# Hallym University

Parallel Systems Programming:

---

*Matrix Multiplication using Cuda programming.*

---

Author: Zolboo Odonkhuu

April 6, 2024

HALLYM UNIVERSITY

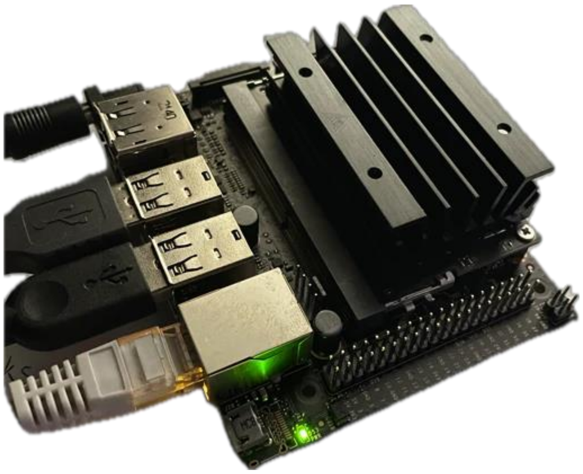

*Figure 1 Jetson Nano*

## 1. Introduction

Throughout our course, we've dived into essential computational math operations, with matrix multiplication being a key focus. This report aims to summarize our learning and practically apply it by analyzing the provided code. By digging into the code, we'll show how theory translates into practice, especially in matrix multiplication using both CPUs and GPUs, with a comparison of their performance. An interesting part of this report involves testing a hypothesis from our professor about optimizing matrix multiplication, which unexpectedly led to slower results. So, we'll also explore why this happened.

## 2. Understanding the Code

The code provided, sourced from my professor's GitHub repository [1], represents a simple yet instructive implementation of matrix multiplication employing both CPU and GPU computing paradigms. The key components and their roles are as follows:

```
__global__ void MatrixMul(int *M, int
*N, int *P, int width)
{
    int accu = 0;

    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int i = by * blockDim.y + ty;
    int j = bx * blockDim.x + tx;

    for(int k = 0; k < width; k++)
    {
    accu +=
   M[i * width + k] * N[k * width + j];
    }

    P[i * width + j] = accu;
}
```

*Figure 2Kernel function MatrixMul for GPU computation*

Above function, adorned with the __global__ qualifier, orchestrates the parallel computation of matrix multiplication on the GPU. It accepts three integer pointers M, N, and P, representing matrices A, B, and the resultant matrix C, along with the width of the matrices. Inside the kernel, each thread computes a single element of matrix C by traversing the corresponding row of matrix A and the corresponding column of matrix B, accumulating the result in the local variable accu, and subsequently storing it in the output matrix P.

```c
int main(void)
{
    GpuTimer timer;
    int i, j, k;
    int size = 1024;
    int *h_A, *h_B, *h_C, *h_gC;
    int *d_A, *d_B, *d_C;

    int sizeByte = sizeof(int) * size * size;
    h_A = (int *) malloc(sizeByte);
    h_B = (int *) malloc(sizeByte);
    h_C = (int *) malloc(sizeByte);
    h_gC = (int *) malloc(sizeByte);

    for(i = 0; i < size * size; i++) h_A[i] = 1;
    for(i = 0; i < size * size; i++) h_B[i] = 2;

    // CPU computation of matrix multiplication
    printf("Host Computing Starts!\n");
    timer.Start();
    for(i = 0; i < size; i++)
        for(j = 0; j < size; j++) {
            h_C[i * size + j] = 0;
            for(k = 0; k < size; k++)
                h_C[i * size + j] +=
                h_A[i * size + k] *
                h_B[k * size + j];
        }
    printf("Host Computing Finished!\n");
    timer.Stop();
    printf("CPU Computing Time: %f ms \n",
timer.Elapsed());
    // End of CPU computation

    // The rest of the code for GPU computation,
copying data, verification, and memory
deallocation is omitted for brevity.

    return 0;
}
```

*Figure 3Main function for CPU computation and comparison*

The main function serves as the control hub for initiating matrix multiplication processes on both CPU and GPU platforms. It commences by allocating memory for the input matrices A and B (h_A and h_B) and the output matrices C (h_C) and C computed on the GPU (h_gC). These matrices are initialized with constant values. Following this, CPU computation is executed through nested loops, whereby corresponding elements of matrices A and B are multiplied, and the results are accumulated in matrix C. Subsequently, GPU computation is triggered by allocating memory on the device for matrices A, B, and C (d_A, d_B, and d_C) using cudaMalloc, followed by transferring data from host to device via cudaMemcpy. GPU computation is then invoked by launching the MatrixMul kernel with suitable block and grid dimensions. Upon completion, the computed result is copied back to the host, and a comparison is performed between the CPU and GPU outcomes to ensure consistency.

