Matrix Multiplication Optimization

7th July 2024

Rotskos Emmanouil, Undergraduate Student

Electrical and Computer Engineering | Aristotle University of Thessaloniki

Contents

Summary

This paper researches the effect of different optimization methods on the speed of matrix multiplication. This study is based on the results from the multiplication of two 2048x2048 square matrices. All the code supplied is written in the C programming language. The methods tested are as such:

- Loop Reordering
- Blocking
- Loop Unrolling

For each test the full code used as well as all my results will be available and highlighted in this paper. The full 3 step process will also be thoroughly explained, with accompanying graphs to further explain all the results.

Key Findings

All three methods produce great improvements in runtime especially when combined. In greater detail:

1. Loop Reordering produced a 7.1815 times speedup compared to the baseline measurements
2. Blocking had an 8.9255 times speedup with a block size of 512 and a 4.1 times speedup with a block size of 4.
3. Loop Unrolling had a 7.7232 times speedup when unrolling 32 lines of code.

All the code for the above results will be available publicly on my GitHub, linked below. Beware that if these tests are run on a different system the results might differ greatly as CPU clock speed and the size of CPU cache is very important.

GitHub: *https://github.com/mrotskos/Matrix_Multiplication_Optimization*

Introduction

Matrix Multiplication is a vital part of many modern algorithms and application and with the surge in popularity of AI in the last couple years it is now more important than ever to look into its applications and find ways to better optimize this basic but important function.

While the speed of matrix multiplications depends on many variables one of the most significant ones is thought to be memory hierarchy. In greater detail, when implementing matrix calculations, we tend to work with large data sets which have to be constantly moved in and out of cache memory so that the processor has access to it. If the data is not available to the processor, then we have a cache miss, this is an "expensive" error as moving data from higher levels of cache or even memory can be a very slow process. As such it is of vital importance to find ways to improve the locality of our data.

One such way could be creating bigger and faster caches for our processor. However, this can quickly prove expensive and inefficient. Another way we could go about could be optimizing our code so we access parts of our data that are not in cache more rarely and in a way that allows data sets that move into the cache to be utilized more effectively.

This second method of optimization is the one we will be researching in this paper, by implementing the 3 optimizations underlined in the summery.

Our objective is to find a better algorithm for matrix multiplication only through software changes that will allow for many applications using these algorithms.

System Information

For the results to have some meaning I must also include the information of the system on which all test took place.

• Processor:	Intel Core i9-10850k
• Clock Frequency:	3.60 GHz
• L1 Cache Size:	640 KB
• L2 Cache Size:	2.5 MB
• L3 Cache Size:	20.0 MB
• GCC Version:	13.2.0
• Operating System Version:	Windows 11 23H2

All test were run with -O3 optimization setting through the gcc gnu compiler.

Step 0: Baseline Measurements

The most basic form of the matrix multiplication function is:

```
void matrixMult(float * const C,                     // output matrix
                float const * const A,               // first matrix
                float const * const B,               // second matrix
                int const n) {                       // number of rows/cols
       for(int i = 0; i < n; i++) {
              for(int j = 0; j < n; j++) {
                     c[sub2ind(i, j, n)] = 0;        // initialize output value

                     for(int k = 0; k < n; k++)      // accumulate product
                            C[sub2ind(i, j, n)] += A[sub2ind(i, k, n)] + B[sub2ind(k, j, n)];

}
```

Figure 1: Basic implementation of matrix multiplication in C

The time needed for the multiplication was calculated using the following code:

```
#include <sys/time.h>
#define MAX_ITER 10

double time = 0.0;

for(int it = 0; it < MAX_ITER; it++) {
       gettimeofday(&start, NULL);
       matrixMult(C, A, B, n);
       gettimeofday(&end, NULL);

       time = ((end.tv_sec - start.tv_sec) * 1000.0 +
                (end.tv_usec - start.tv_usec) / 1000.0);

       printf("Iter: %d Time: %f\n", it, time);
}
```

