CS 239 Programming Exercise 1

Jamaica Mae L. Pepito

CS 239 (MZZQ)

1 Introduction

Graphics Processing Units (GPUs) have evolved from graphics hardware into powerful accelerators for parallel computing. Unlike CPUs, which have few cores optimized for sequential tasks, GPUs have thousands of lightweight cores suited for data-parallel workloads such as vector operations, image processing, and matrix computations [1, 3, 8]. In HPC, scientific simulations, and machine learning, GPUs provide significant performance gains over CPU-only implementations [3, 6].

This exercise examines matrix addition, an instructive operation for studying thread mapping strategies. Three CUDA kernels were implemented: **1 thread per element (1t1e)**, **1 thread per row (1t1r)**, and **1 thread per column (1t1c)**. Comparing these to a CPU baseline highlights differences in memory coalescing, warp efficiency, and occupancy [4, 9], with timing and device properties measured on an NVIDIA GeForce GTX 1650.

2 Methodology

2.1 Device Query and Setup

GPU device properties were queried using the standard CUDA API `cudaGetDeviceProperties` [7]. A helper function `printDeviceProperties` was implemented to display key device specifications, including compute capability, memory size, maximum threads per block, and multiprocessor count, guiding kernel configuration and performance tuning. Results are shown in section 3.1.1.

```
void printDeviceProperties(int dev) {
    cudaDeviceProp prop;
    CUDA_CHECK(cudaGetDeviceProperties(&prop, dev));
    printf("Device %d: %s\n", dev, prop.name);
    printf(" Compute capability: %d.%d\n", prop.major, prop.minor);
    printf(" Total global memory: %zu MB\n",
          (size_t)prop.totalGlobalMem / (1024 * 1024));
    printf(" Max threads per block: %d\n", prop.maxThreadsPerBlock);
    printf(" MultiProcessorCount (SMs): %d\n", prop.multiProcessorCount);}
```

2.2 Matrix Addition Kernel

The three CUDA kernels were implemented to perform matrix addition, each differing in how work is mapped to threads:

2.2.1 (a) 1 thread → 1 element (1t1e).

Each GPU thread computes one matrix element, with thread indices mapped directly to row and column positions:

```
__global__ void kernel_1t1e(float* A, const float* B, const float* C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < N && col < N) {
        int idx = row * N + col;
        A[idx] = B[idx] + C[idx];}}
```

This kernel achieves fine-grained parallelism with a natural 2D mapping of threads to matrix elements. A 16×16 block (256 threads) is used by default, aligning with the warp size for efficient utilization. Consecutive threads access consecutive memory locations, ensuring coalesced global memory access. The flattened index `row * N + col` matches the 1D memory layout of matrices, maintaining both correctness and optimality.

2.2.2 (b) 1 thread → 1 row (1t1r).

Each thread processes an entire row of the matrix, reducing the number of threads launched but introducing strided memory access:

```
__global__ void kernel_1t1r(float* A, const float* B, const float* C, int N) {
    int threadRow = blockIdx.x * blockDim.x + threadIdx.x; // 1D mapping across rows
    if (threadRow < N) {
        int rowStart = threadRow * N;
        // iterate across columns
        for (int col = 0; col < N; ++col) {
            int idx = rowStart + col;
            A[idx] = B[idx] + C[idx]; }}}
```

In this kernel, each thread handles one row of the output, lowering scheduling overhead but causing strided global memory access. Since consecutive threads access distant locations in memory, coalescing is poor, making this kernel less efficient and prone to performance loss at larger N compared to the 1t1e strategy.

2.2.3 (c) 1 thread → 1 column (1t1c).

Each thread is assigned an entire column:

```
__global__ void kernel_1t1c(float* A, const float* B, const float* C, int N) {
    int threadCol = blockIdx.x * blockDim.x + threadIdx.x;
    if (threadCol < N) {
        for (int row = 0; row < N; ++row) {
            int idx = row * N + threadCol;
            A[idx] = B[idx] + C[idx];}}}
```

This approach reduces the number of threads launched but suffers from strided memory access because matrices are stored in row-major order. Consecutive elements of a column are far apart in memory, preventing coalescing and leading to higher latency. Making this kernel less efficient than the 1t1e mapping, though slightly better than the row-based variant.

