ECSE 4961 Proj 1

Ryan Walton

January 31, 2022

1 Introduction

Matrix multiplication is an important, frequently used operation that can be very computationally intensive given its $O(n^3)$ time complexity. This project explores some optimization techniques, specifically the SSE and AVX SIMD instructions available on modern Intel laptop CPUs.

2 Matrix Multiplication Exploration

Matrix multiplication of two matrices is shown below. Matrix A has n rows and p columns, it an nxp matrix, and Matrix B is a pxn matrix. The number of columns in A must match the number of rows in B, a property of matrix multiplication that will be come obvious shortly. The result matrix must match A in number of rows, and B in number of columns

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1p} \\ a_{21} & a_{22} & \dots & a_{2p} \\ \vdots & \vdots & \ddots & \\ a_{n1} & a_{n2} & & a_{np} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & \vdots & \ddots & \\ b_{p1} & b_{p2} & & b_{pm} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1m} \\ c_{21} & c_{22} & \dots & c_{2m} \\ \vdots & \vdots & \ddots & \\ c_{n1} & c_{n2} & & c_{nm} \end{bmatrix}$$

Each element in the result matrix C is the dot product of the associated row vector of A and column vector of B. One of the simplest matrix multiplication algorithms follows this property closely. The algorithm essentially sums the products for each row. However, we will see that this method is nowhere near the most optimized for x86 processors.

$$c_{n_0 m_0} = a_{n_0 1} \cdot b_{1 m_0} + a_{n_0 2} \cdot b_{2 m_0} + \dots + a_{n_0 p} \cdot b_{p m_0}$$

2.1 Chunking the Multiplication

Since each element of the result matrix depends only on a row vector of A and a column vector of B, the multiplication operation can be chunked according to blocks of the result matrix. In the case below, we create a chunk of the matrix multiplication corresponding to an 8x8 section of the result matrix.

Further, we can divide this multiplication by performing multiplication on each matching column vector of A and row vector of B. The result of this multiplication will form an 8x8 matrix, and summing each of these 8x8 resultant matrices will yield the previously chunked 8x8 matrix.

$$\begin{bmatrix} a_{n_0 1} & a_{n_0 2} & \dots & a_{n_0 p} \\ a_{n_1 1} & a_{n_1 2} & \dots & a_{n_1 p} \\ \vdots & \vdots & \ddots & \\ a_{n_7 1} & a_{n_7 2} & & a_{n_7 p} \end{bmatrix} \cdot \begin{bmatrix} b_{1m_0} & b_{1m_1} & \dots & b_{1m_7} \\ b_{2m_0} & b_{2m_1} & \dots & b_{2m_7} \\ \vdots & \vdots & \ddots & \\ b_{pm_0} & b_{pm_1} & & b_{pm_7} \end{bmatrix} = \begin{bmatrix} c_{n_0 m_0} & c_{n_0 m_1} & \dots & c_{n_0 m_7} \\ c_{n_1 m_0} & c_{n_1 m_1} & \dots & c_{n_1 m_7} \\ \vdots & \vdots & \ddots & \\ c_{n_7 m_0} & c_{n_7 m_1} & & c_{n_7 m_7} \end{bmatrix} =$$

$$\begin{bmatrix} a_{n_0 1} \\ a_{n_1 1} \\ \vdots \\ a_{n_7 1} \end{bmatrix} \cdot \begin{bmatrix} b_{1m_0} & b_{1m_1} & \dots & b_{1m_7} \end{bmatrix} + \begin{bmatrix} a_{n_0 2} \\ a_{n_1 2} \\ \vdots \\ a_{n_7 2} \end{bmatrix} \cdot \begin{bmatrix} b_{2m_0} & b_{2m_1} & \dots & b_{2m_7} \end{bmatrix} + \begin{bmatrix} a_{n_0 p} \\ a_{n_1 p} \\ \vdots \\ a_{n_7 p} \end{bmatrix} \cdot \begin{bmatrix} b_{pm_0} & b_{pm_1} & \dots & b_{pm_7} \end{bmatrix}$$

2.2 Base Arithmetic Operation

This 8x1 x 1x8 matrix multiplication forms our smallest arithmetic step of the matrix, an operation that can be performed very quickly with simd instructions. Instead of performing each multiplication on its own, and assigning the product to the corresponding result matrix index, we can multiply 8 elements at once with vectorized math. The first operation is the point-wise multiplication of the undisturbed column vector of A and row vector of B. This results in a vector of the diagonals of the result matrix.

