Matrix Multiplication Optimization Study
Assignment #2, CSC 746, Fall 2024

Daniil Matveev* SFSU
E. Wes Bethel† SFSU

Abstract

In this paper we benchmarked runtime and MFLOPs of various matrix multiplication algorithms, focusing on several speedup approaches: improving spatial locality, utilizing cache usage, and decreasing computational complexity through advanced algorithms. For each of these approaches, we considered several variations of the following algorithms: reordered three-loop, block matrix multiplication, and Strassen-Winograd algorithm. As result, in all three categories, we achieved several hundred times speedup compared to the naive algorithm.

1 Introduction

Matrix multiplication (MM) is a core operations in all kinds of computational problems from weather forecasting to computer graphics. With the development of hardware, the demand for large MM also increased. Thus computer scientists developed various approaches to improve space and time performance of MM including: increasing spatial locality, decreasing memory latency, and/or using algorithms with smaller theoretical complexity.

Since accurate tests are invaluable for deciding between implementations we benchmarked more than 10 implementations of MM. For each implementation, we 1) calculated theoretically the number of floating point operations and 2) measured the wall clock runtime of the algorithm. We then derived MFLOPs metric and interpreted the results by analyzing the memory access patterns of various implementations.

As the main result of the research, we speeded up the naive algorithm several hundred of times and benchmarked difference between various speeding-up approaches. However, even our best implementation was several times slower than the CBLAS implementation, which shows that we did not consider all potential improvements and there are several directions for future work.

2 Implementation

There are many interesting algorithms doing matrix multiplication. In the three following subsections, we focus on three distinct approaches. In the first subsection, we discuss the most obvious way to implement matrix multiplication - in a triple nested loop. The second subsection focuses on block-matrix decomposition and cache optimizations based on its consequences. In the final subsection, we discuss an advanced matrix multiplication algorithm that has asymptotic complexity better than $O(n^3)$.

2.1 Three-nested-loops algorithms

For matrices $A = (a_{ik})_{i,k=1}^{n}$ and $B = (b_{kj})_{j,k=1}^{n}$ their product is the matrix $AB = (\sum_{k=1}^{n} a_{ik}b_{kj})_{i,j=1}^{n}$. In this subsection, we discuss various $C++$ implementations that calculate the result in three loops.

*email:dmatveev@sfsu.edu
†email:ewbethel@sfsu.edu

We need one loop to calculate $\sum_{k=1}^{n} a_{ik}b_{kj}$ for a fixed pair (i, j) and two more loops to iterate over all i, j. Ordering these loops in lexicographical order of iterators: "i" < "j" < "k" we get the following algorithm:

```
// A, B - represent input (nxn)-matrices
// C - represent the product matrix: C=AxB
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++)
      C[i * n + j] += A[i * n + k] * B[k * n + j];
```

Listing 1: C++ implementation of matrix multiplication with loops ordered lexicographically for input and output matrices stored as double arrays in row order.

In total, this implementation has n^3 multiplication and n^3 additions or $2n^3$ floating point operations. Additionally, the implementation has $3n^3$ memory access operations. With maximal optimization, we should expect the algorithm to use only $2n^3 + n^2$ memory accesses, since $C[in + j]$ does not change in the inner loop.

You might notice that it doesn't matter in which order we perform the $C[in + j] += A[in + k]B[kn + j]$ operation. Because of the commutativity of the addition as long as we perform this operation for all tuples (i, j, k), we will get the correct output results. Furthermore from the implementation, we see that although b_{kj} and $b_{k+1\,j}$ are the closest elements in the index space in actual memory we store $B[kn + j]$ and $B[(k + 1)n + j] = B[(kn + j) + n]$ on the distance n from each other. This means that the algorithm 1 does not have spatial locality property since during every two consecutive iterations the algorithm accesses elements of B at the long distance n from each other.

Motivated by these arguments we also tested implementations with all other possible loop orderings. In listing 2 "k" < "i" < "j":

```
for (int k = 0; k < n; i++)
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      C[i * n + j] += A[i * n + k] * B[k * n + j];
```

Listing 2: C++ implementation of matrix multiplication with loops ordered as "k" < "i" < "j". There are 6 permutations of a 3-element set and therefore 6 possible loop orderings.

