Objective MCQs

State True or False

Question 1

Considering a GPU architecture where the warp size is 8, the access expression A[tid+8] is coalesced. Here tid=blockDim.x*blockIdx.x+threadIdx.x
State true or false.

Answer: True

Question 2

A thread block can be distributed across multiple SMs
Answer: False

Question 3

Multiple thread blocks can execute in a single SM.
Answer: True

Question 4

Considering a GPU architecture where the warp size is 8, the access expression "A[tid*8]" is coalesced. Here, tid=blockDim.x*blockIdx.x+threadIdx.x State true or false.
Answer: False

Question 5

Local memory for each thread in GPU is implemented in external DRAM.

Answer: True

Question 6

Warps are executed in an SIMD fashion i.e. the warp scheduler launches warps of threads and each warp typically executes one instruction across parallel threads.

Answer: True

Question 7

For a GPU architecture with warp size **8**, the array access expression "**A[tid+16]**"

where **tid=blockDim.x*blockIdx.x+threadIdx.x** is not coalesced.

Answer: False

Question 8

For a GPU architecture with warp size 32, if an SM has 128 SPs, it can execute at most 2 Warps at a given time.

Answer: False

Question 9

For a GPU architecture with warp size 16, the array access expression "A[16+tid*16]"

where tid=blockDim.x*blockIdx.x+threadIdx.x is coalesced

Answer: False

Question 10

Padding shared memory may be used for reducing shared bank conflicts.

Answer: True

Short Numericals

Question 1

Consider a 1D Array **A** of **N** integers where each element in **A[i]** is initialized to **i%3**.

```
__global__ void selectiveIncrement(int* A,int N)
{
      int tid = blockIdx.x*blockDim.x+threadIdx.x;
   if(A[tid]%2!=0)
      A[tid]++;
}
```

What is the sum of the **9** elements **A[0]** to **A[8]** after the selectiveIncrement kernel executes and modifies **A**?

Options:

A. 9
B. 16
C. 15
D. 0

Answer: C

Question 2

Consider a 1D Array A of N integers where each element in **A[i]** is initialized to **i%2.**

```
__global__ void selectiveIncrement(int* A,int N)
{
      int tid = blockIdx.x*blockDim.x+threadIdx.x;
```

```
        if(A[tid]%2!=0)
            A[tid]++;
}
```

What is the sum of the 16 elements A[0] to A[15] after the **selectiveIncrement** kernel executes and modifies A?

Options:
A.9
B.16
C.15
D.0

Answer: B

Question 3

Consider a CUDA kernel with launch parameter configuration **<<<(32,1,1),(32,1,1)>>>** executing on a GPU architecture with **8** SMs and **128** SPs per SM. The occupancy of a CUDA kernel is equal to the number of active threads executing in one SM divided by the total number of SPs in one SM. What is the occupancy of the given CUDA kernel executing on the GPU architecture?

Options:
A. 25%
B. 50%
C. 75%
D. 100%

Answer: D

Question 4

Consider a CUDA kernel with launch parameter configuration **<<<(16,1,1),(16,1,1)>>>** executing on a GPU architecture with **8 SMs** and **32 SPs per SM**. The occupancy of a CUDA kernel is equal to the number of active threads executing in one SM divided by the total number of SPs in one SM. What is the occupancy of the given CUDA kernel executing on the GPU architecture?

Options:
A.25%
B.50%

C.75%
D.100%

Answer: D

Question 5

Assume there is a 1D input array **A** of **N** integers, where each element is of the form **A[i]=2*i** , where **i = 0** to **N-1**. Consider the following 1D kernel code snippet executing on a GPU architecture where the warp size is **8**.

```
__global__ void divBranch(int* A, int* B, int M, int N, int k){
     int tid = blockIdx.x * blockDim.x + threadIdx.x;
     if(A[tid]%2)
       B[tid]=log(A[tid])
      else
       B[tid]=sqrt(A[tid])

}
```

During the lifetime of **warp '0'**, how many sqrt instructions are executed?

