Optimizing Matrix Multiplication Performance

Leslie Horace
Undergraduate of Computing Sciences
Coastal Carolina University
Conway, SC, USA
lahorace@coastal.edu

***Abstract*—This paper focuses on research for improving matrix multiplication memory performance. Matrix multiplication is used in a variety of scientific computing applications such as image processing, cryptography, and machine learning. In general, matrix multiplication is highly intensive task due to the amount of floating point operations and the depth of traversing matrices in memory. Thus, the elapsed time and number of floating point operations (per second) decrease, as the size of the computed matrices increase. Blocked matrix multiplication is hypothesised to improve these flaws, in which the experiments discussed will illuminate if this is true. Finally, providing more in depth research on optimal parameters for varying matrix sizes.**

***Index Terms*—algorithms, matrix multiplication, memory management**

I. Introduction

A. Matrix Multiplication

Matrix multiplication is the production of a matrix from multiplying two matrices. Let the resulting matrix be denoted as $C_{i,j}$ and multiplied matrices be denoted as $A_{i,k}$ and $B_{k,j}$. The values of $C_{i,j}$ are the dot products from the rows of $A_{i,k}$ and columns $B_{k,j}$. The general equation for each dot product is $C_{i,j} = A_{i,k} \times B_{k,j}$. For matrix multiplication to produce a valid matrix, the cardinally of the rows(k) in $A_{i,k}$ must equal the columns(k) in $B_{k,j}$.

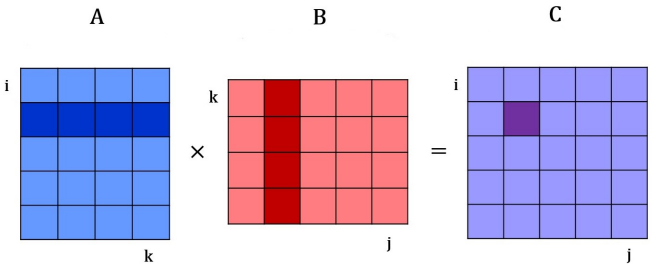

Fig. 1. Two matrices A and B multiplied into C. Colored blocks represent A's current row, B's column, and C's resulting coordinate.

Figure 1 demonstrates a visual representation of the inner computations performed within Algorithm 1. the values of each row in the first matrix is multiplied by the each column in the second matrix.

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + \cdots + A_{in}B_{nj} = \sum_{k=1}^{n} A_{ik}B_{kj} \quad (1)$$

To clarify the computational logic implemented in Algorithm 1, the mathematical definition of matrix multiplication is shown in Equation 1.

Algorithm 1: Naive Matrix Multiplication

```
Data: A[r][n], B[m][c]
Result: C[r][c]
if n = m then
    for i = 0; i < r; i++ do
        for j = 0; j < c; j++ do
            sum = 0;
            for k = 0; k < n; k++ do
                sum += A[i][k] * B[k][j];
                C[i][j] = sum;
            end
        end
    end
end
```

B. Blocked Matrix Multiplication

Blocked matrix multiplication uses the same logic as naive matrix multiplication, but the computations are in divided into partitioned blocks. In Algorithm 2, the outer function *DGEMM* (Double Precision Matrix Multiplication) partitions each block, then calls to the inner function *BlockMM* to perform matrix multiplication. The only condition is that both input matrices sizes must be divisible by the block size, hence the if statement in Algorithm 2.

The idea is to split the computational workload into smaller matrices and perform multiplication a block at a time. The allows limiting the domain size and how far each loop much index to create the resulting matrix. Figure 2 shows a visual representation of how each block is partitioned.

