# Matrix Multiplication Optimization

Xiaoxu Na and Shun-Chiao Chang

**Abstract** - Several attempts have been implemented to improve the matrix multiplication. The experiments are done on crill.uh.edu and macbook air 2012 mid version.
**Index Term** – Matrix Multiplication, Blocked Version of Multiplication, Column Order of Multiplication

## 1 Introduction

Matrixes play a very important role in mathematics and computer science, and they are even used in efficient manipulation and storage of digital data in different fields as computer is widely used in almost all areas. Matrix multiplication is essential since it may affect the run time and the usage of CPU when the matrix becomes too large.

## 2 Backgrounds

### 2.1 The Naïve Algorithm

The naïve algorithm is straightforward as the mathematical definition for the multiplication of two matrixes. In order to get the result of each item in matrix C, we should make a sum of the corresponding row of A by the corresponding column of B. Since each of the $n^2$ entries in the matrix A is multiplied by exactly n entries from the second matrix B, thus the total number of multiplication is $n^2 * n = n^3$, and the total number of additions is $(n-1) * n^2 = n^3 – n^2$. Thus, we classify the naïve algorithm as an $O(n^3)$ algorithm.

### 2.2 Column Order Multiplication

As the matrixes are all stored in the column order, each time we get the row i of A, which is A[i][], we will have a 1 miss if n is larger than the line size of the cache. Thus, we need to wait for A[i][] to fetch from the memory to the cache, which usually takes hundreds of cycles. However, if we may adjust the way matrix A is loaded into the memory, we can save some time.

First, the matrix A was loaded into the memory by column order A[][k] instead of the row order. We pulled out each item B[k][j], and multiplied with the corresponding A[][k], then we got each column of C[][j] for A[][k] * B[k][j] = C[][j] . After rolling over the entire column B[][j], we got the part of C corresponding to each iteration k. Finally, we summed the results up, we got the final results. By using this method, we loaded A, B and C all in the column order.

### 2.3 A Blocked Version of Matrix Multiply

Since A should be used in the row order, if we may divide the matrix A small enough to fit in the cache line, so that each sub matrix A can be loaded into the register with only one fetch, then we may save the time as well.

Assume the case size is much smaller than n, thus we cannot load all three n*n matrixes into the cache together. Assume that the cache block size is b, and we will divide the n*n matrixes into n/B blocks, with each block size B*B. When loading data for blocks, we get $B^2/b$ misses for each block, and totally for matrix A and B we get $2n/B * B^2/b = 2nB/b$ misses for each iteration. Sequentially, there are $n^2/B^2$ blocks, the total misses come to $2nB/b * n^2/B^2 = 2n^3/(b*B)$. On the other hand, the total misses of the naïve way come to

$(n/b + n)*n^2 = (1+b)\, n^3/b$. Since CPU time = hit time + miss rate * miss penalty and miss penalty is hundreds of cycles, reducing miss rate contributes a lot in order to reduce the CPU time. Thus, the bigger B is, the lower the miss rate will be as long as $3B^2 < b$.

### 2.4 Register Blocked Kernel Version

Similar as discussed above, if we make B small enough and $3B^2$ can be loaded into the register at the same time, we utilize the most registers to hold the data in one iteration, and then we can have a fast accumulation without fetching from the lower caches.

### 2.5 Prefetching Version

As we know, the most time the multiplication spends is not the accumulation itself, but fetching the data from lower memory hierarchy. Thus, if we may predict where the cache miss is and prefetch it ahead of time, we can help the register holding all the data the time it is needed for the multiplication.

## 3 Experiment Results

In the experiments, we compared different methods running both on crill and my mac. On the crill, the AMD Opteron 6174 processor has a peak performance 2.2GHz*2 elements/vector *2 operations per FMA = 8.8GFLOP/s. [1] My mac is 2012 mid version, with a 2.0GHz Intel core i7 processor; a 3MB shared level 3 cache and a 4 GB 1600 MHz DDR3 memory. GCC 4.2.1 is used.

### 3.1 Crill Results

It is clearly shown in Fig. 1 that the column order multiplication has a speedup of 2 comparing with the naïve algorithm, which is just what we have explained in advance. However, the block versions didn't work as expected. It did decrease the runtime a bit, but not that much as assumed.

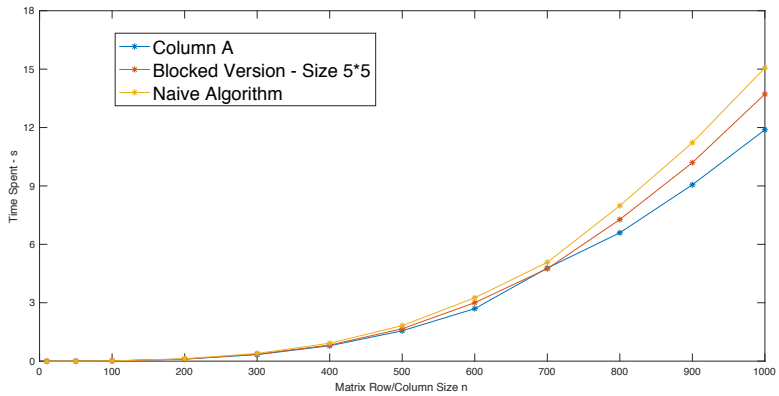


Fig.1 Crill Results

Many issues may cause this. First, there are four 2.2GHz 12-core AMD Opteron processors, totally 48 cores. When we require one node to run the program, it may be several cores running it at the same time. There are communications between the cores in order to get the final results, which may slow down the entire runtime. Second, the main

memory is 64GB, which is supposed to be shared memory. However, if so, different core need to compete for the shared memory for both read and write.

