High Performance Linear Algebra: Matrix Multiplication

Alex Lozano, Luke Venkataramanan

December 2024

Alex Lozano

UT EID:	AJL4846
TACC Username:	alozano0304
Email:	alozano0304@utexas.edu

Luke Venkataramanan

UT EID:	LV8828
TACC Username:	luke_venk
Email:	luke.venkataramanan@utexas.edu

1 Introduction

Linear algebra plays a foundational role in computational engineering and science, with applications ranging from solid-state physics to machine learning. While these linear algebra operations are relatively straight-forward to naively implement, these approaches lack efficiency. In computational engineering, where performance is critical and problems are large scale, it is important to rely on methods that maximize performance.

High-performance linear algebra has evolved as a field over the decades, driven by technological innovations for hardware and the increasing complexity of engineering problems. In the 1950s, FORTRAN was introduced, providing the first practical framework for scientific computation. In the 1970s, the first Basic Linear Algebra Subprograms (BLAS) was created to implement low-level routines to common linear algebra operations, such as matrix multiplication.

This paper focuses on increasing the performance of matrix multiplication, a fundamental operation in linear algebra. As described in the next section, the naive implementation of a matrix-matrix product requires a triple nested loop, which has a time complexity of $O(N^3)$, assuming the matrices have dimensions N-by-N. While this is acceptable for smaller matrices, this is impractical for matrices whose dimensions reach thousands or millions. At such large scales, optimization strategies are essential.

We take advantage of how data is stored in the computer memory in order to increase the speed of matrix multiplication, especially when the sizes of these matrices are very large. Specifically, we found that using a recursive block multiplication technique, which in our case is an example of cache-oblivious programming, significantly improves the performance of computing a matrix-matrix product.

For instance, we found the TACC Stampede3 Supercomputer can find the product of 2 large square matrices, with dimension size ≈ 4000 (over 16 million elements), 800% faster when using the recursive method rather than the naive method. In the following sections, we outline our methodology, implementations, and results, providing insight into how recursive cache-oblivious techniques can unlock higher performance for large matrix computations.

2 Permutations for the Matrix-Matrix Product

The naive implementation of a matrix-matrix product uses the following form:

$$c_{ij} = \sum_k a_{ik} \cdot b_{kj}$$

This takes the inner product of a given row of A and a given column of B to find a single element in matrix C. This approach yields the following triple nested loop:

```
for (i=0; i < a.rows; ++i)
  for (j=0; j < b.cols; ++j)
    for (k=0; k < a.cols; ++k)
      c[i, j] += a[i, k] * b[k, j];
```

This approach can be permuted; in fact, there are 6 different implementations with which you can perform this operation. It would even theoretically be possible to choose random i, j, and k values as long as all combinations are chosen once. Here, we chose to switch the j and k values:

```
for (i = 0; i < a.rows; ++i)
  for (k = 0; k < a.cols; ++k)
    for (j = 0; j < b.cols; ++j)
      c[i, j] += a[i, k] * b[k, j];
```

This implementation adds row-wise to the elements of C, so we call it row-wise addition based, rather than inner product based.

3 Recursive Algorithm for Matrix-Matrix Multiplication

We want our algorithm for matrix-matrix multiplication to be cache oblivious, meaning we use a divide-and-conquer recursive strategy that automatically uses all levels of the cache hierarchy [2]. Our recursive algorithm specifically is designed for square matrices with a dimension of 2^n. This is based on the aptly named block algorithm that separates matrices into 4 submatrices and computes the product, $A \cdot B = C$, as shown:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

The resulting submatrix computations are:

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \end{aligned}$$

This block algorithm can be called recursively to find the matrix-matrix product much faster than the naive approach. We split submatrices into half their dimensions (breaking them into quarters) until they can fit entirely into the L1 and L2 caches, decreasing the time it takes to retrieve full columns of data from extremely large matrices out of RAM. We further discuss this spatial locality below. This block algorithm is the basis for the well-known Straussen algorithm which saves a multiplication step [10]. However, the Straussen algorithm is not very architecturally friendly. It requires a lot of movement between matrices which means more retrieval from RAM to cache, and thus slower recursive functionality. It can be useful to split the first 2 or 3 recursions using Straussen but after that it becomes unhelpful. [9] Shown below is the pseudocode for our recursive algorithm.

```
Divide the dimensions of the matrix by 2;
Return submatrices of A as tuple (A11, A12, A21, A22);
Return submatrices of B as tuple (B11, B12, B21, B22);
Return submatrices of C as tuple(C11, C12, C21, C22);

// C11 = (A11 * B11) + (A12 * B21)
Call recursive formula for C11 += A11 * B11;
Call recursive formula for C11 += A12 * B21;

// C12 = (A11 * B12) + (A12 * B22)
Call recursive formula for C12 += A11 * B12;
Call recursive formula for C12 += A12 * B22;

// C21 = (A21 * B11) + (A22 * B21)
Call recursive formula for C21 += A21 * B11;
Call recursive formula for C21 += A22 * B21;

// C22 = (A21 * B12) + (A22 * B22)
Call recursive formula for C22 += A21 * B12;
Call recursive formula for C22 += A22 * B22;
```