2.2.4 Host Wrapper (for memory allocation, transfers, kernel launch, and timing)

```
float hostMatrixAdd_1t1e(float* h_A, const float* h_B, const float* h_C, int N,
                      const cudaDeviceProp& prop, int runs = 5) {
    size_t bytes = (size_t)N * N * sizeof(float);
    float *d_A, *d_B, *d_C;
    CUDA_CHECK(cudaMalloc(&d_A, bytes));
    CUDA_CHECK(cudaMalloc(&d_B, bytes));
    CUDA_CHECK(cudaMalloc(&d_C, bytes));
    CUDA_CHECK(cudaMemcpy(d_B, h_B, bytes, cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemcpy(d_C, h_C, bytes, cudaMemcpyHostToDevice));
    dim3 block(16, 16);
    dim3 grid((N + block.x - 1) / block.x, (N + block.y - 1) / block.y);
    kernel_1t1e<<<grid, block>>>(d_A, d_B, d_C, N);
    CUDA_CHECK(cudaMemcpy(h_A, d_A, bytes, cudaMemcpyDeviceToHost));
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    return 0.0f; }
```

This host wrapper handles the end-to-end execution of the one-thread-per-element kernel. It first allocates device memory for the input and output matrices, then transfers the input data from host to device. The kernel is launched using a two-dimensional grid and block configuration (defaulting to 16×16 threads per block), ensuring that each thread computes exactly one element. After kernel execution, the result is copied back to the host, and all allocated device memory is released.

2.2.5 CPU Reference for Verification

A simple CPU version was implemented for correctness and baseline timing:

```
void matrix_add_cpu(float* A, const float* B, const float* C, int N) {
    int total = N * N;
    for (int i = 0; i < total; ++i) {
        A[i] = B[i] + C[i];}}
```

A CPU reference was implemented as a single-threaded loop over all $N \times N$ elements, computing $A[i] = B[i] + C[i]$. This serves as both a correctness check and a performance baseline, underscoring the gap between sequential execution and GPU parallelism. GPU outputs were validated against the CPU reference within a tolerance of $\epsilon = 10^{-3}$, with mismatches reported to ensure reliable comparisons.

2.2.6 Input Initialization

For all experiments, the input matrices `B` and `C` were initialized on the host with pseudorandom single-precision floating-point numbers uniformly distributed in the range $[0, 100]$. A fixed random seed (`srand(12345)`) was used to ensure reproducibility across runs. These matrices were then transferred to the device before kernel execution.

2.3 Timing Method

Execution times were measured using CUDA events (`cudaEventRecord` and `cudaEventElapsedTime`), which provide millisecond accuracy and capture only kernel execution, excluding host to device transfers. Each kernel was warmed up once, then executed 5 or 10 times, with the average runtime reported. Error checks after each launch ensured that timings reflected only successful executions:

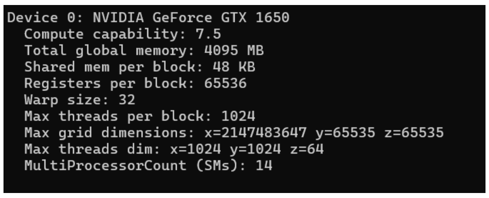
```
cudaEventRecord(start);
kernel_1t1e<<<grid, block>>>(d_A, d_B, d_C, N);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&ms, start, stop);
```

3 Experimental Results and Analysis

3.1 Results

3.1.1 Hardware Setup Results

All experiments were performed on an NVIDIA GeForce GTX 1650 GPU. The device has a compute capability of **7.5**, with a total global memory of **4095 MB**. Each block can access up to **48 KB** of shared memory, and the warp size is **32** threads. The maximum number of threads per block is **1024**, and the GPU contains **14** streaming multiprocessors (SMs). Figure 1 depicts the output of the queried properties of the device, that are installed in the system utilizing `thecudaDeviceProp` type. These properties guided the choice of block size (e.g.,16×16 threads) to ensure optimal warp utilization.



```
Device 0: NVIDIA GeForce GTX 1650
  Compute capability: 7.5
  Total global memory: 4095 MB
  Shared mem per block: 48 KB
  Registers per block: 65536
  Warp size: 32
  Max threads per block: 1024
  Max grid dimensions: x=2147483647 y=65535 z=65535
  Max threads dim: x=1024 y=1024 z=64
  MultiProcessorCount (SMs): 14
```

Figure 1: CUDA device properties for the NVIDIA GTX 1650.

