1. **In CUDA C, what line of code would you write to statically allocate an array of 100 `float` values called `d_values` in GPU constant memory?**

   Answer: __constant__ float d_values[100];

   Notes: Many people wrote __device__ instead of __constant__, which would just store the array in generic global memory. Might have been a case of not reading the question carefully enough?

2. **Name one technique that can be used to increase the speed of data transfers between host and device memory, and briefly describe this technique works.**

   Answer: Pinned memory prevents the data stored in host memory from being moved around due to memory paging. This means that a direct memory access (DMA) transfer between the CPU and GPU can be done, which is much faster.

   Notes: See lecture 2. Some people said to use shared memory L1/L2 cache, but this is wrong as these are techniques to speed up memory access between the GPU's global memory and a thread, not between the GPU and CPU.

3. **Look at the following code from part of a CUDA kernel:**

```
__global__ void myKernel() {
        int array[9];
        for (int i = 0; i < 5; i++) {
                array[i * 2] = i;
        }
```

   In which type of memory are:

   a. **The nine `floats` contained in `array`**
   b. **The local variable `i`**

   most likely to be stored in?

   a. Local memory (which is actually part of global memory, so either was accepted)

   b. Registers

Notes: There was a small mistake in the question (the code should have had `float array[9]` instead of `int array[9]`) but this doesn't change the answer. Local variables created inside a kernel are stored in registers where possible, except if there isn't enough register space available or (as is the case here) if the variable is an array that is accessed using a non-constant index. Here we access the array element i*2, which isn't a constant value, so the array has to go in local memory (see lecture 2 slides 46-48). Some people wrote L1 or L2 cache, which is also technically correct as it's very likely that `array` will be cached automatically.

4. **Fetch-decode-execute pipelining is one example of instruction-level parallelism. Briefly describe how fetch-decode-execute pipelining can increase performance.**

   Answer: Fetch-decode-execute pipelining occurs within a single processing core. It involves decoding the next instruction and fetching the next next instruction from memory at the same time as executing the current instruction. It can increase performance threefold, assuming the next instruction can be predicted.

   Notes: See lecture 1 slides 16-17.

5. **You are writing a CUDA kernel to do a scientific simulation of some physical process. This simulation has around 100 different parameters, which do not change during the course of a simulation but must be read by each kernel. What would be the most appropriate type of memory to store these parameters in?**

   Answer: Constant memory (which is cached in the uniform cache, so that was also marked as correct). This memory is optimized for when multiple threads read a value simultaneously, but the values in constant memory can't be changed in kernel code.

   Notes: See lecture 2, slides 51-56.

6. **A modern GPU contains two levels of cache: L1 and L2. Briefly describe one difference between the L1 and L2 caches in a GPU.**

   Answer: Various options:
   - L1 cache is smaller, L2 cache is bigger.
   - L1 cache is faster, L2 cache is slower.
   - (Biggest difference) L1 cache is specific to each streaming multiprocessor (SM), whereas L2 cache is shared between all SMs.

   Notes: See lecture 2, slides 31-43.

7. **Look at the following code, which launches a CUDA kernel called `myKernel`:**

```
dim3 gridSize(10, 10, 1);
dim3 blockSize(5, 5, 2);
myKernel<<<gridSize, blockSize>>>();
```

In total, how many threads will be created for this kernel launch?

Answer: Each block will contain 5*5*2 = 50 threads. There will be 10*10*1=100 blocks, so in total there will be 50*100=5000 threads.

Notes: The most common error was to miss the *2 in the block size – be careful!

8. **A data processing application has been designed to run on a GPU with 768KB of L2 cache and 16 streaming multiprocessors. The input data are organised into 500KB chunks in global memory, and the code is organised such that each block processes a different chunk of data.**

Is the GPU's L2 cache likely to offer a significant performance boost in this example? Briefly explain why / why not.

Answer: No. Each block processes a different 500KB chunk of data. If there were an individual L2 cache for each streaming multiprocessor (SM) then each block could access its chunk from cache, but since L2 cache is shared between all SMs at most one and a half chunks can actually be cached.

Notes: See lecture 2, slides 31-43. You could potentially quibble over the meaning of "significant" here, so perhaps the question was a little ambiguous.

9. **An MxN matrix (M rows, N columns) is stored in a one dimensional array:**

```
float d_matrix[M*N];
```

The matrix is stored in <u>row major</u> order, i.e. all the elements of the first row are stored at the start of the array, then the elements of the second row, and so on.

Write a formula that will give the index of the matrix element at row i and column j, where i is between 0 and M-1 and j is between 0 and N-1.

Answer: i*N+j

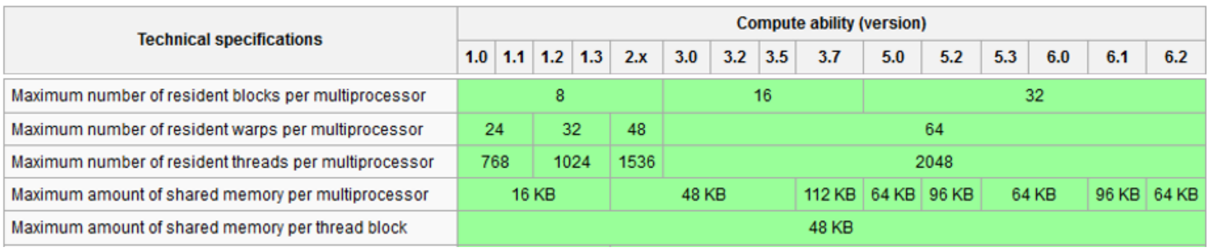
Notes: Very few people got this completely right! Each row has N elements, so the index of the first element in row i is i*N. You then need to access the j'th element in that row, which you get by just adding j. Try drawing this out if it's still not clear.

