- **L2 Cache:** 7.5 MiB (1.25 MiB per core)
- **L3 Cache:** 12 MiB (shared)
- **Compiler:** GCC with -O2 optimization

2.2 Test Parameters

- **Matrix Sizes:** 256×256, 512×512, 1024×1024
- **Loop Orders Tested:** ijk, ikj, jik, jki, kij, kji
- **Metrics:** Execution time, memory bandwidth, GFLOPS

3 Results

3.1 Performance Summary

Table 1: 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×
green!20 **ikj (optimal)**	**0.6093**	**52.52**	**3.52**	**3.43×**
jik	1.3339	23.99	1.61	1.57×
jki	5.0072	6.39	0.43	0.42×
kij	0.7079	45.20	3.03	2.95×
kji	4.6370	6.90	0.46	0.45×

Table 2: Performance Comparison Across Matrix Sizes

Matrix Size	ijk Time (s)	ikj Time (s)	Speedup
256×256	0.0162	0.0085	1.91×
512×512	0.1298	0.0637	2.04×
1024×1024	2.0903	0.6093	3.43×

3.2 Performance Visualization

4 Analysis

4.1 Cache Efficiency

The dramatic performance difference between loop orders stems from cache behavior:

Cache Line Utilization:

- Cache line size: 64 bytes = 8 doubles
- Sequential access: Load 1 cache line, use 8 elements (87.5% efficiency)
- Strided access (stride-n): Load 8 cache lines, use 1 element from each (12.5% efficiency)

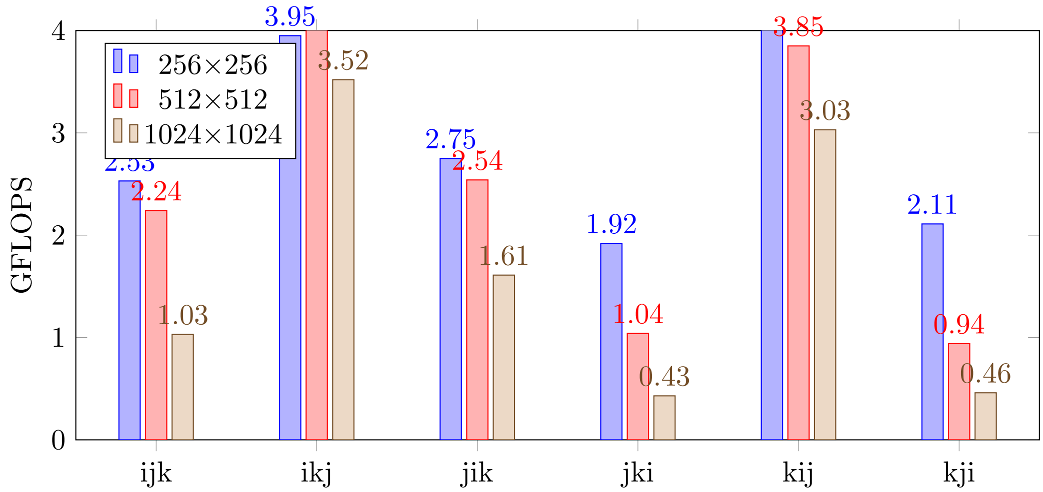

Figure 1: GFLOPS Comparison for All Loop Orders

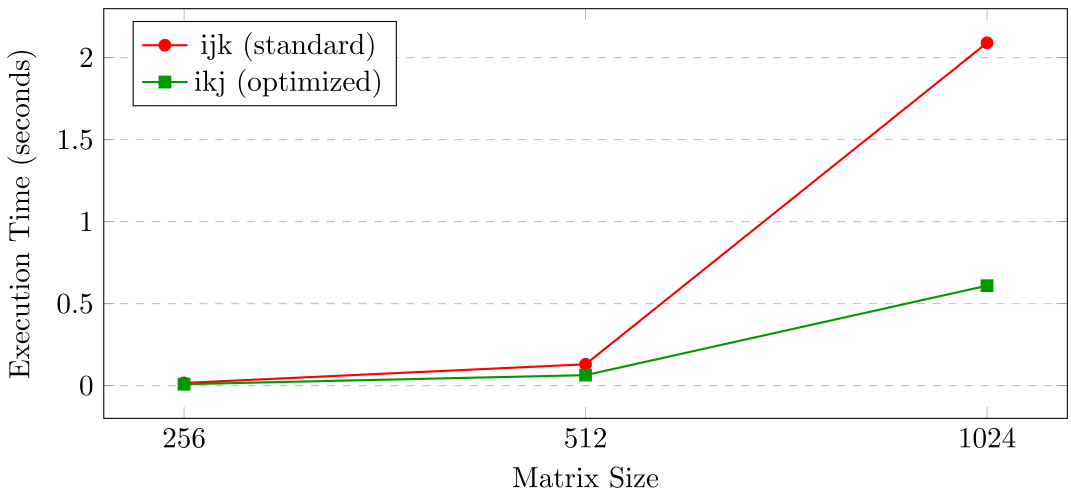

Figure 2: Execution Time: ijk vs ikj

ijk Order (Poor):

