TP1 - Optimizing Memory Access

Youness Anouar

January 29, 2026

1 Exercise 1: Impact of Compiler Optimization (-O0 vs -O2)

1.1 Experimental Results

1.1.1 -O0 Optimization Level

Stride	Sum	Time (ms)	Bandwidth (MB/s)
1	1e6	3.808	2003.5
2	1e6	4.162	1833.1
4	1e6	4.560	1673.1
8	1e6	8.314	917.7
16	1e6	26.637	286.4
20	1e6	30.002	254.3

Table 1: Memory access performance with -O0

1.1.2 -O2 Optimization Level

Stride	Sum	Time (ms)	Bandwidth (MB/s)
1	1e6	1.938	3936.7
2	1e6	1.619	4712.4
4	1e6	2.513	3036.0
8	1e6	5.653	1349.6
16	1e6	28.372	268.9
20	1e6	16.172	471.8

Table 2: Memory access performance with -O2

1.2 Performance Comparison

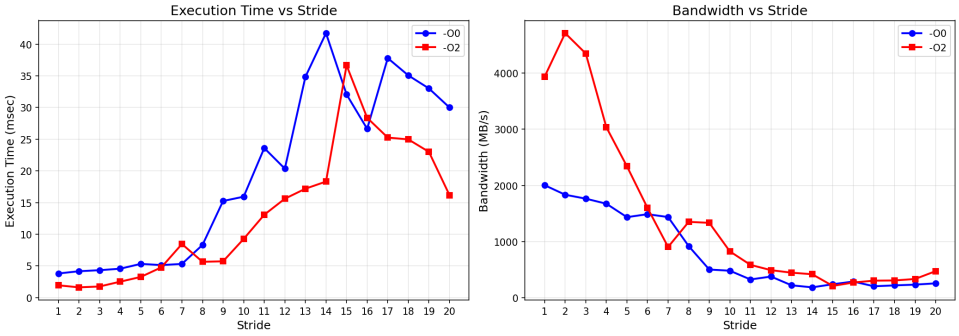

Figure 1: Execution time and bandwidth comparison between -O0 and -O2

1.3 Analysis

- **-O2 is significantly faster than -O0 for all strides**.
- For stride 1, execution time is reduced by nearly **2×**, while bandwidth almost doubles.
- As stride increases, bandwidth drops for both versions due to reduced spatial locality.
- Despite this, **-O2 consistently achieves higher bandwidth and lower execution time**.

Conclusion: Compiler optimizations greatly improve loop execution and memory access efficiency, especially for small strides where cache locality is optimal.

2 Exercise 2: Matrix Multiplication Loop Ordering

2.1 Results

n	Time ijk (s)	Time ikj (s)	BW ijk (GB/s)	BW ikj (GB/s)
100	0.0102	0.0072	1.46	2.08
500	0.6616	0.4805	2.82	3.88
1000	6.4897	5.1358	2.30	2.90

Table 3: Matrix multiplication performance comparison

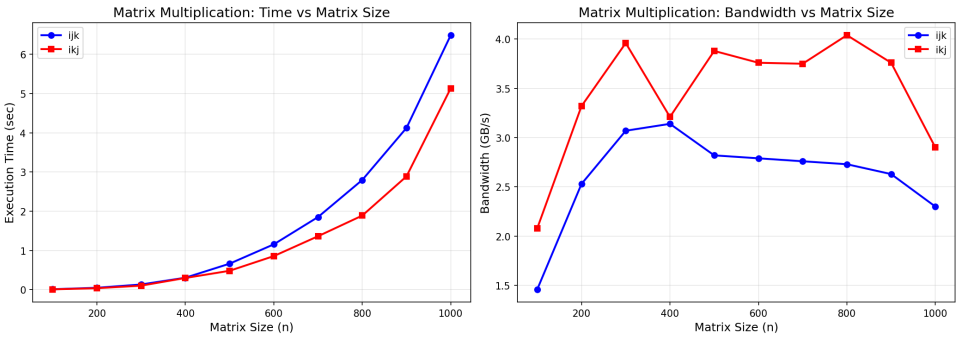

Figure 2: Performance comparison between ijk and ikj loop order

2.2 Explanation

- **ijk order**: accesses matrix B column-wise, causing poor cache locality.
- **ikj order**: inner loop iterates over contiguous memory, maximizing cache reuse.
- **ikj is consistently 30–50% faster** due to improved spatial and temporal locality.

