

ECE 408 Exam 2, Spring 2025, CNN Version  
Tuesday, May 6, 7:00 – 10:00 PM

Name: Heng An Cheng

NetID: hacheng2

UIN: 667692370

- Be sure your exam booklet has exactly **FOURTEEN** pages. You can use page 14 for any spill-over solutions, see instructions on that page.
- We designed 130 points of questions, but this exam is graded out of 100.
- Write your name at the top of each page; Do NOT tear the exam apart.
- This is a closed-book exam; NO handwritten notes are permitted.
- You may not use any electronic devices except for a simple calculator.
- Absolutely no interaction between students is allowed.
- Show all work, and clearly indicate any assumptions that you make.
- Assume all omitted code is implemented correctly.
- Illegible answers will likely be graded as incorrect.
- Don't panic, and good luck!

Problem 1 30 points \_\_\_\_\_

Problem 2 15 points \_\_\_\_\_

✓ Problem 3 15 points \_\_\_\_\_

✓ Problem 4 15 points \_\_\_\_\_

Problem 5 35 points \_\_\_\_\_

Problem 6 20 points \_\_\_\_\_

Name: Heng An Cheng

2

**Problem 1** (30 points): The Ideas of Massively Parallel Programming 2

**Part A** (14 points): For each of the following questions, provide an answer **without** any justification.

(A.1) According to Dr. Robert Searles, state the preferred software to understand the workload across multiple CPUs and GPUs.

Nsight System

(A.2) According to Dr. Katrina Riehl, state the Just-In-Time (JIT) compiler for Python code used to accelerate numeric functions on both CPUs and GPUs.

(A.3) In Neural Networks and Transformer models, what operation is responsible for calculating the mean and standard deviation, and then normalizing the row data to a different mean and standard deviation in a layer?

(A.4) Consider a  $4 \times 4$  matrix multiply-and-accumulate tensor core. How many multipliers and how many adders are needed to compute a single output element, considering the optimized approach from the lecture?

4 multipliers and 2 adders.

(A.5) For a sparse matrix-vector multiplication (SpMV) with  $R$  rows,  $C$  columns, a total of  $N$  non-zero elements, and a maximal of  $L$  non-zeros in each row, how many padding zeros will be added when we convert the matrix from CSR to ELL?

$$R \times L - N$$

$$R \times \boxed{L}$$

(A.6) What is the name of the job management system used on the Delta cluster at NCSA?

(A.7) Write the command to check the status of every job currently managed by the scheduler under your username on Delta. Alternatively, write it for a specific job ID.

**Part B** (4 points): Shengjie needs to perform an inclusive scan on an input of  $2^{34}$  elements. A customized GPU supports at most  $2^{12}$  blocks per grid and at most  $2^{10}$  threads per block. Using Brent-Kung in a hierarchical fashion, with none of the scan work done by the host, how many times does he need to launch the Brent-Kung kernel? Show your work.

1st	$\frac{2^{34}}{2^{10+1}} = 2^{23}$ (blocks)	$\Rightarrow \frac{2^{23}}{2^{12}} = 2^{11}$ (grids)	total launch $= 2^{11} + 1 + 1 + 1$ $= 2051$ #
2nd	$\frac{2^{23}}{2^{11}} = 2^{12}$ (blocks)	$\Rightarrow \frac{2^{12}}{2^{12}} = 1$ (grids)	
3rd	$\frac{2^{12}}{2^{11}} = 2$ (blocks)	$\Rightarrow 1$ grids	
4th	$1$ (blocks)	$\Rightarrow 1$ grids	

Name: Hong An Cheng

3

**Problem 1, continued:**

**Part C (4 points):** IN NO MORE THAN THIRTY WORDS, briefly explain why you wish to use joint register and shared memory tiling, besides the low latency of registers.

Using joint register tiling can reduce atomic operations on shared memory, making it faster than just shared memory tiling.




**Part D (4 points):** As mentioned in lecture, what is the best technique for storing sparse matrices with roughly random non-zero entries scattered throughout? Explain your answer IN NO MORE THAN THIRTY WORDS.

