SC4064 Assignment 1 Report

Pu Fanyi
U2220175K
FPU001@e.ntu.edu.sg

All programs were compiled with `nvcc -O2 -std=c++20` and tested on the platform described in Table 1. CUDA events were used for kernel timing; each kernel was preceded by a warm-up run.

Component	Details
OS	Ubuntu 24.04.4 LTS (Noble Numbat)
Kernel	Linux 5.14.0-284.25.1.el9_2.x86_64
CPU	2× Intel Xeon Gold 6448Y (128 threads)
RAM	2.0 TiB
GPU	8× NVIDIA H100 80 GB HBM3
NVIDIA Driver	550.90.07
CUDA Toolkit	13.1 (V13.1.115)
GCC/G++	14.2.0

Table 1: Experimental environment.

1 Problem 1: Vector Addition

Setup. Two vectors of $N = 2^{30} \approx 1.07 \times 10^9$ single-precision floats are added element-wise: $C[i] = A[i] + B[i]$. Both vectors are initialised directly on the GPU using a hash-based pseudo-random kernel, producing values in $[0, 100]$. Each thread computes one element using global index idx $=$ blockIdx.x $\times$ blockDim.x $+$ threadIdx.x, with a bounds check idx $< N$. Four block sizes were tested; the grid size is $\lceil N/\text{blockSize} \rceil$.

Block Size	Grid Size	Time (ms)	GFLOPS
32	33,554,432	20.161	53.26
64	16,777,216	10.083	106.49
128	8,388,608	5.070	211.76
256	4,194,304	4.597	**233.57**

Table 2: Vector addition performance ($N = 2^{30}$, 1 FLOP per element).

Analysis. Performance improves $4\times$ from block size 32 to 128, with diminishing gains at 256. A block of 32 threads equals a single warp, under-utilising the SM's latency-hiding capacity. Larger blocks (128–256) allow more concurrent warps per SM, improving occupancy. Since vector addition is *memory-bandwidth-bound* (1 FLOP per 12 bytes transferred), GFLOPS reflects effective memory throughput. The peak 233.57 GFLOPS at block size 256 corresponds to ∼2.8 TB/s effective bandwidth, near the H100's HBM3 limit.

2 Problem 2: Matrix Addition

Setup. Two 8192×8192 matrices (67,108,864 elements) are added element-wise: $C[i][j] = A[i][j] + B[i][j]$. Both matrices are initialised on the GPU with pseudo-random values in $[0, 100]$. Two kernel configurations are compared.

2.1 1D Configuration

The matrix is treated as a flat array. Each thread computes one element:

$$\text{idx} = \text{blockIdx.x} \cdot \text{blockDim.x} + \text{threadIdx.x}, \quad i = \lfloor \text{idx}/\text{COLS} \rfloor, \quad j = \text{idx} \bmod \text{COLS}$$

Since the matrix uses row-major storage, linear index `idx` directly addresses element (i, j).

Block Size	Grid Size	Time (ms)	GFLOPS
64	1,048,576	0.639	105.02
128	524,288	0.324	207.13
256	262,144	0.292	**229.65**
512	131,072	0.301	222.79

Table 3: Matrix addition — 1D configuration.

2.2 2D Configuration

A 2D grid of 2D thread blocks maps directly to matrix coordinates:

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

The linear memory offset is $i \times \text{COLS} + j$.

Block Size	Grid Size	Time (ms)	GFLOPS
$(16, 16)$	$(512, 512)$	0.310	216.76
$(32, 8)$	$(256, 1024)$	0.300	**223.36**
$(32, 16)$	$(256, 512)$	0.315	212.87
$(32, 32)$	$(256, 256)$	0.339	197.83

Table 4: Matrix addition — 2D configuration.

Performance comparison. The best 1D result (block 256, **229.65** GFLOPS) slightly outperforms the best 2D result ($(32, 8)$, 223.36 GFLOPS). Both perform identical memory accesses (row-major, coalesced); differences arise from scheduling and occupancy. In the 2D case, $(32, 32)$ blocks of 1024 threads limit concurrent blocks per SM, slightly hurting performance. The $(32, 8)$ variant with 256 threads per block performs best because the x-dimension aligns with consecutive memory addresses, ensuring coalesced access within each warp.

3   Problem 3: Matrix Multiplication

Setup. $C = A \times B$ where $A, B, C \in \mathbb{R}^{8192 \times 8192}$. Both A and B are initialised on the GPU with pseudo-random values in $[0, 1]$. Each thread computes one element of C:

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

Thread-to-element mapping. Using 2D blocks and grid: $i = \text{blockIdx.y} \cdot \text{blockDim.y} + \text{threadIdx.y}$, $j = \text{blockIdx.x} \cdot \text{blockDim.x} + \text{threadIdx.x}$, with row-major storage: $A(i, k)$ at `A[i*K+k]`, $B(k, j)$ at `B[k*N+j]`, $C(i, j)$ at `C[i*N+j]`.

Inner product. Each thread loops over $k = 0, \ldots, K-1$, accumulating `sum += A[i·K+k] × B[k·N+j]` in a register, then writes to $C[i \cdot N + j]$. Total FLOPs: $2MNK = 2 \times 8192^3 \approx 1.1 \times 10^{12}$.

Block Size	Grid Size	Time (ms)	GFLOPS
$(8, 8)$	$(1024, 1024)$	327.22	3,360.18
$(16, 16)$	$(512, 512)$	213.22	5,156.67
$(32, 32)$	$(256, 256)$	185.61	**5,923.77**

Table 5: Matrix multiplication performance (8192×8192, $2MNK$ FLOPs).

Performance comparison. $(32, 32)$ achieves **5,923.77** GFLOPS — a $1.76\times$ speedup over $(8, 8)$. Although matrix multiplication has high arithmetic intensity ($2K$ FLOPs per output element), this naive kernel is *memory-bandwidth-limited* because every thread independently loads an entire row of A and column of B from global memory: each element of A is redundantly fetched N times and each element of B is fetched M times across all threads. Note that the accesses to B are coalesced (threads in a warp differ in `col`, so `B[k*N+col]` addresses consecutive locations), and accesses to the same row of A can be served from L1/L2 cache across threads sharing the same `row`. Larger blocks improve performance by increasing warp-level parallelism and cache utilisation: $(8, 8)$ blocks contain only 2 warps, limiting SM occupancy and causing more frequent cache evictions, whereas $(32, 32)$ blocks with 32 warps keep the SM fully occupied. Even so, the best result (5.9 TFLOPS) reaches only $\sim$12% of the H100's peak ($\sim$51 TFLOPS FP32), confirming that shared-memory tiling would be needed to eliminate the redundant global memory traffic and approach peak throughput.