Benchmark of Matrix Multiplication

Jesus Arencibia Falcon

July 2024

Abstract

This paper will discuss various methods for implementing matrix multiplication, along with their advantages and disadvantages for different situations. Languages explored are Python, Java, C++ and Rust, with methods including naive, improving cache hits, parallelization, vectoring and trying to leverage the GPU.

1 Introduction

Matrix multiplication is becoming increasingly important in the modern world. Two big examples are machine learning and graph theory, both of which are becoming more and more important with the rise of big data. Thus knowing what the best ways to implement matrix multiplication in various situations are crucial for the future of computing.

In this paper, we will explore various ways to speed up matrix multiplication. One major use for matrix multiplication in the past decade is in deep machine learning, as they are used to represent the weights and biases of neural networks, such that a forward pass can be done by multiplying the input by the weight matrix and adding the bias vector, limiting the speed of the forward pass to the speed of matrix multiplication.

2 Methodology

This study explores the performance optimization of matrix multiplication in Java. Java, as a high-level programming language, provides several features that make it suitable for high-performance computing, including robust multithreading capabilities and an extensive standard library for mathematical operations. The efficiency of Java enables faster execution of arithmetic operations involved in matrix multiplication, and its built-in concurrency utilities facilitate the implementation of algorithms.

The matrix multiplication was performed on three matrices, A, B, and C, each of size 1024×1024. Matrices A and B were populated with random values between 0 and 1, while matrix C was initialized to zero. To implement the matrix multiplication, a custom function was created that takes matrices A and B as inputs and computes their product, storing the result in matrix C. The function leveraged Java's concurrency utilities to achieve parallel execution, distributing the computational workload across multiple threads. An ExecutorService was used to manage a pool of threads, and the matrix multiplication task was divided into smaller subtasks, each handled by a separate thread. Proper synchronization mechanisms ensured the correctness of the parallel computation, with results from individual threads aggregated to produce the final output matrix C.

3 Conclusion

Observing the results of the different methods used, we realize that the Java implementation starts out just as slow as Python, but for larger matrices it begins to be faster, becoming as fast as C++.

In conclusion, when it comes to implementing matrix multiplication, a python implementation is not recommended, as it is unaffordably slow. Altough it should be noted that it is possible to create python libraries that use C or C++ under the hood, which would be much faster while still using pyhton for all the more high level code for prototyping and such. When it comes to java, it is a good language for implementing matrix multiplication, becoming as fast as C for large matrices. Given that it is a widely used language, this could lead to it being an acceptable choice for various applications that are written in Java. When it comes to C, it is a good language for implementing matrix multiplication, knowing that it is known to be highly portable, and it is also known to be fast, it is a good choice for implementing matrix multiplication for almost any application, altough it should be considered that C itself is not a great choice for prototyping or fast iterations as it is a low level language. It should be noted that for most modern deep learning applications, something similar to the rust implementation is used, using libraries such as pytorch.

4 Future Work

Future research and development in the field of matrix multiplication, particularly focusing on optimization, can explore several promising avenues:

Adaptive Compression Techniques: Investigate dynamic or adaptive compression methods that adjust based on matrix characteristics to enhance storage efficiency and computational performance.

Scalability and Benchmarking: Scale experiments to larger matrices and diverse computing environments to comprehensively understand the performance characteristics and limitations of methods.

Real-World Applications: Apply optimized matrix multiplication techniques to practical engineering and data processing tasks to validate their effectiveness and utility across various industries.

Exploring these areas could lead to significant advancements in computational methodologies, enhancing the efficiency, scalability, and applicability of matrix multiplication techniques in diverse computational domains.