

PRAC4 : Matrix Multiplication

DIETZ T., JABER A., DONNENFELD T.

November 16, 2025

1 Step 1: Theoretical Analysis

1.1 Iterative Version: Work Partitioning

For the classical iterative matrix multiplication, we parallelize the computation by distributing the **rows of the output matrix** across the available threads. Matrix multiplication has no data dependencies between rows of C because:

$$C[i, j] = \sum_{k=0}^{N-1} A[i, k] B[k, j],$$

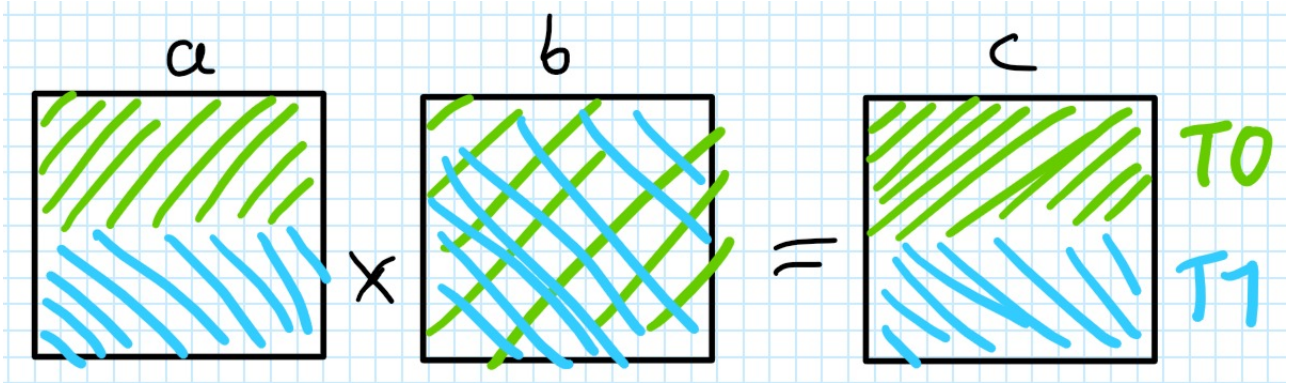
With k threads, we partition rows of C into k blocks

Each thread:

- reads $\frac{N}{k}$ rows of A ,
- reads all of B ,
- writes to disjoint rows of C .

This gives perfect load balance if matrix has data evenly distributed across its rows and columns and no race conditions.

In particular for two threads:



1.2 Divide and Conquer Version

The recursive version repeatedly divides the problem into 8 independent subproblems of size $\frac{SZ}{2} \times \frac{SZ}{2}$ until the base-case size $DQSZ$ is reached.

By analogy with the same 1 dimension problem : we find the number of required splits steps $n = \log_2\left(\frac{N}{DQSZ}\right)$

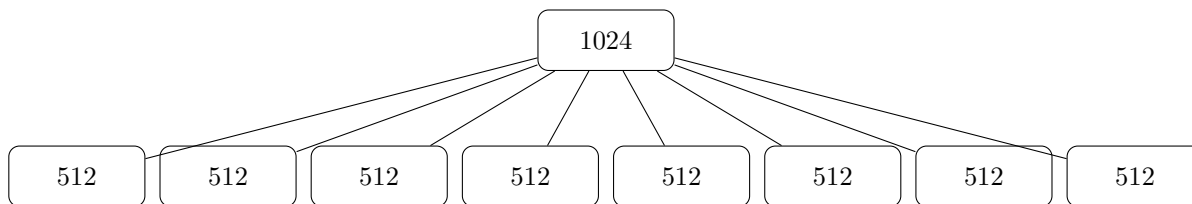
At recursion level i , the number of tasks is 8^i

The total number of tasks is the sum of the terms of the previous geometric sequence, n being the max number of splits seen above. Therefore : $T = \sum_{i=1}^n 8^i = \frac{8^{n+1}-8}{7}$.

The maximum concurrency is the number of "leaf" tasks when all tasks have been created: 8^s

Case for $N = 1024$, $DQSZ = 512$

Here $s = \log_2(1024/512) = 1$, so only one level of recursion.



$$s = 1, \quad T = 8, \quad \text{Maxconcurrency} = 8.$$

Memory Access per Task

We have to be careful with the tasks : sums and products of A and B submatrices can run in parallel, but these results have to be added two by two to form submatrices of C : this causes values in C to be accessed by two tasks possibly at the same time.

2 Step 2: Practical Parallel Implementation

2.1 Iterative Version

We unroll the loops to take advantage of contiguous memory operations and potential SIMD vectorization.

```

1 #pragma omp parallel for
2 for (int i=0; i<N; i+=2)
3     for (int k=0; k<N; k+=2)
4         for (int j=0; j<N; j++)
5             {
6                 type B1 = b[k*N+j];
7                 type B2 = b[(k+1)*N+j];
8
9                 c[i*N+j]      += a[i*N+k]      *B1 + a[i*N+k+1]      *B2;
10                c[(i+1)*N+j] += a[(i+1)*N+k] *B1 + a[(i+1)*N+k+1] *B2;
11            }

```

This reduces memory accesses.

