# GPU Programming

Mohammad Marufur Rahman, and Risal Shahriar Shefin

December 12, 2024

## Contents

## 1 Introduction

GPUs are specialized parallel hardware for floating point operations. They are basically co-processors for traditional CPUs. CPU controls the workflow but delegates highly parallel tasks to the GPU. A CPU is often called the ”brain” of a computer that can work on a variety of tasks. It consists of a few cores and is latency-optimized (time taken to transfer data between different components of the system). It is designed to maximize the performance of a single task within a job. On the other hand, GPU is a specialized type of microprocessor, optimized for throughput (the amount of data processed per unit time). It uses thousands of smaller and more efficient cores for a massively parallel architecture aimed at handling multiple functions at the same time.

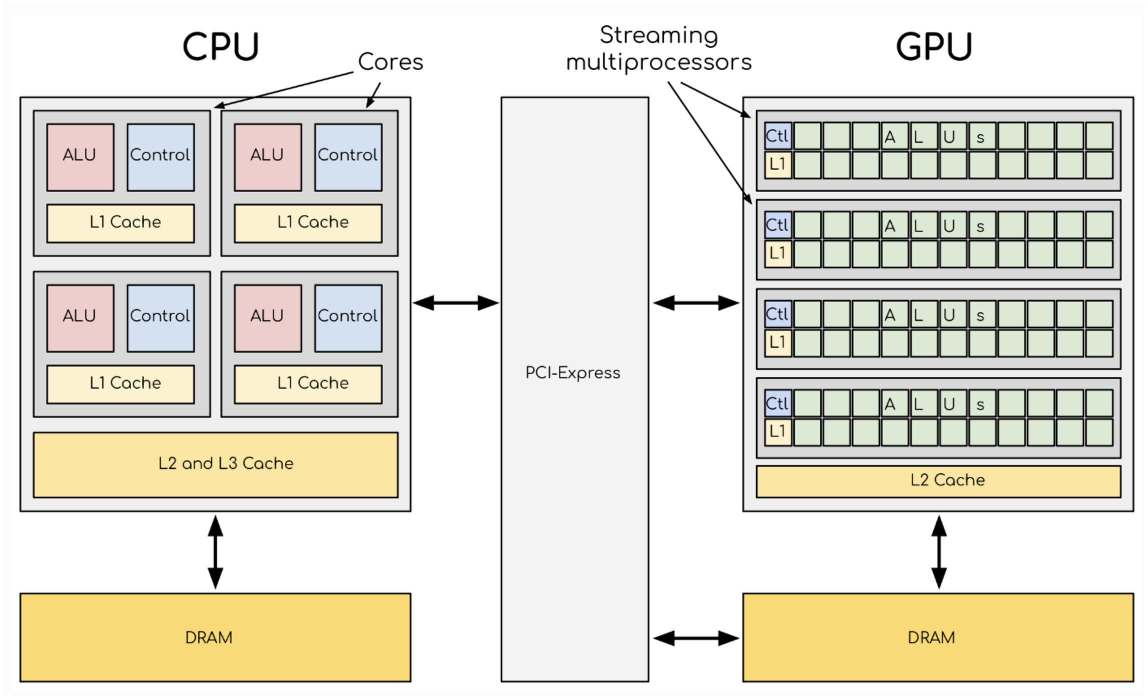

Figure 1: Comparison between CPU and GPU architecture. [2]

CPU consists of a few powerful cores each having its own ALU for performing computation and Control Unit for handling instructions. The GPU consists of hundreds of small cores grouped into Streaming Multiprocessors (SMs), which can execute threads in parallel. Each core in an SM is simpler and optimized for high-throughput computation to a CPU core. Each SM can handle multiple warps (groups of 32 threads in NVIDIA GPUs) at once, and the GPU schedules warps rather than individual threads. GPUs use the SIMT model, where each thread in a warp executes the same instruction simultaneously on different data. To hide memory latency, GPUs leverage thread-level parallelism by switching to another warp while waiting for data. It contains specialized control units responsible for warp scheduling and context switching. Each thread has its own private registers. Memory within an SM that can be shared among threads in a block. Main memory is accessible to all threads, but it is slower compared to registers or shared memory. A block is a group of threads that can communicate and synchronize with each other through shared memory. Each block is assigned to one SM for execution, but it cannot span across multiple SMs. Multiple blocks can be assigned to a single SM, and the number of blocks per SM depends on the available resources.

CUDA (Compute Unified Device Architecture) is NVIDIA's parallel computing platform and application programming interface (API) that enables developers to use GPUs for general-purpose processing tasks beyond traditional graphics rendering. Threads are the smallest unit of execution in CUDA. The collection of threads is called a block. Threads can be organized in 1D, 2D, and

3D layouts within a block. The collection of blocks is called a grid. Blocks can also be organized in 1D, 2D, and 3D layout within a grid. All the blocks in a grid execute the same kernel (CUDA function).