3.1.2 Runtime Results

Table 1a and Table 1b present the averaged runtimes over 5 and 10 runs, respectively, for the CPU baseline and the three CUDA kernel mappings (1t1e, 1t1r, 1t1c). In both cases, GPU implementations provide significant speedups over the CPU for larger matrices. The 1t1e (element-wise) kernel demonstrates the best scalability, while the 1t1r (row-wise) kernel shows poorer performance at large sizes due to strided memory access.

Table 1: Average runtimes (ms) over 5 runs and 10 runs on NVIDIA GeForce GTX 1650.

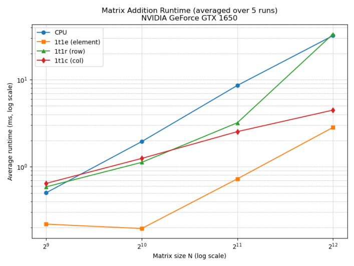
(a) 5 runs

Matrix N	CPU	1t1e	1t1r	1t1c
512	0.501	0.219	0.587	0.643
1024	1.946	0.195	1.122	1.246
2048	8.597	0.726	3.203	2.532
4096	32.109	2.831	33.493	4.475

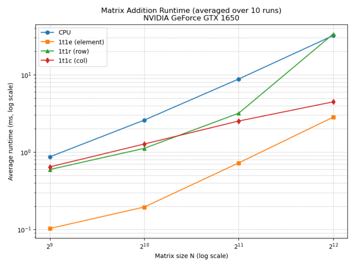
(b) 10 runs

Matrix N	CPU	1t1e	1t1r	1t1c
512	0.868	0.103	0.595	0.644
1024	2.590	0.195	1.118	1.271
2048	8.799	0.725	3.180	2.520
4096	32.019	2.828	33.493	4.456

Figures 2 and 3 present the runtime and relative speedup of CPU and GPU kernels for matrix addition on an NVIDIA GeForce GTX 1650.

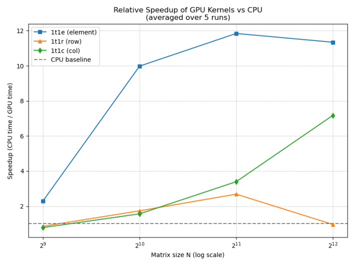


(a) Runtime comparison averaged **over 5 runs**.

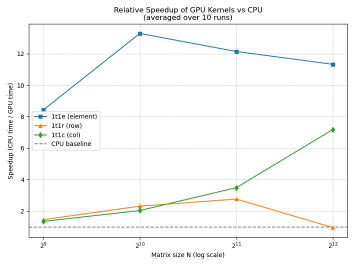


(b) Runtime comparison averaged **over 10 runs**.

Figure 2: Runtime performance of CPU and GPU kernels (1t1e, 1t1r, 1t1c) for matrix addition on an NVIDIA GeForce GTX 1650.



(a) Relative speedup averaged **over 5 runs**.



(b) Relative speedup averaged **over 10 runs**.

Figure 3: Relative speedup of GPU kernels (1t1e, 1t1r, 1t1c) compared to CPU baseline for matrix addition.

For runtime (Figure 2), GPU performance is shown to strongly depend on thread-to-data mapping. For small matrices (e.g. $N = 512$), the CPU performs comparably or better due to kernel launch and memory transfer overheads. As N grows, the advantage of GPU becomes clear, with the 1t1e kernel consistently fastest due to fine-grained parallelism and coalesced memory access. In contrast, 1t1r and 1t1c scale poorly for large matrices because of strided or non-coalesced accesses, resulting in runtimes close to or worse than the CPU. These results indicates that GPU acceleration is beneficial only for sufficiently large problems and that efficient thread mapping is critical to performance. The speedup plots (Figure 3) further illustrate this effect. The 1t1e kernel achieves the highest gains, surpassing $10\times$ speed-up at $N = 2048$ and maintaining strong performance at larger sizes. In contrast, 1t1r shows little improvement and even degrades for large N due to strided memory access, while 1t1c performs slightly better but still lags behind 1t1e.

