High Performance Computing for Scientists and Engineers
Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module – 04
GPU Computing
Lecture – 36
Thread execution in CUDA program - scheduling and memory access

Hello, welcome to the class of High-Performance Computing for Scientists and Engineers and we are discussing the module on GPU Computing. Today's lecture will be on the topic thread execution in CUDA program; scheduling and memory access. So far, we have looked into some fundamental aspects of GPU architecture and some important aspects of CUDA programming.

What we have understood now is that GPU's are devices with large numbers of computing cores, but with small memory especially, there are some issues with memory management in the GPU's, because these memories are not exactly as similar as the CPU memories.

Therefore, if we want to execute some job in CPU, a couple of things, we have to consider; that there are a large number of computing cores in GPU. If we can use them efficiently, we can get very good speed up compared to the sequential core; however, when looking into this large number of GPU cores, we understand that as there will be parallel execution, the overheads and latency issues can be substantially high.

So, how can we scale our job so that we can get good performance is an issue. Fortunately, we really do not need to write the program for scheduling the tasks for getting right scalability in GPU cores or especially in CUDA cores. CUDA itself does some scheduling of the threads so that good performance is obtained; however, it is important that we understand how this scheduling is done, because there are a large number of cores and there are thousands of cores.

We understand that now what we are running in a GPU is not running in 8, 16 or 32 processors, but it is running on a very large number of cores. Our idea on parallel computing tells us that as the number of cores will increase, there will be certain issues, latencies, overheads etcetera might come into picture. So, we have to be quite careful about writing while writing a GPU program and looking into the performance of the GPU program; that we are getting right scalability, because a large number of cores in a sense gives us great flexibility.

We can use as many cores and we can get very high speed up; however, large numbers of cores also pose the risk that there can be high overheads. So, we have to look into that. The next part is that memory access in GPU is not as simple as CPU's. So, you also have to understand when a thread operates how it accesses the memory. If we build some basic understanding on that, we will not indulge ourselves in doing some mistakes which can kill the performance.

Again, I should assert that when we talk about GPU computing we are using API's like CUDA, CUDA itself takes care of a lot of the facts. For MPI or for open MP, a programmer has to think more in terms of resource allocation in terms of load balancing etcetera, but in case of CUDA, CUDA itself does a lot of the things. However, a programmer must be aware how CUDA is doing that, so that he can be sure of the fact that he is not doing something wrong while writing the program and the performance is not degrading. So, we look into these aspects.

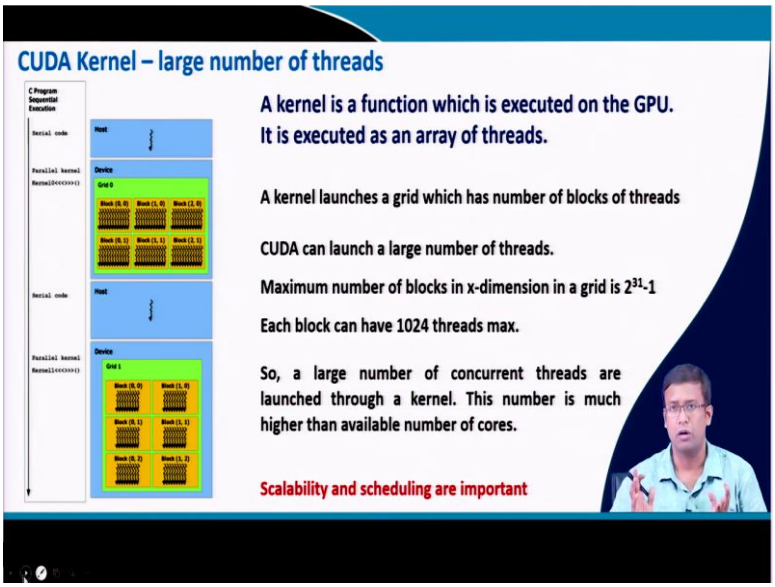
(Refer Slide Time: 04:11)

So, what we look at in this lecture and the subsequent lecture is how the threads are executed in a CUDA program that also involves scheduling of the threads which is very important, because as I said there are thousands of cores. So, more than thousands of threads are there and how these threads will go to different cores and which manner they are executed, what is the pattern of their execution, how the latency is hidden, while executing this large number of threads so that good scalability is obtained.