Each time the function is called, the matrix's dimensions are divided in half, splitting it into 4 submatrices. This process continues until the dimensions reach a predefined base case, or the smallest recursion size. After that, the naive formula is used to compute the corresponding sub-block of C.

```
for(int i = 0; i < dim; i++) {
    for(int j = 0; j < dim; j++) {
        for(int k = 0; k < dim; k++){
            // C[i, j] += A[i, k] * B[k, j]
            out.set(i, j, out.at(i, j) + this->at(i, k) *
                other.at(k, j));
        }
    }
}
```

Once this is completed, the function returns and goes back up the recursive call stack, combining the sub-blocks into the larger blocks of C, until the entire matrix C is computed.

4 The Leading Dimension of A

When we create a matrix, we visualize it as a 2D array of data. However, in reality, this data is stored in the computer's memory as a 1D array of elements. This can either be done in column-major ordering, or row-major ordering. While general purpose languages like Java, C, and C++ (our language of choice for this project) use row-major order by default, we choose to implement column-major ordering to match the examples in the textbook [3].We assume we have a matrix with M rows and N columns. The formula for the location of element (i, j), considering column-major ordering, is $i + j \cdot M$ [6].

Consider a matrix A_1 as a submatrix of a larger matrix A. This is useful because if we wanted to use a specific region of the larger matrix A, it would be impractical to copy all these elements into their own contiguous region of memory. Thus, we describe the elements of our smaller matrix in terms of the memory layout of the larger matrix A.

This introduces a few complications. Previously, we considered that the location of element (i, j) could be found using the formula $i + j \cdot M$. However, if our matrix is embedded within A, we'll need to use a new formula to find this element. This introduces the concept of the leading dimension of A, abbreviated as LDA.

The LDA is the number of elements between the start of one column and the start of the next column [8], or more simply, the number of rows in A. Our new formula becomes the following:

$$\text{Location of element}_{(i,j)} = i + j \cdot \text{LDA}$$

5 Implementing the Matrix Class Using std::span

Now that we have found the formula for the location of an element using the LDA of the larger array within which the matrix is embedded, we shift our focus to the implementation of a Matrix class.

Now, using std::vector for this, as the textbook points out, is problematic. For one, C++ mandates that std::vector maintains tight control over its own memory [3], so we wouldn't be able to make a subarray using the vector's memory directly. Thus, making a subarray with std::vector would require copying all the necessary elements to a new memory location, which is inefficient.

The solution is to utilize std::span, introduced in C++20. Span can be used to efficiently handle subarrays without directly copying data. Instead of copying data or taking ownership over a block of memory, it just uses a pointer to reference the data and keeps track of its size [7].

Let us outline the following scenario. Assume we have a smaller matrix A_1 with M = 3 rows and N = 3 columns. A_1 is embedded within a larger matrix A, with the leading dimension of A being 5 rows, while it has the same number of columns as A. Assume A has the following values:

$$\text{A: } \{2, 5, 8, 1, 7, 3, 9, 6, 4, 0, 2, 9, 1, 5, 8\}$$

Because we are using column-major ordering, A takes the form:

$$\text{A} = \begin{bmatrix} 2 & 3 & 2 \\ 5 & 9 & 9 \\ 8 & 6 & 1 \\ 1 & 4 & 5 \\ 7 & 0 & 8 \end{bmatrix}$$

If we say that for A_1, M = 3, and N = 3, assuming we want A_1 to start from the first element of A, then A_1 should only encapsulate the top 3×3 square of A. Then, A_1 should look like the following:

$$A_1 = \begin{bmatrix} 2 & 3 & 2 \\ 5 & 9 & 9 \\ 8 & 6 & 1 \end{bmatrix}$$

Our implementation of the Matrix class can be found in the source code, but its constructor takes in A_1's number of rows and columns, the leading dimension of A, the index denoting where exactly A_1 starts in A, as well as the data of A.

