TP1 - Optimizing Memory Access

Oussama Laaroussi

January 29, 2026

Exercise 1: Impact of Memory Access Stride

This exercise investigates how accessing memory with different "strides" affects bandwidth and execution time. We compare the performance of compiled code with no optimization (`-O0`) and level 2 optimization (`-O2`).

Code Snippet

The core loop traverses the array jumping by `i_stride`:

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

    // Jumping memory locations by stride
    for (int i = 0; i < N * i_stride; i += i_stride)
        sum += a[i];

    // ... calculate time and rate ...
}
```

Listing 1: Stride access loop in ex1.c

Results and Analysis

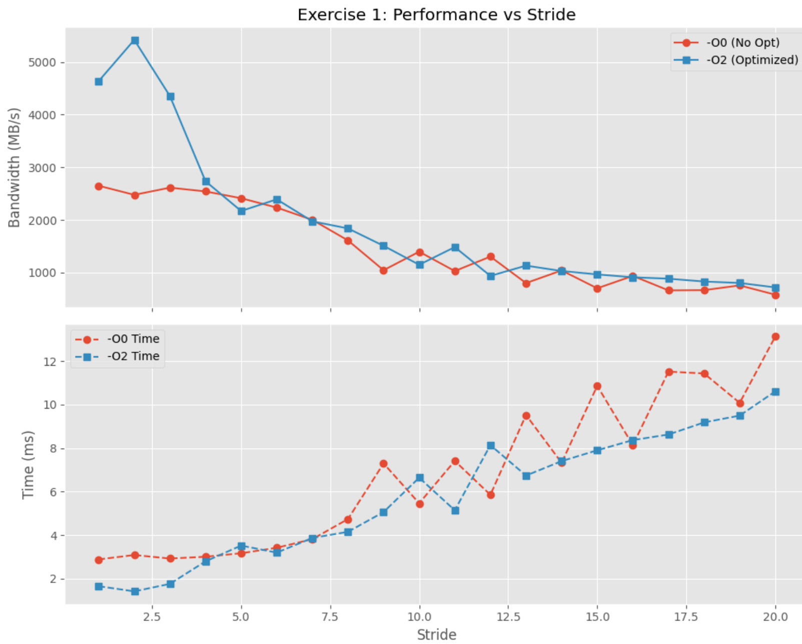

Figure 1: Bandwidth and Execution Time vs Stride

As shown in Figure 1:

- **Bandwidth (Top):** The memory bandwidth drops significantly as the stride increases due to poor spatial locality and increased cache misses.
- **Execution Time (Bottom):** As expected, execution time increases with the stride for the unoptimized version. The `-O2` version is not only faster (lower time) but also maintains a flatter time curve, suggesting the compiler successfully optimized instructions or prefetching (though cache latencies still exist).

Exercise 2: Optimizing Matrix Multiplication

We compare a "Naive" matrix multiplication (loops ordered i, j, k) with an "Optimized" version (loops ordered i, k, j).

Code Analysis

```c
// Naive: Accesses B[k][j] in inner loop (Stride = N)
void multiply(float **A, float **B, float **C, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            C[i][j] = 0.0f;
            for (int k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}

