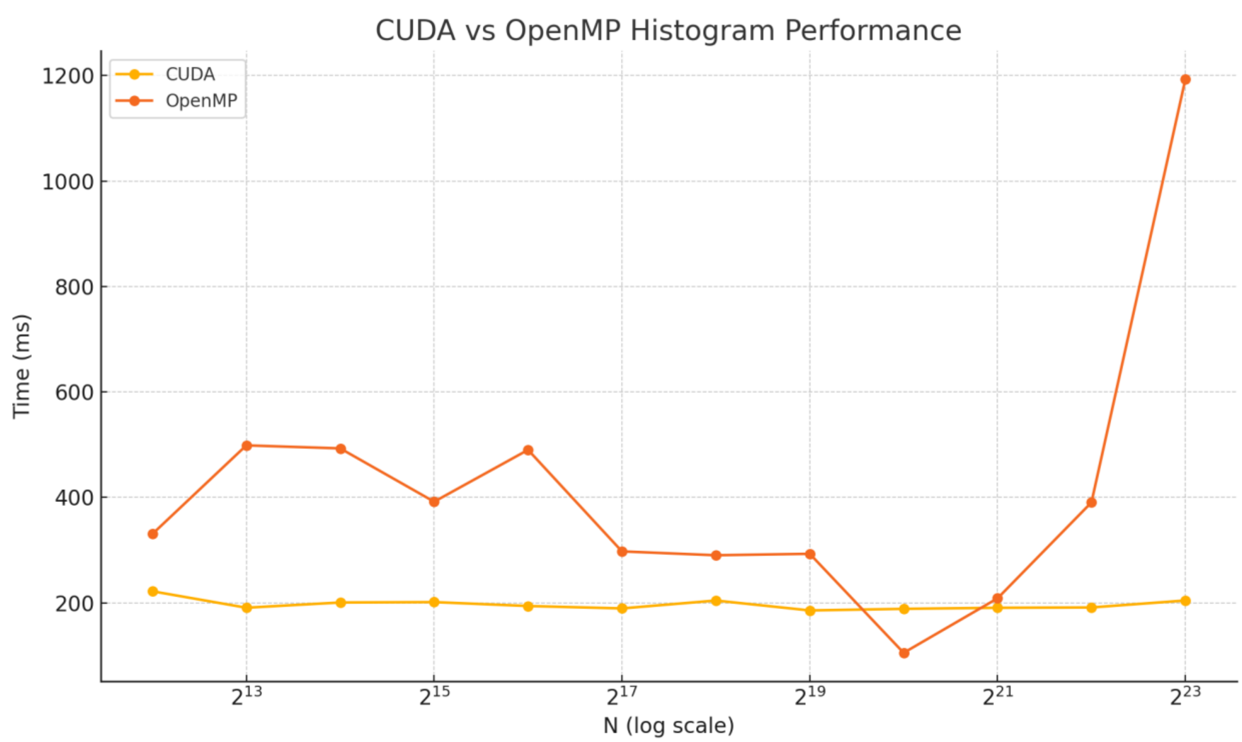
Homework 5

Question 1

Parts A and B

Results

N	CUDA_Time_ms	OMP_Time_ms	Speedup
4096	221.972823	330.690399	1.48977877
8192	190.778115	498.687634	2.61396667
16384	200.716851	492.941988	2.45590734
32768	201.461859	391.854677	1.94505639
65536	194.038183	490.167776	2.52614083
131072	189.469351	297.613384	1.57077323
262144	204.441999	290.29332	1.41992996
524288	185.649171	293.050402	1.57851716
1048576	188.658635	105.263786	0.55795901
2097152	190.656304	208.649307	1.09437403
4194304	191.291167	390.526709	2.04153028
8388608	204.515072	1193.023756	5.83342706



Discussion

In my implementation I started by directly adapting my existing binning algorithm to a CUDA kernel. I realized this would cause a lot of memory contention and many reads and writes from global memory. To avoid this, I instead created private results for each CUDA thread. The problem with this was that it would cause a lot of atomic write operations to keep the global result accurate. To address this, I implemented a version of reduction. I kept a block-based result in shared memory and updated that which limited my atomic updates to one per block rather than on a thread-by-thread basis. Another challenge was indexing the input array such that I was making contiguous memory accesses per thread which was solved by getting rid of striding.

