Project 5: General Matrix Multiplication

Name: Bryan Anthony
SID: 12112230

Environtment:

- Ubuntu 22.04.1 LTS
- g++ (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
- AMD A8-7410 processor
- 8GB memory
- x86 platform

Part 1 - Analysis

BLAS (Basic Linear Algebra Subprograms) consists of routines or functions that provide standard building blocks for performing basic vector and matrix operations. Among these routines, General Matrix Multiplication (GEMM) is one of the most commonly used functions in BLAS libraries. The GEMM operation is defined as $\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C}$.

In the `cblas_dgemm` function, the first three parameters determine the order and transpose options for matrices A and B. There are two types of ordering: row-major and column-major ordering.

1. Row-major order:
   In row-major order, the matrix elements are stored sequentially by rows. Elements within the same row are stored consecutively in memory. When accessing the matrix elements, you traverse the rows first, and then move to the next row. Row-major order is commonly used in programming languages like C and C++.

   For example, a 3x3 matrix in row-major order is stored in memory as [a11, a12, a13, a21, a22, a23, a31, a32, a33].
2. Column-major order:
   In column-major order, the matrix elements are stored sequentially by columns. Elements within the same column are stored consecutively in memory. When accessing the matrix elements, you traverse the columns first, and then move to the next column. Column-major order is commonly used in programming languages like Fortran.

   For example, a 3x3 matrix in column-major order is stored in memory as [a11, a21, a31, a12, a22, a32, a13, a23, a33].

The purpose of this project is to optimize `cblas_dgemm` by using only column-major ordering without transposing matrices A and B. Additionally, there is another parameter involved:

- `M`: Number of rows in matrices A and C.
- `N`: Number of columns in matrices B and C.
- `K`: Number of columns in matrix A and number of rows in matrix B.
- `alpha`: Scaling factor for the product of matrices A and B.
- `A`: Matrix A.
- `LDA`: The leading dimension of matrix A, which represents the number of elements between consecutive rows or columns of A. If row-major order is used, `LDA` is typically the number of columns in A. If column-major order is used, `LDA` is typically the number of rows in A.
- `B`: Matrix B.
- `LDB`: The leading dimension of matrix B, similar to `LDA`, but applied to matrix B.
- `beta`: Scaling factor for matrix C.
- `C`: Matrix C.
- `LDC`: The leading dimension of matrix C, similar to `LDA` and `LDB`, but applied to matrix C.

For simplicity and ease of comparison, this project assumes matrices A, B, and C have the same size (n = m = k) and sets alpha and beta to 1.

To test the optimization, we will compare it with the `cblas_dgemm` function using various sizes: n=m=k=100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000, 2100, 2200, 2300, 2400, 2500, 2600, 2700, 2800, 2900, and 3000.

In this project, I am unable to utilize AVX2 and AVX512 instructions on my laptop, which limits my optimization options. However, I can still achieve significant performance improvements by utilizing SSE (Streaming SIMD Extensions) intrinsics. It's important to note that there are other matrix multiplication optimization techniques, such as the Strassen and Winograd algorithms. However, in this project, I will focus on optimizing cache utilization through register blocking rather than exploring alternative algorithms.

Register blocking is an optimization technique that aims to improve cache performance by dividing matrices into smaller blocks. By employing register blocking, I aim to maximize data locality and minimize memory access overhead. Although my laptop may not support certain advanced optimizations, register blocking can still significantly enhance performance by leveraging cache efficiency.

1. Naive Approach:
    - The naive DGEMM (Double-precision General Matrix Multiplication) approach is a basic algorithm that follows the traditional row-major or column-major traversal.
    - It performs individual element multiplications and additions without any specific optimizations.
    - The naive approach serves as a baseline for comparison.
2. Register Blocking 2x2:
    - Register blocking, also known as loop blocking, improves cache utilization by dividing the input matrices into smaller blocks, typically 2x2 in size.
    - By operating on these smaller blocks, the algorithm can effectively reuse data from the CPU cache, reducing the need for frequent memory accesses.
    - Register blocking 2x2 leads to better cache performance compared to the naive approach, but its effectiveness may vary depending on the hardware architecture.
3. Register Blocking 4x4:
    - Register blocking 4x4 builds upon the concept of register blocking by using larger blocks, typically 4x4 in size.
    - This optimization further enhances cache utilization and reduces memory access overhead.
    - With larger blocks, the algorithm can exploit more data locality and achieve better cache efficiency, resulting in improved performance compared to smaller block sizes.
4. Register Blocking 4x4 with SSE:
    - SSE (Streaming SIMD Extensions) is an instruction set extension that enables parallel processing of data.
    - Register blocking 4x4 with SSE combines the benefits of register blocking 4x4 with the use of SSE intrinsics for vectorized matrix operations.
    - SSE allows simultaneous processing of multiple data elements, resulting in enhanced computational performance.
    - This optimization leverages SSE instructions to perform matrix multiplication more efficiently, taking advantage of parallelism and data-level parallelism.

