GPU Architectures and Programming
Assignment- Week 6
TYPE OF QUESTION: MCQ/MSQ

Number of questions: 10 **Total mark: 10 X 1 = 10**

MCQ Question

Question 1:
A CUDA kernel traverses a 1024×1024 matrix, where each element is a single-precision floating-point number. The kernel will sum all the elements of the matrix and return the total sum. The matrix is stored in row-major order in global memory and Block size is 16×16.Which of the following is the most efficient way to traverse the matrix using CUDA threads in the kernel, ensuring coalesced memory access and efficient use of shared memory?

A. Each thread accesses elements in the matrix in a column-major order, with each thread in the block loads elements into shared memory, and then the entire block performs the summation of its tile.
B. Each block processes a row of the matrix, and threads within the block access elements in a column-major order.
C. Each thread processes a single element in the matrix, with threads accessing elements in the matrix row by row. Each thread in the block loads elements into shared memory, and then the entire block performs the summation of its tile.
D. Each thread accesses one element of the matrix and stores the partial sum into global memory immediately after reading the element then again access partial sum from global memory to perform final summation..

Answer: C
Solution:
A-Accessing elements in column-major is inefficient since threads would access non-contiguous memory locations, leading to poor memory coalescing
B-Processing by rows is correct in terms of memory coalescing, but accessing elements in column-major order within a block is inefficient since threads would access non-contiguous memory locations, leading to poor memory coalescing
D-Storing the partial sums in global memory immediately after reading each element introduces high latency and unnecessary global memory accesses, which can severely slow down performance.

Common data for Questions 2-3:

Consider the following CUDA kernel, which adds corresponding elements of two 1D arrays A and B of size N, and stores the result in array C of size N. Where N>256 and wrap size is 32..The variable stride indicates the step size between successive memory accesses for a thread. Following kernel is launched with a block size of 256 threads.

```
__global__ void addArraysWithStride(int *A, int *B, int *C, int N, int stride) {
```

```
    int index = threadIdx.x + blockIdx.x * blockDim.x;  // Global thread index

    if (index < N && (index%4==0 || index%4==1)) {
        C[index] = A[index] + B[index];  // Add corresponding elements
        index += stride;  // Move by stride for subsequent accesses
        C[index] = A[index] + B[index];
    }
}
```

Question 2:
If the value of stride is 2, then for a thread with threadIdx.x = 5, the thread will access the elements of the arrays at indices.

A. [6,7]
B. [5, 7]
C. [5, 8]
D. None of the above.

Answer: B
Solution:
If the stride is 2 and the thread index is 5, the thread will first access element 5. After moving by a stride of 2, the next element it accesses will be 7. Therefore, the correct indices for this thread are [5, 7].

Question 3:
If the value of stride is 4, then for a thread with threadIdx.x = 19, the thread will access the elements of the arrays at indices.

A. 19 and 21
B. 20 and 22
C. 18 and 20
D. None of the above.

Answer: D
Solution:
19%4=3. Hence thread with threadId.x=19 will not access any data element in the matrix.

Question 4:

Consider an array of size N is stored sequentially in global memory and each thread swaps one element from the beginning with one element from the end of the array, the following kernel is launched with 128 threads per block and 4 blocks. In total, how many warp READ accesses (i.e., global memory read accesses made by all warps) will occur when performing the array reversal given that each warp consists of 32 threads ? (Assume no effect of Cache subsystem and N=1024)

A. 16 warp accesses
B. 32 warp accesses
C. 64 warp accesses
D. 128 warp accesses

Answer:B
Solution:
Each warp accesses 64 elements during the reversal operation (32 from the start and 32 from the end of the array).As there are 1,024 elements in the array and each warp processes 64 elements, the

number of warps required to reverse the array is:
1,024/64=16 warps
Each warp performs 2 memory accesses:
2 read accesses (1 from the beginning and 1 from the end).
So, the total number of warp accesses is the number of warps multiplied by the number of accesses per warp:
16 warps×2 accesses per warp=32 warp accesses.

Question 5:

You are performing matrix multiplication on a GPU for two matrices A and B, both sized 4096×4096, resulting in a matrix C that is also of the same size. Each element in the matrices requires 4 bytes of storage. The GPU has a warp size of 32.The multiplication kernel uses tiling with size 32×32 and thread block dimensions also as 32 × 32. The GPU has 96 KB of shared memory per multiprocessor (SM). If each SM can support a maximum of 2048 threads, calculate the percentage of shared memory utilization per SM.