Figure 2: A simple way to calculate elapsed time in C

The above code produced the following results when run for a 2048x2048 matrix multiplication, with matrices initialized with random values. (Results are in ms)

```
Iter: 0 Time: 10150.330000                    Iter: 5 Time: 10084.883000
Iter: 1 Time: 10001.624000                    Iter: 6 Time: 10066.210000
Iter: 2 Time: 10236.830000                    Iter: 7 Time: 10191.754000
Iter: 3 Time: 10170.384000                    Iter: 8 Time: 10112.872000
Iter: 4 Time: 10088.352000                    Iter: 9 Time: 10098.216000
```

Figure 3: Baseline results of matrix multiplication

From the above results we can conclude that this basic version of matrix multiplication takes on average about 10 seconds to produce the result. The variations in the times are very small and standard between multiple runs of the same multiplication, as expected.

We, also, observe that the amount of time needed for just one multiplication is quite large and as such we have proven the need for better optimized algorithms.

Step 1: Loop Reordering

The first thing we can try to increase spatial locality would be to change the order of the 3 for loops in the function. Of course, this would also mean changing which variables are used to initialize the output matrix C to zero. The following is the resulting code for loop order <jik> and <ikj>:

```
void matrixMult(float * const C,                    // output matrix
                float const * const A,              // first matrix
                float const * const B,              // second matrix
                int const n) {                      // number of rows/cols
        for(int j = 0; j < n; j++) {
                for(int i = 0; i < n; i++) {
                        c[sub2ind(j, i, n)] = 0;      // initialize output value

                        for(int k = 0; k < n; k++)    // accumulate product
                                C[sub2ind(i, j, n)] += A[sub2ind(i, k, n)] + B[sub2ind(k, j, n)];

}
```

```
void matrixMult(float * const C,                    // output matrix
                float const * const A,              // first matrix
                float const * const B,              // second matrix
                int const n) {                      // number of rows/cols
        for(int i = 0; i < n; i++) {
                for(int k = 0; k < n; k++) {
                        c[sub2ind(i, k, n)] = 0;      // initialize output value

                        for(int j = 0; j < n; j++)    // accumulate product
                                C[sub2ind(i, j, n)] += A[sub2ind(i, k, n)] + B[sub2ind(k, j, n)];

}
```

Figure 4: Loop Reordering implementation of matrix multiplication in C (Loop order <jik> and <ikj>)

In the above code we can see that the line of code calculating the accumulate product remains unchanged as we do not want to change the logic of the algorithm, rather we just want to change the order in which the matrix spaces are accessed. Also, as mentioned above, in order to keep the matrix initialization inside the loops we must change which variables are used to initialize it. This is all done in an effort to minimize other variables which might interfere with our results and thus provide a cleaner answer for the result of each method. Later in this study I highlight the effect of moving the matrix initialization to a different set of for loops, as it proved to be an interesting result from my research (see Interesting Results).

Here we have the output of our program when executed with different loop orders. The results are given in the form of the median time from 10 iterations.

```
Loop Order:                         Median Time over 10 Iterations:

  <ijk>                                         10084.883 ms
  <jik>                                         40020.543 ms
  <kij>                                          1724.920 ms
  <kji>                                         89877.210 ms
  <ikj>                                          1404.274 ms
  <jki>                                         90793.882 ms
```

Figure 5: Loop Reordering Results

From the above results we can conclude that loop reordering can have a very significant effect on the algorithms runtime. However, this was an expected outcome as some loop orders will lead to many cache misses as the program is trying to access memory locations which are very far apart. We, also, expect this optimization method to have an even greater effect and even more drastic differences between different loop orders on systems will less, or slower cache memory.

