ECE 408 Exam #2 Study Guide
Spring 2019 ZJUI Section

This guide was taken from 2017. Our coverage is slightly different. For example, we had already covered neural networks before exam 1, so we not focus on them on this exam. But we had not done scans nor reductions, so we will include those topics for exam 2. Look back at the exam 1 materials for appropriate questions on missing topics.

1. Exam format

- You are allowed one A4 cheat sheet with HANDWRITTEN notes on both sides.
- No interactions with humans other than course staff are allowed.
- This exam is designed to take 90 minutes to complete. To allow for any unforeseen difficulties, we will give everyone 180 minutes. Please note that ***SAMPLE EXAMS WERE DESIGNED FOR 150 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

CUDA Basic concepts
- Mapping thread index into data index
- Using multi-dimensional thread indices to easily access multi-dimensional data structures
- Memory hierarchies – characteristics and usage of each type of memory

Memory Bandwidth and Coalescing
- Explain why DRAM bursts are so long today and is getting longer in the future
- Explain the principles and scope of memory coalescing
- How one can use shared memory to increase coalescing via “corner turning”

Tiling
- Derive the necessary array indexing for a tiled matrix multiplication
- Understand the use of barrier synchronization in a tiled algorithm using shared memory

- Overhead due to halo cells in algorithms such as convolution

Control Divergence
- Data structure padding techniques to reduce control divergence
- Loops and if-statements that cause control divergence

Barrier Synchronization
- Use of barrier synchronization in reduction, prefix sum and other global stepping algorithms
- Proper use of barriers in control constructs

Floating-Point Considerations (Textbook Reading)
- Given a number of bits and format, tell the representable numbers
- Explain why floating-point arithmetic is not necessarily associative and can generate different numerical values when evaluation order changes

Reduction and Prefix Sum Patterns
- Work efficiency of parallel prefix-sum algorithms
- Thread block prefix-sum algorithm using shared memory
- Global combination of partial prefix sums

1. For the Kogge-Stone scan kernel based on reduction trees, assume that we have 1024 elements, which of the following gives the closest approximation for the number of useful floating-point add operations performed?
   (A) (1024-1)*2
   (B) (512-1)*2
   (C) 1024*1024
   (D) 1024*10

2. For the Brent-Kung scan kernel based on reduction trees and inverse reduction trees, assume that we have 1024 elements, which of the following gives the closest approximation on the total number of useful floating-point add operations performed in both the reduction tree phase and the inverse reduction tree phase?
   (A) (1024-1)*2
   (B) (512-1)*2
   (C) 1024*1024
   (D) 1024*10

3. 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

4. For the Kogge-Stone scan kernel based on reduction trees, assume that we have 1024 elements in each section and warp size is 32, how many warps in each block will have control divergence during the iteration where stride is 16?
   (A) 0
   (B) 1
   (C) 16
   (D) 32

5. In the previous question, how many warps in each block will have control divergence during the iteration where stride is 64?
   (A) 0
   (B) 1
   (C) 16
   (D) 32

Answers and Explanations:

1. Answer: (D)
   Explanation: The number of useful add operations performed by the Kogge-Stone scan kernel is approximately N*log(N), where N is the number of elements.

2. Answer: (A)
   Explanation: The reduction tree performs N-1 floating-point add operations performed in the reduction tree phase and the inverse reduction performs N-1-log(N) floating-point add operations. So the total is 2*(N-1) – log(N). When N is large, this is approximately 2*(N-1).

3. Answer: (A)
   Explanation: All active threads are consecutive starting with index 0. There are a total of 1024 threads and 64 active threads in the iteration in each block. The active threads form two whole warps. The warps are either all active or all inactive. None will have control divergence.

4. Answer: (B)
   Explanation: All inactive threads are consecutive at the end of the block. There are a total of 1024 threads. When stride is 16, there are 16 inactive threads at the beginning of the block. So, Warp 0 has control divergence. No other warp in the block will have control divergence.

5. Answer: (A)

Explanation: At stride value 64, there are 64 inactive threads that are at the front of the block. Therefore, all threads in the first two warps are inactive. All threads in the remaining warps are active. There is no control divergence.

