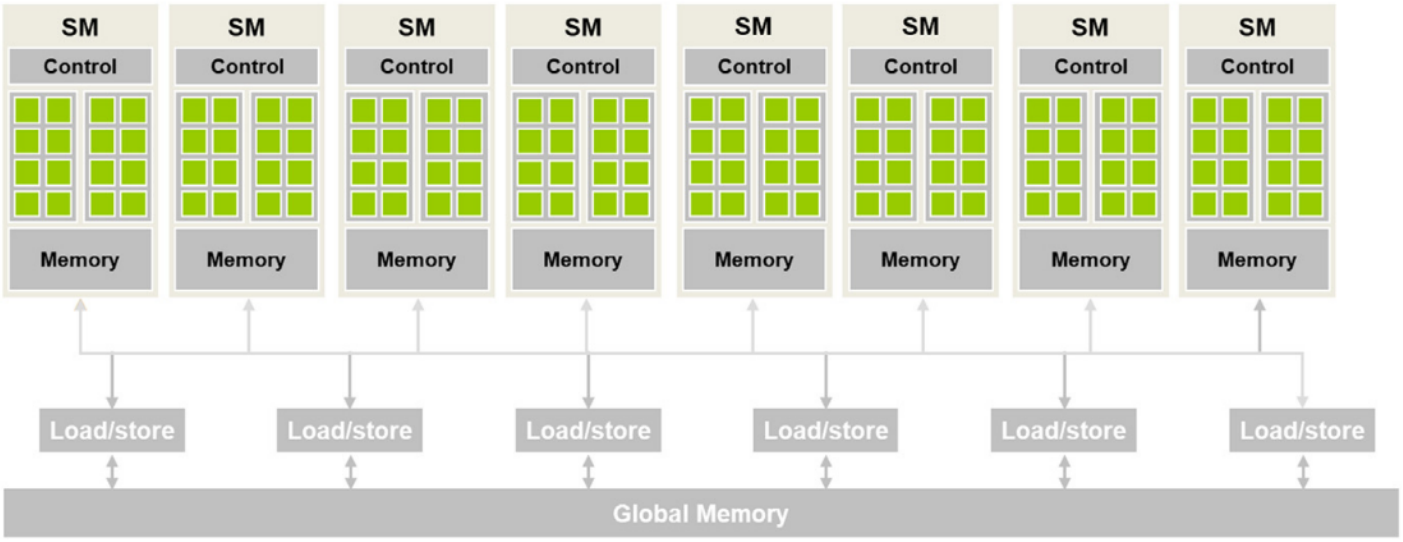
Chapter 4 notes

Streaming Multiprocessors (SM)

Each GPU has little groups of processing units called Streaming Multiprocessors or SMs. These SMs are basically a group of CUDA cores that have a shared memory and logic control unit. The shared memory is different from global memory, which is a memory that is shared among multiple SMs themselves.

Each SM also has a register file that can be accessed by the CUDA Core.



Older GPUs had GDDR SDRAM for the SM memory (not global mem, Graphics Double Data Rate, Synchronised Dynamic RAM). This was not that efficient for computational tasks. After that we got HBM or high bandwidth memory, where the memory was consisting of multiple DRAMs stacked on top of each other. This saved space. Later they improved this and we got HBM2 and HBM2e (which is HBM2 with even more bandwidth). An A100 Device has 108 SMs and each SM can have 64 CUDA cores.

A certain number of blocks gets assigned to an SM. Since the number of SMs is finite, there exists a maximum number of blocks a CUDA machine can execute at the same time. Hence there's a job queue which allows already executed blocks of threads to be taken over by new unexecuted blocks.

Since threads in the same blocks are on the same SM, threads on the same block can have some shared properties and can interact with each other. One of this is barrier synchronization of threads.

Barrier synchronization means syncing threads at a given point in a program. it is done by calling `__syncthreads().` we can only call it on all the threads in a block and not some specific threads in a block. It will impact every single thread. Not doing that causes a deadlock. This usually happens in divergent control flow (an if statement affecting a specific set of threads).