Here's how register blocking works:

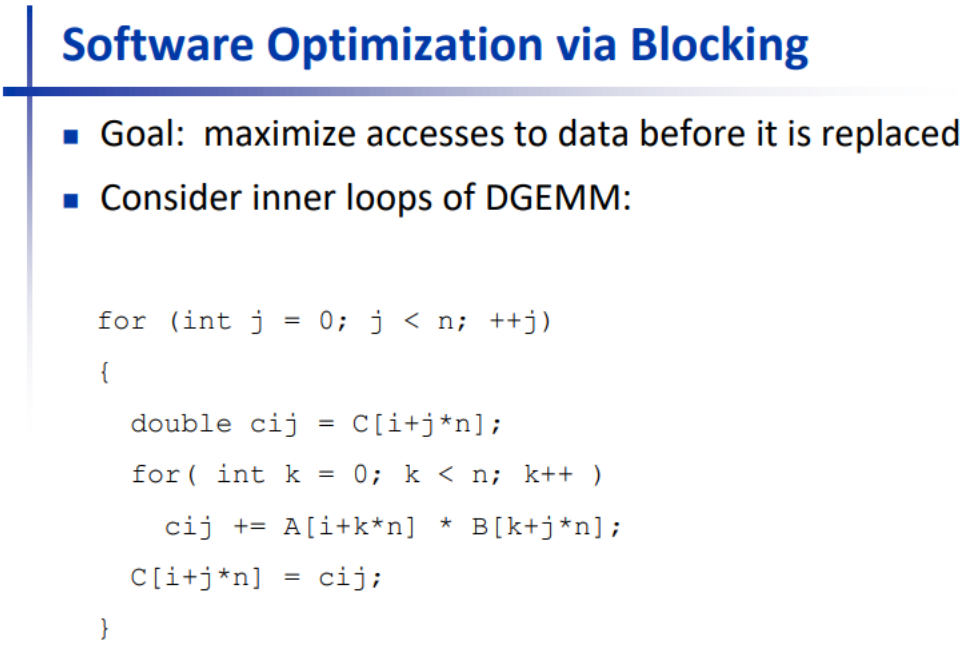
1. **Loop Structure**: The matrix multiplication algorithm is divided into nested loops, typically three loops for iterating over the rows and columns of the matrices.
2. **Partitioning**: The matrices are partitioned into smaller sub-matrices, often referred to as blocks or tiles. The size of these blocks is chosen based on the cache size and cache line size of the target processor.
3. **Block Processing**: The algorithm processes the matrix multiplication in a block-wise manner. It operates on one block at a time, performing the necessary computations within the registers.

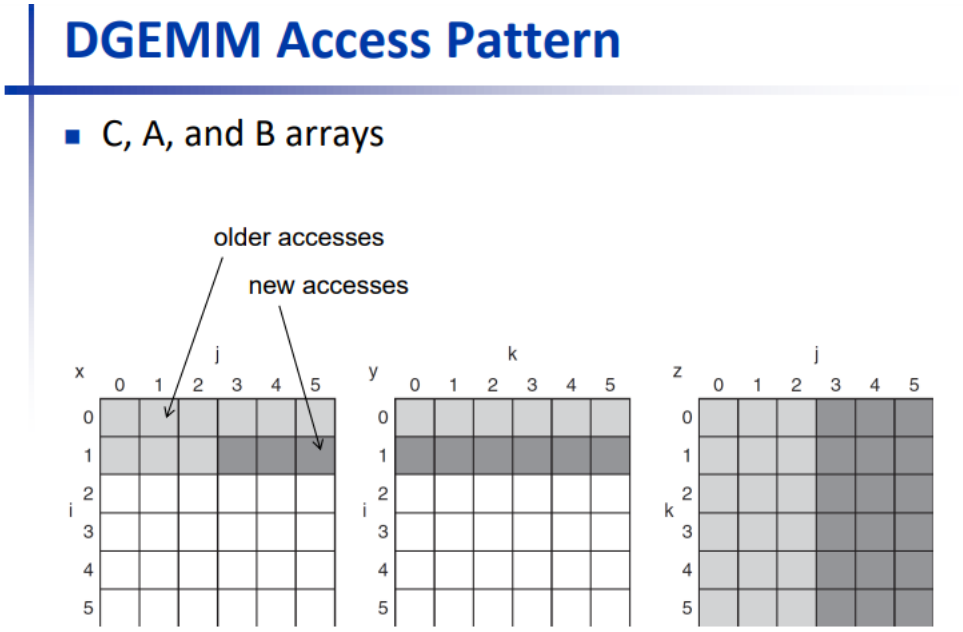
4. **Cache Utilization**: By working on smaller blocks, the algorithm takes advantage of the locality of reference principle. The data elements within a block are reused multiple times before being evicted from the cache, reducing the number of costly memory accesses.
5. **Register Reuse**: The intermediate results are stored in registers instead of the main memory, further reducing memory access latency. This enables efficient computation within the registers, which have much faster access times compared to cache or main memory.
6. **Loop Nest Optimization**: The loop nest is carefully optimized to minimize loop overhead and improve data locality. Techniques such as loop interchange, loop fusion, and loop unrolling may be employed to enhance performance.

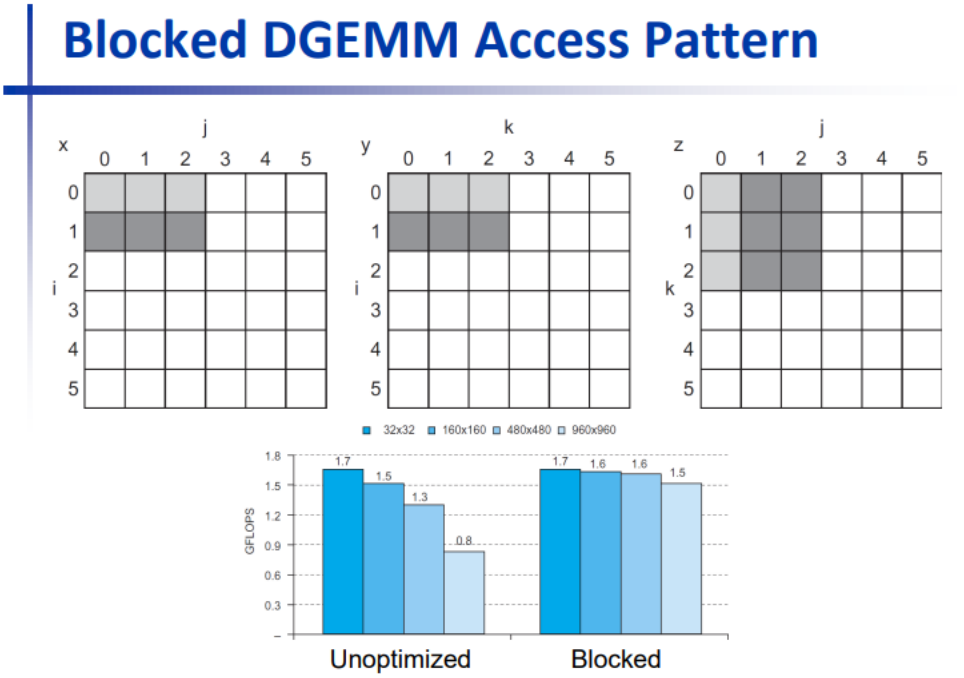
Insights:

- Due to the limitations of AVX2 and AVX512 support on my laptop, SSE intrinsics provide the fastest optimization in this project.
- While matrix multiplication optimizations using Strassen and Winograd algorithms are available, this project focuses on cache utilization through register blocking.
- Register blocking optimizations are effective in improving cache performance by exploiting data locality and reducing memory access overhead.
- The choice of block size (2x2 or 4x4) impacts cache utilization and performance, with larger block sizes generally providing better cache efficiency.
- The incorporation of SSE intrinsics enhances computational performance by leveraging parallel processing and data-level parallelism.

Here is some slides that give a picture about blocking from course CS202 Computer Organization







Part 2 - Code

```
#include "dgemm_opt.hpp"
#include <cblas.h>
#include <iostream>
#include <chrono>
#include <string>
#include <random>
```

```
int len[30] = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100,
               1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000,
               2100, 2200, 2300, 2400, 2500, 2600, 2700, 2800, 2900, 3000};

#define TIME_START start = std::chrono::steady_clock::now();
#define TIME_END(name)                                                          \
  end = std::chrono::steady_clock::now();                                       \
  duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count(); \
  std::cerr << name << "\nm=n=k= " << m << "\ntook time = " << duration << "ms" << std::endl;
void printMatrix(const double *matrix, const int rows, const int cols, std::string note = " ")
{
  std::cout << note << std::endl;
  for (int i = 0; i < rows; ++i)
  {
    for (int j = 0; j < cols; ++j)
    {
      std::cout << matrix[i * cols + j] << " ";
    }
    std::cout << std::endl;
  }
}
int main()
{
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_real_distribution<double> distribution(0.0, 1);
  std::chrono::steady_clock::time_point start, end;
  long long duration;

  for (int j = 0; j < 30; j++)
  {
    int n, m, k;
    n = m = k = len[j];
    double alpha, beta;
    alpha = beta = 1;
    double *A = new double[m * k];
    double *B = new double[n * k];
    double *C = new double[n * m];
    //double *Cref = new double[n * m];

    for (int i = 0; i < m * k; i++)
    {
      A[i] = distribution(gen);
    }
    for (int i = 0; i < n * k; i++)
    {
      B[i] = distribution(gen);
    }
    for (int i = 0; i < n * m; i++)
    {
      C[i] = distribution(gen);
      //Cref[i] = C[i];
    }

    // TIME_START
    // naive_dgemm(m, n, k, alpha, A, k, B, n, beta, C, m);
    // TIME_END("Using naive_dgemm")

    TIME_START
    register_blocking_2x2_dgemm(m, n, k, alpha, A, k, B, n, beta, C, m);
    TIME_END("Using register_blocking_2x2_dgemm")

    // TIME_START
    // register_blocking_4x4_dgemm(m, n, k, alpha, A, k, B, n, beta, C, m);
    // TIME_END("Using register_blocking_4x4_dgemm")

    // TIME_START
    // register_blocking_4x4_SSE_dgemm(m, n, k, alpha, A, k, B, n, beta, C, m);
    // TIME_END("Using register_blocking_4x4_SSE_dgemm")
    // TIME_START
    // MY_MMult(m, n, k, alpha, A, k, B, n, beta, C, m);
    // TIME_END("Using register_blocking_4x4_SSE_dgemm")
    // printMatrix(C, m, n);
    // TIME_START
    // cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k, alpha, A, k, B, n, beta, Cref, m);
    // TIME_END("Using OpenBlass")
    printMatrix(C, m, n);
    free(A);
    free(B);
    free(C);
    //free(Cref);
```

```
  }

  return 0;
}
```