## 3. Performance Comparison

To provide context, the variable 'size' initialized as 'int size = 1024;' denotes the dimension of the matrices used in the matrix multiplication algorithm. Specifically, it signifies that the matrices being multiplied have dimensions of 1024x1024. Additionally, for the purpose of performance comparison, three different matrix sizes were utilized: 512x512, 1024x1024, and 2048x2048.

```
zolboo@zolboo-desktop:~/MatrixMiltiply$ ./matrix512
Host Computing Statrs !
Host Computing Finished !
CPU Computing Time: 3158.312012 ms
GPU Computing Statrs !
GPU Computing Finished !
GPU Computing Time: 221.768692 ms
Success !
zolboo@zolboo-desktop:~/MatrixMiltiply$ ./matrix512
Host Computing Statrs !
Host Computing Finished !
CPU Computing Time: 3148.107910 ms
GPU Computing Statrs !
GPU Computing Finished !
GPU Computing Time: 119.550156 ms
Success !
zolboo@zolboo-desktop:~/MatrixMiltiply$ ./matrix512
Host Computing Statrs !
Host Computing Finished !
CPU Computing Time: 3159.601318 ms
GPU Computing Statrs !
GPU Computing Finished !
GPU Computing Time: 105.315521 ms
Success !
zolboo@zolboo-desktop:~/MatrixMiltiply$
```

*Figure 4Running 512 size matrix 3 times*

| Matrix Size | Experiment | CPU Computing Time (s) | GPU Computing Time (s) |
|---|---|---|---|
| 512x512 | 1 | 3.158 | 0.222 |
| 512x512 | 2 | 3.148 | 0.120 |
| 512x512 | 3 | 3.160 | 0.105 |
| 1024x1024 | 1 | 82.382 | 0.469 |
| 1024x1024 | 2 | 82.378 | 0.330 |
| 1024x1024 | 3 | 82.490 | 1.267 |
| 2048x2048 | 1 | 682.769 | 1.574 |
| 2048x2048 | 2 | 687.095 | 1.755 |
| 2048x2048 | 3 | 673.801 | 1.502 |

This table presents the CPU and GPU computing times in seconds for experiments conducted with matrix sizes of 512x512, 1024x1024, and 2048x2048. Each row represents a specific experiment, detailing the matrix size, experiment number, CPU computing time, and GPU computing time.

| Matrix Size | CPU Computing Time (s) | GPU Computing Time (s) |
|---|---|---|
| 512x512 | 3.158 | 0.222 |
| 1024x1024 | 82.382 | 0.469 |
| 2048x2048 | 673.801 | 1.502 |

Average CPU and GPU computing times

By analyzing the results, we can conclude that GPU acceleration offers a significant performance advantage over CPU processing for matrix multiplication tasks, resulting in substantial reductions in computation time and enhanced overall efficiency.

## 4. Memory Access Patterns and Performance

During the course, our professor proposed an alternative approach to matrix multiplication using CUDA kernels, aiming to optimize performance by multiplying rows of matrices M and N. Surprisingly, experimentation revealed that this method was slower than the traditional approach, prompting further investigation into the underlying factors.

Upon analysis, we identified memory access patterns as a key determinant of performance in CUDA kernels, shedding light on the counterintuitive outcomes observed.

```
zolboo@zolboo-desktop:~/MatrixMiltiply$ ./matrix1024
Host Computing Statrs !
Host Computing Finished !
CPU Computing Time: 82364.796875 ms
GPU Computing Statrs !
GPU Computing Finished !
GPU Computing Time: 338.816132 ms
Success !
zolboo@zolboo-desktop:~/MatrixMiltiply$ ./matrix1024rowrow
Host Computing Statrs !
Host Computing Finished !
CPU Computing Time: 22090.660156 ms
GPU Computing Statrs !
GPU Computing Finished !
GPU Computing Time: 864.437622 ms
Success !
zolboo@zolboo-desktop:~/MatrixMiltiply$ |
```

We conducted experiments to observe the impact of different memory access patterns on the performance of matrix multiplication on both CPU and GPU architectures. Initially, we implemented the traditional row-column multiplication approach, where the following computations were performed:

```
For CPU:
h_C[isize+j] += h_A[isize+k] * h_B[k*size+j];

For GPU:
accu = accu + M[iwidth+k] * N[kwidth+j];
```

This implementation multiplies a row of the first matrix with a column of the second matrix, ensuring coalesced memory access for both matrices on the GPU. The observed computation times were 82364.796875 ms for the CPU and 338.816132 ms for the GPU.

Subsequently, we modified the code to perform row-row multiplication, where the computations were as follows:

```
For CPU:
h_C[isize+j] += h_A[isize+k] * h_B[j*size+k];

For GPU:
accu = accu + M[iwidth+k] * N[jwidth+k];
```

In this case, the CPU implementation exhibited a faster computation time of 22090.660156 ms, as it involves fewer computations and can handle strided memory access patterns reasonably well. However, the GPU implementation slowed down to 864.437622 ms due to uncoalesced memory access for one of the matrices, leading to significant performance degradation.

Code Reference :
[1]https://github.com/jeonggunlee/CUDATeaching/blob/master/03_cuda_lab/01_matmul.cu