202321128

Jeryl Salas

Meng Ai Program

CS 239 Exercise 1

I. Introduction

In today's world, where scientific computing, artificial intelligence (AI), and data science are the prevalent buzzwords in the Technology sector, the demand for higher processing power has increased dramatically. Disciplines that rely on complex algorithms with massive matrix operations would often push the traditional central processing units (CPUs) to their limits. It turns out that the solution to this problem is the graphics cards that we use on our personal computers. Graphics processing units (GPUs) are designed to handle several tasks simultaneously and are mostly used for graphics rendering in software applications and videogames. Interestingly, the advent of Compute Unified Device Architecture (CUDA) has unleashed the full potential of GPUs for numerical computations that are parallelizable, including matrix operations. Unlike CPUs, which are sequential by nature, GPUs can separate matrices into smaller blocks and perform simultaneous operations on them, resulting in more efficient computation. In this exercise, we will test the power of the GPU + CUDA approach to matrix addition in comparison to the traditional CPU method by performing matrix addition on different matrix sizes and using three different methods, namely: element-wise, row-wise, and column-wise matrix addition. The diagram below illustrates how matrix addition will be performed in a GPU + CUDA approach, drawing influence from the illustrations shown in Kirk, D. B., et al. (2010).

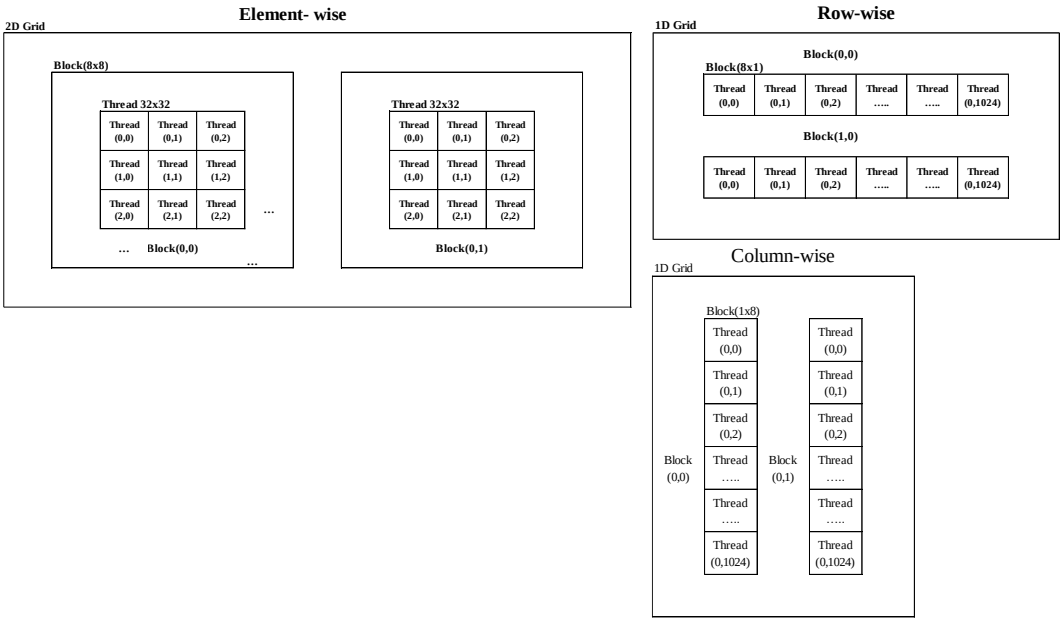

Figure 1: Matrix addition methods in a parallel framework

II. Objectives

Review key concepts in authoring massively parallel programs running on a heterogeneous computer system, as well as the use of C/C++ and CUDA programming.