Multiplying matrices by blocks is said to improve the elapsed time and gigaflops per second according to textbook Computer Organization and Design [2]. This is done by increasing temporal locality and reducing the amount of cache misses when accessing each column of the second matrix. This does not affect the first matrix as it accessed in row major form, which is how it is stored in memory. On the other hand the second matrix must be indexed by columns, thus the time

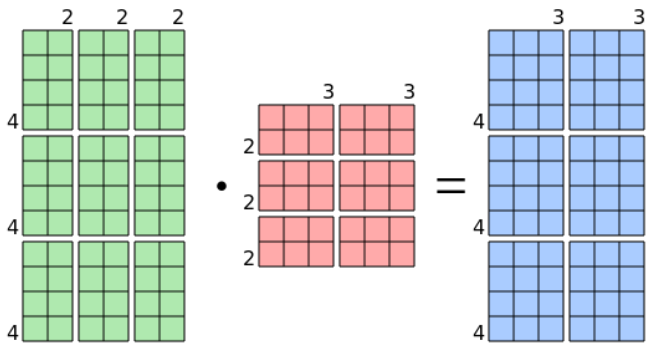

Fig. 2. Two matrices multiplied by blocked into new matrix. Colored blocks are separated to represent partitioning in the matrices [1].

Algorithm 2: Blocked Matrix Multiplication

Data: $block, A[r][n], B[n][c]$
Result: $C[r][c]$

if $n \bmod block = 0$ **then**
  `DGEMM` $(n, block, A, B, C)$
  for $y = 0;\ y < n;\ y + block$ **do**
    for $x = 0;\ x < n;\ x + block$ **do**
      for $z = 0;\ z < n;\ z + block$ **do**
        `BlockMM` $(block, x, y, z, A, B, C)$
      end
    end
  end
end

`BlockMM` $(block, x, y, z, A, B, C)$
for $i = 0;\ i < (x + block);\ ++i$ **do**
  for $j = 0;\ j < (z + block);\ ++j$ **do**
    $sum = 0;$
    for $k = 0;\ k < (y + block);\ k++$ **do**
      $sum\ += A[i][k] * B[k][j];$
      $C[i][j]$ = $sum;$
    end
  end
end

to access each column index increases the gap from the starting index increases.

II. Implementation

The programs used to gather research were written in C language. The 2d arrays form the input matrices were created with program 'make-2d', which takes the row and column sizes as parameters and writes a 2D array of incriminating values into a binary file. For testing purposes a program called 'print-2d' displays the contents of the created binary file. Both 'mm' and 'blocked-mm' performed any error checking in the main function along with the computations for the gathered data.

The naive algorithm was utilized in program 'mm' to perform standard matrix multiplication. This made use of the stdio.h library using *fread* to read the two input matrices into 2D arrays and *fwrite* to write out the resulting matrix shown in Figure 3. The 2d arrays were malloced using double pointers, one to point to each row and one to index the rows.

The blocked matrix multiplication was implemented in a program called 'blocked-mm'. This required a extra argument for block size as shown in Figure 4. The *DGEMM* function scopes out each block and passes pointers to the starting indices to function *BlockMM* to perform matrix multiplication. This program had the same reading and writing methods as 'mm', but the matrices were stored as 1d arrays instead. This way the arrays are in row major form and the columns are indexed mathematically with $M[row * size + col]$. To ensure divisibility by block size, the program only accepts square matrices.

Computing the elapsed times utilized a 'timer' program to calculate the current time. The overall times are captured at the beginning and end of the program. Where as the matrix multiplication times were captured around the call to the function in main. The floating point operations were approximated as $2 \times n^3$, since each loop involves two operations to compute sums. Finally the gigaflops per second are calculated as $GF = [(2n^3/ElapsedTime)(1 \times 10^{-9})]$.

To validate the accuracy of the resulting matrices, program "mydiff" takes two matrices as parameters and compares the differences in the products. One being the output matrix from "mm" and the other matrix of the same size from "blocked-mm". This returned the Total Sum of Squared Error shown in Equation 2 and the Average Percent of Relative Error shown in Equation 3.

$$TSSE(A, B) = \sum_{i=0}^{r-1} \sum_{j=0}^{c-1} (A_{ij} - B_{ij})^2 \tag{2}$$

$$AVGPRE(A, B) = \frac{1}{(r \times c)} \sum_{i=0}^{r-1} \sum_{j=0}^{c-1} \left| \frac{(A_{ij} - B_{ij})}{A_{ij}} \right| \tag{3}$$