There is something called thread divergence which is very important for a programmer to understand. We will discuss thread divergence. It is really a performance killer. If we write a

code which is thread divergent sometime, it can go into a deadlock situation also or sometime it can give us extremely poor performance also. We will see about memory access during the kernel execution or thread executions.

(Refer Slide Time: 05:09)



So, if we look into a CUDA kernel, we can see in a general CUDA program there is a serial code which is run on the host or on the CPU then, it launches the kernel and kernel is a function which runs in the device or in the GPU and when the kernel function is called it launches a grid; a grid is collection of blocks of threads.

So, when the grid is launched many blocks of threads are launched and each block has a number of threads. All these threads are operated after that the kernel execution part is over, then again control goes to the CPU, it executes the CPU code. I mean that there is an asynchronous way of executions. As soon as the kernels are launched, the CPU can take over and start working on its own.

Then again, when the parallel part comes, another kernel is launched. The kernel is the function which is executed in GPU; how does it execute? It is executed in terms of an array of threads; blocks of threads are launched in terms of a grid. When a kernel is executed, it launches a grid of threads. A grid is a collection of blocks, a block is a collection of threads. CUDA can launch a large number of threads.

We can see in any practical CUDA program, there is granularity in terms of threads and we can launch hundreds, thousands or ten-thousands of threads. A great number of threads can be launched through the kernel. So, a grid can have blocks in all 3-dimensions a block can have threads in all 3-dimensions. Maximum number of blocks in the x-dimension of a grid is $2^{31} - 1$, which is a large number again something like the order of 65000 blocks can be launched in y and z-dimension.

So, there can be that many blocks in a kernel. When a kernel is launched, it can call that many numbers of blocks. Each block can have a large number of threads. So, inside each block there can be maximum 1024 threads. These are in the modern GPUs like P 100 ,V 100 tesla series GPUs. So, in total we can launch and $2^{31} * 60000 * 60000 * 1024$; that is the maximum number of threads that we can launch.

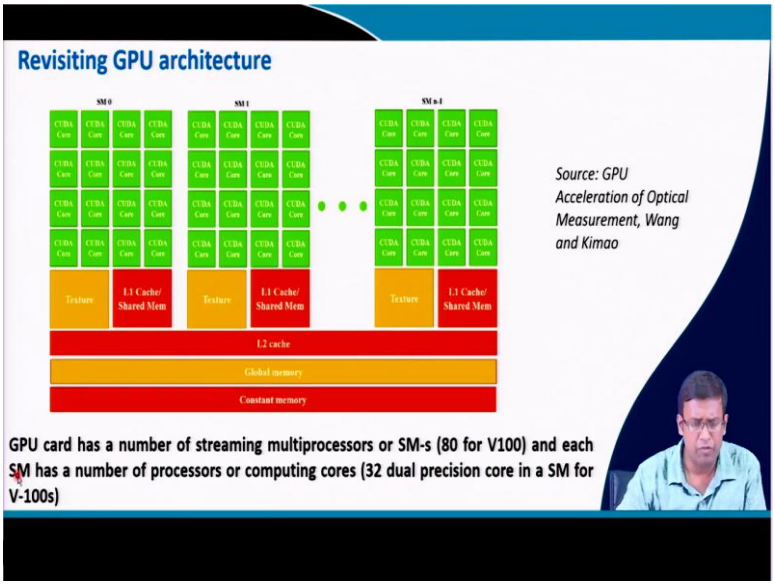
So, a large number of threads can actually be launched when a kernel is operated; however, we do not have that many cores. We have a very large number of cores, there are 2000 ,3000 number of cores available in a GPU. But the number of threads that we are launching are mostly much much more than the number of cores available. Cores are the units of hardware units of execution.

