Non-square Matrix-Matrix product kernel

Small Scale Parallel Programming

Paul Fontanges; Stu No. : 461720

Prepared for 3 March , 2025

Paul Fontanges
Student Number : 461720

Abstract

Matrix multiplication is a fundamental operation in many scientific and engineering applications, requiring efficient computational techniques to handle large-scale data. This report explores the parallelization of a non-square matrix multiplication problem using OpenMP for multi-threaded CPU execution and CUDA for GPU acceleration. The study aims to analyze the performance improvements obtained through these parallelization techniques compared to a sequential implementation. The experiments were conducted on the Crescent2 HPC cluster, evaluating different problem sizes with varying values of n and k. Performance is measured in terms of execution time and floating-point operations per second (FLOPS). The results highlight the advantages of GPU-based computation, particularly when leveraging shared memory optimizations. This work provides insights into optimizing matrix multiplication for high-performance computing environments.

Paul Fontanges
Student Number : 461720

Contents

List of Figures

1 Introduction

Multiplication of large matrices is a fundamental operation in scientific computing, machine learning, and high-performance computing (HPC). In this project, we focus on the parallel implementation of a non-square matrix multiplication problem: computing $C = A \times B$, where A is of size $(n \times k)$, B is of size $(k \times n)$, and C is a square matrix of size $(n \times n)$. The specific constraint where $k \ll n$ presents an interesting challenge in optimizing memory access patterns and computational efficiency.

To accelerate matrix multiplication, we implement and compare different parallel approaches:

- **OpenMP (CPU)**
- **CUDA (GPU)**

The main objectives of this study are:

1. Develop and analyze the performance of sequential, OpenMP, and CUDA implementations.
2. Investigate the impact of different optimization techniques, such as loop ordering and shared memory usage.

The experiments are carried out on the Crescent2 HPC cluster, using various sizes problem ranging from $n = 4096$ to $n = 20480$ and k values from 32 to 128. Performance is evaluated using execution time and floating-point operations per second (FLOPS).

This report is structured as follows: Section 2 describes the methodology and the design choices. Section 3 details the implementation of OpenMP and CUDA kernels. Section 4 presents the experimental results and provides an in-depth analysis of performance. Finally, Section 5 summarizes the findings and suggests potential improvements.

2 Methodology

2.1 Experimental Design

This study focuses on the parallelization of matrix-matrix multiplication for a nonsquare problem, where $C = A \times B$, with A of size $(n \times k)$, B of size $(k \times n)$, and C being a square

matrix of size $(n \times n)$. The small value of k compared to n introduces optimization challenges related to memory access patterns and workload distribution.

To analyze the performance of different parallelization strategies, three implementations are considered:

- **Sequential Implementation**: A baseline CPU version of matrix multiplication.
- **OpenMP Parallelization**: A multi-threaded implementation leveraging shared-memory parallelism.
- **CUDA GPU Parallelization**: A parallel implementation using GPUs.

Performance evaluation is based on execution time and floating point operations per second (FLOPS), tested in the Crescent2 HPC cluster in different problem sizes ($n = 4096$ to $n = 20480$, $k = 32$ to $k = 128$).

2.2 Sequential Matrix Multiplication

The sequential algorithm follows the standard triple-loop approach:

1. Iterate over the rows i of A.
2. Iterate on the columns j of B.
3. Compute the dot product of the i-th row of A with the j-th column of B using a loop over k.

This implementation serves as a reference for speedup calculations.

2.3 OpenMP Implementation

2.3.1 Parallelization Strategy

To accelerate the calculation of the matrix product $C = A \times B$, we utilize OpenMP to distribute the workload across multiple CPU threads. The key parallelization strategy involves:

- **Parallel initialization of** C: The matrix C is initialized to zero.

- **Parallel matrix multiplication**: The core computation is parallelized over the outer loop (i), ensuring that each thread processes different rows of C.
- **Loop ordering**: Find the best loop ordering to optimize memory access patterns.

2.3.2 Expected Performance and Optimization Considerations

- **Thread Scalability**: The speed-up is expected to scale with the number of CPU threads.
- **Memory Access Patterns**: The time elapsed is expected to change with respect to the loop ordering.
- **Synchronization Overhead**: Since each thread writes to different rows in C, synchronization overhead is minimal.