III. Experimentation

A. Experiments and Parameters

The process of experimentation included running varying tests to gather data for both matrix multiplication and blocked matrix multiplication. The native matrix multiplication test used squared matrix sizes 256×256, 512×512, 1024×1024, 2048×2048, and 4096×4096. the blocked matrix multiplication used the same matrix sizes in order to compare results. As well testing different block sizes to find the most efficient sized block per matrix size. The block sizes tested for each were 16, 32, 64, and 128. Table I displays the parameters used to help visualize the experiments. In total, there was 5 sets of data gathered for the native method and 20 sets of data for the blocked method.

The data gathered included the elapsed times and gigaflops per second for the matrix multiplication computation and the overall program. After obtaining all of the output matrix data

```c
void MatrixMultiply(double ** A,  double ** B , double ** C, int mA, int nAB, int mB) {
    for (int x = 0;  x < mA; x ++) {
        for (int z = 0;  z < mB; z++) {
            double sum = 0.0;
            for (int y = 0;  y < nAB; y++) {
                sum += A[x][y] * B[y][z];
                C[x][z] = sum;
            }
        }
    }
}
```

Fig. 3. Naive matrix multiplication function written in C language using Algorithm 1.

```c
void BlockMM(int len, int block, int start_x, int start_z, int start_y, double *A, double*B, double *C) {
    for (int x = start_x; x < (start_x + block); ++x) {
        for (int z = start_z; z < (start_z + block); ++z) {
            double sum = C[x*len+z];
            for( int y = start_y; y < (start_y + block); y++)   {
                sum += A[x*len+y] * B[y*len+z];
                C[x*len+z] = sum;
            }
        }
    }
}

void DGEMM(int len, int block, double *A, double * B , double * C) {
    for ( int z = 0; z < len; z += block )
        for ( int x = 0; x < len; x += block)
            for ( int y = 0; y < len; y += block )
                BlockMM(len, block, x, z, y, A, B, C);
}
```

Fig. 4. Blocked matrix multiplication caller/callee functions written in C language using Algorithm 2.

files, the resulting matrices for native and blocked were compared using the "mydiff" tool mentioned in implementation. This allowed for finding any errors in the resulting dot product values to ensure the resulting data is valid for comparison.

TABLE I
C COMPILER PARAMETERS

Native Matrix Multiplication				
256^2	512^2	1024^2	2048^2	4096^2
Blocked Matrix Multiplication				
Matrix Size	**Block Size**			
256^2	16	32	64	128
512^2	16	32	64	128
1024^2	16	32	64	128
2048^2	16	32	64	128
4096^2	16	32	64	128

B. Hardware and Environment

The machines running operating system is Windows 10. The c code was developed using the open source MSYS2 gcc compiler tool. MSYS2 provides a native build environment for Windows, giving the ability to use tools designed for bash terminals [3]. This allowed for use of a MAKE file with gcc flags for warning(-Wall), debugging (-g), and output files (-o). The following list contains the relevant hardware specifications of the machine used for experimentation.

- Processor: AMD Ryzen 9 3900X (12-Core)
- L1 Data Cache: 12×32 KB (8-Way)
- L1 Inst. Cache: 12×32 KB (8-Way)
- L2 Cache: 12×512 KB (8-Way)
- L3 Cache: 4×16 KB (16-way)
- Memory: 32GB DDR4 (3200MHz)
- Storage: $2 \times$ 1TB SSD's (560 MB/Sec)

It is notable that the computed times and gigaflops per second will vary on other machines. Due the having a newer processor using a fair sized set-associative cache, the performance is fairly efficient for a workstation. A standard laptop may have much longer times and therefore a decrease in gigaflops. Where as high-end server may may increase the performance significantly.