$$\begin{bmatrix} a_{n_0 1} \\ a_{n_1 1} \\ \vdots \\ a_{n_7 1} \end{bmatrix} \odot \begin{bmatrix} b_{1 m_0} \\ b_{1 m_1} \\ \vdots \\ b_{1 m_7} \end{bmatrix} + \begin{bmatrix} a_{n_0 2} \\ a_{n_1 2} \\ \vdots \\ a_{n_7 2} \end{bmatrix} \odot \begin{bmatrix} b_{2 m_0} \\ b_{2 m_1} \\ \vdots \\ b_{2 m_7} \end{bmatrix} + \cdots + \begin{bmatrix} a_{n_0 p} \\ a_{n_1 p} \\ \vdots \\ a_{n_7 p} \end{bmatrix} \odot \begin{bmatrix} b_{p m_0} \\ b_{p m_1} \\ \vdots \\ b_{p m_7} \end{bmatrix} = \begin{bmatrix} c_{n_0 m_0} \\ c_{n_1 m_1} \\ \vdots \\ c_{n_7 m_7} \end{bmatrix}$$

$$\begin{bmatrix} c_{n_0 m_0} & & & & & & & \\ & c_{n_1 m_1} & & & & & & \\ & & c_{n_2 m_2} & & & & & \\ & & & c_{n_3 m_3} & & & & \\ & & & & c_{n_4 m_4} & & & \\ & & & & & c_{n_5 m_5} & & \\ & & & & & & c_{n_6 m_6} & \\ & & & & & & & c_{n_7 m_7} \end{bmatrix}$$

The row vector of B is then modified such that each element is shifted left one position, and the leftmost element wraps to the rightmost position. When point-wise multiplication of A and B is performed now, the result is the shifted diagonals shown below. This is repeated 8 times total, in order to generate the entire result matrix.

$$\begin{bmatrix} a_{n_0 1} \\ a_{n_1 1} \\ \vdots \\ a_{n_7 1} \end{bmatrix} \odot \begin{bmatrix} b_{1 m_1} \\ b_{1 m_2} \\ \vdots \\ b_{1 m_7} \\ b_{1 m_0} \end{bmatrix} + \begin{bmatrix} a_{n_0 2} \\ a_{n_1 2} \\ \vdots \\ a_{n_7 2} \end{bmatrix} \odot \begin{bmatrix} b_{2 m_1} \\ b_{2 m_2} \\ \vdots \\ b_{2 m_7} \\ b_{2 m_0} \end{bmatrix} + \cdots + \begin{bmatrix} a_{n_0 p} \\ a_{n_1 p} \\ \vdots \\ a_{n_7 p} \end{bmatrix} \odot \begin{bmatrix} b_{p m_1} \\ b_{p m_2} \\ \vdots \\ b_{p m_7} \\ b_{p m_0} \end{bmatrix} = \begin{bmatrix} c_{n_0 m_1} \\ c_{n_1 m_2} \\ \vdots \\ c_{n_1 m_7} \\ c_{n_7 m_0} \end{bmatrix}$$

$$\begin{bmatrix} & c_{n_0 m_1} & & & & & & \\ & & c_{n_1 m_2} & & & & & \\ & & & c_{n_2 m_3} & & & & \\ & & & & c_{n_3 m_4} & & & \\ & & & & & c_{n_4 m_5} & & \\ & & & & & & c_{n_5 m_6} & \\ & & & & & & & c_{n_6 m_7} \\ c_{n_7 m_0} & & & & & & & \end{bmatrix}$$

3 Optimization Techniques

3.1 Cache Optimization Chunking

Cache is of limited size, so in order to reduce time spent waiting on cache misses, we can chunk the multiplication into sizes whose operations use data that can fit in the cache. One way to chunk is to take an $nxnxn$ chunk, where an nxn chunk of A multiplies with an nxn chunk of B in order to modify an nxn chunk of C. Thus, the cache space required is about $3b^2n^2$, where b is the size of the data type in bytes. A good starting value for n would be $\sqrt{\frac{C}{3b^2}}$, where C is the chache size in bytes.