2.3.3 Graphical Representation

Figure 1 illustrates the thread distribution in the OpenMP implementation. Each thread computes a subset of rows in C, ensuring load balancing between available CPU cores.

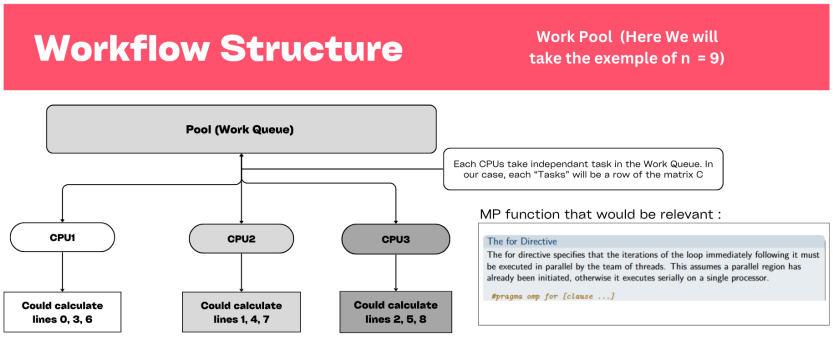

Figure 1: Thread distribution in OpenMP matrix multiplication

2.4 CUDA Implementation

2.4.1 Parallelization Strategy

To accelerate matrix-matrix multiplication on GPU, we leverage CUDA's massively parallel execution model. Each thread computes an element of the output matrix C. The core parallelization strategy is based on:

- **Thread mapping**: Each thread is responsible for computing a single element $C[i, j]$.
- **Memory optimization**: Use of `__shared__` memory to reduce global memory accesses.
- **Tiled computation (GEMM approach)**: Blocks of A and B are loaded into shared memory in tiles of size $TILE\_SIZE \times TILE\_SIZE$.

2.4.2 Expected Performance and Optimization Considerations

- **Memory Coalescing**: The use of shared memory reduces global memory accesses, improving performance.
- **Tiling for Optimization**: Breaking the matrix into `TILE_SIZE` blocks reduces memory latency.
- **Thread Synchronization**: The `__syncthreads()` directive ensures that all threads complete tile computation before proceeding.
- **Scalability**: The performance scales well with the matrix size and available GPU compute power.

2.4.3 Graphical Representation

Figure 2 illustrates the tiled GEMM approach in CUDA, where submatrices are loaded into shared memory for computation before writing results back to global memory.

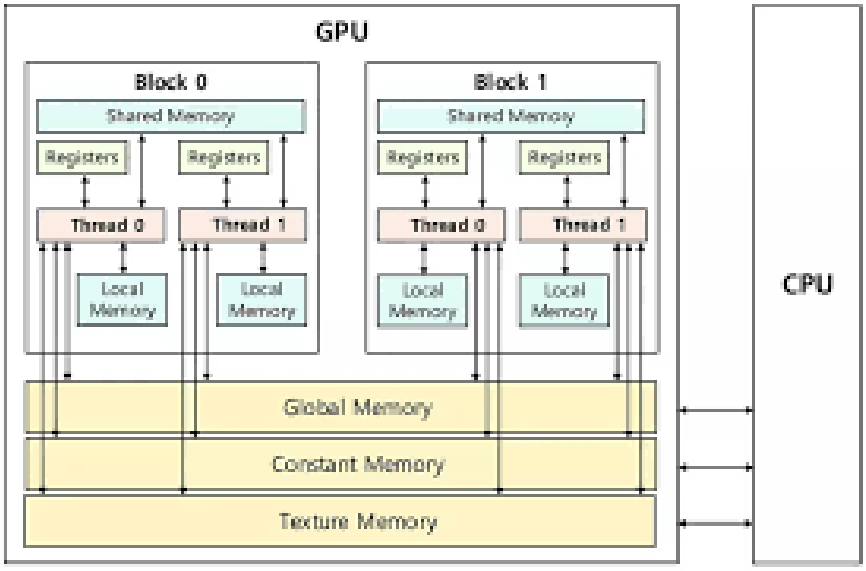

Figure 2: Tiled GEMM Approach in CUDA

3 Implementation

3.1 Implementation choice

To implement this methodology, we have chosen the C language. Regarding the precision of our variables, we will prefer `float` to `double`, as this will enable faster execution. As for the choice of loop order for the openMP script, we'll take the IMJ order. This choice will also be justified in the analysis section.