III. Methodology

The exercise conducted involves performing matrix addition, where we take two matrices, B and C, to produce an output matrix A, which is the sum of those two matrices. Each element of matrix A is the sum of the elements of matrix B and C in the same position.

$$A_{ij} = B_{ij} + C_{ij} \quad (1)$$

The matrices to be used involve only square matrices with float values. The experiment conducted was divided into two sections: CPU and GPU programming. The exercise was first conducted on a C++ code so that we could first review the basic functions of a C++ code as well as compare the results of our C++ program with the CUDA program, which would justify the need to use parallel computation on our GPU's. We will perform five trials on each program and get the average results.

a. C++ programming

The first section of the exercise involves only C++ code that will be executed by the host, which is the CPU. The code starts by randomly generating matrix B and C using float numbers with a range of $[0, 100]$. The matrix sizes and modes are then defined in an array before starting the computations. The starting matrix size would be 4 and would increase exponentially for each iteration, i, which gives us an expression of $N = 4^i$ where N is the size of matrix B and C. The computations would also use three different modes: element-wise, row-wise, and column-wise, which are three different ways of adding two matrices together. The duration of computation for each matrix size and mode is then printed on a table for comparison. The algorithm would use simple looping, with the column-wise mode performing a double-nested loop.

b. CUDA programming

The main section of this exercise makes use of CUDA, which can help parallelize the matrix addition operation by splitting the matrices into blocks and threads. The first part of the code repeats the first section, establishing the matrix sizes and the three different modes that will serve as our kernels for this section. The next step is to write a host function that will serve as our main function in this matrix operation, allocating memory for the input and output matrices, transferring input matrices to the device, launching the kernels, and transferring the output matrix to the host. We

utilize the **cudaDeviceProp** to determine the properties of the device that will be used so that we can tweak our parameters, such as the block dimensions, to make better use of our hardware device. The three modes mentioned earlier are now called kernels, labeled as **kernel_1t1e**, **kernel_1t1r**, and **kernel_1t1c**. These kernels will serve as categories for our analysis of the kernel's performance for different matrix sizes in terms of average runtime (seconds) and gigaFLOPs (GFLOPS), a unit of measurement for calculating floating-point operations per second expressed in billions. This metric is heavily used in Fatahalian, K., et al. (2004), where they compared the performances of GPUs and CPUs for matrix multiplication. The equations used to compute GFLOPS on our matrix operations are as follows:

$$Number\ of\ FLOPs = 2\ \times N\ \times N \tag{2}$$

$$GFLOPS = \frac{Number\ of\ FLOPs}{Duration\ (in\ seconds)} \tag{3}$$

IV. Results and Discussion

In this section, we will go over the results of both C++ programming and CUDA programming. Keep in mind that we are mainly focusing on the CUDA program, and that the C++ program only serves as an added observation for the CPU-GPU comparison.

a. C++ programming

Matrix Size ($N = 4^i$)	Runtime (seconds)		
	Element-wise	Row-wise	Column-wise
4	0	0	0
16	0	0	0
64	0	0	0
256	0	0.001158	0
1024	0.004988	0.003983	0.002987
4096	0.04485	0.040858	0.041862

Table 1: Average runtime for different matrix addition functions

Table 1, which shows the CPU program's runtime in different matrix sizes, demonstrates that it performed very well in smaller matrix sizes, with the runtime even being zero when rounded to a few decimal places. The problem, however, is that the CPU program was only able to reach a matrix size of 4096 before terminating itself with the error of "terminate called after throwing an instance of std::bad_alloc". According to Krzemieński, A., et al. (2019), this error code

indicates that the requested memory size has exceeded the per-allocation size limit dictated by the system, which in our case is the CPU. This only shows that with the use of the basic looping algorithm and the CPU, the system cannot keep up with larger matrix sizes. Now, let us see what the GPU + CUDA can do in matrix addition.