3.2 Row vs Column Major Storage

In order to reduce time for loading of column vectors of A and row vectors of B, the multiplication is performed on two matrices that are stored differently. A is stored in column major fashion, meaning its stored as an array of columns, in which each column is an array of row-wise indices. B is stored in row major fashion, meaning it is stored as an array of rows, in which each row is an array of column-wise indices. In this way, access of a column vector of A is accessing a block of contiguous memory, and access of row vector B is accessing a block of contiguous memory.

3.3 SIMD Diagonal-wise Multiplication

The diagonal multiplication property of matrix multiplication, described earlier, avoids the costly horizontal addition problem inherent in dot product calculation.

An attribute of this design is a very low number of instructions for the inner loop, which is called on the order of $nxmxp$ (n^3 on square matrices). Two loads are performed, 8 fused multiply add (fma, if available, otherwise 8 mul + 8 + add), and 8 rotations. Extra time incurred by accessing and storing diagonals (which obviously cannot be stored in contiguous memory) is irrelevant as it is only called on the order of nxm (or n^2 for square).

3.4 Multithreading

Matrix multiplication can be paralleled due to it's chunking property described earlier. Each CPU will only need to compute a portion of the multiplication, yielding faster runtimes for larger matrices

4 Algorithms

To test the impact of the different optimization techniques, 6 different algorithms were generated. They are numbered with increasing complexity (and, as it turns out, increased performance!), each algorithm adding to the previous. All 6 algorithms are implemented for float multiplication, but only `mul0`, `mul1`, and `mul5` are implemented for `float`, `double`, `int16_t`, and `int32_t`.

4.1 `mul0`

Doesn't implement any of the optimization techniques

```
for each element in resultant matrix:
    element = dot product A row vector and B column vector
```

4.2 `mul1`

Implements Cache Optimization Chunking

```
zero initialize resultant matrix
for each chunk in resultant matrix:
    for each element in chunk:
        element += dot product A row vector and B column vector
```

4.3 `mul2`

Implements SIMD Diagonal-wise Multiplication

```
for each 8x8 section of result matrix:
    set diagonals[0:7] to zero
    for each p:
        load column vector of A into _a element-wise
        load row vector of B into _b element-wise
        for each diagonal:
            diagonals[i] = _a * _b + diagonals[i] // vectorized math
            rotate _b to the left by one index
    store diagonals[0:7] to result matrix chunk
```

4.4 mul3

Implements SIMD Diagonal-wise Multiplication and Cache Optimization Chunking.

```
for each chunk of result matrix:
    for each 8x8 section of chunk:
        load diagonals[0:7] from result matrix
        for each p:
            load column vector of A into _a element-wise
            load row vector of B into _b element-wise
            for each diagonal:
                diagonals[i] = _a * _b + diagonals[i] // vectorized math
                rotate _b to the left by one index
        store diagonals[0:7] to result matrix chunk
```

4.5 mul4

Implements SIMD Diagonal-wise Multiplication, Cache Optimization Chunking, and Row vs Column Major Storage.

```
for each chunk of result matrix:
    for each 8x8 section of chunk:
        load diagonals[0:7] from result matrix
        for each p:
            load column vector of A into _a block-wise
            load row vector of B into _b block-wise
            for each diagonal:
                diagonals[i] = _a * _b + diagonals[i] // vectorized math
                rotate _b to the left by one index
        store diagonals[0:7] to result matrix chunk
```

4.6 `mul5`

Implements SIMD Diagonal-wise Multiplication, Cache Optimization Chunking, Row vs Column Major Storage, and Multithreading

```
process( chunk )
    for each 8x8 section of chunk:
        load diagonals[0:7] from result matrix
        for each p:
            load column vector of A into _a block-wise
            load row vector of B into _b block-wise
            for each diagonal:
                diagonals[i] = _a * _b + diagonals[i] // vectorized math
                rotate _b to the left by one index
        store diagonals[0:7] to result matrix chunk

for chunk in matrix:
    queue.push( chunk )

thread_func()
    while queue.get( chunk ):
        process( chunk )
    done_count++

threads = create n threads ( thread_func )

while done_count < n:
    spin
```