Our method to safely access elements uses the formula we discussed in the previous section, $i + j \cdot LDA$. See below our code snippet of the function.

```
double& at(int i, int j) {
    int index = i + j*LDA;
    return data[index];
}
```

Using this formula, this function successfully accesses the element (i, j) in the smaller matrix A_1, using the LDA and data from larger matrix A. We can confirm this function works as intended by printing out all the values of A_1, and indeed, A_1 looks as expected.

6 Implementing the Matrix Class Using mdspan

To improve on this implementation we decided to use the mdspan feature, introduced in C++23. mdspan is a span with multi-dimensional indexing. Because the mdspan library is not yet implemented in all compilers, in order to utilize mdspan in our code, we had to clone it from GitHub from the Kokkos project [5]. We added these files in our repository in a directory we called "external." Finally, we updated CMakeLists.txt to include the mdspan library and compile using C++23.

In order to format our data correctly, we had to define the extents type for our Matrix class. mdspan requires an extents type to produce an efficient way to store the dimensions of a multi-dimensional array, as well as the layout of the data. For our purposes, we used type int for indexing, with dynamic values for the number of rows and columns, since these values are not known at compile time. We also used layout_left since we utilize column-major ordering.

```
using extents_T = md::extents<int, std::dynamic_extent,
    std::dynamic_extent>;
...
class Matrix {
private:
    int M;
    int N;
    int LDA;
    md::mdspan<double, extents_T, md::layout_left> data;
...
```

Although we could have used the same indexing scheme we used for the span class, we wanted to make use of the multi-dimensioning indexing, in our case, indexing a matrix using rows and columns. Thus, we came up with the following scheme: (1) Taking in indices i and j, calculate the linearized index with the formula we previously discovered in the context of the larger matrix, (2) Solve for the row index in the context of the smaller matrix, (3) Solve for the column index in the context of the smaller matrix, (4) Return the element at those indices using mdspan's 2D indexing. See the code below:

```
double& at(int i, int j) {
    int linearIndex = i + j*LDA; // Column-major ordering
    int rowIndex = linearIndex % M;
    int colIndex = linearIndex / M;
    return *(data.data_handle() + data.mapping()(rowIndex,
        colIndex));
}
```

Let's dive further into detail as to why this works. The first line should be self explanatory. For the second line, in column major ordering, elements in same column are stored contiguously. Thus, the different rows have a repeating pattern, which means the row index of the smaller matrix corresponds to the linearized index of the larger matrix modulus the number of rows in the smaller matrix. In the third line, using similar reasoning to the previous step, the column index of the smaller matrix corresponds to the linearized index of the larger matrix divided by the number of rows in the smaller matrix.

7 Simple Operations: Matrix Addition

We then turned our attention to implementing a function for matrix addition. We used a naive approach; we simply checked that the dimensions of the two matrices were the same, and then iterated through the rows and columns, adding the values in each location to a new matrix. This approach worked even if the matrices had different LDAs, as long as the sizes of their rows and columns were the same.

```
for (int i = 0; i < rowsA; ++i) {
    for (int j = 0; j < colsB; ++j) {
        // C[i, j] = A[i, j] + B[i, j]
        C.set(i, j, this->at(i, j) + other.at(i, j));
    }
}
```

8 Using Preprocessors for Optimized Indexing

Currently, our indexing does not provide any bounds checking. We use preprocessors to either define the mode as either release or debug. If the program is in release mode, in order to optimize indexing, we won't bother bounds checking. If the program is in debug mode, we are willing to sacrifice optimization in order to make debugging easier.

```
#define DEBUG
...
double& at(int i, int j) {
    #ifdef DEBUG
        if (i >= M || j >= N) {
            throw out_of_range("Error: Index out of bounds");
        }
    #endif
```

From then on, the at function is the same as previously shown in section 7.

9 Decomposition into Submatrices

We use mdspan to facilitate the recursive decomposition of the matrices into smaller submatrices. This approach is designed to ensure the submatrices fit in the faster caches, allowing the Arithmetic logic unit (ALU) to compute the multiplication more efficiently.