Each such implementation will use $2n^3$ floating point operations and $3n^3$ memory accesses. The only exception is "i" < "j" < "k" order where the number of memory accesses can be optimized as we mentioned previously.

2.2 Fixed-size-blocks algorithms

An interesting property of a matrix is that we can decompose it on several blocks and consider it as a matrix whose elements are other matrices. Consider an example with 4x4 matrix decomposed on 2x2

blocks:

$$A = \begin{pmatrix} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} & \begin{pmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{pmatrix} \\ \begin{pmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix} & \begin{pmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{pmatrix} \end{pmatrix} =: \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

Using this idea we can suggest a more complicated algorithm of matrix multiplication. An important property of block decomposition is the following: assume you have block decomposition of (n,n)-matrix $A = (A_{ij})_{i,j=1}^{m}$ and (n,n)-matrix $B = (A_{ij})_{i,j=1}^{m}$, then as long as each $A_{ik}B_{kj}$ is properly defined we will have

$$AB = \left(\sum_{k=1}^{m} A_{ik}B_{kj}\right)_{i,j=1}^{m} = \left(\sum_{k=1}^{n} a_{ik}b_{kj}\right)_{i,j=1}^{n}.$$

In listing 3 we give a C++-style pseudocode of block-matrix multiplication. Each multiplication of blocks of size $b * b$ takes $2b^3$ multiplications and additions in *block_dgemm* function and b^2 additions in *add_from_block* function. Now assuming the side of square input matrices - n is divisible by the side of block - b, then we will do $(\frac{n}{b})^3$ outer loop iterations. Therefore the total number of FLOP is $(\frac{n}{b})^3 2b^3 + (\frac{n}{b})^3 b^2 = 2n^3(1 + \frac{1}{2b})$

The usefulness of this idea comes from the fact that while input matrices A, B can be too large to fit into the cache of a processor we can always find block size such that any pair of blocks A_{ik}, B_{kj} fit into cache. Thus properly chosen block size can decrease average memory access time and increase the implementation performance. However, a poorly chosen block size will only slow the implementation since we need additional time to copy blocks into the cache and add them back to the output matrix. We tested the block sizes $b \in \{2, 16, 32, 64\}$.

```
// A, B, C - nxn-matrices
// Aik, Bkj, Cij - bxb-blocks of A,B,C
for (int k = 0; k < n / b; k++)
  for (int i = 0; i < n / b; i++)
    for (int j = 0; j < n / b; j++) {
      get_block(A, i, k, Aik);
      get_block(B, k, j, Bkj);
      block_dgemm(Aik, Bkj, Cij);
      add_from_block(Cij, i, j, C);
    }
```

Listing 3: Here *get_block* is a function copying block from a matrix into additional memory; *block_dgemm* is a function of matrix multiplication chosen for multiplying blocks; *add_from_block* is a function adding the result of block multiplication to the original output matrix. We intentionally did not show how to pass the parameters n and b to all these three functions. While parameter n was passed as a regular parameter, for b we tested two options - pass it as a parameter or hardcode it using template syntaxis in C++. The last method

2.3 Strassen-Winograd algorithm

Both algorithms we considered before are doing the same set of operations and differ only in the way they manage the memory and order the operations. Surprisingly there are algorithms computing the product of two matrices with less than $O(n^3)$ complexity. We considered one of the easiest such algorithms which was originally developed by Strassen, 1969 and later modified by Winograd, 1971.