// Optimized: Accesses B[k][j] sequentially (Stride = 1)
void multiply_optimize(float **A, float **B, float **C, int n) {
    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];
}
```

Listing 2: Naive (ijk) vs Optimized (ikj) implementation

Results and Analysis

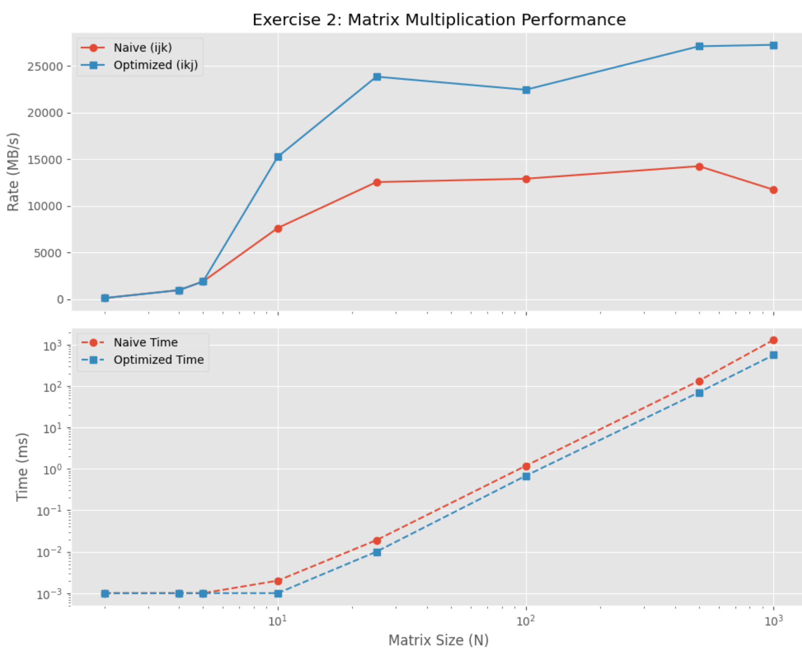

Figure 2: Performance Comparison: Naive vs Optimized Loop Order

Figure 2 demonstrates a massive performance gap:

- **Bandwidth:** The optimized version achieves significantly higher MB/s because the inner loop accesses memory sequentially (stride 1).
- **Execution Time:** Note the logarithmic scale. The naive implementation (i, j, k) takes exponentially longer as N increases. For $N = 1000$, the optimized version is orders of magnitude faster because it minimizes cache misses by accessing `B[k][j]` sequentially instead of jumping rows.

Exercise 3: Blocked Matrix Multiplication

This exercise implements a blocked (tiled) algorithm to pinpoint an optimal block size that fits within the CPU cache.

Results and Analysis

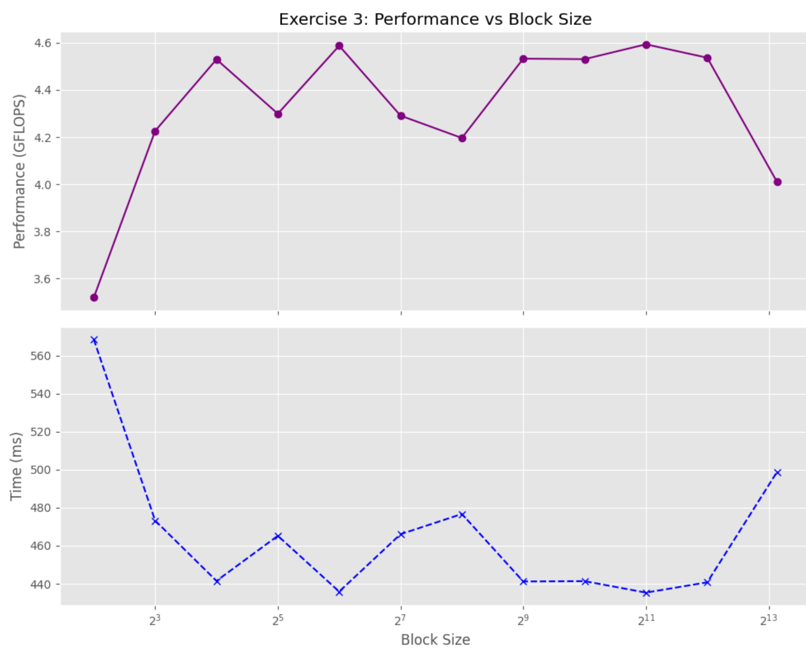

Figure 3: GFLOPS and Time vs Block Size

- **Optimal Size:** The performance peaks (and time hits a valley) at specific block sizes (typically between 16 and 128 depending on architecture). This block size fits the L1/L2 cache perfectly.
- **Time vs Performance:** The execution time is lowest where the GFLOPS are highest. Very small blocks suffer from loop overhead, while very large blocks cause cache thrashing.

Exercise 4: Memory Management and Debugging

In this exercise, we analyzed a program with a memory leak. We used Valgrind to identify the issue and fixed it by properly freeing allocated memory.

Valgrind Analysis: Before Fix

Running the initial code with Valgrind produced the following error report, indicating that 40 bytes were lost (memory leak).