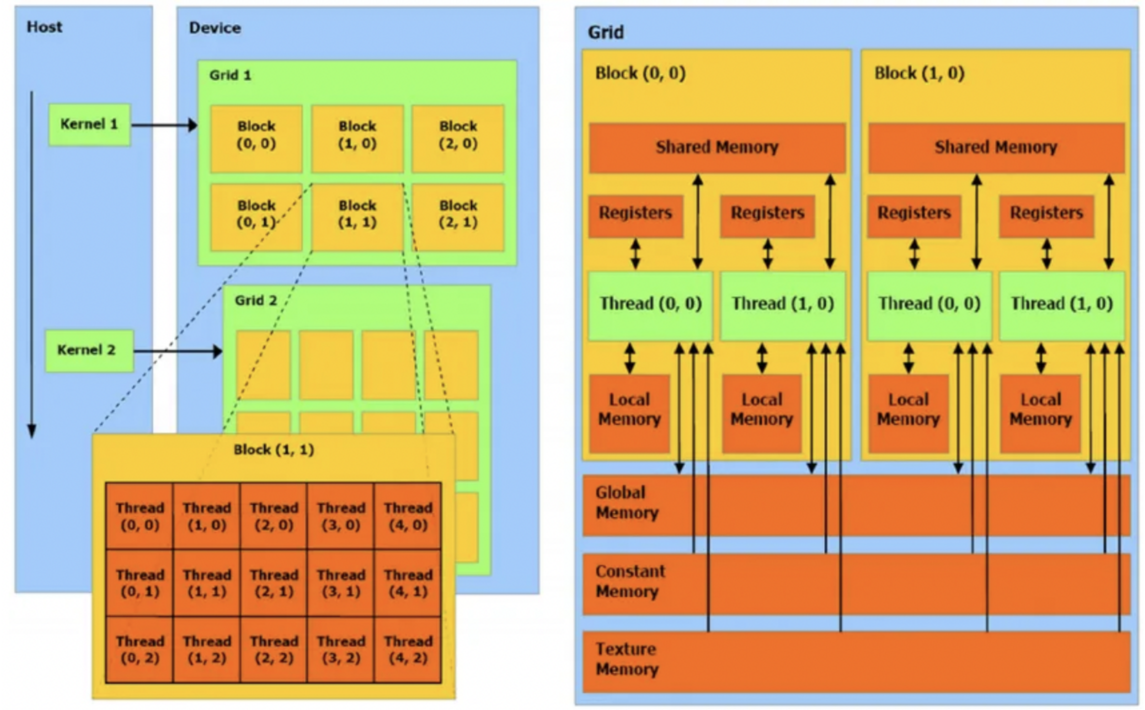

Figure 2: Nvidia CUDA enabled GPU architecture. [3]

CUDA has different types of memory. Registers are the private memory of threads and the fastest memory in the GPU. Each thread has local memory that resides in the global memory. Which is as fast as the global memory. Each block has shared memory that is accessible to all the threads in that block. Shared memory is faster than the global memory. Global memory is accessible to all threads across all blocks. It is used for sharing data across the grids and storing input, intermediate, and output data. There is another type of memory named constant memory. It is read-only to the threads and writable to the host.

## 2 Experimentation

For experiments, we used GPU from the DEAC cluster of Wake Forest University. For programming reference, NVIDIA documentation of CUDA programming was followed [4].

### 2.1 DEAC GPU Information

1. Number of Streaming Multiprocessors (SMs): 80
2. Maximum threads per SM: 2048
3. Maximum shared memory (bytes) per SM: 98304 = 96KB
4. Maximum shared memory (bytes) per block: 49152 = 48KB
5. Maximum total number of threads per block: 1024
6. Maximum block dimensions (number of threads):

   (a) x-dimension: 1024
   (b) y-dimension: 1024
   (c) z-dimension: 1024

7. Maximum Grid Dimensions (number of blocks):
   (a) x-dimension: 2147483647
   (b) y-dimension: 65535
   (c) z-dimension: 65535

## 2.2 Matrix Multiplication

The following setup is used for all of our experiments:

- We've set the thread count per thread block to 1024 which is the max limit. The reason is to utilize parallelism as much as possible.
- Squared matrices are used in this experiment with dimension $N \times N$, where $N \geq 1$.
- Blocked matrix multiplication is used.
- The matrices are divided into $b \times b$ blocks, where $b = 32$.
- Each thread is assigned to only one cell of the output matrix.
- All matrices are stored in 1D matrices following column-major order.

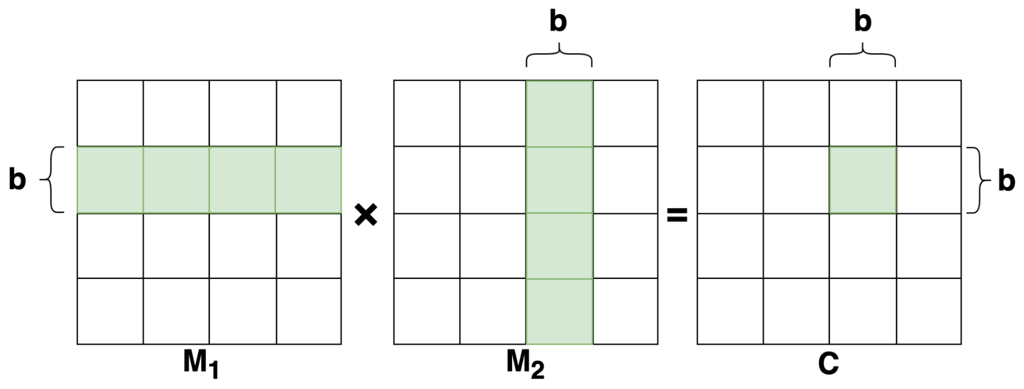

Figure 3: Blocked Matrix Multiplication

### 2.2.1 Approach 1: Naive

In this approach, the threads are organized in a 2D layout inside a thread block. So, both thread blocks and matrix blocks have the same dimension: $b \times b$.

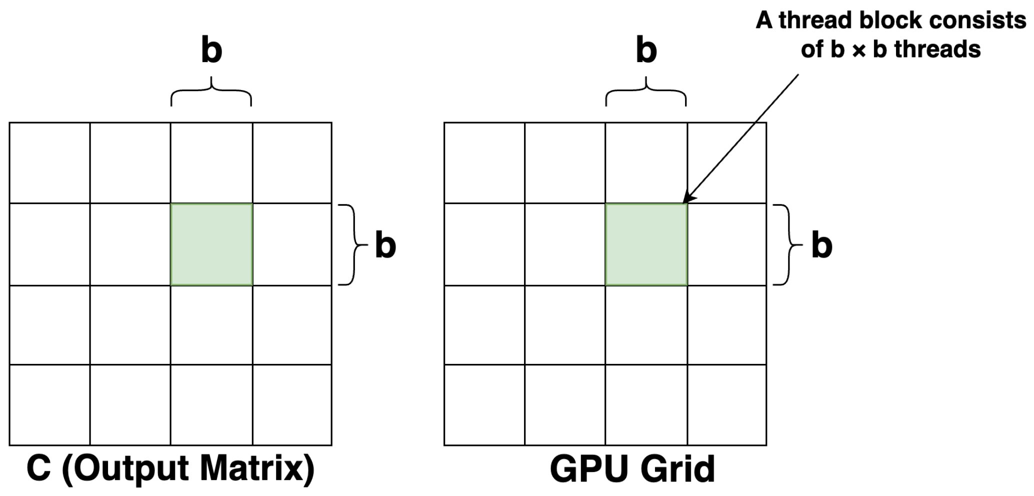

Figure 4: Output Matrix & Thread Block

Following is the pseudocode for the GPU grid and thread block declaration in CUDA programming:

```
dim3 dimBlock(b, b);
dim3 dimGrid(N/b, N/b);
mat_mult_cuda<<<dimGrid, dimBlock>>>(m1, m2, c, N);
```

Now, each thread $t_{ij}$ will read a row $M_{1_i}$, a column $M_{2_j}$ and perform necessary computations for the corresponding cell $c_{ij}$. So, the time complexity for a thread is $\mathcal{O}(N)$.