Histogram and Atomic Operations

- The reason why parallel histogram algorithms need atomic operations
- The use of privatization to increase efficiency

1. Assume that each atomic operation in a DRAM system has a total read-modify-write latency of 100ns. What is the maximal throughput we can get for atomic operations on the same global memory variable?
   (A) 100G atomic operations per second
   (B) 1G atomic operations per second
   (C) 0.01G atomic operations per second
   (D) 0.0001G atomic operations per second

2. For a processor that supports atomic operations in L2 cache, assume that each atomic operation takes 4ns to complete in L2 cache and 100ns to complete in DRAM. Assume that 90% of the atomic operations hit in L2 cache. What is the approximate throughput for atomic operations on the same global memory variable?
   (A) 0.25G atomic operations per second
   (B) 2.5G atomic operations per second
   (C) 0.0735G atomic operations per second
   (D) 100G atomic operations per second

3. In question 1, if a kernel performs 5 floating-point operations per atomic operation, what is the maximal floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?
   (A) 500 GFLOPS
   (B) 5 GFLOPS
   (C) 0.05 GFLOPS
   (D) 0.0005 GFLOPS

4. In Question 2, if a kernel performs 5 floating-point operations per atomic operation, what is the maximal floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?
   (A) 1.25 GFLOPS
   (B) 12.5 GFLOPS
   (C) 0.368 GFLOPS
   (D) 500 GFLOPS

5. In Question 3, after Gather-to-Scatter transformation, there is no longer atomic operation in the kernel. Assume that the kernel performs 2 floating-point operations for every global memory access. Also, assume that kernel performs single-precision floating-point arithmetic and the global memory bandwidth is 160GB/second. What is the approximate floating-point throughput of the kernel execution as limited by the memory bandwidth?
   (A) 1000 GFLOPS
   (B) 100 GFLOPS
   (C) 80 GFLOPS
   (D) 2 GFLOPS

Answers and Explanations

1. Answer: (C)
   Explanation: No other atomic operation can touch the same variable for the entire duration of 100ns. The maximal rate is 1/100n = 0.01G

2. Answer: (C)
   Explanation: The average latency is 4ns * 90% + 100ns *10% = 13.6ns. The average throughput is approximately 1/13.6 = 0.0735G atomic operations per second

3. Answer: (C)
   Explanation: The maximal is 5 operations per every 100ns, or 5*0.01G

4. Answer: (C)
   Explanation: 5 floating-point operations every 13.6 ns, approximately 5/(13.6ns) = 0.368 GFLOPS

5. Answer: (C)
   Explanation: 40G operand fetches per second which supports 80 GFLOPS since each fetched operand is used 2 times.

Sparse Matrix-Vector Multiplication

- Cost and benefit of each format – CSR, ELL, COO, JDS, JDS-Transpose
- No data reuse for matrix elements

1. Given a sparse matrix of integers with m rows, n columns, and k non-zeros. How many integers are needed to represent the matrix in COO?
   (A) m+n+k
   (B) 3m
   (C) 3n
   (D) 3k

2. Given a sparse matrix of integers with m rows, n columns, and k non-zeros. How many integers are needed to represent the matrix in CSR?
   (A) m+n+k

(B) 2k +m+1
(C) 2k+n
(D) 3k

3. Given a sparse matrix of integers with m rows, n columns, and k non-zeros. How many integers are needed to represent the matrix in ELL?
   (A) m+n+k
   (B) 2k+m+1
   (C) 2k+n
   (D) None of the above

4. Given a sparse matrix of integers with m rows, n non-zero elements in the row with the largest number of non-zeros, and k non-zeros. How many integers are needed to represent the matrix in JDS-T? Recall that JDS-T has a transposed representation. Also, assume that we keep track of the number of non-zero's in each row, as we specified in the MP assignment.
   (A) m+n+k
   (B) 2k+m+1
   (C) 2k+2m+n
   (D) 2m+1

5. Assume that a GPU has a global memory bandwidth of 160 GB/s. For a single-precision JDS-T kernel, what is the approximate floating-point throughput of the kernel execution as limited by memory bandwidth?
   (A) 4000 GFLOPS
   (B) 400 GFLOPS
   (C) 40 GFLOPS
   (D) 4 GFLOPS

Answers and Explanations:

1. Answer: (D)
Explanation: Each non-zero element needs three integers: value, row index, column index

2. Answer: (B)
Explanation: Each non-zero element needs two integers: value and column. Each row needs an integer: start but we also need the one more at the end

3. Answer: (D)
Explanation: We need to know the row with the largest number of non-zero elements!

4. Answer: (C)
Explanation: We need k integers for the non-zero elements, k integers for their column indices, n pointers to the beginning of each column after transposition, m integers to track the length of each row, and m integers to track the original row index before permutation

5. Answer: (C)
   Explanation: 160 GB/s supports 40 GB/s single-precision operands per second. Each pair of floating-point multiplication and addition in the inner product requires one matrix element and one vector element. There is no reuse for the matrix elements. In our MP, we do not have anyway to exploit the reuse of the vector elements. So the memory bandwidth limits the floating point execution throughput to approximately 40 GFLOPS.