Each core is like a small arithmetic logical unit and processes the information and crunches the number. There are a number of cores; there are thousands of cores in a GPU. However, thread will go to a core, but the total number of threads that can be launched can be much much more than that. Therefore, it becomes important how we assign threads to the cores, where we have a large number of cores.So, it is a massively parallel system, but we have many many more threads. So, the program has more granularity and a greater number of threads or concurrent instruction streams can be devised. So, when we launch the kernel of course, more than one threads are launched per core. There are a large number of cores, but many more threads. So, each core also gets a large number of threads to operate.

So, how do we schedule the threads; it is important, because a large number of threads as well as a large number of cores can give us all types of problems like latency, unbalanced load, race conditions , cache coherence etcetera. So, how to schedule these threads rightly into the hardware resource or the cores, so that we get good scalability that is the most important part in thread execution of a CUDA program and we will look into that.This scheduling is not

something which we are doing. Some of the parameters we can control as a programmer, but mostly done by the hardware and the CUDA API hardware and compiler.

(Refer Slide Time: 10:20)



If we revise the GPU architecture, we have earlier looked into GPU architecture which is a GPU. So, it is a graphics card like an accessory attached to the motherboard of a CPU and it has an interconnect bus which connects through a PCI bus, a GPU is connected to the CPU.

Inside GPU, we can see the global memory at GPU's device RAM; a constant memory we will discuss about constant memory. A space for the L2 cache modern GPUs has an L2 cache then, something called texture memory L 1 cache or shared memory. We will again discuss it.

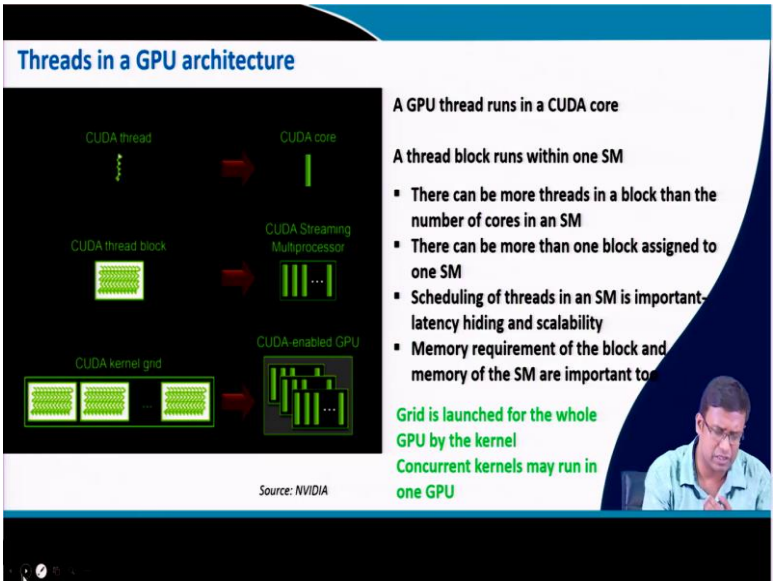
Then, which is connected with the main global memory is many streaming multiprocessors. There are in the V 100 GPU, there are 80 streaming multiprocessors. Inside each streaming multiprocessor, there are streaming processors or CUDA cores. So, when a thread is executed, a thread is basically executed on a CUDA code.

So, CUDA cores or the streaming processors are part of the streaming multiprocessors. GPU has a number of streaming multiprocessors, there are 80 streaming multiprocessors for V 100,and each streaming multiprocessor has a number of processors or computing cores or CUDA cores.

So, in V 100 we can see that 32 dual precision cores are in a streaming multiprocessor. So, the total number of cores available are 32 into 80 and each streaming multiprocessor has 32 cores

dual. We are talking about dual precision cores, because for most of the scientific engineering purposes dual precision cores are the important one.

(Refer Slide Time: 12:09)



Now, if we see how these resources are shared, because there are a large number of resources as well as a large number of tasks are there among the threads. When a thread operates through this thread runs in a CUDA core. So, we can launch a number of threads when we launch the kernel. Each thread goes to one CUDA core. What is a thread? A thread is a member of a block.

A thread cannot be launched as its own, a thread is launched as a part of a block, and each block goes to one streaming multiprocessor. So, what I am sending as a block of thread, this block is being allotted to or assigned to one streaming multiprocessor. So, if a block has 1024 threads, these 1024 threads collectively go to one streaming multiprocessor. How many cores one streaming multiprocessor has? If we talk about V 100, we have just seen it is 80 cores.

So, these 80 cores will take care of these 1024 threads; that means, these 1024 threads will be some way scheduled to operate in these 80 cores. We will see how they will do. Now, 1024 is the max number, there can be a smaller number of threads also. Even you can have one thread per block.

Well, so a thread will go to a core the thread block will go to the streaming multiprocessor and the block also cannot be launched as its own a block is launched as part of a grid and the entire grid goes to the GPU; that means, it populates over all streaming multiprocessors. A GPU

thread runs in a CUDA core and a thread block runs within one streaming multiprocessor. There can be more threads in a block than the number of cores in a streaming multiprocessor.

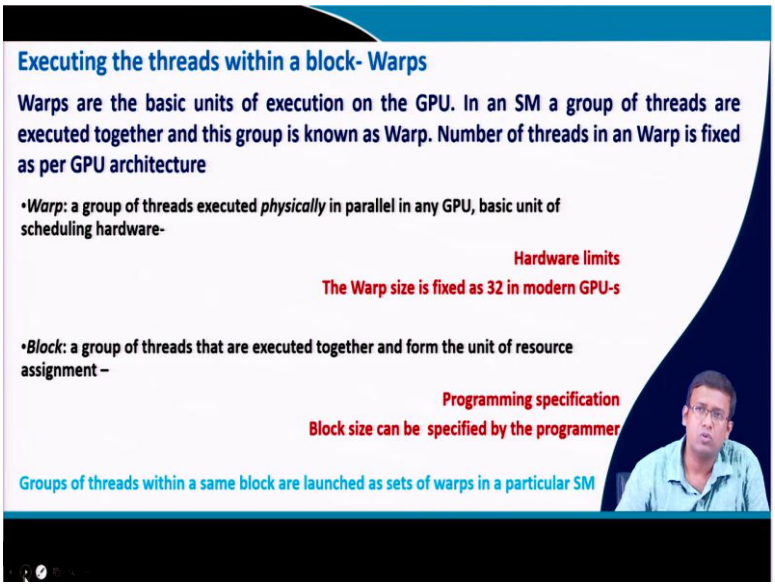
Of course, we are talking about launching more threads than the total number of cores. Therefore, more threads will be in a block. A thread goes to a block the number of threads in a block can be more than the number of cores, but also there are 80 streaming multiprocessors available, and we can launch a much larger number of blocks. We have seen that this is $2^{31} - 1$, we can launch in one direction. So, we can launch a large number of blocks.

Therefore, there can be more than one block assigned to one streaming multiprocessor. One SM can get more than one block. Then, it is important to schedule the threads as well as schedule the blocks in an SM, so that it can hide the latency, because a large number of threads will be there. When these threads are operating, they will have their own latency for memory accessing, for race conditions for some other purposes. Some of the threads will be waiting, because some of the threads are not finished. So, how these threads will be scheduled so that this latency can be hidden and we can get reasonably good scalability. Also, it is important to see, what is the memory requirement of the block? Especially if we talk about on-chip memory if we quickly think of the previous slide each streaming multiprocessor also has some shared memory which we write to some memory unit.

So, if we talk about the on-chip memory utility then, what is the memory requirement of the block? How much memory does the block require? That is also an important aspect while scheduling the blocks in the GPU, because if there are less memory requirements probably more blocks can be scheduled in together, if there are more memory requirements, less blocks can be scheduled. So, what is the memory requirement of the block and what is the total memory available in a streaming multiprocessor. In case we look for chip memory that is important. So, when we launch a grid this is launched for the whole GPU by the kernel.

Also, concurrent kernels may run in one GPU, because kernel execution is asynchronous. Once a kernel is launched, it goes to a GPU, then the control comes back to the CPU and the CPU can do its own work. So, the CPU can concurrently place kernels in the GPU. So, this is also important that different kernels can also be scheduled on the GPU at the same instant.

(Refer Slide Time: 17:03)



So, this scheduling is a very important part here. The basic unit of execution in a GPU is warp. Warp is called a group of threads which are scheduled together on a streaming multiprocessor. So, we understand that in a streaming multiprocessor a block will be allotted. All the threads of the block will go to one streaming multiprocessor. Now, say this block has 512 threads. The streaming multiprocessor has 80 cores.

So, of course, 512 threads cannot operate on their own. A group of threads will be chosen and they will be first launched into available cores in the streaming multiprocessor. Now, if we take 80 threads or we are not doing that CUDA is doing that CUDA hardware and compiler and the API's doing that. So, if all the 80 threads are chosen and these 80 threads are put into the streaming multiprocessor, what will happen? That they will do their own work they will finish their work then next 80 threads will be taken.

Most difficulty that will arise is that, instead of getting an advantage of using 512 threads, we are getting a parallelism by the factor 80 only. But, if we somehow put a smaller number of threads say, less than half of the cores are occupied.

As soon as these cores are looking for something like searching for memory, another set of threads can be launched in the free cores and some of the control will come there they will operate, they will execute data and that gives us better performance.

So, it is not that all the cores of a streaming multiprocessor are active at one instant, but a small number of cores inside the streaming multiprocessors are active and that many threads are chosen. This unit combines the number of threads which will go to a streaming multiprocessor as a unit and operate there, it is called warp. Number of threads in the warp, what is that number? This number is fixed by the GPU architecture and this number is essentially 32.

So, when a kernel is launched, a block is scheduled in a GPU streaming multiprocessor and 32 threads among the blocks are first executed in the streaming multiprocessor. Then, once they are over or once they go to latent state something like that another 32 threads are chosen, and they are executed together.

So, groups of 32 threads are combinedly taken and they go to the streaming multiprocessor and that is executed. So, the group of threads that are executed physically in parallel. So, these 32 are executed in parallel is called the basic unit of scheduling, and this is hardware dependent number, this is known as warp. When a GPU works inside its streaming multiprocessor, one warp is activated first, and this is different from block.

Block is the total group of threads that are considered as a unit and allocated to a streaming multiprocessor. Number of threads in a block can be different, the number of threads in warp is fixed by the hardware specification or architecture specification. So, this is the hardware limit; that means, we can see that not all the cores of the GPU are active at one instant this was taken care of one of the issues like latency heading another issue is the heating, electricity power etcetera the other hardware issues.

In the modern GPUs this number is 32; that when streaming when a kernel is launched a block of threads are assigned to a streaming multiprocessor and among this block 32 threads are chosen and these 32 threads as a warp is activated in a in the streaming multiprocessor. 32 cores of the streaming multiprocessor are active at one go and they are taking up these 32 threads.

Once these 32 threads are launched as a warp, then it can pick up the next warp next 32 sets of threads and put it in the free cores of the streaming multiprocessor. There can be certain overlap. We will see how these overlaps can be done. There can be some dynamic allocation of the warps inside a stream inside one particular block.

The number of threads in the block is programming specification. Programmer can himself tell me that I will take these many numbers of threads and they will go as a block. Block size can be specified as the programmer, but warp size cannot be specified as the programmer, because this is what is fixed by the architecture of the GPU.

So, whatever be the block size, 32 threads will be chosen. If you have less than 32 threads in the block then less will be chosen, but if you have more than 32 threads, 32 will be chosen and they will be active; that means, 32 cores of the streaming multiprocessors are active and they will pick up threads. In case you have less threads in the block, then few of the cores will be inactive you lose in terms of performance.

