ECE 408 Exam #1 Study Guide, Fall 2019

1. Exam format

- 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.
- 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 give everyone 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.

2. Topics to Review from Lectures

2.1. CUDA Basic concepts

- Mapping thread index into data index
- Memory hierarchies – characteristics and usage of each type of memory

1. We need to use each thread to calculate one output element of a vector addition. Assume that variable i should be the index for the element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index?
   (A) i=threadIdx.x + threadIdx.y;
   (B) i=blockIdx.x + threadIdx.x;
   (C) i=blockIdx.x*blockDim.x + threadIdx.x;
   (D) i=blockIdx.x * threadIdx.x;

2. We want to use each thread to calculate two (adjacent) output elements of a vector addition. Assume that variable i should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?
   (A) i=blockIdx.x*blockDim.x + threadIdx.x +2;
   (B) i=blockIdx.x*threadIdx.x*2
   (C) i=(blockIdx.x*blockDim.x + threadIdx.x)*2

(D) i=blockIdx.x*blockDim.x*2 + threadIdx.x

3. We want to use each thread to calculate two output elements of a vector addition. Each thread block processes 2*blockDim.x consecutive elements that form two sections. All threads in each block will first process a section first, each processing one element. They will then all move to the next section, each processing one more element. Assume that variable i should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?
   (A) i=blockIdx.x*blockDim.x + threadIdx.x +2;
   (B) i=blockIdx.x*threadIdx.x*2
   (C) i=(blockIdx.x*blockDim.x + threadIdx.x)*2
   (D) i=blockIdx.x*blockDim.x*2 + threadIdx.x

4. For a vector addition, assume that the vector length is 8000, each thread calculates one output element, and the thread block size is 1024 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements. How many threads will be in the grid?
   (A) 8000
   (B) 8196
   (C) 8192
   (D) 8200

5. If we want to allocate an array of v integer elements in CUDA device global memory, what would be an appropriate expression for the second argument of the cudaMalloc() call?
   (A) n
   (B) v
   (C) n * sizeof(int)
   (D) v * sizeof(int)

6. If we want to allocate an array of n floating-point elements and have a floating-point pointer variable d_A to point to the allocated memory, what would be an appropriate expression for the first argument of the cudaMalloc() call?
   (A) n
   (B) (void *) d_A
   (C) *d_A
   (D) (void **) &d_A

7. If we want to copy 3000 bytes of data from host array h_A (h_A is a pointer to element 0 of the source array) to device array d_A (d_A is a pointer to element 0 of the destination array), what would be an appropriate API call for this in CUDA?
   (A) cudaMemcpy(3000, h_A, d_A, cudaMemcpyHostToDevice);
   (B) cudaMemcpy(h_A, d_A, 3000, cudaMemcpyDeviceTHost);
   (C) cudaMemcpy(d_A, h_A, 3000, cudaMemcpyHostToDevice);
   (D) cudaMemcpy(3000, d_A, h_A, cudaMemcpyHostToDevice);

8. How would one declare a variable err that can appropriately receive returned value of a CUDA API call?
   (A) int err;
   (B) cudaError err;
   (C) cudaError_t err;
   (D) cudaSuccess_t err;

1. Answer: (C)
Explanation: This is the linearized global thread index for the whole grid.

2. Answer: (C)
Explanation: Every thread covers 2 consecutive elements. The starting data index is simply twice the global thread index. Another way to look at it is that all previous blocks cover (blockIdx.x*blockDim.x)*2. Within the block, each thread covers 2 elements so the beginning position for a thread is 2* threadIdx.x. beyond what is covered by all the previous blocks.

3. Answer: (D)
Explanation: All previous blocks cover (blockIdx.x*blockDim.x)*2. The beginning element is consecutive in this case so just add threadIdx.x to it.

4. Answer: (C)
Explanation: ceil(8000/1024.0)*1024 = 8 * 1024 = 8192. Another way to look at it is the minimal multiple of 1028 to cover 8000 is 1024*8 = 8192.

5. Answer: (D)
Explanation: This one should be self-evident.

6. Answer: (D)
Explanation: &d_A is pointer to a pointer of float. To convert it to a generic pointer required by cudaMalloc() should use (void **) to cast it to a generic double-level pointer.