PC System Architecture

- Calculation of PCIe Gen 2 and Gen 3 bandwidth given an X configuration
- Understand the nature, use, and benefit of pinned (page-locked) memory
- Understand the DMA used in CPU-GPU data transfers

1. If Carl's PC has a PCIe Gen3 x16 interconnect for his GPU, what is the closest approximation of the maximal cudaMemcpyAsync() copy throughput from host to GPU that he can expect?
   (A) 100 GB/sec
   (B) 500 GB/sec
   (C) 16 GB/sec
   (D) 1 GB/sec

2. After ran a few test of cudaMemcpy(), Carl realized that the achieved copy throughput was about half of the PCIe bandwidth, what do you think was most likely the main cause of this degradation?
   (A) cudaMemcpy() has a lot of software overhead
   (B) cudaMemcpy() requires an extra copy from the user space to pinged buffer and the extra copy nearly doubles the total copying time.
   (C) The manufacturer lied. The PCIe in the system is actually Gen2.
   (D) The execution time of cudaMemcpy() was measured incorrectly.

3. Peter has a 200MB 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 was 0.0005, a 40x speedup! However, he needs to transfer the data into the GPU memory and transfer 200MB data output data back to the host memory. His system has a PCIe Gen3 x16 interconnect. What would the real speedup be?
   (A) 40x speedup
   (B) 20x speedup
   (C) 8x slow down
   (D) 1.275x slow down

Answers and Explanations:

1. Answer: (C)
   Explanation: cudaMemcpyAsync() operates on pinged memory so there is no need to made an extra copy. The throughput should be close to 16GB/sec.
2. Answer: (B)

Explanation: The most likely reason is that the data needs to be first copied to a pinged memory buffer.

3. Answer: (C)
   Explanation: The data copy will take (200M)/(16*1000M) = 0.0125 sec to copy the input data in and another 0.0125 sec to copy the output data back. Thus the total speedup is 0.02/(0.0005+0.0125+0.0125) = 0.78x speedup, or a 1.275x slow down

CUDA Streams and Task Parallelism

- Use of CUDA streams – creation, and insertion into queues
- Correspondence between stream queues and engine queues
- Loop unrolling and call ordering to overlap computation with data transfer

1. For the following code:

   1) cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),.., stream0);
   2) cudaMemcpyAsync(d_B0, h_B+I, SegSize*sizeof(float),.., stream0);
   3) cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),.., stream0);
   4) cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),.., stream1);
   5) cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float),.., stream1);
   6) cudaMemcpyAsync(h_C+i+SegSize, d_C1, SegSize*sizeof(float),.., stream1);

   Which of the statements could be executed in parallel on the GPU
   (A) 1), 2) and 3)
   (B) 1), 2) and 4)
   (C) 1) and 4)
   (D) 3), and 4)

Answers and Explanations:

1. Answer: (D)
   Explanation: API operations in different streams can be executed in parallel. However, there is only one PCIe copy engine in each direction. So, only 3) and 4) can go in parallel

3. Topics to Review from Lab

Common sources of bugs

- Function prototype problems
- Barrier synchronization problems
- Indexing problems

Performance Issues

- Access patterns that result non-coalesced global memory accesses / shared memory bank conflicts

Convolution

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

Reduction and Prefix Sum

- Reduction trees, memory access patterns, thread utilization, and branch divergence
- Kogge-Stone vs. Brent-Kung thread organization and element indexing
- Parallel execution overhead and tradeoff between parallel execution and sequential execution

Histogram and Privatization

- Levels of privatization and their applicability
- Allocation and indexing of Shared Memory
- Barrier synchronization and final contribution to the global histogram

Question: Privatization

This question tests your understanding of parallel histogram computation and privatization. Assume that we would like to privatize a histogram that has 2048 bins. Each input data value (buffer array elements) will range from 0 to 2047. However, the shared memory can only accommodate 1024 bins for each block. As a compromise, we decide to privatize the first half of the bins into the shared memory. Whenever the data value falls into a bin in the second half, we will have to increment the global bin.