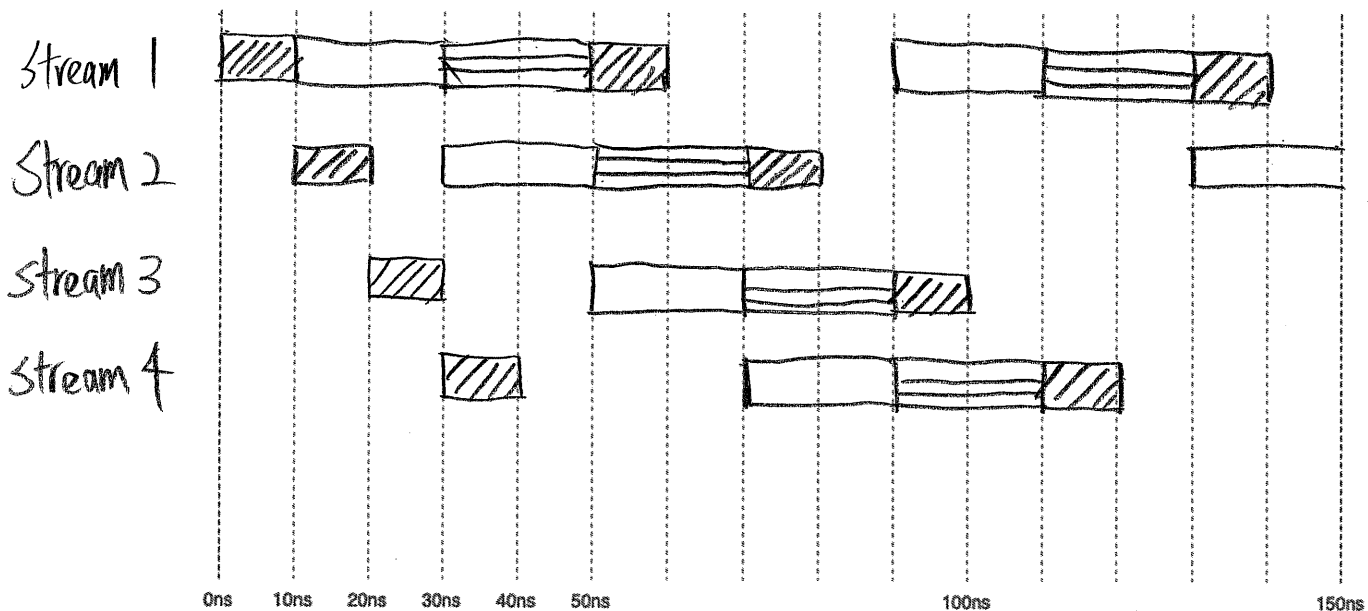
CSR/ELL, because it is roughly random, we can expect almost the same number of non-zero data within the rows.

**Part E (4 points):** Vijay provided the following kernel function prototype:

```
__global__ void magic(float* inputA, float* inputB, float* output, int size);
```

Vijay chose to have 4 streams. According to profiling results, and per API call it takes 10 ns per chunk to transfer memory from host to device, 20 ns per chunk to transfer memory from device to host, and 20 ns per chunk for the provided kernel launch and execution. Draw the expected flow of functions that each stream is performing for the first 150 ns.

 H to D     kernel     D to H



Name: Heng An Cheng

4

**Problem 2 (15 points): CNN Project**

Select either **True** or **False** for each statement, and provide a **one-sentence** justification for your response. Each statement will be graded as follows:

- **+3 points:** Correct answer with adequate justification
- **+2 points:** Correct answer with insufficient justification
- **+1 point:** Correct answer without justification
- **0 points:** Blank/incorrect answer and/or justification

(2.A) Throughput is the amount of work performed by the GPU within a certain time frame, also called achieved occupancy. ☒ True ☐ False

By definition of throughput.