7. Answer: (C)
Explanation: See Lecture slides.

8. Answer: (C)
Explanation: See Lecture slides.

2.2. Parallel Kernel Execution Concepts

- Using multi-dimensional thread indices to easily access multi-dimensional data structures
- Boundary condition checking and control divergence
- Thread capacity and thread block capacity of Streaming Multiprocessor

1. We are to process an 800X600 (800 pixels in the x or horizontal direction, 600 pixels in they or vertical direction) picture with the PictureKernel in Lecture slides:

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {

 // Calculate the row # of the d_Pin and d_Pout element to process
 int Row = blockIdx.y*blockDim.y + threadIdx.y;

 // Calculate the column # of the d_Pin and d_Pout element to process
 int Col = blockIdx.x*blockDim.x + threadIdx.x;

 // each thread computes one element of d_Pout if in range
 if ((Row < m) && (Col < n)) {
  d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
 }
}
```

Assume that we decided to use a grid of 16X16 blocks. That is, each block is organized as a 2D 16X16 array of threads. Which of the following statements sets up the kernel configuration properly? Assume that int variable n has value 800 and int variable m has value 600. The kernel is launched with the statement PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, n, m);

(A) dim3 DimGrid(ceil(n/16.0), ceil(m/16.0), 1); dim3 DimBlock(16, 16, 1);
(B) dim3 DimGrid(1, ceil(n/16.0), ceil(m/16.0); dim3 DimBlock(1, 16, 16);
(C) dim3 dimGrid(ceil(m/16.0), ceil(n/16.0), 1); dim3 DimBlock(16,16,1);
(D) dim3 DimGrid(1, ceil(m/16.0), ceil(n/16.0); dim3 DimBlock(1, 16, 16);

2. In Question 1, how many warps will have control divergence?

(A) 37*16 + 50
(B) 38*16
(C) 50
(D) 0

3. If a CUDA device's SM (streaming multiprocessor) can take up to 1,536 threads and up to 8 thread blocks. Which of the following block configuration would result in the most number of threads in each SM?

(A) 64 threads per block
(B) 128 threads per block
(C) 512 threads per block
(D) 1024 threads per block

4. Assume the following simple matrix multiplication kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
      float Pvalue = 0;
      for (int k = 0; k < Width; ++k) {Pvalue += M[Row*Width+k] * N[k*Width+Col];}
      P[Row*Width+Col] = Pvalue;
  }
}
```

If we launch the kernel with a block size of 16X16 on a 1000X1000 matrix, how many warps will have control divergence?

(A) 1000
(B) 500
(C) 1008
(D) 508

1. Answer: (A)

Explanation: dim3 format is (x, y, z). Since n is the size of the picture in the x direction and m is the size of the y direction, we should use n to set up the x dimension and m to set up the y dimension.

2. Answer: (D)

Explanation: The size of the picture in the x dimension is a multiple of 16 so there is no block in the x direction that has any threads in the invalid range. The size of the picture in the y dimension is 37.5 times of 16. This means that the threads in the last block are divided into halves: 128 in the valid range and 128 in the invalid range. Since 128 is a multiple of 32, all warps will fall into either one or the other range. There is no control divergence.

3. Answer: (C)

Explanation: (A) and (B) are limited by the number of thread blocks that can be accommodated by each SM. (D) is not a divider of 1,536, leaving 1/3 of the thread space open. (C) results in 3 blocks and fully occupies the capacity of 1,536 threads in each SM.

4. Answer: (B)

Explanation: There will be 63 blocks in the horizontal direction. 8 threads in the x dimension in each row will be in the invalid range. Every two rows form a warp. Therefore, there are 1000/2 =500 warps that will straddle the valid and invalid ranges in the horizontal direction. As for the warps in the bottom blocks, there are 8 warps in the valid range and 8 warps in the invalid range. Threads in these warps are either totally in the valid range or invalid range.

2.3. Tiling

- Proper use of CUDA shared memory
- Explain the principles and scope of memory coalescing
- Derive the necessary array indexing for a tiled matrix multiplication
- Understand the use of barrier synchronization in a tiled algorithm
- Estimate the reduction of memory bandwidth usage
- Overhead due to halo cells in algorithms such as convolution
- Understanding how to extend square tiles to rectangular tiles

1. Assume that a kernel is launched with 1000 thread blocks each of which has 512 threads. If a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel?
   (A) 1
   (B) 1000
   (C) 512
   (D) 51200

2. For our tiled matrix-matrix multiplication kernel, if we use a 32X32 tile, what is the reduction of memory bandwidth usage for input matrices M and N?
   (A) 1/8 of the original usage
   (B) 1/16 of the original usage
   (C) 1/32 of the original usage
   (D) 1/64 of the original usage

3. For the tiled single-precision matrix multiplication kernel as shown in Lecture, assume that the tile size is 32X32 and the system has a DRAM burst size of 128 bytes. How many DRAM bursts will be delivered to the processor as a result of loading one M-matrix tile by a thread block?

