

CS211_HW1_Hugo_Wan

My github link for HW1:

<https://github.com/UCR-HPC/cs211-hw1-general-matrix-multiplication-HugoWan0504/tree/main>

- Q1

a. Given the instructions, memory incurs a delay of 100 cycles for r/w.

FLOPs = $2n^3$. For $n = 1000$, $2 * 1000^3 = 2 * 10^9$ FLOPs.

Time for 4 operations per cycle = $2 * 10^9 / 4 = 20 * 10^8 / 4 = 5 * 10^8$ cycles.

Since clock frequency = 2 GHz = $2 * 10^9$ Hz,

FLOPs time = $5 * 10^8 / 2 * 10^9 = \mathbf{0.25 \text{ seconds}}$.

Memory accesses in dgemm0 = $3 * n^3 = 3 * 1000^3 = 3 * 10^9$ accesses.

Cycles of these = $3 * 10^9 * 100 = 3 * 10^{11}$ cycles.

Time for these = $3 * 10^{11} / 2 * 10^9 = \mathbf{150 \text{ seconds}}$.

In dgemm1 A & B, the accesses number for $n^3 = 10^9$ and $n^2 = 10^6$ for C.

Total accesses in dgemm1 = $2 * 10^9 + 10^6 = 2.0001 * 10^9$ accesses.

The cycles for the access = $2.0001 * 10^9 * 100 = 2.0001 * 10^{11}$ cycles.

The time for the access = $2.0001 * 10^{11} / 2 * 10^9 = \mathbf{100.00005 \text{ seconds}}$.

Since FLOPs time is 0.25 seconds, and time for dgemm0 and dgemm1 memory accesses are 150 seconds and 100.00005 seconds respectively.

Therefore, the total time for dgemm0 is $150 + 0.25 = \mathbf{150.25 \text{ seconds}}$.

And the total time for dgemm1 is $100.00005 + 0.25 = \mathbf{100.25005 \text{ seconds}}$.

b. Test dgemm0 and dgemm1 on hpc-001 with $n = 64, 128, 256, 512, 1024, 2048$. Here are the screenshots of the outputs:

Running dgemm1 with n=2048 n=2048 time=19.185031s	Running dgemm1 with n=1024 n=1024 time=2.404475s
Running dgemm1 with n=512 n=512 time=0.301927s	Running dgemm1 with n=256 n=256 time=0.038197s

```
Running dgemm1 with n=128  
n=128  
time=0.004848s
```

```
Running dgemm1 with n=64  
n=64  
time=0.000597s
```

```
Running dgemm0 with n=2048  
n=2048  
time=173.438865s
```

```
Running dgemm0 with n=1024  
n=1024  
time=7.760138s
```

```
Running dgemm0 with n=512  
n=512  
time=1.016493s
```

```
Running dgemm0 with n=256  
n=256  
time=0.095254s
```

```
Running dgemm0 with n=128  
n=128  
time=0.007726s
```

```
Running dgemm0 with n=64  
n=64  
time=0.000858s
```

Reorganize them with a table, including N and time:

N	Dgemm0 Time (seconds)	Dgemm1 Time (seconds)
64	0.000858	0.000597
128	0.007726	0.004848
256	0.095254	0.038197
512	1.016493	0.301927
1024	7.760138	2.404475
2048	173.438865	19.185031

Calculate the performance of each algorithm in Gflops:

$$\text{Gflops} = (2 * n^3 / \text{time}) * 10^9$$

N	FLOPs (2*N^3)	Dgemm0 Gflops	Dgemm1 Gflops
64	524,288	0.611	0.878
128	2,097,152	0.271	0.432
256	16,777,216	0.176	0.432
512	134,217,728	0.132	0.445
1024	1,073,741,824	0.138	0.447
2048	8,589,934,592	0.049	0.448

Since more Gflops = faster performance, my **dgemm1** is **significantly faster and more efficient than dgemm0** across all matrix sizes.