Maybe you wondering why I use std::cerr for outputing the time execution is becuse I can separete whe print output between time execution and the output like this **./a.out > output.log 2> duration.log** when run the compiled program.

```
#pragma once

void naive_dgemm(int M, int N, int K, double alpha, double *A, int LDA, double *B, int LDB, double beta, double *C, int LDC);
void register_blocking_2x2_dgemm(int M, int N, int K, double alpha, double *A, int LDA, double *B, int LDB, double beta, double *C, int LDC
void register_blocking_4x4_dgemm(int M, int N, int K, double alpha, double *A, int LDA, double *B, int LDB, double beta, double *C, int LDC
void register_blocking_4x4_SSE_dgemm(int M, int N, int K, double alpha, double *A, int LDA, double *B, int LDB, double beta, double *C, int
```

```
#include "dgemm_opt.hpp"
#ifdef WITH_SSE
#include "immintrin.h"
#endif
#define A(i, j) A[(i) + (j)*LDA]
#define B(i, j) B[(i) + (j)*LDB]
#define C(i, j) C[(i) + (j)*LDC]

void scalar_times_matrix(double *C, int M, int N, int LDC, double scalar)
{
    int i, j;
    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
        {
            C(i, j) *= scalar;
        }
    }
}

void naive_dgemm(int M, int N, int K, double alpha, double *A, int LDA, double *B, int LDB, double beta, double *C, int LDC)
{
    int i, j, k;
    if (beta != 1.0)
        scalar_times_matrix(C, M, N, LDC, beta);
    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
        {
            for (k = 0; k < K; k++)
            {
                C(i, j) += alpha * A(i, k) * B(k, j);
            }
        }
    }
}

void register_blocking_2x2_dgemm_inner(int M, int N, int K, double alpha, double *A, int LDA, double *B, int LDB, double beta, double *C, i
{
    int i, j, k;
    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
        {
            double tmp = C(i, j);
            for (k = 0; k < K; k++)
            {
                tmp += alpha * A(i, k) * B(k, j);
            }
            C(i, j) = tmp;
        }
    }
}

void register_blocking_2x2_dgemm(int M, int N, int K, double alpha, double *A, int LDA, double *B, int LDB, double beta, double *C, int LDC
{
    int i, j, k;
    if (beta != 1.0)
        scalar_times_matrix(C, M, N, LDC, beta);
    int M2 = M & -2, N2 = N & -2;
```

```
    for (i = 0; i < M2; i += 2)
    {
        for (j = 0; j < N2; j += 2)
        {
            double c00 = C(i, j);
            double c01 = C(i, j + 1);
            double c10 = C(i + 1, j);
            double c11 = C(i + 1, j + 1);
            for (k = 0; k < K; k++)
            {
                double a0 = alpha * A(i, k);
                double a1 = alpha * A(i + 1, k);
                double b0 = B(k, j);
                double b1 = B(k, j + 1);
                c00 += a0 * b0;
                c01 += a0 * b1;
                c10 += a1 * b0;
                c11 += a1 * b1;
            }
            C(i, j) = c00;
            C(i, j + 1) = c01;
            C(i + 1, j) = c10;
            C(i + 1, j + 1) = c11;
        }
    }
    if (M2 == M && N2 == N)
        return;
    // boundary conditions
    if (M2 != M)
        register_blocking_2x2_dgemm_inner(M - M2, N, K, alpha, A + M2, LDA, B, LDB, 1.0, &C(M2, 0), LDC);
    if (N2 != N)
        register_blocking_2x2_dgemm_inner(M2, N - N2, K, alpha, A, LDA, &B(0, N2), LDB, 1.0, &C(0, N2), LDC);
}
void register_blocking_4x4_dgemm_inner(int M, int N, int K, double alpha, double *A, int LDA, double *B, int LDB, double beta, double *C, i
{
    int i, j, k;
    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
        {
            double tmp = C(i, j);
            for (k = 0; k < K; k++)
            {
                tmp += alpha * A(i, k) * B(k, j);
            }
            C(i, j) = tmp;
        }
    }
}

void register_blocking_4x4_dgemm(int M, int N, int K, double alpha, double *A, int LDA, double *B, int LDB, double beta, double *C, int LDC
{
    int i, j, k;
    if (beta != 1.0)
        scalar_times_matrix(C, M, N, LDC, beta);
    int M4 = M & -4, N4 = N & -4;
    for (i = 0; i < M4; i += 4)
    {
        for (j = 0; j < N4; j += 4)
        {
            double c00 = C(i, j);
            double c01 = C(i, j + 1);
            double c02 = C(i, j + 2);
            double c03 = C(i, j + 3);
            double c10 = C(i + 1, j);
            double c11 = C(i + 1, j + 1);
            double c12 = C(i + 1, j + 2);
            double c13 = C(i + 1, j + 3);
            double c20 = C(i + 2, j);
            double c21 = C(i + 2, j + 1);
            double c22 = C(i + 2, j + 2);
            double c23 = C(i + 2, j + 3);
            double c30 = C(i + 3, j);
            double c31 = C(i + 3, j + 1);
            double c32 = C(i + 3, j + 2);
            double c33 = C(i + 3, j + 3);
            for (k = 0; k < K; k++)
            {
                double a0 = alpha * A(i, k);
                double a1 = alpha * A(i + 1, k);
                double a2 = alpha * A(i + 2, k);
```

