ECE 408 Exam 2, Spring 2018

April 23, 2018

- You are allowed one 8.0x11.5 cheat sheet with notes on both sides. The minimal font size for your text on the cheat sheet should be 8pts.

16

- No interactions with humans other than course staff are allowed.
- This exam is designed to take 150 minutes to complete. To allow for any unforeseen difficulties, we will be giving everyone up to 180 minutes.
- This exam is based on lectures, textbook chapters, as well as lab MPs/projects.
- The questions are randomly selected from the topics we covered.
- You can write down the reasoning behind your answers for possible partial credit.
- You must write your answers with pen in order to request regrade.

Good luck!

Name: ________________________

Netid: ________________________

UIN: ________________________

Question 1: (25 points) ______________________

Question 2: (15 points) ______________________

Question 3: (15 points) ______________________

Question 4: (15 points) ______________________

Question 5: (15 points) ______________________

Question 6: (15 points) ______________________

Name: ____

Question 1 (25 points, 30 minutes): multiple-choice and short-answer questions. If you get more than 25 points by answering all questions (1-9), your score will saturate at 25 points.

For multiple-choice questions, give a concise explanation for your answer for possible partial credit. Answer each of the short-answer questions in as few words as you can. Your answer will be graded based on completeness, correctness, and conciseness.

1. (3 points) For the Brent-Kung scan kernel based on reduction trees and inverse reduction trees, assume that we have 2048 elements, which of the following values is the total levels of the reduction tree and the inverse reduction tree?

   (A) 7
   (B) 11
   (C) 21
   (D) 25

   Answer: (C)
   Explanation: For N elements, the reduction tree has log(N) levels and the inverse reduction has log(N) - 1 levels. The total is 2*log(N) -1. For 2048 elements, the answer is 2*log(2048)-1 = 2*11-1 = 21.

2. (3 points) For the Brent-Kung scan kernel based on reduction trees and inverse reduction trees, assume that we have 2048 elements in each section and warp size is 32, how many warps in each block will have control divergence during the reduction tree phase iteration where stride is 16?
   (A) 0
   (B) 1
   (C) 16
   (D) 32

   Answer: (A)
   Explanation: All active threads are consecutive starting with index 0. There are a total of 1024 threads and 64 active threads in each block when the stride is 16. The active threads form entire warps 0 and 1. All other warps are completely inactive. No warps have control divergence.

Name: ________________________________

3. (3 points) For a processor that supports atomic operations in L2 cache, assume that each atomic operation takes 5ns to complete in L2 cache and 500ns to complete in DRAM. What is the approximate hit rate of the L2 cache needed to achieve a throughput of 1/250 G operations per second?
   (A) 99%
   (B) 75%
   (C) 50%
   (D) 25%

   Answer: (C)
   Explanation: The desired latency is 250ns. A little more than half of the atomic operations need to hit in cache to achieve this latency. So, the hit rate should be about 50%.

4. (3 points) Given a sparse matrix of integers with R original rows, L non-zero elements in the original row with the largest number of non-zeros, and a total of N non-zeros. How many more integers are needed to represent the matrix in JDS as compared to CSR?
   (A) R
   (B) 2N
   (C) L
   (D) N

   Answer: (A)
   Explanation: We need R additional integers to keep track of the original row number for each row after sorting.

5. (3 points) For a sparse matrix-vector multiplication (SpMV) with 100,000 rows, a total of 300,000 non-zero elements, and a maximal of 10 non-zeros in each row, how many additional zero elements will be added when we convert the matrix from CSR to ELL?
   (A) 5
   (B) 300,000
   (C) 700,000
   (D) 1,000,000

   Answer: (C)
   Explanation: In ELL, all rows will be filled to 10 elements. This will make the matrix 10*100,000 =1,000,000 elements for ELL. CSR will have only 300,000 elements. The difference is 1,000,000 - 300,000 = 700,000.

Name: ________________________________

6. (3 points) For a sparse matrix-vector multiplication (SpMV) with 100,000 rows, 10,000 columns, a total of 300,000 non-zero elements, how many times on average will each vector element be used?
   (A) 1
   (B) 3 on average
   (C) 30 on average
   (D) 100,000

   Answer: (C)
   Explanation: A total of N accesses will be made to the vector elements. There are C of them. So each us used N/C = 300,000/10,000 = 30 on average.

7. (3 points) Keven has a 640MB array that he would like to process with GPU. He measured that the execution time of the code on CPU was 0.02 seconds. He also implemented a kernel and measured that the kernel execution on the GPU with the data in the GPU memory was 0.0004 seconds, a 50x speedup! However, he needs to transfer the data into the GPU memory. There is negligible data to be transferred back to the CPU. His system has a PCIe Gen3 x16 interconnect. What would the real speedup be?
   (A) 40x speedup
   (B) 20x speedup
   (C) 5x speedup
   (D) 0.5x speedup (100% slow down)

   Answer: (D)
   Explanation: The data copy will take (640M)/(16*1000M) = 0.04 sec to copy the input data to GPU. Thus the total speedup is 0.02/(0.04+0.0004) ≈ 0.02/(0.04) = 0.5x speedup, or 100% slow down

8. (3 points) For the following host code sequence:

```
      1) cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),.., stream0);
      2) cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),..,
      stream1);
      3) cudaMemcpyAsync(d_A2, h_A+i+2*SegSize, SegSize*sizeof(float),..,
stream2);
      4) cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),.., stream0);
      5) cudaMemcpyAsync(h_C+i+SegSize, d_C1, SegSize*sizeof(float),..,
      stream1);
      6) cudaMemcpyAsync(h_C+i+2*SegSize, d_C2, SegSize*sizeof(float),..,
      stream2);
```