(2.B) Converting matrix operations from FP32 to TF32 (Tensor Float 32) in NVIDIA GPUs delivers performance improvements by storing values in a compressed 19-bit format, which reduces memory footprint by approximately 40% compared to standard FP32 operations. ☒ True ☐ False

FP32 stores values in 32-bit format.

$$\frac{19}{32} \approx 60\% \Rightarrow 40\% \text{ reduces}$$

(2.C) When Yifei is profiling to verify whether CUDA streams' data transfer and computation are overlapping, he should use Nsight Compute with command `nsys profile --stats=true ./m3` to check the detailed timeline information. ☐ True ☒ False

the command is wrong, should be `ncu`

(2.D) In Milestone 1, transforming convolution operations into matrix unrolling followed by matrix multiplication delivers significant performance gains, even without kernel fusion or any other optimizations, because modern GPUs are architecturally designed and highly optimized for dense matrix computation workloads. ☐ True ☒ False

For milestone 1 result, the directly convolution operation is faster than unrolling version

(2.E) In the CNN Project, to avoid the trouble of manually calling WMMA APIs while leveraging the performance benefit from kernel fusion, we can potentially use cuBLAS to accelerate the fused kernel. ☒ True ☐ False

Both are using tensor cores to accelerate matmul

Name: Heng An Cheng

5

### Problem 3 (15 points): Kogge-Stone

To prepare for ECE 508, Colin hastily reviewed the lectures for this class and breezed over a common oversight for Kogge-Stone discussed in lecture. To help him out, you decided to modify his code to be the error-free implementation of the algorithm.

In this problem, you will complete the KoggeScan kernel using the Kogge-Stone algorithm as outlined in lecture with a **double buffering approach**. Your task is to fill in each of the blanks in the following code, complete the missing TODO in the function, and add proper synchronizations.

```
#define CHUNK_SIZE 1024 // blockDim.x
```

```
__global__ void KoggeScan(float* in, float* out, int inputSize) {
```

```
    __shared__ float buffer[2][CHUNK_SIZE];
```

```
    int i = blockDim.x * blockIdx.x + threadIdx.x;
```

```
    if (i < inputSize) {
```

```
        buffer[0][threadIdx.x] = in[i];
```

```
    } else {
```

```
        buffer[0][threadIdx.x] = 0;
```

```
    int current_buffer = 0;
```

```
    int next_buffer = 1;
```

```
    for (int stride = 1; stride < blockDim.x; stride *= 2) {
```

```
        if (threadIdx.x >= stride) {
```

```
            buffer[next_buffer][threadIdx.x] = buffer[current_buffer][threadIdx.x] + buffer[current_buffer]  
            [threadIdx.x - stride]
```

```
            // TODO: Update current_buffer and next_buffer
```

```
            current_buffer = (current_buffer + 1) % 2
```

```
            next_buffer = (next_buffer + 1) % 2
```

```
        }
```

```
    if (i < inputSize) {
```

```
        out[i] = buffer[current_buffer][CHUNK_SIZE - 1];
```

```
    }
```

```
}
```

Name: Heng An Cheng

6

#### Problem 4 (15 points): Streams

Hrishi is working on a vector database project for undergrad research, and he has written a dot product kernel to compute dot products of many vectors in parallel, named `vector_dot_product_host`. In this problem, you will complete the host function of `vector_dot_product_host` using streams.

The **host function prototype** is as follows:

```
void vector_dot_product_host(float* A, float* B, float* C, int m, int n);
```

This host function computes the row-by-row dot product between A and B, storing the result in C.

A, B, and C are host pointers. A and B are matrices of size  $m \times n$ . C is an array of m values, where `C[i]` will contain the dot product of row i of A and row i of B. A, B, and C are already allocated or stored in pinned memory.

The **kernel function prototype** is as follows:

```
__global__ void vector_dot_product_kernel  
(float* A, float* B, float* C, int m, int n);
```

This kernel dot products vectors stored in A and B in chunks, and stores the result in C. The kernel function stores a single scalar per dot product. Each thread block launches `BLOCK_SIZE` threads and computes `BLOCK_SIZE` number of dot products.

**\*\*When using macros, always reference the macro names instead of hardcoding their values.\*\***

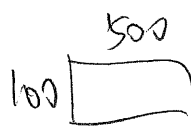
**\*\*Your code must work for any block size and number of vectors.\*\***

The host function uses 3 CUDA streams to overlap data transfer and computation. To achieve this, the batch is divided into *chunks*. Each chunk has `CHUNK_SIZE` output dot products. These chunks are assigned to the 3 streams in a round-robin fashion. For example, if there are 8 chunks, then Chunks 0, 3, 6 are assigned to Stream 0, Chunks 1, 4, 7 are allocated to Stream 1, and Chunks 2, 5 are assigned to Stream 2.