3.1.1 Data

Now for the data, we will choose these pairs of values `(n,k)`, to have a wide range of results to analyze:

- `(4096, 32)`
- `(8192, 48)`
- `(12288, 64)`

- `(16384, 96)`
- `(20480,128)`

3.2 OpenMP Implementation

The OpenMP-optimized implementation is structured as follows:

1. **Memory Allocation**: The matrices A, B, and C are dynamically allocated using *malloc*.
2. **Matrix Initialization**: The matrices A and B are populated with random values.
3. **Computation Phase**:
   (a) $C[i][j]$ is initialized to zero in parallel.
   (b) Each thread computes a subset of rows in C.
   (c) The innermost loop iterates over k to accumulate partial sums.
4. **Performance Measurement**: Execution time is measured using $omp_get_wtime()$, and performance is evaluated in GFLOPS.

3.2.1 OpenMP Code Implementation

The OpenMP implementation follows the structure below:

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        C[i][j] = 0.0;
    }
}

#pragma omp parallel for
for (int i = 0; i < n; i++) {
    for (int m = 0; m < k; m++) {
```

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

3.3 CUDA Implementation

The CUDA-optimized implementation follows these steps:

1. **Memory Allocation**: Allocate matrices A, B, and C on the CPU (host) and GPU (device).
2. **Memory Transfer**: Copy matrices from the host (CPU) to the device (GPU).
3. **Kernel Execution**: The matrix multiplication kernel is launched with a 2D grid of thread blocks.
4. **Performance Measurement**: Measure execution time using CUDA events and compute GFLOPS.
5. **Memory Cleanup**: Copy results back to the host and free GPU memory.

3.3.1 CUDA Code Implementation

The CUDA kernel implementing the tiled GEMM algorithm is structured as follows:

```
__global__ void matrix_multiply_gemm(float *A, float *B, float *C, int n, int k) {
    __shared__ float tile_A[TILE_SIZE][TILE_SIZE];
    __shared__ float tile_B[TILE_SIZE][TILE_SIZE];

    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;
    float sum = 0.0f;
```

```
    for (int m = 0; m < (k + TILE_SIZE - 1) / TILE_SIZE; m++) {
        if (row < n && (m * TILE_SIZE + threadIdx.x) < k)
            tile_A[threadIdx.y][threadIdx.x] = A[row * k + m * TILE_SIZE + threadIdx.x];
        else
            tile_A[threadIdx.y][threadIdx.x] = 0.0f;

        if (col < n && (m * TILE_SIZE + threadIdx.y) < k)
            tile_B[threadIdx.y][threadIdx.x] = B[(m * TILE_SIZE + threadIdx.y) * n + col
        else
            tile_B[threadIdx.y][threadIdx.x] = 0.0f;

        __syncthreads();

        for (int i = 0; i < TILE_SIZE; i++) {
            sum += tile_A[threadIdx.y][i] * tile_B[i][threadIdx.x];
        }

