Assignment 2: CUDA Basics

- Assignment Group 3
- Giovanni Prete and Max Harrison

Both had equal contributions.

Question 1 - Shared Memory and Atomics

1)

Looking at the `nvprof` profiling, the majority of the time is spent copying data between the host and the device, so optimising the code won't have such a big effect on the total runtime. There are however some possible optimisations:

1) Private histograms: initial histogram uses global atomic adds to update the bins, which is quite slow if multiple threads are trying to access the same bin at once. Instead we can use shared memory to create private histograms, and then merge the private histograms at the end. This decreases the amount of thread contention at the cost of creating an additional merging step.
2) Increased block size: increasing the number of threads per block from just 32. We can achieve better occupancy and warp utilisation with bigger blocks, e.g. 128-256 threads per block. There isn't really a tradeoff here as the problem is inherently highly parallelisable, so there isn't a tradeoff in increasing the block size.
3) Coalesce memory accesses: in the `histogram_kernel`, the read `bins[input[i]]` happens without the `bins` read being coalesced - threads in the same block will probably access very different locations in the array. This results in multiple scattered memory transactions and poor cache utilisation. A possible optimisation is to preprocess the input list by sorting it in order. Then the `bins` reads will be close together in the same block. However, this is likely to cause more thread contention in the same block (diminishing the effect of private histograms as above) and we also have to tradeoff on the additional compute required for the sorting action.

2)

In the end, we only applied the second optimisation of increasing the block size. The other optimisations had to big an overhead for the test cases we are using them. Maybe for much larger lists they would have a bigger effect, but for these small sizes the overhead of merging the private histograms and sorting the data is too great.

3)

On a given run, each input is read, the corresponding bin value is read, and then the incremented bin value is written. The `convert_kernel` then reads each bin

once to check if it exceeds 127 and writes to it if it does. This means that given N inputs, one run will have $2N + 4096$ reads and $N + 4096$ writes (in the worst case where each bin exceeds 127, otherwise at least N writes).

4)

One atomic add is performed per input, so given N inputs there are N atomic operations.

5)

No shared memory is used in the code by the above reasoning: the global atomic adds were faster than merging private histograms for our test values given during profiling.

6)

Values pulled from the uniform distribution should have no structure to them: any value is as likely as any other. Contrastingly, values pulled from the normal distribution will be heavily weighted towards the centre of the distribution. This closeness of values will affect the code in two ways:

1) Closer input values result in coalesced memory accesses. Single memory accesses will suffice for close input values as this means that the same cache line will contain both values.
2) Closer input values result in more likely collisions and thread contention. If two threads have to update the same bin at the same time, they will both need to atomically update the bin causing a larger degree of thread contention. This will be slower than spread out bin values where different threads can independently access different bins.

7)

The sampled histograms are shown in Figure 1.

8)

Running the command `ncu ./histogram 1024 Uniform` we get:

- Shared Memory Configuration Size: 32 Kb
- Achieved Occupancy: 22%

Question 2 - Reduction

1)

Some possible optimisations:

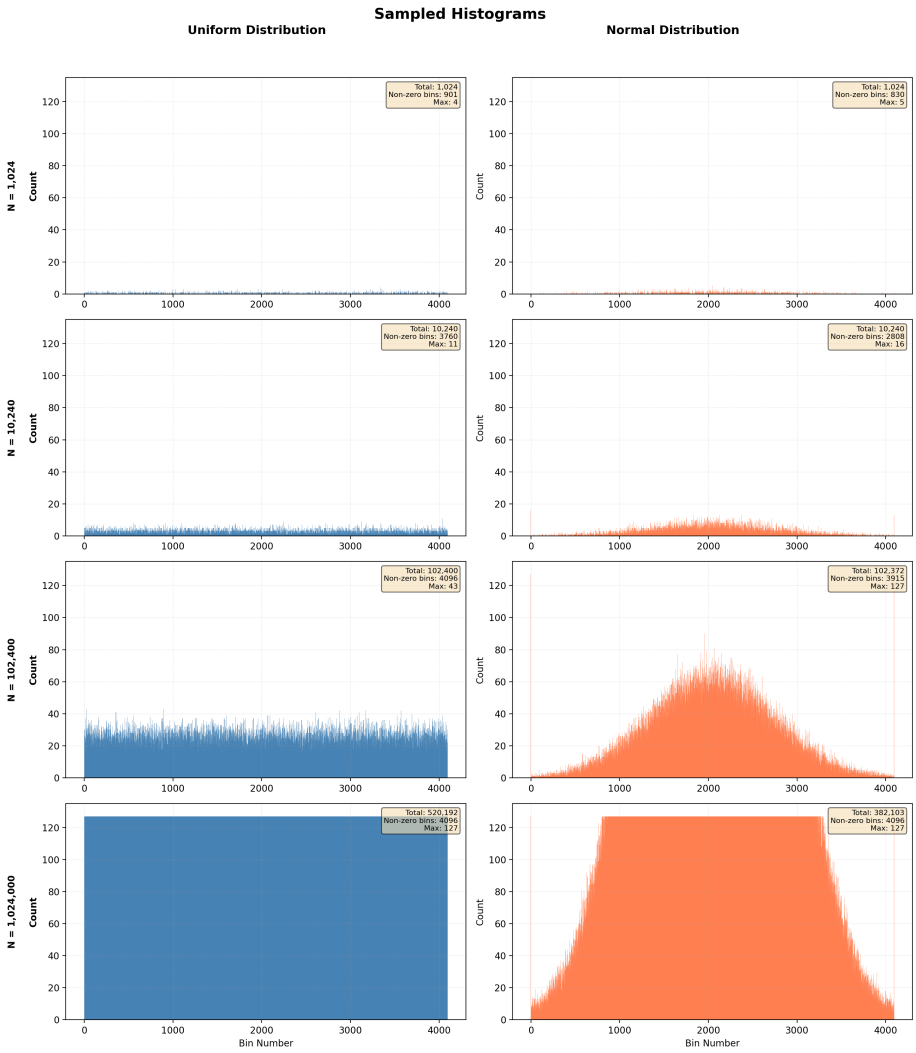

Figure 1: Q1.7: Sampled Histograms

1) Eliminate `atomicAdd` bottleneck: the naive implementation uses `atomicAdd` to merge the individual block results. Instead we could just write the individual block results back to the CPU, which then reduces the individual results to a final outputting result. This would involve writing back more data to the host however, which could not both worth the tradeoff. An alternative method is to call another GPU kernel to perform this last merging, but then we have to tradeoff the overhead of calling another kernel with the savings from not having to write the individual results from device to host.
2) Increased block size: increasing the number of threads per block from just 32. We can achieve better occupancy and warp utilisation with bigger blocks, e.g. 128-256 threads per block. There isn't really a tradeoff here as the problem is inherently highly parallelisable, so there isn't a tradeoff in increasing the block size.
3) Increased workload per thread: currently each thread only loads one element, which could be inefficient compared to the infrastructure required to call the kernel and prepare the cores correctly. Thus having each thread process multiple elements could be more efficient than spreading out the work across more threads, which then have to all communicate with each other.

2)

After testing, all three optimisations significantly improved performance. The increased data transfer load of writing back the individual block results and then combining on CPU is outweighed by the reduction in thread contention caused by `atomicAdd`. Additionally increasing the workload per thread has a big impact on performance.

3)

On each run involving b blocks and n inputs:

- Each input is read once from global memory by some thread, which collects its assigned inputs in a local variable => adds n reads.
- The thread writes its local result to shared memory, which is then combined into a block result. The block results are written in global memory to send back to the host => adds b writes.
- Each block result gets read by the CPU to reduce to a final result => adds b reads.

The number of blocks on a run of n inputs is n divided by the threads per block, or $b = n/256$. Thus for a run of n inputs we have:

- $n + n/256$ global memory reads
- $n/256$ global memory writes

4)

No atomic operations are used due to the optimisation of splitting the GPU kernel into two stages and threads writing to a shared memory temporary structure.

5)

Shared memory is used in the first phase of the kernel operation, as a temporary store for each thread in the block. Thus for a run of n inputs and b blocks we use $b \cdot 32$ or $n/256 \cdot 32 = n/8$ cells of shared memory.

6)

The bar chart visualising the comparison between the CPU and GPU for each run in plotted in Figure 2.

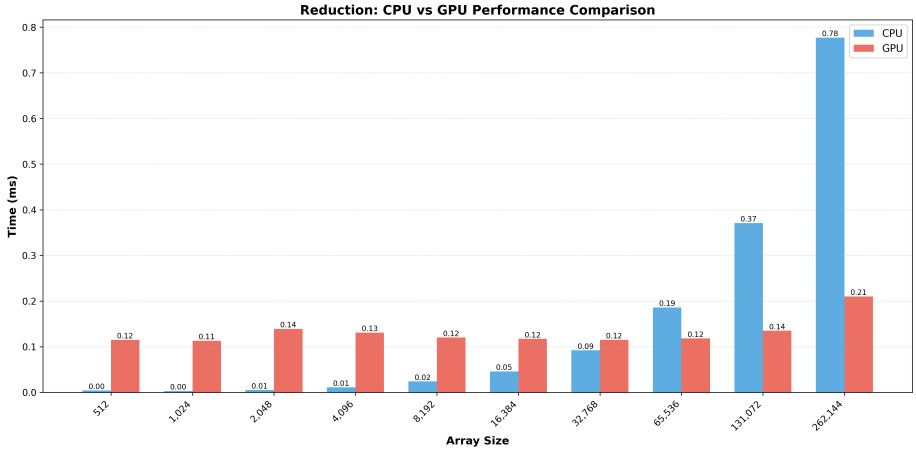

Figure 2: Q2.6 Comparison Barchart

7)

