The Final Exam (total 15 points, due at 9:00 on 27.02.26)

Working in groups and/or using any kind of AI is strictly prohibited! If detected, the exam result will be annulled!

Problem 1 (2 points)

You need to write a kernel that operates on an image of size 3840 x 2160 pixels. Each thread processes one pixel by forming a convolution of a rectangular 9x7 mask centered at the pixel with the appropriate number of surrounding pixels (assume that each pixel is characterized by four floating point parameters), with the "ghost" pixels not contributing to the result. You would like your threadblocks to be square, the mask to be separate for each parameter and to be stored in shared memory. Assuming your device has CUDA compute capability 8.6 and you select the grid dimensions and block dimensions of your kernel to ensure maximum occupancy

Questions:

1) How many warps will have control divergence?
2) Assuming that the algorithm bottleneck is caused by global memory operations, what would be the speedup factor if the pixel data get shifted to the shared memory buffer?

Problem 2 (5 points)

Assume that a student wrote the following simple kernel with 1d execution configuration (each thread calculates one element of the final vector P) for multiplication of a square matrix and a corresponding vector, where Width is not a multiple of blockDim.x:

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

  P[Row] = Pvalue;
}
```

Questions:

1) Find all errors in the code
2) How would you implement data reuse by utilizing a shared memory buffer (provide the modified code)? Assuming that the cache hierarchy does not play a significant role, by what factor will the effective global memory bandwidth increase after this optimization?
3) Will the "shared memory" version from 2) be compute- or memory-bound (assume that you have a RTX 3090 GPU)?

4) Suppose that you want to replace the “for” loop with the parallel reduction algorithm based on the binary tree concept. How should the thread-data mapping and the execution configuration be modified to get the best performance? Give the corresponding code modifications.

Problem 3 (1 point)

Assume you have a new RTX 3090 with the DRAM system having peak bandwidth of 936 GB/s. Assume a thread block size of 1024, warp size of 32, and that A is a floating point precision array in the global memory.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
float temp = A[9*i];
```

What is the maximal global memory effective bandwidth we can hope to achieve in the following access to A with the L1 cache disabled?

Problem 4 (1 point)

Starting from the Fermi architecture, NVIDIA engineers have decreased latency of performing atomic functions by executing them in the L2 cache memory, provided that the needed data is there. Should it not be the case, the data is moved to the cache and then the atomic function is performed.

For a processor that supports atomic operations in L2 cache, assume that each atomic operation takes 4 ns to complete in L2 cache and 100 ns to complete in DRAM. Assume that 35% of the atomic operations hit the L2 cache. What is the approximate throughput for atomic operations on the same global memory variable?

Problem 5 (3 points)

The following scalar product code tests your understanding of the basic CUDA model.The following code computes 2,048 dot products, each of which is calculated from a pair of 256-element vectors. Use the code to answer the following questions. Assume the compute capability of your device to be 8.0.

```
1  #define  VECTOR_N  2048
2  #define  ELEMENT_N  256
3  const int  DATA_N = VECTOR_N*ELEMENT_N;
4  const int  DATA_SZ = DATA_N*sizeof(float);
5  const int  RESULT_SZ = VECTOR_N*sizeof(float);
```

```
. . .
6  float *d_A, *d_B, *d_C;
. . .
7  cudaMalloc((void **)&d_A, DATA_SZ);
8  cudaMalloc((void **)&d_B, DATA_SZ);
9  cudaMalloc((void **)&d_C, RESULT_SZ);
. . .
10  scalarProd <<< VECTOR_N, ELEMENT_N >>>(d_C, d_A, d_B, ELEMENT_N);
11
12  __global__  void
13  scalarProd(float *d_C, float *d_A, float *d_B, int ElementN)
14  {
15    __shared__  float  accumResult[ELEMENT_N];
16    //Current vectors bases
17    float *A = d_A + ElementN*blockIdx.x;
18    float *B = d_B + ElementN*blockIdx.x;
19    int  tx = threadIdx.x;
20
21    accumResult[tx] = A[tx] * B[tx];
22
23    for  (int  stride = ElementN / 2;  stride > 0;  stride >>= 1)
24    {
25      __syncthreads();
26      if  (tx < stride)
27        accumResult[tx] += accumResult[stride + tx];
28    }
30    d_C[blockIdx.x]  = accumResult[0];
31  }
```

Questions:

1) How many accesses to shared memory are done for each block?
2) List the source code lines, if any, that cause shared memory bank conflicts.
3) Identify an opportunity to reduce the bandwidth requirement on the global memory. How would you achieve this? How many accesses can you eliminate?

