IIT JODHPUR

Major Examination: CSL7520 GPU Programming

Guidelines (Total time: 180 minutes, Maximum Marks: 50):

- Please read the question paper very carefully. **NO clarification is required in any question.** In case of any doubt, assume whatever you wish to and state that in your answer.
- Usage of LLM/AI tool is NOT allowed. Hints are provided in few questions. Please follow them.

---

1. Suppose we wish to implement the famous ML model of K-Means algorithm (which essentially means separation of data samples into a certain number of clusters that are identified by their respective centroids). The steps of this learning model are listed in the following step-by-step procedure:

```
K-MEANS({x_1, ..., x_N}, K)
 1  (s_1, s_2, ..., s_K) ← SELECTRANDOMSEEDS({x_1, ..., x_N}, K)
 2  for k ← 1 to K
 3  do μ_k ← s_k
 4  while stopping criterion has not been met
 5  do for k ← 1 to K
 6     do ω_k ← {}
 7     for n ← 1 to N
 8     do j ← arg min_j' |μ_j' − x_n|
 9        ω_j ← ω_j ∪ {x_n}  (reassignment of vectors)
10     for k ← 1 to K
11     do μ_k ← (1/|ω_k|) Σ_{x∈ω_k} x  (recomputation of centroids)
12  return {μ_1, ..., μ_K}
```

   If you decide to implement the above algorithm with CUDA programming paradigm for any of the NVIDIA GPU architectures, how would you proceed (assuming large values of data samples)? Hint: For CUDA programming, you can take the help of code fragments in the next questions. [6]

2. Inefficient code development is a serious obstacle in achieving maximum performance from GPU hardware. Explain *how bad/inefficient* is the code in the fragments (a) & (b) as shown below? Why? [6]

(a)

```
__shared__ float partialSum[];

unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
     stride < blockDim.x; stride *= 2)
{
   __syncthreads();
   if (t % (2 * stride) == 0)
      partialSum[t] += partialSum[t + stride];
}
```

(b)

```
__shared__ float partialSum[];

unsigned int t = threadIdx.x;
for (unsigned int stride = (blockDim.x >> 1);
     stride > 0; stride >>= 1)
{
   __syncthreads();
   if (t < stride)
      partialSum[t] += partialSum[t + stride];
}
```

3. You are supplied the below code for implementation of some image processing application. For the images, let's say that we consider width = 2048 and height = 2048. For the below code, it is given that each array element is a 32-bit float stored in global memory. You can also assume that all loads and stores shown are from/to global memory (no caching/shared-memory reuse considered in this scenario).

   In order to evaluate performance of the above kernel, Compute to Global Memory Access (CGMA) ratio can be quite useful and may help us in further optimizations such as changing the memory layout etc. Therefore, compute the CGMA metric for **(a) one interior grid point (which means that a thread that executes the body of if), and (b) the entire interior region (all points that satisfy the if) ?** [Hint: Note that CGMA ratio = FLOPs/(number of global memory accesses)]. [6]

```
__global__ void stencil2D(const float* __restrict__ in,                    float*
__restrict__ out,  int width, int height)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;

   if (x >= 1 && x < width - 1 &&
      y >= 1 && y < height - 1)
   {
      int idx = y * width + x;
      float center = in[idx];
      float left   = in[idx - 1];
      float right  = in[idx + 1];
      float up     = in[idx - width];
      float down   = in[idx + width];
      float sum = center + left + right + up + down;
      out[idx] = 0.2f * sum;
   }
}
```

4. LU decomposition or factorization of a matrix is the factorization of a given square matrix into two triangular matrices, one upper triangular matrix and one lower triangular matrix, such that the product of these two matrices gives the original matrix. It is a fundamental technique (as shown below) in linear algebra used to solve systems of linear equations, invert matrices, and compute determinants.