- Q2

Test dgemm2 on hpc-001 with n=64, 128, 256, 512, 1024, 2048:

Here are the screenshots of the outputs:

```
[twan012@cluster-001-login-node cs211-hw1-code-files]$ srun ./main dgemm2 2048 1
n=2048
time=81.852896s
```

```
[twan012@cluster-001-login-node cs211-hw1-code-files]$ srun ./main dgemm2 1024 1
n=1024
time=5.208451s
```

```
[twan012@cluster-001-login-node cs211-hw1-code-files]$ srun ./main dgemm2 512 1
n=512
time=0.676252s
```

```
[twan012@cluster-001-login-node cs211-hw1-code-files]$ srun ./main dgemm2 256 1
n=256
time=0.049432s
```

```
[twan012@cluster-001-login-node cs211-hw1-code-files]$ srun ./main dgemm2 128 1
n=128
time=0.005555s
```

```
[twan012@cluster-001-login-node cs211-hw1-code-files]$ srun ./main dgemm2 64 1
n=64
time=0.000654s
```

Reorganize them with a table, including N, time, FLOPs, and Gflops:

$$\text{Gflops} = (2 * n^3 / \text{time}) * 10^9$$

N	Dgemm2 Time (seconds)	FLOPs (2*N^3)	Dgemm2 Gflops
64	0.000654	524,288	0.801
128	0.005555	2,097,152	0.378
256	0.049432	16,777,216	0.338
512	0.676252	134,217,728	0.198
1024	5.208451	1,073,741,824	0.206
2048	81.852896	8,589,934,592	0.105

According to the table, Dgemm2 is the fastest at N = 64 and is generally faster with less N.

- Q3

Test dgemm3 on hpc-001 with n=64, 128, 256, 512, 1024, 2048:

Here are the screenshots of the outputs:

```
[twan012@cluster-001-login-node cs211-hw1-code-files]$ ./main dgemm3 2048  
n=2048  
time=120.577144s
```

```
[twan012@cluster-001-login-node cs211-hw1-code-files]$ ./main dgemm3 1024 1  
n=1024  
time=6.575126s
```

```
[twan012@cluster-001-login-node cs211-hw1-code-files]$ ./main dgemm3 512 1  
n=512  
time=0.853417s
```

```
[twan012@cluster-001-login-node cs211-hw1-code-files]$ ./main dgemm3 256 1  
n=256  
time=0.074728s
```

```
[twan012@cluster-001-login-node cs211-hw1-code-files]$ ./main dgemm3 128 1  
n=128  
time=0.008196s
```

```
[twan012@cluster-001-login-node cs211-hw1-code-files]$ ./main dgemm3 64 1  
n=64  
time=0.001097s
```

Reorganize them with a table, including N, time, FLOPs, and Gflops:

N	Dgemm3 Time (seconds)	FLOPs ($2 \cdot N^3$)	Dgemm3 Gflops
64	0.001097	524,288	0.478
128	0.008196	2,097,152	0.256
256	0.074728	16,777,216	0.224
512	0.853417	134,217,728	0.157
1024	6.575126	1,073,741,824	0.163
2048	120.577144	8,589,934,592	0.071

(Next page for more, formatting)

Compare the performance of dgemm3 with dgemm0~2:

N	Dgemm0 Gflops	Dgemm1 Gflops	Dgemm2 Gflops	Dgemm3 Gflops
64	0.611	0.878	0.801	0.478
128	0.271	0.432	0.378	0.256
256	0.176	0.432	0.338	0.224
512	0.132	0.445	0.198	0.157
1024	0.138	0.447	0.206	0.163
2048	0.049	0.448	0.105	0.071

According to the table, and given that performance improves as Gflops increases, **dgemm1 is the fastest** and most efficient algorithm overall. While dgemm2 and dgemm3 offer some benefits for smaller matrices, they don't scale as well with larger sizes. **Dgemm0 shows the lowest performance** overall, especially as the matrix size increases, making it the least efficient among the four.

- Q4 Test first functions of the dgemm6 series on hpc-001 with n=10 & 10000:

Here is the screenshot for the outputs:

```
[twan012@cluster-001-login-node cs211-hw1-code-files]$ srun ./main dgemm6_ijk 10 1
n=10
time=0.000003s
[twan012@cluster-001-login-node cs211-hw1-code-files]$ srun ./main dgemm6_ikj 10 1
n=10
time=0.000004s
[twan012@cluster-001-login-node cs211-hw1-code-files]$ srun ./main dgemm6_jik 10 1
n=10
time=0.000005s
[twan012@cluster-001-login-node cs211-hw1-code-files]$ srun ./main dgemm6_jki 10 1
n=10
time=0.000004s
[twan012@cluster-001-login-node cs211-hw1-code-files]$ srun ./main dgemm6_kij 10 1
n=10
time=0.000004s
[twan012@cluster-001-login-node cs211-hw1-code-files]$ srun ./main dgemm6_kji 10 1
n=10
time=0.000004s
[twan012@cluster-001-login-node cs211-hw1-code-files]$
```

Organize Dgemm6 series with a table:

N	ijk	ikj	jik	jki	kij	kji
10	0.000003	0.000004	0.000005	0.000004	0.000004	0.000004

Given from the instructions, the cache size has 60 lines, line size has 10 doubles, so the total cache capacity is $60 * 10 = 600$ doubles. And the matrix size is $10 \times 10 = 100$ elements for each matrix.