Which of the statements could be executed in parallel on the GPU

(A) 1) and 2)
(B) 2) and 3)
(C) 2) and 4)
(D) 4) and 5)

Answer: (C)
Explanation: API operations in different streams can be executed in parallel. However, there is only one PCIe copy engine in each direction. So, only 2) and 4) can go in parallel among the choices given

9. (3 points) In the following OpenACC code, which of the following is false?
(A) Statement 1 will be executed redundantly executed by the the 32 gangs.
(B) The n iterations of loop i will be divided and distributed to the 32 gangs for execution.
(C) Statement 2 will be executed a total of n times.
(D) Statement 3 will be executed a total of m times.

```
#pragma acc parallel num_gangs(32)
{
    Statement 1;
    #pragma acc loop gang
    for (int i=0; i<n; i++) {
     Statement 2;
    }
    for (int j=0; j<m; j++) {
     Statement 3;
    }
}
```

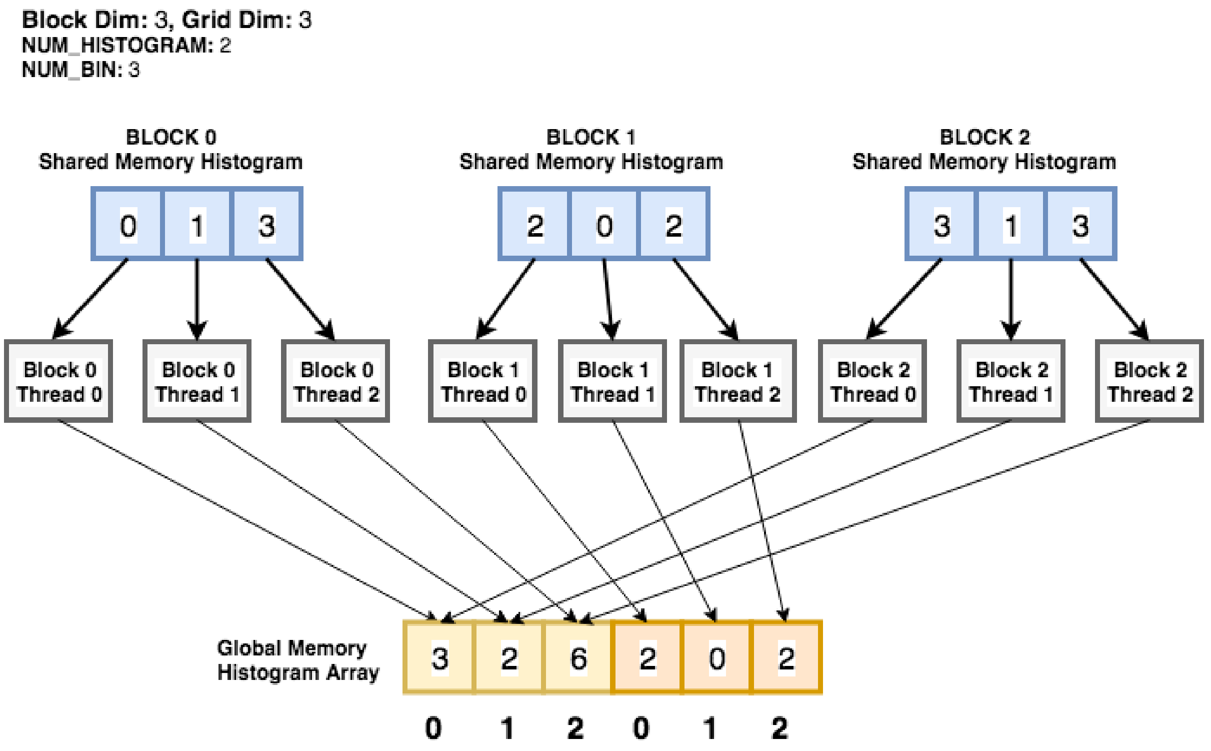
Answer: (D)
Explanation: The j loop is in a parallel eregion but not marked as an acc Loop, thus the entire loop will be redundantly executed by all the 32 gangs. Satement 3 will be executed 32*m times.

Name: ______________________________________

Question 2 (15 points, suggested time allocation 25 minutes): This question tests your understanding of parallel histogram computation and privatization.

You are part of a team that tallies up the voting result of a president's election. The electron was between **four** different candidates. You have 300 million of votes that you need to process. Knowing GPUs can process the mass amount of data, you decided to create a **modified** version of the histogram kernel you learned in class to count how many votes each candidate got.

In the modified version of the histogram kernel, instead of having one global histogram, you have an array of global histograms so that you can have less contention on the global histograms. The global histogram array index is calculated as blockIdx.x % NUM_HISTOGRAM. Carefully analyze the host code as it will help. Below is a visualization of the modified kernel for a small example of three candidates, three thread blocks and an array of two global histograms.