        __syncthreads();
    }

    if (row < n && col < n)
        C[row * n + col] = sum;
}
```

4 Analysis

4.1 Precision choice and loop choice

Here we will take the results of one code implemented with `float` and another code implemented with `double`. We will then choose our variable precision based on the results of this test.

Here, Figure 3 illustrates that a simple precision (`float`) for variables gives better per-

formance (as expected).

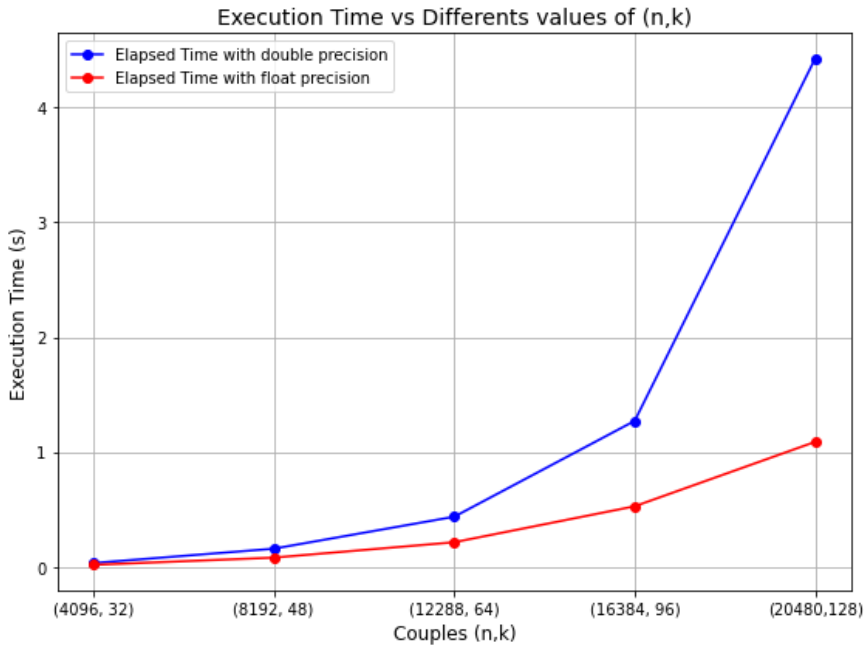

Figure 3: performance of `float` variables versus `double` variables

To choose the order of the loop for the `openMP` script, we will proceed as follows. We will test all possible loop orders for the same data pair and take the one with the best results.

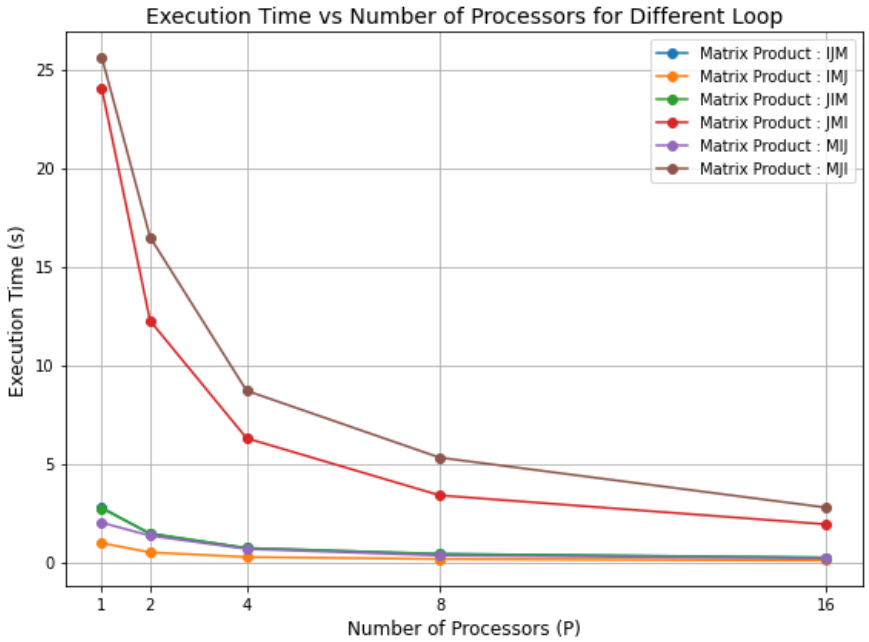

Figure 4: Loop order performances

This is to be expected `imj` have better results than the other loops, as this order corresponds to the "ordering" of the memory.

4.2 Correctness test

For this test, we will use matrices A, B and C, initialized as follows:

$$A = \begin{bmatrix} 3 & 6 \\ 7 & 5 \\ 3 & 5 \\ 6 & 2 \end{bmatrix}$$

$$B = \begin{bmatrix} 9 & 1 & 2 & 7 \\ 0 & 9 & 3 & 6 \end{bmatrix}$$

The expected result will then be :

$$C = \begin{bmatrix} 27 & 57 & 24 & 57 \\ 63 & 52 & 29 & 79 \\ 27 & 48 & 21 & 51 \\ 54 & 24 & 18 & 54 \end{bmatrix}$$

Both (OpenMP and CUDA scripts) pass the test. For the remainder, we will not display the results due to lack of space on the command prompt (the `print_matrix` functions and their initialization for the test will be kept in the script but commented out).

4.3 Performance testing

Now let's compare our results. First, we will have two CUDA scripts (one with shared memory and one without). We will then evaluate these scripts on two criteria:

1. Execution Time
2. GFlops
3. impact of k values (with n fixed)

4.3.1 Execution Time Performance

