Compute Bound CPU Matrix Multiplication

Rami Pellumbi*

December 15, 2023

*M.S., Statistics & Data Science

1 Introduction

Inspired by a YouTube video 'Adding Nested Loops Makes this Algorithm 120x FASTER', this project aims to implement a highly optimized matrix multiplication algorithm on CPUs. The video demonstrates how the performance of a naive matrix multiplication algorithm can be improved by simply reordering the loops and implementing a blocking strategy. The video also highlights the importance of memory access patterns in matrix multiplication.

Matrix multiplication is a cornerstone operation in numerous computational fields, ranging from scientific computing to machine learning. At its core, the operation involves the element-wise multiplication and summation of elements across two matrices to produce a third matrix. The theoretical simplicity of this operation belies its computational complexity, particularly when dealing with large matrices. Matrix multiplication scales with the size of the input matrices, often resulting in a significant computational challenge for even modern day processors. This challenge is accentuated by the fact that CPUs, with their limited number of cores and sequential processing capabilities, are often outperformed by GPUs in parallelizable tasks like matrix multiplication. However, understanding and optimizing matrix multiplication on CPUs is crucial, as CPUs are more universally accessible and are often the primary computing resource available in many environments.

The difficulty in optimizing matrix multiplication on CPUs stems from several factors. First, the computational intensity: as the size of the matrices increases, the number of calculations grows cubically, leading to a steep increase in the required computational resources. Second, memory access patterns play a critical role: as the size of the matrices increase, the total memory accesses increase quadratically. Efficient matrix multiplication algorithms must minimize cache misses and effectively utilize the CPU cache hierarchy. This is challenging due to the non-contiguous memory access patterns inherent in matrix multiplication.

The current state of the art in matrix multiplication optimization are built on top of Basic Linear Algebra Subprograms (BLAS). The magic of BLAS lies in its ability to significantly optimize these computationally intensive operations. These routines are meticulously engineered to exploit the underlying architecture of CPUs to their fullest, leveraging techniques such as loop unrolling, blocking for cache, and efficient use of SIMD instructions. These optimizations allow BLAS to achieve performance levels that are often an order of magnitude faster than naive implementations. This project started as 'investigate this videos claims' and ended with the goal of matching the performance of BLAS by implementing a highly optimized matrix multiplication algorithm on CPUs.

2 CPU Specifications

The Intel Xeon(R) Platinum 8268 CPU in the Cascade Lake family is used for profiling. The processor has 24 cores, each with 2 threads, for a total of 48 threads and operates at 3.5GHz. With 512-bit floating-point vector registers and two floating-point functional units, each capable of Fused Multiply-Add (FMA), a Cascade Lake core can deliver 32 double-precision floating-point operations per cycle.[1] This brings the theoretical peak performance of the CPU (with no multithreading enabled) to

$$3.5\ \texttt{GHz} * 24\ \texttt{cores} * 32\ \frac{\texttt{FLOPs}}{\texttt{core}} = 2688\ \texttt{GFLOPs}.$$

Cache Specifications

L1 Each of the 24 cores has a 32K `L1d` cache. It is 8-way set associative, utilizes a write-back policy, and is dedicated to data storage. The cache line size is 64 bytes. Moreover, each core has a 32K `L1i` cache. It is 8-way set associative, utilizes a write-back policy, and is dedicated to instruction storage. The cache line size is 64 bytes.

[1] https://www.nas.nasa.gov/hecc/support/kb/cascade-lake-processors_579.html

L2 Each of the 24 cores has a 1024K `L2` cache. It is 16-way set associative, utilizes a write-back policy, and handles both instructions and data.

L3 There is a single 36608K `L3` cache shared across all cores. It is 11-way set associative, utilizes a write-back policy, and handles both instructions and data.

Memory Details

The processor has a maximum theoretical memory bandwidth of 131.13 `GiB/s`. Each `double` takes up 8 bytes of memory. This means that the CPU memory bandwidth is

$$\texttt{140.8GB/s} \cdot \frac{\texttt{double}}{\texttt{8B}} = 17.6 \texttt{ GFlops/s}.$$

3 The Problem

Matrix multiplication inherently becomes compute-bound and delves into the nuances of compute and memory access that escalate as matrix sizes scale.

3.1 Compute-Bound Nature of Matrix Multiplication

The essence of a compute-bound task lies in its primary limitation being the processing power of the CPU rather than I/O or memory speeds. As illustrated in the Section 2, the theoretical peak performance of the CPU is 2688 `GFLOPs`. However, the theoretical peak memory bandwidth is only 17.6 `GFlops/s`; 152x slower. Thus, any task that requires more than 152 CPU cycles per memory access will be compute bound (and any task requiring less than 152 CPU cycles per memory access is memory bound).