(A) (3 Points) Complete the following kernel to implement the modified histogram. Note that each thread block will still have its own privatized histogram in the shared memory.

```
/* histo_kernel is launched with the following parameters */
  #define NUM_VOTERS        300000000 // 300 million
  #define NUM_HISTOGRAM     16
  #define NUM_BIN           4
```

Name: ______________________________________

```
    unsigned int histogram_array[NUM_HISTOGRAM * NUM_BIN] = {0};
    unsigned int histogram[NUM_BIN] = {0};

    dim3 gridDim(10000);
    dim3 blockDim(1000);
    histo_kernel<<<DimGrid, DimBlock>>>(votes, NUM_VOTERS, histogram_array);

    for(int i = 0; i <NUM_HISTOGRAM; i++) {
        for(int j = 0; j < NUM_BIN; j++) {
            histogram[j] += histogram_array[i*NUM_BIN + j];
        }
    }

__global__ void histo_kernel(unsigned int *votes, long size, unsigned int
*histo_array){
    __shared__ unsigned int histo_private[__NUM_BIN_(+.5)_];

// Each element of the votes[] array contains an integer that selects one of
// the candidates. It value ranges from 0 to NUM_BIN-1

     int i = threadIdx.x + blockIdx.x * blockDim.x;
      int stride = __blockDim.x * gridDim.x_(+.5)_;

     // Reset histogram
     if( threadIdx.x <NUM_BIN)
         histo_private[threadidx.x] = 0;
     __syncthreads();

     while (i < size) {
      atomicAdd(&(histo_private[__votes[i]_(+.5)_]) , 1);
      i += stride;
     }
     __syncthreads();

    // contribute to one of the global histograms
    if( threadIdx.x <NUM_BIN){
      int histo_index = blockIdx.x % NUM_HISTOGRAM;
      atomicAdd(&(histo_array[_histo_index * NUM_BIN + threadIdx.x_(+1)_]),
                  histo_private[__threadIdx.x_(+.5)_]);
    }
}
```

(B) (2 Point) How many times does the 500th thread of the 25th block(threadIdx.x = 500, blockIdx.x =25) iterate in the **while loop**?

Answer: 30

Explanation: 300000000 / (10000 * 1000)

(C) (2 Points) How many **non-atomic global memory reads/writes** and **shared-memory/global-memory atomic operations** are being performed by **all the threads** executing the kernel?

Non-atomic Global Memory reads: 300,000,000

Non-atomic Global Memory writes: 0

Shared-memory atomic operations: 300,000,000

Global-memory atomic operations: 10000 * 4

Explanation:

For the following questions, consider only the atomic operations in the process of analyzing the kernel code. Assume that

- each atomic operation in the global memory has a constant total latency of 100ns.
- each atomic operation in a shared memory has a total latency of 1ns.

(D) (2 Points) If the votes were evenly spread out (25% each), what is the theoretical minimum runtime of the **original histogram kernel** (i.e., NUM_HISTOGRAM = 1)?

Answer: 1.0075 ms

Explanation: (1000*30)* .25 * 1 ns + 10000 * 100 ns = 10e6 + 7.5e3 ns

(E) (2 Points) If the votes were evenly spread out (25% each), what is the theoretical minimum runtime of the **modified histogram kernel**?

Answer: 0.07 ms

Explanation: (1000*30) * .25 * 1 ns + 10000 / 16 * 100 ns = 6.25e4 + 7.5e3 ns

(F) (2 Points)If the vote resulted in a distribution of (90%, 5%, 5%, 0%), what is the theoretical minimum runtime of the **original histogram kernel**?

Answer: 1.027 ms

Explanation: The execution time of the privatized histogram phase of each thread block will be dominated by the atomic operations to the most popular candidate bin. The time to contribute to the global histogram will remain the same as the case with even distribution. (1000*30) * .9 * 1ns + 10000 * 100 ns

(G) (2 Points) If the vote resulted in a distribution of (80%, 5%,5%,10%), what is the theoretical minimum runtime of the **modified histogram kernel**?

Answer: 0.0865 ms
Explanation: (1000*30) * .8 * 1ns + 10000 / 16 * 100 ns

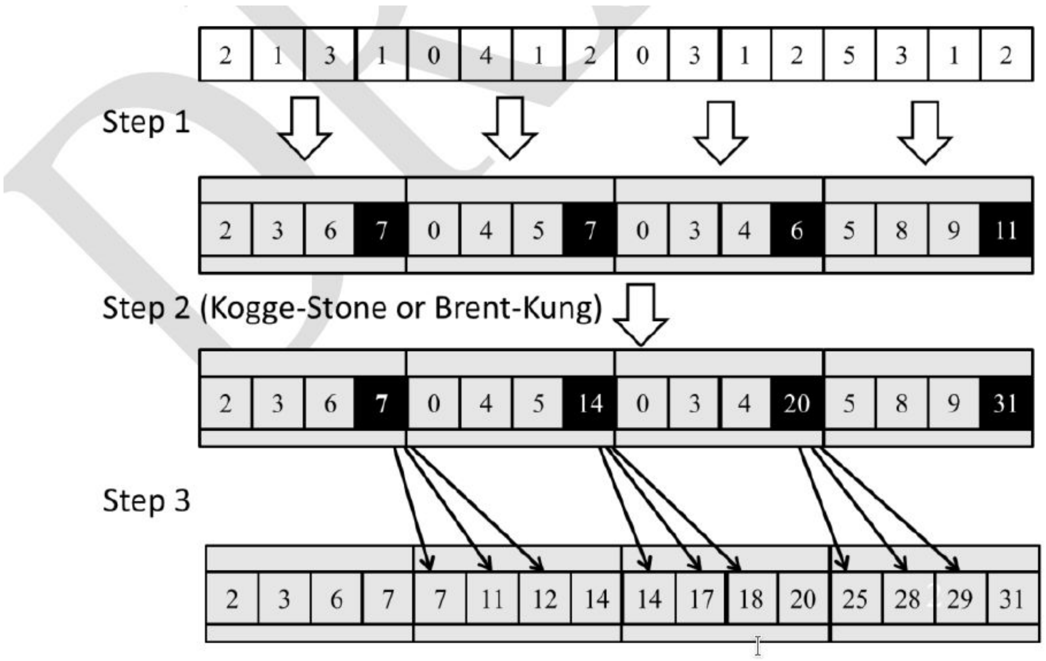
(H) (Extra Credit 1 Point) What technique could further help with a dataset that has a large concentration of identical data values in localized areas in the histogram kernel? (Hint: This technique was discussed in the textbook.)
Aggregation

Name: ______________________________________

Question 3: Scan (15 points, suggested time allocation 25 minutes):

To further optimize the scan kernel, we can perform the scan operations more efficiently by further dividing the first kernel of the prefix-sum MP 5.2 into three steps. Note that all the three steps are performed within one kernel.

