Comparing different Matrix Multiplication techniques CP#2, CSC 746, Fall 2023

Shubh Pachchigar*
SFSU

ABSTRACT

This study delves into the relationship between various memory utilization procedures and their impact on computational performance. Our investigation centers on the fundamental task of matrix multiplication, conducted across an increasingly larger problem sizes, with the goal of understanding and exploiting the memory utilization capabilities inherent in the underlying hardware architecture. Three distinct implementations of matrix multiplication are carefully examined throughout this study. The first implementation represents the elementary approach to matrix multiplication, while the second involves blocking technique aimed at optimizing memory utilization. The third and final implementation utilizes a vendor-provided BLAS code, serving as a benchmark reference for comparison. Performance assessment is conducted utilizing MFLOPS as the primary metric, enabling a precise and quantitative evaluation and comparison of the three implementations. Our findings reveal that the utilization of blocking/tiling techniques does indeed yield performance improvements. However, it is noteworthy that even with these enhancements, the performance still falls short of the efficiency achieved by the reference BLAS code.

1 INTRODUCTION

In the realm of computer algorithms, a multitude of opportunities for enhanced speed and efficiency lies not solely by crafting optimized algorithms but performance improvements can be achieved through the strategic manipulation of how data is arranged in memory. This study delves deep into the details of this phenomenon, shedding light on the 'how' and 'why' behind it. To do so, we center our investigation around a common computational task: matrix multiplication, implemented in three distinct ways. Our primary aim is to study the behavior of computer memory, specifically the cache, and decipher how exploiting its capabilities can lead to significant reductions in program runtime. Each of the three scenarios explored in this study leverages fast memory in slightly different manners, offering unique insights into the optimization potential.

The first approach employs a straightforward matrix multiplication algorithm, known for its cubic polynomial time complexity. This method has substantial computational costs as it demands fetching multiple rows and columns from the main memory which might or might not fit completely into cache, resulting in infrequent data transfers. Detailed exploration of this method is undertaken in Section 2.1. In contrast, the second scenario adopts a more sophisticated strategy known as "blocking." Instead of processing large rows and columns, the matrix is divided into smaller, equally sized blocks, enabling matrix multiplication to be executed on these more manageable chunks which can easily reside in cache. Additionally, a technique referred to as "copy optimization" is applied. This arrangement optimally organizes the blocks, facilitating their direct retrieval from the high-speed cache memory. Further elaboration on this approach is provided in Section 2.2. The third scenario involves the utilization of a pre-existing library code, serving as our reference benchmark. This code employs multiple techniques, including cache blocking and loop optimizations. A comprehensive description of this approach is outlined in Section 2.3.

Following sections will shed light on the significant difference in performance among these three implementations. Notably, the naive algorithm shows a computational bottleneck, resulting in a marked decline in MFLOPS—a metric employed to measure system performance. Conversely, the blocked matrix multiplication, coupled with copy optimization, demonstrates a remarkable ability to harness cache memory efficiently, breaking down the problem into more manageable segments. Nevertheless, the reference code continues to outperform the rest. A pivotal takeaway from this study demonstrates the effectiveness of blocking with copy optimization, in enhancing performance when compared to naive algorithm. This work explores the trends between memory usage and algorithmic execution, offering valuable insights into computational efficiency.

*email:spachchigar@sfsu.edu

2 IMPLEMENTATION

This section presents the C++ implementations of the three distinct scenarios designed to explore cache behavior and its effect on overall performance. Each of these implementations will be discussed comprehensively. The first subsection will provide pseudocode, the specific implementation and ways on how to interpret it along with its computational complexity. Moving forward, the second subsection introduces a technique known as "blocking," known for its capacity to enhance cache utilization by breaking the problem into smaller, more manageable segments. A detailed explanation of this approach, along with the pseudocode, will be provided. Lastly, the third subsection serves as a reference benchmark, against which the other two implementations will be compared. This section will show the library used and provide insights into its underlying operations.

2.1 Basic Matrix Multiplication

This implementation describes the basic matrix multiplication using three nested loops without any optimization. In this approach as shown in Listing 1, we use one-dimensional array with a size of N^2, where N represents the problem size. All matrices are stored in column-major order, and the multiplication is carried out accordingly. The process involves taking a column from matrix A and multiplying it with the corresponding row of matrix B. This operation is repeated for all rows, resulting in a dot product where N values are multiplied and N values are added. The use of the expression $j+kn$ for the A array pointer allows it to move more frequently compared to the B array pointer, which moves with $in+k$. Even when N is relatively small, this approach may not take full advantage of the cache due to the disparate access patterns between columns and rows within a one-dimensional array. Consequently, it fails to exhibit any cache locality principles.