(A) 16
(B) 32
(C) 64
(D) 128

4. Assume that A is a global memory float array that is properly aligned to a DRAM burst section boundary. Further assume that the number of threads in the x-dimension of each thread block is greater than or equal to the warp size. Which of the following accesses in a kernel will make the most effective use of the DRAM bandwidth? Assume that k and Width are integer variables that do not depend on threadIdx.x or threadIdx.y. The Width value can be assumed to be a multiple of the DRAM burst size.

(A) A[2*threadIdx.x]
(B) A[threadIdx.x*Width + k]
(C) A[threadIdx.x+Width*k]
(D) A[k*threadIdx.x]

5. Assume a DRAM system with a burst size of 512 bytes and a peak bandwidth of 240 GB/s. Assume a thread block size of 1024 and warp size of 32 and that A is a float array in the global memory. What is the maximal memory data access throughput we can hope to achieve in the following access to A?

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
float temp = A[8*i];
```

(A) 240 GB/s
(B) 120 GB/s
(C) 60 GB/s
(D) 30 GB/s

6. Assume a tiled matrix multiplication that handles boundary conditions as explained in Lecture. If we use 32X32 tiles to process square matrices of 1000X1000, within EACH thread block, what is the maximal number of warps that will have control divergence due to handling boundary conditions for loading M matrix tiles throughout the kernel execution?
(A) 32
(B) 24
(C) 16
(D) 8

7. For a tiled 2D convolution, if each output tile is a square with 256 elements on each side and the mask is a square with 5 elements on each side, how many elements are in each input tile?
   (A) 256*256 = 65,536
   (B) 5*5 = 25
   (C) (256+2) * (256+2) = 66,564
   (D) (256+4) * (256+4) = 67,600

8. For a tiled 2D convolution, assume that we load an entire input tile, including the halo elements into the shared memory when calculating an output tile. Further assume that the tiles are internal and thus do not involve any ghost elements. If each output tile is a square with 256 elements on each side and the mask is a square with 5 elements on each side, which of the following best approximate the average number of times each input element will be accessed from the shared memory during the calculation of an output tile?
   (A) 256
   (B) 25
   (C) 24
   (D) 4.9

9. For a tiled 3D convolution, assume that we load an entire input tile, including the halo elements into the shared memory when calculating an output tile. Further assume that the tiles are internal and thus do not involve any ghost elements. If the mask is a cube with 5 elements on each side and due to the limited size of the shared memory, each output tile is a cube with 16 elements on each side. What is the average number of times each input element will be accessed from the shared memory during the calculation an output tile?
   (A) 256
   (B) 64
   (C) 24
   (D) 4

10. For a tiled 3D convolution, assume that we load an entire input tile, including the halo elements into the shared memory when calculating an output tile. Further assume that the tiles are internal and thus do not involve any ghost elements. If the mask is a cube with 5 elements on each side what is the trend of the average number of times each input element will be accessed from the shared memory during the calculation an output tile as a function of the input tile width?
    (A) Increases with the width of the tile size with a limit of 25
    (B) Decreases with the width of tile size with a limit of 25
    (C) Increases with the width of the tile size with a limit of 125
    (D) Decreases with the width of the tile size with a limit of 125

1. Answer: (B)
   Explanation: Shared memory variables are allocated to thread blocks. So, the number of versions is the number of thread blocks, 1000.

2. Answer: (C)
   Explanation: Each element in the tile is used 32 times, as explained in Lecture

3. Answer: (B)
   Explanation. For an 32X32 M-tile, each row in the tile consists of 32 consecutive words and is accessed by a warp. The total amount of data in the row is just a single burst. We have 32 rows in a tile so there will be 32 bursts delivered to the processor.

4. Answer: (C)
   Explanation: All consecutive threads in A[threadIdx.x+Width*k] access consecutive memory locations. Since A is properly aligned to the DRAM burst sections and Width*k will be always a multiple of DRAM burst sizes, all DRAM burst bytes will be fully utilized. All other accesses are strided accesses that will waste at least some of the burst byres.

5. Answer: (D)
   Explanation: Each warp is going to access 64 bytes of data from a 512-byte section. It will waste at least 7/8 of the DRAM bandwidth. The access cannot achieve more 30 GB/s.

6. Answer: (A)
   Explanation: Control divergence happens due to the handling of right-side boundary. For thread blocks that process tiles that are totally within the valid range in the y-dimension, all 32 warps in a block will experience divergence at the right boundary. For the thread block that process the bottom M tiles, only 8 warps will experience control divergence because 24 warps will always fail the boundary test.

7. Answer: (D)
   Explanation: As shown in Lecture 7.3, Slide 12, the number of elements in an input tile is (tile_width+mask_width-1)*(tile_widt+mask_width-1), where tile-width is the width the output tiles.

8. Answer: (C)
   Explanation: As shown in Lecture 7.3, Slide, the answer is $tile\_width^2 * mask\_width^2 / (tile\_width+mask\_width-1)^2 = 256^2 * 5^2 / (256+5-1)^2 = 24.2$

9. Answer: (B)
   Explanation: As generalized from Lecture 7.3, the answer is $(tile\_width^3 * mask\_widt^3)/(tile\_width+mask\_width-1)^3 = 16^3 * 5^3/(16+5-1)^3 = 64$

10. Answer: (C)

The average number of times an input tile element is accessed from the shared memory is $(tile\_width^3 * mask\_widt^3)/(tile\_width+mask\_width-1)^3$. For a given mask_width, the value increases as the tile_width increases. When tile-width becomes much larger than tile_width, the mask_width term can be dropped from the denominator. This makes the expression $mask\_width^3$.

3. Topics to Review from Lab

The answers have not been fully verified. Please check with TAs and Prof. Hwu if you would like to challenge any of the answers.

Common sources of bugs

- Function prototype problems
- Barrier synchronization problems
- Indexing problems

Performance Issues

- Access patterns that result non-coalesced global memory accesses
- Sources of control divergence
- Thread utilization

Tiling

- Square vs. Rectangular tiling
- Handling boundary conditions in tiling
- Barrier synchronization usage

Convolution

- Different strategies for convolution implementations, their pros and cons, and how they reflect on kernel launch configurations

Reduction Tree

- Reduction trees, memory access patterns, and branch divergence

Lab Question 1. CUDA Basics. For the vector addition kernel and the corresponding kernel launch code, answer each of the sub-questions below.

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
   {

        1. int i = threadIdx.x + blockDim.x * blockIdx.x;
        2.  int Stride = blockDim.x * gridDim.x;
```

```
3.   while (i < n} {
4.     C_d[i] = A_d[i] + B_d[i];
5.     i+= Stride;
6.   }
7.  }

8. int vectAdd(float* A, float* B, float* C, int n)
9. {
//assume that size has been set to the actual length of
//arrays A, B, and C
10.  int size = n * sizeof(float);
11.
12.  cudaMalloc((void **) &A_d, size);
13.  cudaMalloc((void **) &B_d, size);
14.  cudaMalloc((void **) &C_d, size);
15.  cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
16.  cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
17. vecAddKernel<<<8,1024>>>(A_d, B_d, C_d, n);
18. cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
}
```

