TP1 - Optimizing Memory Access

DC_CI2_2026

EL HANI Mohammed-Rida

January 2026

1 Exercise 1: Impact of Memory Access Stride

1.1 Overview

This exercise explores how memory access patterns affect performance by measuring the impact of different strides when traversing an array. The program allocates an array of doubles and performs summation with strides ranging from 1 to 20.

1.2 Comparison of Optimization Levels

1.2.1 Without Optimization (-O0)

When compiled with `-O0`, the code executes exactly as written without compiler optimizations. Performance degrades significantly as stride increases because:

- Each increase in stride reduces spatial locality
- Cache lines are loaded but only one element is used
- No prefetching or loop unrolling is applied
- Memory bandwidth is wasted proportionally to the stride

1.2.2 With Optimization (-O2)

The `-O2` optimization level enables several performance improvements:

- Loop unrolling reduces branch overhead
- Better register allocation
- Instruction scheduling to hide memory latency
- Automatic prefetching hints

While performance still degrades with larger strides, the decline is less severe than with `-O0`.

1.3 Execution Time and Bandwidth Analysis

The key observations are:

- **Stride = 1:** Optimal performance due to sequential access pattern. All elements in each cache line are utilized.
- **Increasing stride:** Performance degrades because cache line utilization drops. At stride=20, only 5% of loaded data is actually used.
- **Bandwidth:** The measured bandwidth (MB/s) decreases with stride because it only accounts for useful data, not total memory traffic.

1.4 Impact of Loop Unrolling

Loop unrolling, automatically applied at `-O2`, provides several benefits:

- **Reduced overhead:** Fewer loop control instructions per iteration
- **Instruction-level parallelism:** Multiple independent operations can execute simultaneously
- **Register utilization:** More values can be kept in registers
- **Prefetching:** Longer code sequences give hardware prefetchers more opportunity

However, unrolling cannot overcome poor cache locality at large strides. The fundamental issue remains: strided access wastes memory bandwidth.

1.5 Key Takeaway

Sequential memory access (stride=1) is critical for performance. When designing algorithms, prioritize data structures and access patterns that maximize spatial locality.

2 Exercise 2: Optimizing Matrix Multiplication

2.1 Standard Implementation (ijk order)

The standard nested loop structure is:

```
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:

- Matrix A: row-wise access (good - sequential)
- Matrix B: column-wise access (poor - stride = n)
- Matrix C: single element repeatedly updated (good - stays in register)

The column-wise traversal of B causes approximately n cache misses per inner loop iteration, severely degrading performance for large n.

2.2 Optimized Implementation (ikj order)

By swapping the j and k loops:

```
for (int i = 0; i < n; i++)
    for (int k = 0; k < n; k++)
        for (int j = 0; j < n; j++)
            c[i][j] += a[i][k] * b[k][j];
```

Memory access pattern:

- Matrix A: element `a[i][k]` is loaded once and reused n times
- Matrix B: row-wise access (sequential)
- Matrix C: row-wise access (sequential)

All arrays are now accessed sequentially in the innermost loop, maximizing cache efficiency.

2.3 Performance Comparison

Expected results:

- **Execution time:** ikj order is 3-10× faster for large matrices ($n \geq 256$)
- **Memory bandwidth:** ikj order achieves much higher effective bandwidth
- **Cache behavior:** ikj order has significantly lower miss rates

2.4 Explanation

In C, arrays are stored in row-major order, meaning consecutive elements in a row are contiguous in memory. The ikj ordering ensures that the innermost loop traverses memory sequentially for both B and C, while A benefits from temporal reuse. This transforms the algorithm from cache-hostile to cache-friendly.

3 Exercise 3: Blocked Matrix Multiplication

3.1 Blocking Technique

Blocked (tiled) matrix multiplication divides matrices into $B \times B$ blocks and computes the result block-by-block. Instead of computing individual elements, entire blocks of C are computed using corresponding blocks from A and B.

Key advantage: Once a $B \times B$ block is loaded into cache, it participates in B^2 operations before being evicted, dramatically improving temporal locality.

3.2 Experimental Results

For matrix size $n = 512$ with 2 repetitions, compiled with `-O2`:

Block Size (B)	Time (ms)	GFLOP/s	Bandwidth (GB/s)
8	123.774	4.337	0.136
16	100.399	5.347	0.167
32	71.660	7.492	0.234
64	64.991	8.261	0.258
128	65.476	8.200	0.256

Table 1: Performance vs. block size for $n = 512$

3.3 Analysis

Observations:

- Time decreases sharply from $B = 8$ to $B = 64$
- GFLOP/s increases correspondingly
- Performance plateaus at $B = 64$ and slightly regresses at $B = 128$
- Best performance: $B = 64$ (lowest time, highest GFLOP/s)

3.4 Optimal Block Size

The optimal block size is $B = 64$, achieving:

- Execution time: 64.991 ms
- Performance: 8.261 GFLOP/s
- Speedup: $\approx$1.9$\times$ compared to $B = 8$

3.5 Why $B = 64$ is Optimal

Small blocks ($B = 8, 16$):

- Tiles fit easily in L1 cache but perform insufficient work per tile
- High loop overhead relative to computation
- Poor amortization of memory access costs

Moderate blocks ($B = 32, 64$):

- Optimal balance between cache fit and computational intensity
- Working set ($3 \times B^2$ doubles $\approx$ 24-96 KB) fits in L1/L2 cache
- Each cache line participates in B^2 operations
- Good vectorization opportunities

Large blocks ($B = 128$):

- Working set ($3 \times 128^2 \times 8$ bytes $\approx$ 384 KB) stresses L2 cache
- Cache associativity conflicts may occur
- Increased evictions reduce blocking benefits

3.6 Note on Bandwidth

The reported bandwidth values (0.136-0.258 GB/s) are much lower than actual DRAM bandwidth ($\sim$50-100 GB/s). This is because the calculation counts only "useful" data (matrix elements), not actual memory traffic including cache lines and reused data. For performance comparison, use execution time and GFLOP/s metrics.

4 Exercise 4: Memory Management and Valgrind

4.1 Objective

Use Valgrind's Memcheck tool to detect and fix memory leaks in a C program that allocates, initializes, duplicates, and prints integer arrays.

4.2 Program Functions

allocate_array(size):

- Allocates `size * sizeof(int)` bytes using `malloc()`
- Returns pointer to allocated memory
- Exits with error if allocation fails

initialize_array(arr, size):

- Fills array with values: `arr[i] = i * 10`
- For SIZE=5: creates array [0, 10, 20, 30, 40]

print_array(arr, size):

- Displays all array elements to stdout

duplicate_array(arr, size):

- Allocates new array of same size
- Copies data using `memcpy()`
- Returns pointer to the copy
- Creates a second independent heap allocation

free_memory(arr):

- Should release heap memory using `free(arr)`
- Original version had this function empty

4.3 Memory Leak and Fix

Problem: The program creates two heap allocations (`array` and `array_copy`), but the original `free_memory()` function was empty, causing memory leaks.

Fix applied:

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

int main() {
    // ... program code ...
    free_memory(array);
    free_memory(array_copy);
    return 0;
}
```

This ensures every `malloc()` has a matching `free()`.

4.4 Valgrind Usage

Compilation with debug symbols:

```
gcc -g -o memory_debug memory_debug.c
```

Run with Valgrind Memcheck:

```
valgrind --leak-check=full --track-origins=yes ./memory_debug
```

Flags:

- `--leak-check=full`: Detailed information about each leaked block
- `--track-origins=yes`: Tracks origin of uninitialized values

4.5 Expected Valgrind Output

After fixing the leaks, Valgrind reports:

- "All heap blocks were freed – no leaks are possible"
- 0 bytes definitely lost
- 0 bytes indirectly lost
- 0 bytes still reachable

Leak categories:

- **Definitely lost:** Memory with no remaining pointers
- **Indirectly lost:** Memory referenced only by leaked blocks
- **Still reachable:** Memory not freed but still pointed to at exit

