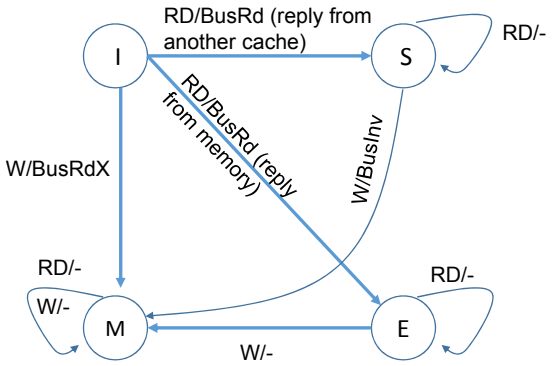
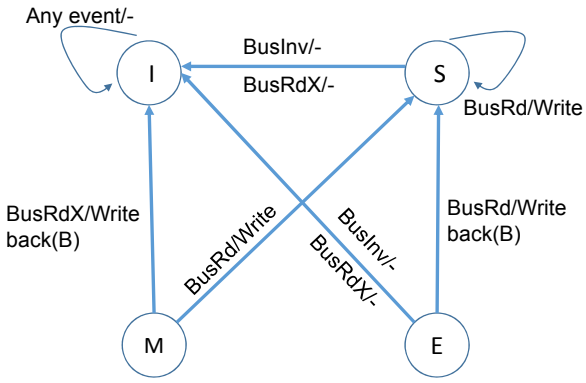
Cache coherence:

1. May give you the transition diagrams for MSI (slides 20 and 21) and ask you to provide a similar transition diagrams for MESI.

Response to processor events:
- Read(RD)
- Write (W)



Response to Bus events:
- Read request for block B: BusRd (B),
- Write request for block B: BusRdX(x),
- Invalidate block B: BusInv(B)



Cache coherence:

1. May give you the transition diagrams for MSI (slides 20 and 21) and ask you to provide a similar transition diagrams for MESI.

2. May give you the left most column of slide 17 and ask you to fill up the other two columns

3. May ask you to specify the sequence of messages between the Local, Host and Remote nodes for a particular scenario of blocks (ex. Slide 35-38)

Chapter 4:

4. For each of the following questions, you need to justify your answer.

Using vector instructions can you implement a “load with stride” operation using “load with gather” operation?

Using a vector instructions can you implement a “load with gather” operation using “load with stride” operation?

Which one of these sentences is true and which is false?

- All the threads of a thread block execute in lock-step — False
- _syncthreads() is a barrier for all the threads in a thread block — True
- Variable declared as _global_ in a CUDA kernel are allocated in the shared memory — False
- Shared memory in CUDA is shared by all the threads in a kernel — False
- Global memory in CUDA is shared by all the threads — True
- cudaMemcpy() can be called from a Kernel to copy data between host and global memory — False
- cudaMemcpy() is used to copy data between host and global memory — True

Assuming that you wrote a cuda kernel that declares a shared memory array consisting of 4K bytes and that the compiler determined that each thread in that kernel needs 16 integer registers. Assume also that your GPU has 4 SMs, each with a register file of 2048 integer registers and a shared memory of 16K bytes. If your application will execute kernel <<<nblocks, blksize>>>, answer the following questions:

- What is the maximum number of threads that can execute simultaneously on the GPU?
  Each SM has 2048 registers and each thread needs 16 registers
  → each SM can support 128 threads → 4 SMs can support 512 threads
- What is the maximum number of thread blocks that can execute simultaneously on an SM??
  Each SM has 16K bytes of shared memory and each thread block needs 4K bytes
  → each SM can support at most 4 thread blocks simultaneously.
- To execute the maximum number of threads simultaneously what is the value of nblocks and blksize that you would use when launching the kernel
  Kernel<<<16,32>> or <<<8, 64>>> or <<4, 128>>

Show the output of the content of array A after the execution of the following program:

```
_global_ F(int *A)
{
      int  idx = blockIdx.x * blockDim.x + threadIdx.x ;
       A[idx] = idx ;
       A[blockIdx.x] = blockIdx.x ;
} ;

 void main()
{  Allocate a 16 element int array A  in the GPU global memory and initialize its elements to 0 ;
   F<<<2,4>>>(A) ;
}
```