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Decompose $[A] = [L][U]$

$$\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Below is an example of the above technique that also highlights how the decomposed matrices could be obtained by repeatedly solving the resultant equations:

$$A = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 8 & 14 \\ 2 & 6 & 13 \end{bmatrix} = LU \text{ where } L = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \text{ and } U = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}.$$

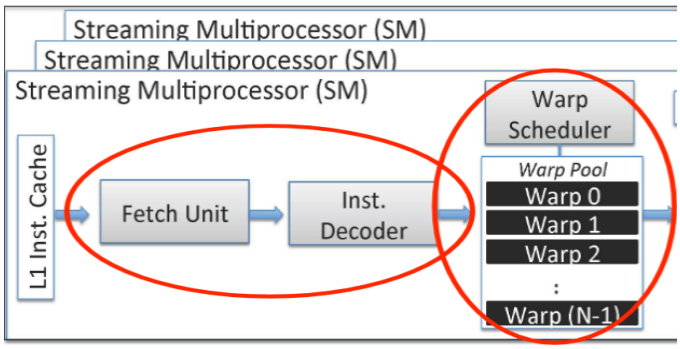
Multiplying out LU and setting the answer equal to A gives

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21}U_{11} & L_{21}U_{12} + U_{22} & L_{21}U_{13} + U_{23} \\ L_{31}U_{11} & L_{31}U_{12} + L_{32}U_{22} & L_{31}U_{13} + L_{32}U_{23} + U_{33} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 8 & 14 \\ 2 & 6 & 13 \end{bmatrix}.$$

With proper optimization opportunities, how could you implement a CUDA kernel for achieving the LU decomposition for large-sized matrices (that are typical in real-life applications)? [6]

5. As different warps may have different execution times on GPU hardware, warp scheduling is an important step to ensure efficient execution of the user code on the GPU hardware. So, warp scheduler design must be thought carefully. A proposal for warp scheduler design (as shown in below Figure) is to make it criticality-aware. This proposal outlines the following steps:

   - Assign warps different weight based on their criticality.
   - The scheduling algorithm prioritizes and schedule warps by the weight of warps.
   - Slower warps receive more time slots to run in advance.



Taking a suitable example (let's say- Breadth First Search implementation on large-sized graphs), examine if the above warp scheduling algorithm is actually useful to some extent? [6]

6. Consider width = 1024 and height = 1024, both A and B tightly packed row-major, base addresses 128-byte aligned. Consider the configuration of dim3 blockDim(32, 1), dim3 gridDim(32, 32). Let's focus on block (0,0) and warp 0 (all threads in that block), please answer the following: [3+3]

   - For the load A[y * width + x], determine if this access is fully coalesced. Justify by computing the addresses and segments.
   - For the store B[x * height + y], determine if warp 0's store is coalesced or not, and quantify the number of 128-byte segments touched.

```
__global__ void ValUpdate(float *A, float *B, int width, int height)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bx = blockIdx.x;
    int by = blockIdx.y;

    int x = bx * blockDim.x + tx;
    int y = by * blockDim.y + ty;

    if (x < width && y < height)
    {
        // A is row-major: A[y][x]
        float val = A[y * width + x];

        // B is also row-major, but we write a transpose-like pattern
        // B[x][y]
        B[x * height + y] = val;
    }
}
```

7. It appears that the below kernel for matrix multiplication is reasonably well written. (a) Explain how varying the values of *tx* and *ty* impacts the overall performance of the code? (b) What would be the impact when __syncthreads() function is removed from the code? [4 + 2]