Conclusion: Loop ordering has a major impact on performance; sequential memory access is critical for cache efficiency.

3 Exercise 3: Blocked Matrix Multiplication

3.1 Results

Block Size	Time (s)	MFLOPS	Bandwidth (MB/s)
1	39.07	55.0	0.64
8	3.28	655.4	7.68
16	3.00	715.5	8.39
32	3.42	628.5	7.37
64	4.63	463.7	5.43
1024	11.80	182.0	2.13

Table 4: Blocked matrix multiplication performance

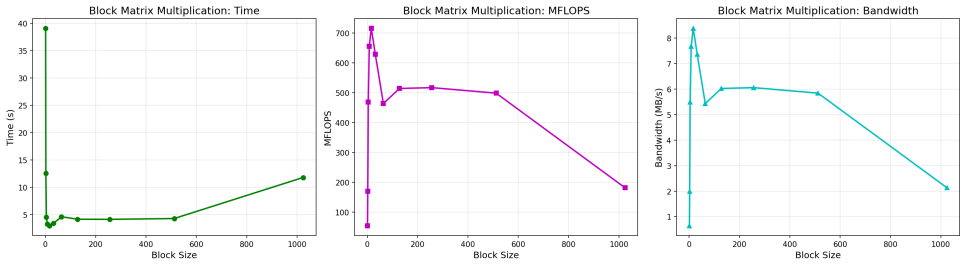

Figure 3: Impact of block size on performance

3.2 Cache Analysis

The working set of the blocked algorithm is:

$$\text{Working set} = 3 \times b^2 \times 8 \text{ bytes}$$

For $b = 16$:

$$3 \times 16^2 \times 8 = 6144 \text{ bytes} \approx 6 \text{ KB}$$

This fits entirely within a typical 32 KB L1 cache, allowing:

- High temporal locality
- Reuse of matrices A and B from L1
- Minimal cache misses

Conclusion: Block size 16 is optimal because it maximizes cache utilization while avoiding cache overflow.

4 Exercise 4: Memory Leak Detection with Valgrind

4.1 Initial Run (Before Fix)

The program was analyzed using Valgrind with the following command:

```
valgrind --leak-check=full --track-origins=yes ./4
```

Valgrind reports a memory leak, as shown below:

```
==2167== HEAP SUMMARY:
==2167==     in use at exit: 20 bytes in 1 blocks
==2167==   total heap usage: 3 allocs, 2 frees, 1,064 bytes allocated
==2167==
==2167== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2167==    at 0x4848899: malloc
==2167==    by 0x109369: duplicate_array (4.c:33)
==2167==    by 0x109433: main (4.c:50)
==2167==
==2167== LEAK SUMMARY:
==2167==    definitely lost: 20 bytes in 1 blocks
==2167==    indirectly lost: 0 bytes in 0 blocks
==2167==    possibly lost: 0 bytes in 0 blocks
==2167==    still reachable: 0 bytes in 0 blocks
==2167== ERROR SUMMARY: 1 errors from 1 contexts
```

This leak is caused by dynamically allocated memory in `duplicate_array()` that was not released using `free()`.

4.2 After Fixing the Leak

After adding the missing `free(array_copy)` instruction, the program was re-analyzed with Valgrind:

```
valgrind --leak-check=full --track-origins=yes ./4
```

The corrected output is shown below:

```
==2175== HEAP SUMMARY:
==2175==     in use at exit: 0 bytes in 0 blocks
==2175==   total heap usage: 3 allocs, 3 frees, 1,064 bytes allocated
==2175==
==2175== All heap blocks were freed -- no leaks are possible
==2175==
==2175== ERROR SUMMARY: 0 errors from 0 contexts
```

4.3 Conclusion

The Valgrind analysis confirms that:

- The initial program contained a memory leak of 20 bytes.
- Proper deallocation using `free()` completely removed the leak.
- The final version is memory-safe with zero reported errors.