I. In Step 1 (see the figure below), each thread operates on its own small section and perform sequential scan in the shared memory. The number of these small sections are the same as the number of threads in a block. (The figure below shows a small example where each block has 4 threads, each thread handles 4 elements, and each block is handling 16 elements).
II. In Step 2, all threads preform Kogge-Stone scan **only** on every last element of the small sections in the shared memory.
III. In Step 3, each threads will add the last element (partial sum of its own section) to the elements in the next section, except for the last one since it's already computed in the second step.



This algorithm only completes the scan on the data belongs each block. In order to perform a complete scan, there will be another two kernels adding partial sums to data of other blocks (like the hierarchical scan in MP5.2). This is just an attempt to make the per-block scan kernel more efficient by having each thread to handle more than one element.

For this question, we will be focusing only on the scan kernel described above. You can also assume that the number of input data elements is a multiple of the number of threads or the size of the shared memory (i.e, don't worry about the boundary conditions).

Name: ______________________________________

(A) (6 points) Fill in the blanks to complete the scan kernel described above. (Hint: use the figure when you analyze the kernel code.)

```
#define TILE_WIDTH 1024  // Number of threads in each block
#define SHARE_LIMIT 4*1024 // Total number of elements being scanned in each block

// Number of elements in each small section to be processed by each thread
#define NUM_PER_SEC (SHARE_LIMIT/TILE_WIDTH)

__global__ void scan(float* arr)
{
    // Shared memory to hold input elements for each block
    __shared__ float scanShare[SHARE_LIMIT];

    // Explicitly store the last element of each small sub-section
    __shared__ float sectionEndShare[TILE_WIDTH];

    size_t tx = threadIdx.x;
    size_t offset = blockIdx.x * SHARE_LIMIT;
    size_t subsection_start = tx*NUM_PER_SEC;

    // Step 1

    // - Each thread will perform a scan on its own small section
    for (size_t i = 0; i < NUM_PER_SEC; i++) {
        size_t currIdx = subsection_start + i;
        scanShare[currIdx] = arr[offset + currIdx];
        if (i > 0){
            scanShare[currIdx] += scanShare[currIdx - 1];
        }
    }
    sectionEndShare[tx] = scanShare[(tx+1)*NUM_PER_SEC - 1];

    // Step 2
    // - Kogge-Stone scan on the end of each section
    for (size_t stride = 1; stride < TILE_WIDTH; stride *= 2) {
        __syncthreads();
        if (tx >= stride) {
            sectionEndShare[tx] += sectionEndShare[tx - stride];
        }
    }
    scanShare[(tx+1)*NUM_PER_SEC - 1] = sectionEndShare[tx];

    // Step 3
    // Add partial sums to necessary elements
    __syncthreads();
    for (size_t i = 0; i < NUM_PER_SEC - 1; i++) {
```

Name: ___________________________________

```
        if (tx > 0){
            scanShare[subsection_start + i] += sectionEndShare[tx - 1];
        }
    }

    // Write results back to global memory
    __syncthreads();
    for (size_t i = 0; i < NUM_PER_SEC; i++) {
        size_t currIdx = subsection_start + i;
        arr[offset + currIdx] = scanShare[currIdx];
    }
}

// Code on host side:

// num is the number elements in the input array
int numBlocks = (num-1) / (SHARE_LIMIT) + 1;

dim3 dimBlock(TILE_WIDTH, 1, 1);
dim3 dimGrid(numBlocks, 1, 1);
scan<<<dimGrid, dimBlock>>>(dev_arr);

// Assume there will be other hierarchical scans to complete scan on the entire
array
...
```

(B) (5 points) After completing this kernel, you found out that the scan results is incorrect. It turns out that the implementation of Kogge-Stone in Step 2 is buggy. Explain what the problem is and rewrite the Kogge-Stone part such that it is correct.

Explanation:

New Kogge-Stone:

```
    // - Kogge-Stone scan on the end of each section
    for (size_t stride = 1; stride < TILE_WIDTH; stride *= 2) {
        float temp = sectionEndShare[tx];
        ________________________________________;
        if (tx >= stride){
        ________________________________________;
        }
        ___________________________________________;
        sectionEndShare[tx] = temp;
    }
```

Solution:

```
     // - Kogge-Stone scan on the end of each section
    for (size_t stride = 1; stride < TILE_WIDTH; stride *= 2) {
        float temp = sectionEndShare[tx];
        __syncthreads();
        if (tx >= stride){
            temp += sectionEndShare[tx - stride];
        }
        __syncthreads();
        sectionEndShare[tx] = temp;
    }
```

(C) (4 points) When reading (step 1) and writing data (step 3) from/to global memory, there are still room for improvement. In order to further optimize it, we can first load all the data in a coalesced manner and then perform scan (this is what we call corner turning technique in the class). We can then have better memory coalescing and fewer control divergence.

