SC4064 Assignment 1 Report

Qian Jianheng Oscar

January 31, 2026

System Requirements

The code was developed and tested on the following system:

Component	**Specification**
OS	Ubuntu 24.04.3 LTS
Kernel	5.14.0-284.25.1.el9_2.x86_64
Architecture	x86_64
GPU	NVIDIA H100 80GB HBM3
NVIDIA Driver	550.90.07
CUDA Version	13.1

For detailed environment setup, refer to `README.md` in the code repository.

Measurement Method

Kernel execution time is measured using CUDA events (`cudaEventRecord`, `cudaEventElapsedTime`) around the kernel launch. Unless stated otherwise, reported times exclude host-side initialization and host–device memory transfers.

FLOPS definition used (as implemented in code): For vector/matrix addition, I compute FLOPs as $2 \times$ `numElements`$/t$ (i.e., 2 FLOPs per element), matching the implementation used to produce the reported numbers. For matrix multiplication, I use the conventional $2MNK$ FLOP count.

For a measured kernel time t (seconds),

$$\text{FLOPS} = \frac{\#\text{floating-point operations}}{t}, \qquad \text{GFLOPS} = \frac{\text{FLOPS}}{10^9}.$$

1 Problem 1: Vector Addition

1.1 Overview

I implement vector addition

$$C[i] = A[i] + B[i]$$

for vectors of type `float` and length $N \geq 2^{30}$. The input vectors A and B are initialized with random values in $[0, 100]$.

1.2 Error-checking macro

All CUDA API calls are wrapped using the Week 2 error-checking macro.

Macro snippet:

```
// Error handling MACRO
#define CUDA_CHECK(call) do { \
  cudaError_t err = call; \
  if (err != cudaSuccess) { \
    fprintf(stderr, "CUDA Error: %s (at %s:%d)\n", \
      cudaGetErrorString(err), __FILE__, __LINE__); \
    exit(err); \
  } \
} while (0)
```

1.3 Thread indexing and launch configuration (1D grid, 1D blocks)

Each thread computes one element.

Global thread index:

$$i = \text{blockIdx}.x \cdot \text{blockDim}.x + \text{threadIdx}.x.$$

Bounds check: compute only if $i < N$.

Grid size (computed at runtime):

$$\text{gridDim}.x = \frac{N + \text{blockDim}.x - 1}{\text{blockDim}.x}$$

where the division is integer division in C/C++.

Kernel snippet:

```
// CUDA kernel for vector addition
__global__ void vectorAdd(const float* A, const float* B, float* C, int
   n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;  // Calculate global
     thread index
  if (i < n) {
    C[i] = A[i] + B[i];  // Perform vector addition
  }
}

// Calculate grid size
int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
```

1.4 Results

Using the FLOPS definition in my code (2 FLOPs per element for addition),

$$\#\text{FLOPs} = 2N.$$

Block size (threads)	Grid size (blocks)	Kernel time (ms)	Achieved GFLOPS
32	$\lceil N/32 \rceil$	21.1654	50.73
64	$\lceil N/64 \rceil$	10.0874	106.44
128	$\lceil N/128 \rceil$	5.0725	211.68
256	$\lceil N/256 \rceil$	4.5934	233.76

Table 1: Vector addition performance for different block sizes.

1.5 Discussion

Table 1 shows that performance improves substantially as the block size increases from 32 to 256 threads. In my measurements, the achieved throughput increased from 50.73 GFLOPS (32 threads/block, 21.1654 ms) to 233.76 GFLOPS (256 threads/block, 4.5934 ms).

This trend is consistent with improved GPU utilization at larger block sizes: more threads are available to hide memory latency, and kernel-launch/scheduling overhead is amortized over more work per block. Since vector addition is primarily memory-bandwidth bound, performance gains typically taper off once the device is close to saturating memory bandwidth; here, the improvement from 128 to 256 threads/block (211.68 GFLOPS to 233.76 GFLOPS) is smaller than the improvements at lower block sizes.

2 Problem 2: Matrix Addition

2.1 Overview

I compute

$$C(i,j) = A(i,j) + B(i,j)$$

for 8192×8192 matrices of type `float`. Matrices A and B are initialized with random values in $[0, 100]$, and C is initialized to 0.

Let $n = 8192$ (matrix dimension), so the total number of elements is n^2.

2.2 Version 1: 1D configuration (1D grid, 1D blocks)

2.2.1 Global thread index

$$\text{tid} = \text{blockIdx}.x \cdot \text{blockDim}.x + \text{threadIdx}.x.$$

2.2.2 Mapping from thread index to matrix element

In the 1D kernel, the matrix is treated as a flattened 1D array in row-major order. The global thread index `i` directly corresponds to the flattened element index:

$$\text{idx} = \text{tid}.$$

If the row/column indices are needed,

$$\text{row} = \left\lfloor \frac{\text{tid}}{n} \right\rfloor, \qquad \text{col} = \text{tid} \bmod n.$$

This matches the kernel bounds check `tid < n*n`.