(A) Assume that the size of A, B, and C is 20,000 elements each. How many thread blocks will be generated?

Answer:

(B) Assume that the size of A, B, and C is 20,000 elements each. How many warps are there in each block?

Answer:

(C) Assume that the size of A, B, and C is 20,000 elements. How many threads will be created in the grid?

Answer:

(D) Assume that the size of A, B, and C is 20,000 elements each. Is there any control divergence during the execution of the kernel? If so, identify the block number and warp number that causes the control divergence. Explain why or why not.

Answer:

(E) Assume that the size of A, B, and C is 10,000 elements each. Is there any control divergence during the execution of the kernel? If so, identify the line number of the statement that causes the control divergence. Explain why or why not.

Answer:

Answers:

(A) 8

(B) 32

(C) 8,192

(D) No. We have 8,192 threads. All threads will iterate at least two iterations to process 16,384 elements. During the last iteration, only 20,000 – 16,384 = 3,616 threads will be active. Theses threads form 113 warps. So, all threads in all the first 113 warps will be active. The rest 143 warps will see all their threads inactive.

(E) We have 8,192 threads. All threads will iterate at least one iteration to process 8,192 elements. During the last iteration, only 10,000 – 8,192 = 1,808 threads will be active. Theses threads form 56.5 warps. So, all threads in all the first 56 warps will be active. Warp 24 on Block 1 will have control divergence. All remaining 199 warps will see all their threads inactive and thus see no control divergence.

Lab Question 2. Many numerical libraries offer matrix multiplication functions that accept one of the matrices in transposed form. Redo MP-2 simple matrix multiplication assuming that the first input matrix M is in transposed form (M_T).