2.2 Blocked Matrix Multiplication with Copy Optimization

This pseudocode in Listing 2 represents the implementation because the original code is quite extensive. We initiate the process by allocating memory three times the size of the block. We also take into account the size of a double datatype on the machine which is

```
void square_dgemm(int n, double *A, double *B,
    double *C)
{
  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[j + k * n] * B[i * n + k
    ];
      }
    }
  }
}
```

Listing 1: Basic Matrix Multiplication: take pointer to three matrices and performs C=C+A*B

useful for copy. We then store pointers to these memory segments in Acopy, Bcopy, and Ccopy. These arrays are spaced apart by 'blocksize' * 'blocksize' memory locations. We begin by copying the contents of matrix C into Ccopy using the memcpy function. This operation copies 'blocksize' number of values row by row. It repeats this process, moving to the next row, until it has copied 'blocksize' * 'blocksize' elements into the first memory division. It's crucial to observe that, by doing this, we transform the otherwise randomly accessed C matrix into a contiguous memory layout. This ensures that due to spatial locality, the data is brought into the fast memory or cache when an attempt to read is made. While we may still encounter compulsory misses, this method effectively reduces capacity misses, thanks to the smaller 'blocksize'. We follow a similar strategy by copying matrix A into Acopy and matrix B into Bcopy again by using memcpy function. Once all arrays are prepared, we proceed with the standard matrix multiplication operation. For each iteration, we focus on a group of columns in matrix A, storing only a fixed block, and similarly, we handle each group of rows of matrix B, storing only a fixed block of size 'blocksize' * 'blocksize.' Once again, the technique presented in Section 2.1 is employed here, where the iteration over column groups in matrix A is faster compared to matrix B. The final outcome should be identical to the standard matrix multiplication result. Crucially, due to the efficient organization of the copy arrays, all blocks are strategically placed in the cache due to the principle of spatial locality. Once multiplied the Ccopy is copied back to the original C array. The performance improvement becomes evident when we examine the number of floating-point operations per second, as discussed in Section 3.

2.3 CBLAS Reference

This implementation uses a library function from CBLAS header file and invokes the function *cblas_dgemm*(). Because of all matrices are stored as one dimensional array in column major order we pass in string named CblasColMajor to indicate column major order and CblasNoTrans tells that no transpose should be performed on A and B. All of this is shown in Listing 3. Here alpha and beta which indicate scaling of matrices are set to 1 and *N* represent the size of matrices. The *cblas_dgemm*() function is part of the Basic Linear Algebra Subprograms (BLAS) library, which is a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. The BLAS library is widely used in scientific computing and machine learning applications to speed up matrix operations. The *cblas_dgemm*() function uses a variety of techniques to speed up matrix multiplication, including loop optimization techniques and cache-blocking. Loop optimization techniques involve optimizing the order of nested loops and

```
void square_dgemm_blocked(int n, int block_size,
    double *A, double *B, double *C)
{
   ALLOCATE SPACE FOR THREE 1D MATRICES

   for (int i0=0;i0<n;i0+=block_size)
   {
      for (int j0 = 0; j0 < n; j0 += block_size)
      {
         COPY C[i,j] -> Ccopy
         for (int k0 = 0; k0 < n; k0 += block_size
    )
         {
            COPY A[k0,j0] -> Acopy
            COPY B[k0,i0] -> Bcopy

            for (int i1 = 0; i1 < block_size; i1
    ++)
            {
               for (int j1 = 0; j1 < block_size;
    j1++)
               {
                  for (int k1 = 0; k1 < block_size
    ; k1++)
                  {
                     Ccopy[i1 * block_size + j1]
    += Acopy[k1 * block_size + j1] * Bcopy[i1 *
    block_size + k1];
                  }
               }
            }
            COPY Ccopy -> C[i,j]
         }
      }
   }
}
```

Listing 2: Blocked Matrix Multiplication: take pointer to three matrices and performs C=C+A*B by copying first into a cache and then performing normal multiplication

minimizing the number of memory accesses to improve cache locality.

```
void square_dgemm(int n, double* A, double* B,
    double* C) {
    cblas_dgemm(CblasColMajor, CblasNoTrans,
    CblasNoTrans, n, n, n, 1., A, n, B, n, 1., C,
    n);
}
```

