1 Problem 1: Vector Addition

1.1 Block Size Configurations

One–dimensional blocks of sizes 32×1, 64×1, 128×1, and 256×1 were evaluated. The array size was fixed at $N = 2^{30}$ elements and grid dimension was computed dynamically via

$$\text{gridDim} = \left\lceil \frac{N}{\text{blockDim}} \right\rceil = \frac{N + \text{blockDim} - 1}{\text{blockDim}}.$$

1.2 Performance Results

Table 1: Vector Addition Performance

Block Size	Grid Dimension	Time (ms)	GFLOPS
32×1	33,554,432	32.022	33.53
64×1	16,777,216	16.039	66.95
128×1	8,388,608	9.710	110.59
256×1	**4,194,304**	**9.685**	**110.87**

2 Problem 2: Matrix Addition

2.1 Thread Index Calculations

1D Configuration A 16384×1 grid of blocks was launched, with each block containing 1024×1 threads. The 2D matrix indices (i, j) are first flattened into a linear index

$$t = iN + j.$$

Thread and block indices are computed as

$$\text{threadIdx.x} = t \bmod 1024, \qquad \text{blockIdx.x} = \left\lfloor \frac{t}{1024} \right\rfloor.$$

Thread $(iN + j) \bmod 1024$ inside block $\lfloor (iN + j)/1024 \rfloor$ processes element (i, j).

2D Configuration A 4×4 grid of blocks was launched, with each block containing 32×32 threads. Thread and block indices directly correspond to matrix coordinates:

$$\text{threadIdx.y} = i \bmod 32, \quad \text{threadIdx.x} = j \bmod 32,$$

$$\text{blockIdx.y} = \left\lfloor \frac{i}{32} \right\rfloor, \quad \text{blockIdx.x} = \left\lfloor \frac{j}{32} \right\rfloor.$$

Thread $(i \bmod 32,\ j \bmod 32)$ inside block $(\lfloor i/32 \rfloor,\ \lfloor j/32 \rfloor)$ directly computes element (i, j).

2.2 Performance Results

Table 2: Matrix Addition Performance on A100

Configuration	Time (ms)	GFLOPS
1D	**0.62054**	**108.15**
2D	0.64000	104.86

2.3 Performance Comparison

Both kernels are global memory bound, with prior profiling via NSight on a RTX3060 GPU revealing close to 80% of warp stalls on both kernels due to the floating point add instruction waiting on the two loads from global memory, and the DRAM and L2 cache very close to the memory roofline. 2D blocks deal with multiple rows, leading to poorer cache locality and more wavefronts (twice as much as 1D in RTX3060), resulting in less performance.

3 Problem 3: Matrix Multiplication

3.1 Thread Mapping

Each thread computes one output element:

$$C_{ij} = \sum_{k=0}^{K-1} A_{ik} B_{kj}$$

$$i = blockIdx.y \cdot blockDim.y + threadIdx.y, \quad j = blockIdx.x \cdot blockDim.x + threadIdx.x$$

3.2 Inner Product Computation

The kernel implements a tiled double-buffered-shared-memory matrix multiplication with vectorised global loads via registers. Each thread block computes a $blockDim_y \times blockDim_x$ output tile. Tiles are processed sequentially. Initially the first tile is preloaded. For each subsequent iteration:

- **Barrier.** `__syncthreads()` ensures the tile is fully resident before computation.
- **Preload.** Threads cooperatively stage the next A and B tiles from global memory into shared memory. Each thread performs vectorised `float4` loads directly into registers and writes to shared memory, giving coalesced accesses and reducing instruction count.

  A tiles consist of $blockDim_y \times k_{\text{step}}$ elements, and B tiles consist of $k_{\text{step}} \times blockDim_x$ elements.
- **Compute.** The tile is consumed entirely from shared memory. Each thread maintains 8 register accumulators and performs an unrolled inner product. The loop is unrolled in chunks of 8, resulting in $k_{\text{step}}/8$ iterations.

After all tiles are processed, accumulators are written directly back to global memory via register stores.

Unrolling and having 8 accumulators help increase instruction level parallelisation, and the double buffer and vectorised loads help to hide latency and reduce the number of load-compute cycles (and thus barriers).

3.3 Performance Results

Table 3: Matrix Multiplication Performance on A100 GPU

Block	Threads	Time (ms)	GFLOPS
32×32	1024	**199.014**	**5524.78**
16×16	256	223.813	4912.64
16×32	512	206.262	5330.65
32×16	512	215.956	5091.36
16×64	1024	201.561	5454.98
64×16	1024	213.578	5148.06
8×8	64	340.737	3226.86
8×16	128	303.304	3625.12
16×8	128	265.814	4136.39
8×32	256	283.010	3885.06
32×8	256	255.838	4297.68
8×64	512	273.360	4022.21
64×8	512	252.268	4358.51
8×128	1024	275.979	3984.04
128×8	1024	253.722	4333.54

Block	Threads	Time (ms)	GFLOPS
4×4	16	820.832	1339.51
4×8	32	470.663	2336.09
8×4	32	404.662	2717.11
4×16	64	414.936	2649.83
16×4	64	362.753	3031.02
4×32	128	375.466	2928.39
32×4	128	340.842	3225.86
4×64	256	354.327	3103.10
64×4	256	335.160	3280.55
4×128	512	351.865	3124.81
128×4	512	332.128	3310.50
4×256	1024	364.649	3015.26
256×4	1024	337.199	3260.72

3.4 Performance Comparison

Prior profiling with Nsight Compute on a RTX3060 GPU indicated that the kernel is shared-memory bound. Approximately 68% of warp stall cycles are attributed to the Memory Input/Output (MIO) pipelines, which service shared-memory operations. In contrast, DRAM utilisation remains negligible and SM throughput exceeds 90%, indicating that performance is limited by on-chip memory bandwidth rather than global memory or arithmetic throughput.

$k_{\text{step}} = \min(\text{blockDim.x}, \text{blockDim.y})$ contributes the most to performance. Configurations with $\min(\text{blockDim}) \geq 16$ achieve around 5 TFLOPS and beyond, while reducing this value to 8 lowers performance to approximately 3-4 TFLOPS, and to around 1-3 TFLOPS at 4. This is because k_{step} determines tile size. As each thread always works on one output arithmetic intensity is capped at 2 float loads and 1 float store per 2 float operations but larger tile sizes reduce the frequency of shared-memory loads and barriers, amortise shared-memory accesses and improve latency hiding which directly impact the shared-memory bottleneck.

For $\min(blockDim) < 16$ blocks with $blockDim_y < blockDim_x$ perform better than the reverse. This is due to poorer L1 cache locality. When $blockDim_x < 32$ each warp consists of multiple rows of threads, splitting global loads across disjoint memory regions. This effect is stronger on the RTX 3060, with all block sizes regardless of $\min(blockDim) < 16$ experiencing lower effective bandwidth due to reduced cache efficiency.

However blocks with $\min(blockDim) \geq 16$ experience a reverse trend which is not reflected on the RTX3060. This could be because A100 has a much larger L2 cache and stronger cache/memory subsystem, reducing the penalty of L1 misses. With lower penalties, multiple rows per warp adds to performance as shared memory accesses during calculation have fewer conflicts, and thus greater instruction level parallelism.

We also see that the larger the blocks the better the performance. This is due to large blocks having more warps, allowing the warp schedule to hide warp stall latency.