Fill in the blanks such that the access pattern for reading memory (step 1) is coalesced. (Writing to memory in step 3 can be done similarly.) Note that all of the following code is still in the kernel.

```
    // Step 1
    //  - Load data into shared memory
    for (size_t i = tx; i < SHARE_LIMIT; i+=TILE_WIDTH) {
        scanShare[i] = arr[offset + i];
    }
    __syncthreads();

    // - Each thread will perform a scan on its own small section
    for (size_t i = 1; i < NUM_PER_SEC; i++) {
        int currIdx = subsection_start + i;
        scanShare[currIdx] += scanShare[currIdx - 1];
    }
```

(D) (3 points) Calculate the total number of arithmetic FLOPs in step 2 per block (i.e, only considering the Kogge-Stone part). Choose only one answer.
Below are some equations that you might need:

Name: ____________________________________

$log_2(1024) = 10,\ log_2(4096) = 12$

$1 + 2 + ... + N/2 = N - 1$

a. $4096 * 10$
b. $4096 * 12$
c. $4096 * 1024$
d. $4096 * 2 - 2 - 12$
e. $4096 * 10 - (4096 - 1)$
f. $4096 * 12 - (4096 - 1)$

$\sum (N - Stride)$ for stride = 1, 2, …, N/2

$$\sum_{i=0}^{log_2(N)-1} (N - 2^i) = N * log_2(N) - (N - 1) = 4096 * 12 - (4096 - 1)$$

Is it possible perform scan on the same amount of data (4*1024 floating point numbers) using Kogge-Stone kernel directly in a single block? Why or why not?

No. It would require 4096 threads which exceeds the maximum number of threads per block.

(E) (2 points) We already learned in class that both Kogge-Stone and Brent-Kung scan kernel can speedup the scan operations. What is the main drawback of the Kogge-Stone scan kernel? What is the main drawback of the Brent-Kung scan kernel?

Kogge-Stone: The use of extra hardware resources could be very problematic. If the number of input elements is large, then the efficiency will be limited by the hardware resources.

Brent-Kung: The active threads doing "useful" work drops quickly thus wasting hardware resources. / Control divergence.

Name: ________________________________

Question 4. Sparse Matrix Multiplication (15 points, suggested time allocation 25 minutes):

This question tests your knowledge of Sparse Matrix representation and operation. We first give you the kernel codes for CSR, ELL, JDS, and JDS-T formats (You may not need to read through all the kernels. They are here help you to recall what we discussed in lecture). Based on the given code, please answer the multiple-choice questions and short answer questions below. Please note that in this problem, you have to give explanation for each question for full points.

CSR Kernel:

```
__global__ void SpMV_CSR(int num_rows, float *data,
    int *col_index, int *row_ptr, float *x, float *y) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < num_rows) {
      float dot = 0;
      int row_start = row_ptr[row];
      int row_end   = row_ptr[row+1];
      for (int elem = row_start; elem < row_end; elem++) {
        dot += data[elem] * x[col_index[elem]];
      }
      y[row] = dot;
    }
  }
```

ELL Kernel:

```
__global__ void SpMV_ELL (int num_rows, float *data, int *col_index, int
num_elem, float *x, float *y) {
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  if (row < num_rows) {
    float dot = 0;
    for (int i = 0; i < num_elem; i++) {
           dot += data[row+i*num_rows] * x[col_index[row+i*num_rows]];
    }
    y[row] = dot;
   }
}
```

JDS Kernel:

```
__global__ void SpMV_JDS(int num_rows, float *data,
  int *col_index, int *jds_row_ptr,int *jds_row_perm,
  float *x, float *y) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < num_rows) {
      float dot = 0;
      int row_start = jds_row_ptr[row];
      int row_end   = jds_row_ptr[row+1];
      for (int elem = row_start; elem < row_end; elem++) {
```

```
8.          dot += data[elem] * x[col_index[elem]];
          }
9.        y[jds_row_perm[row]] = dot;
        }
      }
```

JDS-T Kernel:

```
1.__global__ void SpMV_JDS_T(int num_rows, float *data,
  int *col_index, int *jds_t_col_ptr, int *jds_row_perm,
  float *x, float *y) {
2.      int row = blockIdx.x * blockDim.x + threadIdx.x;
3.      if (row < num_rows) {
4.        float dot = 0;
          unsigned in sec = 0;
5.        while (jds_t_col_ptr[sec+1]-jds_t_col_ptr[sec] > row){
6.          dot += data[jds_t_col_ptr[sec]+row]*
                   x[col_index[jds_t_col_ptr[sec]+row]];
7.          sec++;
          }
8.        y[jds_row_perm[row]] = dot;
        }
      }
```

(A) (2 points) If we want to have a sparse matrix multiplication kernel which minimizes the control divergence between threads (given number of threads generated is a multiple of 32), which kernel above is our **best** option and give explanation?

CSR **[✓ ELL]** **JDS** **JDS-T**

Explanation: By padding in the ELL format which makes every row has the same number of element, the ELL format has no control divergence when the number of threads generated is a multiple of 32.

(B) (3 points) In terms of memory accessing, generally speaking, which kernel (or kernels) access the memory for the input matrix (the data array in this problem) in a coalesced manner? (Circle all possible choices and give explanation)

CSR **[✓ ELL]** **JDS** **[✓ JDS-T]**

Explanation: ELL format and JDS-T format have the coalesced memory access which means adjacent thread access the adjacent memory location.

For the following questions, consider the original dense form of matrix = $\begin{bmatrix} 3 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 0 & 0 & 0 & 9 \end{bmatrix}$.
Assuming all the kernels are invoked using:

```
dim3 BlockDim (4, 1, 1);
```

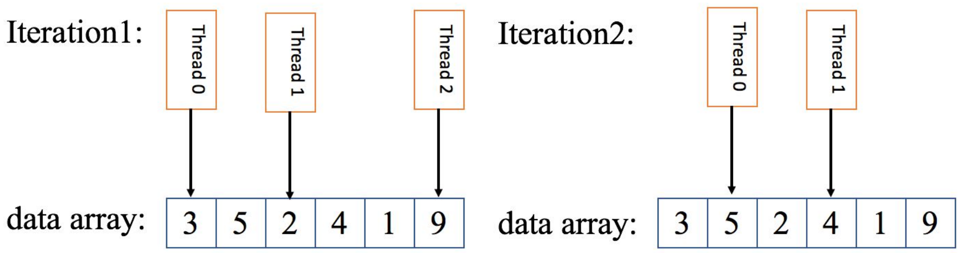
Name: ______________________________

```
dim3 GridDim (1, 1, 1);
```

Draw a diagram for each of the following sparse matrix format to show how the data is stored and the access pattern of all threads in first two iterations.