Listing 3: CBLAS Reference: takes pointer to three matrices and performs C=C+A*B by introducing multiple optimizations

3 Evaluation

This section delves deeper into how the implementations presented in Section 2 are evaluated across a range of increasing problem sizes. The initial part outlines the required environment for replicating the experiments, while the subsequent sections provide detailed descriptions of the experiments conducted. The first experiment compares the basic implementation described in Section 2.1 with the reference BLAS implementation, as detailed in Section 2.3. This comparison provides insights into the relative performance of these two approaches. The second experiment extends this analysis by comparing the blocked implementation, as outlined in Section 2.2,

with the reference BLAS implementation across varying problem sizes and block sizes. This examination sheds light on the impact of different factors on performance.

3.1 Computational platform and Software Environment

The system in use is Perlmutter at NERSC, equipped with hardware specifications including a 2 MiB L1 cache, 32 MiB L2 cache, and 256 MiB L3 cache. It runs on the SUSE Linux Enterprise Server 15 SP4 operating system, version 5.14.21-150400.24.69 12.0.74-cray shasta c, and is powered by an AMD EPYC 7763 Milan CPU capable of clock rates ranging from 1.5 GHz to 2.45 GHz. The system has 256GB of DRAM, utilizes the GCC 11.2.0 compiler from Cray Inc. with default optimization levels set to O3, and employs CMake version 3.20.4 alongside GNU Make version 4.2.1, built for x86 architecture.

3.2 Methodology

In this section, we will subject the implementations discussed in Section 2 to a series of tests and comparisons with a reference implementation. Each of these implementations will be executed with varying problem sizes, [64, 128, 256, 512, 1024, 2048]. Each problem size corresponds to the dimensions of a square array, where each 'N' implies an 'N*N' matrix. All three implementations will undergo testing with randomly generated values sampled from a Gaussian distribution. Following the completion of these experiments, we will evaluate and compare their performance using the MFLOPS metric, which measures millions of floating-point operations per second. This metric allows us to assess how effectively both the hardware and software leverage their capabilities to achieve optimal performance. We will conduct these assessments through two distinct experiments. To measure the elapsed time of the matrix multiplication, we will employ the chrono timer library. To calculate MFLOPS, we require two essential components: the total number of arithmetic operations, which depends upon the problem sizes, and the elapsed time of the code. The formula used to calculate MFLOPS for both the blocked and basic matrix multiplication implementations is described below:

$$\text{MFLOPS} = \frac{num\ of\ arithmetic\ ops}{elapsed\ time} = \frac{2N^3}{elapsed\ time}$$

3.3 Experiment 1: Comparing Basic vs Reference BLAS

The experiment in this section aims to compare the basic matrix multiplication against the reference BLAS implementation. We seek to determine which implementation is better and makes more efficient use of the underlying hardware. Both implementations undergo testing across a range of increasing problem sizes, as outlined in 3.2. Table 1 displays the problem sizes alongside the corresponding elapsed times for both the basic matrix multiplication and the reference BLAS implementation. A clear trend emerges from this table: as the problem size doubles, the basic implementation experiences an exponential increase in elapsed time, whereas the BLAS code exhibits relatively stable performance. We can further visualize this distinction with a graph. Fig. 1 illustrates how the MFLOPS (which is log-scaled)of both implementations compare across various problem sizes. Once again, we observe the same pattern: an exponential decay in performance for the basic implementation, due to its inverse relationship with elapsed time, and the consistent and stable performance of the reference BLAS implementation. The decay can be attributed to the frequent memory access which introduces a latency spike thus affecting performance.

3.4 Experiment 2: Comparing Blocked MM vs Reference BLAS

In this section, we describe an experiment that compares the blocked implementation, as detailed in Section 2.2, with the reference BLAS code. The objective here is to investigate whether the introduction

Problem Size (N)	Basic MM (ms)	BLAS (ms)
64	0.21	0.06
128	2.98	0.13
256	36.89	0.92
512	812.39	5.90
1024	8736.75	44.02
2048	184983.20	351.35

Table 1: Comparison of Basic Multiplication and Reference BLAS elapsed runtimes for different problem sizes.