In the figure 5 below, we can see that both types of script take at most 1 second for the largest matrix products. This is much less than for the serial implementation, which shows that the code has been well parallelized. Furthermore, we can see that implementation via CUDA is much faster than with openMP (around 40% without shared memory and 70% with). For example, results concerning the data pair `(20480,128)` are :

- `0.034831` for the CUDA (with shared memory)
- `0.061906` for the CUDA (without shared memory) script
- `1.088262` for the openMP script

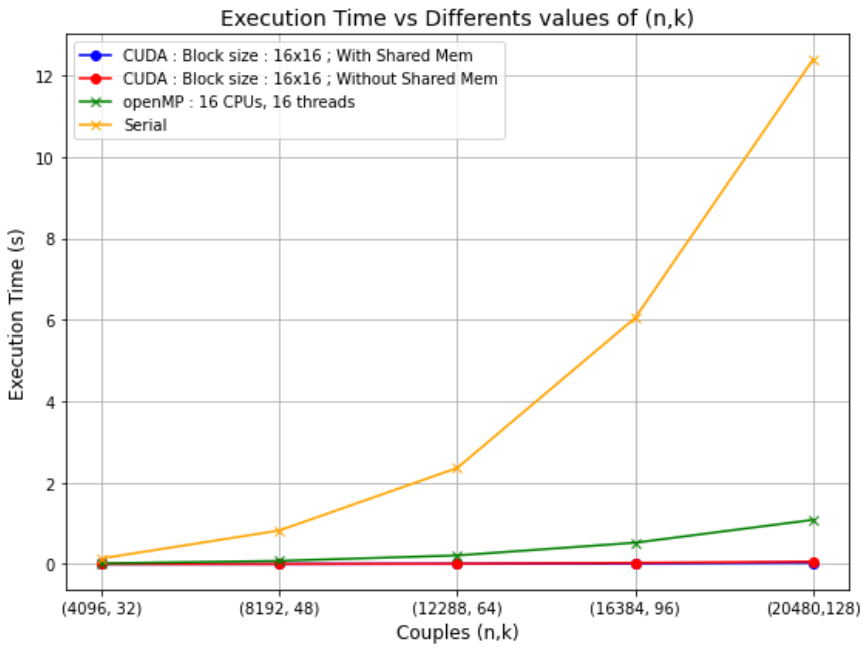

Figure 5: Execution Time of CUDA and openMP Script for different data pair `(n,k)`

4.3.2 GFlops Performance

Now concerning `GFlops` we can see in figure 6, below that the openMP implementation, although it still outperforms the serial version in terms of GFlops, and far outperforms the CUDA versions. Note that the CUDA version with shared memory performs significantly better than the one without shared memory.

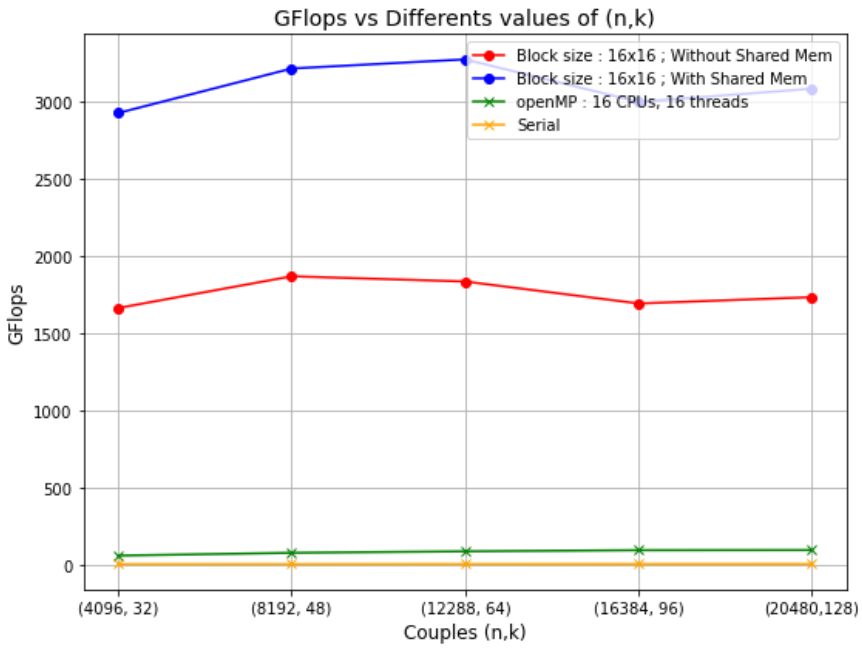

Figure 6: GFlops performance of CUDA and openMP Script for different data pair `(n,k)`

4.3.3 k values impact

Now we will take a fixed value of n (here $n = 20480$]) and test our programs for several values of k $(32, 48, 64, 96, 128)$ to see if this variable has an impact on GFlops. Here we can see in this figure 7 that as k increases, so does GFlops, but not for the CUDA script with shared memories (which decrease each time the k increases) which is interesting.

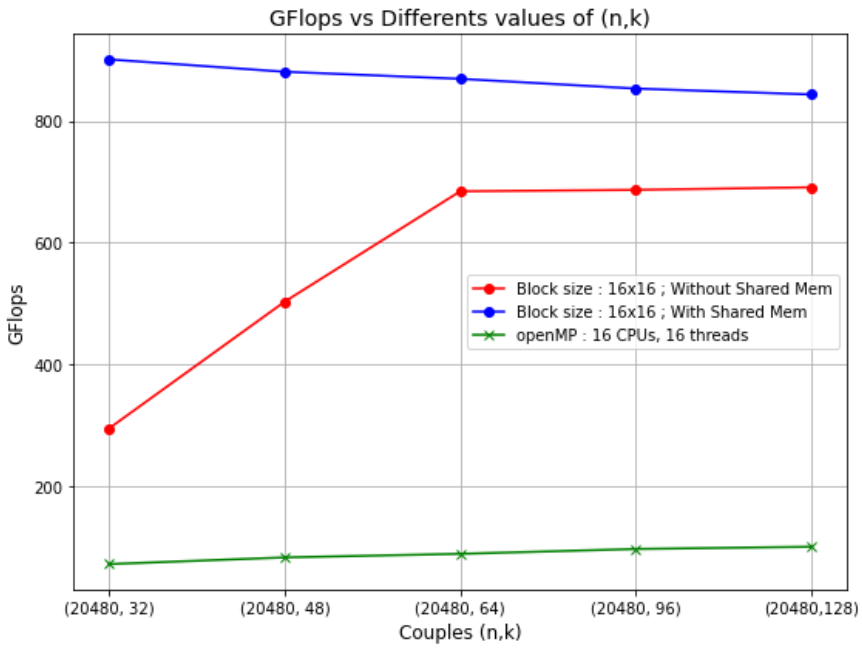

Figure 7: GFlops performance regarding different values of k

5 Conclusions

This report investigated the parallelization of a non-square matrix multiplication problem using OpenMP and CUDA implementations. The results demonstrate that GPU-based execution with CUDA significantly outperforms CPU-based parallelization with OpenMP, particularly when employing shared memory optimizations.

Key findings include:

- OpenMP provides a substantial speedup over the sequential version but is limited by the number of available CPU cores.
- CUDA implementation achieves much higher GFLOPS, benefiting from massive parallelism and optimized memory usage.
- The use of shared memory in CUDA further enhances performance by reducing costly global memory accesses.