**\*\*Your task is to complete the host function on the next page.\*\***

*The remainder of this page has been intentionally left blank.*

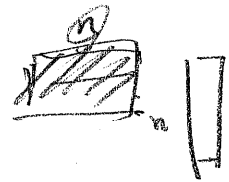
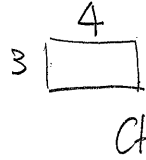
Name: Heng An Cheng



$$\frac{500 \times 100}{500} = 100$$

Problem 4, continued:

$m \times n$



```
#define BLOCK_SIZE 64
#define CHUNK_SIZE 5000
```

```
void vector_dot_product_host(float* A, float* B, float* C, int m, int n) {
    float *device_A; cudaMalloc(&device_A, m * n * sizeof(float));
    float *device_B; cudaMalloc(&device_B, m * n * sizeof(float));
    float *device_C; cudaMalloc(&device_C, m * sizeof(float));
    cudaStream_t streams[3];
    // TODO: Initialize streams
```

```
    for(int i=0; i<3; i++) cudaStreamCreate(&streams[i]);
```

```
    for (int chunk = 0; chunk < ceil(1.0 * m / floor(1.0 * CHUNK_SIZE / n)); ++chunk) {
```

```
        cudaStream_t stream = streams[chunk % 3];
```

```
        int current_chunk_size = min(CHUNK_SIZE, n - chunk * CHUNK_SIZE)
```

```
        int offset = (chunk - 1) * CHUNK_SIZE + current_chunk_size;
```

```
        cudaMemcpyAsync(device_A + offset, A + offset,
                        current_chunk_size * sizeof(float),
                        cudaMemcpyHostToDevice, stream);
```

```
        cudaMemcpyAsync(device_B + offset, B + offset,
                        current_chunk_size * sizeof(float),
                        cudaMemcpyHostToDevice, stream);
```

```
        dim3 grid_dim(ceil(1.0 * m / BLOCK_SIZE));
```

```
        dim3 block_dim(BLOCK_SIZE);
```

```
        // TODO: Launch the kernel and copy output to host
```

```
        vector_dot_product_kernel <<< grid_dim, block_dim,
```

```
                                device_A + offset,
                                device_B + offset, device_C + offset, stream>>>
        cudaMemcpyAsync(C + offset, device_C + offset, current_chunk_size * sizeof(float),
```

```
        // TODO: Wait for streams to complete work and destroy streams
```

```
        for(int i=0; i<3; i++) {
```

```
            cudaStreamSynchronize(streams[i]);
```

```
            cudaStreamDestroy(streams[i]);
```

```
        }
        cudaFree(device_A); cudaFree(device_B); cudaFree(device_C);
    }
```

Name: Heng An Cheng

8

**Problem 5** (35 points): Help Howie! VI: Plus Sign Sum Again?

Sometimes in life, things might look negative — a tough week, a rough grade, or just one too many deadlines. But if you take a step back and look at the big picture, you'll realize that even if things seem negative at first glance, they'll end up helping you grow into something greater than what you started with!

Inspired by the philosophy of finding strength through setbacks, you decide to help Howie implement the **Minus Sign Sum** (a new version of the Plus Sign Sum from Exam 1 that Daksh derived), which is a 2D matrix-to-vector operation that looks like a minus sign.

The operation is defined as follows: Given an  $N \times N$  (square) input matrix stored in row-major order, the Minus Sign Sum outputs a vector of size  $N$  where each element  $out_i$  is calculated by summing all elements in the  $i$ -th row of the input matrix.