Matrix multiplication, especially for large matrices, is intensely compute-intensive. The standard algorithm for matrix multiplication involves three nested loops, iterating over the rows and columns of the input matrices. For two matrices of size $N \times N$, this results in a computational complexity of $O(N^3)$. Each element in the resultant matrix is computed by taking the dot product of a row from the first matrix and a column from the second, involving N multiplications and $N - 1$ additions. As N increases, the number of arithmetic operations grows cubically, making the task increasingly compute-bound.

3.2 Memory Access Patterns and Their Impact

Apart from the computational intensity, memory access patterns significantly influence the performance of matrix multiplication algorithms. As matrix sizes grow, the memory footprint of these matrices also expands, often surpassing the size of the CPU caches, e.g., multiplying 2 $N \times N$ double precision matrices requires

$$3 * N * N * 8 \texttt{ bytes},$$

which exceeds the size of the `L3` cache at just $N = 1250$. This mismatch leads to frequent cache misses and the necessity to fetch data from the slower main memory, introducing latency.

The non-contiguous memory access patterns in matrix multiplication exacerbate this issue. Accessing elements row-wise in one matrix and column-wise in another leads to strided accesses, which are inefficient for cache utilization. In large matrices, these strided accesses mean that the processor cannot effectively prefetch data, leading to increased cache misses and memory latency.

3.3 The Challenge of Optimizing for Larger Matrices

Optimizing matrix multiplication for larger matrices thus becomes a dual challenge: managing the cubic growth in computational requirements and minimizing memory latency due to inefficient access patterns. Effective optimization requires a nuanced understanding of both the computational architecture and the

memory hierarchy of CPUs. Techniques like loop reordering, tiling, or blocking, which aim to maximize data locality and cache utilization, become critical. However, these techniques must be meticulously tailored to the specific architecture, as the optimal configuration can vary significantly based on factors like cache size, cache line size, and the number of cores.

4 Approach 1: Keep it Serial

The first batch of programs focused on optimizin a serial matrix multiplication algorithm. First, the naive algorithm was implemented. Then, the loops were reordered to improve cache performance. Finally, a blocking strategy was implemented to further improve cache performance. All algorithms were written to compute $C = A \times B$, where A is an $N \times P$ matrix and B is a $P \times M$ matrix. The resultant matrix C is of course $N \times M$. All matrix multiplies were compared against the BLAS implementation of matrix multiplication (`dgemm`) for correctness. Unless otherwise stated, matrices are stored in row-major order.

4.1 The Naive Algorithm

The naive algorithm is the most straightforward implementation of matrix multiplication. It uses the standard `ijk` access pattern and is implemented as follows:

```
for (int i = 0; i < N; i++)
{
    int iA = i * P; // get to index for start of row i
    for (int j = 0; j < M; j++)
    {
        int iC = i * M + j; // row i column j of C

        double cvalue = 0.;
        for (int k = 0; k < P; k++)
        {
            int jB = k * M + j;
            // row i column k of A multiplied with row k column j of B
            C[iC] += A[iA + k] * B[jB];
        }
    }
}
```

This is terribly inefficient for a few reasons:

1. The innermost loop performs 2 loads and 1 store.
2. B is accessed column-wise $(k * M + j)$. This access pattern is inefficient in a row-major storage system since elements are not contiguous in memory. This leads to a compulsory miss rate of 1.[2]
3. There is poor temporal locality - only somewhat utilized in accessing elements of A and C, as the same elements of these matrices are accessed multiple times within the inner loops. Each has a compulsory miss rate of 0.125.

The code is so ill performing that only select matrix sizes were evaluated. The results are shown in Table 1. The results are in seconds and are the average of 3 runs. The MKL libraries `dgemm` implementation is included for comparison.

[2] A compulsory miss refers to the cache miss that occurs when the first access to a block is not in the cache. The cache line size is 64 bytes. Thus, for access to a consecutive doubles (8 bytes), the compulsory miss rate is $8/64 = 0.125$ (in ideal conditions).

Table 1: ijk Access Pattern Mean Performance (s)

N	P	M	Mean Time	Mean Blas Time
1024	1024	1024	1.691292	0.0607420
2048	2048	2048	16.156843	0.2742953

The Intel MKL `dgemm` implementation is 28x faster than the naive implementation for $N = 1024$ and 59x faster for $N = 2048$. The naive implementation is clearly not scalable.

4.2 Loop Reordering

The next step was to attempt to remedy the access pattern inefficiencies by reordering the loops. The `kij` access pattern was chosen. This pattern is implemented as follows:

```
for (int k = 0; k < P; k++)
{
    for (int i = 0; i < N; i++)
    {
        int iA = i * P + k; // get row i column k of A for reuse
        double r = A[iA];
        for (int j = 0; j < M; j++)
        {
            int iB = k * M + j; // row k column j of B
            int iC = i * M + j; // row i column j of c
            C[iC] += r * B[iB];
        }
    }
}
```

The `kij` access pattern offers significant improvements over the `ijk` pattern for matrix multiplication, primarily due to its more efficient use of the CPU cache. The outermost loop iterates over the k-dimension (columns of A), followed by the i (rows of A) and j (columns of B) dimensions. This restructuring has significant implications for memory access patterns.

