

Assignment 3 Submission Structure

Submission Requirements

PDF File

- **Name:** DD2360HT25_HW3_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]/  
│   └── README.md  
├── Q[2]/  
│   ├── vecAdd_streams.cu  
│   ├── batch_test_N.sh  
│   ├── batch_test_Sseg.sh  
│   ├── plot_vs_N.py  
│   ├── plot_vs_Sseg.py  
│   ├── Makefile  
│   └── README.md
```

Group Member Contribution

- Jinye Gong: Q[1], Q[2]

Q1 – Thread Scheduling and Execution Efficiency

Picture width: $X = n$ (columns)

Picture height: $Y = m$ (rows)

Block size: $\text{blockDim} = (64, 16) \rightarrow \text{threadsPerBlock} = 64 * 16 = 1024$

Warp size: 32 threads $\rightarrow \text{warpsPerBlock} = 1024 / 32 = 32$

Warp numbering inside a block uses a linear thread id

$$\text{linear}_{\text{tid}} = \text{threadIdx.y} \cdot 64 + \text{threadIdx.x}$$

and each consecutive group of 32 threads forms one warp.

Because 64 is a multiple of 32, **no warp ever spans two different threadIdx.y values** (no warp crosses a row inside a block).

Divergence happens if, inside the same warp, some threads satisfy $(\text{Row} < m \ \&\& \ \text{Col} < n)$ and others do not.

- **1. Assume X=800 and Y=600. Assume that we decided to use 2D 64X16 thread blocks. How many warps will be generated during the execution of the kernel? How many warps will have control divergence? Please explain your answers.**

Grid dimensions:

$$\text{gridDim.x} = \left\lceil \frac{800}{64} \right\rceil = 13, \quad \text{gridDim.y} = \left\lceil \frac{600}{16} \right\rceil = 38.$$

Number of blocks:

$$\text{blocks} = 13 \times 38 = 494.$$

Total warps:

$$\text{total warps} = 494 \times 32 = 15,808.$$

X direction (columns):

12 full blocks cover $12 \times 64 = 768$ columns.

The 13-th (rightmost) block covers columns 768–831, but the picture only goes up to column 799.

In that block, `Col = 768 + threadIdx.x`, and `Col < 800` \Leftrightarrow `threadIdx.x < 32`.

So for each row in the rightmost block:

`threadIdx.x = 0..31` \rightarrow all threads **inside** the picture.

`threadIdx.x = 32..63` \rightarrow all threads **outside** the picture.

Warp layout per row:

Warp A: `threadIdx.x = 0..31` \rightarrow all `if` condition true.

Warp B: `threadIdx.x = 32..63` \rightarrow all `if` condition false.

No warp mixes true and false \rightarrow **no divergence from the right edge.**

Y direction (rows):

37 full blocks cover $37 \times 16 = 592$ rows.

The 38-th (bottom) block covers rows 592–607, but the picture only goes up to row 599.

In that block, `Row = 592 + threadIdx.y`, and `Row < 600` \Leftrightarrow `threadIdx.y < 8`.

So:

`threadIdx.y = 0..7` \rightarrow rows inside the picture.

`threadIdx.y = 8..15` \rightarrow rows outside.

Because each warp stays within a single `threadIdx.y` (a single row in the block), every warp is either all in range or all out of range \rightarrow **no divergence from the bottom edge.**

Answer for 1

Total warps: **15,808**

Warps with control divergence: **0**

- Now assume $X=600$ and $Y=800$ instead, how many warps will have control divergence? Please explain your answers.

Grid dimensions:

$$\text{gridDim.x} = \left\lceil \frac{600}{64} \right\rceil = 10, \quad \text{gridDim.y} = \left\lceil \frac{800}{16} \right\rceil = 50.$$

Number of blocks:

$$\text{blocks} = 10 \times 50 = 500.$$

Total warps:

$$\text{total warps} = 500 \times 32 = 16,000.$$

Y direction:

$$\text{threads} = 50 \times 16 = 800.$$

So Y is an exact multiple of the block height. There is no partial block at the bottom; all rows are fully inside the picture → **no divergence from the bottom.**

X direction (rightmost column of blocks, `blockIdx.x = 9`):

The first 9 blocks cover $9 \times 64 = 576$ columns.