Problem 6 (2 points)

Consider the scan (prefix sum) kernel based on the algorithm of Harris (actually, it was originally proposed by Blelloch) discussed in the class (e.g., Lecture 9, Slide 22)

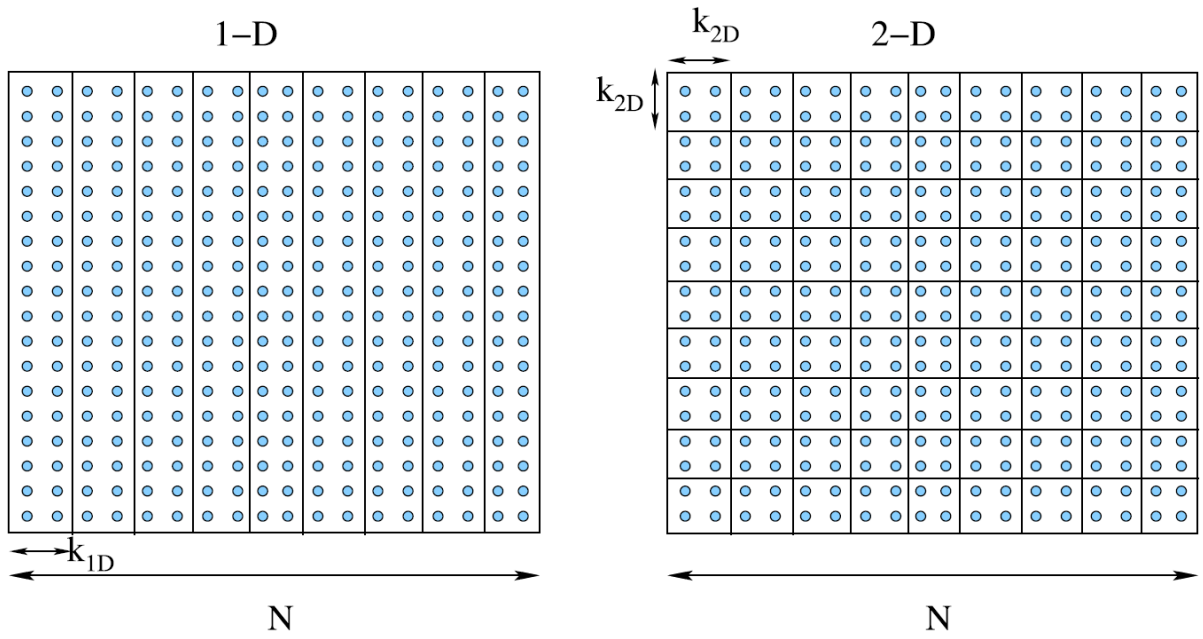
Questions:

1) What would happen if you removed the last __synchthreads() in the algorithm?
2) Assume that you 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?

Problem 7 (1 point)

Suppose that you need to solve a huge heat diffusion problem using a stencil-based method resulting from finite differencing (see slide 25 in lecture 6), and the N x N cell data do not fit into a single GPU. Rather, you have P gpus at your disposal and you would like to use all of them. Assuming that

1) N is evenly divisible by P
2) Neglecting the intra-node communication and assuming that the inter-node communication is full-duplex (sending and receiving data can be made simultaneously), whereas each inter-node data communication has communication startup latency t_{start} , and the communication cost per value per data transaction between two gpus t_{comm} .
3) That the processing cost per single update for a single cell is t_{comp}



, estimate the t_{start}/ t_{comm} ratio, when the 1D problem decomposition (see the left figure, here each gpu node has k_{1D} = N/P cells in x direction and N cells in the y direction) leads to a better algorithm performance than the 2D problem decomposition (see the right figure, assume that here $k_{2D}^2=N^2/P$, since the work should be evenly divided between P gpus).