- The choice of loop ordering and memory layout significantly impacts execution time, emphasizing the need for optimization strategies.

Future work could explore additional optimizations, such as using mixed precision arithmetic, auto-tuning kernel parameters, and utilizing multiple GPUs for distributed matrix computation. The insights gained from this study contribute to the efficient implementation of matrix operations in high-performance computing environments.

6 Reference

1. Joel C. Adams, Richard A. Brown, Suzanne J. Matthews, and Elizabeth Shoop *Parallel Computing for Beginners - Matrix Multiplication.* Available `https://www.learnpdc.org/PDCBeginners/5-applications/matrix-multiply.html`.

2. Vipul Vaibhaw *Matrix Multiplication: Optimizing the code from 6 hours to 1 sec.* Available `https://vaibhaw-vipul.medium.com/matrix-multiplication-optimizing-the-code-from`

3. Simon Boehm *How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog.* Available `https://siboehm.com/articles/22/CUDA-MMM`

4. Dhanush *Mastering CUDA Matrix Multiplication: An Introduction to Shared Memory, Tile Memory Coalescing, and Bank Conflicts.* Available `https://medium.com/@dhanushg295/mastering-cuda-matrix-multiplication-an-introduction-to-shared-memory-`

A openMP Code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

// matrix allocation
float** allocate_matrix(int rows, int cols) {
    float** matrix = (float**)malloc(rows * sizeof(float*));
    for (int i = 0; i < rows; i++) {
        matrix[i] = (float*)malloc(cols * sizeof(float));
    }
    return matrix;
}

// matrix initialisation with fixed values (test)
void initialize_fixed_matrix(float** matrix, int rows, int cols, float fixed_values[rows
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = fixed_values[i][j];
        }
    }
}

// matrix initialisation with random values
void initialize_matrix(float** matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = (float)(rand() % 10);
        }
    }
}
```

```
void print_matrix(const char* name, float** matrix, int rows, int cols) {