IV. Results and Conclusions

The resulting elapsed times obtained for blocked and naive matrix multiplication are shown in Figure 5 as the overall and Figure 6 for matrix multiplication. Comparing the two graphs the times were approximately the same. This is due to the reading and writing tasks being fairly efficient for the overall, where the matrix multiplication being $O(n^3)$ performs the most intensive tasks. Looking at the values obtained from the *unblocked* matrix multiplication, the times increased significantly as the matrix size increased. On the other hand, the times for blocked matrix multiplication increased with matrix size, but only a small amount. In particular there is a drastic decrease in the amount of time elapsed for the largest matrix size where unblocked was nearly 1000 seconds and blocked were roughly between 290 and 350 seconds.

The speed up of elapsed time for the overall is shown Figure 7 and mm function is shown in Figure 8. The speed up in general ranged from approximately $1.1\times's$ to $3.4\times's$ faster for increasing matrix sizes. Given this observation, blocking may be a scalable solution for applications that use matrix multiplication.

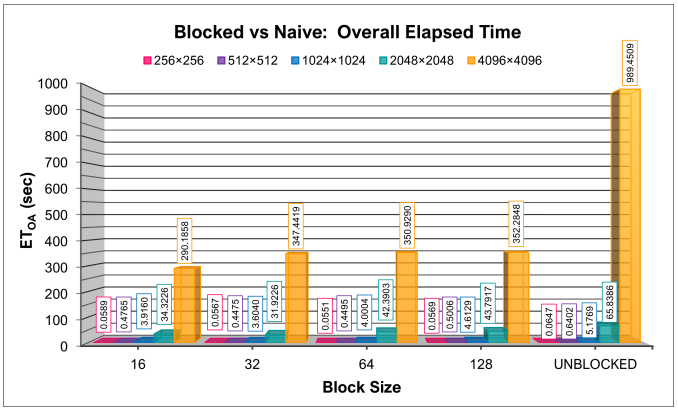

Fig. 5. Bar graph for overall programs blocked versus unblocked elapsed time.

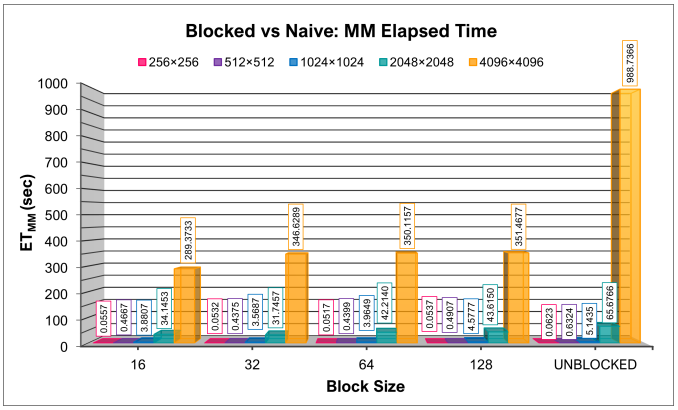

Fig. 6. Bar graph for mm functions blocked versus unblocked elapsed time.

The graphs shown in Figure 9 and Figure 10 represent the gigaflops per second for blocked vs unblocked multiplication.

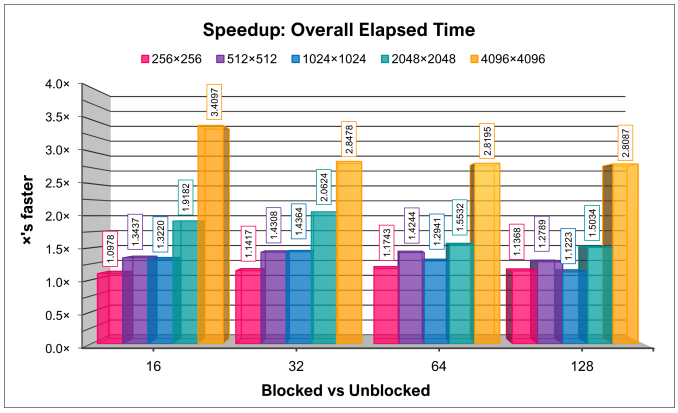

Fig. 7. Bar graph for speed up of overall programs blocked versus unblocked elapsed time.