We create 4 submatrices in each iteration of the blocked algorithm: the top left, the top right, the bottom left, and the bottom right. These submatrices are then also broken down recursively into their own submatrices, until it can entirely fit in the cache. We use a function called getSubmatrices(), which returns a tuple of these 4 submatrices, each using their own helper function.

```
Matrix TopLeft(int m, int n) {
    return Matrix(m, n, LDA, data.data_handle(), topLeftIndex);
}

Matrix TopRight(int m, int n) {
    return Matrix(m, n, LDA, data.data_handle(), topLeftIndex +
        n*LDA);
}

Matrix BotLeft(int m, int n) {
    return Matrix(m, n, LDA, data.data_handle(), topLeftIndex +
        m);
}

Matrix BotRight(int m, int n) {
    return Matrix(m, n, LDA, data.data_handle(), topLeftIndex +
        n*LDA + m);
}

tuple<Matrix, Matrix, Matrix, Matrix> getSubmatrices(){
    return {TopLeft(M/2, N/2), TopRight(M/2, N/2), BotLeft(M/2,
        N/2), BotRight(M/2, N/2)};
}
```

10 Methods for Matrix Multiplication

Our recursive matrix multiplication scheme has 3 main functions: BlockedMatMult, RecursiveMatMult, and MatMult. Each of these are methods of our Matrix class. If we consider an example where matrices $A \cdot B = C$, then A would be equivalent to ”this”, B would be ”other”, and C would be ”out.”

BlockedMatMult is our starter function, which then calls our recursive formula. This function first initializes an empty output matrix C. The function then breaks the matrices A, B, and C into their submatrices, passing the 8 of them into the first iteration of the recursive formula. We later add parallelization to this top level function.

```
Matrix BlockedMatMult(Matrix& other) {
    auto [M, N, LDA] = this->getDimensions();

    double* CDataPtr = new double[M*N];
    fill_n(CDataPtr, M * N, 0.0);
    Matrix C = Matrix(M, N, LDA, CDataPtr);

    auto [tla, tra, bla, bra] = this->getSubmatrices();
    auto [tlb, trb, blb, brb] = other.getSubmatrices();
    auto [tlc, trc, blc, brc] = C.getSubmatrices();
    int dim = M/2;

    // C11 = (A11 * B11) + (A12 * B21)
    tla.RecursiveMatMult(tlb, tlc, dim); // C11 += A11 * B11
    tra.RecursiveMatMult(blb, tlc, dim); // C11 += A12 * B21

    // C12 = (A11 * B12) + (A12 * B22)
    tla.RecursiveMatMult(trb, trc, dim); // C12 += A11 * B12
    tra.RecursiveMatMult(brb, trc, dim); // C12 += A12 * B22

    // C21 = (A21 * B11) + (A22 * B21)
    bla.RecursiveMatMult(tlb, blc, dim); // C21 += A21 * B11
    bra.RecursiveMatMult(blb, blc, dim); // C21 += A22 * B21

    // C22 = (A21 * B12) + (A22 * B22)
    bla.RecursiveMatMult(trb, brc, dim); // C22 += A21 * B12
    bra.RecursiveMatMult(brb, brc, dim); // C22 += A22 * B22

    return C;
}
```

RecursiveMatMult is our recursive function that, depending on the size of the matrix, breaks it further into submatrices, or computes the product naively. Once enough recursions are done, the matrices will fit into the CPU caches, and this dramatically speeds up computation time as data can be loaded into the register while Fused Multiply-Add (FMA) is occurring in the ALU.

```
void RecursiveMatMult(Matrix& other, Matrix& out, int dim){
    if (dim > x){
        dim /= 2;
        auto [tla, tra, bla, bra] = this->getSubmatrices();
    ...
        tla.RecursiveMatMult(tlb, tlc, dim); // C11 += A11 * B11
        tra.RecursiveMatMult(blb, tlc, dim); // C11 += A12 * B21
    ...
    }
    else {
        this->MatMult(other, out, dim);
    }
}
```

Once our submatrix is small enough, we call MatMult, which computes the product naively. The implementation for this is straightforward and matches what we discussed in section 3.

11 Performance and Timing

Now we can see how fast our recursive algorithm is compared to the naive implementation. All these tests were done with base case matrix size of 2×2 and compiled with the optimization level -O3, both of which were experimentally tested to be the fastest parameters.