5 Exercise 5: HPL Benchmark

5.1 Hardware Specifications

- **Processor:** Intel Xeon Platinum 8276L
- **Configuration:** 2 sockets, 28 cores per socket
- **Base frequency:** 2.2 GHz
- **Vector units:** AVX-512 with FMA
- **Peak performance per core:** 32 DP FLOP/cycle

Theoretical single-core peak:

$$P_{core} = 1 \times 2.2 \times 10^9 \text{ Hz} \times 32 \text{ FLOP/cycle} = 70.4 \text{ GFLOP/s} \quad (1)$$

5.2 Experimental Setup

HPL executed with:

- MPI processes: 1 (P=1, Q=1)
- OpenMP threads: 1
- Single-core, single-threaded runs

Data extraction: Performance read from HPL output line `WR00L2L2`:

```
WR00L2L2  N  NB  P  Q  Time  Gflops
```

5.3 Example Run

For $N = 1000$, $NB = 1$:

- **Output:** `WR00L2L2 1000 1 1 1 0.18 3.7328e+00`
- **Validation:** PASSED
- **Performance:** 3.7328 GFLOP/s
- **Efficiency:** 3.7328/70.4 = 5.3% of peak

5.4 Measured vs. Theoretical Performance

For the example run:

- Measured: $P_{HPL} = 3.7328$ GFLOP/s
- Theoretical: $P_{core} = 70.4$ GFLOP/s
- Ratio: 5.3%

This large gap is expected for small problem sizes and non-optimal configurations.

5.5 Efficiency Calculation

The efficiency formula is:

$$\eta = \frac{P_{HPL}}{P_{core}} \tag{2}$$

This should be computed for all 36 experimental runs ($4 \times 9 = 36$ combinations of N and NB).

5.6 Influence of Matrix Size (N) and Block Size (NB)

5.6.1 Effect of Increasing N

GFLOP/s typically increases:

- Larger problems have better computational intensity
- BLAS-3 operations dominate: $O(n^3)$ computation vs. $O(n^2)$ data
- Fixed overheads become relatively smaller

Execution time increases:

- Time grows as $O(n^3)$ for matrix multiplication
- Approximately $2n^3$ floating-point operations required

5.6.2 Effect of Block Size (NB)

Very small NB (1, 2, 4):

- Excessive panel factorization overhead
- Poor cache blocking
- Inefficient BLAS operations
- Result: Low GFLOP/s

Moderate NB (16, 32, 64, 128):

- Good cache blocking
- Efficient BLAS-3 kernels
- Balanced factorization vs. update costs
- Result: Maximum GFLOP/s

Very large NB (256):

- Working sets may exceed cache capacity
- Cache associativity conflicts
- Panels become too large
- Result: Performance may plateau or decrease

5.7 Identifying Best Block Size

For each fixed N, select the NB that produces maximum GFLOP/s in the `WR00L2L2` output.

Procedure:

1. Group results by matrix size N
2. Compare GFLOP/s across all NB values
3. Select NB with highest GFLOP/s

Note: Optimal NB may vary with N depending on cache hierarchy.

5.8 Why Measured Performance is Below Peak

Algorithm structure:

- HPL includes panel factorization with pivoting (not pure GEMM)
- Panel operations are BLAS-2 (less compute-dense)
- Pivoting introduces dependencies and irregular memory access

Hardware limitations:

- Peak assumes perfect AVX-512 FMA utilization every cycle
- Real execution has stalls, branch mispredictions, dependencies
- Memory bandwidth becomes bottleneck when data doesn't fit in cache
- Cache misses introduce unavoidable latency

Implementation overhead:

- BLAS library overhead and function calls
- Data layout conversions
- Synchronization operations

Frequency variations:

- Actual CPU frequency may differ from nominal 2.2 GHz
- Thermal throttling or power limits
- Turbo boost behavior is workload-dependent

Problem size effect:

- For $N = 1000$, matrices barely fit in L3 cache
- Setup costs are not well amortized
- Insufficient work to saturate execution units

Expected efficiency ranges:

- Small problems ($N \leq 5000$): 5-30% of peak
- Medium problems ($N \approx 10000$): 30-60% of peak
- Large problems ($N \geq 20000$, optimal NB): 60-85% of peak