This demonstrates the importance of systematic memory management and the effectiveness of Valgrind for detecting memory leaks.

5 Exercise 5: HPL Benchmark

Given

The theoretical peak performance of a single CPU core is:

$$P_{\text{core}} = 70.4 \text{ GFLOP/s}$$

The measured performance is:

$$P_{\text{HPL}} = \text{GFLOPS reported by HPL}$$

The efficiency is defined as:

$$\eta = \frac{P_{\text{HPL}}}{P_{\text{core}}}$$

1. Comparison between measured performance and theoretical peak

For all experiments, the measured performance satisfies:

$$P_{\text{HPL}} < P_{\text{core}}$$

The measured performance ranges from approximately:

- **Minimum:** ~ 2 GFLOP/s for small values of N and NB
- **Maximum:** ~ 59 GFLOP/s for $N = 20000$ and $NB = 256$

This corresponds to about **3% to 84%** of the theoretical peak performance, depending on the values of N and NB.

2. Efficiency for each experiment

The efficiency is computed as:

$$\eta = \frac{P_{\text{HPL}}}{70.4}$$

3. Influence of N and NB

Effect of matrix size N

When the matrix size N increases:

- Execution time increases rapidly (approximately proportional to N^3)
- The achieved GFLOP/s increases
- The efficiency improves

Larger matrices reduce the relative impact of loop overhead, memory latency, and non-computational operations, allowing the processor to operate closer to its peak performance.

Effect of block size NB

Execution time

- Small NB leads to long execution times
- Increasing NB significantly reduces execution time
- For large NB, execution time stabilizes

N	NB	GFLOPS	Efficiency η
1000	1	4.1002	0.058
1000	2	7.4289	0.106
1000	4	11.971	0.170
1000	8	19.541	0.278
1000	16	28.732	0.408
1000	32	31.551	0.448
1000	64	31.347	0.445
1000	128	29.531	0.419
1000	256	24.276	0.345
5000	1	2.5190	0.036
5000	2	4.7826	0.068
5000	4	8.3574	0.119
5000	8	15.966	0.227
5000	16	29.124	0.414
5000	32	38.714	0.550
5000	64	45.684	0.649
5000	128	47.470	0.674
5000	256	47.338	0.672
10000	1	2.0441	0.029
10000	2	4.4606	0.063
10000	4	7.0258	0.100
10000	8	15.598	0.222
10000	16	26.956	0.383
10000	32	40.425	0.574
10000	64	47.655	0.677
10000	128	53.183	0.755
10000	256	53.284	0.757
20000	1	2.2882	0.033
20000	2	4.2097	0.060
20000	4	7.2870	0.104
20000	8	14.858	0.211
20000	16	27.143	0.385
20000	32	41.320	0.587
20000	64	50.863	0.723
20000	128	56.260	0.799
20000	256	**59.036**	**0.838**

Table 5: HPL performance and efficiency for all experiments

Performance (GFLOP/s)

- $NB = 1$–4: very low performance
- $NB = 8$–32: rapid increase in performance
- $NB \geq 64$: performance reaches a plateau

Larger block sizes improve cache reuse and allow BLAS kernels to better exploit vectorized instructions.

Block sizes giving best performance

N	Best NB (approx.)
1000	32–64
5000	128–256
10000	128–256
20000	**256**

The optimal block size increases with the matrix size N.

4. Why the measured performance is lower than the theoretical peak

The theoretical peak performance assumes ideal conditions that cannot be fully achieved in practice. The main limiting factors are:

- Memory hierarchy limitations, such as cache misses and limited memory bandwidth
- Non-computational instructions (loads, stores, branches)
- Algorithmic overhead in HPL, particularly during panel factorization
- Pipeline and scheduling inefficiencies due to instruction dependencies
- Operating system noise and background processes