b. CUDA programming

Matrix Size ($N = 4^i$)	Average runtime (Seconds)			GFLOPS		
	Element-wise	**Row-wise**	**Column-wise**	**Element-wise**	**Row-wise**	**Column-wise**
4	0.08292322	0.00055344	0.00042702	3.85899E-07	5.78202E-05	7.49379E-05
16	0.00044864	0.00036412	0.00037098	0.001141227	0.00140613	0.001380128
64	0.00049392	0.00065346	0.00062426	0.016585682	0.012536345	0.013122737
256	0.00308026	0.00341154	0.0038213	0.042552252	0.038420186	0.034300369
1024	0.04393948	0.0462424	0.04911512	0.047728193	0.045351279	0.042698705
4096	0.6660854	0.698719	0.9703632	0.05037557	0.048022785	0.03457925
16384	19.55186	13.26218	32.77708	0.027458815	0.040481347	0.016379461
65536	0.00160254	0.00188704	0.00002978	5360.199803	4552.068102	288446.4269
262144	0.00000358	0.00007042	0.00001052	38390769.13	1951703.401	13064539.3
1048576	0.00000276	0.00001198	0.00000892	796747556.4	183557867.7	246527270.8
4194304	0.00000238	0.00000928	0.00000912	14783349617	3791419406	3857935536
16777216	0.00000362	0.00001068	0.00001008	1.55511E+11	52710669796	55848209665
67108864	0.0000031	0.00001296	0.00000234	2.90555E+12	6.95E+11	3.84923E+12
268435456	0.00000232	0.00001058	0.00000226	6.21186E+13	1.36215E+13	6.37678E+13
1073741824	0.00000224	0.00000966	0.00000222	1.02939E+15	2.387E+14	1.03867E+15

Table 2: Average runtime and GFLOPS for element-wise, row-wise, and column-wise kernels