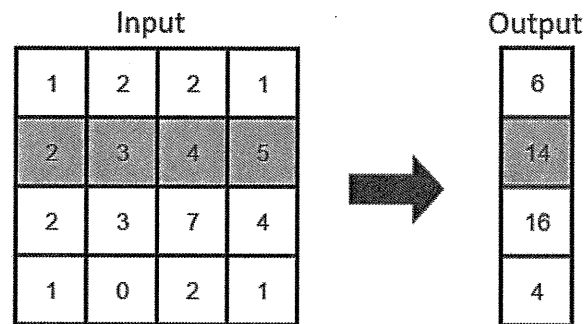


Figure 1: Example of Input and Output of the Minus Sign Sum Operation

Howie has left a few notes for you:

- You must employ the **improved** version of the **reduction** strategy covered in lecture, which utilizes shared memory and minimizes branch divergence.
- Each thread in your kernel should load exactly **two** elements from the input matrix into shared memory.
- Input matrix dimension  $N$  can be any positive value.
- Each block of threads use 1 row of the input matrix to calculate 1 result value in the output vector.
- The output of the kernel is the final Minus Sign Sum Operation output (as illustrated above).

**Part HL** (0 points): Howie really liked almost all your restaurant recommendations from Exam 1! Therefore, he would now like to try out all the good places for a quick snack/drink in the Champaign-Urbana area. Please let him know what your favorite shop is (bakery, coffee shop, etc):

StarBucks

**Part A** (15 points): Write the kernel code that performs the Minus Sign Sum on the next page.

**Note:** Howie really appreciates your recommendations from Exams 1 and 2. Therefore, he decided to provide you the kernel launch configuration of `minus_sign_sum_kernel`:

```
dim3 dimGrid((N - 1) / 512 + 1, N);
dim3 dimBlock(256);
minus_sign_sum_kernel<<<dimGrid, dimBlock>>>(...);
// All other host code omitted, you may assume that they are correct,
// please implement the kernel code on the next page.
```



Name: Heng An Cheng

9

Problem 5, continued:

```
// N is the width of the square input matrix
// in points to the input matrix data (properly allocated)
// out points to where output vector needs to be stored (properly allocated)
__global__ void minus_sign_sum_kernel(float* in, float* out, int N) {
```

```
    int bx = blockIdx.x; int by = blockIdx.y; int tx = threadIdx.x;
```

```
    __shared__ float sum[2*256];
```

```
    int id = blockIdx.x*512 + tx + by*N;
```

```
    if (id < N) sum[tx] = in[id];
```

```
    else sum[tx] = 0
```

```
    if (id+256 < N) sum[tx+256] = in[id+256];
```

```
    else sum[tx+256] = 0; -- syncthreads();
```

```
    for (int stride = 1; stride < 256; stride *= 2) {
```

```
        int index = (id+1)*(stride*2-1);
```

```
        if (index < 2*256 && index >= stride) {
```

```
            sum[index] += sum[index-stride];
```

```
        }
```

```
        -- syncthreads();
```

```
    }
```

```
    for (int stride = 128; stride > 0; stride /= 2) {
```

```
        int index = (id+1)*(stride*2-1);
```

```
        if (index >= 0 && index+stride < N) {
```

```
            sum[index] += sum[index+stride];
```

```
        }
```

```
        -- syncthreads();
```

```
    }
```

```
    // Continue at the back.
```

Name: Heng An Cheng

10

### Problem 5, continued:

After a filling meal at Bangkok Thai, Howie took a look at your Minus Sign Sum kernel implementation and realized that the Plus Sign Sum can be obtained using multiple calls of the Minus Sign Sum kernel! He immediately tasked Charles with writing the host code that computes the Plus Sign Sum using only Minus Sign Sum kernel calls.

However, after writing the below host code, Charles realized that he would need another custom kernel to convert the outputs of the two Minus Sign Sum kernels into the output of a Plus Sign Sum kernel.

The definition and code implementation of the Plus Sign Sum from Exam 1 are included in the appendix on page 15.