The total elements of the three matrices A, B, and C are $3 * 100 = 300$ elements. And $300 < 600$, so the matrices must fit in the cache. No possible cache misses, and therefore, the overall miss rates of the dgemm6 series are 0%.

Here is a table where N = 10, OMR stands for Overall Miss Rate:

Order	A read	A miss	B read	B miss	C read	C miss	OMR
ijk	100	0	100	0	100	0	0%
ikj	100	0	100	0	100	0	0%
jik	100	0	100	0	100	0	0%
jki	100	0	100	0	100	0	0%
kij	100	0	100	0	100	0	0%
Kji	100	0	100	0	100	0	0%

Now, what will these values be when the matrix size is 10000×10000 ?

The matrix size of each matrix A, B, and C is 10000×10000 , containing 100 million elements. Block size remain $10 \times 10 = 100$ elements. The total blocks per matrix are $10000/10 \times 10000/10 = 1000 \times 1000 = 1,000,000$ blocks per matrix.

Since the cache can only hold 600 elements, the cache can only hold $600 / 100 = 6$ blocks at a time, the need to reload blocks is frequent. For each block, it is very likely to incur a cache miss when accessing new blocks. Hence, any block of A, B, or C that is needed after these blocks are evicted will result in a cache miss.

(It took me awhile to run any function of the dgemm6 series in the terminal. It had crashed the terminal multiple times that drove me crazy. So, I will use approximate values as the answers. Update: It ran over 30 minutes time limit, and I refuse to increase time limit here.)

Here is a table where N = 10000, OMR stands for Overall Miss Rate:

Order	A read	A miss	B read	B miss	C read	C miss	OMR
ijk	100 M	High	100 M	High	100 M	High	High
ikj	100 M	High	100 M	High	100 M	High	High
jik	100 M	High	100 M	High	100 M	High	High
jki	100 M	High	100 M	High	100 M	High	High
kij	100 M	High	100 M	High	100 M	High	High
Kji	100 M	High	100 M	High	100 M	High	High

In conclusion, when the matrix size is small (10x10), the cache is sufficiently large to store all matrix elements, resulting in no cache missing for any loop order. This shows that when the working set size is smaller than the cache size, the choice of loop order does not impact performance significantly. Moreover, when the matrix size is 10000x10000, the total number of reads for matrices A, B, and C is 100 million each. Since the cache can only hold 6 blocks at a time, we expect a high number of cache misses for all loop orders, as blocks will need to be frequently reloaded into the cache. Consequently, the overall cache miss rate will be high for each of the loop orders, and the performance will be heavily impacted by the cache miss penalty.

- Q5 Test second functions of the dgemm6 series on hpc-001 with n=10000:

Similar issue from Q4, my terminal kept on crashing and doesn't allow me to get my outputs in time. Here is the table with the reasonable estimates:

Here is a table where N = 10000, OMR stands for Overall Miss Rate:

Order	A read	A miss	B read	B miss	C read	C miss	OMR
ljk2	100 M	High	100 M	High	100 M	High	High
lkj2	100 M	High	100 M	High	100 M	High	High
Jik2	100 M	High	100 M	High	100 M	High	High
Jki2	100 M	High	100 M	High	100 M	High	High
Kij2	100 M	High	100 M	High	100 M	High	High
Kji2	100 M	High	100 M	High	100 M	High	High

Here is my observation and conclusion for Q5:

When comparing the blocked (ijk2) and non-blocked (ijk) versions of the GEMM algorithm, it's clear that ijk2 (the blocked version) performs significantly better because it makes more efficient use of the cache. By breaking the matrices into 10x10 blocks, ijk2 reduces cache misses. Each block stays in the cache longer, which means less data needs to be reloaded from memory. On the other hand, ijk (the non-blocked version) accesses the entire matrix in sequence, which doesn't consider the limited size of the cache, causing frequent cache evictions and thus more cache misses.

