Questions 1: Find the bisection bandwidth (the bisection width * link bandwidth), the total network bandwidth (the sum of the bandwidths of all the links), the diameter and the node degree for each of the following network topologies assuming that the bandwidth of each link in a network is B. You can find the definitions of the above measures in the book (Section 6.8) and on the slides.

- A ring network with N nodes (see the figure on page 536)
- A 2-D torus with N nodes (called 2-D grid in Figure 6.14).
- A binary tree network with N nodes (not a fat tree)
- A binary hypercube network with N nodes (N is a power of 2).

<table>
<thead>
<tr>
<th>Topology</th>
<th>Bisection band width</th>
<th>Network Bandwidth</th>
<th>Diameter</th>
<th>Node degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ring</td>
<td>2 * B</td>
<td>N * B</td>
<td>N/2</td>
<td>2</td>
</tr>
<tr>
<td>2-D torus</td>
<td>2 \sqrt{N} * B</td>
<td>2 * N * B</td>
<td>\sqrt{N}</td>
<td>4</td>
</tr>
<tr>
<td>Binary tree</td>
<td>B</td>
<td>\geq 1.5 N * B</td>
<td>\geq 2 \log N</td>
<td>1, 2 or 3</td>
</tr>
<tr>
<td>Binary hypercube</td>
<td>B * N / 2</td>
<td>B * N * (log N) / 2</td>
<td>Log N</td>
<td>Log N</td>
</tr>
</tbody>
</table>

Note that the above table assumes that B is the bidirectional bandwidth of a link connecting two nodes. That is, if a link connects nodes, a and b, then B is the sum of the bandwidth from a to b and the bandwidth from b to a.
**Question 2.** Given the cuda code for parallel reduction of N integers

```
_global_void plus_reduce(int *input, int N, int *total)
{
    int tid = threadIdx.x;
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    _shared_ int x[blocksize];
    x[tid] = input[i]; // assuming that N is a multiple of the block size
    _syncthreads();

    for(int s=blockDim.x/2; s>0; s=s/2)
    {
        if(tid < s) x[tid] += x[tid + s];
        _syncthreads();
    }
    If (tid == 0 ) atomicAdd(total, x[tid]); /* total is a global variable */
}
```

Rewrite the kernel without using shared memory.

```
_global_void plus_reduce(int *input, int N, int *total)
{
    int tid = threadIdx.x;
    int i = blockIdx.x*blockDim.x + threadIdx.x;

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

**Question 3.** 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 2 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:

a. What is the maximum number of threads that can execute on the GPU?
   - Each SM has 2048 registers and each thread needs 16 registers → each SM can support 128 threads → 2 SMs can support 256 threads

b. What is the maximum number of thread blocks that can execute on the GPU??
   - Each SM has 16K bytes of shared memory and each thread block needs 4K bytes → each SM can support 4 thread blocks → 2SMs can support 8 thread blocks.

c. To execute the maximum number of threads what is the value of nbblocks and blksize that you would use when launching the kernel
   - `Kernel<<<8, 32>>>` or `<<<4, 64>>` or `<<<2,128>>`
**Question 4.** True/False questions:

- 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**

**Question 5.** 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>
</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>
</tr>
</tbody>
</table>

**Question 6.** What is wrong with the following cuda code?

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

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