Compute Bound CPU Matrix Multiplication

Rami Pellumbi*

December 14, 2023

*M.S., Statistics & Data Science

1 Introduction

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 aims to match 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.
The cache specifications are:

- 24×32K `L1d` cache: 8-way set associative, write-back policy, dedicated to data storage. The cache line size is 64 bytes.
- 24×32K `L1i` cache: 8-way set associative, stores instructions for execution. The cache line size is 128 bytes.
- 24×1024K `L2` cache: 16-way set associative, write-back policy, inclusive, handles both instructions and data.

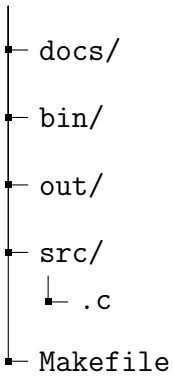
- $1 \times$ 36608K `L3` cache: 11-way set associative, write-back policy, shared across all cores, non-inclusive, handles both instructions and data.

3 Approach

The first thing done is computing the theoretical peak performance of matrix multiplication on the CPU via Intel's Math Kernel Library (MKL). This is done by running the `mkl_peak` program

4 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.
- **Shell Scripts**: The shell scripts are used to submit the job for the relevant task to slurm via `sbatch`.

5 Code Explanation, Compilation, and Execution

This section outlines the steps required to build and execute the code. 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.

5.1 Automated Building and Execution

All related code is in the `src/` directory. There are multiple programs:

-

To run any one experiment,

5.2 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.

5.3 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.

6 Conclusion