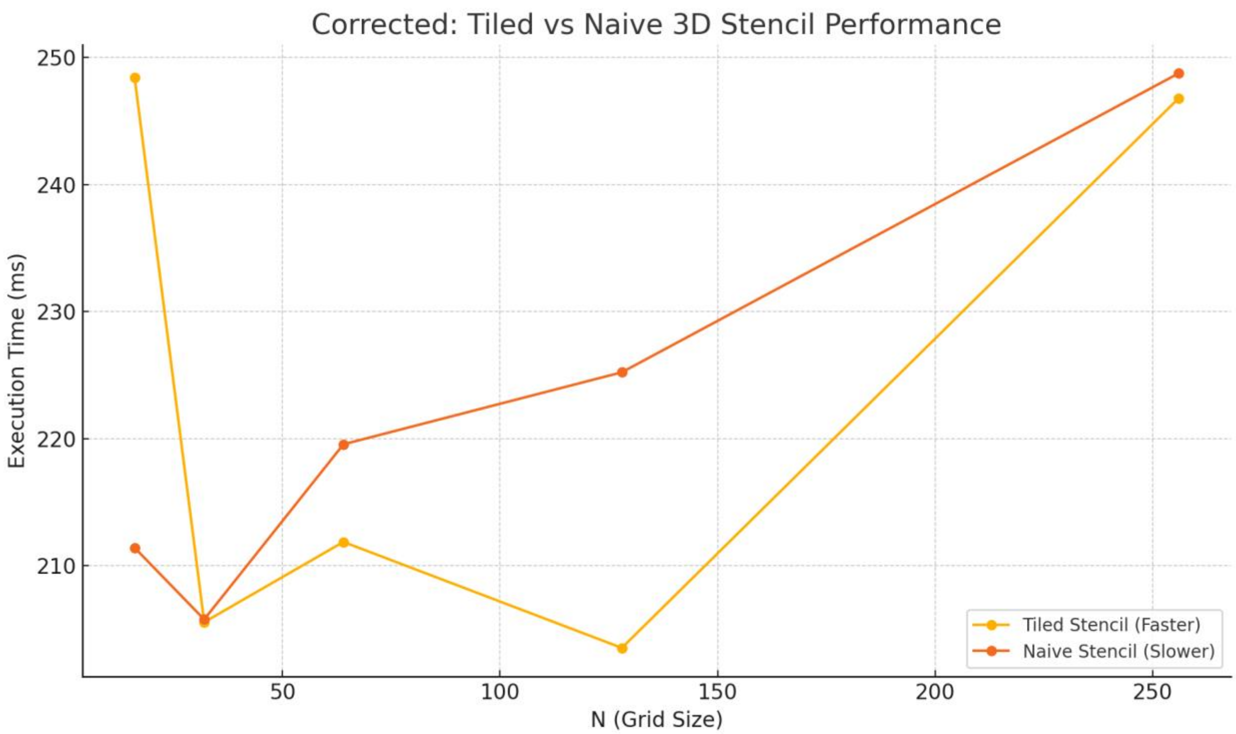
The time difference compared to OpenMP was not as large as I had expected but the reason is most likely since the memory transfer takes up a large amount of time. As the problem set started growing to 2^22 and higher the difference became very clear. I believe that this trend would continue with larger input problem sizes and then the true benefit of using the GPU would be seen.