(A) Write a new sgemm kernel for this transposed M input:

(B) Assume that the height and width values of the input matrices are in host variables Height_M_T, Width_M_T, Height_N, Width_N, write the host code that sets up the grid dimensions for launching a kernel with 32x32 threads per block.

(C) Explain the benefit of having a transposed M as input for a CUDA GPU.
Answer:

(D) What is the minimal block width in the x dimension for full memory bandwidth utilization?

Answer:

(E) Does it make sense for us to have a version of the tiled matrix multiplication kernel that accepts transposed M input matrix?

Answer:

Answers:

(A)

```
__global__ void sgemm(float* M_T, float* N, float* P, int HeiM_T, int WidM_T, int WidN)
{
// Height N is the same as Height M_T

// Calculate the row index of the P element, this will be used as the Col index for M_T
 int Row = blockIdx.y*blockDim.y+threadIdx.y;
```

```
// Calculate the column index of P element and N
 int Col = blockIdx.x*blockDim.x+threadIdx.x;

 if ((Row < WidM_T) && (Col < WidN)) {
        float Pvalue = 0;
// each thread computes one element of the block sub-matrix
        for (int k = 0; k < HeiM; ++k){
                Pvalue += M[k*WidM+Row] *  N[k*WidN+Col];
        }
        P[Row][Col] = Pvalue;
  }
}
```

(B)

```
dim3 DimGrid((WidM/32, WidN/32, 1);
if(WidM%32) DimGrid.x++;
if(WidN%32) DimGrid.y++;
dim3 DimBlock(32,32, 1);
```

(C) With the transposed input, the accesses to M_T are now coalesced. This should improve the speed significantly.

(D) The number of threads in the X dimension should be at least the DRAM burst size. So if the DRAM burst size is 128 bytes, the 32X32 block configuration would fully utilize the memory bandwidth.

(E) Read the textbook section that explains “corner turning”.

Lab Question 3

To improve the performance of her tiled matrix multiplication kernel, Jill decided to try to use rectangular tiling. Instead of using 32x32 input and output tiles, she would like to try to use 16x64 (16 in the Y dimension and 64 in the X dimension) input M tiles and 64x16 input N tiles.

(A) Based on the input tile dimensions, what is the maximal output tile dimensions that can be supported by these input tiles? Explain your answer.

Answer:

(B) The code below shows her code. Fill in the missing index calculations.

```
01      #define M_TILE_H 16                 // Height of input M tiles (Y dimension)
02      #define M_TILE_W 64                 // Width of input M tiles (X dimension)
03      #define N_TILE_H 64                 // Height of input N tiles (Y dimension)
04      #define X_N_TILE_WIDTH 16           // Weight of input N tiles (X dimension)
02
03  __global__ void sgemm(float* M, float* N, float* P, int HeiM, int WidM, int WidN) {
04
```

```
05    __shared__ float Mds[M_TILE_H][ M_TILE_W];
06    __shared__ float Nds[N_TILE_H][N_TILE_W];
07
08    int bx = blockIdx.x;  int by = blockIdx.y;
09    int tx = threadIdx.x; int ty = threadIdx.y;
10
11    // Identify the row and column of the P element to work on
12    int Row = by * _____ + ty;
13    int Col = bx * _____ + tx;
14
15    float Pvalue = 0;
16    // Loop over the M and N tiles required to compute P element
17    for (int m = 0; m < (WidM - 1)/ M_TILE_W + 1; ++m) {
18
19       // Collaborative load of M and N tiles into shared memory
20       if(Row < HeiM) {
21          Mds[ty][tx] = M[_____________________________;
22       } else {
23          Mds[ty][tx] = 0.0;
24       }
25       if(Col < WidN) {
26          Nds[ty][tx] = N[_____________________________];
27       } else {
28          Nds[ty][tx] = 0.0;
29       }
30       __syncthreads();
31
32       if (Row < HeiM && Col < WidN) {
33          for (int k = 0; k < M_TILE_W; ++k) {
34             Pvalue += _____________________________;
35          }
36       }
37       __syncthreads();
38    }
39    if (Row < HeiM && Col < WidN) P[Row*Width + Col] = Pvalue;
40 }
```

Answer:

(C) What is the expected number of times each input element is reused in the Shared Memory?

Answer:

(D) Compare this tiling configuration with what you had in MP3. What are the pros and cons of this rectangular configuration?

Answer:

Answers:

(A) If we draw a picture of the input and output tiles like the one we showed for square tiles in Lecture slides, the output tile has the same height as the input M tile and the same width as the input N tile. So the maximal output tile dimensions that can be supported by the input tiles are 16x16.