Firstly let's explain how this method works. Assume we want to find product $XY = Z$. Then firstly divide X, Y, Z on 4 blocks:

$$X =: \begin{pmatrix} a & b \\ c & d \end{pmatrix}, Y =: \begin{pmatrix} A & C \\ B & D \end{pmatrix}, Z =: \begin{pmatrix} t & u \\ v & w \end{pmatrix}.$$

Now compute t, u, v, w using scheme below:

$$\begin{aligned} t &:= aA & t &:= t + bB \\ u &:= (c-a)(C-D) & u &:= w + v + (a+b-c-d)D \\ v &:= (c+d)(C-A) & v &:= w + u + d(B+C-A-D) \\ w &:= t + (c+d-a)(A+D-C) & w &:= w + u + v \end{aligned}$$

As you can verify this scheme uses only seven multiplications. Importantly we can compute the output using only 15 block additions and no additional memory. To achieve this we used the computational scheme proposed by Boyer et al., 2009 which we display in figure 1. We do not give pseudocode of the algorithm due to its length, but the actual C++ implementation can be found in our repo by this link dgemm-strassen-winograd.

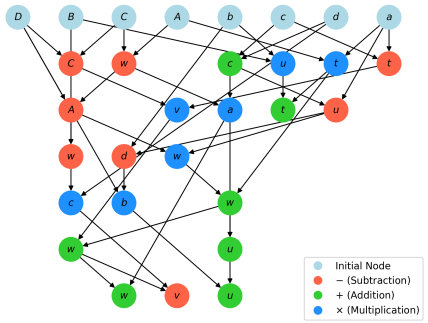

Figure 1: The graph of operations in the implementation of the Strassen-Winograd algorithm we used. As you can see there are 7 blue nodes

Now the computation complexity of our implementation is $T(n) = 7T(n/2) + 15(n/2)^2 = 7T(n/2) + \frac{15}{4}n^2$ with initial condition $T(1) = 1$. Assume $n = 2^k$ (and therefore $k = \log_2(n)$), then:

$$\begin{aligned} T(n) &= \frac{15}{4}n^2 + 7\left(\frac{15}{4}(n/2)^2 + \ldots + 7\left(\frac{15}{4}(n/2^{k-1})^2 + 7\right)\right) = \\ &= \frac{15}{4}n^2\left(1 + \frac{7}{4} + \frac{7^2}{4^2} + \ldots + \frac{7^{k-1}}{4^{k-1}}\right) + 7^k = \\ &= \frac{15}{4}n^2\frac{(\frac{7}{4})^k - 1}{\frac{7}{4} - 1} + 7^k = 5n^2\left(\frac{7}{4}\right)^{\log_2 n} - 5n^2 + 7^k = \\ &= 5n^2(2^{\log_2(7/4)})^{\log_2 n} - 5n^2 + 7^k = \\ &= 5n^{2+\log_2(7/4)} - 5n^2 + 7^k = 5n^{\log_2 7} - 5n^2 + 7^k = \\ &= 5n^{\log_2 7} - 5n^2 + 7^{\log_2 n} = 5n^{\log_2 7} - 5n^2 + (2^{\log_2 7})^{\log_2 n} \\ &= 6n^{\log_2 7} - 5n^2 \end{aligned}$$

For $n = 2$ this formula correctly returns 22, for $n = 4$ it returns 214, etc. Now solving numerically $2n^3 \geq 6n^{\log_2 7} - 5n^2$ we see that n should be > 286 to get a smaller number of floating point operations. Therefore we also considered a variation of this algorithm such that it stops recursion early and performs a regular matrix multiplication if the input matrix size is lower than a chosen threshold. We considered this thresholds from the set $\{16, 64, 256\}$. In figure 2 we show the complexity of the algorithm with a chosen threshold.

3 Results

In the previous section, we described the algorithms we are testing in this paper. In this section, we describe the environment in which we tested the algorithms and present the performance results.

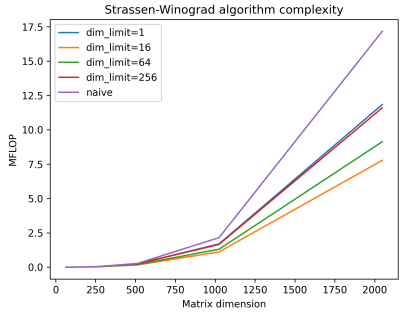

Figure 2: MFLOP for Winograd-Strassen algorithm with different number cutoff parameters for switching to the traditional algorithm.