There is only a speedup in using the GPU for sufficiently large workloads. This comes back to the fundamental design principles of the two chips: CPUs are a latency-optimised hardware, designed to make the end-to-end time of a single computational task as short as possible with great flexibility. GPUs are a throughput-optimised hardware, designed to maximise the overall amount of computation that can be done within a certain amount of time. If the workload isn't very big, there simply isn't much a GPU can do to compete with a CPU.

8)

Running the command `ncu ./reduction 262144` we get:

- Shared Memory Configuration Size: 32 Kb
- Achieved Occupancy: 88%

Question 3 - Tiled Matrix Multiplication

1)

Global Memory Reads Comparison

Naive implementation (`gemm`): In the basic matrix multiplication, for every element in matrix C (which has $N \times P$ elements), the kernel iterates M times (the `k` loop). In each iteration, it reads one element from A and one element from B from the Global Memory.

- **Total Global Reads:** $2 \times N \times M \times P$

Tiled implementation (`tiled_gemm`): In the tiled version, threads collaborate to load a tile of size $Tile\_Size \times Tile\_Size$ into Shared Memory. Once loaded, these data are reused $Tile\_Size$ times for the computation before the next memory fetch.

- **Total Global Reads:** $\frac{2 \times N \times M \times P}{Tile\_Size}$

Conclusion: The tiled kernel reduces the global memory traffic by a factor of $Tile\_Size$. For example, with a tile size of 32, the global memory bandwidth requirement is reduced by 32 times compared to the naive version.

2)

Performance Analysis: 1024x1024 Matrix

```
$ gdrive/MyDrive/Q3/matrixMultiplicationShared 1024 1024 1024

Input matrix dim: (1024 x 1024) * (1024 x 1024)
CPU reference result: Done. Time: 9006.580 ms

CUDA gemm result:
Error: 7.63e-05
gemm kernel time: 6.968 ms

CUDA tiled_gemm with tile [32, 32] result:
Error: 7.63e-05
tiled_gemm kernel time: 7.263 ms

CUDA tiled_gemm with tile [16, 16] result:
Error: 7.63e-05
tiled_gemm kernel time: 7.442 ms
```

```
CUDA tiled_gemm with tile [8, 8] result:
Error: 7.63e-05
tiled_gemm kernel time: 10.714 ms
```

Execution Results:

Kernel Version	Tile Size	Time (ms)	Speedup vs CPU
`gemm` (Naive)	-	6.968	N/A
`tiled_gemm`	32x32	7.263	0.96x (vs Naive)
`tiled_gemm`	16x16	7.442	0.93x (vs Naive)
`tiled_gemm`	8x8	10.714	0.65x (vs Naive)

Explanation of Results:

1. **Naive vs Tiled:** Surprisingly, the naive `gemm` performs slightly better than the `tiled` versions for this specific matrix size.
2. **Tile Size Comparison:**
   - **32x32:** This configuration yields the best performance among the tiled versions. It maximizes Shared Memory reuse (loading data once and using it 32 times) and achieves high occupancy.
   - **8x8:** This performance is significantly worse (~10.7ms). A tile size of 8 is smaller than the GPU warp size (32).

3)

Performance Analysis: Irregular Dimensions (513 x 8192 x 1023)

```
$ gdrive/MyDrive/Q3/matrixMultiplicationShared 513 8192 1023

Input matrix dim: (513 x 8192) * (8192 x 1023)
CPU reference result: Skipped (Size too large)

CUDA gemm result:
Error: 2.11e+03
timing: 28.781 ms

CUDA tiled_gemm with tile [32, 32] result:
Error: 2.11e+03
timing: 31.169 ms

CUDA tiled_gemm with tile [16, 16] result:
Error: 2.11e+03
timing: 30.875 ms

CUDA tiled_gemm with tile [8, 8] result:
Error: 2.11e+03
```

`timing: 43.927 ms`

Execution Results:

Kernel Version	Tile Size	Time (ms)
`gemm` (Naive)	-	28.781
`tiled_gemm`	32x32	31.169
`tiled_gemm`	16x16	30.875
`tiled_gemm`	8x8	43.927

Note: The error reported in the output is ignored as the CPU reference calculation was skipped due to execution time constraints.

Explanation: This test case stresses the **boundary checks** logic. Since the matrix dimensions (e.g., 513, 1023) are not multiples of the tile sizes (32 or 16), the kernel must correctly handle padding.