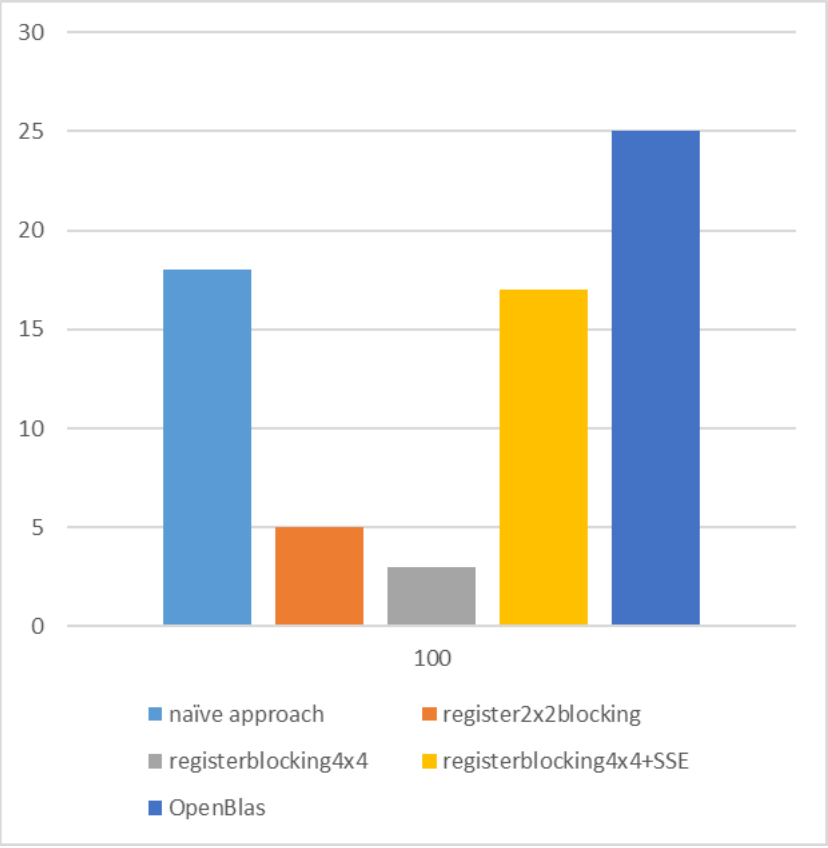
```
                double a3 = alpha * A(i + 3, k);
                double b0 = B(k, j);
                double b1 = B(k, j + 1);
                double b2 = B(k, j + 2);
                double b3 = B(k, j + 3);
                c00 += a0 * b0;
                c01 += a0 * b1;
                c02 += a0 * b2;
                c03 += a0 * b3;
                c10 += a1 * b0;
                c11 += a1 * b1;
                c12 += a1 * b2;
                c13 += a1 * b3;
                c20 += a2 * b0;
                c21 += a2 * b1;
                c22 += a2 * b2;
                c23 += a2 * b3;
                c30 += a3 * b0;
                c31 += a3 * b1;
                c32 += a3 * b2;
                c33 += a3 * b3;
            }
            C(i, j) = c00;
            C(i, j + 1) = c01;
            C(i, j + 2) = c02;
            C(i, j + 3) = c03;
            C(i + 1, j) = c10;
            C(i + 1, j + 1) = c11;
            C(i + 1, j + 2) = c12;
            C(i + 1, j + 3) = c13;
            C(i + 2, j) = c20;
            C(i + 2, j + 1) = c21;
            C(i + 2, j + 2) = c22;
            C(i + 2, j + 3) = c23;
            C(i + 3, j) = c30;
            C(i + 3, j + 1) = c31;
            C(i + 3, j + 2) = c32;
            C(i + 3, j + 3) = c33;
        }
    }
    if (M4 == M && N4 == N)
        return;
    // boundary conditions
    if (M4 != M)
        register_blocking_4x4_dgemm_inner(M - M4, N, K, alpha, A + M4, LDA, B, LDB, 1.0, &C(M4, 0), LDC);
    if (N4 != N)
        register_blocking_4x4_dgemm_inner(M4, N - N4, K, alpha, A, LDA, &B(0, N4), LDB, 1.0, &C(0, N4), LDC);
}
#ifdef WITH_SSE
void register_blocking_4x4_SSE_dgemm_inner(int M, int N, int K, double alpha, double *A, int LDA, double *B, int LDB, double beta, double *
{
    int i, j, k;
    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
        {
            double tmp = C(i, j);
            for (k = 0; k < K; k++)
            {
                tmp += alpha * A(i, k) * B(k, j);
            }
            C(i, j) = tmp;
        }
    }
}

void register_blocking_4x4_SSE_dgemm(int M, int N, int K, double alpha, double *A, int LDA, double *B, int LDB, double beta, double *C, int
{
    int i, j, k;
    if (beta != 1.0)
        scalar_times_matrix(C, M, N, LDC, beta);
    int M4 = M & -4, N4 = N & -4;
    __m128d valpha = _mm_set1_pd(alpha); // broadcast alpha to a 128-bit vector
    for (i = 0; i < M4; i += 4)
    {
        for (j = 0; j < N4; j += 4)
        {
            __m128d c0 = _mm_setzero_pd();
            __m128d c1 = _mm_setzero_pd();
            __m128d c2 = _mm_setzero_pd();
            __m128d c3 = _mm_setzero_pd();
```

```
            for (k = 0; k < K; k++)
            {
                __m128d a = _mm_mul_pd(valpha, _mm_loadu_pd(&A(i, k)));
                __m128d b0 = _mm_load1_pd(&B(k, j));
                __m128d b1 = _mm_load1_pd(&B(k, j + 1));
                __m128d b2 = _mm_load1_pd(&B(k, j + 2));
                __m128d b3 = _mm_load1_pd(&B(k, j + 3));
                c0 = _mm_add_pd(_mm_mul_pd(a, b0), c0);
                c1 = _mm_add_pd(_mm_mul_pd(a, b1), c1);
                c2 = _mm_add_pd(_mm_mul_pd(a, b2), c2);
                c3 = _mm_add_pd(_mm_mul_pd(a, b3), c3);
            }
            _mm_storeu_pd(&C(i, j), _mm_add_pd(c0, _mm_loadu_pd(&C(i, j))));
            _mm_storeu_pd(&C(i, j + 1), _mm_add_pd(c1, _mm_loadu_pd(&C(i, j + 1))));
            _mm_storeu_pd(&C(i, j + 2), _mm_add_pd(c2, _mm_loadu_pd(&C(i, j + 2))));
            _mm_storeu_pd(&C(i, j + 3), _mm_add_pd(c3, _mm_loadu_pd(&C(i, j + 3))));
        }
    }
    if (M4 == M && N4 == N)
        return;
    // boundary conditions
    if (M4 != M)
        register_blocking_4x4_SSE_dgemm_inner(M - M4, N, K, alpha, A + M4, LDA, B, LDB, 1.0, &C(M4, 0), LDC);
    if (N4 != N)
        register_blocking_4x4_SSE_dgemm_inner(M4, N - N4, K, alpha, A, LDA, &B(0, N4), LDB, 1.0, &C(0, N4), LDC);
}
#endif
```