3.2 Analysis

For small matrices (e.g., $N = 512$), the CPU performs comparably to the GPU since kernel launch and memory transfer overheads dominate execution, but as N increases the GPU advantage becomes clear. The 1t1e kernel scales best because each thread computes one element with coalesced memory access and balanced workload, achieving $\sim 13\times$ speedup at $N = 1024$ and sustaining ~ 11–$12\times$ at $N = 2048$–4096. In contrast, the 1t1r kernel suffers from strided memory access and poor warp utilization: while slightly faster than the CPU at $N = 2048$, its runtime grows to ~ 33 ms at $N = 4096$, offering almost no speedup. The 1t1c kernel performs better than 1t1r thanks to partial coalescing, reaching $\sim 7\times$ speedup at $N = 4096$, but still lags behind 1t1e. The results show that GPU performance depends on thread-to-data mapping, with 1t1e giving the most optimal scalability.

4 Conclusion

This study compared three CUDA kernel mappings for matrix addition: one thread per element (1t1e), one thread per row (1t1r), and one thread per column (1t1c). The results showed that 1t1e consistently delivered the best performance, achieving up to $11\times$ speedup over the CPU at $N = 4096$ due to balanced workload distribution and coalesced memory access. In contrast, 1t1r and 1t1c were constrained by strided or partially coalesced accesses, with 1t1r exhibiting the poorest scalability and runtimes comparable to the CPU. The results show that CUDA performance is shaped by memory access, thread mapping, and tuning block sizes to hardware features like warp size and occupancy.

Authorship Note

The CUDA kernels (Listings 1–3), host wrapper (Listing 4), and CPU reference were implemented by the author. The methodology, experiments, and analysis sections are also original work. Certain parts, such as the use of the `cudaGetDeviceProperties` API and the general structure of CUDA timing with events, follow standard CUDA programming practices described in the NVIDIA CUDA Programming Guide [6, 7]. Additionally, the general idea of mapping threads to elements, rows, or columns and host wrapper was adapted from publicly available CUDA Matrix Addition exercise and example [2, 5]. However, all kernel implementations presented in Listings 1–3 are implemented by the author. Any performance comparisons, analyses, and conclusions drawn are the author's own.

References

[1] Pieter Hijma et al. Optimization techniques for gpu programming. *ACM Computing Surveys*, 55 (12):1–39, 2023. `doi:10.1145/3570638`. URL `https://dl.acm.org/doi/full/10.1145/3570638`.

[2] jcbacong. Cuda matrix addition example. `https://github.com/jcbacong/CUDA-matrix-addition`, 2025.

[3] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2nd edition, 2012.

[4] Zhihao Lin, Hyesoon Kim, and Zhiling Lan. Gpu performance vs. thread-level parallelism. *ACM Transactions on Architecture and Code Optimization*, 19(4):1–26, 2022. `doi:10.1145/3554746`. URL `https://hzhou.wordpress.ncsu.edu/files/2022/12/gpuduet_taco17_final.pdf`.

[5] MassedCompute FAQ. Can you provide an example of how to use cuda kernels for matrix addition on rtx a6000 ada? `https://massedcompute.com/faq-answers/?question=Can%20you%20provide%20an%20example%20of%20how%20to%20use%20CUDA%20kernels%20for%20matrix%20addition%20on%20RTX%20A6000%20ADA?`, 2025.

[6] NVIDIA Corporation. *CUDA C Programming Guide*, 2025. URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[7] NVIDIA Corporation. *CUDA Runtime API: Device Management*, 2025. `https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html`.

[8] NVIDIA Developer. Gpu gems contributors. `https://developer.nvidia.com/gpugems/gpugems/contributors`, 2004.

[9] S. Zhao and L. Boström. Performance analysis of cuda-based general matrix multiplication (gemm). Master's thesis, KTH Royal Institute of Technology, 2025. URL `https://kth.diva-portal.org/smash/get/diva2:1985710/FULLTEXT01.pdf`.