Question 2

Part A

Results

N	Naive_Time_ms	BlockSize	Tiled_Time_ms	Speedup
16	211.422095	10	248.441509	0.85099344
32	205.777921	10	205.560193	1.00105919
64	219.56009	10	211.875853	1.03626764
128	225.229473	10	203.535652	1.10658487
256	248.763258	10	246.799443	1.00795713



Discussion

Based on the results it is clear that the tiled implementation is consistently faster, but just barely. This could be due to the overhead of creating multiple overlapping halos for each different block.

Part B

To optimize the 3D stencil kernel I used tiling in shared memory to minimize the need for multiple global memory accesses. The performance improvement was moderate since better data reuse across threads worked effectively with larger problem sizes. Multiple further optimization attempts were conducted which produced no meaningful performance improvements. I applied vectorized memory loads (such as float2 and float4) for faster neighbor access in the Z-dimension yet faced conditional complexity and alignment constraints that negated their advantages during single-pass stencil operations. Loop unrolling was considered but proved ineffective because the kernel operates without inner loops for each thread. Tests showed that implementing shared memory bank conflict avoidance and converting the 3D shared tile to a 1D buffer increased complexity without improving runtime performance. The stencil operation's simplicity and memory-bound nature make synchronization and memory indexing overheads exceed potential performance improvements. While tiling alone provided a reliable speed improvement over the basic implementation it turned out that additional tuning efforts did not produce substantial performance gains.

Question 3

One of the most crucial progressions in the Hopper H100 design is the launch of the Transformer Engine, a dedicated new feature within its 4th-generation Tensor Cores. Where Ampere's 3rd-gen Tensor Cores introduced TF32 and supported mixed precision formats like FP16 and BF16, Hopper does something even more advanced: It supports FP8 precision in two formats (E4M3 and E5M2). These are the latest from NVIDIA on the path of transforming the way people use neural networks. With the Transformer Engine, NVIDIA is now giving us a tool to enable "dynamic precision" of huge models.

1. The Engine, per layer, can switch between FP8 and FP16, offering up to 4x the speed of training large transformer-based models with the same (or better) numerical stability and accuracy.

2. This is mainly due to the compute and memory architecture of the 4th-gen Tensor Core.

Hopper significantly improves Multi-Instance GPU (MIG) capabilities, as well. While Ampere introduced MIG to allow a single GPU to be securely partitioned into up to seven independent instances, Hopper enhances this by enabling NVLink communication between MIG instances — something Ampere lacked. This makes it possible for individual MIG instances on different Hopper GPUs to share data over NVLink v4, drastically increasing bandwidth and reducing latency in multi-GPU workloads. This update makes Hopper much more suitable for multi-tenant, distributed training or inference scenarios, especially in cloud and containerized environments.

A key capability of the Tensor Memory Accelerator (TMA) is that it addresses one of the main shared-memory performance bottlenecks in GPU kernels. With Ampere, moving data between global and shared memory needed a lot of manual loads and synchronizations that added latencies.

Hopper winds up bettering this situation with hardware support in the TMA for asynchronous, parallel operations. Now, global memory and shared memory can perform async-like data transfers that are pretty much the same as what you’d do in a compute shader and that happen, get this, in parallel with computations. That obviously reduces the need to synchronize and introduces zero overhead for any ops that can run in parallel.