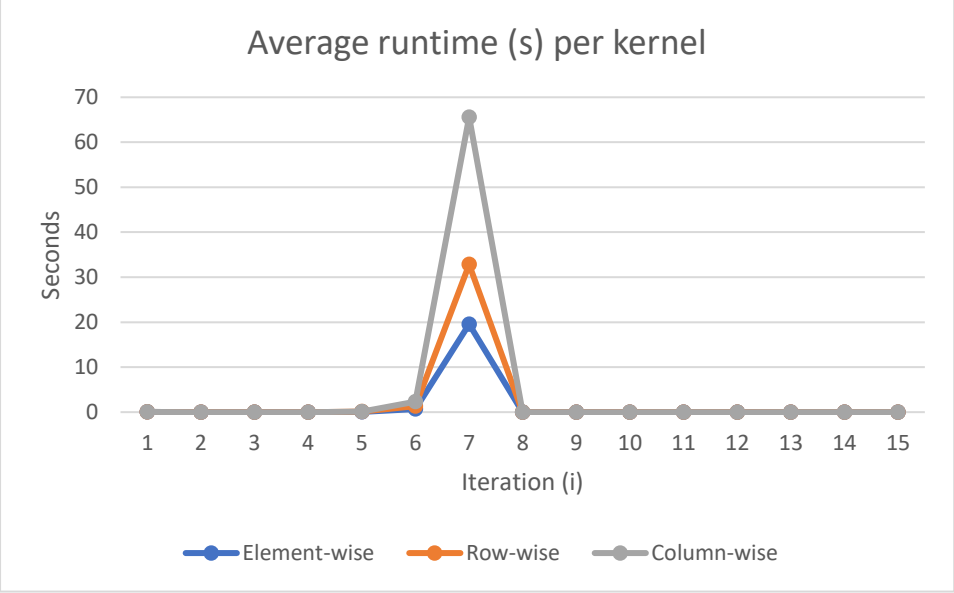

Figure 2: Average runtime per kernel line chart

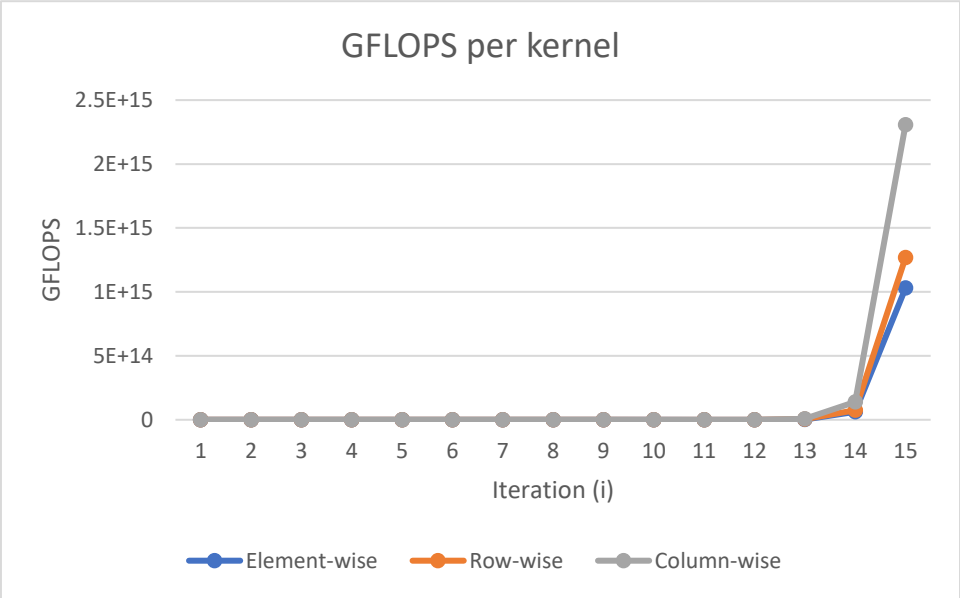

Figure 3: GFLOPS per kernel line chart

Table 2 shows the average runtime and GFLOPS of the kernels, demonstrating that it can perform matrix addition on larger matrix sizes. So large, in fact, that we have reached 1 billion-sized matrices in comparison to the CPU's 4096. One thing that caught me off guard was how the kernel's runtime is relatively stable despite the increasing matrix sizes. There is also a big jump in the runtime for a matrix of size N = 16,384, which is the seventh iteration of the matrix addition operation. Other than that, the results are mostly stable. When we average the runtime in different iterations for the three kernels, we see that the row-wise kernel is the fastest, the element-wise kernel is second, and the column-wise kernel comes last. Yet, it is hard to determine if the row-wise kernel is truly the better-performing kernel between the two, especially if we have an outlier in our results. However, in Figure 3, which shows the GFLOPS of each kernel, the GFLOPS of kernel_1t1c (column-wise) drastically increased after the 13th iteration, blowing ahead of kernel_1t1e and kernel_1t1r. This drastic change can be seen from Table 2, where the GFLOPS increased to a whopping 6.37678E+13 for matrix size N = 268,435,456.

V. Future study

For future experiments, I would like to see how CUDA + GPUs can go up against CPUs with optimized algorithms such as blocking or using well-known packages like Automatically Tuned Linear Algebra Software (ATLAS), as stated by Fatahalian, K., et al. (2004), who observed that CPUs outperformed GPUs in matrix multiplication. I would also like to personally discover if this is still the case today, given the emergence of CUDA and better GPU support for scientific computing. Lastly, it would be interesting to apply CUDA programming to well-known matrix decompositions such as QR decomposition and eigenvalue decomposition, which are widely used in machine learning applications.

VI. References and Code

- Kirk, D. B., & Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Burlington, MA: Morgan Kaufmann Publishers.

- Fatahalian, K., Sugerman, J., & Hanrahan, P. (2004). Understanding the Efficiency of GPU Algorithms for Matrix-matrix Multiplication. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (pp. 133-137). Grenoble, France: ACM. doi:10.1145/1058129.1058148