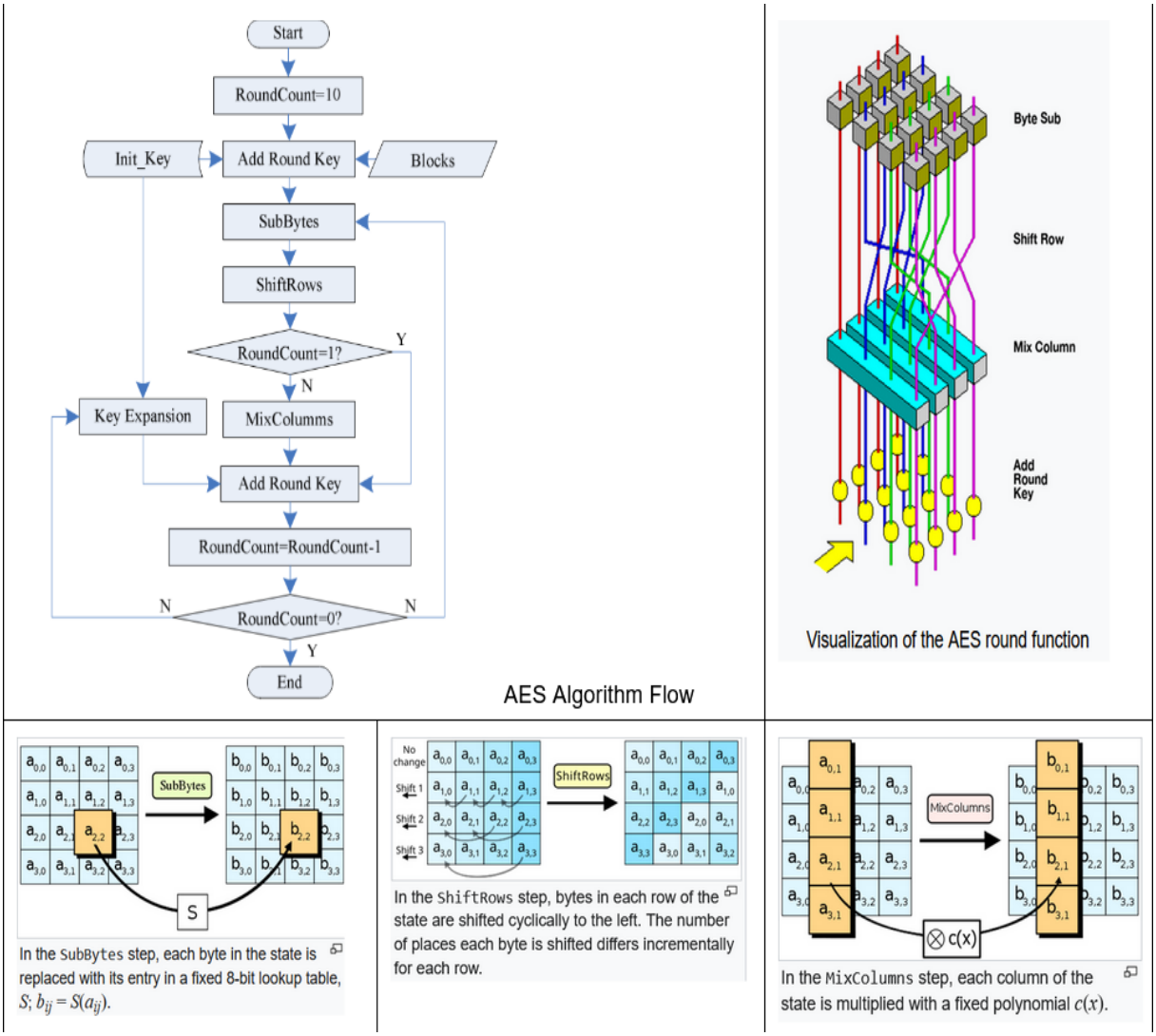
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

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

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;
    // Loop over the Md and Nd tiles required to compute the Pd element
    for (int m = 0; m < Width / TILE_WIDTH; ++m)
    {
        // Collaborative loading of Md and Nd tiles into shared memory
        Mds[ty][tx] = Md[Row * Width + (m * TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[(m * TILE_WIDTH + ty) * Width + Col];
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[Row * Width + Col] = Pvalue;
}
```

8. The AES Encryption algorithm (also known as the Rijndael algorithm) is a symmetric block cipher algorithm with a block/chunk size of 128 bits. It converts these individual blocks using keys of 128. Once it encrypts these blocks, it joins them together to form the ciphertext. It is based on a substitution-permutation network, also known as an SP (substitution-permutation) network. It consists of a series of linked operations, including replacing inputs with specific outputs (substitutions) and others involving bit shuffling (permutations). The key steps of this algorithm are shown as below:

   - Substitution-Permutation Network (SPN): This is a framework for building block ciphers. It takes a block of plaintext and a secret key as input and applies a series of operations in multiple "rounds".
   - Rounds: Each round consists of a set of mathematical operations & these rounds are based on an SPN structure.
   - Substitution: This step uses S-boxes (substitution boxes) to replace parts of the data with other parts, adding confusion to the relationship between the key and the ciphertext.



   - Compare the two strategies of parallelization of the AES algorithm: a) Block-level parallelism which refers to many blocks, sequential rounds per block. b) Round-level pipelining which refers to rounds as pipeline stages for a stream of blocks. c) Intra-round parallelism which refers to parallelize SubBytes/ShiftRows/MixColumns within a round. [6]
   - Compare these two choices regarding impact on performance when parallelized version of AES is executed: One thread per 16-byte block vs one thread per 4-byte word vs one warp per block? [2]