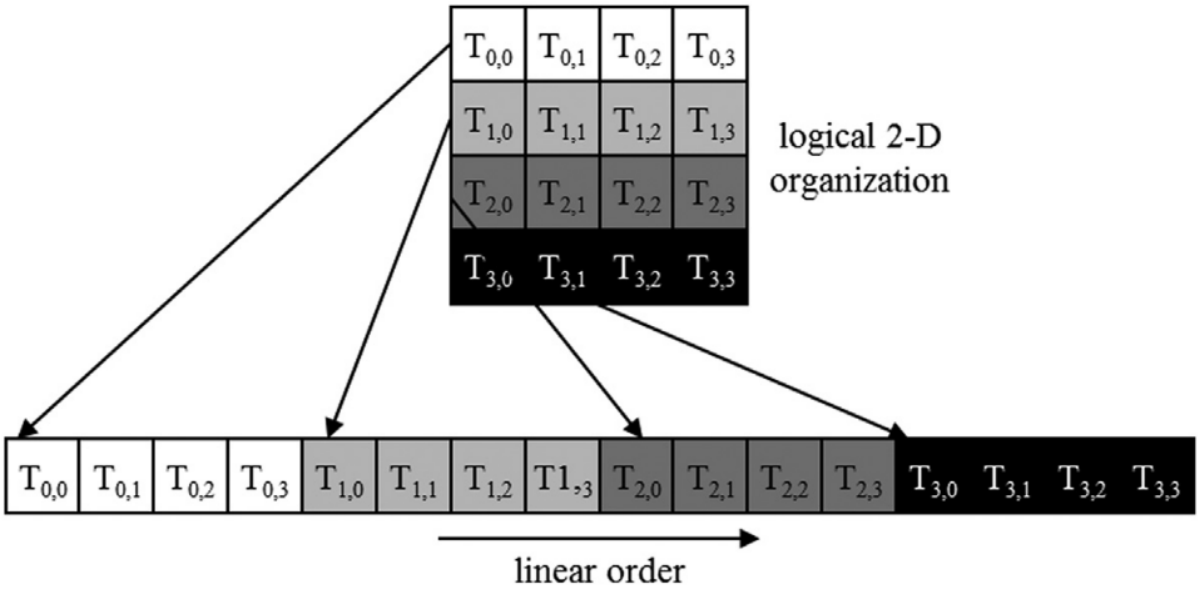
It is good practice to ensure that the correctness of our program's output does not depend on threads being synchronized.

SIMD Hardware and Warps

So warps are the groups of 32 threads in a block on which the same instruction is executed. The threads in a block are divided into warps based on the thread index. threadIdx.x 0 to 31 is the first warp, 32 to 63 is the second warp.. and so on.

when there's more than one dimension, this happens for threadIdx.y too, and the threads are linearized in a row-major column in blocks of 32. if there's any threads that are not called but they are not exactly multiples of 32, CUDA will fill inactive threads to complete the warp till the next index

for example, there's only 16 threads in the next example, so 16 more inactive threads will be filled in



We know about thread hierarchy: Grids -> Blocks -> Threads

However there's a hardware hierarchy for arrangement too, which doesn't necessarily correspond to grid->block->thread. Each SD contains multiple CUDA cores, but those CUDA cores inside the SD are also organized into processor blocks. So the hardware hierarchy becomes SD -> processor blocks -> CUDA cores. Each processor block in ampere architecture has 16 cores, and each SD has 64 cores so there's 4 processor blocks.

Control divergence and independent thread scheduling

it's straightforward to understand what might happen during a SIMD execution of a single decision flow (no if/else or conditional flow). However, it becomes tricky when we have multiple decision flow. In multiple decision flow, say an if/else statement, some threads in a warp will execute the if statement and some will execute the else statement.

If we think about it this is not single instruction multiple data anymore because the instructions in the two paths will be different. **The way SIMD resolves this is by running the control flows "twice"**, once by making the threads that pass the IF statement active and then next the threads that pass the else statement be active. this keeps us from running into any thread synchronization issues at the cost of executing twice. this is called independent thread scheduling.