Options:

A. 0
B. 4
C. 8
D. 16

Answer: C

Memory Questions

Question 1

Consider the following code snippet executing on a GPU architecture where the number of shared memory banks is 8 and the bank width is 4 bytes.

```
#define SZ 64
 __global__ void setRowReadCol(float *out)
 {
      __shared__ float shared_mem[SZ];
      unsigned int gid = threadIdx.y * blockDim.x + threadIdx.x;
      unsigned int tid = threadIdx.x;
      shared_mem[tid] = A[gid];
       __syncthreads();
      apply_transform(shared_mem[expr]);
 }
```

The above kernel is launched with a configuration of <<<32,8>>>. The function apply_transform is a device function. Consider the following different expressions for the last shared load statement which is the argument to this function and determine whether shared memory bank conflicts occur or not for each expression.

a) expr= (tid+1)*7	i) shared memory bank conflicts occur
b) expr=tid*8	ii) shared memory bank conflict free
c) expr=(tid+1)*6 + 1	

Select the correct match and pair options from below.

A. a->ii, b->i, c->i
B. a->ii, b->ii, c->i
C. a->i, b->i, c->i
D. a->ii, b->ii, c->ii
E. None of the above

Answer: A

Question 2

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
#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];
 }
```

The kernel is launched with parameters <<<(32,32),(32,32)>>>. Consider the last assignment statement. The accesses for the last shared load statement depends on the values of tidx and tidy. Determine whether shared memory bank conflicts occur or not for shared loads of tile given the following values of tidx and tidy.

a) tidx=threadIdx.y tidy=threadIdx.x	i) shared memory bank conflicts occur
b) tidx=threadIdx.x tidy= tidx	ii) shared memory bank conflict free
c) tidx=threadIdx.x tidy= SZ-1-threadIdx.x	

Select the correct match and pair options from below.

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

E. None of the above

Answer: A

Question 3

Consider the following code snippet executing on a GPU architecture where

i) width of L1 cache line is 64 bytes

ii) width of L2 cache line is 16 bytes

iii) warp size is 16 As we know global memory access width = |warp|*4 bytes = 64 bytes.

```
__global__ void mem_access(float *out, float *in)
{
      int tidx = blockIdx.x*blockDim.x + threadIdx.x;
      out[tidx] = in[tidx*4];
}
```

Assume that the number of threads being launched is 512. The size of the in array is 2048 and the size of the out array is 512. What are the total number of L1 writes to L2 cache?

Answer: 512

Question 4

Consider the following code snippet executing on a GPU architecture where
i) width of L1 cache line is 32 bytes
ii) width of L2 cache line is 8 bytes
iii) warp size is 8. As we know global memory access width = |warp|*4 bytes = 32 bytes.
Thus global memory access will be 32 bit aligned.

```
__global__ void mem_access(float *out, float *in)
{
int tidx = blockIdx.x*blockDim.x + threadIdx.x;
out[tidx] = in[tidx*2];
}
```

Assume that the number of threads being launched is 512. The size of the in array is 1024 and the size of the out array is 512. What is the total number of L1 writes to L2 cache for the array in?

Answer: 512

REDUCTION

Question 1

Consider the following reduction kernel code snippet.

```
__global__ void reduce( int * g_idata , int * g_odata , unsigned
int n ) {
     extern __shared__ int sdata [];
     unsigned int tid = threadIdx . x ;
     unsigned int i = blockIdx . x * blockDim . x + threadIdx .
     x ;
     sdata [ tid ] = ( i < n ) ? g_idata [ i ] : 0;
     __syncthreads () ;
     for(unsigned int s=1; s < blockDim.x; s *= 2)
     {
          if((tid % (2*s)) == 0)
               sdata [ tid ] += sdata [ tid + s ];
          __syncthreads () ;

     }

     if ( tid == 0)
          g_odata [ blockIdx . x ] = sdata [0];
}
```

Assume a host program which launches the above reduction kernel multiple times. The maximum number of threads in a block is 64 in each run. For an array of size 2^24, how many times will the kernel be invoked? Assume that the GPU device is only used to perform the reduction operation

Answer: 4

Question 2

Consider the following reduction kernel code snippet.