The 10-th block covers columns 576–639, but the picture only goes up to column 599.

For `blockIdx.x = 9`, `Col = 576 + threadIdx.x`, and `Col < 600` ⇔ `threadIdx.x < 24`.

So in each row of the rightmost block:

`threadIdx.x = 0..23` → in range (if = true)

`threadIdx.x = 24..63` → out of range (if = false)

Warp layout:

Warp A: `threadIdx.x = 0..31` → 24 threads in range, 8 out of range → **divergent warp**

Warp B: `threadIdx.x = 32..63` → all out of range → not divergent

Each such block has 16 rows → **16 divergent warps per rightmost block.**

There are 50 blocks in that rightmost column (`gridDim.y = 50`):

$$\text{divergent warps} = 50 \times 16 = 800.$$

Answer for 2

Warps with control divergence: **800**

(Only the warps with `blockIdx.x = 9` and `threadIdx.x = 0..31` diverge; all others are either fully active or fully inactive.)

- **Now assume X=600 and Y=899, how many warps will have control divergence? Please explain your answers.**

Grid dimensions:

$$\text{gridDim.x} = \left\lceil \frac{600}{64} \right\rceil = 10, \quad \text{gridDim.y} = \left\lceil \frac{899}{16} \right\rceil = 57.$$

Number of blocks:

$$\text{blocks} = 10 \times 57 = 570.$$

Total warps:

$$\text{total warps} = 570 \times 32 = 18,240.$$

Now both directions have partial blocks. We split the analysis into three block regions.

```
**Right edge, not bottom row (`blockIdx.x = 9`, `blockIdx.y = 0..55`)**
```

Warp A (0..31):

`x = 0..23` → in-range → `true`

`x = 24..31` → out-of-range → `false`

→ Warp A has both true and false → **divergent**.

Warp B (32..63):

All `x >= 24` → all out-of-range → all `false`

→ Warp B is **not divergent**.

So in this block:

Each row has 1 divergent warp (warp A)

There are 16 rows

Each right-edge block (not at bottom) has 16 divergent warps.

Bottom row but not rightmost column (`blockIdx.x = 0..7`, `blockIdx.y = 54`)