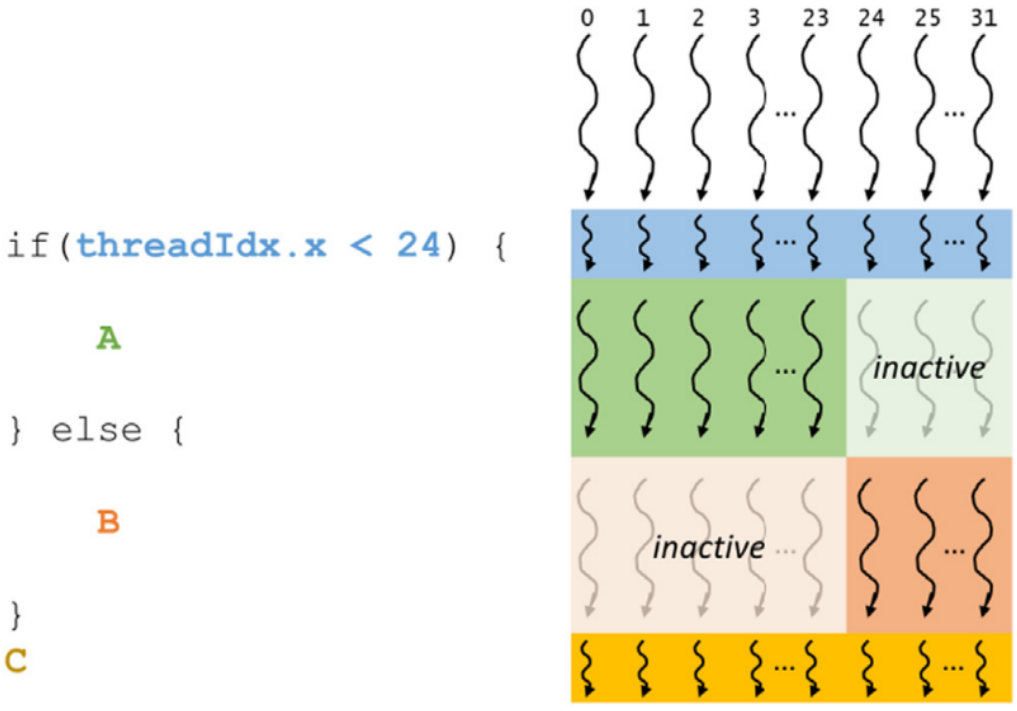


FIGURE 4.9

Example of a warp diverging at an if-else statement.

After the volta architecture's introduction (V100), NVIDIA introduced flexible scheduling for independent threads. This allows for concurrent execution of passes instead of doing it sequentially like before. this decreases the penalty from control flow divergence.

Thread divergence without IF/ELSE

There can be thread divergence without if/else too, if the calculation depends on the value of the the thread index in some way. This can lead to some threads finishing faster than others

Example:

```
__global__ void loopDivergenceKernel(int *data, int size) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int count = 0;

    while (count < size && data[tid] < tid) {
        data[tid]++;
        count++;
    }
}
```

Here we can see that we're directly comparing against the threadid, the tid. This results in some threads finishing faster and some going slower cause threadids are all different.

Understanding control divergence is important and we should minimize it : the performance impact of control divergence can also **decrease** as we grow our data when the number of warps that diverge (like if the warps are unused) are fixed. This means if there's initially a 1 in 4 warp divergence for every 100 warps, when you scale that to 1000, only to find that there's now a 1 in 32 thread divergence. In higher dimensional CUDA kernels, the base control divergence is naturally higher.

Warp Scheduling

In a computationally intensive task, when an SM is assigned warps, the number of threads generally assigned is more than the thread capacity of an SM. This means we need some kind of tracking/scheduling system. Also when some warps are performing a long-latency operation (like fetching from global memory), it is important that they do not block other warps which are read-to-go and are not performing long latency operations.

This basically "hides" the long latency of other warps. This tolerance for warp latency is basically why GPUs do not need as advanced branch prediction like CPUs do. Most of the GPU chip is dedicated to floating point arithmetic.

This warp scheduling or selecting the ready-to-go warps **does not** waste any time, it has zero overhead. This is why it's called zero-overhead warp scheduling! This is done by storing all the execution states of warps in the hardware registers instead of other memory.

Latency tolerence can increase the overall performance by having each SM take on more warps than their warp limit. An A100's SM can only support 64 threads or two warps at a given time.

Occupancy

Let's review what we've learnt. An SM can take on more threads than it can run at once. This is beneficial and important because SMs can perform latency hiding, where warps that are not ready give way to warps that are ready so that no warp has to wait in line while the warp being processed waits for its long latency task like memory fetching to complete.

However, there's a hardware limit on the number of threads that SMs can take on. This is the maximum number of thread slots that the SM has, the number of threads that can be allotted to a block + the number of blocks that can be assigned to an SM, and the size of the threads on the shared memory and the registers. Registers store variables and intermediate results while shared memory stores the data shared by all the threads on the SM.

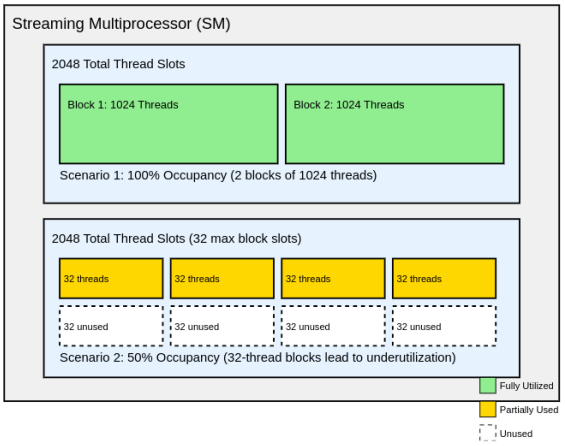
Because the GPU has to juggle all these factors, it often cannot allot a the maximum possible number of threads. This needs to non-optimal performance and the need for a metric - occupancy : the ratio between the number of threads subscribed to the GPU and the number of maximum threads that can be subscribed to the GPU.

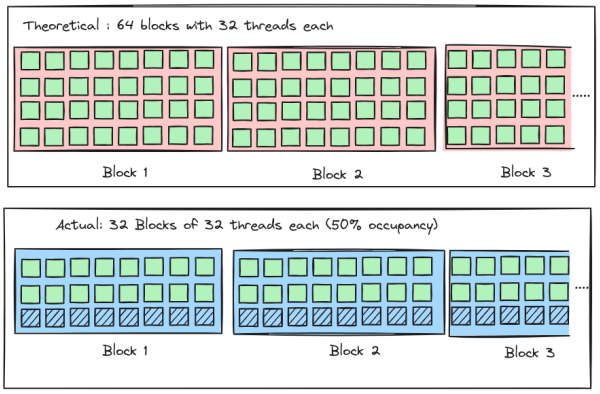
SMs have dynamic thread slot allocation for each block. This is like fitting a piece to your puzzle, except the place where the piece goes will optimize itself for you. Here, the piece being the block (and the threads) and the puzzle slot being the block and thread slots. I understand this with an example: suppose a grid is launched with the maximum limit for threads in an SM : 2048. These 2048 threads are then divided into two blocks of 1024

threads each. The occupancy is 100% in this scenario. If we launch the kernel with a grid size of 512 threads, the available thread slots are divided into 4 blocks. It is not necessary for all the thread slots to be filled, but it is important for the block size to be optimally fit.

Dynamic allocation is important because fixed partitioning it will fail to support blocks that require more threads than the fixed partition allocates. Fixed partition can also over-allocate threads which means a waste of compute. However, dynamic partitioning leads to some underutilization and unoptimized routes.

Suppose in a Volta V100 SM, we have 2048 total thread slots, and 32 total block slots. We then try to assign a block that has a size of 32 threads. If we call a grid with 2048 threads, this means we will theoretically need 64 block slots (each with 32 thread slots) to reach max capacity. However the V100 has only 32 max block slots with 64 threads each. This results in the data being partitioned such that each block is only half full, and leads to a waste of 1024 threads. In this case, the occupancy is 50%!





Another simple way in which occupancy is impacted is when the maximum number of threads per block is not divisible by the assigned block size. Which means that there simply will be threads wasted. This is why we must keep threads to be multiples of 32 or 256 or basically the max block size to be divisible by our assigned block size.

Lastly, the size of our threads on the register matters a lot. If our thread needs more registers per thread, then this means the compiler has to automatically reduce the number of blocks so that there's enough number of registers available to each thread.

We notice that a lot of these performance problems in parallelization are caused by the discrete nature of blocks and warps and dynamic partition. If a block has the be removed, it has to be done for the entire block at once. This affects all threads in the block.

Exercise

Question 1

```
__global__ void foo_kernel(int* a, int* b) {
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (threadIdx.x < 40 || threadIdx.x >= 104) {
        b[i] = a[i] + 1;
    }
    if (i % 2 == 0) {
        a[i] = b[i] * 2;
    }
    for (unsigned int j = 0; j < 5 - (i % 3); ++j) {
        b[i] += j;
    }
```

```
}

void foo(int* a_d, int* b_d) {
    unsigned int N = 1024;
    foo_kernel <<< (N + 128 - 1) / 128, 128 >>> (a_d, b_d);
}
```

a. What is the number of warps per block?

The number of warps per block is calculated by dividing the number of threads per block by the number of threads in a warp = $128/32 = 4$.

b. What is the number of warps in the grid?

The number of warps in the grid is calculated by the number of blocks in the grid multiplied by the number of warps in a block:

$$N = 1024, \frac{(N + 128 - 1) * 4}{128} = 9 * 4 = 36$$

c. For the statement on line 04:
i. How many warps in the grid are active?

The first condition is the divergent condition:

```
if (threadIdx.x < 40 || threadIdx.x >= 104)
```

This means the warps that are outside this will not be divergent, and hence active at the first pass:
Hence warp 0 (thread 0 to 31) and warp 1 will be active (8 threads), warp 2 will be inactive, and then warp 3 is active but only 24 threads will be active.
This means 3 warps are active per block, and there are 9 blocks total

This means 9 * 3 = 27

ii. How many warps in the grid are divergent?
Total warps in the grid that are divergent are the ones which are not active, which means 36-27 = 9 warps are divergent

iii. What is the SIMD efficiency (in %) of warp 0 of block 0?
That will be $32/32 * 100\% = 100\%$

iv. What is the SIMD efficiency (in %) of warp 1 of block 0?
That will be 8 active threads for 32 total threads in a warp = 25%

v. What is the SIMD efficiency (in %) of warp 3 of block 0?
That will be 24/32 = 75% SIMD efficiency

d. For the statement on line 07:

```
    if (i % 2 == 0) {
        a[i] = b[i] * 2;
    }
```

How many warps in the grid are active?
We can see that further the active threads are divided, so about 16 of them will be active for warp 1, 4 of them for warp 2, and 12 of them for warp 3
So all the three warps are active but the threads inside them are divergent

How many warps in the grid are divergent?
All the

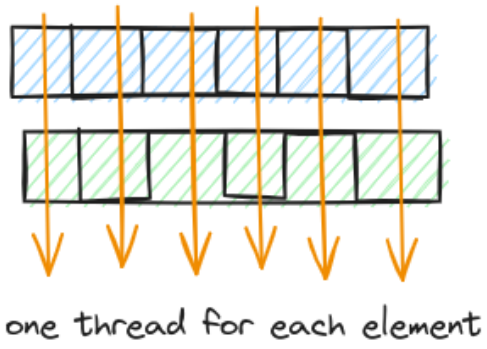
What is the SIMD efficiency (in %) of warp 0 of block 0?
50% threads are active than before, so that means 50% of all the SIMD efficiency before, so that means 50% of 100% is 50%

Question 2.
For a vector addition, assume that the vector length is 2000, each thread calculates one output element, and the thread block size is 512 threads. How many threads will be in the grid?

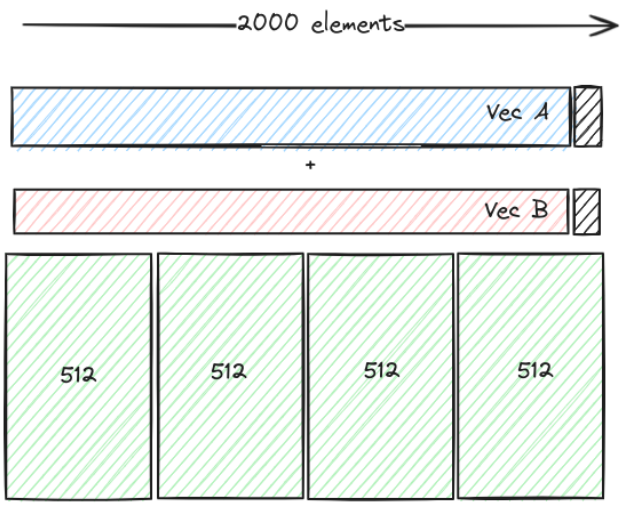
Answer :

Let's review our situation : I have 2000 elements and each thread corresponds to 1 single additive operation.

That means the number of threads should essentially be the number of elements being summed, however that will not be the case because the block size is discrete, and 2000 is not divisible by 512.

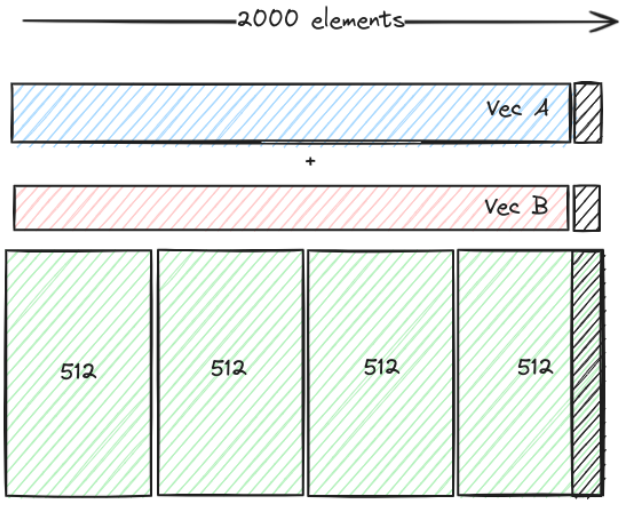


That means we need the number of blocks that is good enough to fit more than 2000 blocks, which will be $512 \times N > 2000$, hence N comes out to be 4 blocks



Question 3
For the previous question, how many warps do you expect to have divergence due to the boundary check on vector length?

The boundary check is when we check whether the thread iterator `i` in the condition `if(i<n)` is true or not. So all the threads that are beyond n, that is they satisfy `i>=n` instead, will be out of bounds and not execute. This divergence will only occur in the last block because, as we can see from the diagram below, the first three blocks are completely occupied. But some threads in the last block are divergent (marked in black lines)



To calculate the warps that do not execute, we must calculate the number of warps active in the last block, then subtract it from the total number of warps in the last block. In order for that, we must remember that even if some threads are out of bounds, as long as there are some executing threads, the warp is considered active.

Number of warps in last block = $512/32 = 16$
Number of completely active warps in the last block = $floor(\frac{2000-512\times3}{32}) = 14$
Number of divergent warps = $15 - 14 = 2$

Question 4
Consider a hypothetical block with 8 threads executing a section of code before reaching a barrier. The threads require the following amount of time (in microseconds) to execute the sections: 2.0, 2.3, 3.0, 2.8, 2.4, 1.9, 2.6, and 2.9; they spend the rest of their time waiting for the barrier. What percentage of the threads' total execution time is spent waiting for the barrier?

What we need to calculate here is the percent of the total waiting time per total execution time. The total waiting time can be given by the summation of the difference of the thread which took the longest time to execute minus the execution time of all the threads. The thread that takes the maximum time is the one which takes 3.0ms.

$$\text{Total waiting time} = 3.0 * 8 - (2.0 + 2.3 + \cdots + 2.9) = 24.0 - 19.9 = 4.1ms$$

Now total percent of waiting that is spent waiting for the barrier is : 4.1/17.1 * 100 = 20.06%

Question 5
A CUDA programmer says that if they launch a kernel with only 32 threads in each block, they can leave out the `__syncthreads()` instruction wherever barrier synchronization is needed. Do you think this is a good idea? Explain.

The 32 threads form a single warp. All the threads in a single warp are executed in lockstep so there should be no divergence. Hence it is safe to execute it without `__syncthreads()`.

Question 6
If a CUDA device's SM can take up to 1536 threads and up to 4 thread blocks, which of the following block configurations would result in the most number of threads in the SM?

a. 128 threads per block
b. 256 threads per block
c. 512 threads per block
d. 1024 threads per block

C is the answer cause 512 * 3 = 1536

Question 7
Assume a device that allows up to 64 blocks per SM and 2048 threads per SM. Indicate which of the following assignments per SM are possible. In the cases in which it is possible, indicate the occupancy level.

a. 8 blocks with 128 threads each
b. 16 blocks with 64 threads each
c. 32 blocks with 32 threads each
d. 64 blocks with 32 threads each
e. 32 blocks with 64 threads each

We will ignore the register dependency for this question.
a. 8 blocks with 128 threads each :

This is possible because $8 \times 128 = 1024$ which is less than 2048. To calculate the occupancy we need to calculate the number of warps, which will be the number of threads divided by the warp size (32) = $\frac{1024}{32} = 32$ warps.

Occupancy is the percentage of active warps in an SM divided by the total warp capacity of an SM. Hence occupancy for this case will be:

$$Occupancy = \frac{1024}{2048} \times 100\% = 50\%$$

b. 16 blocks with 64 threads each

Here, the total number of threads will be $64 \times 16 = 1024$, hence it is possible to accommodate the number of threads. Hence the number of warps is still the same as the last case, 32. Hence the occupancy will remain the same, 50%.

c. 32 blocks with 32 threads each

This will also be the same as last case, $32 \times 32 = 1024$, which is 32 warps and hence 50% occupancy.

d. 64 blocks with 32 threads each

In this case we will have 2048 threads but more the number of blocks we will have is more than the number of block slots in a V100, which is 32. Hence we will end up partitioning the given blocks into 32 blocks with 32 threads each and scheduling the rest afterwards. This means we a total of 1024 active threads, and hence 32warps once again. So the occupancy remains 50%.

e. 32 blocks with 64 threads each

The total number of theoretical threads in this configuration will be $32 \times 64 = 2048$. Since the number of blocks is equal to the maximum block slots, this configuration should fit on the V100. The number of active warps is 64, hence we get a warp occupancy of :

$$Occupancy = \frac{64}{64} \times 100\% = 100\%$$

Question 8

Consider a GPU with the following hardware limits: 2048 threads per SM, 32 blocks per SM, and 64K (65,536) registers per SM. For each of the following
kernel characteristics, specify whether the kernel can achieve full occupancy.

If not, specify the limiting factor.
a. The kernel uses 128 threads per block and 30 registers per thread.
b. The kernel uses 32 threads per block and 29 registers per thread.
c. The kernel uses 256 threads per block and 34 registers per thread.

a. The kernel uses 128 threads per block and 30 registers per thread.

The total number of blocks is $\frac{2048}{128} = 16$ blocks. We observe two things : the block size completely divides the total number of threads and the total number of blocks is less than the max number of block slots in an SM.

Let us now calculate the number of registers that we will be using, which is equal to the number of threads $\times$ the number of registers per thread, and will come out to be $2048 \times 30 = 61440$, which is less than 64k registers. Hence we will not be limited by registers.

The total occupancy should be the percentage of the number of active warps divided by the total number of warps in the SM, which will be $\frac{64}{64} \times 100\% = 100\%$.

b. The kernel uses 32 threads per block and 29 registers per thread.

If the kernel uses 32 threads per block, we will have $2048/32 = 64$ blocks. The number of theoretical blocks is bigger than the number of blocks in an SM. Hence we will be restricted to 32 blocks with 32 threads, that is 1024 threads.

If we have 1024 threads and 29 registers per thread, we need 29696 registers in total, which is less than 64k. Hence we are not limited by registers and our occupancy will be $32/64 \times 100\% = 50\%$.

c. The kernel uses 256 threads per block and 34 registers per thread.

If the kernel uses 256 threads per block, the number of theoretical blocks will be $2048/256 = 8$ blocks. Since the number of threads/block is divisible by total number of threads in an SM and the number of blocks is less than the total block slots, we face no limitations there.

Let's now check for the number of total number of registers used, which will be $2048\ threads \times\ 34\ registers/thread = 69632$, which is greater than the register limit provided (65,536 registers). Hence we need to remove the number of warps that will not fit. We can calculate the number of warps that will fit by calculating the registers per warp and dividing the number of total registers in an SM by that number.

Number of registers per warp will be $32 \times 34 = 1088$. The total number warps that will fit in an SM will be:

$$Total\ warps = floor(\frac{number\ of\ total\ registers}{number\ of\ registers\ per\ warp}) = 65536/1088 = 60$$

Hence we can now calculate the occupancy:

$$Occupancy = \frac{number\ of\ active\ warps}{warp\ limit\ of\ SM} \times 100\% = \frac{60}{64} \times 100\% = 93.7\%$$

Question 9
A student mentions that they were able to multiply two 1024 $\times$ 1024 matrices using a matrix multiplication kernel with 32 $\times$ 32 thread

blocks. The student is using a CUDA device that allows up to 512 threads per block and up to 8 blocks per SM. The student further mentions that each thread in a thread block calculates one element of the result matrix. What would be your reaction and why?

There's a few issues with this implementation, first being that the block size is 32×32 which means 1024 threads per block, which is greater than the 512 block limit. Secondly, if the matrix is 1024×1024 and each thread corresponds to a single element of the output matrix (which will be 1024×1024), it would mean that the number of blocks required will be $(1024/32)^2 = 1024\ blocks$. That is way greater than 8. Hence the student's claim is unjustified and probably incorrect.