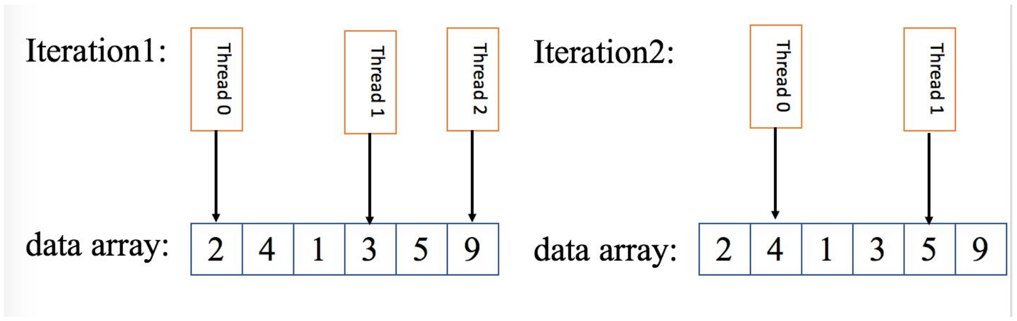
We give you an example for the **CSR** format. You need to draw similar diagrams for **JDS**, and **JDS-T** formats. For each part, give explanation for full points.

(Example) (0 point) **CSR** format:
Diagram:



Explanation: For the CSR format, the data array compresses all zero elements and store non-zeros row-by-row. Each thread in CSR takes care of one row in the original matrix and for each iteration, each thread calculates one element in that row.

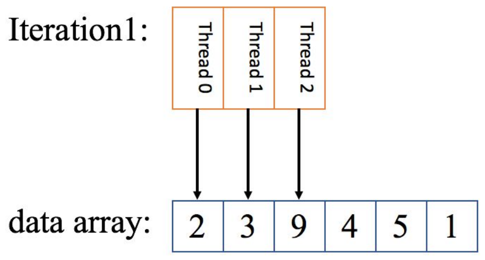
(C) (5 points) **JDS** format:
Diagram:

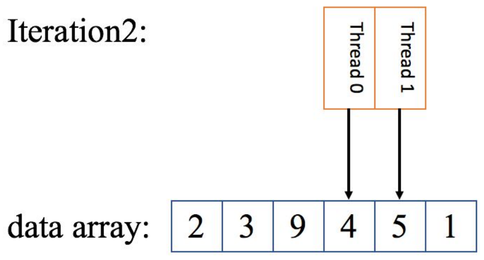


Explanation: From CSR to JDS, we sort the rows according to the length (say longest->shortest). Thus, the order of the data array would be changed accordingly. The thread access pattern is like CSR format: each thread in CSR takes care of one row in the original matrix and for each iteration, each thread calculates one element in that row.

(D) (5 points) **JDS-T** format:
Diagram:

Name: ______________________________





Explanation: For the JDS-T format, we further transpose the data in JDS format and use a column pointer to store the start position of each row after the transpose. After this transposition, each thread still access same set of element but with memory coalesced access pattern.

Name: ______________________________

Question 5. Convolutional Neural Network(CNN) (15 points, suggested time allocation 25 minutes):

A basic convolution layer consists of filter W, input X and output Y. We want to accelerate the forward propagation of convolution layers in the training process.

- W is the convolution filter weight tensor, organized a tensor W[M, C, K, K], where
    - M is the number of output feature maps,
    - C is the number of input feature maps,
    - K is the height and width of each filter.

Tensors are stored as multi-dimensional arrays in the memory.

- X is the input feature map, organized as a tensor X[B, C, H, W], where
    - B is the number of images in one mini-batch (recall that mini-batch is used to efficiently updates gradients while keeping relatively fast convergence),
    - H is the height of each input feature map and
    - W is the width of each input feature map.
- Y is the output feature map, organized as a tensor Y[B, M, H_out, W_out], where
    - H_out = H - K + 1 is the height of each output feature map and
    - W_out = W - K + 1 is the width of each output feature map.

For all questions below assume the CNN is implemented using convolution.

(A) (2 points) Consider the following 2 declarations for the block dimension:
dim3 WHDim(W_out, H_out, 1);
dim3 HWDim(H_out, W_out, 1);
Which declaration would cause the CNN to execute quicker and why?

Answer: WHDim because since the minor (vertical) dimension is mapped to the consecutive threads in the x-dimension, memory accesses will be coalesced

(B) (3 points) Consider the following 2 declarations for the block dimension where the first one maps threads to the output feature map elements and the second one maps threads to the input feature map elements:
dim3 outputDim(W_out, H_out, 1);
dim3 inputDim(W, H, 1);
If you use shared memory in your kernel, name one advantage of each declaration.