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]	A[13]	A[14]	A[15]
0	1	2	3	4	5	6	7	0	0	0	0	0	0	0	0

Show the output of the content of array A after the execution of the following program:

```
_global_ F(int *A)
{     int row = blockIdx.y * blockDim.y + threadIdx.y ;
      int  col  = blockIdx.x * blockDim.x + threadIdx.x ;
       A[row][col] = blockIdx.x + blockIdx.y + threadIdx.x ;
} ;

 void main()
{  Allocate an 6x6 array A  in the GPU global memory ;
  initialize A's elements to 0 ;
  dim3 grid(2,2) ;     // a 2x2 array of blocks
  dim3 blocks(3,3); // each block is a 3x3 array of threads
   F<<<grid,blocks>>>(A) ;
}
```

x

A[0,0]	A[0,1]	A[0,2]	A[0,3]	A[0,4]	A[0,5]
0	1	2	1	2	3
A[1,0]	A[1,1]	A[1,2]	A[1,3]	A[1,4]	A[1,5]
0	1	2	1	2	3
A[2,0]	A[2,1]	A[2,2]	A[2,3]	A[2,4]	A[2,5]
0	1	2	1	2	3
A[3,0]	A[3,1]	A[3,2]	A[3,3]	A[3,4]	A[3,5]
1	2	3	2	3	4
A[4,0]	A[4,1]	A[4,2]	A[4,3]	A[4,4]	A[4,5]
1	2	3	2	3	4
A[5,0]	A[5,1]	A[5,2]	A[5,3]	A[5,4]	A[5,5]
1	2	3	2	3	4

y

Show the output of the content of array A after the execution of the following program:

```
_global_ F(int *A)
{     int row = blockIdx.y * blockDim.y + threadIdx.y ;
      int  col  = blockIdx.x * blockDim.x + threadIdx.x ;
       A[threadIdx.y][threadIdx.x] = blockIdx.x ;
} ;

 void main()
{ Allocate an 6x6 array A  in the GPU global memory ;
  initialize A's elements to 0 ;
  dim3 grid(2,2) ;     // a 2x2 array of blocks
  dim3 blocks(3,3); // each block is a 3x3 array of threads
   F<<<grid,blocks>>>(A) ;
}
```

x

A[0,0]	A[0,1]	A[0,2]	A[0,3]	A[0,4]	A[0,5]
0 or 1	0 or 1	0 or 1	0	0	0
A[1,0]	A[1,1]	A[1,2]	A[1,3]	A[1,4]	A[1,5]
0 or 1	0 or 1	0 or 1	0	0	0
A[2,0]	A[2,1]	A[2,2]	A[2,3]	A[2,4]	A[2,5]
0 or 1	0 or 1	0 or 1	0	0	0
A[3,0]	A[3,1]	A[3,2]	A[3,3]	A[3,4]	A[3,5]
0	0	0	0	0	0
A[4,0]	A[4,1]	A[4,2]	A[4,3]	A[4,4]	A[4,5]
0	0	0	0	0	0
A[5,0]	A[5,1]	A[5,2]	A[5,3]	A[5,4]	A[5,5]
0	0	0	0	0	0

y

Rewrite the following cuda kernel without using shared memory. The kernel adds n integers stored in the global array "input[]" into a global variable, "total", and is called as

reduce<<<nb, n/nb>>(input, n, total)

Where n is multiple of nb.

```
_global_void reduce(int *input, int *n, int *total_sum)
{
      int tid = threadIdx.x;
      int idx = blockIdx.x*blockDim.x + threadIdx.x;
      _shared_ int x[blocksize];
      x[tid] = input[idx] ;
      _syncthreads();

      for(int h=blockDim.x/2; h>0; h=h/2)
      {
            if(tid < h) x[tid] += x[tid + h];
            _syncthreads();
      }
      If (tid == 0 ) atomicAdd(total_sum, x[tid]);
      }
```

```
_global_void reduce(int *input, int n, int *total_sum)
{
      int tid = threadIdx.x;
      int idx = blockIdx.x*blockDim.x + threadIdx.x;

      for(int h=blockDim.x/2; h>0; h=h/2)
      {
            if(tid < h) input[idx] += input[idx + h];
            _syncthreads();
      }
      If (tid == 0 ) atomicAdd(total_sum, input[idx]);
}
```