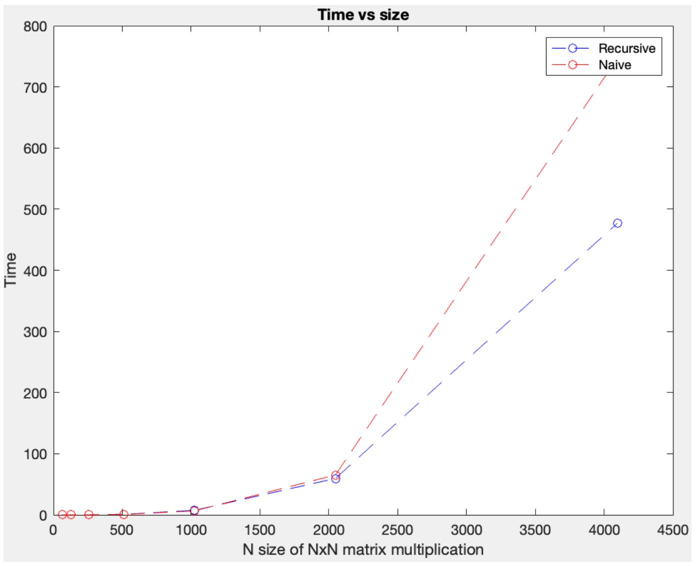

Figure 1: Time vs. N for NxN matrix

Figure 1 shows how the time increases on the order of $O(n^3)$ for the Naive (red) implementation, and $O(n^2)$ for the Recursive (blue) implementation. Note how the times diverge as N increases, with the recursive taking nearly half the time at N = 4096. In this case, the matrices have 16,777,216 elements each, with the naive computation taking about 12 minutes.

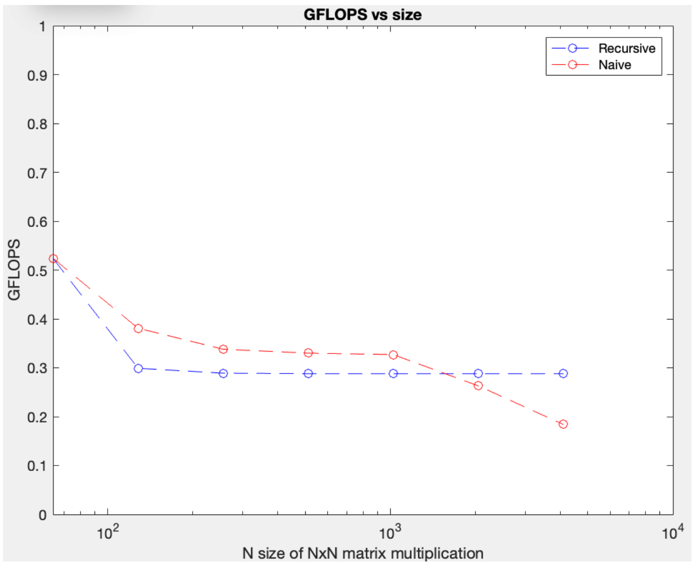

Figure 2: GFLOPS vs. N for NxN matrix

Figure 2 shows the same graph but with Gigaflops rather than time. This graph essentially shows how many floating point operations per second each algorithm can do. Interestingly, at smaller matrix sizes, the naive can outperform the recursive algorithm. This is likely due to the overhead of the recursive algorithm. Increasing the smallest recursion size would help with these issues, but would hinder us at larger matrix sizes. At larger matrix sizes, the naive version takes an increasingly large amount of time. This is because its trying to grab each row and column and at some point they don't fit into the L1, L2, or L3 cache, and this creates high amounts of latency. The recursive formula levels off at around .31 GFLOPS, as it continuously reuses the same data. This is what we would expect.

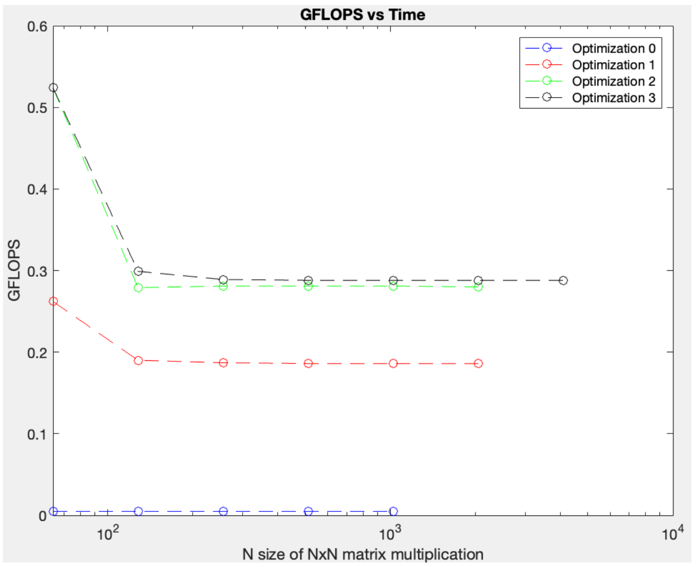

Figure 3: Recursive Approach - Varying optimization levels

We also tested the different optimization levels to find out which one would be faster. Figure 3 shown in GFLOPS shows that O3 is marginally faster than O2, which are both faster than any other optimization level. O0 was by far the worst, it was almost impossible to compute large matrices, we couldn't even finish testing. O3 can be dangerous, sometimes the large amount of machine code needed can actually speed up compilation and run time, so it was important to us to find the best level of Optimization. O3 and O2 were very close. O1 took about twice the time as O3 and O0 was nearly 100× slower. We ended up using O3 for all other tests.