    printf("%s:\n", name);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%6.2f ", matrix[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

void matrix_multiply_omp(float** A, float** B, float** C, int n, int k) {
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = 0.0f;
        }
    }

    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
      for (int m = 0; m < k; m++) {
      for (int j = 0; j < n; j++) {
          C[i][j] += A[i][m] * B[m][j];
      }
      }
     }
}

void free_matrix(float** matrix, int rows) {
    for (int i = 0; i < rows; i++) {
```

```
        free(matrix[i]);
    }
    free(matrix);
}

int main(int argc, char *argv[]) {

    // read (n and k) arguments
    if (argc < 3) {
        printf("Use : %s <n> <k>\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]);
    int k = atoi(argv[2]);

    if (n <= 0 || k <= 0) {
        printf("Error: n and k must be positive integer.\n");
        return 1;
    }

    // Test Matrix
    //float test_A[4][2] = {{3, 6}, {7, 5}, {3, 5}, {6, 2}};
    //float test_B[2][4] = {{9, 1, 2, 7}, {0, 9, 3, 6}};

    // matrix allocation
    float** A = allocate_matrix(n, k);
    float** B = allocate_matrix(k, n);
    float** C = allocate_matrix(n, n);

    // matrix initialisation with fixed values
    //initialize_fixed_matrix(A, n, k, test_A);
```

```
    //initialize_fixed_matrix(B, k, n, test_B);

    // matrix initialisation with random values
    initialize_matrix(A, n, k);
    initialize_matrix(B, k, n);

    double start, end;
    start = omp_get_wtime();

    //print_matrix("Matrix A", A, n, k);
    //print_matrix("Matrix B", B, k, n);

    matrix_multiply_omp(A, B, C, n, k);

    end = omp_get_wtime();

    double elapsed_time = end - start;

    double flops = (2.0 * k * n * n) / elapsed_time;

    printf("Matrix product with n=%d and k=%d\n", n, k);
    printf("Execution Time : %.6f secondes\n", elapsed_time);
    printf("Performance : %.2f GFLOPS\n", flops / 1e9);

    //print_matrix("Matrice C (Résultat)", C, n, n);

    free_matrix(A, n);
    free_matrix(B, k);
    free_matrix(C, n);

    return 0;
}
```