`x = 0..2` → in-range → `true`

`x = 2..15` → out-of-range → `false`

Bottom row but not rightmost column has 0 divergent warp.

Bottom-right corner block(`blockIdx.x = 9`, `blockIdx.y = 56`)

Warp A (0..31):

`x = 0..23` → in-range → `true`

`x = 24..31` → out-of-range → `false`

→ Warp A has both true and false → **divergent**.

Warp B (32..63):

All `x >= 24` → all out-of-range → all `false`

→ Warp B is **not divergent**.

Bottom-right corner block has 3 divergent warps.

Answer for 3

From right-edge (non-bottom) blocks: `56 blocks × 16 divergent warps/block =`

`896`

From bottom-right corner block: `3` divergent warps

Total:

$$\text{divergent warps} = 896 + 3 = 899$$

Q2 – CUDA Streams

- **Compared to the non-streamed vector addition, what performance gain do you get? Present in a plot (you may include comparison at different vector length)**

We compare two GPU implementations of `vecAdd`:

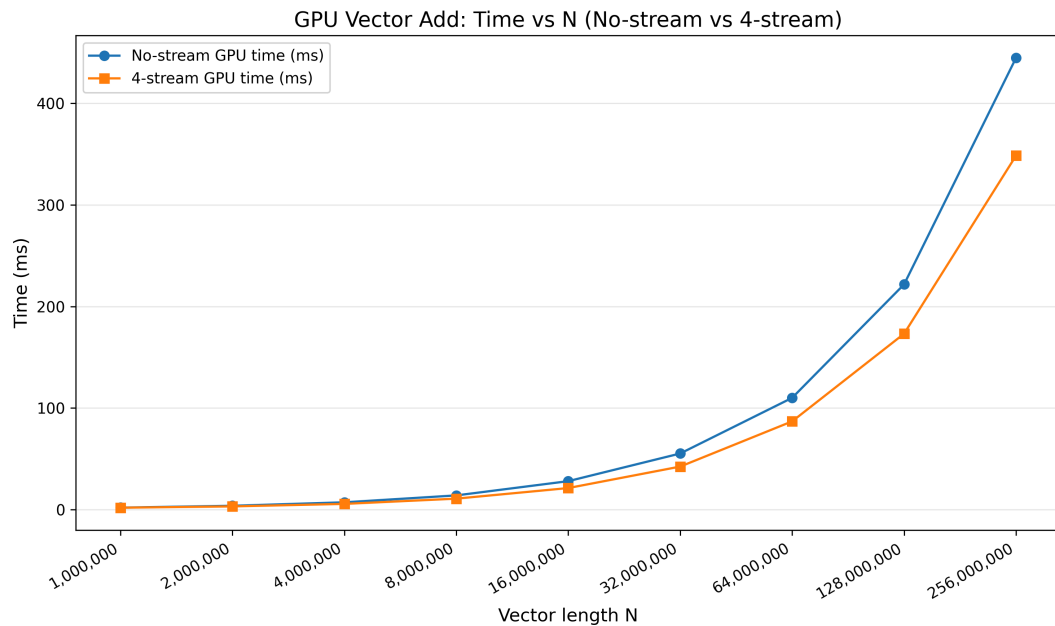
Non-streamed version – single `cudaMemcpy` H2D, single kernel launch, single `cudaMemcpy` D2H.

Streamed version – input vector divided into segments, processed with 4 CUDA streams using asynchronous H2D / kernel / D2H to overlap communication and computation.

For each vector length N , we measure the **total GPU time** (H2D + kernel + D2H) and define the speedup as

$$\text{speedup} = \frac{T_{\text{no-stream}}}{T_{\text{4-stream}}}$$

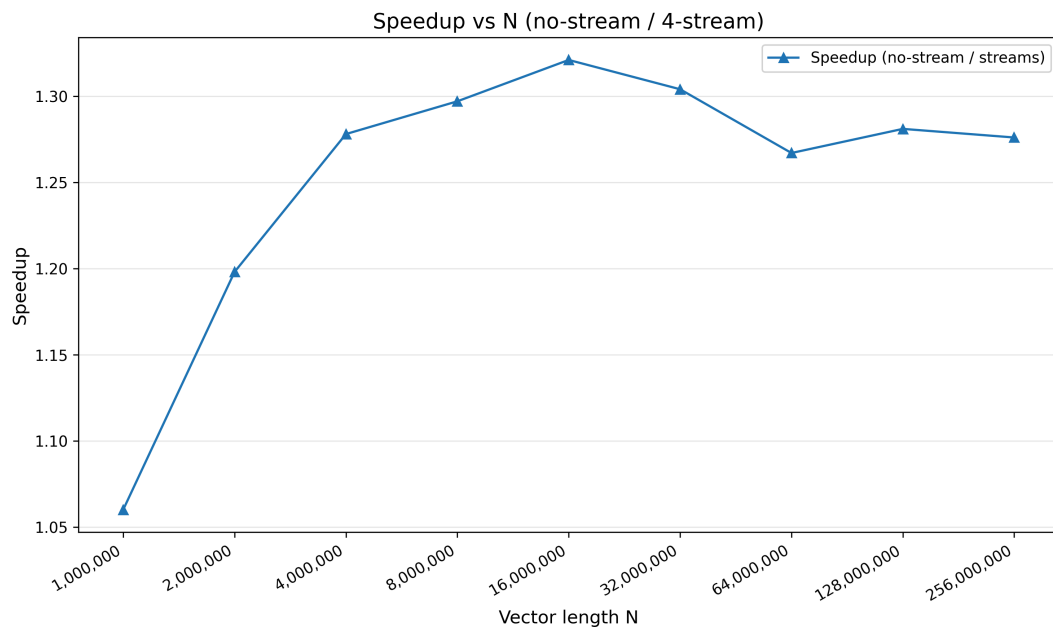
1. Time vs vector length



In the *Time vs N* plot:

- Both implementations scale roughly linearly with N , as expected for a bandwidth-bound $O(N)$ kernel.
- For all tested sizes, the 4-stream implementation is faster than the non-streamed version.
- The absolute gap between the two curves widens as N increases (for the largest N , the streamed version saves on the order of tens to over a hundred milliseconds).