12 Recursive Strategy: Data Reuse

With the subject of data reuse we introduce the terms cache hit and cache miss. When the CPU requests data from the main memory (RAM), it loads data into the cache in fixed-size chunks called cache lines [1]. Because our matrices store data in column-major ordering, entire columns of data are stored contiguously in the memory, and thus in the cache lines.

A cache hit occurs when the requested data is already in the cache, allowing the CPU to fetch the data with minimal delay. A cache miss occurs when the requested data is not already in the cache, forcing the CPU to retrieve an entire cache line from the RAM, causing significant delays.

The naive matrix-matrix product implementation is unlikely to reuse data. Here is a what a for loop for such an implementation would look like:

```
for (int i = 0; i < rowsA; i++) {      // Rows of A,
    for (int j = 0; j < colsB; j++) {  // Columns of B, C
        for (int k = 0; k < sharedDim; k++) { // Shared dimension
            (columns of A, rows of B)
            C[i][j] += A[i][k] * B[k][j]; // Matrix multiplication
        }
    }
}
```

As you can see, in the outer loop, we iterate over the rows of A. We hold i constant in a given iteration as k changes in the inner loop, which means we traverse the columns of a given row in A. As previously mentioned, our memory is stored in a column-major order. Thus, as we advance along different columns of a row, we continuously access non-contiguous memory, leading to constant cache misses and no data reuse.

Furthermore, in the inner loop, we iterate over the columns of B. We change k while holding j constant, so we traverse the rows of a given column in B. This actually will lead to cache hits and some data reuse, since columns of data are stored contiguously in column-major ordering. However, as j changes, the cache must be reloaded with a new column of data.

Overall, the naive matrix-matrix product implementation leads to several cache misses and minimal data reuse. On the other hand, the recursive strategy does lead to data reuse. As matrices are recursively split into smaller blocks, their dimensions eventually become small enough for them to fit in the CPU cache. For example, once A_{11} and B_{11} are loaded into the cache, all their elements are reused multiple times to compute C_{11} without any cache misses.

In the naive implementation, there were cache misses as we traverse the columns of a given row in A, since columns are stored contiguously in column-major ordering. However, when we recursively split A into submatrices small enough to fit into the cache, accessing the next column doesn't cause cache misses, unlike in the naive approach.

In conclusion, the naive approach causes more cache misses and is unlikely to lead to data reuse, while the recursive strategy causes cache hits and leads to data reuse. Data reuse leads to more efficient performance because loading data is slower than executing the actual operations.

13 Cache Sizes of Processor

If the product is already contained in the cache, operations on any of these elements would already be cache hits and not cache misses. While this still would be true if we continued to recurse into smaller submatrices, this would just lead to unnecessary operations, which wouldn't speed up retrieval of data anyways. These unnecessary operations would just lead to more computational overhead; if anything, this would slow things down.

The TACC Stampede3 supercomputer uses an Intel Xeon CPU MAX 9480 ("Sapphire Rapids HBM"). The cache sizes are 48 KB L1 data cache per core, 1 MB L2 per core, and 112.5 MB L3 per socket.