(B)

Line 12: M_TILE_H
Line 13: N_TILE_W
Line 21: Row * WidM + (m * M_TILE_W)+tx
Line 22: (m*N_TILE_H+ty) * WidN + Col
Line 34: Mds[ty][k] * Nds[k][tx]

Please try to run the code on WebGPU as another attempt for MP3.

(C) 16, each M tile row is used to calculate 16 P elements in the same row. Each N tile column is used to calculate 16 P elements in the same column.

(D)
Cons:

1. It results in less memory reuse but requires the same amount of shared memory for each block.
2. If the DRAM burst size is 32 words, the new method will use only half of the memory bandwidth when loading input tiles and writing output elements.

Pros:

1. It allows each thread to execute twice the number of inner-product iterations (64 vs. 32) between syncthreads
2. The thread block size is 256. We may be able to fit more thread blocks into a streaming multiprocessor. Since these thread blocks do syncthreads independently, we may have a better utilization of the execution resources since there may be more thread blocks that are not executing syncthreads at any moment in time.

Lab Question 4. To further improve the memory access efficiency and reduce the memory bandwidth consumption of his 3D convolution kernel, John decided to use rectangular prism tiles rather than cube tiles. Assume that the convolution mask is 3x3x3. His current cube tile design is 6x6x6 for output tiles and 8x8x8 for input tiles (Strategy 2). He decided to try to change the output tiles 14x6x6 and input tiles to input tiles to 16x8x8.

(A) For an internal 16x8x8 input tile, what is the average number of times an input element is accessed from the shared memory? Show your work.

Answer:

(B) Does the number of threads in each block fit within the limits of CUDA block size? Why or why not?

Answer:

(C) How many thread blocks will be generated if we process a 512x768x256 volume of data with the new kernel? Show your work.

Answer:

(D) For an internal 8x8x8 input tile in the original cube tile design, what is the average number of times each input element is reused once it is loaded into the shared memory? Show your work.

Answer:

(E) How many thread blocks will be generated if we process a 512x768x256 volume of data with the original cube tiling kernel? Show your work.

Answer:

(F) Compare the pros and cons of the new rectangular kernel vs. the cube kernel. Assume that the DRAM burst size is 128 bytes.

Answer:

(G) Fill in the index calculations and missing code of the following rectangular prism kernel.

```
#define TILE_X 14          // Output tile width in the X-dimension
#define TILE_Y  6        // Output tile width in the Y-dimension
#define TILE_Z  6          // Output tile width in the Z-dimension
#define MASK_WIDTH 3
#define MASK_SIZE  MASK_WIDTH * MASK_WIDTH * MASK_WIDTH


__constant__ float mask[MASK_WIDTH][MASK_WIDTH][MASK_WIDTH];

__global__ void conv3d(float *input, float *output, const int z_size, const int y_size, const int x_size) {
    __shared__ float inputTile [TILE_Z+MASK_WIDTH-1][TILE_Y+MASK_WIDTH-1][TILE_X+MASK_WIDTH-1];
    int tx = threadIdx.x;  int ty = threadIdx.y;  int tz = threadIdx.z;
    int bx = blockIdx.x;  int by = blockIdx.y;  int bz = blockIdx.z;
```

```
    int x_o = bx * __________ + tx
    int y_o = by * __________ + ty;
    int z_o = bz * __________ + tz;

    int x_i = _____________________;
    int y_i = _____________________;
    int z_i = _____________________;

    if (x_i >= 0 && y_i >= 0 && z_i >= 0 && x_i < x_size && y_i < y_size && z_i < z_size)
        inputTile[tz][ty][tx] = input[_____________________________________________];
    else
        inputTile[tz][ty][tx] = 0.0;

    ___________________________________;

    float acc = 0.0;

    if(tz < TILE_Z && ty < TILE_Y && tx < TILE_X) {

        for(int z_mask = 0; z_mask < Z_MASK_WIDTH; z_mask++) {

            for(int y_mask = 0; y_mask < Y_MASK_WIDTH; y_mask++) {

                for(int x_mask = 0; x_mask < X_MASK_WIDTH; x_mask++) {

                    acc += mask[_____][_________][_____] * inputTile[_______][_________][________];
                }

            }

        }

        if(z_o < z_size && y_o < y_size && x_o < x_size)
            output[(z_o * y_size + y_o) * x_size + x_o] = acc;

    }
}
```

Answers:

(A) (14*6*6) ((3*3*3) / (16*8*8) = 13.3

(B) The number of threads is the same as the block size so the block size is 16*8*8 = 1024. It fits into the limit of the CUDA block size, which is 1024.