2. Speedup vs vector length



In the *Speedup vs N* plot :

- For the smallest tested size $N = 10^6$, the speedup is about **1.06×**, the streamed implementation is only slightly faster than the non-streamed one. At this scale, kernel-launch and stream-management overheads are still comparable to the useful work, so the benefit from overlap is limited.

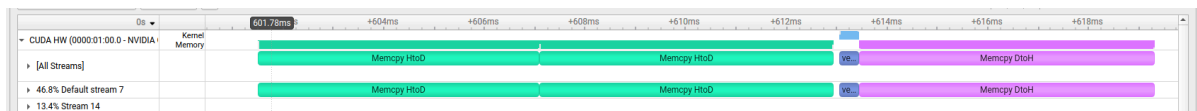
- As N increases, the speedup rises steadily and reaches a peak of about **1.32×** for $N \approx 1.6 \times 10^7$. In this regime, each segment contains enough work so that data transfers and kernel execution in different streams can effectively overlap, reducing the total execution time.
- For larger vectors, the speedup remains relatively stable in the range **1.27×**–**1.32×**, indicating a consistent performance gain of roughly **27–32%** for the 4-stream implementation compared to the non-streamed version at medium and large problem sizes.

3. Summary sentence for the report

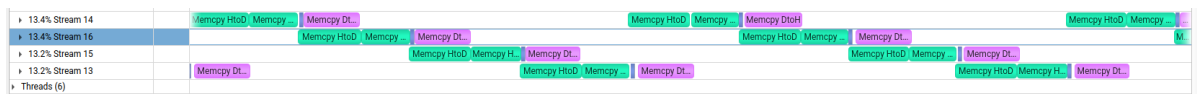
Using 4 CUDA streams for segmented asynchronous transfers and computation improves vector addition performance by roughly **27–32%** over the non-streamed implementation for medium and large vector sizes, with a modest $\approx 6\%$ gain at the smallest tested size.

- **Use nvprof to collect traces and the NVIDIA Visual Profiler (nvvp) to visualize the overlap of communication and computation.**

no streams



4 streams



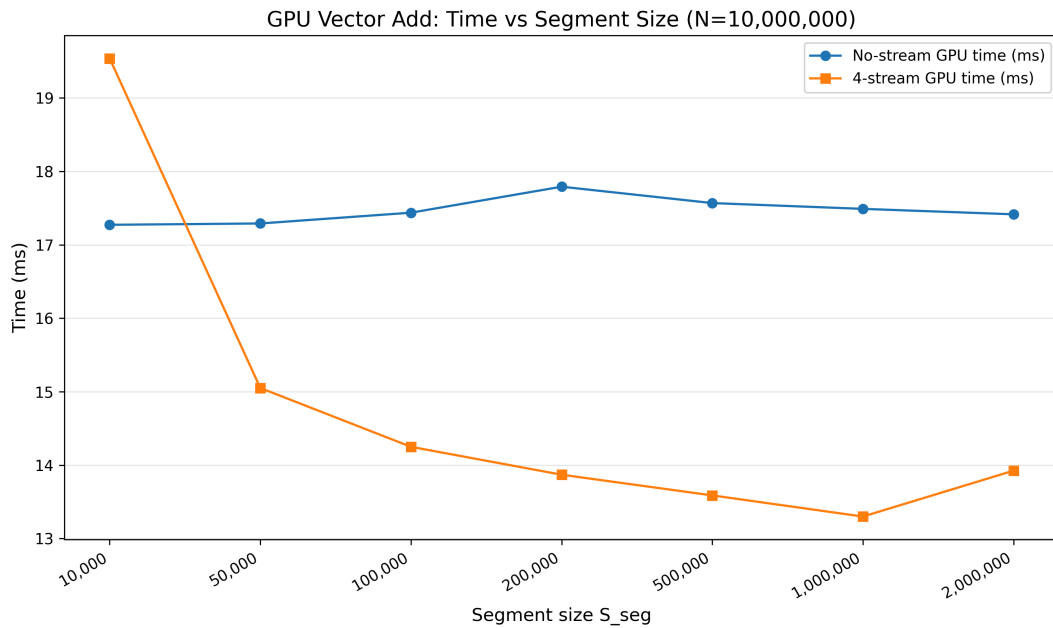
In the Nsight Systems timeline, the non-streamed version shows a single CUDA stream where the H2D memcopy, the `vecAdd` kernel and the D2H memcopy execute strictly one after another.