```
==5868== Memcheck, a memory error detector
...
==5868== HEAP SUMMARY:
==5868==     in use at exit: 40 bytes in 2 blocks
==5868==   total heap usage: 3 allocs, 1 frees, 1,064 bytes allocated
==5868==
==5868== 20 bytes in 1 blocks are definitely lost in loss record 1 of 2
==5868==    at 0x4846828: malloc
==5868==    by 0x109208: allocate_array (ex4.c:8)
==5868==
==5868== LEAK SUMMARY:
==5868==    definitely lost: 40 bytes in 2 blocks
```

Listing 3: Valgrind output before fix

The Fix

We modified the code to free the array that was duplicated.

```c
void free_memory(int *arr) {
    // FIX: Free the memory allocated by malloc
    free(arr);
}
```

Listing 4: Fixed free_memory function

Valgrind Analysis: After Fix

After applying the fix, Valgrind confirms that all heap blocks were freed and no leaks remain.

```
==6214== Memcheck, a memory error detector
...
==6214== HEAP SUMMARY:
==6214==     in use at exit: 0 bytes in 0 blocks
==6214==   total heap usage: 3 allocs, 3 frees, 1,064 bytes allocated
==6214==
==6214== All heap blocks were freed -- no leaks are possible
==6214== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Listing 5: Valgrind output after fix

Exercise 5: HPL Benchmark

We ran the High-Performance Linpack (HPL) benchmark to measure the floating-point performance of a single core on an Intel Xeon Platinum 8276L (Theoretical Peak ≈ 70.4 GFLOP/s).

Experimental Results

We tested various Matrix Sizes (N) and Block Sizes (NB).

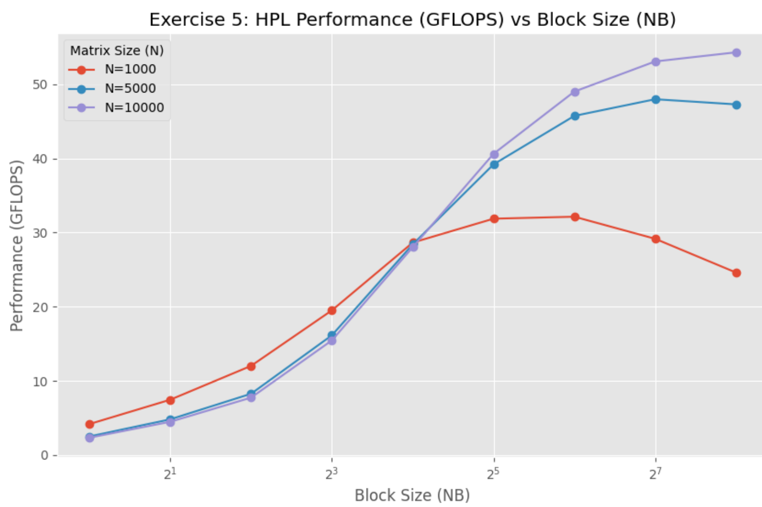

Figure 4: HPL Performance (GFLOPS) vs Block Size (NB) for different N

Analysis

1. **Impact of Matrix Size (N):** Performance generally increases with N.
   - Small matrices (e.g., $N = 1000$) do not saturate the compute units or the memory bandwidth efficiently, achieving roughly 30 GFLOPS.
   - Larger matrices ($N = 10000$) allow the CPU to maintain high throughput for longer periods, reaching over 50 GFLOPS.
2. **Impact of Block Size (NB):**
   - **Small NB (1-8):** Performance is very poor. The overhead of function calls and lack of vectorization potential limits throughput.
   - **Optimal Range:** The performance plateaus and peaks around $NB = 128$ or $NB = 256$. This block size provides a good balance between cache locality and minimizing loop overhead.

3. **Efficiency:** The maximum measured performance is approximately **54.3 GFLOP/s**. Given the theoretical peak of **70.4 GFLOP/s**, the efficiency is:

$$\eta = \frac{54.3}{70.4} \approx 77\%$$

   The gap is due to real-world constraints such as memory latency (not all data is always in L1 cache), branch misprediction, and OS overhead.