(C) ceil(512/14.0) * ceil(768/6.0) * ceil(256/6.0) = 203,648

(D) (6*6*6) * (3*3*3) / (8*8*8) = 11.39

(E) ceil(512/6.0) * ceil(768/6.0) * ceil(256/6.0) = 86 * 128 * 43 = 473,344

(F)

Pros:

1. The input elements are reused more in the shared memory than the original tile configuration due to larger tile size.
2. The memory bandwidth is better utilized since the input tile width in the X dimension is the same as the DRAM burst size. The original tile configuration uses more than double the memory bandwidth since it wastes half of the memory bandwidth.

Cons:

1. The new kernel generates fewer, larger thread blocks. This may increase the negative impact of the barrier synchronizations.

(G)

```
#define TILE_X 14          // Output tile width in the X-dimension
#define TILE_Y   6        // Output tile width in the Y-dimension
#define TILE_Z   6          // Output tile width in the Z-dimension
#define MASK_WIDTH 3
#define MASK_SIZE  MASK_WIDTH * MASK_WIDTH * MASK_WIDTH


__constant__ float mask[MASK_WIDTH][MASK_WIDTH][MASK_WIDTH];

__global__ void conv3d(float *input, float *output, const int z_size, const int y_size, const int x_size) {
    __shared__ float inputTile [TILE_Z+MASK_WIDTH-1][TILE_Y+MASK_WIDTH-1][TILE_X+MASK_WIDTH-1];
    int tx = threadIdx.x;  int ty = threadIdx.y;  int tz = threadIdx.z;
    int bx = blockIdx.x;  int by = blockIdx.y;  int bz = blockIdx.z;

    int x_o = bx * TILE_X + tx
    int y_o = by * TILE_Y + ty;
    int z_o = bz * TILE_Z + tz;

    int x_i = x_o - MASK_WIDTH/2;
    int y_i = y_o - MASK_WIDTH/2;
    int z_i = z_o - MASK_WIDTH/2;

    if (x_i >= 0 && y_i >= 0 && z_i >= 0 && x_i < x_size && y_i < y_size && z_i < z_size)
        inputTile[tz][ty][tx] = input[(z_i * y_size + y_i) * x_size + x_i];
    else
        inputTile[tz][ty][tx] = 0.0;

    __syncthreads();

    float acc = 0.0;

    if(tz < TILE_Z && ty < TILE_Y && tx < TILE_X) {

        for(int z_mask = 0; z_mask < Z_MASK_WIDTH; z_mask++) {

            for(int y_mask = 0; y_mask < Y_MASK_WIDTH; y_mask++) {

                for(int x_mask = 0; x_mask < X_MASK_WIDTH; x_mask++) {

                    acc += mask[z_mask][y_mask][x_mask] * inputTile[tz+z_mask][ty+y_mask][tx+x_mask];
```

```
                }

            }

        }

        if(z_o < z_size && y_o < y_size && x_o < x_size)
            output[(z_o * y_size + y_o) * x_size + x_o] = acc;

    }
}
```

Lab Question 5. Out of curiosity, Emily decided to try Strategy 3 in her 2D kernel. She decided to use square 32x32 input and output kernel. Assume that the mask is 3x3.

(A) For an input tile, what is the average number of times each input element is reused once it is loaded into the shared memory. (Hint: draw a picture any check the number of cases that you need to analyze.)

Answer:

(B) How many thread blocks will be generated if we process 512x768 input data?

Answer:

(C) Fill in the missing index calculations and boundary condition checks in the following Strategy 3 kernel.