A. 15.57%
B. 16.67%
C. 20.0%
D. 18.33%

Answer:B
Solution:
Tile size per block: 32×32
Threads per block=32×32=1024
Each block requires two shared memory tiles for matrices A and B:
Matrix A tile: 32×32×4=4096 bytes
Matrix B tile: 32×32×4=4096 bytes
Total shared memory per block=4096+4096=8192 bytes=8 KB
The maximum number of threads per SM is 2048, so the number of thread blocks that can fit = 2048 / 1024 =2 blocks
Shared memory used by 2 blocks=2×8 KB=16 KB
Memory Usage Percentage=(16 KB/96 KB)×100=16.67%

Common data for Questions 6-8:

Consider the following code snippet executing on a GPU architecture where the number of shared memory banks is 32 and the bank width is 4 bytes.

```
#define SZ 32
__global__ void setRowReadCol(float *out) {
__shared__ int tile[SZ][SZ];
unsigned int gid = threadIdx.x * blockDim.y + threadIdx.y;
tile[threadIdx.x][threadIdx.y] = gid;
__syncthreads();
int tidx = ______ ;
int tidy = _______;
out[gid] = tile[tidx][tidy];
}
```

Question 6:
The kernel is launched with parameters <<<(32,32),(32,32)>>>. Consider the last shared load operation. The accesses for this operation depends on the values of tidx and tidy. State whether the following statement is true or false.

"For tidx=threadIdx.y, tidy=threadIdx.x, shared load transactions are bank conflict free"

A. True
B. False

Answer: A
Solution:
Since, the size of the shared tile is 32x32 and the number of banks is 32, each column of the tile gets mapped to a single bank. Now threads are packed into warps as follows.
warp 0: (0,0) ,(0,1),.....,(0,31)
warp 1: (1,0) ,(1,1),.....,(1,31)
and so on.
The first element of each pair is the threadIdx.y value while the second element is the threadIdx.x value. Thus for any warp, the values of the threadIdx.x change for a fixed value of threadIdx.y. Given this, shared memory access expressions of the form tile[threadIdx.y][threadIdx.x] would be bank conflict free, since threads in any warp access elements from different columns.

Question 7:
For the code snippet in Question 5, state whether the following statement is true or false.
"For tidx=threadIdx.x, tidy=threadIdx.y, shared load transactions are bank conflict free"

A. True
B. False

Answer: B
Solution:
For tile[threadIdx.x][threadIdx.y], shared load transactions experience bank conflicts, since each warp essentially accesses a column of the matrix which is mapped to the same bank.

Question 8:
For the code snippet in Question 5, state whether the following statement is true or false.
"For tidx=threadIdx.x, tidy=threadIdx.x, shared load transactions have bank conflicts"

A. True
B. False

Answer: B
For tile[threadIdx.x][threadIdx.x], the principal diagonal elements belonging to different columns are accessed by each warp. So there are no bank conflicts.

Question 9:
In a GPU system with memory transaction width of N, a global memory access can coalesce (bring in a single transaction) N consecutive floating point data values. Consider the following code snippet.

```
__global__ void mem_access(float* A)
{
        int tid = threadIdx.x;
        for(int i=1; i<=32; i=i*2)
```

```
        A[tid*i]+=2;
}
```

Let the number of threads be 256 (tid = 0 to 255) and the size of the array A be 8192. Assuming a warp size of 16, compute the total number of global memory transactions made in the for loop for transaction widths of 16 elements. Assume no caching occurs.

A. 752
B. 992
C. 768
D. 1024

Answer: A

Solution:

Given the fact that the total number of threads is 256 and the warp size is 16, the total number of warps is 16. For every transaction width, the total number of memory transactions will be the sumtotal of the number of memory transactions in each iteration which are as follows

iteration 1: Array elements 0 to 256 are accessed. Warps of size 16 can access 16 elements at a time for transaction width 16 aTotal number of memory transactions is therefore 256/16

iteration 2: Array elements 0,2,4,...,512 are accessed. Warps of size 16 can access 8 elements at a time for transaction width 16 Total number of memory transactions is 256/8

iteration 3: Array elements 0,4,8,... 1024 are accessed. Warps of size 16 can access 4 elements for transaction width 16. Total number of memory transactions is 256/4.

iteration 4: Array elements 0,8,16,... 2048 are accessed. Warps of size 16 can access 2 elements for transaction width 16 Total number of memory transactions is 256/2.

iteration 5,6: Warps of size 16 can access only 1 element. Number of memory transactions for both iterations will be 256/1.

The total number of memory transactions for warp size of 16 and transaction width 16 will be 256/16+256/8+256/4+256/2+256+256=752

Question 10:

Consider the following kernel code snippet.

```
#define BDIMX 32
#define BDIMY 32
__global__ void setRowReadCol(int *out)
{
__shared__ int tile[BDIMY][BDIMX];
unsigned int row_idx = threadIdx.y * blockDim.x + threadIdx.x;
unsigned int col_idx = threadIdx.x * blockDim.y + threadIdx.y;
tile[row_idx] = row_idx;
__syncthreads();
out[row_idx] = tile[col_idx];
}
```

The kernel is executed on a GPU architecture where the number of shared memory banks is 16 and the bank width is 4 bytes. The kernel is launched with the following configuration.

```
dim3 block (BDIMX, BDIMY);
dim3 grid (1,1);
```

Assuming the size of an integer is 4 bytes, match and pair the following.

i. Number of shared loads per warp request	a. 1
ii. Number of shared stores per warp request	b. 16
iii. Number of global loads per warp request	c. 32
iv. Number of global stores per warp request	

A. i → b, ii->b, iii->a, iv->a
B. i → b, ii->a, iii->a, iv->a
C. i → c, ii->c, iii->a, iv->a
D. i → a, ii->a, iii->b, iv->b

Answer: B

*********END**********