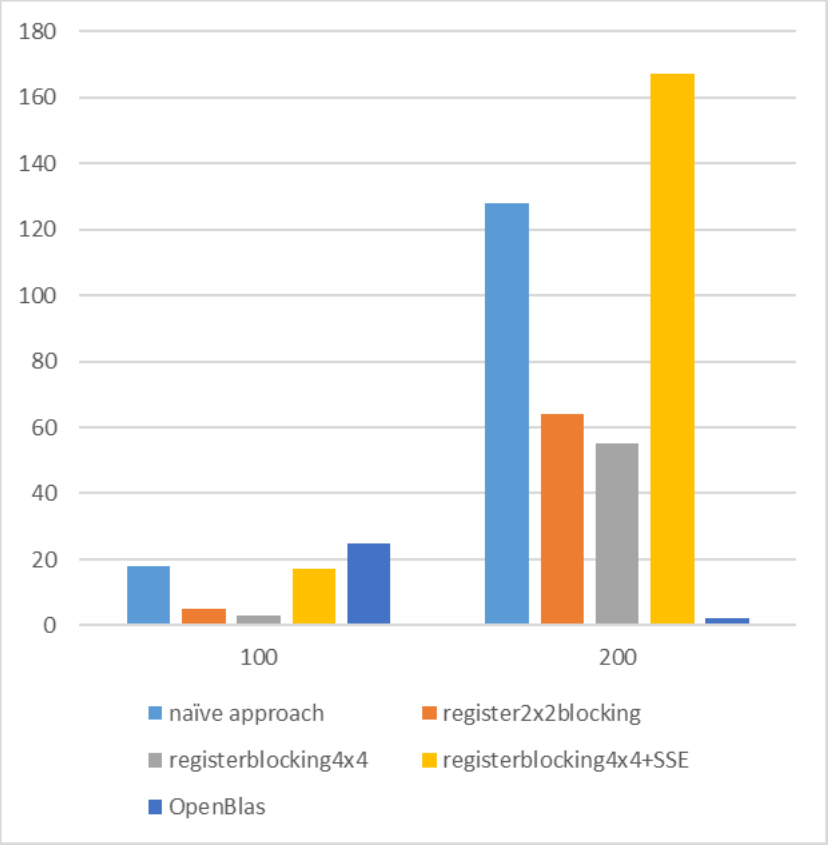
I have also included another optimization technique proposed by Yujiazhai in the file along with my CPP file. This technique has demonstrated promising results and achieved performance levels comparable to OpenBLAS.

Yujiazhai's optimization heavily relies on the AVX512 instruction set. Although I cannot provide the complete code here due to its extensive length of 755 lines, I will share the results obtained from this optimization in the following section.