3.1 Computational platform and software environment

We performed tests on a machine with SLES 15.4 OS (linux) provided to us by NERSC. The machine had an AMD EPYC 7763 CPU with 3.1GHz maximal clock rate and 4/64/256 MiB of L1/L2/L3 cache correspondingly. CPU was connected to 503GiB of RAM with 205GB/s memory bandwidth.

We implemented all previously discussed algorithms in $C++$ language and compiled the code with GCC 12.3 compiler with '-march=native' and '-O3' optimization flags. In our tests, we used CBLAS from Cray's LibSci library.

3.2 Test methodology

To estimate the runtime of each algorithm we developed a benchmark program. This program generates input and output square matrices with the dimension from a given list, fill them with random numbers, and then runs the matrix multiplication algorithms measuring their runtime and checking the correctness of the output comparing the output matrix to the result of the CBLAS library's dgemm.

In section 2 for each algorithm we theoretically calculated the number of floating point operations that will be performed on input matrices with a given dimension n. Using this theoretical complexity and runtime we then derived MFLOP/s metric: $\text{MFLOPs} = \frac{(\text{theoretical \# FLOP})/10^6}{\text{runtime}}$.

There was one important exception to this rule - since we wanted to compare our programs to CBLAS we also needed to derive FLOP for their implementation. Unfortunately, Cray's LibSci library is not open-sourced so we could not check what algorithm they are using. However, since other BLAS implementations are using cubic algorithms (e.g. netlib) we assumed that the theoretical FLOP-complexity of CBLAS implementation is $2n^3$.

3.3 Three-loops MMs vs CBLAS

Recall from section 2.1 that our motivation for developing algorithms with loop orderings different from natural lexicographical ordering "i" < "j" < "k" was that during each inner loop iteration, we perform $C[in+j] = A[in+k]B[kn+j]$ memory access, what means that the algorithm has bad spatial locality. As you can see from figure 3 this hypothesis was exactly right and algorithms with "i" < "k" < "j"/"k" < "i" < "j" have much better performance than all other loop orderings. Furthermore for large problem size "i" < "k" < "j" outperforms "k" < "i" < "j" which is also expected since we have two memory accesses of type $in+\ldots$ and only one of type $kn+\ldots$. Despite these improvements, we did not get close to BLAS performance.

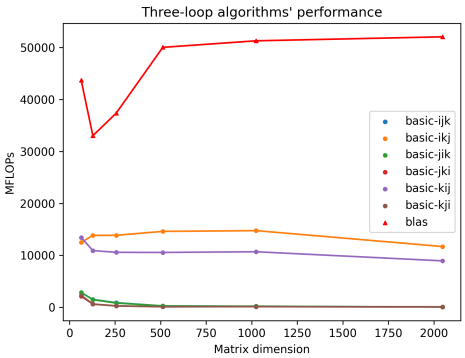

Figure 3: Comparison of various loop orderings influence on MFLOP/s metric. Here and in following figures matrix dimension is the number of rows/columns in a square matrix. "i" < "k" < "j" ordering outperforms lexicographical orderings 58 times on average and 196 times at maximum achieved on the problem size 2048.

3.4 Block-matrix MMs vs CBLAS

Now for block-matrix implementation, our incentive was to utilize cache usage. As we mentioned in 3.1 in our system we had 4MiB of L1 cache or $4 \cdot 2^20 = 2^{22}$B. The size of double in C++ is 8B, therefore three blocks of size 64x64 will take $3 \cdot (2^6)^2 \cdot 8 = 3 \cdot 2^15$B, what means that the size of all tested blocks perfectly fits into L1 cache. This means that performance should increase with increasing block size. As you can see from figure 4 this hypothesis is confirmed by the results of our experiment. Similarly to the previous section, these improvements did not help us achieve BLAS performance.