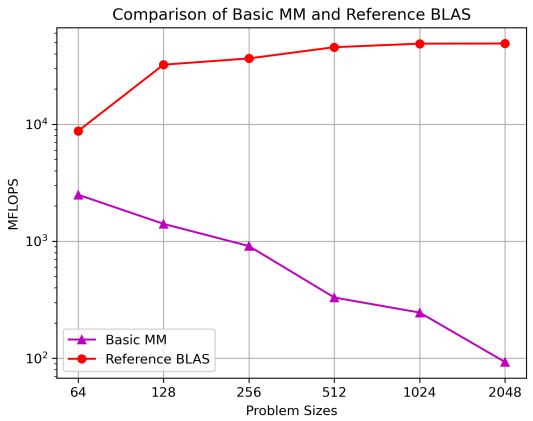

Figure 1: Comparison of MFLOPS of Basic MM vs.Reference BLAS with increasing problem sizes. The vertical axis to log-scaled in the Python script.

of the blocking technique with copy optimization enhances or diminishes performance when compared to the reference implementation. The experiment maintains the same problem sizes as outlined in 3.2, but now introduces varying block sizes, ranging from [2, 16, 32, 64]. Each block size corresponds to a one-dimensional array of size 'blocksize * blocksize'. The entire code is tested across all the mentioned block sizes, in addition to the various problem sizes, and the results are presented graphically in Figure 2. The plotted data reveals several insights. When using the smallest block size of 2, we observe the lowest MFLOPS (log-scaled). The major reason for difference is might be because the computation for these small block size of 2 is done so fast that most of time is spend in moving data from main memory to cache thus dimishing the speedup provided by computations from cache. With increasing block sizes, we notice a relatively stable MFLOPS performance as the problem size increases. The reason for stable performance for increasing problem sizes can be attributed to speedup from reduced memory access time due to cache, this mitigates the detrimental effects of memory latency. Notably, this performance is significantly higher than that achieved by the basic implementation. However, it's important to emphasize that the reference BLAS implementation consistently outperforms all block sizes. In summary, findings suggest that due to the efficient utilization of cache through spatial locality, we observe a increase in MFLOPS for the system, even though it leads to more complicated code.

3.5 Findings and Discussion

This study sought to understand the role of cache in system performance. Our first experiment revealed that an inefficient memory access pattern, without using any of cache locality principles, detrimentally affects system performance as problem size increases. Conversely, in the second experiment, we demonstrated that a straightforward blocking technique, which divides the problem into smaller

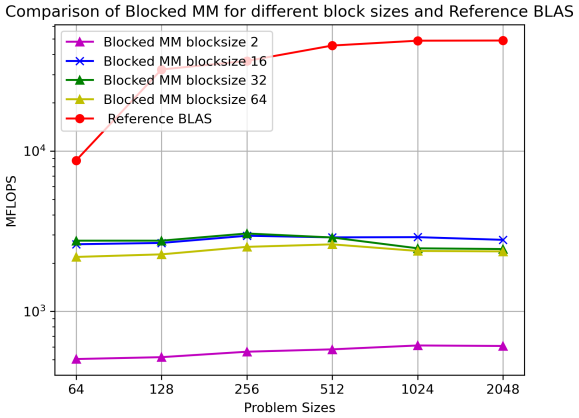

Figure 2: Comparison of Blocked MM vs BLAS with increasing problem sizes for increasing block sizes. The vertical axis is log-scaled in the Python script that generates the chart.

segments, significantly enhances system performance, as evidenced by the plotted results as high as 10x. The key takeaway from the comparison between Figure 1 and Figure 2 is quite evident. The basic implementation showed a marked inefficiency, peaking at 2500 MFLOPS (Mega FLOPS) for the smallest problem size of 64 and declining to a mere 100 MFLOPS for the largest. In contrast, the blocked implementation reached its highest performance, approximately 3000 MFLOPS for a problem size of 256 and block size 16 and lowest for 500MFLOPS for a problem size of 64 and block size 2. Notably, the reference implementation consistently delivered a peak output of around 45 Giga FLOPS. Hence, we can assert that the strategic arrangement of memory in the cache, as demonstrated in Section 2.2, significantly boosts system performance. It is essential to underscore that both implementations in Sections 2.1 and 2.2 featured an equivalent number of arithmetic operations, as described in Section 3.2. In conclusion, the reference BLAS implementation emerges as the top performer, while the basic implementation lags behind, demonstrating the least efficiency. The blocked code, on the other hand, provides the most stable results and showcases the benefits of optimizing memory utilization.