- Nvidia Developer Zone. (n.d.). CUDA Toolkit Documentation: CUDART Device Management [Web page]. Retrieved from https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html


```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cuda_runtime.h>
#include "device_launch_parameters.h"
#include <chrono> // For measuring runtime
#include <iomanip> // For displaying results on a table


// We want to print the device properties
void queryDeviceProperties() {
    cudaDeviceProp deviceProp;
    int deviceCount;
    cudaError_t error;

    error = cudaGetDeviceCount(&deviceCount);
    if (error != cudaSuccess) {    // In case there is an error in cudaGetDeviceCount (page 247)
        std::cerr << "We weren't able to detect any device: " << cudaGetErrorString(error) << std::endl;
        return;
    }

    for (int i = 0; i < deviceCount; ++i) {
        error = cudaGetDeviceProperties(&deviceProp, i);
        if (error != cudaSuccess) {        // If this specific device gets an error (Reference 3)
            std::cerr << "cudaGetDeviceProperties failed for device " << i << ": " << cudaGetErrorString(error) << std::endl;
            continue;
        }
        std::cout << "Device " << i << ": " << deviceProp.name << std::endl;

        // print the device i properties (Reference 3)
        std::cout << "  Compute capability: " << deviceProp.major << "." << deviceProp.minor << std::endl;
        std::cout << "  Max threads per block: " << deviceProp.maxThreadsPerBlock << std::endl;
        std::cout << "  Total global memory: " << deviceProp.totalGlobalMem << " bytes" << std::endl;
    }
}

// Kernel function where each thread produces one output matrix element (page 60)
__global__ void kernel_1t1e(float* A, const float* B, const float* C, int N) {
```

```
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i < N && j < N) {
        int idx = i * N + j;
        A[idx] = B[idx] + C[idx];
    }
}

// Kernel function where each thread produces one output matrix row (page 60)
__global__ void kernel_1t1r(float* A, const float* B, const float* C, int N) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < N) {
        for (int j = 0; j < N; ++j) {
            int idx = row * N + j;
            A[idx] = B[idx] + C[idx];
        }
    }
}

// Kernel function where each thread produces one output matrix column (page 60)
__global__ void kernel_1t1c(float* A, const float* B, const float* C, int N) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (col < N) {
        for (int i = 0; i < N; ++i) {
            int idx = i * N + col;
            A[idx] = B[idx] + C[idx];
        }
    }
}

//  host function by allocating memory for the input and output matrices,
// transferring input data to device, launch the kernel, transferring the output
//  data to host, and freeing the device memory for the input and output data.
// (page 52)

void matrixAdditionHost(float* A, const float* B, const float* C, int N, int mode) {
    // We query device properties to determine block and grid dimensions (page 247)
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, 0); // We only have one GPU device
    dim3 threadsPerBlock;
    dim3 numBlocks; // (page 55)

    if (mode == 0) { // For element-wise addition - kernel_1t1e (Reference 3:
https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html))
        threadsPerBlock.x = deviceProp.maxThreadsPerBlock;
        threadsPerBlock.y = deviceProp.maxThreadsPerBlock;
        numBlocks.x = (N + threadsPerBlock.x - 1) / threadsPerBlock.x;
        numBlocks.y = (N + threadsPerBlock.y - 1) / threadsPerBlock.y;
    }
    else if (mode == 1) { // For row-wise addition - kernel_1t1r (Reference 3:
https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html)
        threadsPerBlock.x = deviceProp.maxThreadsPerBlock;
        numBlocks.x = (N + threadsPerBlock.x - 1) / threadsPerBlock.x;
        numBlocks.y = 1;
    }
    else if (mode == 2) { // For column-wise addition - kernel_1t1c (Reference 3:
https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html)
        threadsPerBlock.x = 1;
        numBlocks.x = N;
        numBlocks.y = (N + deviceProp.maxThreadsPerBlock - 1) /
deviceProp.maxThreadsPerBlock;
    }
    else {
        std::cerr << "Invalid mode. Using element-wise addition." << std::endl; // If error,
it would use kernel_1t1e by default
        threadsPerBlock.x = deviceProp.maxThreadsPerBlock;