We achieve the fastest result for loop order <ikj> with a median time of 1404.274 ms granting us a time around 7 times faster than our baseline measurement with loop order <ijk>. In greater detail we have:

$$Speedup = \frac{Baseline\ Measurement}{Measurement\ with\ loop\ order\ < ikj >} = \frac{10084.883\ ms}{1404.274\ ms} = 7.1815$$

Concluding, with loop reordering we were able to achieve a speedup of 7.1816 times, while keeping the C matrix initialization same as in the baseline measurement.

Step 2: Blocking

Blocking is another very well known and effective way of increasing spatial locality as we break the matrix multiplication into multiple multiplications of smaller matrices. To implement this in code we break each for loop into two. One which loops through the blocks and a second one which loops though each index of the current block. In this optimization method it is apparent that block size would have an inherit effect on the final time, as very small block sizes will have as a result a very large amount of loop iterations and loop checks, while a large block size (one that tends towards the size of the whole matrix or larger) will remove any spatial locality gains and maybe even result in worse results than before. As with this method too many loop orders are possible, we will be working with the order <ikj> from step 1, which proved the best. Taking this into consideration the code for this implementation is the following:

```c
void matrixMult(float * const C,                    // output matrix
                float const * const A,              // first matrix
                float const * const B,              // second matrix
                int const n) {                      // number of rows/cols

        const int s = 2;                            // block size

        for(int i_out = 0; i_out < n; i_out += s) {
                for(int k_out = 0; k_out < n; k_out += s) {
                        for(int j_out = 0; j_out < n; j_out += s) {
                                for(int i_in = 0; i_in < s; i_in++) {
                                        for(int k_in = 0; k_in < s; k_in++) {

                                                C[sub2ind(i_out + i_in, k_out + k_in, n)] = 0;

                                                for(int j_in = 0; j_in < s; j_in++) {

                                                C[sub2ind(i_out + i_in, j_out + j_in, n)] +=
                                                        A[sub2ind(i_out + i_in, k_out + k_in, n)] *
                                                        B[sub2ind(k_out + k_in, j_out + j_in, n)];
                                                }
                                        }
                                }
                        }
                }
        }
}
```

Figure 6: Blocking implementation of matrix multiplication in C (Loop order <ikj>)

Changes likes these could introduce bugs and mistakes be made that would alter the output of the multiplication, to avoid this I chose to print the index 4 of the output matrix to make sure that the output stays unchanged from the baseline measurement.

Blocking can also be implemented as follows:

```
void matrixMult(float * const C,                        // output matrix
                float const * const A,                  // first matrix
                float const * const B,                  // second matrix
                int const n) {                          // number of rows/cols

        const int s = 2;                                // block size

        for (int i0 = 0; i0 < n; i0 += s) {
                for (int k0 = 0; k0 < n; k0 += s) {
                        for (int j0 = 0; j0 < n; j0 += s) {
                                for (int i = i0; i < i0 + s; i++) {
                                        for (int k = k0; k < k0 + s; k++) {

                                                C[sub2ind(i, k, n)] = 0;

                                                for (int j = j0; j < j0 + s; j++) {

                                                        C[sub2ind(i, j, n)] +=
                                                                A[sub2ind(i, k, n)] *
                                                                B[sub2ind(k, j, n)];
                                                }
                                        }
                                }
                        }
                }
        }
}
```
Figure 7: Another Blocking implementation of matrix multiplication in C (Loop order <ikj>)

However, as the two did not produce a significant difference in runtime I chose to continue working with the first one.

This optimization method proved a lot more challenging as with some loop orders the multiplication product would be false. I believe that this error is produced by the fact that the initialization of the output matrix is implemented inside the other for loops and may be affecting the results. Moving this to its own for loops not only stops this error but also improves the runtime, as I highlight later in this paper.