For the 4-stream version, the timeline displays four CUDA streams; H2D memcopies in one stream overlap with kernel execution and D2H memcopies in other streams, clearly indicating that communication and computation are overlapped.

- **What is the impact of segment size on performance? Present in a plot (you may choose a large vector and compare 4-8 different segment sizes)**

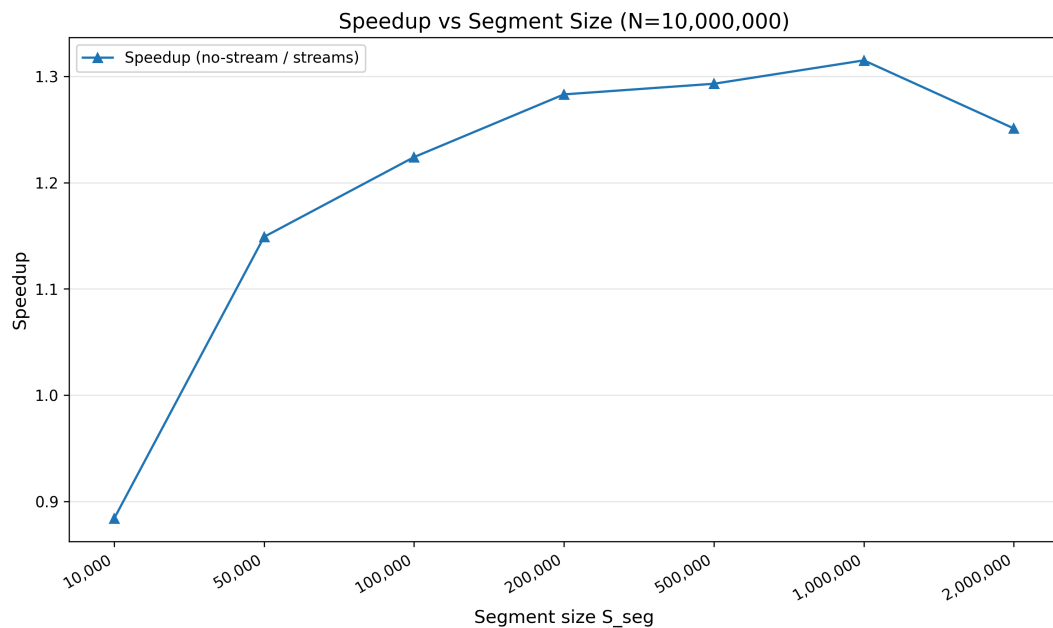
To study the impact of the segment size, we fixed the vector length to $N = 10\,000\,000$ and varied the segment size S_{seg} between 10^4 and 2×10^6 .

The results are shown in the *Time vs Segment Size* and *Speedup vs Segment Size* plots.



From the **Time vs Segment Size** plot we observe that:

- The **non-streamed** execution time is almost flat (around 17–18 ms) and does not depend on the segment size, since the baseline implementation does not use segmentation.
- For the **4-stream implementation**, the time **strongly depends on** S_{seg} . With very small segments, the streamed version is even **slower** than the baseline (~19.5 ms vs. ~17.3 ms). In this regime, the overhead of launching many small kernels and many asynchronous memcopies dominates, and there is little useful overlap.
- As S_{seg} increases, the 4-stream time steadily decreases and reaches a minimum of about **13.3–13.4 ms** for segment sizes around 5×10^5 – 10^6 elements. Here each segment is large enough to amortize launch and memcopy overheads while still providing enough segments to keep the pipeline full.



The **Speedup vs Segment Size** plot shows the same trend in terms of relative performance:

- For very small segments, the speedup is below 1 streams hurt performance. The speedup then rises with S_{seg} and peaks at about **1.3×** for $S_{\text{seg}} \approx 5 \times 10^5$ elements ($\approx 30\%$ faster than the non-streamed version).
- For very large segments, the speedup slightly drops again ($\approx 1.26\times$), because the number of segments is too small and the overlap between communication and computation becomes less effective.

In summary, the segment size has a significant impact on performance:

too small segments cause excessive kernel and memcpy overhead;

too large segments reduce the degree of overlap.