Answer:

```
#define MASK_WIDTH 3
#define TILE_WIDTH 32
__constant__ float mask[MASK_WIDTH][MASK_WIDTH][MASK_WIDTH];


__global__ void conv2d(float *input, float *output, const int y_size, const int x_size) {

    __shared__ float inputTile [TILE_WIDTH][TILE_WIDTH-1][TILE_WIDTH-1];
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int row = blockIdx.y * TILE_WIDTH + ty;
    int col = blockIdx.x * TILE_WIDTH + tx;
    int radius = MASK_WIDTH/2;
```

```
    float output = 0.0f;

     if ((row < y_size) && (col < x_size) ) {
       inputTile[ty][tx] = input[_____________________];
     }
      else{
       inputTile[ty][tx] = 0.0f;
     }

    __syncthreads();

    If (row < size_y && col < size_x} {

    int This_tile_start_point_x = blockIdx.x * blockDim.x;
    int Next_tile_start_point_x = (blockIdx.x + 1) * blockDim.x;

    int This_tile_start_point_y = blockIdx.y * blockDim.y;
    int Next_tile_start_point_y = (blockIdx.y + 1) * blockDim.y;

    int input_start_point_x = _____________________;
    int input_start_point_y = _____________________;

    float Pvalue = 0;
     for (int i = 0; i < MASK_WIDTH; i ++) {
        int input_index_y = N_start_point_y + i;
        for (int j = 0; j < MASK_WIDTH; j++) {
           int input_index_x = N_start_point_x + j;
           if (input_index_y >= This_tile_start_point_x
                   && input_index_y < Next_tile_starting_point_y
                   && input_index_x >= This_tile_start_point_x
                   && input_index_y < Next_tile_start_point_y) {
              Pvalue += inputTile[________][__________]*mask[___][___];
         } else {
            Pvalue += input[_____________________] * mask[__][____];
         }
        }
     }
     P[i] = Pvalue;
}
```

(D) Compare the pros and cons of the new Strategy 3 kernel vs. a Strategy 2 kernel.

Answer:

Answers:

(A) All edge elements of the output square except for the corner ones will have to access three of their input elements from the global memory. The corner output elements will have to access five of their input elements from the global memory. Thus the total number of global memory accesses during the convolution calculation will be

3*(TILE_WIDTH-2) *4 + 5 * 4 = 3 * 30 * 4 – 20 = 340

The total number of memory accesses served by the shared memory is thus

32*32 * 3* 3 – 340 = 9216 - 340 = 8,876

The average number of reuses for each input element loaded into the shared memory is thus

8,876/(32*32) = 8.67

Another way to look at it is that total number of global memory accesses made for calculating the 32x32 output elements is

1024 + 340 = 1,364

So the average number of global accesses for input elements per output element calculation is

1,364 / 1,024 = 1.33

It is interesting to compare this with Strategy 2, where the average number of global memory accesses for input elements per output element calculation is

1,024 / ((32-2) * (32-2)) = 1,024 /900 = 1.14

So Strategy 2 is slightly better than strategy 3. (However, if some of the accesses to halo elements are serviced by the L2 cache, this benefit diminishes quite quickly.)

(B) ceil(512/32.0) * ceil(768/32.0) = 16 * 24 = 384

(c)

```
#define MASK_WIDTH 3
```

```
#define TILE_WIDTH 32
__constant__ float mask[MASK_WIDTH][MASK_WIDTH][MASK_WIDTH];


__global__ void conv2d(float *input, float *output, const int y_size, const int x_size) {

    __shared__ float inputTile [TILE_WIDTH][TILE_WIDTH-1][TILE_WIDTH-1];
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int row = blockIdx.y * TILE_WIDTH + ty;
    int col = blockIdx.x * TILE_WIDTH + tx;
    int radius = MASK_WIDTH/2;

    float output = 0.0f;

     if ((row < y_size) && (col < x_size) ) {
       inputTile[ty][tx] = input[row*x_size + col];
     }
      else{
       inputTile[ty][tx] = 0.0f;
     }

    __syncthreads();

    If (row < size_y && col < size_x} {

    int This_tile_start_point_x = blockIdx.x * blockDim.x;
    int Next_tile_start_point_x = (blockIdx.x + 1) * blockDim.x;

    int This_tile_start_point_y = blockIdx.y * blockDim.y;
    int Next_tile_start_point_y = (blockIdx.y + 1) * blockDim.y;

    int input_start_point_x = col - radius;
    int input_start_point_y = row – radius;

    float Pvalue = 0;
     for (int i = 0; i < MASK_WIDTH; i ++) {
        int input_index_y = N_start_point_y + i;
        for (int j = 0; j < MASK_WIDTH; j++) {
           int input_index_x = N_start_point_x + j;
           if (input_index_y >= This_tile_start_point_x
                   && input_index_y < Next_tile_starting_point_y
                   && input_index_x >= This_tile_start_point_x
                   && input_index_y < Next_tile_start_point_y) {
```

```
                    Pvalue += inputTile[ty-radius+i][tx-radius+j]*mask[i][j];
                } else {
                    Pvalue += input[input_index_y*x_size + input_index_x] * mask[i][j];
                }
              }
            }
            P[i] = Pvalue;
        }
    }
```

(D)

Pros:

1. Simple input tile loading code
2. Better thread utilization
3. Simple output element writing code

Cons:

1. More complex calculation code
2. More global memory accesses