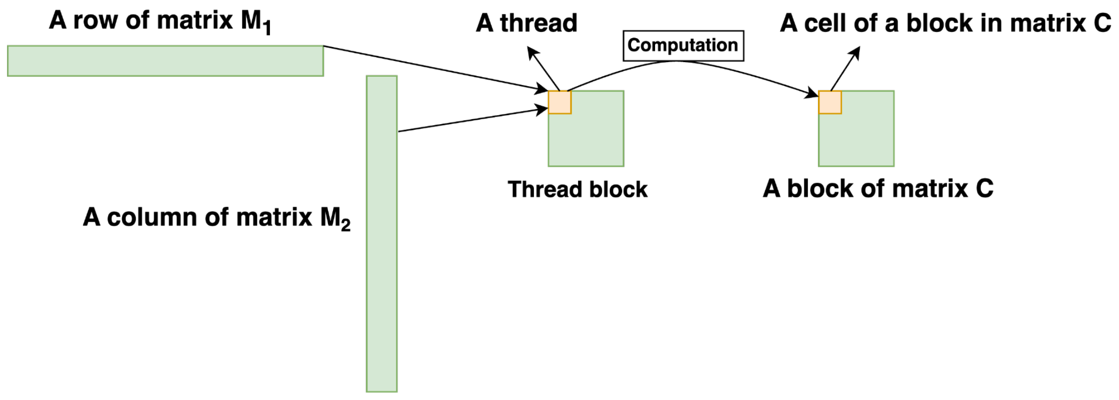

Figure 5: Computation of A Thread

Following is the pseudocode of a thread's operation in CUDA programming:

```
int row = blockIdx.y * b + threadIdx.y;
int col = blockIdx.x * b + threadIdx.x;
float sum = 0;
for (int k = 0; k < n; k++) {
    sum += m1[row, k] * m2[k, col)];
```

```
}
c[row,col] = sum;
```

---

The complete C++ CUDA program of this approach is written below:

---

```
#include <iostream>
#include <cstdio>
#include <cassert>
#include <chrono>
#include <set>
using namespace std;

__host__ __device__ inline int get_idx(const int& row, const int& col, const int& n) {
    return row + col*n;}

__global__ void mat_mult_gpu(float *ad, float *bd, float* cd, int n)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if(row >= n || col >= n) return;

    float sum = 0;
    for (int k = 0; k < n; k++) {
        sum += ad[get_idx(row, k, n)] * bd[get_idx(k, col, n)];
    }

    cd[get_idx(row, col, n)] = sum;
}

void run(int N)
{
    // declaration
    float* a = (float *)malloc(N * N * sizeof(float));
    float* b = (float *)malloc(N * N * sizeof(float));
    float* c = (float *)malloc(N * N * sizeof(float));
    float* resultFromGpu = (float *)malloc(N * N * sizeof(float));

    // value population
    srand48(42); // seed
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            a[get_idx(i, j, N)] = drand48();
            b[get_idx(i, j, N)] = drand48();
        }
    }

    // GPU call section starts
    float *dev_a, *dev_b, *dev_c;
    cudaMalloc((void**)&dev_a, N * N * sizeof(float));
    cudaMalloc((void**)&dev_b, N * N * sizeof(float));
    cudaMalloc((void**)&dev_c, N * N * sizeof(float));

    cudaMemcpy(dev_a, a, N * N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, N * N * sizeof(float), cudaMemcpyHostToDevice);

    int blockDimSize = 32;
    int gridDimSize = N/blockDimSize;
    if (N % blockDimSize) gridDimSize++;

    dim3 dimGrid(gridDimSize, gridDimSize);
    dim3 dimBlock(blockDimSize, blockDimSize);

    // Time CUDA kernel execution
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    mat_mult_gpu<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);

    // Wait for the GPU to finish before exiting
    cudaDeviceSynchronize();

    // Stop timer
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float cudaTime;
    cudaEventElapsedTime(&cudaTime, start, stop);

    printf("CUDA Execution Time: %0.6f ms\n", cudaTime);

    // Copy the result from gpu
    cudaMemcpy(resultFromGpu, dev_c, N * N * sizeof(float), cudaMemcpyDeviceToHost);

    // Free Memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
    free(a);
    free(b);
    free(c);
    free(resultFromGpu);
}

int main()
{
    int n = 1024;
    run(n);
    return 0;
}
```

### 2.2.2 Approach 2: Memory Access Utilization (Global Memory Coalescing)

The approach discussed above can be optimized by utilizing global memory access order. The GPU coalesces global memory loads and stores issued by threads of a warp into as few transactions as possible [5]. So, consecutive global memory access within a warp will be optimized. It is called global memory coalescing. If we can know beforehand which threads are most likely to be grouped in a warp, we can control their global memory access order to enable this optimization. If the threads are organized in a 2D layout within a thread block, the threads in it will be grouped into a warp according to the row-major order.