10. **The performance of a GPU can be optimized by maximizing occupancy, i.e. the number of threads that are resident in each streaming multiprocessor. Briefly describe why higher occupancy can give better performance.**

Answer: Each warp can be in one of three states: running, ready, or waiting. The waiting state normally means that the threads in the warp are currently in the middle of reading or writing data to/from memory. If every warp in a streaming multiprocessor (SM) is waiting, that SM can't do anything and is "stalled". A stalled SM is a waste of resources, so we want to avoid that. By having as many threads (and therefore warps) resident in an SM as possible, we maximize the chance that one of them is in the ready state.

Notes: Your answer didn't need to be as long as the one above; you just need to say that increased occupancy increases the chance that one warp will be in the ready state. See lecture 3, slides 15-23.

11. **The following table shows some properties relating to the different levels of CUDA Compute Ability (also called Compute Capability):**



Technical specifications	Compute ability (version)														
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2
Maximum number of resident blocks per multiprocessor	8					16				32					
Maximum number of resident warps per multiprocessor	24		32		48	64									
Maximum number of resident threads per multiprocessor	768		1024		1536	2048									
Maximum amount of shared memory per multiprocessor	16 KB				48 KB				112 KB	64 KB	96 KB	64 KB		96 KB	64 KB
Maximum amount of shared memory per thread block	48 KB														

Consider a CUDA kernel that uses blocks of 200 threads, which also makes use of a shared memory array of size 10KB. When this kernel runs on a Compute Capability 5.0 device, what will be the maximum number of blocks that can be resident in a multiprocessor?

Answer: 6 blocks. You need to consider all the things that can limit occupancy, and take the minimum of them. The maximum number of threads is 2048, which would allow ten blocks of 200 threads. However, each block uses 10KB of shared memory, and there is only 64KB available per multiprocessor. This means that only six blocks will fit (64/10=6, round down).

Notes: See lecture 3 slides 20-22 and lecture 5 slides 70-72.

12. **Coalesced memory access results in faster memory transfers, and happens when multiple threads simultaneously access neighbouring memory addresses. Consider an application that involves simulating a large number of particles, where each particle has fifty floating point attributes associated with it. Each thread is responsible for updating the state of one particle.**

    There are two strategies for how the particles can be stored in global memory:

    a. **Store all of the attributes of the first particle first, followed by all of the attributes for the second particle, and so on.**
    b. **Store the first attribute values for all of the particles first, followed by the second attribute values for all of the particles, and so on.**

    Which of these strategies is more likely to result in memory accesses to the particles array being coalesced?

    Answer: Strategy b. Each thread updates one particle, so all the threads will all be accessing the same attribute of their respective particles simultaneously. If the same attributes are stored together in memory (strategy b) then threads in a block will be accessing nearby memory addresses.

    Notes: See lecture 3, slides 24-40.

13. **Consider a CUDA kernel which takes as input an array of N floating point values. This array is then divided into non-overlapping chunks of M floating point values, giving N/M chunks in total (assume M evenly divides N). Each thread reads all of the values in a chunk from global memory, calculates the sum of them, and stores the result in an output array. The code for this kernel is shown below:**

```
__device__ float d_input[N];
__device__ float d_output[N / M];

__global__ void addKernel()
{
      int globalId = blockIdx.x * blockDim.x + threadIdx.x;
      float sum = 0.0f;
      if (globalId < N / M) {
            for (int i = 0; i < M; i++) {
                  sum += d_input[globalId * M + i];
            }

            d_output[globalId] = sum;
      }
}
```

What is the compute to global memory access (CGMA) ratio for this kernel? Give the value that the CGMA ratio will tend towards as M becomes larger.

Answer: Because there is no shared memory being used each thread is completely independent, so we can just consider how many floating point operations and memory accesses a single thread does.

Floating point computations: Each thread adds up M floating point values, so in total there are M-1 operations.

Memory accesses: Each thread reads M values from global memory and writes back one result, so in total there are M+1 memory accesses.

So $CGMA = \frac{M-1}{M+1} \approx \frac{M}{M} = 1$

Notes: Lots of people had trouble with this, so I'll go through it in the lecture and give you some more CGMA practice before the real test. The most important thing is to decide whether you're working out the total number of operations / memory accesses for a thread or a block, and to be consistent. Another common mistake was to include the operations used to calculate indices (e.g. `globalId * M + i`) – since these aren't floating point operations you should ignore them.

14. When CUDA threads co-operate to load some data into shared memory it is important to call the `__syncthreads()` function between the code to load the data and the code to access it from shared memory. Why is this?

Answer: You need to make sure all threads have finished loading their data into shared memory before the code continues and accesses it. The __syncthreads function will force all threads to wait until they've all got to that point in the code, ensuring that they've all finished loading.

Notes: See lecture 4, slide 21.

15. **In general, using if / else statements in a kernel can cause branch divergence, where each warp must be executed multiple times. Look at the following kernel:**

```
if ((threadIdx.x < 32) == 0) {
      doSomething();
}
else {
      doSomethingElse();
}
```

Will this if / else statement result in branch divergence? Explain why / why not.

Answer: No. Remember that divergence will occur if threads within a warp follow a different path through the code. A block is split into 32 thread warps simply by taking the first 32 threads in the block for the first warp, the next 32 for the second warp, etc. So in this example, all of the threads in the first warp (threadIdx.x between 0 and 31) will call doSomething(), and all of the other threads will call doSomethingElse().

Notes: This was the question fewest people got right! See lecture 3, slides 11-14 and lecture 4, slides 4-20.

You can use this blank page for working (it won’t be marked)

You can use this blank page for working (it won't be marked)

You can use this blank page for working (it won’t be marked)