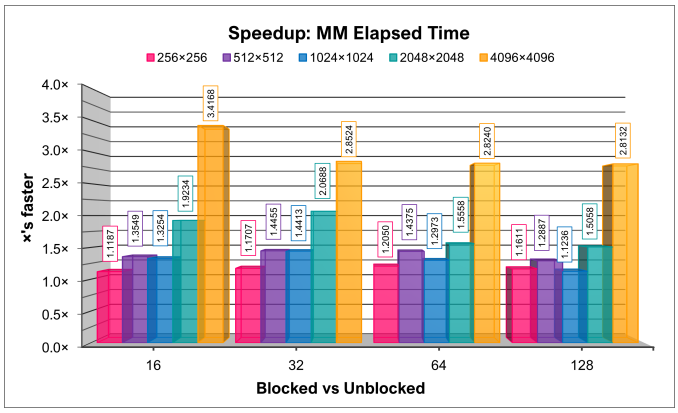

Fig. 8. Bar graph for speed up of mm functions blocked versus unblocked elapsed time.

The values in both graphs are roughly the same, due to having similar elapsed times. Recall the claim stated in Section I, that using blocked matrix multiplication increases the amount of gigaflops per second [2]. Looking at the results, we can now safely see that this claim is true. The trends in the gigaflops for unblocked decrease about 0.4 gf/sec from small to large matrix sizes. Yet the gigaflops from using blocks are higher in general and only decrease about 0.1 to 0.2 gf/sec. In higher performance systems, the scale for this may be much higher than this, given that these results were obtained on a workstation.

The speed up for the gigaflops per second are shown in Figure 11 and in Figure 12. It may be noticeable that these values are the same as the speed up for elapsed time. In reality they are approximately the same and only differing after 4 or 5 decimal places. To summarize the results of the experiment, Table II highlights the best cases and Table III displays the worst cases. Each table displays the best/worst block size per matrix size and the correlated speedup for each. Since the speed up of elapsed times and gigaflops were approximately the same for both the overall program and mm function, the values in the tables represent the speed up for both.

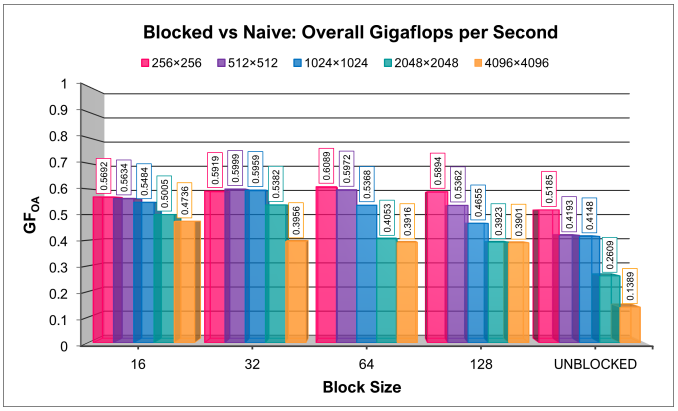

Fig. 9. Bar graph showing the overall programs blocked versus unblocked gflops/sec.

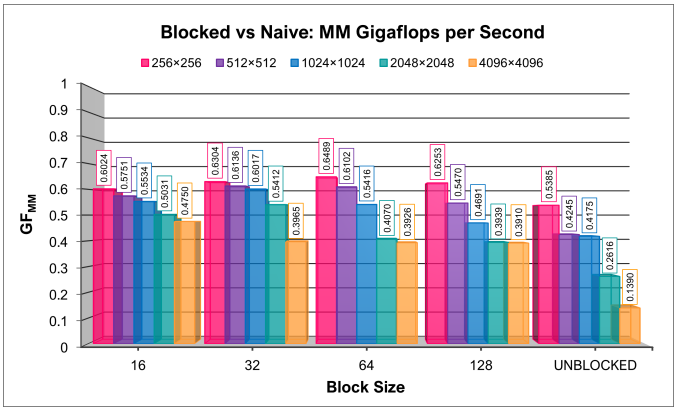

Fig. 10. Bar graph showing the mm functions blocked versus unblocked gflops/sec.