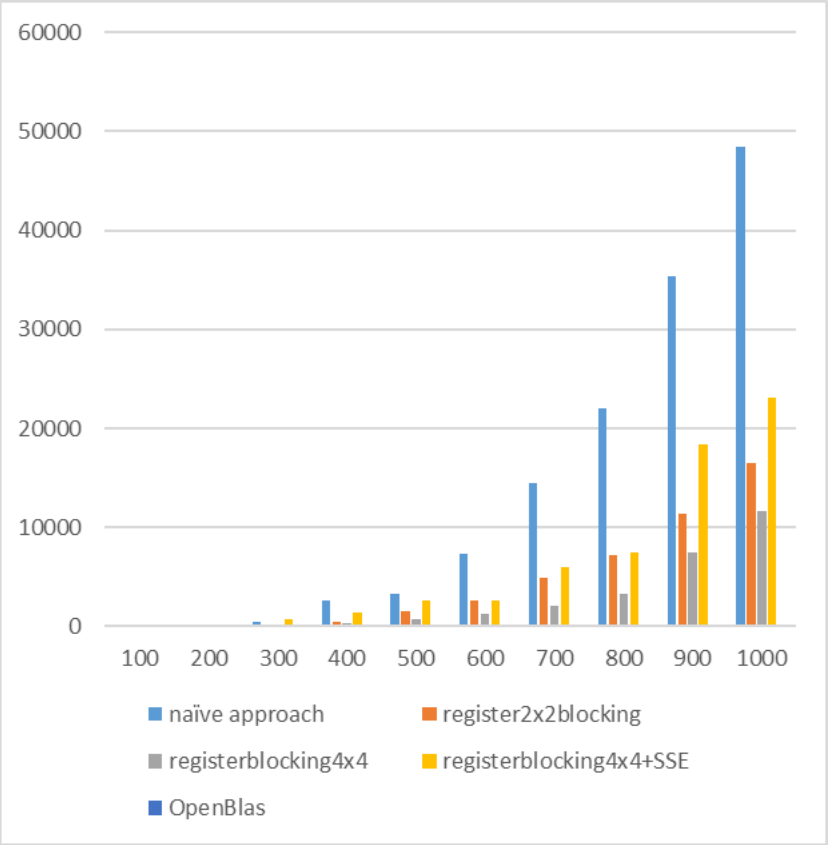
Part 3 - Result & Verification



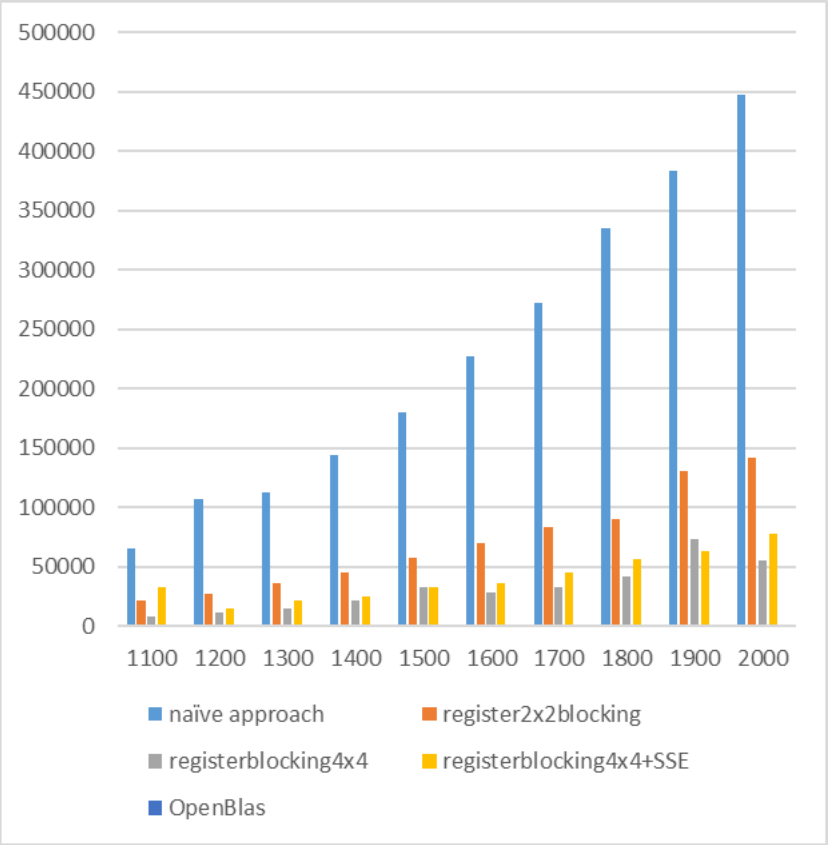
When considering the matrix size of 100x100, it is observed that OpenBLAS is the slowest among all other optimization techniques.



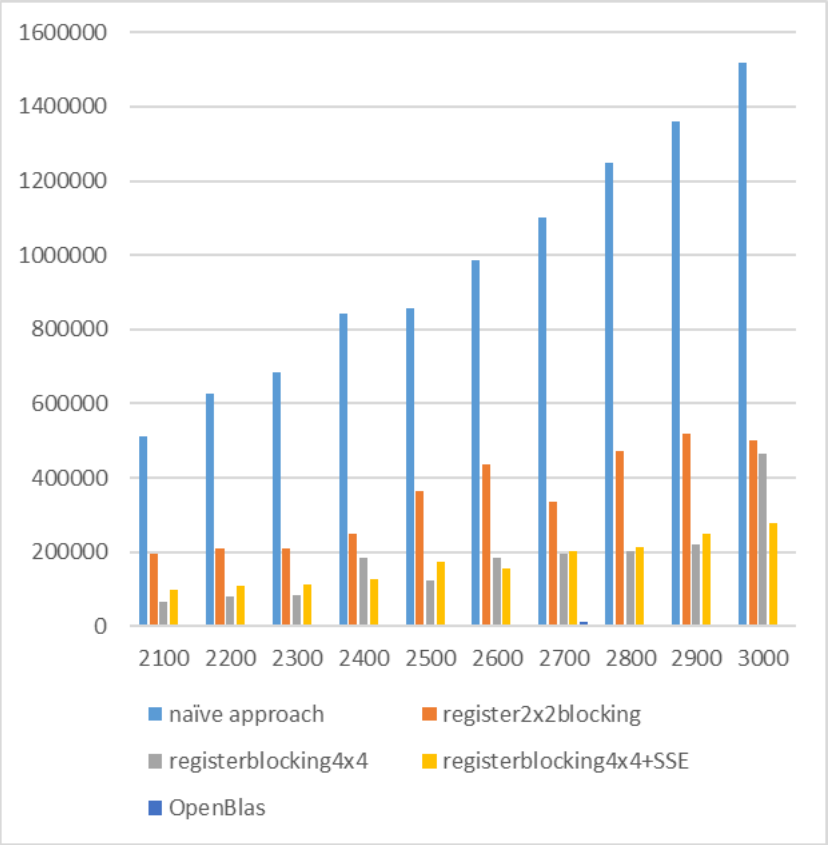
However, as the matrix size increases to 200x200, OpenBLAS shows remarkable improvement and becomes the fastest among the tested optimizations. In contrast, the slowest approach at this point is register blocking 4x4 with SSE. This significant difference between OpenBLAS and other methods becomes apparent.



Further increasing the matrix size to 1000x1000, register blocking 4x4 emerges as the fastest approach, while the naive approach becomes the slowest. The second slowest method now is register blocking 4x4 with SSE.



At a matrix size of 2000x2000, register blocking 2x2 becomes the second slowest after the naive approach, and register blocking 4x4 becomes the fastest after OpenBLAS. Surprisingly, at a size of 1900x1900, register blocking 4x4 with SSE outperforms other methods and becomes the fastest after OpenBLAS.