Kernel snippet (1D):

```
// CUDA kernel for matrix addition - 1D
__global__ void matrixAdd1D(const float* A, const float* B, float* C, int
    n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;  // Calculate global
      thread index
  if (i < n * n) {
    C[i] = A[i] + B[i];  // Perform matrix addition
  }
}
```

2.3 Version 2: 2D configuration (2D grid, 2D blocks)

2.3.1 Global thread indices

$$\text{row} = \text{blockIdx}.y \cdot \text{blockDim}.y + \text{threadIdx}.y, \qquad \text{col} = \text{blockIdx}.x \cdot \text{blockDim}.x + \text{threadIdx}.x.$$

2.3.2 Mapping from thread indices to memory

Row-major linear index:

$$\text{idx} = \text{row} \cdot n + \text{col}.$$

Bounds check: compute only if row $< n$ and col $< n$.

Kernel snippet (2D):

```
// CUDA kernel for matrix addition - 2D
__global__ void matrixAdd2D(const float* A, const float* B, 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;
    C[idx] = A[idx] + B[idx];  // Perform matrix addition
  }
}
```

2.4 Performance comparison

- **1D configuration:** 1.4634 ms, 9.17×10^{10} FLOPS (91.7 GFLOPS)
- **2D configuration:** 0.3508 ms, 3.83×10^{11} FLOPS (383 GFLOPS)

Explanation: The 2D configuration is faster by approximately $1.4634/0.3508 \approx 4.17\times$ and achieves about $3.83 \times 10^{11}/9.17 \times 10^{10} \approx 4.18\times$ higher throughput.

Both kernels perform one addition per matrix element and are primarily limited by global-memory traffic. However, the 2D launch maps the thread indices directly to (row, col), which typically yields a more natural access pattern: threads in the same warp advance along the column dimension (`threadIdx.x`), producing contiguous memory accesses and good memory coalescing. In contrast, the 1D kernel flattens the matrix into a single index; while accesses are still contiguous, the 1D launch can lead to less effective scheduling/occupancy depending on the chosen block size and how the workload is partitioned. Overall, the measured results indicate that the chosen 2D grid/block configuration provides better effective utilization for this workload on the tested GPU.

3 Problem 3: Matrix Multiplication

3.1 Overview

I compute $C = A \times B$ for square matrices with $M = K = N = 8192$ using `float` elements. Matrices A and B are initialized with random values in $[0, 1]$, and C is initialized to 0.

3.2 Thread-to-element mapping

I launch a 2D grid with 2D blocks. Each thread computes one output element $C(i, j)$:

$$i = \text{blockIdx}.y \cdot \text{blockDim}.y + \text{threadIdx}.y, \qquad j = \text{blockIdx}.x \cdot \text{blockDim}.x + \text{threadIdx}.x.$$

Compute only if $i < M$ and $j < N$.

3.3 Per-thread computation

Each thread computes the inner product

$$C(i,j) = \sum_{k=0}^{K-1} A(i,k)\,B(k,j).$$

In code, this is implemented using an accumulator `sum` and a loop over k.

Kernel snippet:

```
__global__ void matrixMulKernel(float *A, float *B, float *C, int M, int
    K, int N) {
    // Compute row and column index
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Boundary check
    if (row < M && col < N) {
        float sum = 0.0f;
        // Compute the inner product
        for (int k = 0; k < K; k++) {
            sum += A[row * K + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
```

3.4 Block-size sweep

Block size ($x \times y$)	Kernel time (ms)	Throughput (TFLOPS)
8×8	328.22	3.35
16×16	212.96	5.16
32×32	185.26	5.94

Table 2: Matrix multiplication performance for different 2D block sizes.

3.5 Discussion

Throughput improves as the block size increases from 8×8 to 32×32 (3.35 to 5.94 TFLOPS), and the kernel time decreases from 328.22 ms to 185.26 ms.

Larger blocks increase the amount of work issued per block and can improve occupancy and latency hiding, reducing the impact of global-memory latency in the naive matrix-multiplication kernel. Additionally, larger 2D blocks can improve the reuse of loaded data within a warp schedule (even without explicit shared-memory tiling), which can reduce overheads and improve instruction throughput.

The gain from 16×16 to 32×32 is smaller than from 8×8 to 16×16, suggesting diminishing returns as the configuration approaches a more efficient utilization of the GPU's compute resources.