2.2 Divide and Conquer Version

We parallelize the recursion using OpenMP tasks:

```

1 #pragma omp parallel
2 {
3     #pragma omp single
4     {
5         #pragma omp task { ... }
6         #pragma omp task { ... }
7         #pragma omp task { ... }
8         #pragma omp task { ... }
9     }
10 }

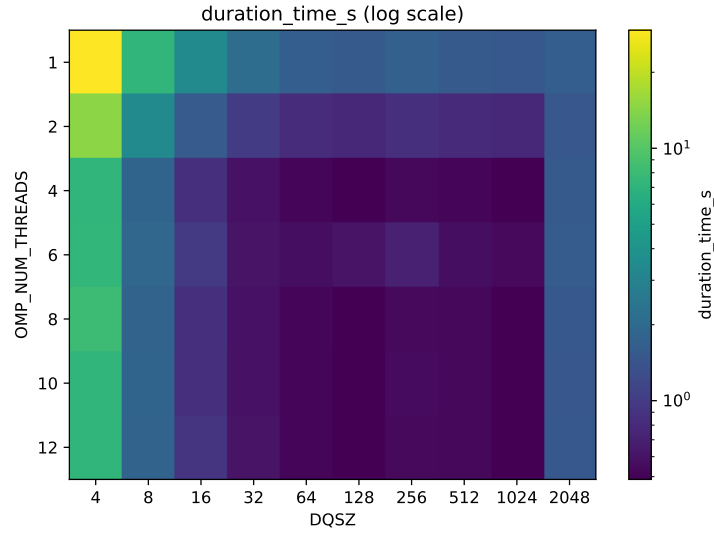
```

3 Step 3: Performance Analysis

We benchmarked $N = 2048$ for a range of thread counts and $DQSZ$ values. Counters collected using `perf stat`:

- execution time,
- instructions, cycles, IPC,
- cache misses,
- speedup and efficiency.

3.1 Heatmap 1: Execution Time

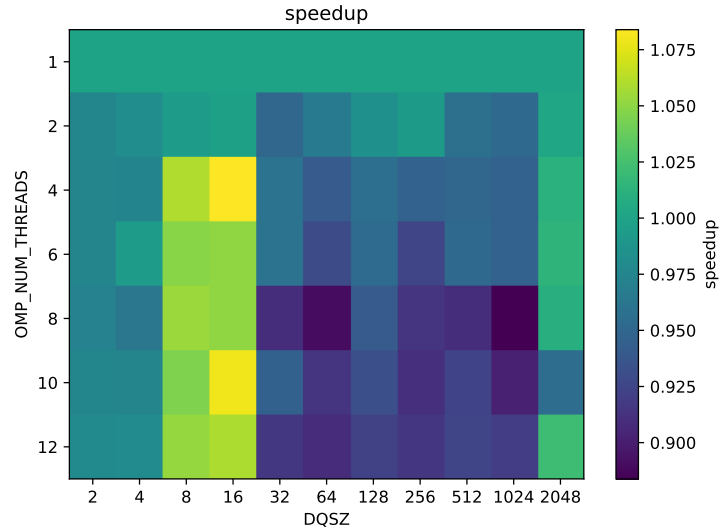


Observations:

- Large and very small $DQSZ$ (4, 8, 2048) yields very poor performance
- Single threaded runs yield very poor performance
- $DQSZ = 64$ or $DQSZ = 128$ produce the best runtimes.
- There seem to be a special increase in performance if the number of thread is a multiple of 4 (looking at nb threads=6 in the graph)

This graph shows the overall best configurations for this problem on the machine.

3.2 Heatmap 2: Speedup (vs same- $DQSZ$, single-thread)

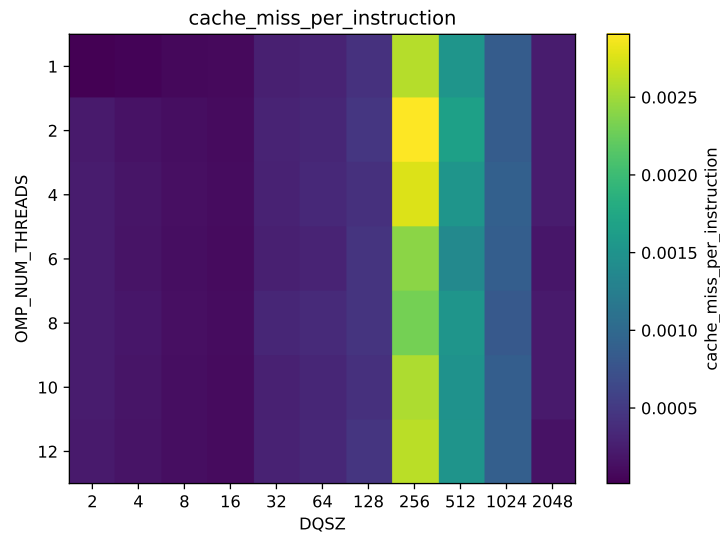


Interpretation:

- Speedup is highest for small base-case sizes (8 and 16).
- Large $DQSZ$ values generate too few tasks, limiting parallelism.
- Small $DQSZ$ values enable enough fine-grained tasks to keep all threads busy.

This graph explains that for some $DQSZ$ we take much more advantage of parallelism, but doesn't prove that global execution time is better (memory bandwidth limitations).

3.3 Heatmap 3: Cache Misses per Instruction



Interpretation :

- Cache misses per instruction are minimal for $DQSZ = 8$ and $DQSZ = 16$.
- Their working sets fit inside caches (L1/L2 ?)

- Larger $DQSZ$ values overflow L1/L2 and drastically increase miss rate.

We find a sweet spot of $DQSZ=16$ for cache optimization, probably related to the size of the L1 cache compared to the $DQSZ$ size in memory.