**Part B (2 points):** Knowing you're an expert in CUDA, Charles passed the task down to you. Fill in the blanks in the below host code to launch an additional kernel that produces the output of a Plus Sign Sum for the given input using the results of the two (2) Minus Sign Sum kernel calls.

```
// --- In Properly Allocated DEVICE Memory: ---
// d_inp points to the input matrix data
// d_inp_T points to the transposed input matrix data
// d_out points to where the Plus Sign Sum output needs to be stored
// -----
// N is the width of the input square matrix
void plus_sign_sum(float* d_inp, float* d_inp_T, float* d_out, int N) {
    float *d_row_sums; cudaMalloc((void **) &d_row_sums, sizeof(float) * N);
    float *d_col_sums; cudaMalloc((void **) &d_col_sums, sizeof(float) * N);

    // Initialize sums to zero
    cudaMemset(d_row_sums, 0, sizeof(float) * N);
    cudaMemset(d_col_sums, 0, sizeof(float) * N);

    dim3 dimGrid((N - 1) / 512 + 1, N);
    dim3 dimBlock(256);

    minus_sign_sum_kernel
        <<<dimGrid, dimBlock>>>(d_inp, d_row_sums, N);
    minus_sign_sum_kernel
        <<<dimGrid, dimBlock>>>(d_inp_T, d_col_sums, N);

    // You will implement this kernel in Part C
    dim3 dimGrid2((N * N - 1) / 128 + 1);
    dim3 dimBlock2(128);
    minus_to_plus_kernel<<<dimGrid2, dimBlock2>>>
        (d_inp, d_row_sums, d_col_sums, d_out, N);
    cudaDeviceSynchronize();

    cudaFree(d_row_sums); cudaFree(d_col_sums);
}
```

Name: Heng An Chong

11

**Problem 5, continued:**

**Part C** (10 points): Write the `minus_to_plus` kernel used in **Part B**.

**Note:** Each thread in the kernel should produce **at most 1** output value.

```
// Refer to the host code in Part C for parameter details
__global__ void minus_to_plus_kernel(float* in, float* row_sums,
                                     float* col_sums, float* out, int N) {
    int bx = blockIdx.x; int tx = threadIdx.x;
```

```
    int id = bx * blockDim.x + tx;
```

```
    if (id < N) {
```

```
        out[id] = row_sums[id] + col_sums[id]
                  - in[id];
```

```
    }
```

```
}
```

**Part D** (8 points): After some basic profiling, Howie discovers that the end-to-end performance of the approach described in **Part B** is not as good as he thought, despite the use of an efficient reduction algorithm.

Provide two (2) potential explanations on why compared to the Plus Sign Sum implementation from Exam 1 (included in appendix), this host + kernel code implementation might not yield good performance. Explain **EACH IN NO MORE THAN THIRTY WORDS**.

Name: Heng An Cheng

12

**Problem 6 (20 points): Histograms**

Prof. Kindratenko is optimizing parallel histogramming algorithms, and he has written the following histogramming kernels. `global_histogram` is a non-privatized global memory histogram. `shared_histogram` implements a shared-memory privatized histogram like the one you wrote in lab 7. `joint_histogram` adds another "layer" of privatization using local memory.

Assume the following about all kernels:

- Each atomic operation in the global memory has a constant total latency of 100ns, and each atomic operation in a shared memory has a total latency of 1ns.
- `int32_t` is a 32-bit signed integer, and `uint32_t` is a 32-bit unsigned integer.