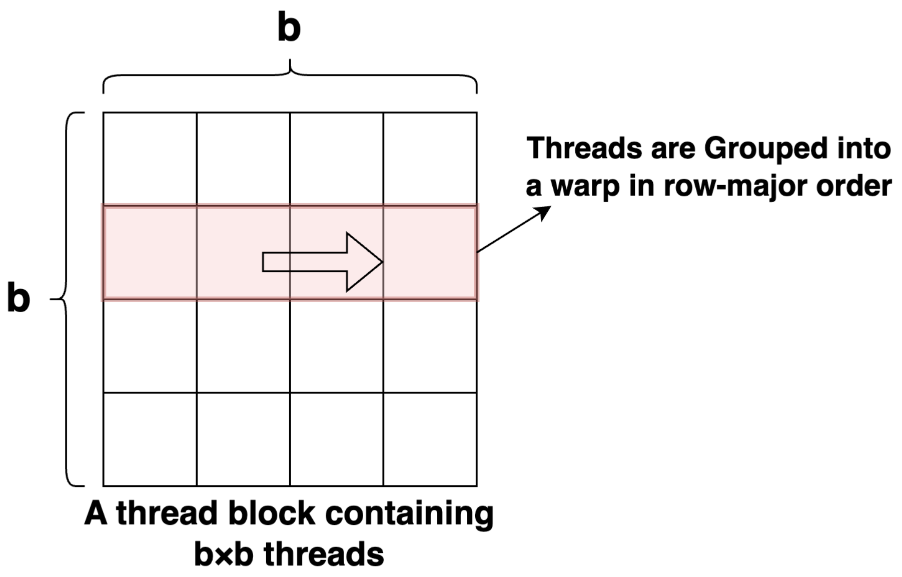
b
b
Threads are Grouped into a warp in row-major order
A thread block containing b×b threads

Figure 6: Warp Group Strategy for a 2D Thread Block

If the threads are organized in a 1D layout within a thread block, the threads are grouped into a warp based on their thread ids. That means threads having consecutive thread ids are most likely to be in the warp together.

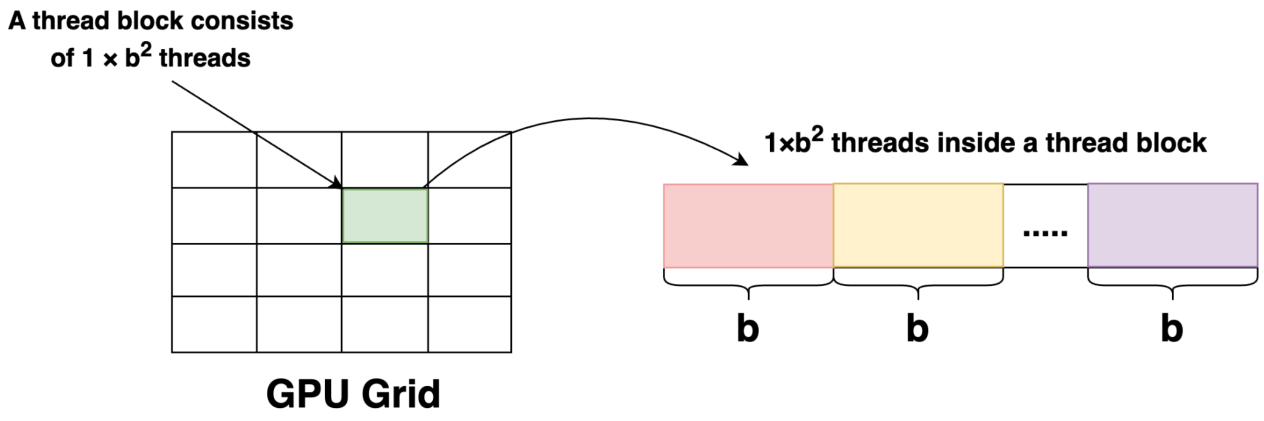
A thread block consists of 1 × b2 threads
1×b2 threads inside a thread block
.....
b
b
b
GPU Grid

Figure 7: 1D Thread Block

We preferred the 1D layout because this is the simplest layout and makes it easier for us to manage mapping to the respective memories. With this layout, we can assign consecutive threads of a thread block to corresponding consecutive cells of the output matrix in column-major order. It is also beneficial for accessing the Matrix $M_2$ because to compute a column of the output matrix $C$, we need to access the corresponding column from the Matrix $M_2$. That is to say, if some consecutive threads are assigned to some consecutive cells of a column in the output matrix, all of these threads will access the same column of matrix $M_2$.

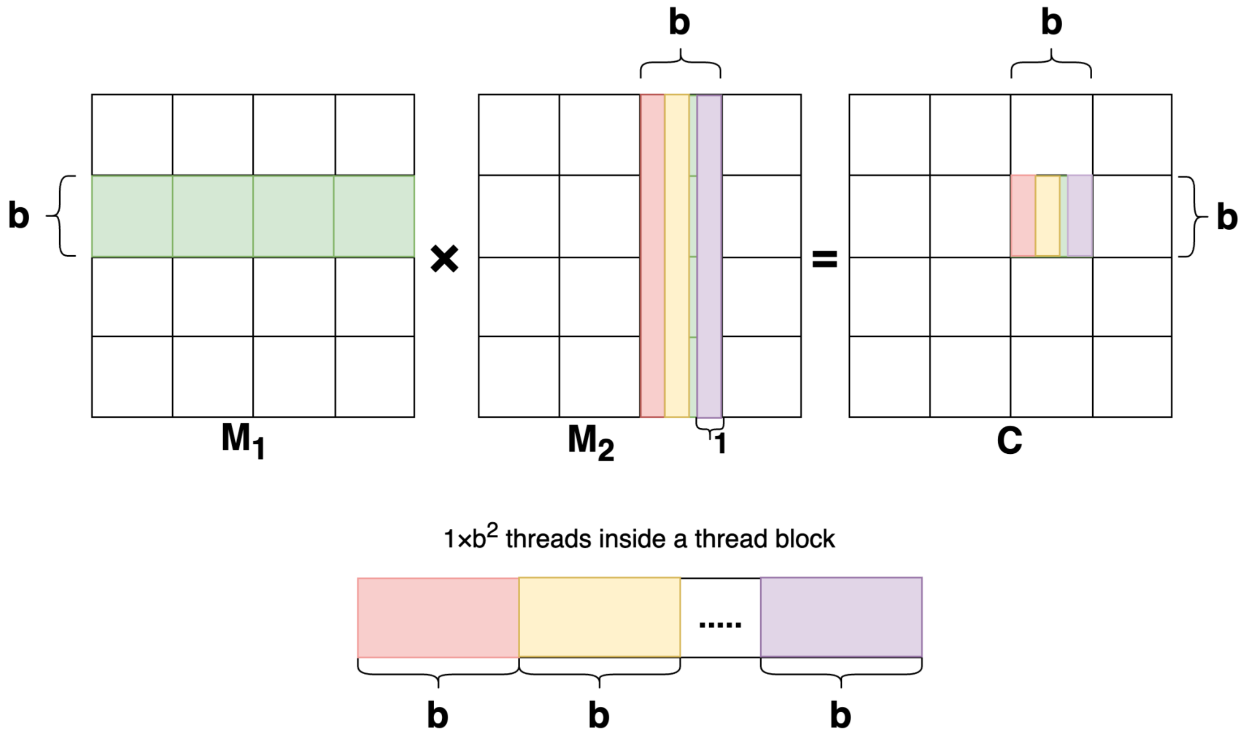

Figure 8: Thread Assignment

We can observe the memory access patterns of the first 2 threads of the red strip to get a better understanding:

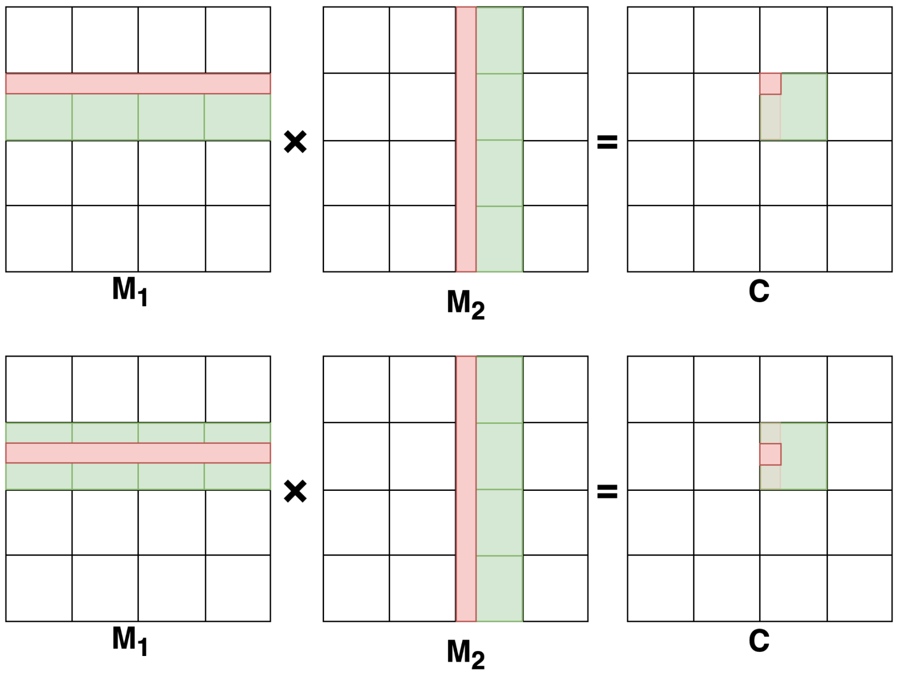

Figure 9: Memory Access Pattern

So, if the threads of the red strip are grouped into a warp, they will access the consecutive global memory of output matrix C and the same column of matrix $M_2$. As a result, the GPU will be able to optimize these memory accesses within that warp. The following curve demonstrates the performance comparison between the current approach and approach 1.

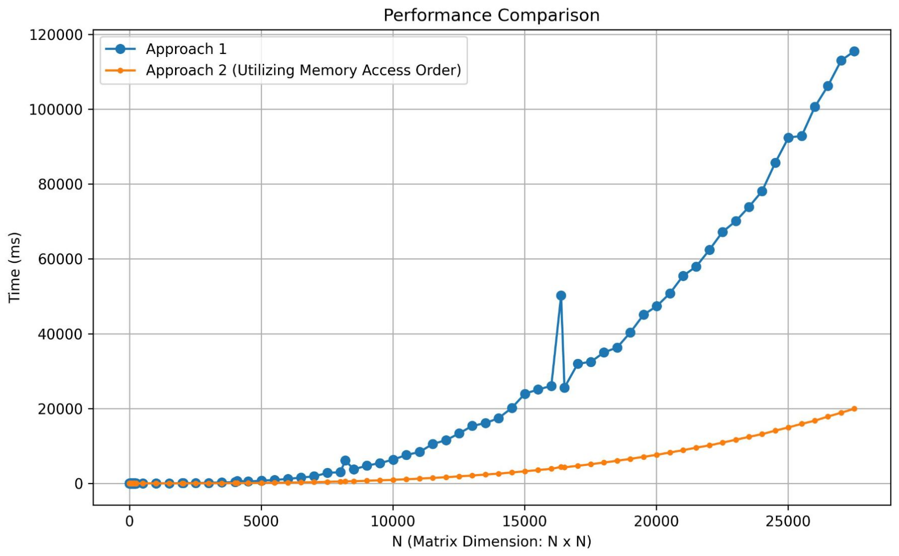

Figure 10: Runtime comparison between approaches 1 and 2

It can be observed that the approach 2 has significant improvement in the runtime. For around $27000 \times 27000$ matrices, approach 2 reduced the runtime from around 120 seconds to only 20 seconds. Following is the GPU grid & thread block declaration and complete C++ CUDA program for this approach:

**Grid and thread block declaration:**

```
dim3 dimBlock(b*b); //approach 1 had dimBlock(b, b);
dim3 dimGrid(N/b, N/b);
mat_mult_cuda<<<dimGrid, dimBlock>>>(m1, m2, c, N);
```