```
__global__ void reduce( int * g_idata , int * g_odata , unsigned
int n ) {
     extern __shared__ int sdata [];
     unsigned int tid = threadIdx . x ;
     unsigned int i = blockIdx . x * blockDim . x + threadIdx .
     x ;
     sdata [ tid ] = ( i < n ) ? g_idata [ i ] : 0;
     __syncthreads () ;
     for(unsigned int s=1; s < blockDim.x; s *= 2)
     {
          if((tid % (2*s)) == 0)
               sdata [ tid ] += sdata [ tid + s ];
          __syncthreads () ;

     }

     if ( tid == 0)
          g_odata [ blockIdx . x ] = sdata [0];
}
```

Assume a host program which launches the above reduction kernel multiple times. The maximum number of threads in a block is 256 in each run. For an array of size 2^24, how many times will the kernel be invoked?

Answer: 3

Question 3

Consider the following reduction kernel code snippet.

```
__global__ void reduce(int *g_idata, int *g_odata, unsigned int
n){
     extern __shared__ int sdata[];
     unsigned int tid = threadIdx.x;
     unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
     sdata[tid] = (i < n) ? g\_idata[i] : 0;
     __syncthreads();
     for (unsigned int s=blockDim.x/2; s>0; s>>=1)
```

```
    {
      if  (tid < s)
        sdata[tid] += sdata[tid + s];
      __syncthreads();
    }
    if (tid == 0)
      g_odata[blockIdx.x] = sdata[0];
}
```

Assume a host program which launches the above reduction kernel multiple times. The maximum number of threads in a block is 1024 in each run. For an array of size 2^24, how many times will the kernel be invoked? Assume that the GPU device is only used to perform the reduction operation.

Answer: 3

Question 4

Consider the following reduction kernel code snippet.

```
__global__ void reduce(int *g_idata, int *g_odata, unsigned int
n){
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = (i < n) ? g\_idata[i] : 0;
    __syncthreads();
    for (unsigned int s=blockDim.x/2; s>0; s>>=1)
    {
      if  (tid < s)
        sdata[tid] += sdata[tid + s];
      __syncthreads();
    }
    if (tid == 0)
      g_odata[blockIdx.x] = sdata[0];
```

```
}
```

Assume there exists a host program which launches the above reduction kernel multiple times. The maximum number of threads in a block is 32 in each run. For an array of size 2^24, how many times will the kernel be invoked? Assume that the GPU device is only used to perform the reduction operation.

Answer: 5

Fusion/Coarsening Questions

Question 1

Consider the given GPU architecture-

- Streaming Multiprocessors (SM): 20
- Max. active threads per SM: 2048
- Max. thread blocks per SM: 32
- Registers per SM: 64K
- Shared Memory per SM:128KB

The given kernel transpose of a given data set (without any coarsening).

```
# define TILE_DIM 32
# define BLOCK_ROWS 32
__global__ void transposeCoalesced ( float *odata , float *idata
,const int nx ,const int ny)
{
     __shared__ float tile [ TILE_DIM ][ TILE_DIM ];
     int x = blockIdx .x * TILE_DIM + threadIdx .x;
     int y = blockIdx .y * TILE_DIM + threadIdx .y;
     int width = gridDim .x * TILE_DIM ;
     for (int j = 0; j < TILE_DIM ; j += BLOCK_ROWS )
          tile[threadIdx.y+j][threadIdx.x] =
     idata[(y+j)*width+x];
```

