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 max 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.
  Note that the limit of 128 threads/Sm (4 warps/SM) places an additional limit on the max number of blocks that can execute 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:

```c
__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);
}
```

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

```c
__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 a 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);
}
```

<p>| | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| x | y |   |   |   |   |   |
Show the output of the content of array A after the execution of the following program:

```c
_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);
}
```

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.

```c
_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 half=blockDim.x/2; half>0; half=half/2) {
        if(tid < half) x[tid] += x[tid + half];
        _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 half=blockDim.x/2; half>0; half=half/2) {
        if(tid < half) input[idx] += input[idx + half];
        _syncthreads();
    }
    if (tid == 0) atomicAdd(total_sum, input[idx]);
}
```
What is wrong with the following code?

```c
__global__ F(int *A)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    A[idx] = idx;
    if (idx < blockDim * gridDim / 2) __syncthreads();
    A[blockIdx.x] = blockIdx.x;
}
```

`__syncthreads()` is allowed in conditional code only if the conditional is uniform across the entire thread block.