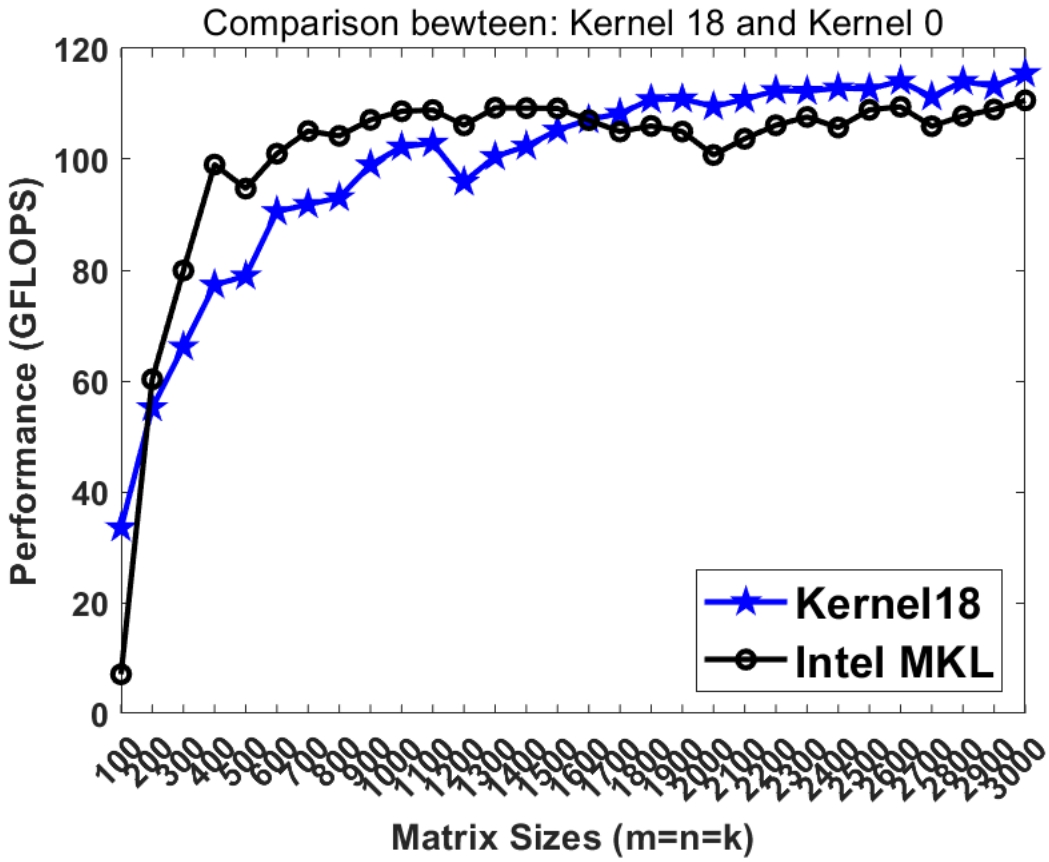


Continuing the size increment to 3000x3000, eventually register blocking 4x4 with SSE remains the fastest after OpenBLAS. The following table presents the execution times in milliseconds for the naive approach, register blocking 2x2, register blocking 4x4, register blocking 4x4 with SSE, and OpenBLAS:

	naïve approach	register2x2dgemm	registerblocking4x4	registerblocking4x4+SSE	OpenBlas
100	18	5	3	17	25
200	128	64	55	167	2
300	482	243	120	704	51
400	2635	483	305	1427	46
500	3225	1520	704	2603	25
600	7329	2567	1219	2566	59
700	14488	4860	2130	6014	112
800	22059	7244	3238	7483	103
900	35366	11327	7458	18372	159
1000	48412	16501	11607	23147	203
1100	65565	21609	8819	32508	288
1200	107373	27919	11560	15461	371
1300	113174	36417	15480	22185	457
1400	144247	45354	21931	25729	571
1500	180216	57265	33379	32694	719
1600	227611	69964	28318	36441	869
1700	271884	84102	33272	45301	1182
1800	335035	90085	42351	56844	1222

1900	382998	130305	72926	63553	1474
2000	447568	141555	55644	77837	1694
2100	512460	196524	65013	97787	2133
2200	625997	208768	81823	110986	2239
2300	685785	208472	82836	112251	2603
2400	843356	248438	184855	128453	6716
2500	858651	364757	124088	172934	3632
2600	986992	434816	186348	156482	4016
2700	1100015	337004	194143	201771	11124
2800	1249652	473640	204171	214939	4763
2900	1361230	520099	221271	248565	5483
3000	1516463	502137	464501	277106	5963

The optimization performed by Yujiazhai using AVX512 result compare to Intel MKL that is as fast as OpenBlas.



The comparison was conducted based on the GFLOPS (GigaFLOPS) metric, which measures the number of billions of floating-point operations per second. As we can see, the Kernel18 optimization developed by Yujiazhai demonstrates performance that is nearly on par with Intel MKL. This indicates that Yujiazhai's optimization achieves comparable speed and efficiency to the industry-leading Intel MKL library.

Lastly, I would like to mention another optimization technique that I haven't discussed yet. It involves performing the initial DGEMM operation once and then subsequently performing a second DGEMM operation using any kind of optimization. This approach can result in faster execution times.

The reason behind this is that during the first DGEMM operation, the matrix data is loaded into the cache. As a result, when the second DGEMM operation is performed, the required matrix data is already present in the cache, leading to faster access and computation.

By leveraging the preloaded data in the cache, the second DGEMM operation can benefit from improved cache utilization and reduced memory access latency, resulting in overall faster execution.

Part 4 - Difficulties & Solutions

One of the challenges I encountered during this project was that many cache optimization techniques relied on AVX2 and AVX512 instructions, which unfortunately were not supported on my laptop. As a result, I had to adapt and utilize SSE (Streaming SIMD Extensions) intrinsics instead, which provided a more limited set of vectorization capabilities.

Furthermore, since DGEMM (Double-precision General Matrix Multiplication) was a new concept for me, I needed to invest time in learning and understanding its intricacies before I could begin implementing the optimizations. This required studying the algorithm, exploring existing implementations, and familiarizing myself with the underlying principles and techniques involved.

By overcoming these obstacles, I was able to devise alternative strategies using SSE intrinsics and apply them to optimize the cache utilization of the matrix multiplication process. While it was a different approach from the AVX2 and AVX512 optimizations, it allowed me to maximize the performance potential on my laptop given its limitations.

Part 5 - References


```
* https://github.com/flame/how-to-optimize-gemm
* http://apfel.mathematik.uni-ulm.de/~lehn/sghpc/gemm/index.html
* https://github.com/wjc404/GEMM_AVX512F
* https://github.com/xianyi/OpenBLAS/tree/develop/kernel/x86_64
* https://github.com/yzhaiustc/Optimizing-DGEMM-on-Intel-CPUs-with-AVX512F
```