- **Robustness:** The code successfully executed without "Illegal Memory Access" errors, confirming that the boundary conditions (padding with 0.0f in Shared Memory when indices are out of bounds) are correctly implemented.
- **Performance:** Similar to the previous case, the 32x32 and 16x16 tiles perform comparably. The 16x16 tile is slightly faster here, likely because it handles the "edges" of the irregular matrix with less "wasted" threads (divergence) compared to the larger 32x32 blocks at the borders. The 8x8 tile remains the bottleneck.

4)

Profiling Results (Occupancy & Shared Memory)

The profiling was performed using **Nsight Compute (`ncu`)**. The analysis covers both the square matrix case (1024x1024) and the irregular matrix case (513x8192) to evaluate the kernel's robustness and resource usage.

1. Achieved Occupancy Occupancy represents the ratio of active warps on an Streaming Multiprocessor (SM) to the maximum number of active warps supported by the hardware.

Tile Configuration	Achieved Occupancy (1024x1024)	Achieved Occupancy (513x8192)	Theoretical Occupancy
Tile 32x32	99.98%	99.99%	100%
Tile 16x16	98.64%	97.58%	100%
Tile 8x8	98.69%	97.68%	100%

2. Shared Memory Usage The Shared Memory is dynamically allocated per block based on the tile dimensions. The usage is deterministic and scales quadratically with the tile size.

Formula: $Memory_{bytes} = 2 \times TileX \times TileY \times sizeof(float)$

- **Tile 32x32:**
  - Calculation: $2 \times 32 \times 32 \times 4$ bytes.
  - **Usage: 8.19 KB / block**.
- **Tile 16x16:**
  - Calculation: $2 \times 16 \times 16 \times 4$ bytes.
  - **Usage: 2.05 KB / block**.
- **Tile 8x8:**
  - Calculation: $2 \times 8 \times 8 \times 4$ bytes.
  - **Usage: 0.51 KB / block** (512 bytes).

5)

Scaling Analysis & Performance Comparison

The bar chart compares the execution time of the CPU implementation against the GPU Naive and Tiled versions (32x32, 16x16, 8x8) across matrix sizes from $N = 256$ to $N = 8192$. A logarithmic scale is used to visualize the significant gap between CPU and GPU performance.

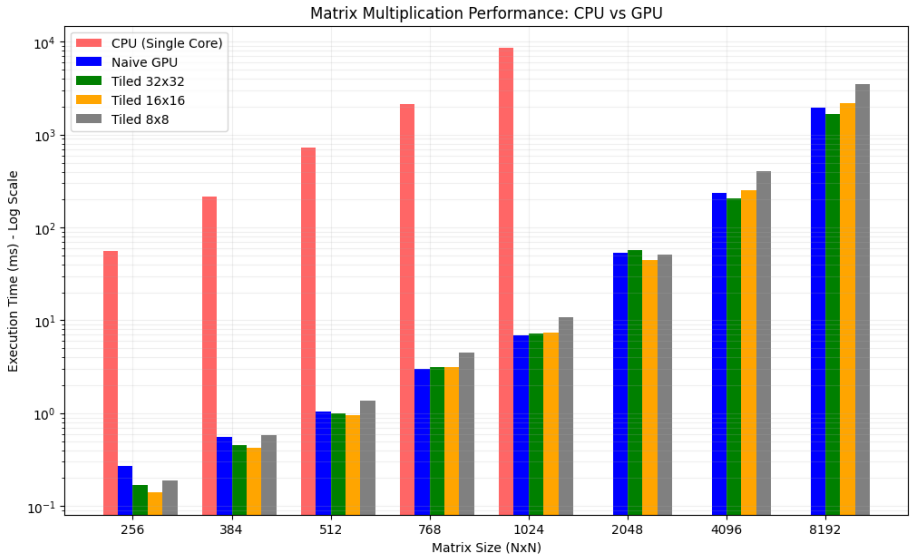

Figure 3: Q3.5: Execution Time vs. Input Matrix Sizes for Floats

1. **CPU vs. GPU:** The CPU (Red) shows an exponential rise in execution time, reaching nearly 10^4 ms at $N = 1024$. The GPU kernels are

approximately **1000x faster**, rendering the CPU impractical for larger sizes (hence omitted for $N \geq 2048$).

2. **Tile Size Impact:**
   - **Small Matrices ($N \leq 512$):** The **16x16 tile (Orange)** performs best, as it balances occupancy without the overhead of larger thread blocks on small grids.
   - **Large Matrices ($N = 8192$):** The **32x32 tile (Green)** becomes the most efficient, outperforming all others by maximizing data reuse in Shared Memory.
   - **Poor Performance of 8x8 (Gray):** The 8x8 configuration is consistently the slowest GPU kernel.
3. **Naive vs. Tiled:** The Naive implementation (Blue) remains highly competitive, often beating the 8x8 tile.