(A) Complete the following kernel to implement the partial privatization of the histogram.

```
__global__ void histo_kernel(unsigned char *buffer,    long size, unsigned int *histo)
{
    __shared__ unsigned int histo_private[1024];
    int i;
    for (i =    ____________________;_i <____________;__i+=_______________) histo_privat[i] = 0;
     __syncthreads();
    int i = threadIdx.x + blockIdx.x * blockDim.x;
          // stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
          if   ( _____________________________) atomicAdd( &(private_histo[buffer[i]), 1);
              else atomicAdd( ____________________________________________________,1 );
          i += stride;
     }
  __syncthreads();
  for ( i= _____________________________;   i < _____________________________; i +=    ___________________     )
          atomicAdd( ____________________________________________ );
}
```

```
__global__ void histo_kernel(unsigned char *buffer,  long size, unsigned int *histo)
{
  __shared__ unsigned int histo_private[1024];
  int i;
  for ( i =  threadIdx.x;   i < 1024;   i+=blockDim.x ) histo_private[i] = 0;
   __syncthreads();
  i = threadIdx.x + blockIdx.x * blockDim.x;
      // stride is total number of threads
  int stride = blockDim.x * gridDim.x;
  while (i < size) {
        if  ( buffer[i] < 1024 ) atomicAdd( &(private_histo[buffer[i]), 1);
        else atomicAdd( &histo[buffer[i]]   ,1 );
        i += stride;
   }
  __syncthreads();
```

(B) Can you think of a better partial privatization strategy that will likely result in less contention in the while loop? Outline your strategy.

Each thread can start its privatized section at a position

Sparse Matrix

- Index calculation for the various formats.

- Control divergence and memory coalescing.

Sparse Matrix Multiplication in JDS_T

This question tests your knowledge of Sparse Matrix representation and operation.

(A) In the following JDS_T kernel, fill in the missing indexing expressions for accessing data (input matrix), x (input vector) and y (output vector).

```
1. __global__ void SpMV_JDS_T(int num_rows, float *data,    int *col_index, int *jds_t_col_ptr,
                              int jds_row_index, 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[____________________] * x[_____________________________________];
7.            sec++;
          }
8.       Y[_________________] = dot;
      }
   }
```

```
1. __global__ void SpMV_JDS_T(int num_rows, float *data,  int *col_index, int *jds_t_col_ptr,
                              int jds_row_index, 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_index[row]] = dot;
      }
   }
```

(B) Assume a matrix that has 32 original rows, 64 columns, and 10 non-zeros in every row. After we transform the matrix into JDS-transposed layout, and launch the SpMV_JDS_T kernel. Is there any control divergence? Why or why not?

There is no control divergence. All 32 threads will take 10 iterations.

(C) In (B), are the memory accesses to the matrix in the for-loop (line 6) coalesced? Why or why not?

The memory accesses to the matrix are coalesced. All elements in the same column of the original matrix are laid out consecutively.

Final Project

- Basic convolution layer kernel
- Tiled convolution layer kernel
- Kernel for unrolling X (input feature maps) used in the matrix-matrix multiplication implementation of convolution layer
- Properties of unrolled matrix and the parallelism, data reuse in the corresponding grid executing the matrix multiplication

Convolution Neural Network

This question tests your understanding of the convolution layer of a CNN. We will start with a basic kernel implementation.

W is the convolution filter weight tensor, organized a tensor W[M, C, K, K], 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.

X is the input feature map, organized as a tensor X[C, H_out+K-1, W_out+K-1], where H_out is the height of each output feature map, W_out is the width of each output feature map.

Y is the output feature map, organized as a tensor Y[M, H_out, W_out].

(A) Fill in the missing parts of the basic kernel implementation. Use multidimensional indexing notation for simplicity. The kernel has each thread block to calculate a tile of output feature map elements. Each thread generates one output map element.