TABLE II
BEST BLOCK SIZES AND MAXIMUM SPEEDUP

Matrix Size	Block Size	Speedup(OA)	Speedup(MM)
256 × 256	64	1.1743	1.2050
512 × 512	32	1.4308	1.4455
2048 × 2048	32	1.4364	1.4413
1024 × 1024	32	2.0624	2.0688
4096 × 4096	16	3.4097	3.4168

[a]Speed up values approximated at 4 decimal places.

TABLE III
WORST BLOCK SIZES AND MINIMUM SPEEDUP

Matrix Size	Block Size	Speedup(OA)	Speedup(MM)
256 × 256	16	1.0978	1.1187
512 × 512	128	1.2789	1.2887
2048 × 2048	128	1.1223	1.1236
1024 × 1024	128	1.5034	1.5058
4096 × 4096	128	2.8087	2.8132

[a]Speed up values approximated at 4 decimal places.

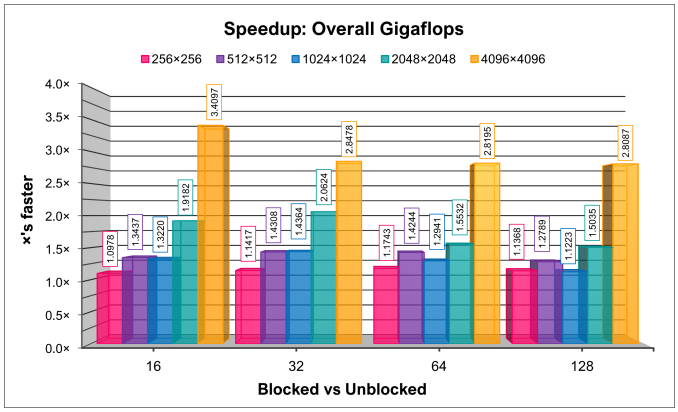

Fig. 11. Bar graph showing the speed up of overall programs blocked versus unblocked gflops/sec.

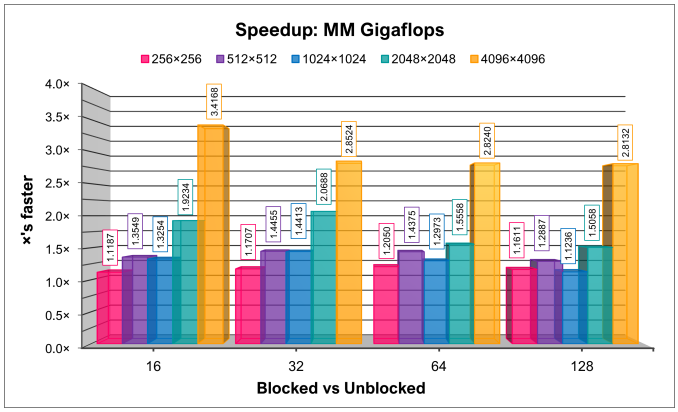

Fig. 12. Bar graph showing the speed up of mm functions blocked versus unblocked gflops/sec.

V. FUTURE WORK

This experiment gave a lot of insight as well as proposing many questions after observing the results. It would be interesting to learn more about the correlation between block size and cache size to find the most optimal parameters for matrix multiplication. This often done in real world applications to give consistent performance on various different hosts.

Aside from this, it would also aid in an experiment to determine if blocked matrix multiplication is actually a scalable solution. This would involve running experiments on many different machines with larger matrices. Then the results would determine how much the speed up increases on average for increasing matrix sizes. Or possibly reveal other behavior such as a limit on how much the speed can increase.

REFERENCES

[1] Figure showing blocked matrix multiplication. [Online]. Available: https://tinyurl.com/5t5d2y5a
[2] D. A. Patterson and J. L. Hennessy. (2021) Computer organization and design - interactive version (mips). [Online]. Available: https://learn.zybooks.com/zybook/CODMIPSR56
[3] Msys2 software distribution and building platform for window. [Online]. Available: https://www.msys2.org/