        threadsPerBlock.y = deviceProp.maxThreadsPerBlock;
        numBlocks.x = (N + threadsPerBlock.x - 1) / threadsPerBlock.x;
```

```
        numBlocks.y = (N + threadsPerBlock.y - 1) / threadsPerBlock.y;
    }

    // We use CUDA API called cudaMalloc that allocates memory on the GPU for matrices A, B,
and C (page 48)
    float* d_A, * d_B, * d_C;
    cudaMalloc((void**)&d_A, N * N * sizeof(float));
    cudaMalloc((void**)&d_B, N * N * sizeof(float));
    cudaMalloc((void**)&d_C, N * N * sizeof(float));

    // CUDA API that copies data from the host (CPU) to the device (GPU) (page 51)
    cudaMemcpy(d_B, B, N * N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, C, N * N * sizeof(float), cudaMemcpyHostToDevice);

    // Kernel launch depending on the mode that is entered on int main (page 62)
    switch (mode) {
    case 0: // For element-wise addition - kernel_1t1e
        kernel_1t1e << <numBlocks, threadsPerBlock >> > (d_A, d_B, d_C, N);
        break;
    case 1: // For row-wise addition - kernel_1t1r
        kernel_1t1r << <numBlocks, threadsPerBlock >> > (d_A, d_B, d_C, N);
        break;
    case 2: // For column-wise addition - kernel_1t1c
        kernel_1t1c << <numBlocks, threadsPerBlock >> > (d_A, d_B, d_C, N);
        break;
    default:
        std::cerr << "Invalid mode entered. Using element-wise addition instead." <<
std::endl;
        kernel_1t1e << <numBlocks, threadsPerBlock >> > (d_A, d_B, d_C, N);
    }

    // CUDA API that copies the resulting matrix A from the device (GPU) back to the host
(CPU)
    cudaMemcpy(A, d_A, N * N * sizeof(float), cudaMemcpyDeviceToHost); //(page 51)

    // Was adviced to free up memory (page 51)
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}

// Print matrix function
void printMatrix(const float* matrix, int N) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            std::cout << matrix[i * N + j] << "\t";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

// Function used to generate random matrices for matrix B and C
void generateRandomMatrix(float* matrix, int N) {
    for (int i = 0; i < N * N; ++i) {
        matrix[i] = static_cast<float>(rand()) / static_cast<float>(RAND_MAX) * 100.0f;
    }
}

int main() {

    // Clear terminal screen
    // (REF: https://stackoverflow.com/questions/228617/how-do-i-clear-the-console-in-both-
windows-and-linux-using-c)
    #ifdef _WIN32
    std::system("cls");
    #else
    std::system("clear");
    #endif
```

```
    srand(time(NULL)); // Seed for random number generation (REF:
https://mathbits.com/MathBits/CompSci/LibraryFunc/rand.htm)
    queryDeviceProperties();
    std::cout << "\n";
    std::cout << "-------------------------Device 0 performance-------------------------"
<< std::endl;
    std::cout << "\n";

    // Matrix sizes and modes placed in an array
    int sizes[] = { 4, 16, 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576, 4194304,
16777216, 67108864, 268435456, 1073741824};
    const char* modeDescriptions[] = { "     Element-wise", "        Row-wise", "
Column-wise" };
    std::cout << std::setw(15) << "Matrix Size";
    for (const auto& modeDesc : modeDescriptions) {
        std::cout << std::setw(20) << modeDesc;
    }
    std::cout << std::endl;

    // Loop through different values of N and modes (0 to 2)
    for (int i = 0; i < sizeof(sizes) / sizeof(sizes[0]); ++i) {
        int N = sizes[i];
        std::cout << std::setw(16) << N;

        for (int mode = 0; mode <= 2; ++mode) {
            auto start = std::chrono::high_resolution_clock::now();

            // Matrices
            float* B = new float[N * N];
            float* C = new float[N * N];
            float* A = new float[N * N];

            // We generate random matrices B and C  and perform matrix addition
            generateRandomMatrix(B, N);
            generateRandomMatrix(C, N);
            matrixAdditionHost(A, B, C, N, mode);
            auto end = std::chrono::high_resolution_clock::now();
            std::chrono::duration<double> duration = end - start;
            std::cout << std::setw(20) << duration.count() << " s";

            // Clean matrices right after (REF: https://cboard.cprogramming.com/cplusplus-
programming/41002-delete-delete[].html)
            delete[] B;
            delete[] C;
            delete[] A;
        }
        std::cout << std::endl;
    }

    return 0;
}
```