Assume that the blockDim is set to (TILE_WIDTH, TILE_WIDTH,1) and that gridDim is set to (M, H_grid*W_grid, 1).

```
__global__ void ConvLayerForward_Basic_Kernel(int C, int W_grid, int K,
   float* X, float* W, float* Y)
{
        int m = blockIdx.x;
        int h =    blockIdx.y / W_grid    + threadIdx.y;
        int w = blockIdx.y % W_grid + threadIdx.x;
        float acc = 0.;
        for (int c = 0;    c < C; c++) {    // sum over all input channels
           for (int p = 0; p < K; p++)    // loop over KxK    filter
              for (int q = 0; q < K; q++)
                    acc += X[_____,_______,_______] * W[_____,_____,_______,_______];
        }
        Y[___,_____,_____] = acc;
}
```

Answer: X[c, h+p, w+q], W[m, c, p, q], Y[m, h, q]

(B) Define the meaning of variables h, and w in ConLayerForward_Basic_Kernel.

h: _____

w: _____

Answer:

h is the row index of the output feature map generated by each thread
w is the column index of the output generated by each thread

(C) Fill in the missing parts of the tiled kernel implementation. Use multidimensional indexing notation for simplicity. The kernel has each thread block to calculate a tile of output feature map elements. Each thread generates one output map element.

```
__global__ void
ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W, float* Y)
{
   int m, h0, w0, h_base, w_base, h, w;
   int X_tile_width = TILE_WIDTH + K-1;
   extern __shared__ float shmem[];
   float* X_shared = &shmem[0];
   float* W_shared = &shmem[X_tile_width * X_tile_width];
```

```
  m = blockIdx.x;
  h_base = (blockIdx.z / W_grid) * TILE_SIZE;   // vertical base out data index for the block
  w_base = (blockIdx.z % W_grid) * TILE_SIZE; // horizontal base out data index for the block

  tx = threadIdx.x;
  ty = threadIdx.y;
  h = h_base + tx;
  w = w_base + ty;

  float acc = 0.;
  int c, j, k, p, q;
  for (c = 0; c < C; c++) {                    // sum over all input channels
                                               // load weights for W [m, c,..],
// tx and ty used as shorthand for threadIdx.x and threadIdx.y
      if (( ty < K) && ( tx < K))
       W_shared[___,___]= W [__,__,__,__];
       _____// load tile from X[n, c,…] into shared memory

      for (int i = h; i < h_base + X_tile_width; i += TILE_WIDTH) {
       for (int j = w; j < w_base + X_tile_width; j += TILE_WIDTH)
        X_shared[__________,___________] = X[__, __, __, __]
      }

     for (p = 0; p < K; p++) {
      for (q = 0; q < K; q++)
       acc = acc + X_shared[____, _____] * W_shared[__,__];
     }

  }
  Y[__,__,__,__] = acc;
}
```

Answer:

```
// tx and ty used as shorthand for threadIdx.x and threadIdx.y
      if (( ty < K) && ( tx < K))
       W_shared[ty, tx]= W [m, c, ty, tx];
      __syncthreads();
                                               // load tile from X[n, c,…] into shared memory

      for (int i = h; i < h_base + X_tile_width; i += TILE_WIDTH) {
       for (int j = w; j < w_base + X_tile_width; j += TILE_WIDTH)
        X_shared[i - h_base, j - w_base] = X[n, c, i, j]
```

```
    }
    __syncthreads();

    for (p = 0; p < K; p++) {
      for (q = 0; q < K; q++)
        acc = acc + X_shared[h + p, w + q] * W_shared[p, q];
    }
    __syncthreads();
  }
  Y[n, m, h, w] = acc;
```

(D) For a 72x64 input feature map, 8x8 tiles and 3x3 convolution filters, if we use the tiled 2D convolution, what this the average number of times that each input feature map element is reused once it is loaded into the shared memory?

Answer:

$$\text{TILE\_WIDTH}^2 * K^2 / \text{X\_TILE\_WIDTH}^2 = 8^2 * 3^2 / (8+3-1)^2 = 5.76$$

(E) If we use each thread block to generate one tile of output feature map elements, how many thread blocks will be generated when we launch the kernel?