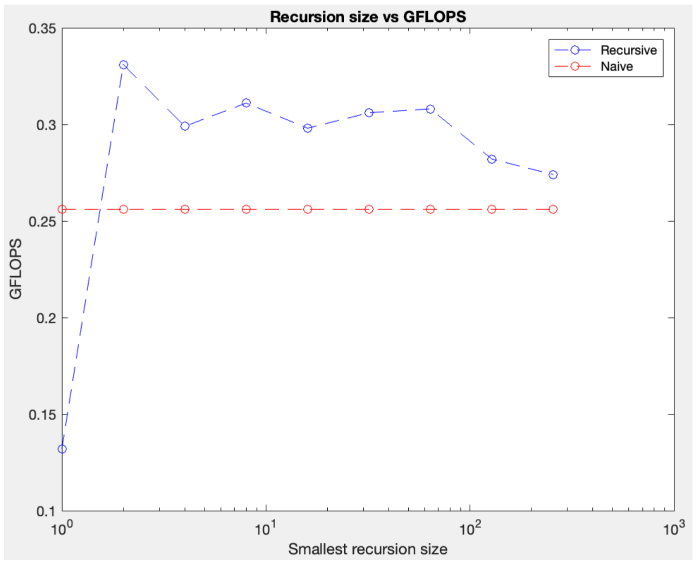

Figure 4: GFLOPs vs. Smallest Recursion Size

We tested various cutoff points for the smallest recursion size. We found that for a 1024×1024 matrix, the fastest base case matrix size was a 2×2 matrix. On the lower end, at a base case size of 1, the recursive takes nearly quadruple the time as naive. At higher recursion sizes, 4×4-64×64, the GFLOPs stay nearly constant. Once the recursion size reaches 128 there is a steady decrease that asymptotically approaches the naive GFLOPs. From this we can assume the smallest cache size is around $64(N) \cdot 64(N) \cdot 3(matrices) \cdot 64(bits) = 786KB$ which is approximately equivalent to the L2 cache size of 1 MB.

14 Parallelism

The following 4 equations target independent areas of the C matrix, meaning we can execute these in parallel, since the Stampede3 processor has at least 4 cores (in fact it has more than 140,000 cores).

$$
\begin{aligned}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\
C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}.
\end{aligned}
$$

In order to parallelize our top level function, BlockedMatMult, we use the OpenMP library. We use 3 main commands. *#pragma omp parallel* creates a parallel region for the 4 threads to execute the following recursions in parallel. *#pragma omp single* ensures the blocks of code are executed by one thread. Finally, *#pragma omp task* creates the tasks executed by any thread, enabling parallelism.

```
Matrix BlockedMatMult(Matrix& other) {
    // SAME AS PREVIOUSLY SHOWN IN SECTION 11
    ...

    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            {
                // C11 = (A11 * B11) + (A12 * B21)
                tla.RecursiveMatMult(tlb, tlc, dim); // C11 += A11
                    * B11
                tra.RecursiveMatMult(blb, tlc, dim); // C11 += A12
                    * B21
            }
            #pragma omp task
            {
                // C12 = (A11 * B12) + (A12 * B22)
                tla.RecursiveMatMult(trb, trc, dim); // C12 += A11
                    * B12
                tra.RecursiveMatMult(brb, trc, dim); // C12 += A12
                    * B22
            }
            #pragma omp task
            {
                // C21 = (A21 * B11) + (A22 * B21)
                bla.RecursiveMatMult(tlb, blc, dim); // C21 += A21
                    * B11
                bra.RecursiveMatMult(blb, blc, dim); // C21 += A22
                    * B21
            }
            #pragma omp task
            {
                // C22 = (A21 * B12) + (A22 * B22)
                bla.RecursiveMatMult(trb, brc, dim); // C22 += A21
                    * B12
                bra.RecursiveMatMult(brb, brc, dim); // C22 += A22
                    * B22
            }
        }
    }
    #pragma omp taskwait
}
```

In the command line, we also have to run the command `OMP_NUM_THREADS=4`, in order to specify the number of threads we are using. This change to our function reduced the time of our computation to almost 25% of the original computation, which makes sense since we are parallelizing the process into 4 threads.