### 3.2 Mac Results

It run smoothly on my mac, from Fig 2 we can obvious tell both the column order version and the blocked version worked very well, giving a speedup of almost 2. And among all block sizes from 2*2, 5*5 to 10*10, the Fig 3 dedicated that block size 5*5 worked the best.

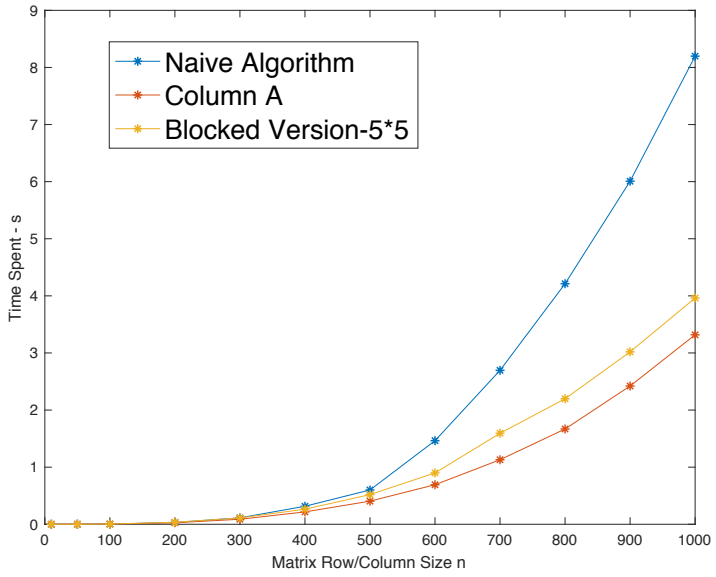


Fig.2 Mac Results

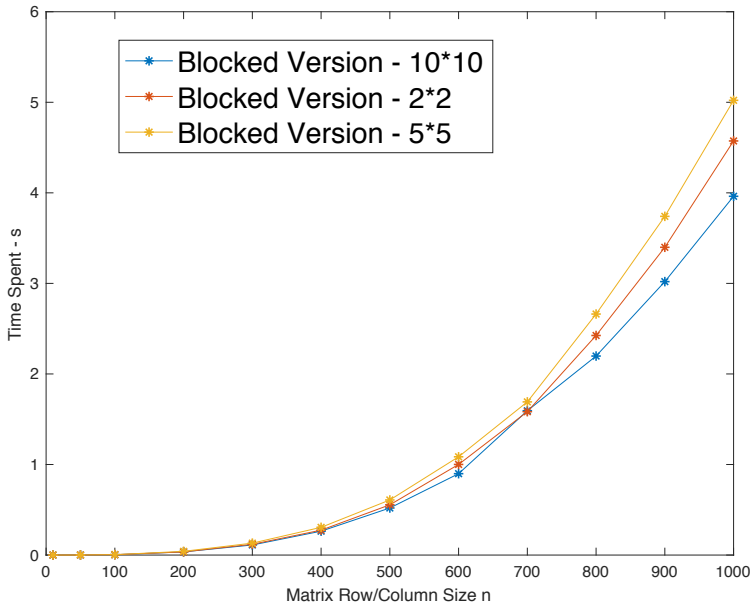


Fig.3 Mac Results – Different Block Sizes

For the block algorithm, we did two different versions, however, we didn't find the column version worked better. Maybe the block is already small enough for the cache to hold, thus, within the block whether the column way or the original way do not matter that much.

We did try prefetching method as well. However, it didn't come out a good result. Since timing plays an important role in prefetching, it needs more time to figure out where is the best point and when is the best time to prefetch the data. If prefetching starts too late, it is useless; on the other hand, if it begins too early, it may occupy a lot of space in cache, and when the data is needed, it may be some changes caused the cache line with dirt validated. Thus, more time should be engaged in the experiment.

### 3.3 Code Instructions

The column and row size of the matrixes are the same due to they are all square n*n matrixes. It is defined as static int Size = n in c code, for different n, the matrix is therefore occupy n*n = m size of double. The block size is defined as static int Block Size = B*B, where B is the column and the row size of the block. By compiling the file, just use: gcc – o mhb1000-5 MatrixHeapBlock.c, where nhb represents matrix heap block with size of 1000*1000 and the block size of 5*5.

## 4 Further Work

Nowadays, memory is much more complicated and optimized. The complier is more optimized; even some compliers can implement prefetching and optimization by itself. An optimization enabled by -fprefetch-loop-arrays that prefetches arrays used in loops. [1] Since prefetching time is really significant, by knowing more about the complier, the cache and the memory, we may come to a better solution.

**Acknowledges**

The author would like to express the gratitude to Dr. Panruo Wu.

[1] studio7designs. "The Crill Cluster", *http://pstl.cs.uh.edu/resources/crill*.

[2] Free Software Foundation, LLC. "Data Prefetch Support", *https://gcc.gnu.org/projects/prefetch.html*, 01/30/2018.