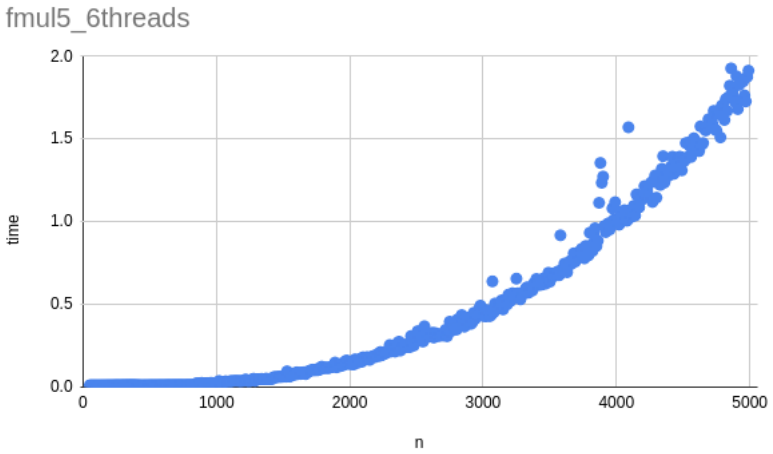
5 Results

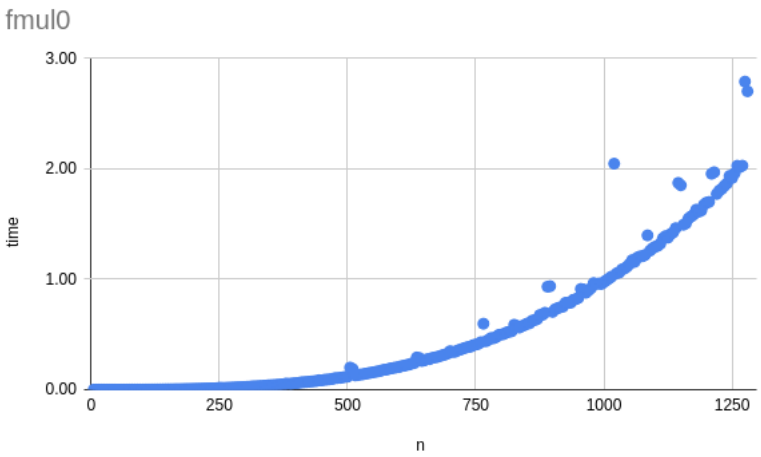
The results of the trials are shown in the tables below. The `mul5_1thread` trial emulates `mul4` for double, `int16_t`, and `int32_t` which do not have `mul4` implementations.

1x1 x 1x1 Multiplication				
	float	double	int16_t	int32_t
mul5_6threads	0.0103311	0.0106091	0.0104133	0.0102404
mul5_1thread		0.0101652	0.0103478	0.0104085
mul4	2.1844e-05			
mul3	1.582e-06			
mul2	2.364e-06			
mul1	1.245e-06	1.85e-06	2.315e-06	2.971e-06
mul0	1.913e-06	1.917e-06	1.646e-06	2.134e-06

1000x1000 x 1000x1000 Multiplication				
	float	double	int16_t	int32_t
mul5_6threads	0.0243238	0.0478476	0.0223278	0.0347667
mul5_1thread		0.117359	0.0623525	0.11409
mul4	0.0582221			
mul3	0.095679			
mul2	0.176258			
mul1	0.750126	0.784101	0.446072	1.19678
mul0	0.923172	1.30934	0.807685	0.996054

10000x10000 x 10000x10000 Multiplication				
	float	double	int16_t	int32_t
mul5_6threads	14.5277	35.254	15.2274	24.0364
mul5_1thread		129.596	69.9017	110.88
mul4	59.6739			
mul3	102.459			
mul2	259.664			
mul1	800.621	878.397	516.979	1216.54
mul0	5657.6	7109.83	4386.27	5444.66





6 Conclusion

As can be seen in the results tables, the more complex algorithms are typically not preferable for small matrices, but as the size increases, the benefits of the optimizations become very apparent. Fastest optimization is over 100 times faster than the non-optimized version for large matrices.

The two plots show times to calculate matrices of increasing size. The time complexity for both algorithms is on the order of n^3, as can be seen in the graphs. However, fmul5 using six threads was much faster than the non-optimized fmul0.

In conclusion, restructuring the computation allowed the hardware to be utilized more efficiently for matrix multiplication.