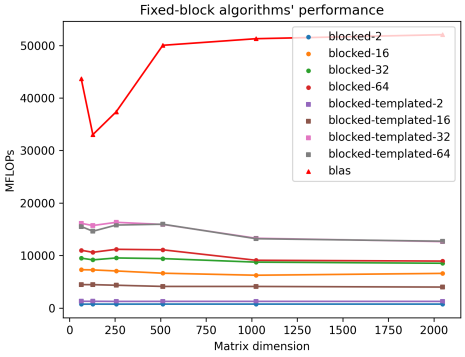

Figure 4: Comparison of block size influence on the performance of blocked algorithm. As it should be expected implementation with templated block size is much faster than those with block size passed as a parameter. The reason is that with maximal optimization compiler uses some additional optimization based on the block size. Templated-32 blocked algorithm outperforms basic-ijk algorithm 61 times on average and 208 times at the maximum achieved on the problem size 2048.

3.5 Strassen MMs vs CBLAS

As for our final implementation - Strassen-Winograd algorithm the figure with MFLOP/s results (5) is not inversely proportional to the runtime since the algorithms blas and Strassen-Winograd have different number FLOP. For this reason, we also included a figure with runtime itself (6). Regardless of the choice of metric (runtime/MFLOPs) the Strassen-Winograd algorithm which switches on regular multiplication has the best performance among all other variations.

This algorithm is also the fastest of all other implementations tested in this paper except BLAS which is again a huge winner by far.

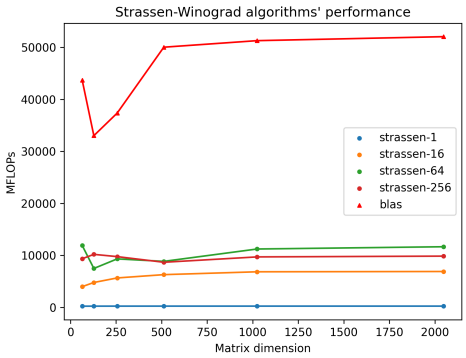

Figure 5: MFLOPs performance results for Strassen-Winograd algorithms with various cutoff parameters. We see that the algorithm with cutoff 64 performs best, while the algorithm with cutoff 1 performs worst.

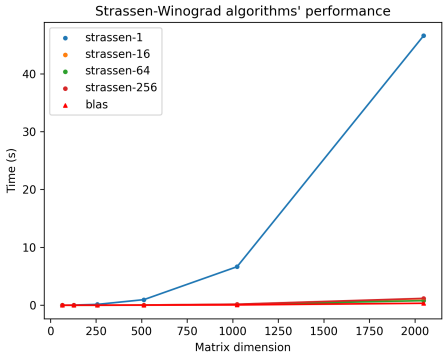

Figure 6: Strassen-32 implementation outperforms basic-ijk implementation 88 times on average and 369 at maximum achieved on problem size 2048. However, it is still ≈ 3 times slower on average than BLAS implementation.

3.6 Overall findings and discussion

In this paper, we discussed several ways to increase matrix multiplication performance. Two of them were focused on increasing spatial locality/utilizing cache usage i.e. on improving hardware-related features of the code. The third approach focused on improving the algorithm itself by decreasing its theoretical complexity. In all approaches, we confirmed our hypotheses about potential improvement and as a result got several implementations of matrix multiplication which are several dozens-hundreds times faster than the naive algorithm. However, all these approaches are still much slower than CBLAS implementation, therefore we have to continue our research. Potential directions for future research include SIMD, pipelining, and usage of optimal small matrix multiplication algorithms (such as Sedoglavic, n.d.).

4 AI STATEMENT

While writing this paper I *used* the Grammarly tool for checking grammar (free version), Google Search/Google Scholar for web searches. I *didn't use* AI for writing the code/doing the calculations/writing the report.

REFERENCES

Boyer, B., Dumas, J.-G., Pernet, C., & Zhou, W. (2009). Memory efficient scheduling of strassen-winograd's matrix multiplication algorithm. *Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, 55–62.

Sedoglavic, A. (n.d.). Fast matrix multiplication database [Accessed: 2024-09-21].

Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische mathematik*, *13*(4), 354–356.

Winograd, S. (1971). On multiplication of $2\times$ 2 matrices. *Linear algebra and its applications*.