This same trend is seen with the other loop orders in the dgemm6 series. The blocked versions (like ikj2, jik2, etc.) perform better than the non-blocked ones (ikj, jik, etc.) because blocking helps improve how the cache is used. For large matrices like 10000x10000, the blocked algorithms overall result in fewer cache misses and better performance compared to the non-blocked versions.

For the kji-kji loop order, $A[0][0]$'s first access results in a cache miss, but it is likely reused due to the kji access pattern; $A[17][21]$ is part of a different block from $A[0][0]$, so it results in a cache miss on first access. $B[100][130]$ is accessed based on k and j, resulting in a cache miss the first time it's accessed; $B[101][134]$ is likely in the same block as $B[100][130]$, so it may not result in an additional cache miss. $C[68][90]$'s first access causes a cache miss, but it could be reused if it remains in the cache; $C[2000][1297]$ is far from other accesses, so it will almost certainly cause a cache miss on first access, with additional misses depending on reuse.

- Q6

```
Running dgemm6_ijk2 with Block size = 8x8
n=2048
time=328.280109s
Running dgemm6_ikj2 with Block size = 8x8
n=2048
time=68.150898s
Running dgemm6_jik2 with Block size = 8x8
n=2048
time=348.972856s
Running dgemm6_jki2 with Block size = 8x8
n=2048
time=405.175425s
Running dgemm6_kij2 with Block size = 8x8
n=2048
time=69.834915s
Running dgemm6_kji2 with Block size = 8x8
n=2048
time=419.032592s
Running dgemm6_ijk2 with Block size = 16x16
n=2048
time=317.620499s
Running dgemm6_ikj2 with Block size = 16x16
n=2048
time=68.147403s
Running dgemm6_jik2 with Block size = 16x16
n=2048
time=351.018266s
Running dgemm6_jki2 with Block size = 16x16
n=2048
time=406.106918s
Running dgemm6_kij2 with Block size = 16x16
n=2048
time=69.853406s
Running dgemm6_kji2 with Block size = 16x16
n=2048
time=424.568914s
```

The results from Q6 highlight the impact of blocking sizes on the performance of the GEMM algorithms. I observe that as I increase the block size from 8x8 to 16x16, the execution times are significantly reduced for most of the algorithms. For example, dgemm6_ijk2 saw a drop from 328 seconds to 317 seconds as the block size increased, and similar improvements were seen for the other functions like dgemm6_jik2 and dgemm6_kij2. This shows that a larger block size allows the algorithm to make better use of cache and memory, leading to fewer cache misses and faster execution. However, for some functions like dgemm6_jki2, the gains were less significant, suggesting that the optimal block size might vary depending on the specific loop order. Overall, the results demonstrate the importance of selecting an appropriate block size to improve the efficiency of matrix multiplication algorithms, especially when working with larger matrices like 2048x2048.

- Q7

```
Running with -O0, Block size = 8x8
n=2048
time=13.247669s
Running with -O0, Block size = 16x16
n=2048
time=13.112353s
Running with -O0, Block size = 32x32
n=2048
time=13.108859s
Running with -O1, Block size = 8x8
n=2048
time=3.955963s
Running with -O1, Block size = 16x16
n=2048
time=3.820818s
Running with -O1, Block size = 32x32
n=2048
time=3.946346s
Running with -O2, Block size = 8x8
n=2048
time=3.813755s
Running with -O2, Block size = 16x16
n=2048
time=3.809522s
Running with -O2, Block size = 32x32
n=2048
time=3.817443s
Running with -O3, Block size = 8x8
n=2048
time=3.631409s
Running with -O3, Block size = 16x16
n=2048
time=3.603351s
Running with -O3, Block size = 32x32
n=2048
time=3.612189s
```

The results from Q7 show how different block sizes and optimization levels affect performance. At the -O0 level, where no optimizations are applied, the execution times were the slowest, around 13 seconds for all block sizes. However, moving to -O1 showed a significant improvement, with times dropping to 3.8 to 3.9 seconds, proving that even basic optimizations make a big difference. Further improvements were seen with -O2, although the gains were less pronounced, with times just under 3.8 seconds. The best results came with -O3, with times as low as 3.6 to 3.8 seconds. Interestingly, the block sizes (8x8, 16x16, and 32x32) didn't have a major impact at higher optimization levels, suggesting that compiler optimizations effectively manage memory and caching. In conclusion, while block size adjustments can improve performance, compiler optimizations — especially from -O1 to -O3 — play a much larger role in speeding up matrix multiplication. Thus, focusing on optimization flags seems to be more impactful than tuning block sizes alone.