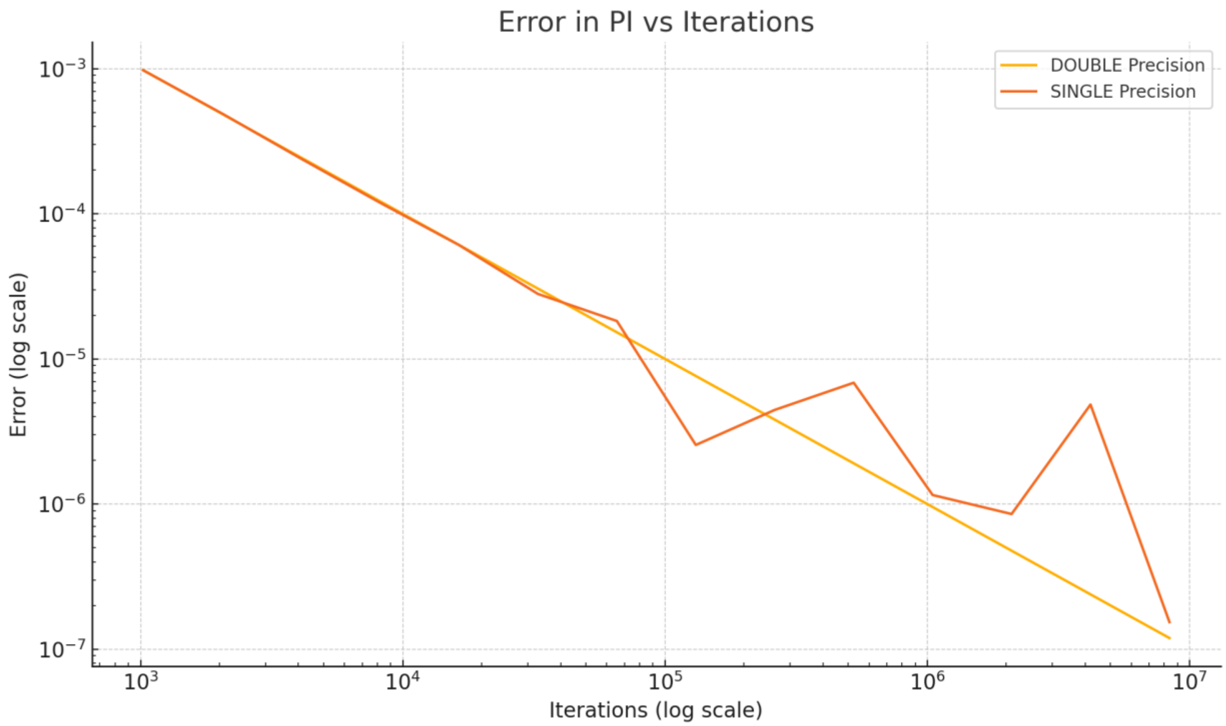
Finally, Hopper is NVIDIA's first GPU to support Confidential Computing, enabling data to remain encrypted not just at rest or in transit, but even while being processed on the GPU. It does this using a Trusted Execution Environment (TEE) and a Memory Encryption Engine (MEE) that encrypts GPU memory on the fly using hardware-managed keys. This allows for secure GPU workloads where even the host system cannot access the data, a critical feature for use cases in federated learning, privacy-sensitive inference, and industries with strict regulatory compliance like healthcare and finance. This level of end-to-end protection was not available on Ampere and represents a major leap in GPU-level security.

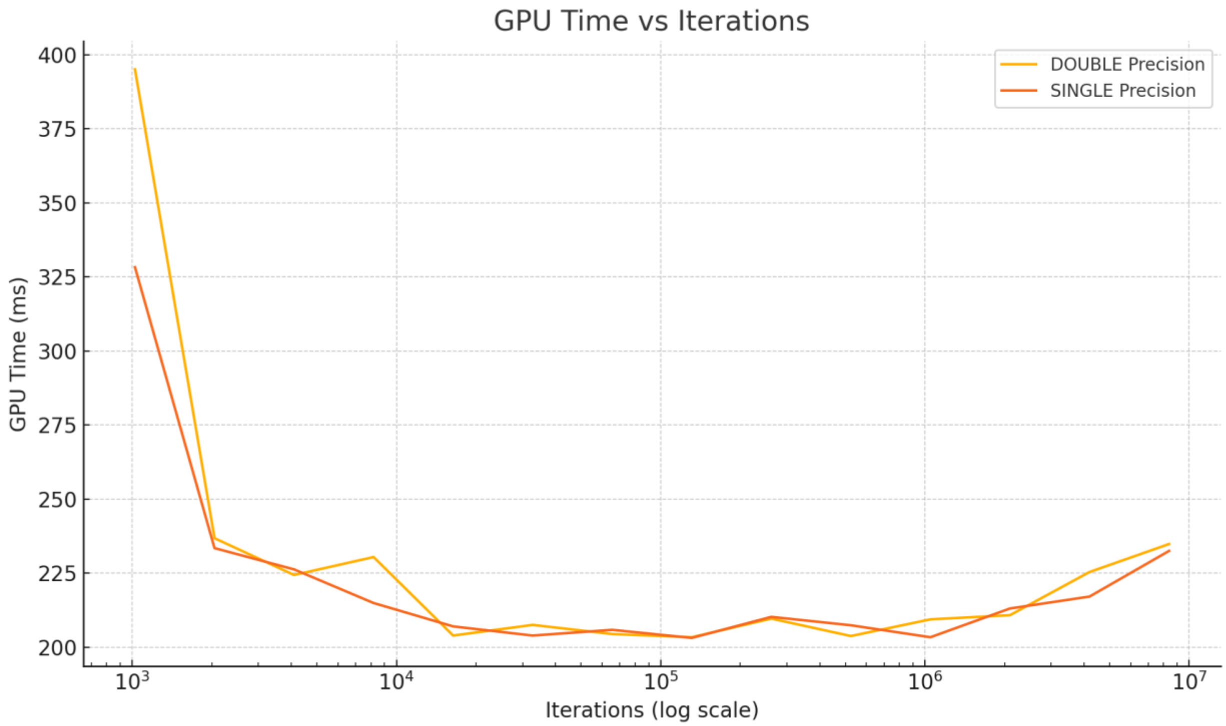
Bonus

Results

Precision	Iterations	Computed_PI	GPU_Time_ms	Error
DOUBLE	1024	3.14061609	395.0571	0.00097656
SINGLE	1024	3.1406162	328.29	0.00097645
DOUBLE	2048	3.14110437	236.8408	0.00048828
SINGLE	2048	3.1411018	233.4974	0.00049085
DOUBLE	4096	3.14134851	224.4352	0.00024414
SINGLE	4096	3.1413527	226.3231	0.00023995
DOUBLE	8192	3.14147058	230.4751	0.00012207
SINGLE	8192	3.1414733	214.972	0.00011935
DOUBLE	16384	3.14153162	203.9978	6.10E-05
SINGLE	16384	3.1415317	207.0785	6.10E-05
DOUBLE	32768	3.14156214	207.5877	3.05E-05
SINGLE	32768	3.1415646	203.9873	2.81E-05
DOUBLE	65536	3.14157739	204.4996	1.53E-05
SINGLE	65536	3.1415744	205.9238	1.83E-05
DOUBLE	131072	3.14158502	203.4277	7.63E-06
SINGLE	131072	3.1415901	203.1903	2.55E-06
DOUBLE	262144	3.14158884	209.709	3.81E-06
SINGLE	262144	3.1415882	210.3001	4.45E-06
DOUBLE	524288	3.14159075	203.8297	1.91E-06
SINGLE	524288	3.1415858	207.461	6.85E-06
DOUBLE	1048576	3.1415917	209.4604	9.54E-07
SINGLE	1048576	3.1415915	203.4228	1.15E-06
DOUBLE	2097152	3.14159218	210.8672	4.77E-07

SINGLE	2097152	3.1415918	213.1269	8.54E-07
DOUBLE	4194304	3.14159242	225.431	2.38E-07
SINGLE	4194304	3.1415975	217.1469	4.85E-06
DOUBLE	8388608	3.14159253	234.8723	1.19E-07
SINGLE	8388608	3.1415925	232.5535	1.54E-07





Discussion

Based on the results its clear that the precision makes a huge difference in convergence as the iteration count increases. This can be due to incorrect estimation of extremely small values that the double precision can handle but the single precision cannot.