```
    __syncthreads ()
    x = blockIdx .y * TILE_DIM + threadIdx .x;
    y = blockIdx .x * TILE_DIM + threadIdx .y;
    for (int j = 0; j < TILE_DIM ; j += BLOCK_ROWS )
        odata[(y+j)*width+x] =
    tile[threadIdx.x][threadIdx.y+j];
}
```

The calling parameter in host code is given below

```
dim3 block ( 32, 32, 1);
dim3 grid ( 64, 64, 1);
transposeCoalesced <<<grid , block >>>(odata , idata , nx , ny);
```

Calculate the optimal coarsening factor for block-level coarsening, assuming the kernel is coarsened only along the x-dimension for the above program.

Options:

A. 2
B. 4
C. 8
D. 16
E. None of the above

Answer: D

Question 2

Consider the given GPU architecture.

- Streaming Multiprocessors (SM): 20
- Max. active threads per SM: 2048
- Max. thread blocks per SM: 32
- Registers per SM: 64K

● Max Shared Memory per SM: 128KB

The given kernel performs transpose of a given data set (without any coarsening).

```
# define TILE_DIM 32
# define BLOCK_ROWS 32
__global__ void transposeCoalesced ( float *odata, float *idata,
const int nx ,const int ny)
{ __shared__ float tile [ TILE_DIM ][ TILE_DIM ];
     int x = blockIdx .x * TILE_DIM + threadIdx .x;
     int y = blockIdx .y * TILE_DIM + threadIdx .y;
     int width = gridDim .x * TILE_DIM ;
     for (int j = 0; j < TILE_DIM ; j += BLOCK_ROWS )
          tile[threadIdx.y+j][threadIdx.x] =
     idata[(y+j)*width+x];
     __syncthreads ()
     x = blockIdx .y * TILE_DIM + threadIdx .x;
     y = blockIdx .x * TILE_DIM + threadIdx .y;
     for (int j = 0; j < TILE_DIM ; j += BLOCK_ROWS )
          odata[(y+j)*width+x] =
     tile[threadIdx.x][threadIdx.y+j];
}
```

The launch parameter in host code is given below

```
dim3 block ( 32, 32, 1);
dim3 grid ( 64, 64, 1);
transposeCoalesced <<<grid , block >>>(odata , idata , nx , ny);
```

Calculate the optimal coarsening factor for thread-level coarsening assuming that the kernel is coarsened only across the x-dimension for the above program.

Options:

A. 2
B. 4
C. 8

D. 16
E. None of the above

Answer: D

Question 3

Consider two independent CUDA kernels as given below to be run on a target GPU with 16 SMs and each SM supporting a maximum of 2048 active threads per SM. Assume there are as many SFUs as the number of active threads per SM.

Calling parameters are :

```
kernel1<<< 1024, 1024 >>>, kernel2<<< 1024, 1024 >>>
```

Kernel descriptions follow next.

```
__global__ void kernel1 (float  out1, float  in1, float  in2 )
{
     int  l_tid = threadIdx.x ;
     if (l_tid<512)
          out1[g_tid] = in1[g_tid] + in2[g_tid] ;
     else
     out1[g_tid] = sin(in1[g_tid]) + cos( in2[g_tid] );
}

__global__ void kernel2 (float  out2, float  in1, float  in2 )
{
     int  l_tid = threadIdx.x ;
     if (l_tid<512)
          out2[g_tid] = sin(in1[g_tid]) + cos( in2[g_tid] );
     else
     out2[g_tid] = in1[g_tid] + in2[g_tid] ;
}
```

Assume that each float type sin/cos operation takes 20 cycles and each float addition operation takes 4 cycles. Also, ignore penalties incurred due to memory accesses and data transfer overhead. Calculate the total number of cycles it will take to compute for the inner thread fused version of the two given kernels. Consider execution time only due to sin, cos, addition operations.

Answer: 3072

Question 4

Consider the following two kernels that are to be fused. The target GPU has 32 SMs and supports a maximum 1024 active threads per SM. Assume there are as many SFUs as the number of active threads per SM. Assume that block scheduling to SM happens here in synchronous fashion.

Calling parameter is kernel1<<< 512, 1024 >>>, kernel2<<< 512, 1024 >>>