Answer: outputDim maps to the output so it will use fewer threads, this also has less control divergence if using shared memory and strategy 2. inputDim maps to the input so it can load the shared memory in a single pass and gives inherent memory coalescing in the load as opposed to outputDim which would have to be adjusted

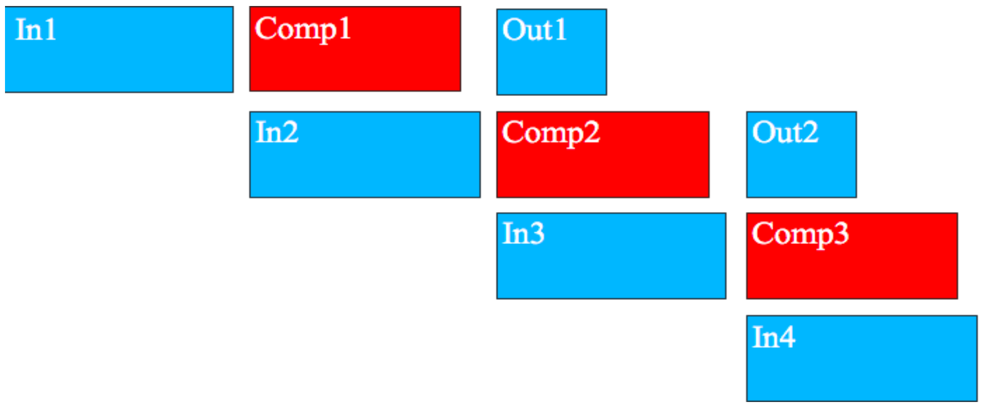
Name: ____________________________________

(C) (3 points) Consider the case where the convolution filter weight tensor W is too big to fit into constant memory. We decide to instead put it into shared memory and call it Wshared. Assuming the weights are single-precision floating-point numbers, what is the minimum amount of shared memory in bytes we need to allocate for each block and how many copies of Wshared will be allocated and loaded in the kernel execution with the following block and grid dimensions.

```
dim3 gridDim(B, M, C);
dim3 blockDim(W_out, H_out, 1);
```

Answer: Wshared needs a minimum of K * K * 4 bytes large since we parallelize across batch, output feature, and input feature so we only need a single piece of the filter. We will generate B * M * C copies of Wshared which is the number of blocks.

(D) (3 points) Consider the case where we decide to use 3 CUDA Streams and are not given the memory already copied on the device for us like in the project. In the case where the memory transfer of the required input for a given computation takes longer than the computation itself and the time to transfer the output is even shorter, which of the following input memory section sizes will have the lowest execution time assuming no time overhead for launching of kernels for each section and that the weight tensor W is already in the device constant memory? Explain why. This is illustrated for you below.



a) sizeof(float) * B * C * H * W bytes
b) sizeof(float) * B/3 * C * H * W bytes
c) sizeof(float) * 1 * C * H * W bytes
d) all of the above are equivalent

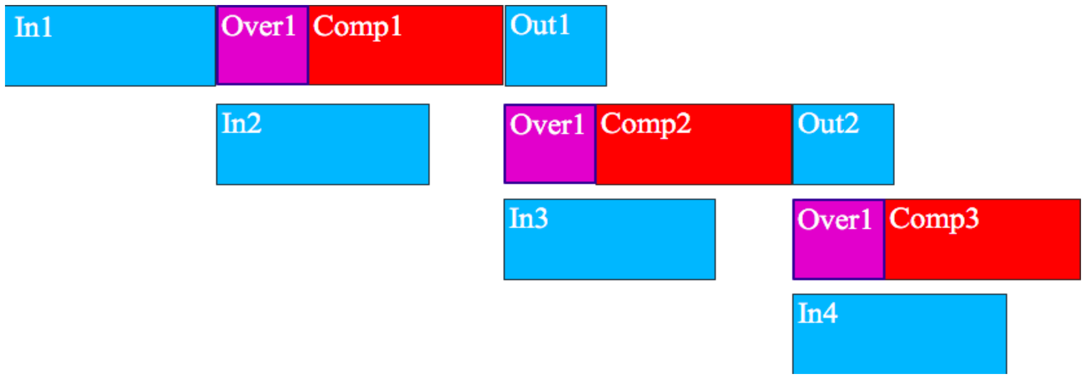
Answer: c. For the case of a, once we load all of X we can compute the entire output so this is equivalent to the case of a single stream and will take the longest. For cases b and c, the computation's time cost and output transfer time cost for each

Name: ______________________________________

stream iteration is masked by the input data transfer meaning the length of time overall will be the time it takes to load all of the input, plus the time it takes to perform the computation of 1 section, and finally the time it takes to transfer the last section. This means that the overall time is minimized by having smaller sections, so c will run faster.

Name: ____________________________________

(E) (4 points) Consider now that we have a constant cost overhead to begin a computation and it is large enough that the the computation plus the overhead for an input section takes longer than the memory transfer of the input section. Assuming B is 15 and the time to transfer the entire input is 45 seconds and the time to transfer the entire output is 15 seconds, will your answer change for certain sized overhead? Explain why and how long the time the overhead must be if yes or why not if no. This is illustrated for you below.



Answer: It can. (b) will run faster than (c) if the overhead takes longer than 4/3 seconds. (a) will run faster than (c) if the overhead takes longer than 4 seconds. (a) will run faster than (b) if the overhead takes longer than 20 seconds. From the previous scheme, now the input transfer time of the next stream is masked by the computation of the current stream, meaning the total time will be the time it takes to perform the entire computation, plus the time it takes to transfer the first input section and the last output section. The total execution time can be defined as Computation + #Sections * Overhead + (Input + Output Transfer) / # Sections. Thus the difference between any two schemes with X and Y sections is (X - Y) * Overhead + (Y - X)(Input + Output) / XY, which results in a trade off between (X - Y) * Overhead and (Y - X)(Input + Output) / XY. If (X - Y) * Overhead > (X - Y)(Input + Output) / XY, then the overhead is too costly and Y is better than X. Putting in X as 15 and Y as 3, we get 45 * Overhead > 60, or Overhead > 4/3 to switch from c to b. If we put X as 15 and Y as 1, we get 14 * Overhead > 14 * 4 or Overhead > 4 to switch from c to a. If we put X as 3 and Y as 1 we get 2 * Overhead > 2 * 20 or Overhead > 20 to switch from b to a. Note that this answer is independent of how long the computation takes and only depends on the number of sections and time for memory transfer and overhead. This is because we are trying to balance the benefit of cutting the transfer into sections vs. the cost of adding more overhead.