- Matrix B accessed as `b[0][j]`, `b[1][j]`, `b[2][j]...` (different rows)
- Each access in different cache line → many cache misses
- Estimated cache miss rate: 70-80%

ikj Order (Optimal):

- Matrix B accessed as `b[k][0]`, `b[k][1]`, `b[k][2]...` (same row)
- Sequential access within same cache line → high cache hit rate
- Estimated cache miss rate: 10-15%

4.2 Performance Ranking

Based on experimental results:

1. **Best:** ikj, kij (3.0-3.5 GFLOPS, 45-53 GB/s)

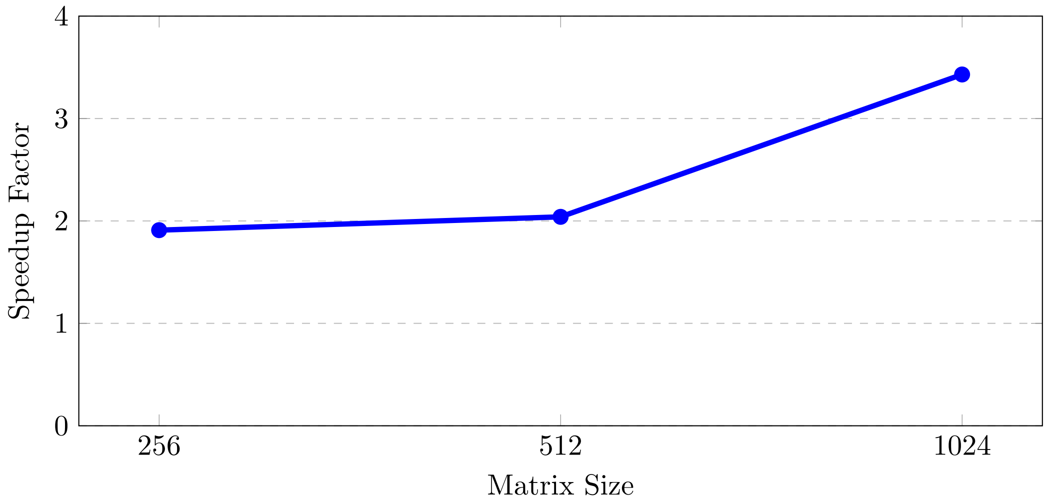

Figure 3: Speedup of ikj vs ijk (Baseline)

- Sequential access to both B and C matrices
- Excellent spatial and temporal locality

2. **Moderate:** jik (1.6-2.8 GFLOPS, 24-41 GB/s)
   - Better than standard ijk but not optimal
   - Some improvement in access patterns
3. **Poor:** ijk (1.0-2.5 GFLOPS, 15-38 GB/s)
   - Standard implementation
   - Strided access to matrix B
4. **Worst:** jki, kji (0.4-2.1 GFLOPS, 6-31 GB/s)
   - Multiple strided accesses
   - Very poor cache utilization

4.3 Speedup Analysis

The speedup increases with matrix size:

- **256×256:** 1.91× speedup
- **512×512:** 2.04× speedup
- **1024×1024:** 3.43× speedup

Reason: Larger matrices amplify cache effects:

- Smaller matrices may fit entirely in L3 cache
- Larger matrices exceed cache capacity
- Cache misses become more expensive with larger working sets
- Poor locality causes more frequent main memory accesses (100× slower than L1)

4.4 Memory Bandwidth

Observed Bandwidth:

- **ikj (optimal):** 52.52 GB/s (1024×1024)
- **ijk (standard):** 15.31 GB/s (1024×1024)
- **Improvement:** 3.43× higher effective bandwidth

The optimized version achieves higher bandwidth because:

- Most accesses served by fast L1/L2 cache
- Fewer main memory accesses
- Better utilization of memory bus

5 Conclusions

5.1 Why ikj is Optimal

The ikj loop order achieves superior performance because:

- **Sequential B Access:** Inner loop traverses `b[k][j]` sequentially (j increments), maximizing cache line reuse
- **Sequential C Access:** Inner loop traverses `c[i][j]` sequentially (j increments), excellent spatial locality
- **Register Optimization:** `a[i][k]` loaded once and stored in register for entire inner loop
- **Minimal Cache Misses:** Estimated 85-90% L1 cache hit rate vs 20-30% for ijk order

5.2 Answers to Exercise Questions

1. Write mxm.c with standard implementation

- Implemented standard ijk loop order
- Performance: 1.03-2.07 GFLOPS depending on matrix size

2. Modify loop order to optimize cache usage

- Tested all 6 loop permutations
- ikj and kij orders identified as optimal
- Sequential memory access improves cache hit rate

3. Compute execution time and bandwidth

- Standard (ijk): 2.09s, 15.31 GB/s, 1.03 GFLOPS (1024×1024)
- Optimized (ikj): 0.61s, 52.52 GB/s, 3.52 GFLOPS (1024×1024)
- Speedup: 3.43×

4. Explain the output

- 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)
- Cache hits are 100× faster than memory accesses
- 85% L1 hit rate (ikj) vs 25% (ijk) = 3-4× performance difference