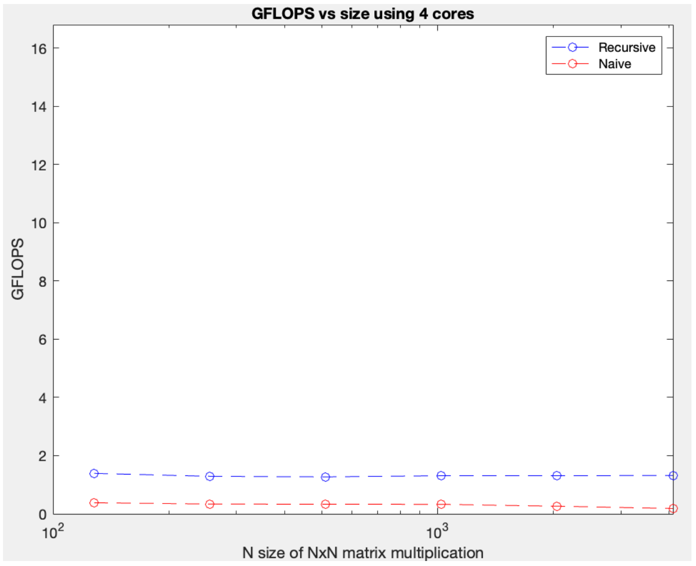

Figure 5: GFLOPs vs. N for NxN matrix

After OpenMP usage, we increased our core count from 1 to 4, allowing us to work at 4× the speed of the maximum of our Xeon 9480. We calculated the max to be $2.1e10^9(Hz) * 48(cores) * 16(operations) = 16.8$ GFLOPs, our maximum output. Figure 5 shows how our recursive formula leads to more GFLOPs, compared to the naive implementation. Runnning at 1.2 GFLOPs, our recursive is 8 times faster at large matrix sizes. This graph is normalized by the maximum GFLOPs of our processor, of which which our program only uses 1/12 the number of cores. Thus, we are still very far off from reaching maximum parallel capacity of our processor.

15 Discussion and Summary

In conclusion, our recursive block approach significantly outperforms the naive matrix multiplication implementation, achieving much faster computations for larger matrices. By leveraging the hierarchical memory structure of processors, particularly the cache, we minimize costly latency and we maximize data reuse. By breaking the matrices down into submatrices small enough to fit into the cache, we increase spatial and temporal locality and boost the performance of our computations.

Our work is valuable for large-scale scientific computations where matrix operations are a bottleneck. However, since our approach was cache-oblivious, the performance of our approach depends on the specific hardware characteristics.

One current limitation is that our program only handles matrices with dimensions equal to a power of two. We do this because we are recursively breaking matrices into quarters, requiring their dimensions to be divisible by two. In reality, block matrix multiplication is still possible when dimensions are not equal to powers of two, as long as the partitions within the matrix line up [4], but we implemented our approach as such for simplicity.

Another potential limitation is that we use C style arrays to initialize our data for our matrices. We took this approach because we ran into issues where we would use std::vector and the .data() function to create our Matrix objects, but if these matrices were created in functions which returned a new matrix from operations like additions or multiplication, as soon as the function returned, the original vector would go out of scope. This led to the original data pointer to be null, leading to memory issues. Instead, we opted to use C-style arrays which allocates data on the heap, so we could circumvent these scope issues. However, using C-style arrays requires manual memory management, and these raw pointers can be hard to read and maintain. Regardless, we were diligent with memory management, and we did not run into any issues.

An opportunity for future extension would be to use a cache-aware approach, specifically tailoring the algorithm to work best with the processor of Stampede3. Because we know the cache sizes Stampede3 uses, we could break the smallest matrices into the largest sizes such that the matrices of A, B, and C all fit within the L1 cache. However, because cache sizes vary greatly between processors, this would mean the high performance of our code would not be portable [2].

Another opportunity for future extension could be to apply our findings to a real-world application, such as machine learning. Our high performance matrix multiplications could be used in neutral network training, especially for large-scale models. However, the primitive data types used in machine learning aren't always doubles, so we could improve our Matrix class by templatizing it for other data types.

References

[1] Creel. Performance x64: Cache blocking (matrix blocking). https://www.youtube.com/watch?v=G92BCtfTwOE, 2017.

[2] Victor Eijkhout. *Introduction to High-Performance Scientific Computing*, volume 1. Published under the CC-BY 4.0 license, 2022.

[3] Victor Eijkhout. *Introduction to Scientific Programming in C++17/Fortran2008*, volume 3. Published under the CC-BY 4.0 license, 2022.

[4] Nathaniel Johnston. Linear algebra - lecture 12: Block matrices. https://www.youtube.com/watch?v=KCUgWj5nhYc, 2021.

[5] Kokkos. mdspan. https://github.com/kokkos/mdspan, 2023.

[6] Shashank Prasanna. Row-major vs. column-major matrices: A performance analysis in mojo and numpy. *Modular*, 2024.

[7] Mike Shah. Stl std::span — modern cpp series ep. 115. https://www.youtube.com/watch?v=OQu2pZILjDo, 2023.

[8] Robert van de Geijn. 1.2.2 the leading dimension of a matrix. https://www.youtube.com/watch?v=PhjildK5oO8, 2019.

[9] Robert van de Geijn. *Linear Algebra Foundations to Frontiers-On Programming for High Performance:*. Published under the CC-BY 4.0 license, November 25, 2021.

[10] S. Winograd. On multiplication of 2 × 2 matrices. *Linear Algebra and its Applications*, 4(4):381–388, 1971.