1. The compulsory miss rate of A is reduced to zero (as opposed to 0.125 in the `ijk` pattern).
2. The compulsory miss rate of B is reduced to 0.125 (as opposed to 1 in the `ijk` pattern).
3. The compulsory miss rate of C remains at 0.125 (as opposed to 0.125 in the `ijk` pattern).

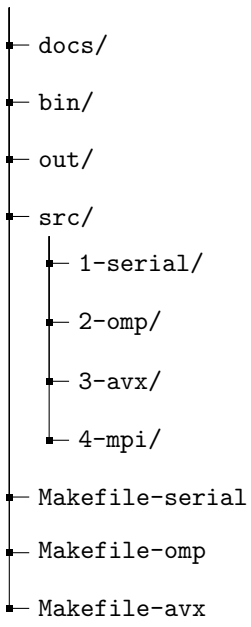
The results are shown in Table 2. The results are in seconds and are the average of 3 runs. The MKL libraries `dgemm` implementation is included for comparison.

Table 2: kij Access Pattern Mean Performance (s)

N	P	M	Mean Time	Mean Blas Time
1024	1024	1024	0.3560277	0.0552923
1024	1024	8192	5.2299977	0.2942117
8192	1024	8192	46.0622990	2.0431417
8192	8192	1024	39.8449800	1.9560290
8192	8192	8192	370.1064950	15.4767510

Appendix A1: Project Organization

The project is laid out as follows:



- **docs/**: This folder contains LaTeX files and other documentation materials that pertain to the report.
- **bin/**: The `bin` folder holds compiled objects and executable files, centralizing the output of the compilation process.
- **out/**: The `out` folder stores the outputs from each task. It also houses the csv file containing data generated by the programs.
- **src/**: This directory houses the source files (`.c`) that make up the benchmarks. Each subdirectory contains the source files for the relevant experiments.
- **Shell Scripts**: The shell scripts are used to submit the job for the relevant task to slurm via `sbatch`.
- **Makefiles**: The Makefiles are used to compile the code. There is a Makefile for each set of experiments. The relevant shell scripts make use of the appropriate Makefile.

Appendix A2: Code Explanation, Compilation, and Execution

The provided Bash scripts automate the entire process, making it straightforward to compile and run the code. All the below steps assume you are in the root of the project directory.

7.1 Automated Building and Execution

All related code is in the `src/` directory. There are multiple programs, each batch of which is contained in its own subdirectory.

Serial Programs

- `t1-serial-ijk.c`: Standard serial matrix multiplication.
- `t2-serial-kij.c` Serial matrix multiplication algorithm with better cache performance.
- `t3-serial-blocking.c` Serial tiled matrix multiplication.
- `t4-serial-blocking-T.c` Serial tiled matrix multiplication where B is transposed.

Open MP Programs

- `t5-omp.c`: Utilized OpenMP to parallelize the serial matrix multiplication algorithm in `t4`.
- `t6-omp-divisible-local-blocks.c`: Fixes the false sharing problem in `t5`: which is when multiple cores are accessing the same cache line on the shared cache. This program assumes matrix dimensions are multiples of the chosen block (tile) size.
- `t7-omp-non-divisible-local-blocks.c`: Same as `t6` but does not assume matrix dimensions are multiples of the chosen block size.

AVX Programs

These programs are inspired by [1], [2], and [3].

- `t8-serial-divisible-avx-blocking.c`: Serial matrix multiplication with AVX instructions. This program assumes that the matrix dimensions are multiples of the hard coded optimal block sizes.
- `t9-omp-divisible-avx-blocking.c`: Parallel matrix multiplication with OpenMP and AVX instructions. This program assumes that the matrix dimensions are multiples of the hard coded optimal block sizes.
- `t10-omp-non-divisible-avx-blocking.c`: Parallel matrix multiplication with OpenMP and AVX instructions. This program does not assume anything about the matrix dimensions and can handle non-multiples of the hard coded optimal block sizes.

To run any one set of experiments, simply run the relevant Bash script. For example, to run the serial experiments, submit the `build-run-serial.sh` script via `sbatch`: `sbatch build-run-serial.sh`. This will compile the code and run the experiments. The output files will be stored in the `out/` directory.

7.2 Post-Build Objects and Executables

Upon successful compilation and linking, an `obj/` subdirectory will be generated within the directory. This directory will contain the compiled output files. Additionally, the executable files for running each program will be situated in the `bin/` subdirectory.

7.3 Output Files From `sbatch`

The output files generated from running the code by submitting the relevant Bash script via `sbatch` will be stored in the `out` directory.

References

[1] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. Anatomy of high-performance many-threaded matrix multiplication. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1049–1059, 2014.

[2] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), may 2008.

[3] Stefan Hadjis. Blas-level cpu performance in 100 lines of c. Website, 2015.