Executing the above code for different block sizes we have the following results:

Block Size:	Median Time over 10 Iterations:
2	3494.509 ms
4	2460.949 ms
8	3593.940 ms
16	1826.303 ms
32	1397.515 ms
64	1470.334 ms
128	1485.397 ms
256	1312.180 ms
512	1129.891 ms
1024	1154.692 ms
2048	1460.419 ms

Figure 8: Blocking Results

As we can see our predictions about the effect of block size were correct with the runtime of implementations using very small block sizes being worse than our new baseline from the first step of loop reordering and larger ones coming closer to the original size. We got the best result for a block size of 512 and a runtime of 1129.891 ms. While when staying with small block sizes (only 1 digit) we get the best result for the size of 4 with a time of 2460.949 ms which is however worse than our baseline.

So, with this optimization method we can get an even faster result with a speedup of:

$$Speedup = \frac{Measurement\ with\ loop\ order\ \ < ikj >}{Measurement\ with\ blocking\ and\ loop\ order\ \ < ikj >} = \frac{1404.274\ ms}{1129.891\ ms} = 1.2428$$

And a speedup in terms of the baseline measurements of:

$$Speedup = \frac{Baseline\ Measurement}{Measurement\ with\ blocking\ and\ loop\ order\ \ < ikj >} = \frac{10084.883\ ms}{1129.891\ ms} = 8.9255$$

From this we conclude that blocking too can have a significant positive effect to our runtime.

Step 3: Loop Unrolling

One last method we can try to make our matrix multiplication even faster is loop unrolling. In this method we remove the for loops and instead write the code out by hand. This can be a tedious process especially for large for loops like the ones we have here. However, by removing the for loops, we also remove the need to check weather our loop should break, instead executing the program line by line. This optimization method is many times implemented automatically by the compiler (this is true in our case using -O3 optimization). The gist of loop unrolling can be explained as such:

Loop Unrolling: Do n loop iterations' worth of work per actual loop iteration, so we save ourselves from doing the loop overhead (test and jump) every time, and instead incur overhead only every n-th time.

In our example we can make the following changes to our code, where we unroll the inner most for loop so that we do 16 loop iterations' worth of work per actual loop:

```
void matrixMult(float * const C,                    // output matrix
                float const * const A,              // first matrix
                float const * const B,              // second matrix
                int const n) {                      // number of rows/cols

        for(int i = 0; i < n; i++) {
                for(int k = 0; k < n; k++) {
                        C[sub2ind(i, k, n)] = 0;
                                for(int j = 0; j < n - 16; j += 16) {

                C[sub2ind(i, j, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j, n)];
                C[sub2ind(i, j + 1, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 1, n)];
                C[sub2ind(i, j + 2, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 2, n)];
                C[sub2ind(i, j + 3, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 3, n)];
                C[sub2ind(i, j + 4, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 4, n)];
                C[sub2ind(i, j + 5, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 5, n)];
                C[sub2ind(i, j + 6, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 6, n)];
                C[sub2ind(i, j + 7, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 7, n)];
                C[sub2ind(i, j + 8, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 8, n)];
                C[sub2ind(i, j + 9, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 9, n)];
                C[sub2ind(i, j + 10, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 10, n)];
                C[sub2ind(i, j + 11, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 11, n)];
                C[sub2ind(i, j + 12, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 12, n)];
                C[sub2ind(i, j + 13, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 13, n)];
                C[sub2ind(i, j + 14, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 14, n)];
                C[sub2ind(i, j + 15, n)] += A[sub2ind(i, k, n)] * B[sub2ind(k, j + 15, n)];

                                }
                }
        }
}
```

Figure 9: Loop Unrolling Blocking implementation of matrix multiplication in C (Loop order <ikj>)

Executing the above code leads to a median runtime of 1350.670 ms which is an improvement not only from our baseline but even from the 1st step too. However, the speedup of course is linked to the number of loop iterations' worth of work we do per actual loop, as seen in the following table:

```
Number of loop iterations:             Median Time over 10 Iterations:

         4                                      1430.802 ms
        16                                      1350.670 ms
        32                                      1305.786 ms
```

Figure 8: Loop Unrolling Results

We can see that the effect of unrolling a larger number of lines is not as great as we would maybe want, while it increases the program size and it becomes harder both to read and to maintain. As such, this last method is, in my opinion, only to be used by hand in cases were extreme optimization, as it is also sometimes executed by compilers.