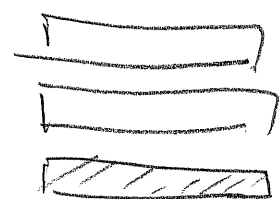
```
1 #define HISTOGRAM_LENGTH 32
2 /* data: the input array of integers you wish to histogram.
3 * histogram: a global histogram array that should store the results after the kernel is finished
4 * input_len: length of data
5 */
6 __global__ void global_histogram(uint32_t* data, uint32_t* histogram, uint32_t input_len) {
7     uint32_t tIdx = blockIdx.x * blockDim.x + threadIdx.x;
8     uint32_t stride = blockDim.x * gridDim.x;
9     for (uint32_t i = tIdx; i < input_len; i = i + stride) {
10         atomicAdd(&(histogram[data[i]]), 1);
11     }
12 }
13
14 __global__ void shared_histogram(uint32_t* data, uint32_t* histogram, uint32_t input_len) {
15     uint32_t tIdx = blockIdx.x * blockDim.x + threadIdx.x;
16     uint32_t stride = blockDim.x * gridDim.x;
17     __shared__ uint32_t shHistogram[HISTOGRAM_LENGTH];
18     for (int32_t i = threadIdx.x; i < HISTOGRAM_LENGTH; i += blockDim.x) {
19         shHistogram[i] = 0;
20     }
21     __syncthreads();
22     for (uint32_t i = tIdx; i < input_len; i = i + stride) {
23         atomicAdd(&(shHistogram[data[i]]), 1);
24     }
25     __syncthreads();
26     for (int32_t i = threadIdx.x; i < HISTOGRAM_LENGTH; i += blockDim.x) {
27         atomicAdd(&(histogram[i]), shHistogram[i]);
28     }
29 }
30
31 __global__ void joint_histogram(uint32_t* data, uint32_t* histogram, uint32_t input_len) {
32     uint32_t tIdx = blockIdx.x * blockDim.x + threadIdx.x;
33     uint32_t stride = blockDim.x * gridDim.x;
34     uint32_t localHistogram[HISTOGRAM_LENGTH];
35     __shared__ uint32_t shHistogram[HISTOGRAM_LENGTH];
36     for (int32_t i = threadIdx.x; i < HISTOGRAM_LENGTH; i += blockDim.x) {
37         shHistogram[i] = 0;
38     }
39     for (int32_t i = 0; i < HISTOGRAM_LENGTH; ++i) {
40         localHistogram[i] = 0;
41     }
42     __syncthreads();
43     for (uint32_t i = tIdx; i < input_len; i += stride) {
44         ++localHistogram[data[i]];
45     }
46     for (int32_t i = 0; i < HISTOGRAM_LENGTH; ++i) {
47         atomicAdd(&(shHistogram[i]), localHistogram[i]);
48     }
49     __syncthreads();
50     for (int32_t i = threadIdx.x; i < HISTOGRAM_LENGTH; i += blockDim.x) {
51         atomicAdd(&(histogram[i]), shHistogram[i]);
52     }
53 }
```

240 x 512 threads  
↓

3200000

261 x 261 threads.

10 32



3200

0 ~ 512

261

100

01

Stride = 30

512

32

per block

Name: Heng An Cheng

15 blocks  
13

512

**Problem 6, continued:**

**Part A** (12 points): Yue decides to benchmark the all three kernels with an input array of length 32,000,000. All three kernel are launched with  $\text{dimGrid} = 240$  and  $\text{dimBlock} = 512$ . For **Part A**, only consider the runtime of the atomic operations. Show your work.

Threads

(A.1) What is the theoretical minimum runtime of `global_histogram`?

→ share

$\frac{160}{512}$

$$32000000 \times 100 \text{ ns} = 32 \text{ s} \#$$

32 threads

(A.2) What is the theoretical minimum runtime of `shared_histogram`?

$$32000000 \times 1 \text{ ns} + 240 \times 32 \times 100 \text{ ns} \quad (\text{per block})$$
$$= 0.320768 \text{ s} \#$$

(A.3) What is the theoretical minimum runtime of `joint_histogram`?

$$32 \times 1 \text{ ns} \times 240 \times 512 + 240 \times 32 \times 100 \text{ ns}$$

(per thread) (per block)

$$= 0.00470016 \text{ s}$$

**Part B** (8 points): Xiyue applied the kernel to another dataset with `HISTOGRAM_LENGTH = 4000`, and he noticed with the same `input_len`, `dimGrid`, and `dimBlock` in **Part A**, `joint_histogram` is much slower than `shared_histogram`. Provide two (2) potential explanations **EACH IN NO MORE THAN THIRTY WORDS**.

1. Maximum number of registers per thread = 255 < 4000, too much registers might cause error and make it slower.
2.  $4000 \times 1 \times 240 \times 512 > 32000000$   
 $\Rightarrow$  make it slower #

Name: Benz An Cheng

14

Use this single-sided page to write any spill-over solutions. Clearly label which problem each solution belongs to and write "SEE LAST PAGE" on the corresponding problem's page. This is the only additional page we will scan and grade along with the single-sided problem pages. IF YOU WRITE YOUR SOLUTION ON THE BACK SIDE OF ANY OTHER PAGE, WE WILL NOT SCAN AND GRADE IT!