B CUDA Code

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

#define TILE_SIZE 16  // optimized block size

// Kernel with shared memory
__global__ void matrix_multiply_gemm(float *A, float *B, float *C, int n, int k) {
    // Matrices en mémoire partagée
    __shared__ float tile_A[TILE_SIZE][TILE_SIZE];
    __shared__ float tile_B[TILE_SIZE][TILE_SIZE];

    int row = blockIdx.y * TILE_SIZE + threadIdx.y;  //  i
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;  //  j
    float sum = 0.0f;  // Stockage du résultat temporaire

    // Load matrix by blocs
    for (int m = 0; m < (k + TILE_SIZE - 1) / TILE_SIZE; m++) {
        // load  submatrix in the shared memory
        if (row < n && (m * TILE_SIZE + threadIdx.x) < k)
            tile_A[threadIdx.y][threadIdx.x] = A[row * k + m * TILE_SIZE + threadIdx.x];
        else
            tile_A[threadIdx.y][threadIdx.x] = 0.0f;

        if (col < n && (m * TILE_SIZE + threadIdx.y) < k)
            tile_B[threadIdx.y][threadIdx.x] = B[(m * TILE_SIZE + threadIdx.y) * n + col
        else
            tile_B[threadIdx.y][threadIdx.x] = 0.0f;
```

```
        __syncthreads();

        // product of submatrix
        for (int i = 0; i < TILE_SIZE; i++) {
            sum += tile_A[threadIdx.y][i] * tile_B[i][threadIdx.x];
        }

        __syncthreads();
    }

    // Final result write
    if (row < n && col < n)
        C[row * n + col] = sum;
}

void print_matrix(const char* name, float* matrix, int rows, int cols) {
    printf("%s:\n", name);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%6.2f ", matrix[i * cols + j]);
        }
        printf("\n");
    }
    printf("\n");
}

int main(int argc, char *argv[]) {
    // read (n and k) arguments
    if (argc < 3) {
        printf("Use : %s <n> <k>\n", argv[0]);
        return 1;
```

```
    }

    int n = atoi(argv[1]);
    int k = atoi(argv[2]);

    if (n <= 0 || k <= 0) {
        printf("Error: n and k must be positive integer.\n");
        return 1;
    }

    // matrix allocation on CPU
    float *h_A = (float*)malloc(n * k * sizeof(float));
    float *h_B = (float*)malloc(k * n * sizeof(float));
    float *h_C = (float*)malloc(n * n * sizeof(float));

    // matrix initialisation with fixed values (test)
    //float test_A[4][2] = {{3, 6}, {7, 5}, {3, 5}, {6, 2}};
    //float test_B[2][4] = {{9, 1, 2, 7}, {0, 9, 3, 6}};

 //for (int i = 0; i < n; i++)
    //    for (int j = 0; j < k; j++)
    //        h_A[i * k + j] = test_A[i][j];

    //for (int i = 0; i < k; i++)
    //    for (int j = 0; j < n; j++)
    //        h_B[i * n + j] = test_B[i][j];

    // matrix initialisation with random values
    for (int i = 0; i < n * k; i++) h_A[i] = (float)(rand() % 10);
    for (int i = 0; i < k * n; i++) h_B[i] = (float)(rand() % 10);

    // matrix allocation on GPU
```

```
float *d_A, *d_B, *d_C;
cudaMalloc((void**)&d_A, n * k * sizeof(float));
cudaMalloc((void**)&d_B, k * n * sizeof(float));
cudaMalloc((void**)&d_C, n * n * sizeof(float));

// matrix copy CPU -> GPU
cudaMemcpy(d_A, h_A, n * k * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, k * n * sizeof(float), cudaMemcpyHostToDevice);

dim3 blockSize(TILE_SIZE, TILE_SIZE);
dim3 gridSize((n + TILE_SIZE - 1) / TILE_SIZE, (n + TILE_SIZE - 1) / TILE_SIZE);

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

matrix_multiply_gemm<<<gridSize, blockSize>>>(d_A, d_B, d_C, n, k);

cudaEventRecord(stop);
cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);

cudaMemcpy(h_C, d_C, n * n * sizeof(float), cudaMemcpyDeviceToHost);

//print_matrix("Matrix A", h_A, n, k);
//print_matrix("Matrix B", h_B, k, n);
//print_matrix("Matrix C (result)", h_C, n, n);

printf("Matrix product with n=%d and k=%d\n", n, k);
printf("Execution Time on GPU : %.6f secondes\n", milliseconds / 1000.0);
```

```
    double flops = (2.0 * k * n * n) / (milliseconds / 1000.0);
    printf("Performance GPU : %.2f GFLOPS\n", flops / 1e9);

    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    free(h_A); free(h_B); free(h_C);

    return 0;
}
```