**Complete C++ program:**

```
#include <iostream>
#include <cstdio>
#include <cassert>
#include <chrono>
#include <set>
using namespace std;

#define BLOCK_SIZE 32

__host__ __device__ inline int get_idx(const int& row, const int& col, const int& n) {
    return row + col*n;}

__global__ void mat_mult_gpu(float *ad, float *bd, float* cd, int n)
{
    // column-major ordering
    int row = blockIdx.y * BLOCK_SIZE + (threadIdx.x % BLOCK_SIZE);
    int col = blockIdx.x * BLOCK_SIZE + (threadIdx.x / BLOCK_SIZE);

    if(row >= n || col >= n) return;

    float sum = 0;
    for (int k = 0; k < n; k++) {
        sum += ad[get_idx(row, k, n)] * bd[get_idx(k, col, n)];
    }

    cd[get_idx(row, col, n)] = sum;
}

void run(int N)
{
    // declaration
    float* a = (float *)malloc(N * N * sizeof(float));
    float* b = (float *)malloc(N * N * sizeof(float));
    float* c = (float *)malloc(N * N * sizeof(float));
    float* resultFromGpu = (float *)malloc(N * N * sizeof(float));

    // value population
    srand48(42); // seed
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            a[get_idx(i, j, N)] = drand48();
            b[get_idx(i, j, N)] = drand48();
        }
    }

    // GPU call section starts
    float *dev_a, *dev_b, *dev_c;
    cudaMalloc((void**)&dev_a, N * N * sizeof(float));
    cudaMalloc((void**)&dev_b, N * N * sizeof(float));
    cudaMalloc((void**)&dev_c, N * N * sizeof(float));

    cudaMemcpy(dev_a, a, N * N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, N * N * sizeof(float), cudaMemcpyHostToDevice);


    int blockDimSize = BLOCK_SIZE;
    int gridDimSize = N/blockDimSize;
    if (N % blockDimSize) gridDimSize++;

    dim3 dimGrid(gridDimSize, gridDimSize);
    dim3 dimBlock(blockDimSize * blockDimSize);
    // blockDim 1-dimensional for global memory coalescing

    // Time CUDA kernel execution
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    mat_mult_gpu<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);

    // Wait for the GPU to finish before exiting
    cudaDeviceSynchronize();

    // Stop timer
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float cudaTime;
    cudaEventElapsedTime(&cudaTime, start, stop);
    printf("CUDA Execution Time: %0.6f ms\n", cudaTime);

    // Copy the result from gpu
    cudaMemcpy(resultFromGpu, dev_c, N * N * sizeof(float), cudaMemcpyDeviceToHost);

    // Free Memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
    free(a);
    free(b);
    free(c);
    free(resultFromGpu);
}

```

```
int main()
{
    int n = 1024;
    run(n);
    return 0;
}
```

### 2.2.3 Approach 3: Shared Memory

We can extend the approach 2 to make our code more optimized. Each element of the matrices $M_1$ and $M_2$ are accessed multiple times by threads within a thread block. So, it would be better if we could move the data from global memory to a faster memory and then access data from this faster whenever needed. Here comes the shared memory. Shared memory is private to a thread block and faster than the global memory.

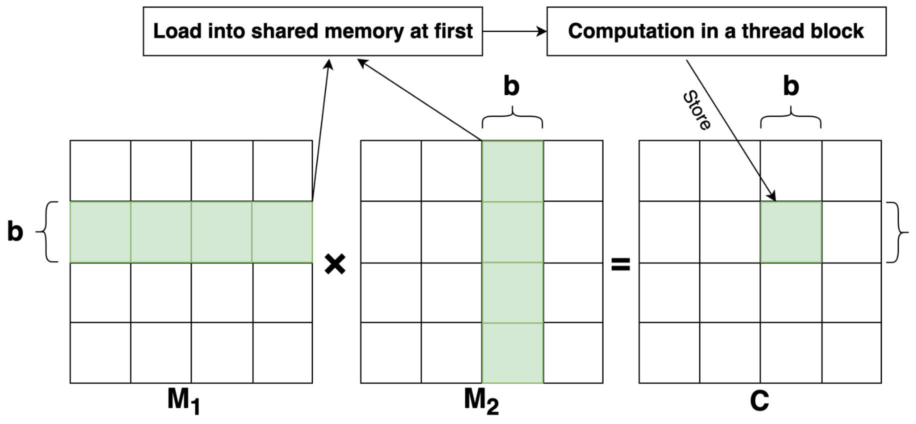
Load into shared memory at first
Computation in a thread block
Store
b
b
b
M1
M2
C

Figure 11: Shared Memory As a Buffer

The issue with the shared memory is that it's size is limited. In the DEAC cluster's GPU, a thread block's shared memory can store at most 48 KB of data. So, it isn't possible to load all data into the shared memory at once for larger matrices. To deal with this issue, we can load chunks of data one by one. For example, let's say a thread block needs $b \times N$ data from matrix $M_1$ and $N \times b$ data from the matrix $M_2$. To fit the data into shared memory, we'll load chunks of size $b \times d$ from matrix $M_1$ and $d \times b$ data from matrix $M_2$ one by one, where $d$ is a suitable constant dependent on the size of the shared memory. After loading a chunk of data, threads will perform computations on them and store the partial results in their registers. Upon completion of loading all chunks and computations, threads will store the final results in the corresponding cells of the output matrix.

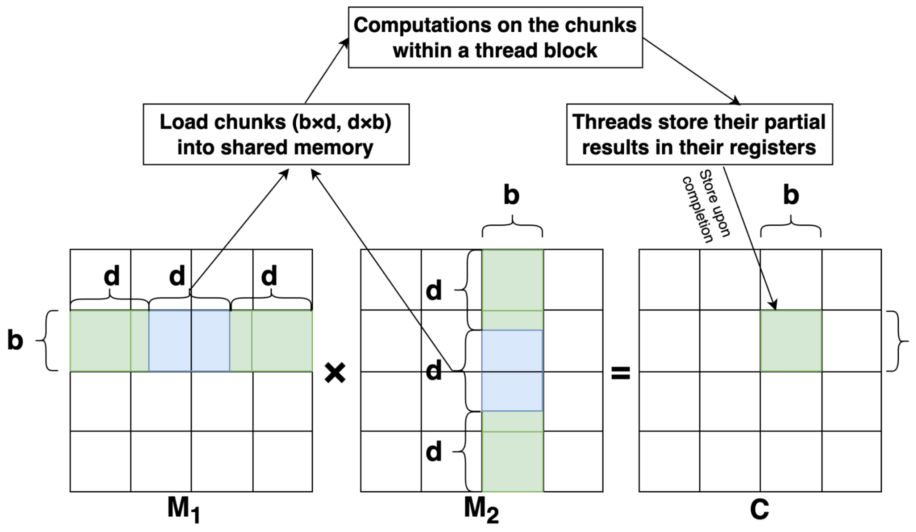
Computations on the chunks within a thread block
Load chunks (b×d, d×b) into shared memory
Threads store their partial results in their registers
Store upon completion
b
d
d
d
b
d
d
d
b
×
=
M1
M2
C

Figure 12: Shared Memory with Chunk of Data

The following plot compares the runtime performance of approach 2 and approach 3:

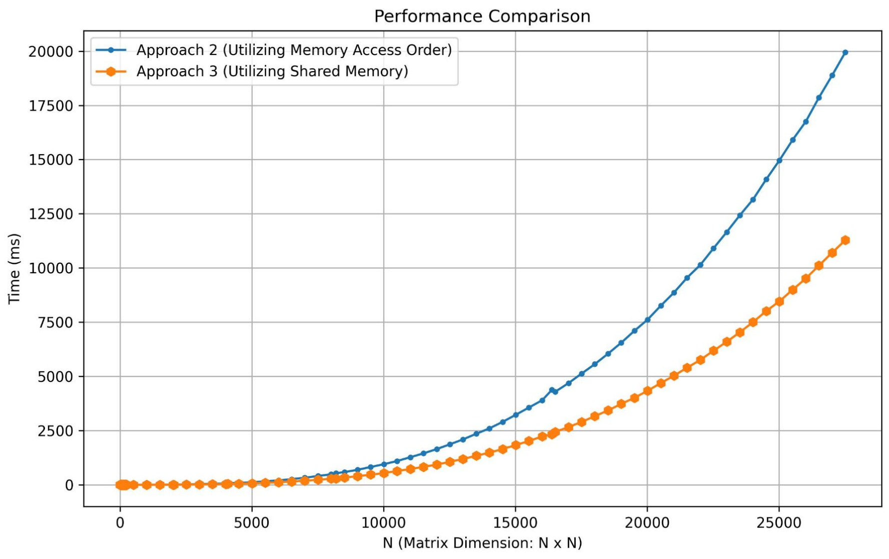

Figure 13: Runtime comparison between approaches 2 and 3

It can be observed that the approach 3 nearly halved the runtime. For $27000 \times 27000$ matrices, the runtime of approach 2 is 20 seconds whereas approach 3 takes around 11.5 seconds. Following is the pseudocode of the GPU kernel and the complete C++ CUDA program of this approach:

**Pseudocode:**

```
int thread_row = threadIdx.x % b, thread_col = threadIdx.x / b;
int row = blockIdx.y * b + thread_row;
int col = blockIdx.x * b + thread_col;
__shared__ float a_shared[b * d], b_shared[d * b];
```

```
float sum = 0; // stores in register
for(x, data_chunk_count) {
  //Within a thread block consisting of 1×b² threads, 2×b×d data will be loaded in total
  //So, each thread will load 2×b×d / b² = 2×d/b data into the shared memory
  Load respective d/b data from m1 into a_shared
  Load respective d/b data from m2 into b_shared
  __syncthreads();
  for (int k = 0; k < d; k++) {
    sum += a_shared[thread_row, k] * b_shared[k, thread_col]
  }
  __syncthreads();
}
c[row, col] = sum;
```

---

**Complete C++ program:**

---

```
#include <iostream>
#include <cstdio>
#include <cassert>
#include <chrono>
#include <set>
using namespace std;

#define BLOCK_DIM 32
#define DATA_BLOCK_DIM 64 // must be <= n and >= BLOCK_DIM. Related to The amount of data a thread

__host__ __device__ inline int get_idx(const int& row, const int& col, const int& n) {
    return row + col*n;} // n=row_dimension

__global__ void mat_mult_gpu(float *ad, float *bd, float* cd, int n)
{
    // column-major ordering
    int thread_row = threadIdx.x % BLOCK_DIM, thread_col = threadIdx.x / BLOCK_DIM;
    int row = blockIdx.y * BLOCK_DIM + thread_row;
    int col = blockIdx.x * BLOCK_DIM + thread_col;

    const int data_block_count = n / DATA_BLOCK_DIM + (n % DATA_BLOCK_DIM != 0);
    __shared__ float a_shared[BLOCK_DIM * DATA_BLOCK_DIM], b_shared[DATA_BLOCK_DIM * BLOCK_DIM];

    // regarding data loading
    int a_col_start = thread_col, b_row_start = thread_row;

    // initialization
    float sum = 0;

    // load data blocks and perform computations on them
    for (int b = 0; b < data_block_count; b++) {
        // load a's block*data_block data into shared memory
        for (int i = a_col_start, j = thread_col; j < DATA_BLOCK_DIM;
            i += BLOCK_DIM, j += BLOCK_DIM) {
            a_shared[get_idx(thread_row, j, BLOCK_DIM)] = (i<n && row<n)?
                ad[get_idx(row, i, n)] : 0;
        }
        a_col_start += DATA_BLOCK_DIM;

        // load b's data_block*block data into shared memory
        for (int i = b_row_start, j = thread_row; j < DATA_BLOCK_DIM;
            i += BLOCK_DIM, j += BLOCK_DIM) {
            b_shared[get_idx(j, thread_col, DATA_BLOCK_DIM)] = (i<n && col<n)?
                bd[get_idx(i, col, n)] : 0;
        }
        b_row_start += DATA_BLOCK_DIM;

        __syncthreads(); // sync for loading completion

        for (int k = 0; k < DATA_BLOCK_DIM; k++) {
            sum += a_shared[get_idx(thread_row, k, BLOCK_DIM)] *
                b_shared[get_idx(k, thread_col, DATA_BLOCK_DIM)];
        }

        __syncthreads(); // sync threads before loading new data in the next iteration
    }

    if (row < n && col < n) cd[get_idx(row, col, n)] = sum;
}

void run(int N)
{
    // declaration
    float* a = (float *)malloc(N * N * sizeof(float));
    float* b = (float *)malloc(N * N * sizeof(float));
    float* resultFromGpu = (float *)malloc(N * N * sizeof(float));

    // value population
    srand48(42); // seed
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            a[get_idx(i, j, N)] = drand48();
            b[get_idx(i, j, N)] = drand48();
        }
    }

    // GPU call section starts
    float *dev_a, *dev_b, *dev_c;
    cudaMalloc((void**)&dev_a, N * N * sizeof(float));
    cudaMalloc((void**)&dev_b, N * N * sizeof(float));
    cudaMalloc((void**)&dev_c, N * N * sizeof(float));

    cudaMemcpy(dev_a, a, N * N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, N * N * sizeof(float), cudaMemcpyHostToDevice);

    int blockDimSize = BLOCK_DIM;
    int gridDimSize = N/blockDimSize;
    if (N % blockDimSize) gridDimSize++;

    dim3 dimGrid(gridDimSize, gridDimSize);
    // blockDim 1-dimensional for global memory coalescing
    dim3 dimBlock(blockDimSize * blockDimSize);

    // Time CUDA kernel execution
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    mat_mult_gpu<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);

    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("CUDA Error: %s\n", cudaGetErrorString(err));
    }

    // Wait for the GPU to finish before exiting
    err = cudaDeviceSynchronize();
    if (err != cudaSuccess) {
        printf("CUDA Runtime Error after kernel execution: %s\n", cudaGetErrorString(err));
    }

    // Stop timer
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float cudaTime;
    cudaEventElapsedTime(&cudaTime, start, stop);

    printf("CUDA Execution Time: %0.6f ms\n", cudaTime);

    // Copy the result from gpu
    cudaMemcpy(resultFromGpu, dev_c, N * N * sizeof(float), cudaMemcpyDeviceToHost);

    // Free Memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
    free(a);
    free(b);
    free(resultFromGpu);
}

int main()
{
    int n = 1024;
    run(n);
    return 0;
}
```

