Aryan Saadati
301472611

Greg Baker
CMPT 295 - Mini Project

Matrix Multiplication Optimization Report

Introduction:

The goal of this report is to showcase multiple algorithms for calculating the product of two matrices. The report presents and compares various performance measurements such as time complexity, operation duration, memory locality, and memory cache usage.

All the operation agree on a set of assumptions which include:

- All of the matrices are implemented using a 1 dimensional array.
- All of the matrices are square and have a length divisible by 8 to accommodate the use of vector SIMD instructions.
- All of the matrices are formed of 32-bit floating point numbers (i.e. the float data type in C).

Implementations:

Basic Matrix Multiplication in Cpp and Assembly: Time Complexity O(n^3)

Overview:

This algorithm uses a triply nested loop that accesses rows of the first matrix and columns of the second matrix. The inner loop multiplies each entry on the rows with their respective entry on the column and accumulates the dot product into the appropriate entry in the result matrix. The Assembly implementation uses the same logic.

Performance:

- Without optimization, the Assembly implementation is on average twice as fast as the C++ implementation.
- Using the -O3 optimization flag, the C++ implementation becomes twice as fast as the Assembly implementation.

	Assembly	Assembly	Cpp	Cpp	Cpp
Size	512 x 512	2048 x 2048	512 x 512	512 x 512	2048 x 2048
Optimization	N/A	N/A	X	✔	✔
Resulted Time:	540.2 ms	107380.1 ms	1091.9 ms	218.9 ms	101640.4 ms

This implementation performs poorly due to terrible memory locality. On an array of length 2048, or 4194304 numbers, cachegrind reports a total of 134,582,921 L1 cache misses out of 418,313,593 total references. That is a 30% L1 cache miss rate caused by the algorithm's dependence on accessing columns of numbers in a linear array.

SIMD Vector Matrix Multiplication: Time Complexity O(n^3)

Overview:

This algorithm uses a quadruple nested loop. Instead of reading one number at a time, it reads eight values into a floating point register. The fourth innermost loop reads eight values from a column into an array of size 8, which is then loaded into a floating point register. Despite the fourth loop, the time complexity remains O(n^3).

Performance:

- This algorithm performs poorly on memory locality.
- Without optimization, it is slower than basic matrix multiplication.
- Using -O3 optimization, it performs 20% faster than basic matrix multiplication.
- Assembly implementation is faster than unoptimized code but falls short compared to optimized C++ code.

	Assembly	Assembly	Cpp	Cpp	Cpp
Size	512 x 512	2048 x 2048	512 x 512	512 x 512	2048 x 2048
Optimization	N/A	N/A	X	✔	✔
Resulted Time:	296.8 ms	125689.4 ms	1218.7 ms	177.6 ms	111345.3 ms

Tiled (Blocked) Matrix Multiplication: Time Complexity O(n^6)

Overview:

This algorithm uses six nested loops. The first three loops read chunks of memory from RAM, and the second three perform calculations on the data read.

Performance:

- Despite the O(n^6) time complexity, the number of L1 cache misses significantly decreases to 2%.
- No Assembly implementation was done, as optimized C++ code consistently outperformed Assembly.

	Cpp	Cpp	Cpp
Size	512 x 512	512 x 512	2048 x 2048
Optimization	X	✔	✔
Resulted Time:	568.8 ms	93.6 ms	7763.2 ms

This implementation significantly outperforms any of the other implementations, specifically on a large size of data input where it is about 15 times faster on average.

Threaded Matrix Multiplication: Time Complexity O(n^3)

Overview:

The Threaded Matrix Multiplication implementation uses the Basic Matrix Multiplication algorithm but instead uses threads to try to optimize the algorithm.

Performance:

- There was not any significant gain especially in large array sizes.
- This algorithm also had the same downfall which was the large amount of L1 cache misses.

- My workstation has an older generation CPU with a small cache size so even with my attempt at utilizing all of its 8 cores using the cpp threads library it seems like the cache is just not enough.

	Cpp	Cpp	Cpp
Size	512 x 512	512 x 512	2048 x 2048
Optimization	X	✔	✔
Resulted Time:	1029.4 ms	190.8 ms	100814.8 ms

Threaded and Tiled Matrix Multiplication:

Overview:
This approach combines the good memory locality of the tiled algorithm with multi-threading.

Performance:

- There was not much of a speed improvement when using the multithreaded tiled algorithm in comparison to the non threaded implementation.

Here are the two comparisons side by side:

	Tiled	Tiled/Threads	Tiled	Tiled/Threads
Size	512 x 512	512 x 512	512 x 512	512 x 512
Optimization	X	X	✔	✔
Resulted Time:	1029.4 ms	785.6 ms	190.8 ms	199.0 ms

	Tiled	Tiled/Threads	Tiled	Tiled/Threads	Tiled	Tiled/Threads
Size	2048 x 2048	2048 x 2048	4096 x 4096	4096 x 4096	8192 x 8192	8192 x 8192
Optimization	✔	✔	✔	✔	✔	✔
Resulted Time:	7514.9 ms	7086.7 ms	68888.9 ms	67993.3 ms	635652.2 ms	654762.6 ms

Conclusion:

The primary challenge in optimizing matrix multiplication algorithms is poor memory locality due to accessing columns in a linear array, leading to significant cache misses. The basic and SIMD implementations suffered from this issue, with SIMD showing minimal improvement. The tiled algorithm significantly improved performance by reducing cache misses through efficient memory access. Threaded implementations, especially on a shared CSIL CPU, failed to yield significant gains, likely due to limited cache size and resource sharing. The combined tiled and threaded approach offered minimal additional benefits. Efficient memory management remains crucial for optimizing matrix multiplication and in my case overshadowing the benefits of parallel processing.