DD2360 Homework 4 — Group 2 Report (Q2 + Bonus)

Submission Requirements

ZIP File Structure

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

```
assignment_4/
├── q2/
|   ├── hw4-heat-template.cu
|   ├── Makefile
|   ├── run_experiments.sh
|   ├── plot_results.py
|   ├── README.md
|   └── results/
|       ├── flops_analysis.csv
|       ├── flops_vs_dimX.png
|       ├── error_vs_nsteps.csv
|       ├── error_vs_nsteps.png
|       ├── prefetch_comparison.csv
|       └── prefetch_comparison.png
├── bonus/
|   ├── vecMult.cu
|   ├── Makefile
|   ├── run_experiments.sh
|   ├── plot_results.py
|   ├── README.md
|   └── results/
|       ├── exp1_1024x2048.txt
|       ├── performance_comparison.csv
|       ├── performance_comparison.png
|       └── wmma_accuracy.png
└── DD2360HT25_HW4_Group2.md
```

Group Member Contribution

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

Question 1 –1D Convolution

Questions to answer in the report

1. Name 3 applications domain where convolution are used?

   Image & video processing: blurring, sharpening, edge detection, denoising, feature extraction.

Deep learning (CNNs): convolution layers for feature extraction in image classification, object detection, segmentation, etc.

Signal processing : FIR filtering, echo cancellation, matched filtering, and modeling LTI system responses.

2. Explain in your own words how you design the tiled kernel and how you choose the tile size.

   (1) How to design the tiled kernel:

   The tiled kernel was designed by letting each thread block compute a consecutive "tile" of output elements. Before computing, the block cooperatively loads the needed input segment into **shared memory**, including a small **halo** on both sides (because the mask radius is 2). After a `__syncthreads()`, each thread computes its output using the cached shared-memory values and writes the result to global memory. This reduces redundant global memory reads because neighboring threads reuse the same input data from shared memory.

   (2) How to choose the tile size:

   First, ensure the tile size is a multiple of 32 (the warp size) so execution is efficient and no warps are wasted. I avoid choosing tiles that are too small (e.g., 32/64) because they generate many blocks, increasing scheduling/launch overhead, and they may not provide enough parallelism to hide memory latency. I also avoid tiles that are too large (e.g., 1024) because a single block can consume more resources and reduce occupancy; moreover, the tiled version requires `__syncthreads()`, and larger blocks tend to incur higher synchronization wait time. Finally, I decide based on measurements: I benchmark common sizes such as 128, 256, and 512 using CUDA event timing and pick the one with the lowest average runtime (in my tests, 128–256 usually performs best).

3. How many global memory reads and writes are performed in the basic implementation and the tiled implementation, respectively?

   Taking tilesize 256 as an example:

   Basic implementation

   For the whole output array (assuming $N \geq 5$):

   - **Global reads from input N:**

     $$\text{Reads from } N = 5N - 6$$

     (the four boundary outputs read fewer than 5 elements; total reduction is 6)

   - **Global reads from mask M:**

     $$\text{Reads from } M = 5N - 6$$

   - **Global writes to output P:**

     $$\text{Writes to } P = N$$

   Tiled implementation (tileSize = 256)

   Global reads from input N

   Each block loads:

   - **256 center elements**, plus
   - **2 halo elements on the left and 2 on the right** ($total 2R = 4$)

   Across the whole array:

- Every valid input element is loaded as a center exactly once **N reads**
- Halo elements are redundantly loaded at block boundaries

$$2R(B-1) = 4(B-1) reads$$

$$\text{Reads from } N = N + 4(B-1)$$

Global reads from mask M

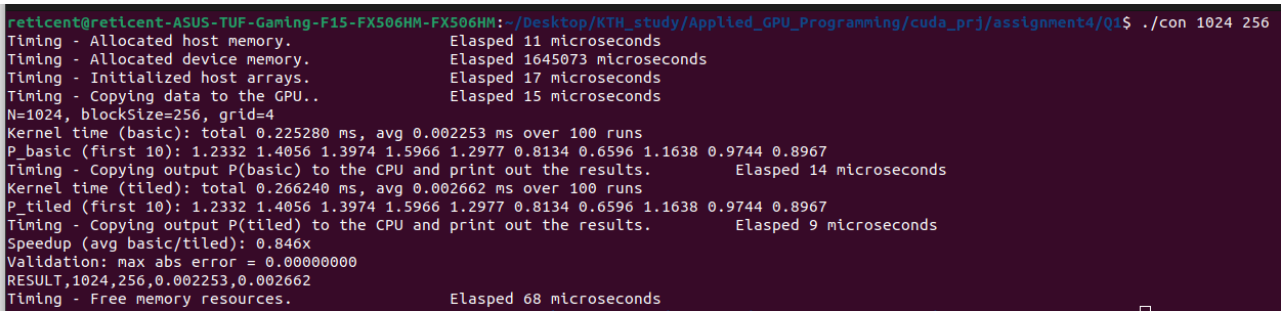
Each output still uses 5 mask values:

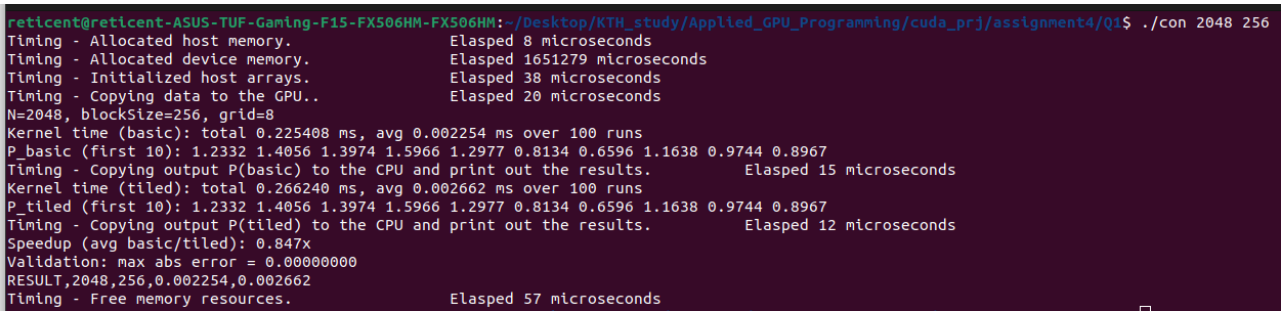
$$\text{Reads from } M = 5N$$

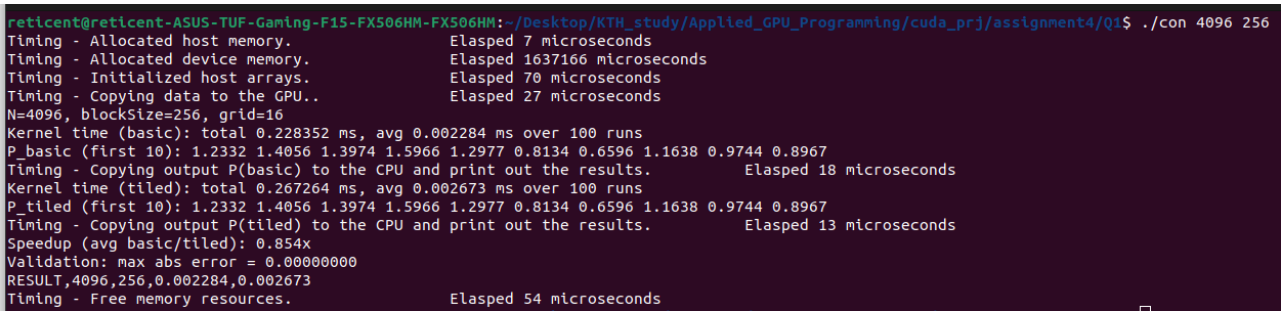
Global writes to output P

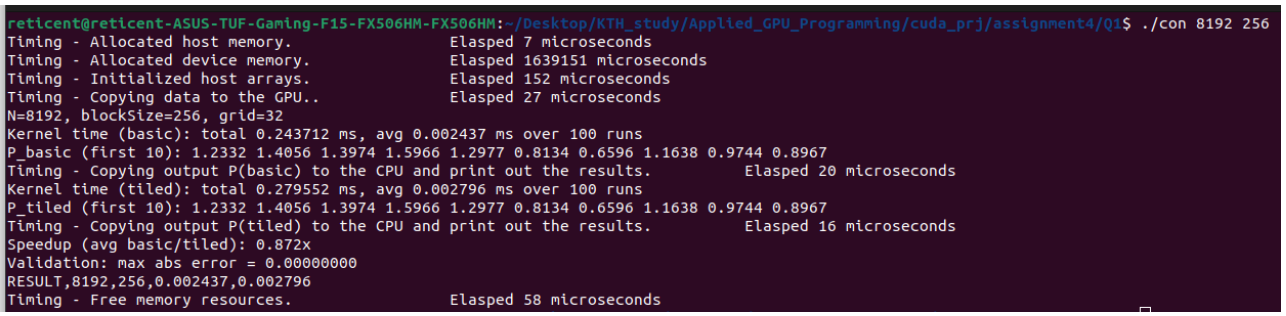
$$\text{Writes to } P = N$$

4. Run with your program with N of 1024, 2048, 4096, 8192, 16384. Include the screenshot of your output.





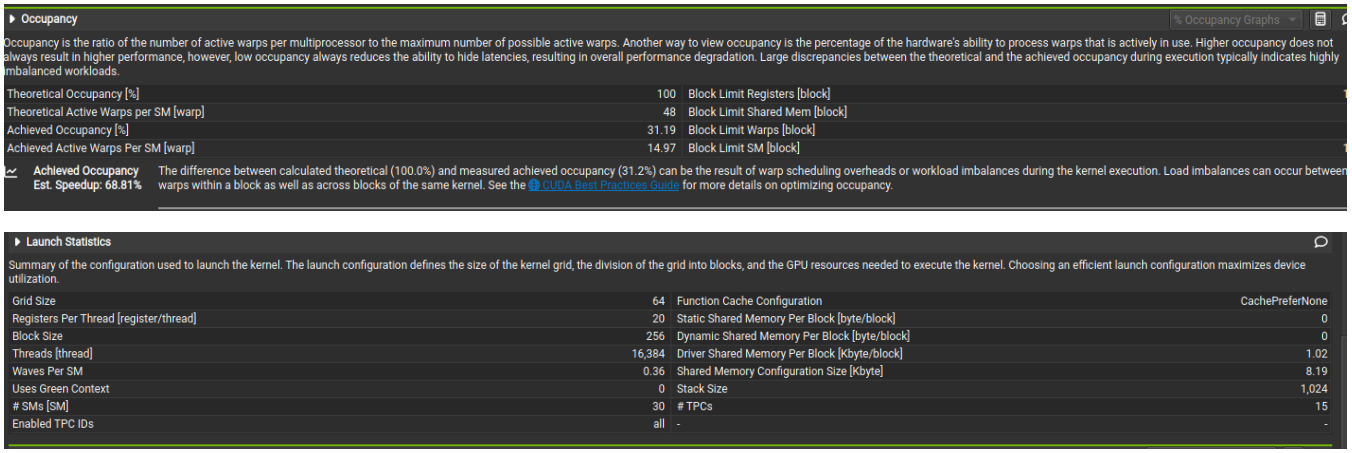




```
reticent@reticent-ASUS-TUF-Gaming-F15-FX506HM-FX506HM:~/Desktop/KTH_study/Applied_GPU_Programming/cuda_prj/assignment4/Q1$ ./con 16384 256
Timing - Allocated host memory.                Elasped 11 microseconds
Timing - Allocated device memory.              Elasped 198190 microseconds
Timing - Initialized host arrays.              Elasped 285 microseconds
Timing - Copying data to the GPU..             Elasped 34 microseconds
N=16384, blockSize=256, grid=64
Kernel time (basic): total 0.226208 ms, avg 0.002262 ms over 100 runs
P_basic (first 10): 1.2332 1.4056 1.3974 1.5966 1.2977 0.8134 0.6596 1.1638 0.9744 0.8967
Timing - Copying output P(basic) to the CPU and print out the results.         Elasped 28 microseconds
Kernel time (tiled): total 0.266240 ms, avg 0.002662 ms over 100 runs
P_tiled (first 10): 1.2332 1.4056 1.3974 1.5966 1.2977 0.8134 0.6596 1.1638 0.9744 0.8967
Timing - Copying output P(tiled) to the CPU and print out the results.         Elasped 27 microseconds
Speedup (avg basic/tiled): 0.850x
Validation: max abs error = 0.00000000
RESULT,16384,256,0.002262,0.002662
Timing - Free memory resources.                Elasped 55 microseconds
```

5. Profile your program using the largest N. Report Achieved Occupancy and Shared Memory usage. Analyze your results.

For the basic kernel:



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]	1
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]	
Achieved Occupancy [%]	31.19	Block Limit Warps [block]	
Achieved Active Warps Per SM [warp]	14.97	Block Limit SM [block]	

Achieved Occupancy
Est. Speedup: 68.81%

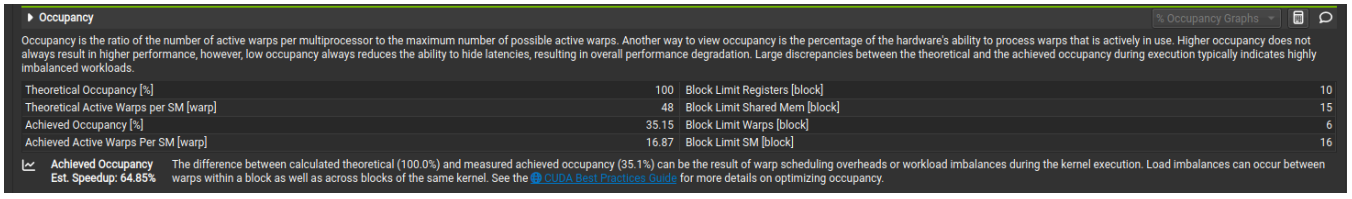
The difference between calculated theoretical (100.0%) and measured achieved occupancy (31.2%) can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel. See the CUDA Best Practices Guide for more details on optimizing occupancy.

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	64	Function Cache Configuration	CachePreferNone
Registers Per Thread [register/thread]	20	Static Shared Memory Per Block [byte/block]	0
Block Size	256	Dynamic Shared Memory Per Block [byte/block]	0
Threads [thread]	16,384	Driver Shared Memory Per Block [Kbyte/block]	1.02
Waves Per SM	0.36	Shared Memory Configuration Size [Kbyte]	8.19
Uses Green Context	0	Stack Size	1,024
# SMs [SM]	30	# TPCs	15
Enabled TPC IDs	all	-	-

For the tiled kernel:



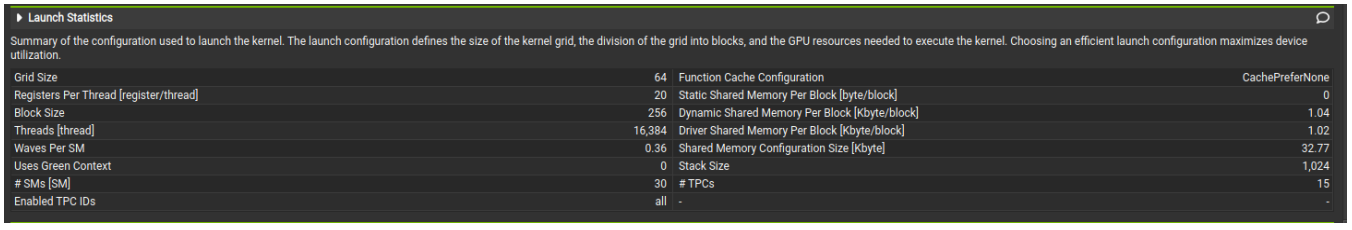
Occupancy

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]	10
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]	15
Achieved Occupancy [%]	35.15	Block Limit Warps [block]	6
Achieved Active Warps Per SM [warp]	16.87	Block Limit SM [block]	16

Achieved Occupancy
Est. Speedup: 64.85%

The difference between calculated theoretical (100.0%) and measured achieved occupancy (35.1%) 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.



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	64	Function Cache Configuration	CachePreferNone
Registers Per Thread [register/thread]	20	Static Shared Memory Per Block [byte/block]	0
Block Size	256	Dynamic Shared Memory Per Block [Kbyte/block]	1.04
Threads [thread]	16,384	Driver Shared Memory Per Block [Kbyte/block]	1.02
Waves Per SM	0.36	Shared Memory Configuration Size [Kbyte]	32.77
Uses Green Context	0	Stack Size	1,024
# SMs [SM]	30	# TPCs	15
Enabled TPC IDs	all	-	-

For N=16384 and blockSize=256, the basic kernel achieves 31.19% achieved occupancy (14.97 active warps/SM), while the tiled kernel achieves 35.15% achieved occupancy (16.87 active warps/SM). The grid contains only 64 blocks on a GPU with 30 SMs, so the workload is undersubscribed: on average each SM receives about ~2 blocks, i.e., ~16 warps, which is far below the theoretical maximum of 48 warps/SM. Therefore, the achieved occupancy is primarily limited by the small grid size rather than by per-block resource limits.

In terms of shared memory, the basic kernel uses 0 B shared memory per block, while the tiled kernel uses 1.04 KB dynamic shared memory per block (≈1040 B), matching (blockSize + 2R)·sizeof(float) = (256+4)·4. This shared memory footprint is small and does not significantly restrict occupancy; the tiled kernel's performance is instead influenced by shared-memory staging overhead and synchronization.

Question 2 – NVIDIA libraries and Unified Memory

This exercise solves the 1D heat equation with an explicit finite-difference iteration. Each iteration performs a sparse matrix-vector multiplication (SpMV) and a BLAS AXPY update:

$$\text{tmp} \leftarrow A \cdot \text{temp}, \qquad \text{temp} \leftarrow \alpha \cdot \text{tmp} + \text{temp}$$

We used:

- **Unified Memory (`cudaMallocManaged`)** for the CSR sparse matrix and vectors.
- **cuSPARSE** generic API `cusparseSpMV()` for SpMV.
- **cuBLAS** `cublasDaxpy()` and `cublasDnrm2()` for the vector update and norms.
- **UM Prefetch** (`cudaMemPrefetchAsync`) to reduce page migration overhead (optional `--no-prefetch` to disable).

Implementation highlights

- **Unified memory allocation**: `temp`, `tmp`, CSR arrays `A`, `ARowPtr`, `AColIndx`.
- **Prefetching**:
  - Prefetch to CPU before host-side initialization (matrix + boundary conditions).
  - Prefetch to GPU before the iteration loop.
- **Library setup/teardown**:
  - Create/destroy **cuBLAS** handle, set pointer mode to host.
  - Create/destroy **cuSPARSE** handle + CSR and dense-vector descriptors.
  - Query `cusparseSpMV_bufferSize()` and allocate temporary buffer.
- **Relative error**:
  - Compute the "exact" steady-state solution on GPU as a linear ramp between boundary temperatures.
  - Compute relative error as (|\text{exact}-\text{temp}|*2 / |\text{temp}*|2) using cuBLAS.

Q2.1 — FLOPS estimation for SpMV, and results vs dimX

For CSR SpMV, each non-zero contributes roughly **2 FLOPs** (one multiply + one add). For a tridiagonal matrix of size `dimX`, the non-zeros are:

$$n_{nz} = 3 \cdot \text{dimX} - 6$$

Over `iterations` iterations, total FLOPs for SpMV is approximately:

$$\text{FLOPs}_{total} \approx 2 \cdot n_{nz} \cdot \text{iterations}$$

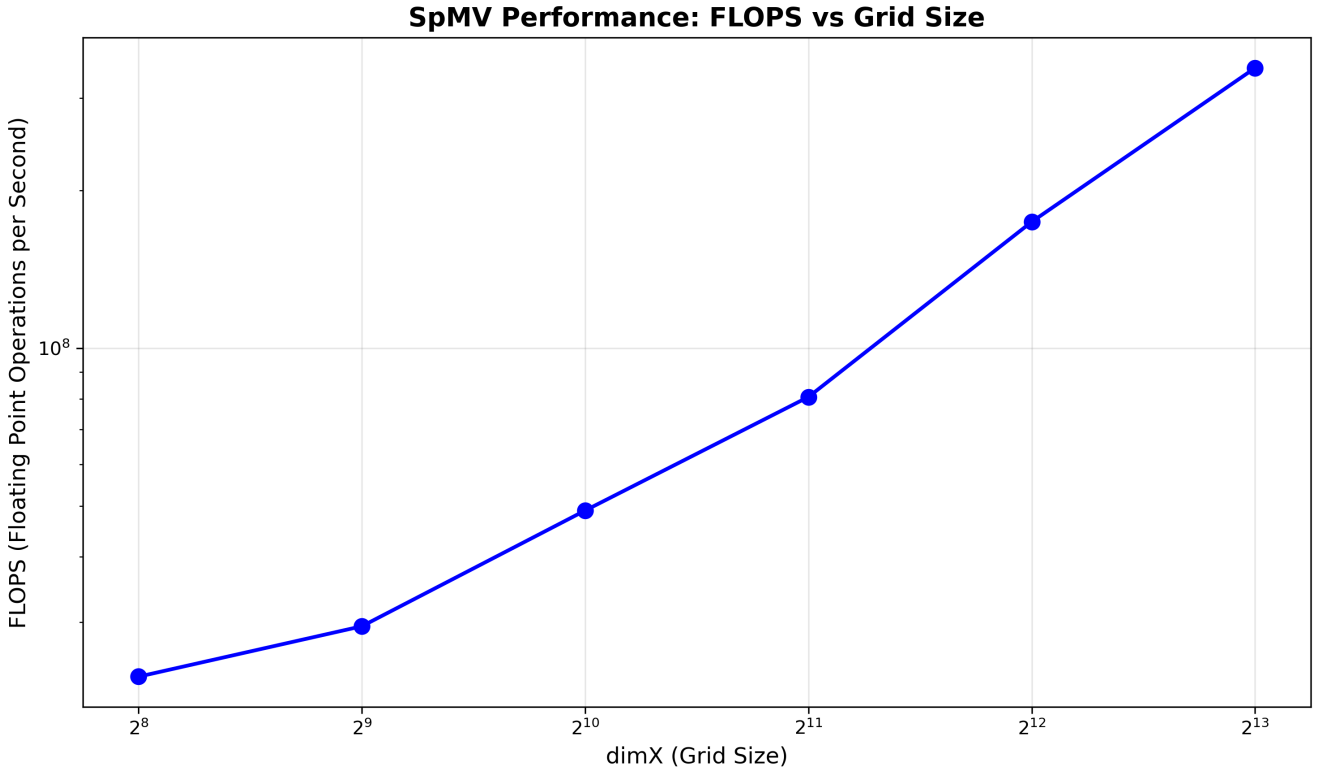
Given total measured iteration time `time_us` (microseconds), the achieved FLOPS is:

$$\text{FLOPS} \approx \frac{\text{FLOPs}_{total}}{\text{time\_us} \times 10^{-6}}$$

Measured results (from `q2/results/flops_analysis.csv`):

dimX	nnz (3*dimX-6)	iteration time (us)	iterations	FLOPS
256	762	64,549	1000	2.361e7
512	1,530	103,852	1000	2.947e7
1024	3,066	125,140	1000	4.900e7
2048	6,138	152,080	1000	8.072e7
4096	12,282	140,965	1000	1.743e8
8192	24,570	143,447	1000	3.426e8

Plot:



Observations

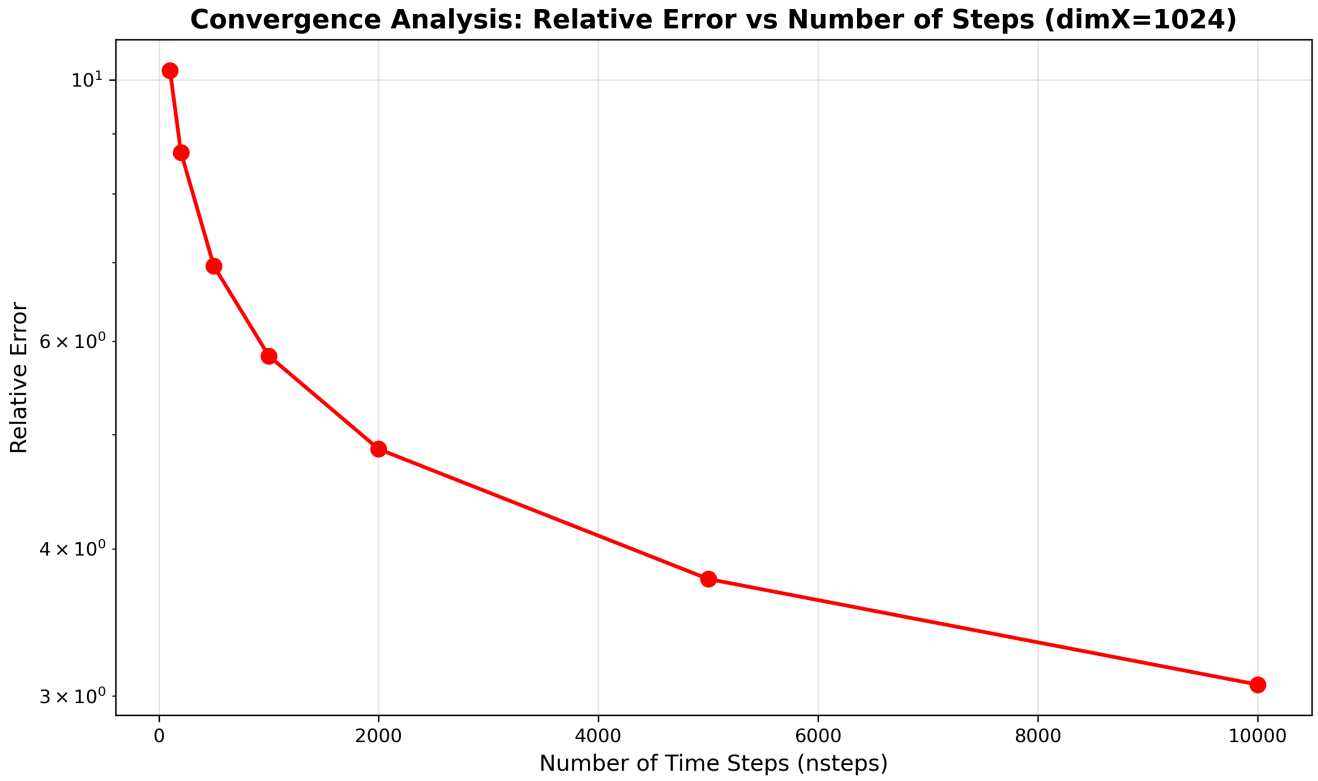
- FLOPS increases with `dimX` and reaches $\sim 3.4 \times 10^8$ FLOPS at `dimX=8192`.
- For small `dimX`, the kernel is dominated by launch/overhead and insufficient parallelism; for larger `dimX`, GPU utilization improves and the throughput increases.

Q2.2 — Relative error vs nsteps (dimX=1024)

We fixed `dimX=1024` and varied `nsteps` from 100 to 10000 (see `q2/results/error_vs_nsteps.csv`). The measured relative error decreases monotonically:

nsteps	relative error
100	10.185756
200	8.676701
500	6.947132
1000	5.829536
2000	4.859128
5000	3.770096
10000	3.067374

Plot:



Observations

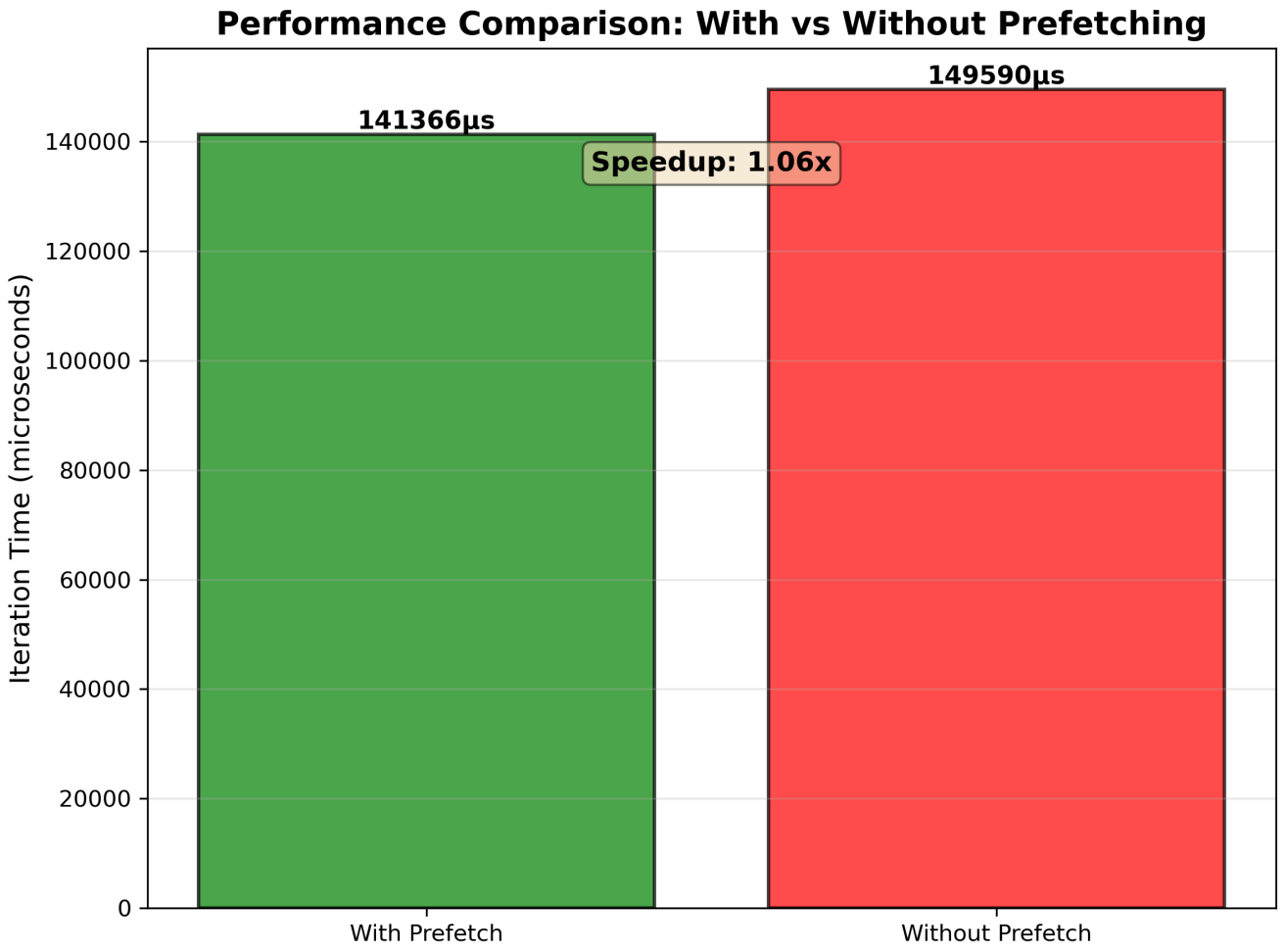
- The error decreases as `nsteps` increases, meaning the explicit iteration moves the solution closer to the steady-state profile.
- The decrease becomes slower at larger `nsteps`, which is consistent with convergence becoming progressively harder as we approach steady state.

Q2.3 — Unified Memory prefetching vs no-prefetch

We compared total iteration time for `dimX=1024, nsteps=1000` with and without UM prefetching (flag `--no-prefetch` disables it):

Setting	iteration time (us)
With Prefetch	141,366
Without Prefetch	149,590

Plot:



Performance impact

- Prefetch improves runtime by about **1.06× (~6%)** in our experiment.
- Explanation: with prefetching, pages are migrated proactively before the iteration loop, reducing on-demand page faults/migrations during `cusparseSpMV()` and cuBLAS calls.

Bonus – Tensor Core (WMMA)

We extended the previous GEMM assignment with a **WMMA** kernel `wmma_gemm()` to use **Tensor Cores**:

- **Load** matrix tiles into WMMA fragments

- **MMA** via `wmma::mma_sync()`
- **Store** accumulated result back to memory

Bonus.1 — A(1024×2048) × B(2048×1024): fragments in use and total fragments

From our program output (`bonus/results/exp1_1024x2048.txt`):

- **Fragment dimensions**: **16×16×16** (WMMA standard tile)
- Output matrix (C) is **1024×1024**, so the number of 16×16 output tiles is:

$$\left(\frac{1024}{16}\right)\left(\frac{1024}{16}\right) = 64 \times 64 = 4096$$

- Per output tile, we use 4 fragment objects (A, B, accumulator, C), so total fragment objects used:

$$4096 \times 4 = 16384$$

  which matches the printed **"Total fragments used: 16384"**.

Bonus.2 — Profiling Tensor Core utilization (method)

We prepared the profiling workflow in `bonus/README.md` (Nsight Compute):

```
ncu --set full ./vecMult 1024 2048 2048 1024
```

Relevant metrics to report typically include:

- **Tensor activity / tensor pipe utilization** (e.g., Tensor Active / HMMA instruction activity)
- **Half precision functional unit utilization** (e.g., `half_precision_fu_utilization`)
- **Throughput as % of peak** (e.g., `sm__throughput.avg.pct_of_peak_sustained_elapsed`)

To study thread-block configuration effects, the WMMA launch shape `(blockDim.x=32, blockDim.y=#warpsPerBlock)` can be varied (e.g., 1/2/4/8 warps per block) and re-profiled.

Windows notes (command line)

On Windows, use **Nsight Compute CLI** (`ncu`). `nvprof` is deprecated and generally not supported on newer drivers/GPUs.

Different threads-per-block: in our implementation,

- `--wmma-warps 1` → 32 threads/block
- `--wmma-warps 2` → 64 threads/block
- `--wmma-warps 4` → 128 threads/block
- `--wmma-warps 8` → 256 threads/block

Run the same `ncu` command with `--wmma-warps {1,2,4,8}` and compare.

Observation template (fill with your NCU numbers)

warpsPerBlock	threadsPerBlock	Tensor Active (pct of peak)	HMMA inst executed (sum)
1	32	2.04%	4,194,304
2	64	3.80%	4,194,304
4	128	6.24%	4,194,304
8	256	7.36%	4,194,304

In general, when the problem size is large enough (e.g., 8192×8192), Tensor Core utilization tends to stay high across reasonable block configurations; changes are typically driven by occupancy and memory behavior rather than whether Tensor Cores are used at all.

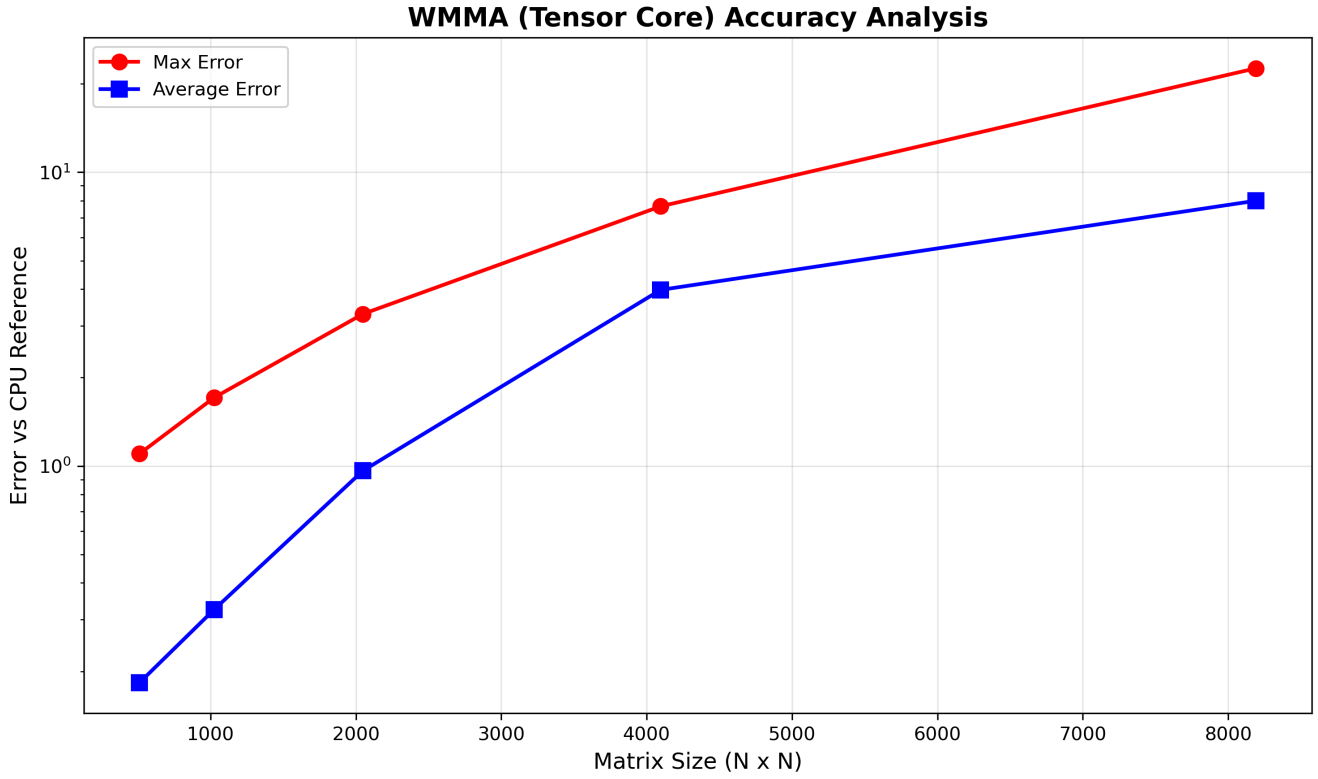
Answer (observation): Yes—changing threads per block changes the observed Tensor Core utilization. In our 2048×2048 experiment, the executed HMMA instruction count stays constant (4,194,304), confirming Tensor Cores are used in all cases, but the tensor pipe active metric increases with more warps per block: **2.04%** → **3.80%** → **6.24%** → **7.36%** for **32/64/128/256 threads per block**. This indicates improved warp-level parallelism/occupancy and better ability to keep the tensor pipeline busy as the block contains more warps, with diminishing returns by 8 warps per block.

Bonus.3 — Accuracy loss (WMMA vs CPU reference)

Because WMMA uses **FP16 inputs** (with FP32 accumulation), it introduces quantization/rounding error compared to the CPU reference (DataType).

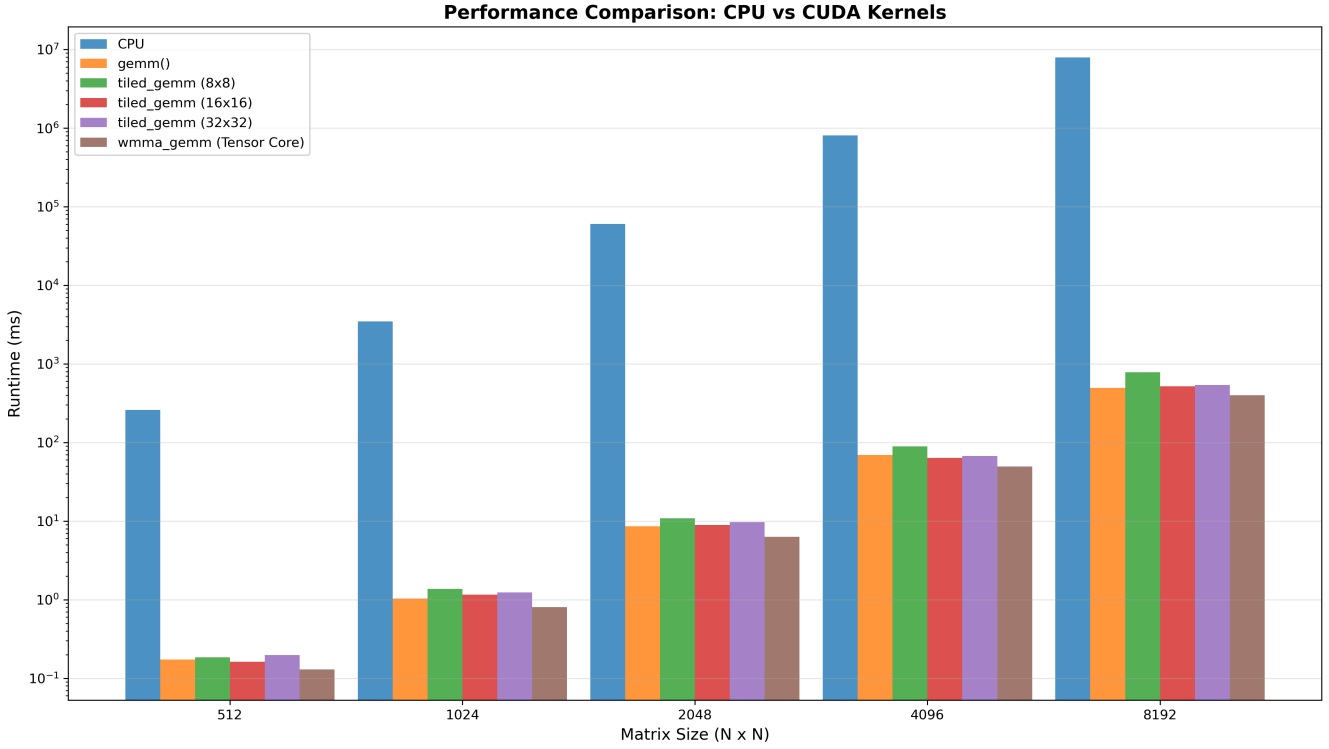
- For the non-square case (1024×2048 × 2048×1024), our output reports:
  - **max error**: 3.097656
  - **avg error**: 9.831694×10^{-1}

For square matrices, the WMMA errors increase with size (from `bonus/results/performance_comparison.csv`), and the plot is:



Bonus.4 — Runtime comparison: CPU vs gemm() vs wmma (5 sizes)

We benchmarked 5 square sizes (512/1024/2048/4096/8192) and plotted a log-scale bar chart:



Key observations

- GPU kernels (`gemm`, tiled, `wmma`) are **orders of magnitude faster** than CPU for large sizes.
- `wmma_gemm()` is consistently the fastest GPU implementation in our measurements (e.g., at 8192: **397 ms** WMMA vs **497 ms** naive `gemm`).

- The speedup of WMMA over the naive CUDA kernel is typically around **~1.2×–1.35×** across the tested sizes, while trading off numerical accuracy due to FP16 inputs.