```
__global__ void kernel1 (float  *out1, float  in1, float  in2 )
{
int  l_tid = threadIdx.x ;
     if (l_tid<512)
          out1[g_tid] = in1[g_tid] + in2[g_tid] ;
     else
     out1[g_tid] = sin(in1[g_tid]) + cos( in2[g_tid] );
}

__global__ void kernel2 (float  out2, float  in1, float  in2 )
{
     int  l_tid = threadIdx.x ;
     if (l_tid<512)
          out2[g_tid] = sin(in1[g_tid]) + cos( in2[g_tid] );
     else
     out2[g_tid] = in1[g_tid] + in2[g_tid] ;
}
```

Assume that each *sin,cos* operation takes 20 cycles and each floating point addition operation takes 4 cycles. Ignore penalties incurred due to memory accesses and data transfer overhead. Calculate the total number of cycles it will take to compute for the inner thread fused version for the given two kernels. Consider execution time only due to sin, cos, addition operations.

Answer: 1536

Question 5

For the given two kernels identify the suitable type of fusion for getting better performance compared to executing them individually. The target GPU has 32 SMs and supports a maximum 1024 active threads per SM. Assume there are as many SFUs as the number of active threads per SM. Assume that block scheduling to SM happens here in synchronous fashion.

Calling parameter is kernel1<<< 512, 1024 >>>, kernel2<<< 512, 1024 >>>

```
__global__ void kernel1 (float  *out1, float  in1, float  in2 )
{
int  l_tid = threadIdx.x ;
     if (l_tid<512)
          out1[g_tid] = in1[g_tid] + in2[g_tid] ;
     else
     out1[g_tid] = sin(in1[g_tid]) + cos( in2[g_tid] );
}

__global__ void kernel2 (float  out2, float  in1, float  in2 )
{
     int  l_tid = threadIdx.x ;
     if (l_tid<512)
          out2[g_tid] = sin(in1[g_tid]) + cos( in2[g_tid] );
     else
     out2[g_tid] = in1[g_tid] + in2[g_tid] ;
}
```

Assume that blocks are scheduled in a round robin fashion to the underlying hardware i.e. block 0 is mapped to SM 0, block 1 is mapped to SM1 etc. Calculate the total number of cycles it will take (only due to sin,cos,addition) to compute for the **inter block fused** version of the given two kernels.

Answer: 1536

Question 6

Consider two independent CUDA kernels as given below to be run on a target GPU with 16 SMs and each SM supporting a maximum of 2048 active threads per SM. Assume there are as many SFUs as the number of active threads per SM.

Calling parameters are :

```
kernel1<<< 1024, 1024 >>>, kernel2<<< 1024, 1024 >>>
```

Kernel descriptions follow next.

```
__global__ void kernel1 (float  out1, float  in1, float  in2 )
{
     int  l_tid = threadIdx.x ;
     if (l_tid<512)
          out1[g_tid] = in1[g_tid] + in2[g_tid] ;
     else
     out1[g_tid] = sin(in1[g_tid]) + cos( in2[g_tid] );
}

__global__ void kernel2 (float  out2, float  in1, float  in2 )
{
     int  l_tid = threadIdx.x ;
     if (l_tid<512)
          out2[g_tid] = sin(in1[g_tid]) + cos( in2[g_tid] );
     else
     out2[g_tid] = in1[g_tid] + in2[g_tid] ;
}
```

Assume that blocks are scheduled in a round robin fashion to the underlying hardware i.e. block 0 is mapped to SM0, block 1 is mapped to SM1 etc. Calculate the total number of cycles it will take (only due to sin,cos,add) to compute for the inter block fused version of the two kernels given in question 9.

Answer: 3072