Answer:

$$(72/8) * (64/8) = 72 \text{ thread blocks}$$

(F) Fill in the missing parts of the unroll kernel implementation. Use multidimensional indexing notation for simplicity. The kernel has each thread block to calculate a tile of output feature map elements. Each thread generates one output map element.

```
void unroll_host_code(int C, int H, int W, int K, float* X, float* X_unroll)
{
  int H_out = H – K + 1;
  int W_out = W – K + 1;
  int num_threads = C * H_out * W_out;
  int num_blocks = ceil((C * H_out * W_out) / CUDA MAX_NUM_THREADS);
  unroll_Kernel<<<num_blocks, CUDA MAX_NUM_THREADS>>>();
}

__global__ void unroll_Kernel(int C, int H, int W, int K, float* X, float* X_unroll)
{
  int c, s, h_out, w-out, h_unroll, w_base, p, q;
```

```
    int t = blockId.x * CUDA MAX_NUM_THREADS + threadId.x;
    int H_out = H – K + 1;
    int W_out = W – K + 1;
    int W_unroll = H_out * W_out;

    if (t < C * W_unroll) {
      c = t / W_unroll;
      s = t % W_unroll;
      h_out = s / W_out;
      w_out = s % W_out;
      h_unroll = _____;
      w_base = c * K * K;
      for(p = 0; p < K; p++)
        for(q = 0; q < K; q++) {
          w_unroll = _____;
          X_unroll(_____, _____) = X(____, _____, _____);
        }
    }
}

    if (t < C * W_unroll) {
      c = t / W_unroll;
      s = t % W_unroll;
      h_out = s / W_out;
      w_out = s % W_out;
      h_unroll = h_out * W_out + w_out;
      w_base = c * K * K;
      for(p = 0; p < K; p++)
        for(q = 0; q < K; q++) {
          w_unroll = w_base + p * K + q;
          X_unroll(h_unroll, w_unroll) = X(c, h_out + p, w_out + q);
        }
    }
```

(G) How many times on average will be each X element be replicated by the unrolling kernel?

Answer:

The size of the unrolled matrix will be C*K*K x H_out*W_out.
The size of the input feature maps is C*(H_out+K-1)*(W_out+K-1)

The ratio of the two gives the answer: K*K *(H_out*W_out)/((H_out+K-1)*(W_out+K-1).
Note that C and M does not play a role. Why?

You should explore the effect of different H_out and W_out on the answer. What happens when H_out and W_out are much larger than K? What is the intuition?

(H) Think about the level of parallelism (number of thread blocks) that will be present when launching matrix multiplication kernel for a convolution layer, (1) for the C1 layer in the LeNet, (2) towards the end of the network, such as C5.

Parallelization

- Understanding dependence constraints
- Understanding communitivity and associativity

Question: Consider the following fragment of C code:

```
for(unsigned int x = 0; x < 512; ++x) {
  for(unsigned int y = 0; y < 512; ++y) {
    for(unsigned int z = 0; z < 512; ++z) {
      out[x][y] = out[x][y] <OP> in[x][y][z];
    }
  }
}
```

Explain how you would optimally parallelize this code and why if:
(a) <OP> was not associative nor communitive
(b) <OP> was associative and communitive

You need to state to what you will assign your blocks and your threads to and why, then write out the kernel function code.

Assume out[x][y] is initialized correctly.

Part (a):

<OP> is not associative/communitive => the z loop must be done sequentially
The values of x and y are both small (<=512) so there is no need to go to 2D grids.
y is the contiguous dimension => threads assigned to y dimension
remaining dimension is x => blocks assigned to x dimension

Kernel code:

```
unsigned int x = blockIdx.x;
unsigned int y = threadIdx.x;
tmp = out[x][y]; // Use temporary to avoid global memory access
for(unsigned int z = 0; z < 512; ++z) {
```

```
            tmp = tmp <OP> in[x][y][z];
    }
    out[x][y] = tmp;
```

Part (b):

<OP> is associative and communitive => the z loop could be done using a reduction by a block
=> assign a thread for each z value
assign a block (256 threads) to each (x,y) pair

Kernel code:

```
unsigned int y= blockIdx.y;
unsigned int x = blockIdx.x;
unsigned int z = threadIdx.x;
__shared__ in_out_s[512]; // Use shared memory to avoid global access

in_out_s[z] = in[x][y][z];

for(unsigned int stride = 256; stride >= 1; stride >>= 1) {
   __syncthreads(); // Don't forget to sync
   if(z < stride) { // Stride in a manner that favors coalescing
      in_out_s[z] = in_out_s[z] <OP> in_out_s[z + stride];
   }
}

__syncthreads(); // The synchronization here is not necessary

if(z == 0) {
   out[x][y] = in_out_s[z];
}
```