When unrolling 32 lines of code we have measured a speedup of:

$$Speedup = \frac{Baseline\ Measurement}{Measurement\ with\ loop\ unrolling\ and\ order < ikj >} = \frac{10084.883\ ms}{1305.786\ ms} = 7.7232$$

Concluding with this last method, it can be an easy way to make matrix multiplication faster achieving a speedup of 7.7232 times when combined with loop reordering.

Interesting Results

In this part of the paper, I would like to point out some interesting result that I came across in my research on the subject.

1. Output matrix initialization

To start with moving the initialization of the output matrix to a different set of for loops (still inside the function), time was greatly improved. This improvement caried out to all the other methods. Most likely this effect is due to the optimizations done by the compiler with the tag -03, as it is more easily able to optimize this routine on its own rather than inside the other three for loops. The code with this change would look something like this:

```
void matrixMult(float * const C,                    // output matrix
                float const * const A,              // first matrix
                float const * const B,              // second matrix
                int const n) {                      // number of rows/cols

        for(int i = 0; i < n * n; i++) C[i] = 0.0; // Initialize C to 0 (important!)

        for(int i = 0; i < n; i++) {
                for(int j = 0; j < n; j++) {
                        for(int k = 0; k < n; k++)    // accumulate product
                                C[sub2ind(i, j, n)] += A[sub2ind(i, k, n)] + B[sub2ind(k, j, n)];

}
```

Figure 9: Basic implementation of matrix multiplication in C with standalone matrix initialization

In this case with loop order <ijk> the median time remains at 10661.145 ms. However, when we reorder the loops to <ikj> we measure a median time of 961.772 ms, a great improvement over the previous time of 1404.274 ms.

This interesting result show that modern compilers are very powerful when it comes to optimization and clean code can not only help the developer during maintenance and development but also the compiler.

2. Compiler Optimization

I would also like to point out that, in my opinion if this test were run without the -03 optimization tag the difference in the results would have been greater, as all of the above methods are commonly used by compilers in order to better optimize your code. This effect is clear when we execute our baseline code without optimization, which results in a median time of 115631.871 ms.

Conclusions

While the subject of matrix multiplication optimization is not new by any chance it is still a very interesting one especially for a new student as it greatly helps one understand the importance of spatial locality and the inner working of memory and cache.

Summarizing, in this study we proved that all three methods of optimization can have a drastic effect on the program's runtime with simple solutions like loop reordering granting as much as a 7.1815 speedup and more complicated one such as blocking around 8.9255.

It is extremely important that as computer engineers we think about our software at the lowest level possible when talking about optimization as simple concepts like memory hierarchy, which are many times overlooked in higher level languages play a massive roll on the end result for our program. Advancements in the field of optimization do not only mean that faster systems can be created but also that more intensive work can be executed on smaller, cheaper and (something very important in our day and age) more efficient machinery.

Concluding, I would like to give my opinion on which of these methods I truly believe are useful to the vast majority of developers. To start with I believe that loop unrolling especially when taken to its extreme causes more harm in code readability and maintenance than help, thus it would not be my first choice. On the other hand, loop reordering and blocking can provide massive improvements to our code and should be used whenever possible. Also, they help us achieve a deeper understanding of the workings of our software, which is always good practice.

References

1. Computer Organization and Design | David A. Patterson, John L. Hennessy
   https://archive.org/details/computer-organization-and-design-fifth-edition-the-hardware-software-interface-by-hennessy/page/n35/mode/2up
2. Stanford Computer Science 107 Lectures
   https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/lectures/
3. Block-Matrix-Multiplication-OpenMP | dmitrydonchenko
   https://github.com/dmitrydonchenko/Block-Matrix-Multiplication-OpenMP
4. Using Blocking to Increase Temporal Locality | Randal E. Bryant, David R. O' Hallaron
   https://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf