Assignment 2 Submission Structure

Submission Requirements

PDF File

- **Name**: `DD2360HT25_HW2_Group2.pdf`
- **Content**:
    - Each group member's contributions
    - Answers to each exercise

ZIP File Structure

The zip file should contain code for each programming question in separate folders:

```
assignment1/
├── Q[1]/
│   ├── hw2_histogram_template.cu
│   ├── generate_histogram_data.py
│   ├── plot_histograms.py
│   ├── Makefile
│   └── README.md
├── Q[2]/
│   ├── hw2_reduction_template.cu
│   ├── generate_timing_data.py
│   ├── plot_timing.py
│   ├── Makefile
│   └── README.md
└── Q[3]/
    ├── vecMult.cu
    ├── batch_test.sh
    ├── plot.py
    ├── Makefile
    └── README.md
```

Group Member Contribution

- Shitong Guo: 50%, Q[1], Q[2]
- Jinye Gong: 50% Q[2], Q[3]

Q[1] – Shared Memory and Atomics (Histogram)

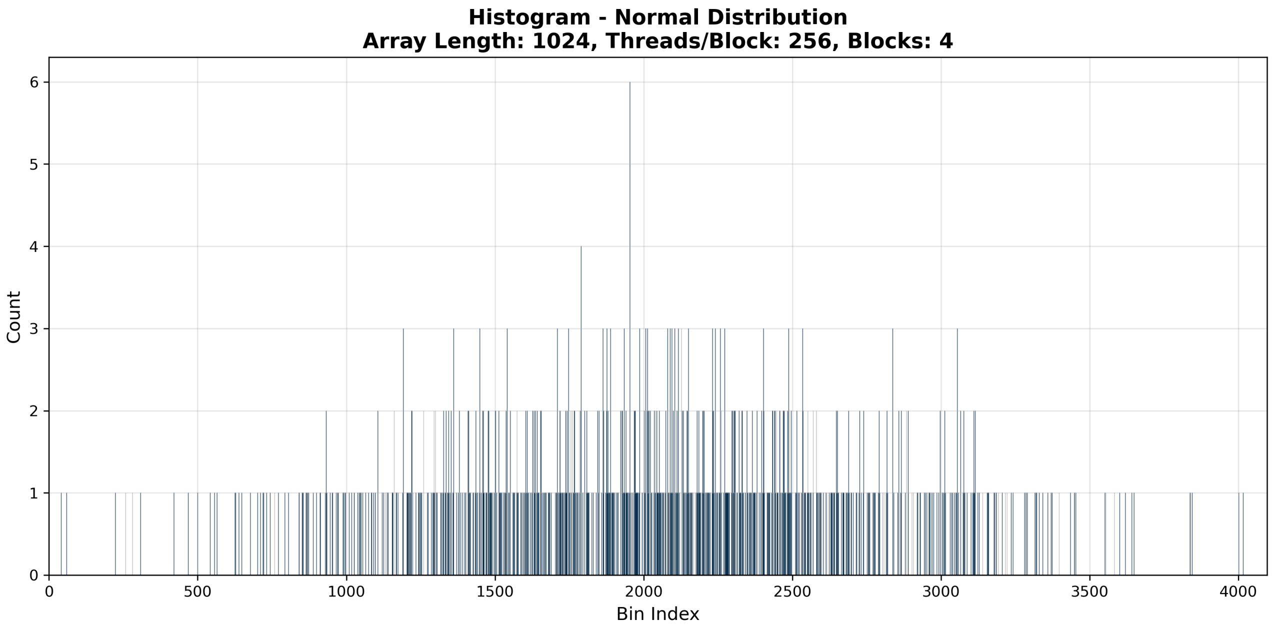
- **Optimizations attempted & impact**
    1. Per-block shared-memory histogram + global merge: moved most atomic traffic to fast shared memory → large speedup.
    2. Parallel shared-memory initialization (strided loop) → removed serial zeroing cost.
    3. Conditional global atomics (`if (s_bins[i] > 0)`) → avoided needless updates.

4. Two-kernel design (histogram + convert) → kept logic simple; neutral/slight gain.

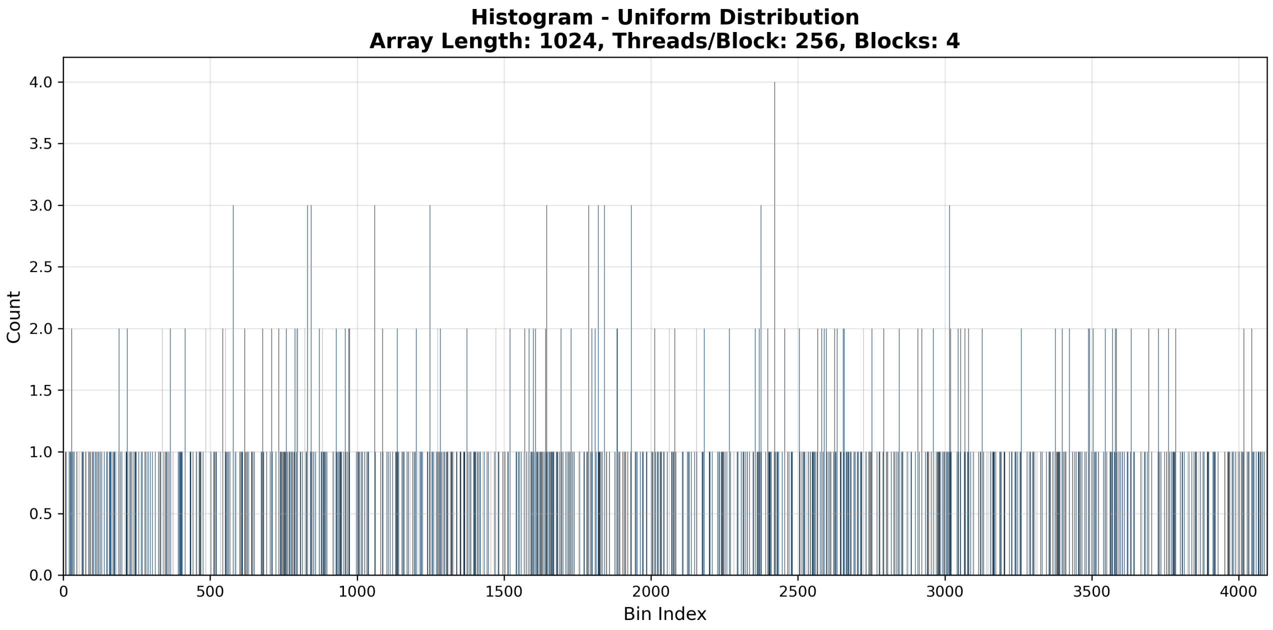
- **Optimizations kept**
  Shared-memory privatization + tree-style reduction inside a block, parallel zeroing, conditional merge. They gave the best trade-off between performance and implementation effort.
- **Global memory traffic**
  - Define the variables:
    - N: `inputLength`
    - M: `NUM_BINS` (4096)
    - G: Grid Size, $\lceil N/256 \rceil$
  - Total global reads: $N + (G \times M) + M$
  - Total global writes: $(G \times M) + M$
- **Explanation:**
  1. **Input Reading (Read):** inside `histogram_kernel`, every thread reads `input[idx]`.
     - Count: N reads.
  2. **Histogram Merge (Read-Modify-Write):** At the end of `histogram_kernel`, threads update the global `bins`. An `atomicAdd` technically performs a Read, a specific arithmetic operation, and a Write.
     - The loop runs for every bin M. Every block G performs this.
     - The code checks `if (s_bins[i] > 0)`. In a worst-case scenario (or uniform distribution), most bins will have data.
     - Count: $G \times M$ reads and $G \times M$ writes.
  3. **Saturation Kernel (Read & Write):** The `convert_kernel` runs once over the final bins.
     - It reads `bins[idx]` (M reads).
     - It writes `bins[idx]` (M writes, assuming it writes back even if not changed, or fewer if optimized, but structurally it accesses the memory).
- **Atomic operations**
  - Shared memory: N .
  - Global memory: ≤ $G \times M'$, M' is the number of bins with a non-zero count in that block.
  - Total: $G \times M' + N$
- **Shared memory usage**
  `NUM_BINS * sizeof(unsigned int) = 4096 * 4 = 16 KB` per block (`extern __shared__`), which limits blocks/SM but is required to keep per-bin counts.
- **Effect of value distribution**
  - **Uniform Distribution (Type 0):**
    - Values are spread roughly equally across all 4096 bins.
    - **Contention:** Low. The probability of multiple threads in the same "warp" (group of 32 threads executing simultaneously) trying to write to the exact same bin index is statistically low. The atomic operations run mostly in parallel.

    Normal Distribution (Type 1):

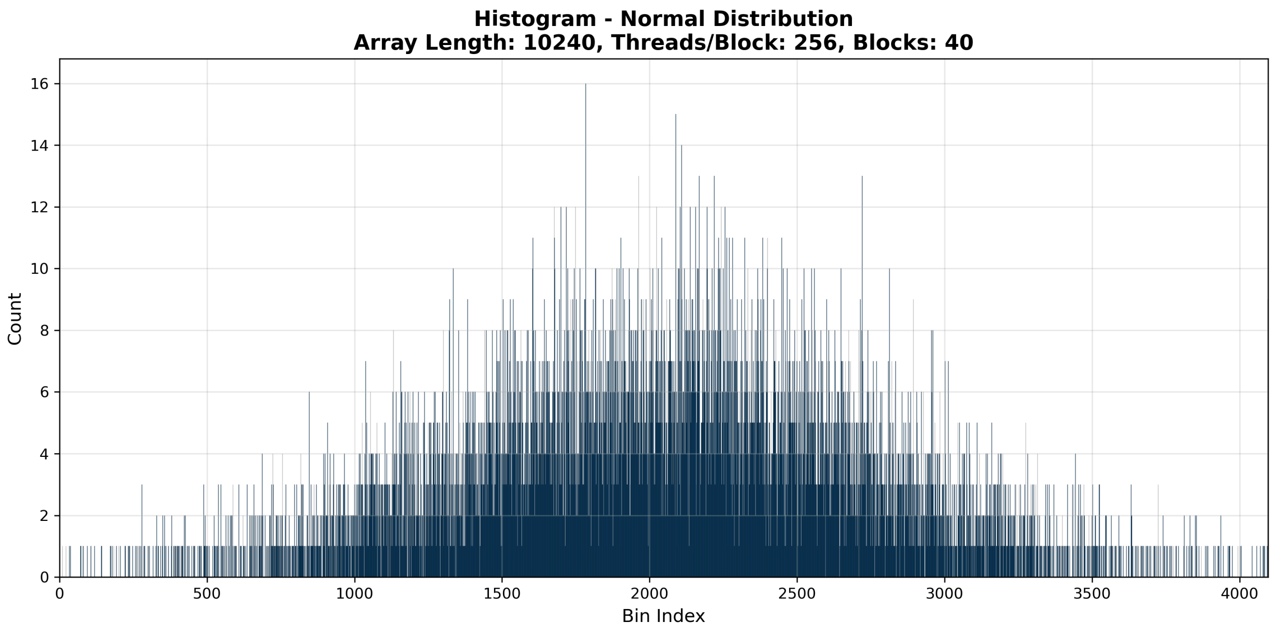
        - Values are clustered heavily around the mean (bin 2048).
        - **Contention:** Very High. Many threads within the same warp will likely hold values that map to the same few central bins.
        - **Result:** Hardware serialization. The GPU hardware must detect that threads are targeting the same memory address and force them to execute one after another (sequentially) rather than in parallel. This creates a bottleneck and will make the Normal distribution execution significantly slower than the Uniform distribution.
- **Histogram plots (array lengths 1,024 → 1,024,000)**
    - **1024 normal**



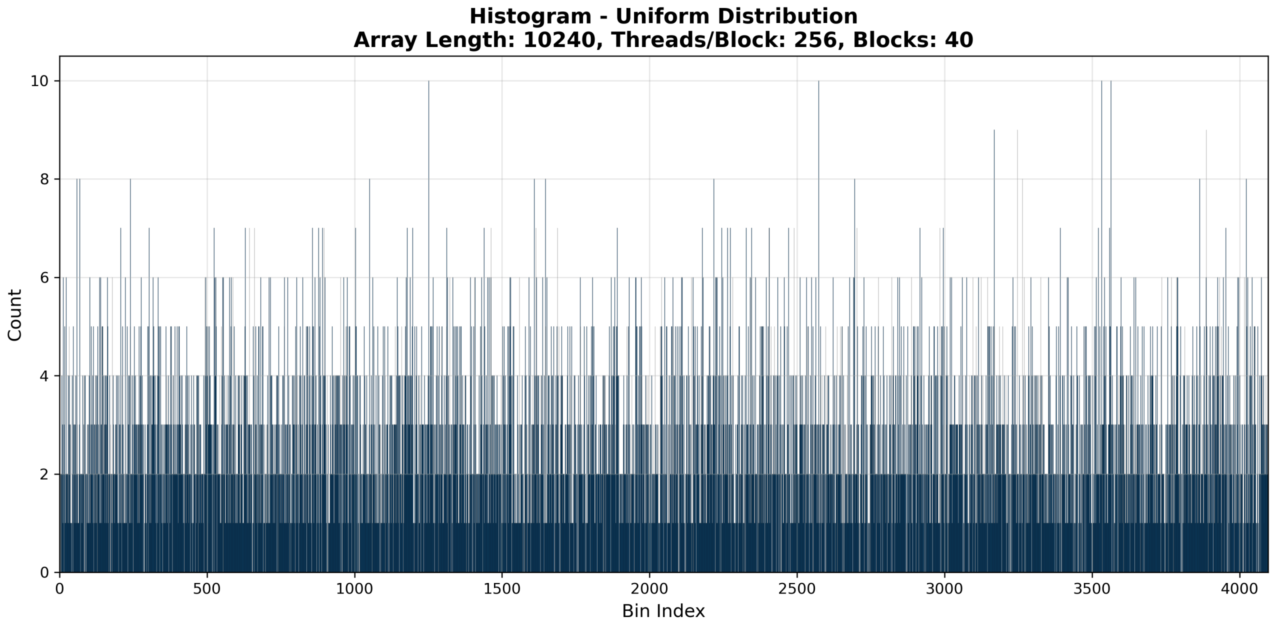
    - **1024 uniform**



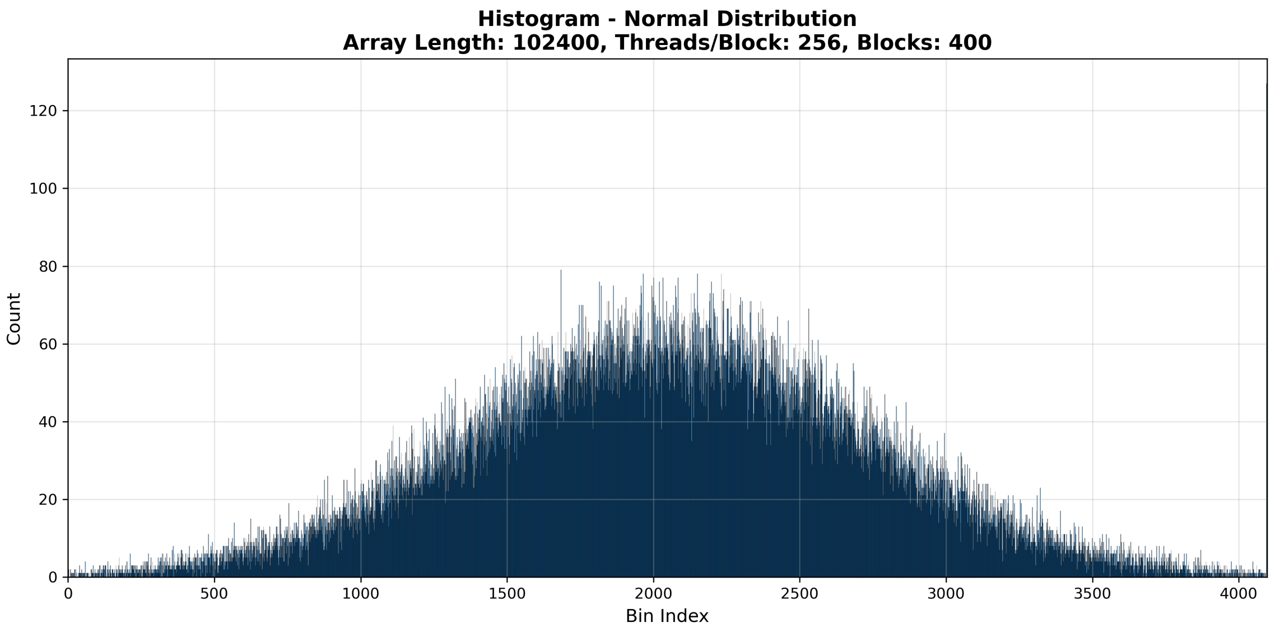
    - **10240 normal**



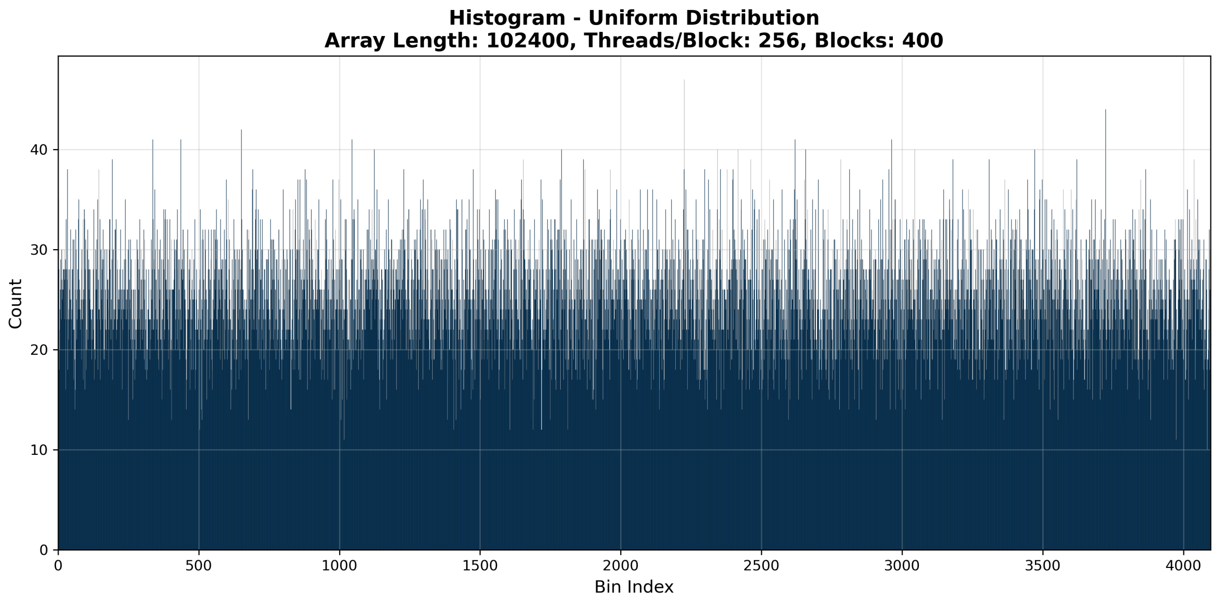
- **10240 uniform**



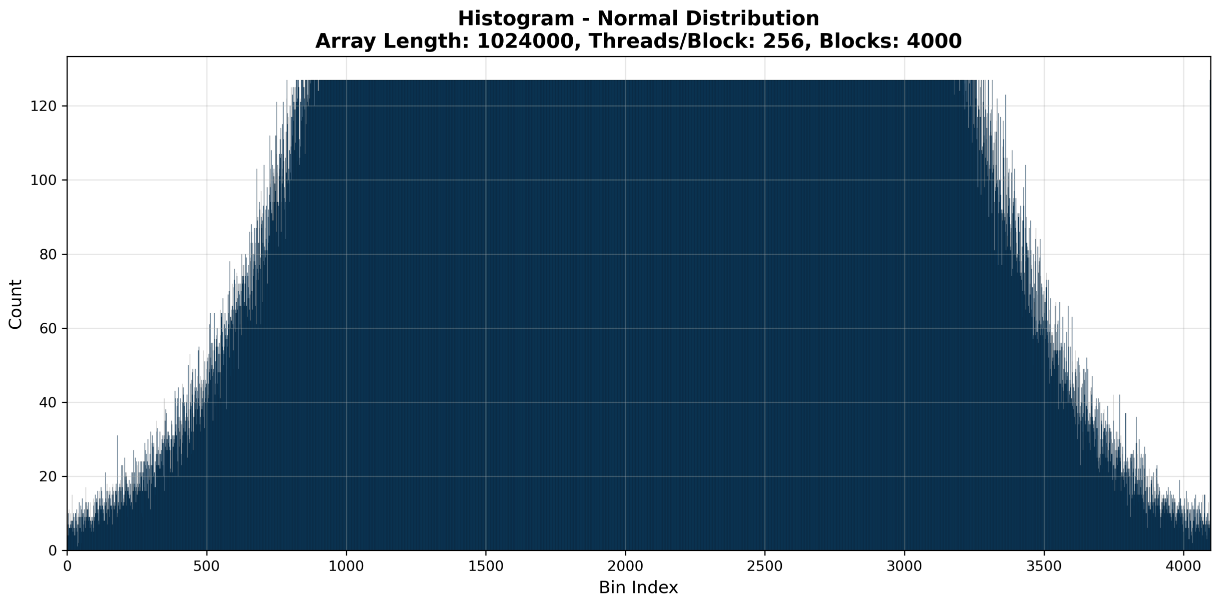
- **102400 normal**



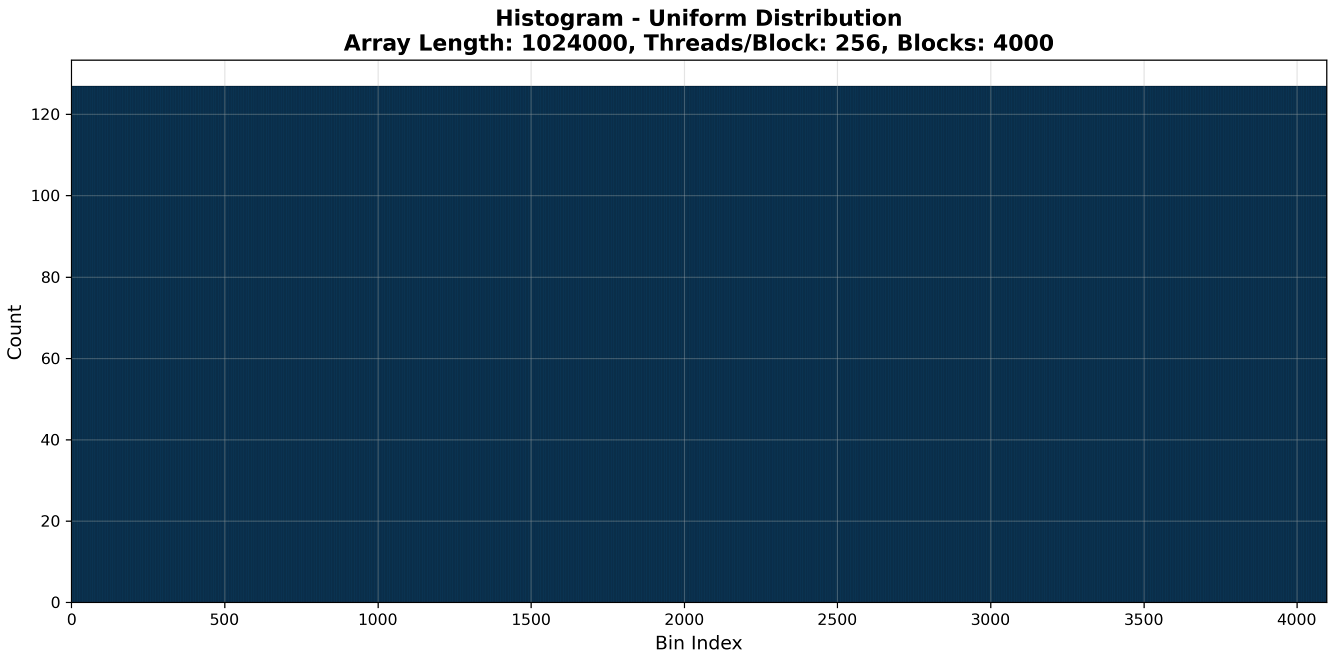
- 102400 uniform



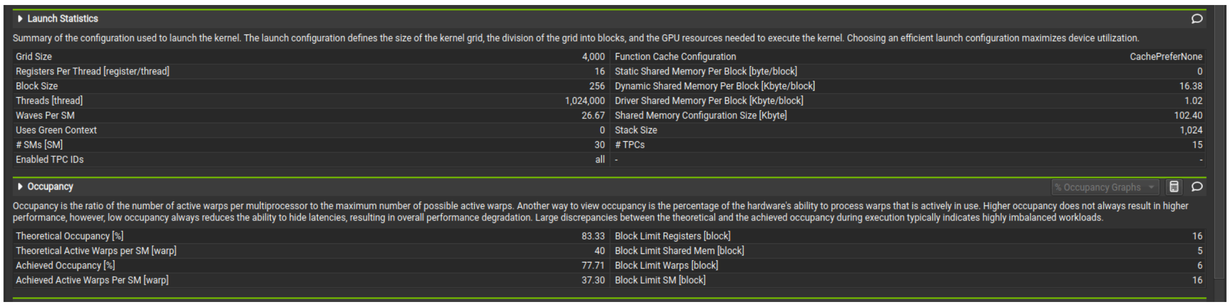
- 1024000 normal



- **1024000 uniform**



- **Speedup observation**
  GPU beats CPU once N is large enough. For small N the launch/transfer overhead dominates and CPU is competitive.
- **Profiling (N = 1,024,000)**
  - Shared memory size: 16 KB per block.
  - Achieved Occupancy: 77.71%



Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	4,000	Function Cache Configuration	CachePreferNone
Registers Per Thread [register/thread]	16	Static Shared Memory Per Block [byte/block]	0
Block Size	256	Dynamic Shared Memory Per Block [Kbyte/block]	16.38
Threads [thread]	1,024,000	Driver Shared Memory Per Block [Kbyte/block]	1.02
Waves Per SM	26.67	Shared Memory Configuration Size [Kbyte]	102.40
Uses Green Context	0	Stack Size	1,024
# SMs [SM]	30	# TPCs	15
Enabled TPC IDs	all	-	-

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	83.33	Block Limit Registers [block]	16
Theoretical Active Warps per SM [warp]	40	Block Limit Shared Mem [block]	5
Achieved Occupancy [%]	77.71	Block Limit Warps [block]	6
Achieved Active Warps Per SM [warp]	37.30	Block Limit SM [block]	16

Optimization to try: Warp-Aggregated Atomics

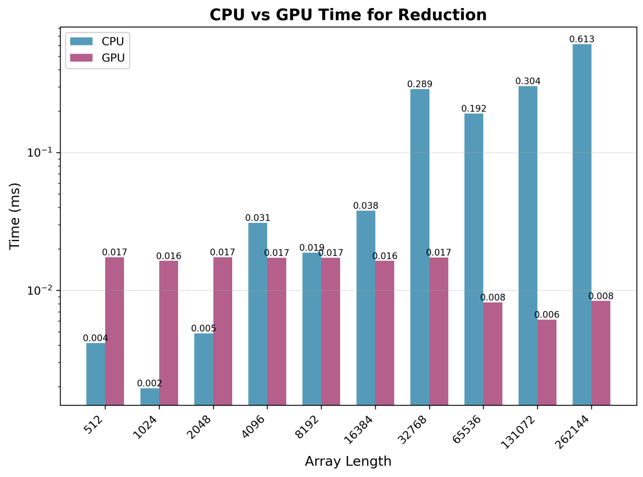
Description: Currently, the shared memory implementation suffers from high contention (serialization) when the input follows a Normal distribution. This is because multiple threads in the same warp often try to increment the same bin index (e.g., around the mean value), causing the hardware to serialize these `atomicAdd` operations.

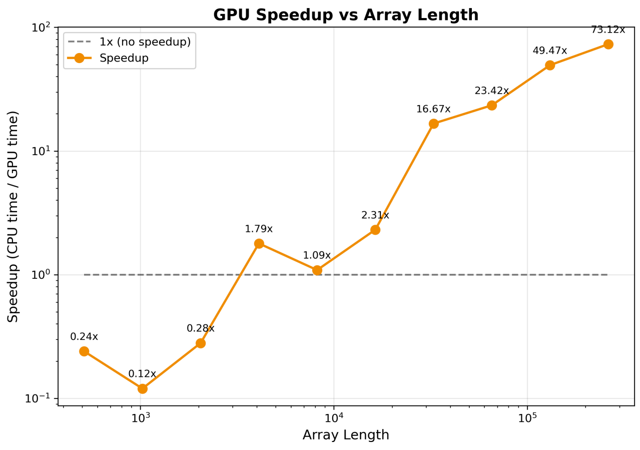
Q[2] – Reduction

- **Optimizations attempted & impact**
  1. Block-level shared-memory tree reduction → reduces global atomics from N to numBlocks (major win).
  2. Coalesced loads + zero padding for out-of-bounds threads → keeps memory bandwidth high.

3. Single atomicAdd per block → minimal contention.

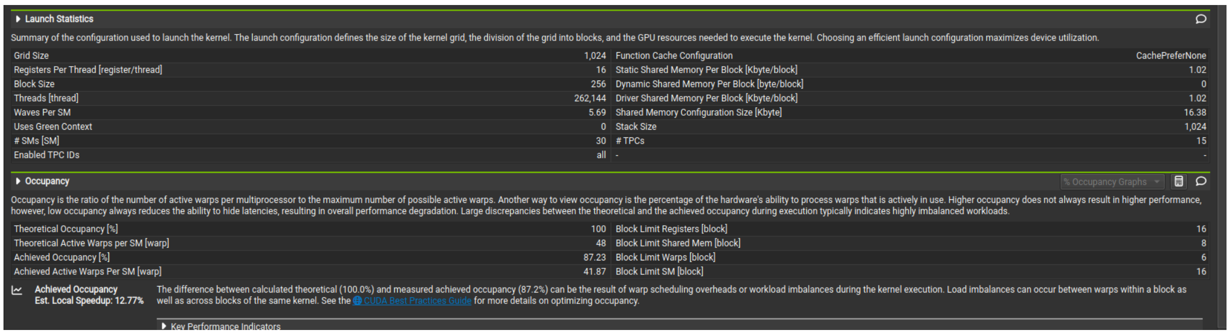
- **Optimizations kept**
  Shared-memory reduction with tree pattern and one atomic per block offered best balance; further tweaks (warp shuffles, multi-element threads) noted for future work.
- **Global memory traffic**
  Reads: N (each float once). Writes: 1 memset + `numBlocks` atomic additions to the global accumulator.
- **Atomic operations**
  `NumBlocks` $\lceil N/256 \rceil$
  - **Explanation**:
    - The reduction is done in two stages:
    - **Intra-block:** Done via standard arithmetic in shared memory (no atomics needed due to `__syncthreads` synchronization).
    - **Inter-block:** Done via `atomicAdd`. Since we cannot synchronize across different blocks, we use an atomic operation to safely accumulate the partial sums calculated by each block into the final global variable.
- **Shared memory usage**
  `THREADS_PER_BLOCK * sizeof(float) = 256 * 4 = 1 KB` per block, allowing high SM residency.
  - **Explanation:**
    - We declare `__shared__ float sdata[THREADS_PER_BLOCK];`.
    - `THREADS_PER_BLOCK` is defined as 256.
    - A `float` is 4 bytes.
    - Calculation: $256 \times 4 \text{ bytes} = 1024 \text{ bytes}$.
- **Timing sweep (512 → 262,144)**





  - Observations: GPU loses for tiny N (overhead bound), breaks even near ~8K, and achieves double-digit speedups (10–70×) once arithmetic dominates.
- **Speedup explanation**
  Large N fully utilizes thousands of CUDA cores and amortizes PCIe/launch overhead; CPU remains sequential and becomes bandwidth bound.
- **Profiling (N = 262,144)**
  - Shared memory configuration: 1 KB per block.

- Achieved Occupancy: 87.23%



Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	1,024	Function Cache Configuration	CachePreferNone
Registers Per Thread [register/thread]	16	Static Shared Memory Per Block [Kbyte/block]	1.02
Block Size	256	Dynamic Shared Memory Per Block [byte/block]	0
Threads [thread]	262,144	Driver Shared Memory Per Block [Kbyte/block]	1.02
Waves Per SM	5.69	Shared Memory Configuration Size [Kbyte]	16.38
Uses Green Context	0	Stack Size	1,024
# SMs [SM]	30	# TPCs	15
Enabled TPC IDs	all	-	-

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	100	Block Limit Registers [block]	16
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]	8
Achieved Occupancy [%]	87.23	Block Limit Warps [block]	6
Achieved Active Warps Per SM [warp]	41.87	Block Limit SM [block]	16

Achieved Occupancy
Est. Local Speedup: 12.77%
The difference between calculated theoretical (100.0%) and measured achieved occupancy (87.2%) can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel. See the CUDA Best Practices Guide for more details on optimizing occupancy.

Optimization to try: Warp Shuffle Reduction (Register-Level Operations)

Description: The current tree-based reduction relies on shared memory for inter-thread communication. However, reading and writing to shared memory requires latency, and `__syncthreads()` barriers are necessary to prevent race conditions.

Q[3] Tiled Matrix Multiplication

- Questions:
  1. How many global memory reads are performed by the tiled_gemm() kernel? Compare it with that of the gemm() version.

     For the naive **gemm()** kernel, each element of **C** reads **K** elements from **A** and **K** elements from B, Since there are **M × N** elements in **C**, the total number of global memory reads is

     $$R_{\text{gemm}} \approx 2MNK$$

     For the **tiled_gemm()** kerne:

     In each K-tile iteration, every block reads from global memory **tileX × tileY** elements of A and **tileX × tileY** elements of B, for a total of **2 · tileX · tileY** global memory loads per tile. Since the **K** dimension is partitioned into $\frac{K}{\text{tileX}}$ tiles (assuming **K** is divisible by **tileX**), there are $\frac{K}{\text{tileX}}$ K-tiles in total.

     $$R_{\text{block}} = 2\,\text{tileX}\,\text{tileY} \cdot \frac{K}{\text{tileX}} = 2\,\text{tileY}\,K$$

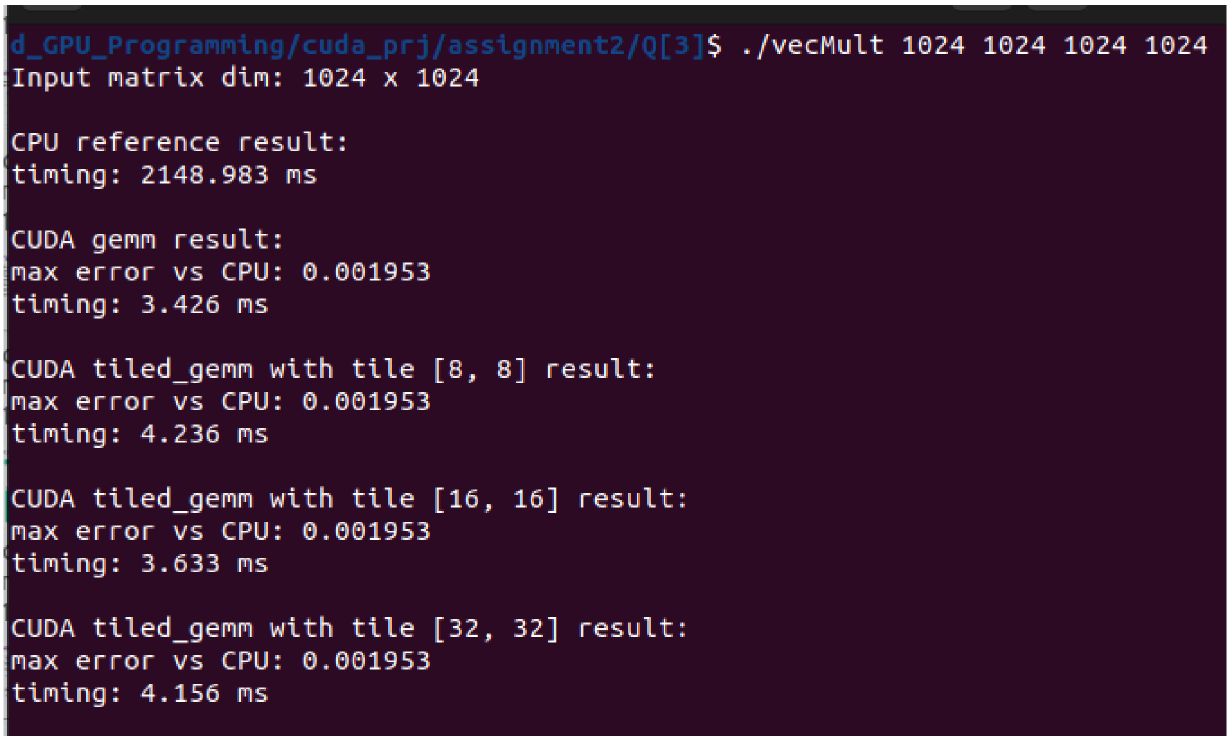
     Number of blocks in the entire grid:

     $$\text{blocks} = \frac{M}{\text{tileY}} \cdot \frac{N}{\text{tileX}}$$

     Total read count:

     $$\begin{aligned} R_{\text{tiled}} &= \frac{M}{\text{tileY}} \cdot \frac{N}{\text{tileX}} \cdot R_{\text{block}} \\ &= \frac{M}{\text{tileY}} \cdot \frac{N}{\text{tileX}} \cdot (2\,\text{tileY}\,K) \\ &= 2MNK \cdot \frac{1}{\text{tileX}} \end{aligned}$$

  2. Run with your program with a matrix A of (1024x1024) and B of (1024x1024), select 3 tile sizes for testing. Include the screenshot of your output. Explain the results.

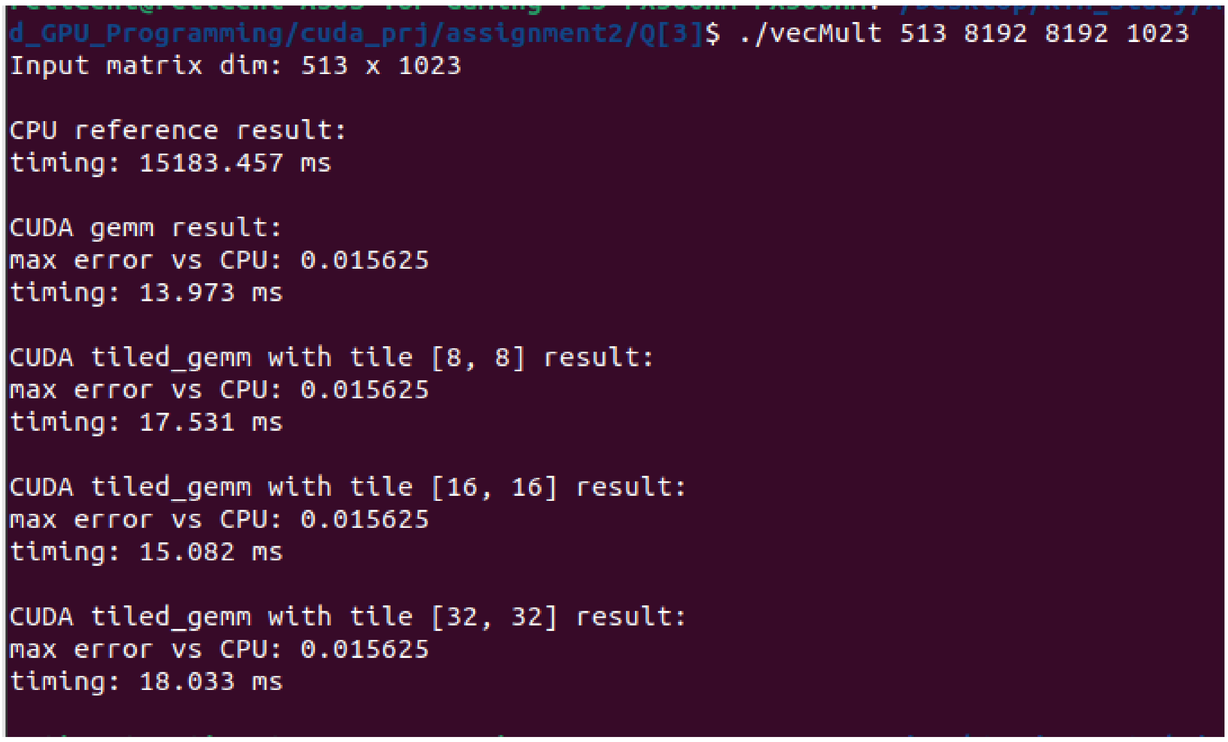


```
d_GPU_Programming/cuda_prj/assignment2/Q[3]$ ./vecMult 1024 1024 1024 1024
Input matrix dim: 1024 x 1024

CPU reference result:
timing: 2148.983 ms

CUDA gemm result:
max error vs CPU: 0.001953
timing: 3.426 ms

CUDA tiled_gemm with tile [8, 8] result:
max error vs CPU: 0.001953
timing: 4.236 ms

CUDA tiled_gemm with tile [16, 16] result:
max error vs CPU: 0.001953
timing: 3.633 ms

CUDA tiled_gemm with tile [32, 32] result:
max error vs CPU: 0.001953
timing: 4.156 ms
```

For the 1024×1024 case, all CUDA kernels match the CPU reference with a maximum error of about 2×10^{-3}, so they are numerically correct. The naive CUDA `gemm` runs in 3.426 ms compared to 2148.983 ms on the CPU, showing a large speedup from massive parallelism.

The tiled kernels with tile sizes 8×8, 16×16 and 32×32 take 4.236 ms, 3.633 ms and 4.156 ms, respectively, and are slightly slower than the naive version. Although tiling reduces global memory reads by reusing tiles of A and B in shared memory, for this problem size the naive kernel already benefits from cache and coalesced accesses, so the extra shared-memory and synchronization overhead dominates. Among the tiled versions, 16×16 gives the best trade-off between data reuse and resource usage.

3. Run with your program with a matrix A of (513x8192) and B of (8192x1023), select 3 tile sizes for testing. Include the screenshot of your output. Explain the results.



```
d_GPU_Programming/cuda_prj/assignment2/Q[3]$ ./vecMult 513 8192 8192 1023
Input matrix dim: 513 x 1023

CPU reference result:
timing: 15183.457 ms

CUDA gemm result:
max error vs CPU: 0.015625
timing: 13.973 ms

CUDA tiled_gemm with tile [8, 8] result:
max error vs CPU: 0.015625
timing: 17.531 ms

CUDA tiled_gemm with tile [16, 16] result:
max error vs CPU: 0.015625
timing: 15.082 ms

CUDA tiled_gemm with tile [32, 32] result:
max error vs CPU: 0.015625
timing: 18.033 ms
```

For the 513×8192 × 8192×1023 case, all CUDA kernels match the CPU reference with a maximum error of about 1.56×10^{-2}. The CPU takes about 15.2 s, while the naive CUDA `gemm` runs in 14 ms, giving a speedup of roughly three orders of magnitude.

The tiled kernels with 8×8, 16×16 and 32×32 tiles are slightly slower (17.5 ms, 15.1 ms and 18.0 ms) than the naive kernel. Although tiling reduces the number of global memory reads by reusing tiles of A and B in shared memory, in this problem the naive kernel already benefits from cache and coalesced accesses, and the extra overhead of loading tiles and synchronizing threads dominates. Among the tiled configurations, 16×16 provides the best trade-off between data reuse, occupancy and per-block resource usage.

4. Profile your program in 2 and 3. Report Achieved Occupancy and Shared Memory usage.

For the program in 2:

1024×1024 case: Achieved Occupancy and Shared Memory usage

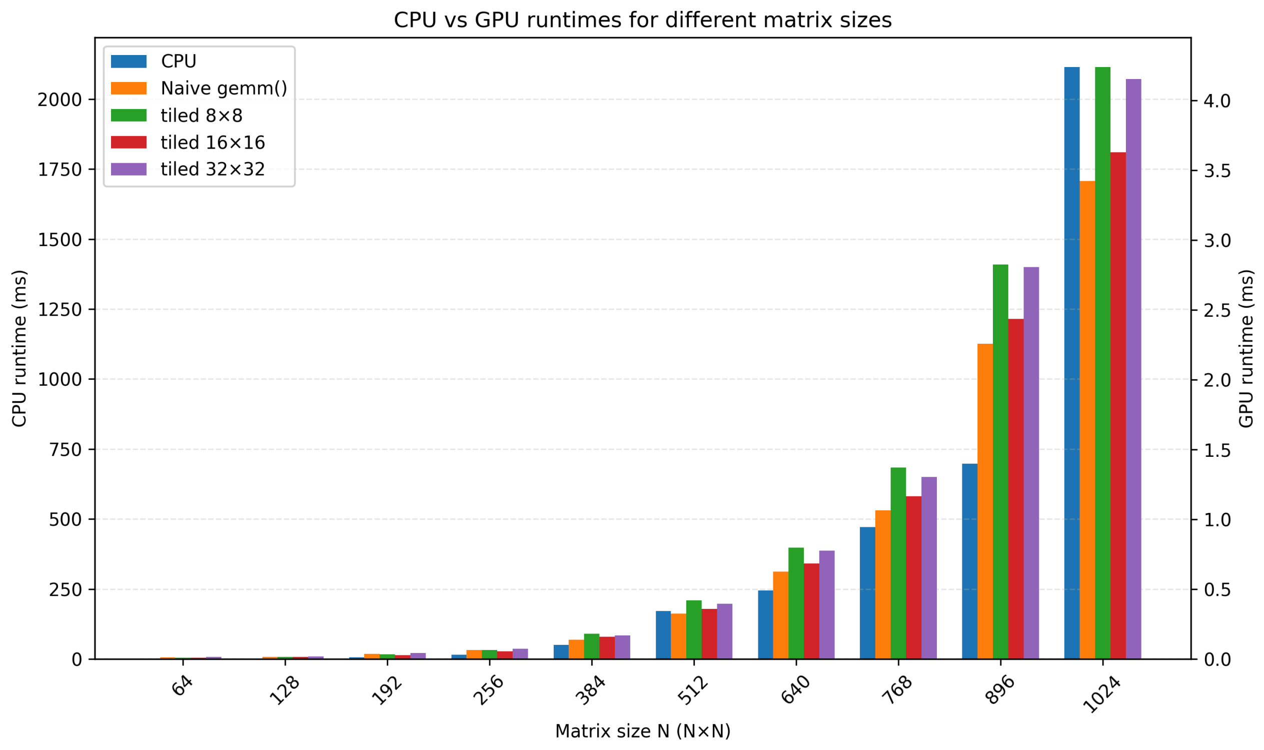
Kernel	Tile size	Block size (threads)	Dynamic shared mem / block	Achieved Occupancy
matrixMultGPU (naive)	–	1024	0 B	~66.5%
tiled_gemm	8 × 8	64	512 B	~66.1%
tiled_gemm	16 × 16	256	~2 KB	~98%
tiled_gemm	32 × 32	1024	~8 KB	~66.7%

For the program in 3:

513×8192 · 8192×1023 case: Achieved Occupancy and Shared Memory usage

Kernel	Tile size	Block size (threads)	Dynamic shared mem / block	Achieved Occupancy
matrixMultGPU (naive)	–	1024	0 B	~65.4%
tiled_gemm	8 × 8	64	512 B	~65.4%
tiled_gemm	16 × 16	256	~2 KB	~96.8%
tiled_gemm	32 × 32	1024	~8 KB	~66.7%

5. Run your program with 8-10 different matrix sizes that increases from small size to larger sizes. Plot a bar chart comparing the runtime of the CPU version, the gemm(), and the three tiled gemm() kernels of these problems. Explain your results.



The bar chart shows the runtime of the CPU implementation, the naive CUDA `gemm()` kernel, and three tiled `tiled_gemm()` kernels (8×8, 16×16, 32×32 tiles) for increasing matrix sizes (N\times N). The CPU times are plotted on the left y-axis, while the GPU times are plotted on the right y-axis because they differ by several orders of magnitude.

As (N) increases, the CPU runtime grows very quickly (approximately (O(N^3))) and reaches the order of seconds for the largest matrices. In contrast, all CUDA kernels remain in the millisecond range even for the largest sizes, yielding a speedup of roughly two to three orders of magnitude compared to the CPU version.

Among the GPU kernels, the naïve `gemm()` and the tiled versions all scale similarly with (N), but their absolute runtimes differ slightly. For small matrices the differences are minor, and the overhead of loading tiles into shared memory and calling `__syncthreads()` can make the tiled versions slightly slower than the naïve kernel. For larger matrices, the 16×16 tiled configuration is usually the fastest among the tiled variants, while 8×8 provides limited data reuse along the K dimension and 32×32 suffers from higher per-block resource usage and synchronization overhead. Overall, the naïve kernel already has coalesced global memory accesses and benefits from the cache hierarchy, so tiling does not bring a dramatic speedup in this implementation; in some sizes it is slightly slower because the additional shared-memory overhead compensates for the reduced global memory traffic.