## 2.3 MKL BLAS and cuBLAS

BLAS refers to Basic Linear Algebra Subprograms. MKL BLAS is the Intel oneAPI Math Kernel Library (oneMKL) implementation of the BLAS to perform linear algebra operations in the CPU

[6]. Similarly, for NVIDIA GPU, there is a library called cuBLAS. The cuBLAS library is an implementation of BLAS on top of the NVIDIA CUDA runtime [7]. These 2 libraries are used to evaluate our approach's runtime in terms of matrix multiplication.

Approach 3 has the best performance we've got so far. Following is the runtime performance comparison between approach 3, MKL BLAS, and cuBLAS:

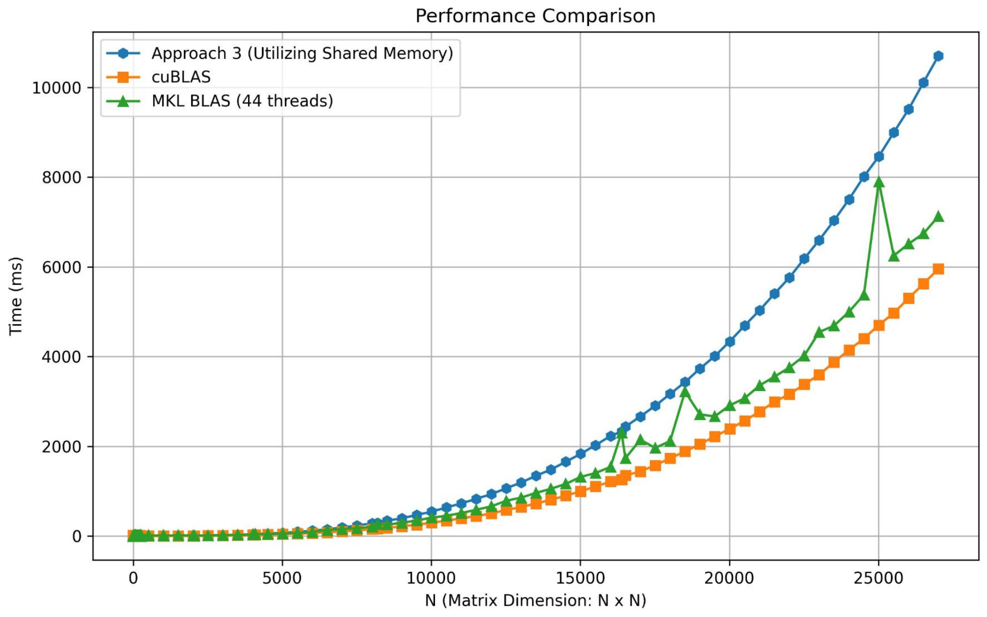

Figure 14: Runtime comparison between approach 3 and BLAS

For matrix dimension around 10000 × 1000 all three approaches gave very close performance. Then required execution time began to slowly differ. MKL BLAS is very fast and its execution time was very close to the cuBLAS. For the 27000 × 27000 matrices, its runtime differs from the cuBLAS by only around 1 second. cuBLAS takes around 6 seconds for this matrix dimension whereas our approach 3 takes around 11.5 seconds. Considering the amount of data involved, the runtime difference is not that high. Also, the BLAS libraries utilize machine-level instructions and for different matrix dimensions, they can use different implementations for better performance. These might be the reasons behind the runtime performance difference.

## 3 Conclusion

In this project, GPU architecture and related concepts regarding CUDA programming were explored. This exploration provided with valuable insights of CUDA enabled GPU memory model and resource utilization techniques that can be utilized to improve CUDA programs, and maximize throughput. This knowledge was utilized through hands-on experiences with different matrix multiplication algorithms optimized for GPU. In this project, how GPU resources can be utilized to get better performance for matrix multiplication were rigorously analysed. Finally, the performances were measured by comparing them with existing library implementations' performances. Overall, these learnings can serve as a strong foundation for future endeavors in programming on GPU and other similar devices utilizing parallelism.

## References

[1] John Hennessy, *Alphabet Presentation*. Available at: `https://www.kisacoresearch.com/sites/default/files/presentations/09.00-alphabet-johnhennessy.pdf`. Accessed: December 11, 2024.

[2] ENCCS, *GPU History*. Available at: `https://enccs.github.io/gpu-programming/1-gpu-history`. Accessed: December 11, 2024.

[3] M. Safari and M. Huisman, "Formal verification of parallel prefix sum and stream compaction algorithms in CUDA," Theor. Comput. Sci., vol. 912, pp. 81–98, 2022, doi: 10.1016/j.tcs.2022.02.027.

[4] NVIDIA, *Even Easier Introduction to CUDA*. Available at: `https://developer.nvidia.com/blog/even-easier-introduction-cuda/`. Accessed: December 11, 2024.

[5] NVIDIA Developer Blog, *How to Access Global Memory Efficiently in CUDA C++ Kernels*, 2023. Available at: `https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/` (Accessed: December 11, 2024).

[6] Intel, *BLAS and Sparse BLAS Routines*, 2024. Available at: `https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-1/blas-and-sparse-blas-routines.html`. Accessed: December 11, 2024.

[7] NVIDIA, *cuBLAS Library Documentation*. Available at: `https://docs.nvidia.com/cuda/cublas/`. Accessed: December 11, 2024.