Name: ______________________________

Question 6. Use of visual profiler to analyze the performance of kernel execution. (15 points, suggested time allocation 20 minutes):

The profile shown on the next page was generated using nvprof and the NVIDIA Visual Profiler:

- It is annotated with two timestamps and four durations.
- It is a matrix multiplication of a [250x80] [80x1000] = [250x1000] matrix of `floats`, requiring approximately 40 million floating-point operations.

The times have been adjusted to make the math easier without a calculator.

You may find the following equalities useful when computing the answers

8 / 7 = 1.14
32/7 = 4.58
8 / 2.5 = 3.2

(A) (3 points) How many GFLOPS (billion floating-point operations per second) does the GPU kernel achieve?

4e7 flop / 2.5e-4s = 1.6e11 flops = 160 GFlops

(B) (4 points) From the perspective of the host, the *performance* is the number of floating-point operations divided by the elapsed time between the beginning of the first cudaMemcpy to the end of the last cudaMemcpy. What is the *performance* of this matrix multiplication?

4e10 fllops = 4e7 flop / 1e-3s = 40 GFlops

(C) (4 points) If the kernel was optimized to produce a speedup of 2, what would the *peZXs*

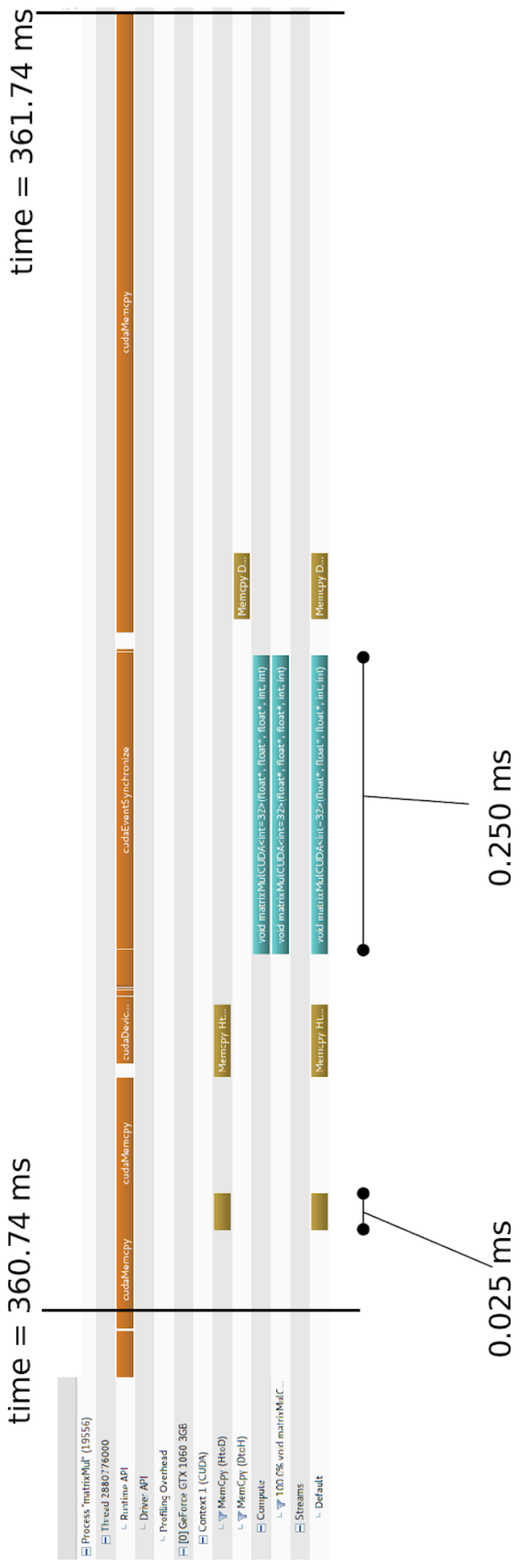
4e7 flop / ((7/8) * 1e-3) = 32/7 * 1e10 flops = 45.7 GFlops
OR: 1.14x speed up

(D) (4 points) Estimate the host-to-device performance on the CPU to GPU link, assuming the [250x80] `float` matrix was copied using the annotated (0.025ms) cudaMemcpy

8e4 bytes / 2.5e-5s = 8e9 / 2.5 = 3.2e9 B/s (or 3.2 GB/s)
OR: 0.8 G floating point per second

(0 pt if no unit)
(no partial credit for this question)

Name: ______________________________________


time = 360.74 ms
time = 361.74 ms
0.025 ms
0.250 ms
Process "matrixMul" (19556)
Thread 2880776000
Runtime API
Driver API
Profiling Overhead
[0] GeForce GTX 1060 3GB
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
100.0% void matrixMulC...
Streams
Default
cudaMemcpy
cudaMemcpy
cudaDevic...
cudaEventSynchronize
cudaMemcpy
Memcpy Ht...
Memcpy D...
void matrixMulCUDA<int=32>(float*, float*, float*, int, int)
void matrixMulCUDA<int=32>(float*, float*, float*, int, int)
void matrixMulCUDA<int=32>(float*, float*, float*, int, int)
Memcpy Ht...
Memcpy D...