So, that is also important to see. In case you have threads, which are not in case the block size number of threads in a block is not multiple of 32. Say, you have 59 threads in a block. So, 32 will go first and the remaining 27 will go in as the next warp. So, 5 will be inactive. You will lose in terms of performance, because you are also running many blocks. The next block will come and that will also lose in terms of performance, because some of the cores will be idle; that means, when a GPU program runs inside each streaming multiprocessor the streaming multiprocessors also work parallelly.

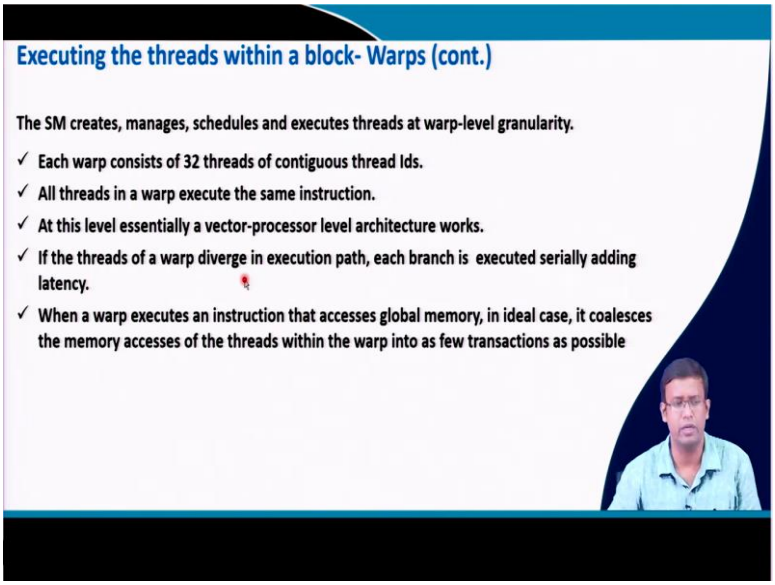
But one block goes to one streaming multiprocessor and inside one streaming multiprocessor only 32 are chosen and active at one point of time and then the next 32 will be chosen and that is how they are scheduled. As one block goes to one streaming multiprocessor, if your block number is less than the number of streaming multiprocessors, you are losing also, because some of the streaming multiprocessors are inactive. Also, if your block size is less than the warp size then you are losing, because some of the cores are inactive and these are two very important aspects. Groups of threads within the same block are launched as sets of warps in a particular streaming multiprocessor.

So, you say if two blocks are assigned to one streaming multiprocessor and each of the blocks have 16 threads. These 2 ,16 threads of 2 different blocks cannot be chosen as part of the same warp; a warp that is chosen in the warp are the threads within the same block.

So, there will be the first 16 activated and then the next 16 will come as the next warp. So, threads cannot be picked up from different blocks and mixed with another blocks thread and put into the same warp that cannot be done. Warp is always restricted to a particular block.

Well, this gives us certain ideas about scheduling in GPU's and this will be required later for understanding it better.

(Refer Slide Time: 25:33)



Now, how the warps are executed? Streaming multiprocessor creates, manages, schedules and executes threads at warp level granularity. So, at one go one warp is executed. Each warp consists of 32 threads of contiguous thread Ids. So, they cannot be chosen randomly from thread Id 0 to thread Id 31 will be 1 warp thirty 32 to 63 will be another warp so on.

All threads in a warp executes the same instruction. There cannot be different instructions for different threads. All threads will execute the same instruction following a single instruction multiple data or single instruction multiple thread model. At this level, essentially a vector processor level architecture works that it takes data from the same location with different memory elements and does the same operations over them.

If the threads of the warp diverge an execution prime branch, each branch is serially executed serially adding latency. This particular path divergence of the execution path we will see in a while we will see in detail, but this basically means that some of the threads inside a warp are scheduled to do something else the other threads are scheduled to do something else.

How will you do that? You will do you do it using some if else condition if thread Ids are this, they will do operation A other otherwise, they will do operation B. If it is done it is seen that

while operation A is been executed the other threads were not supposed to do operation B, they are sitting idle; that the, they are going into latent stage. These cores will be sitting idle.

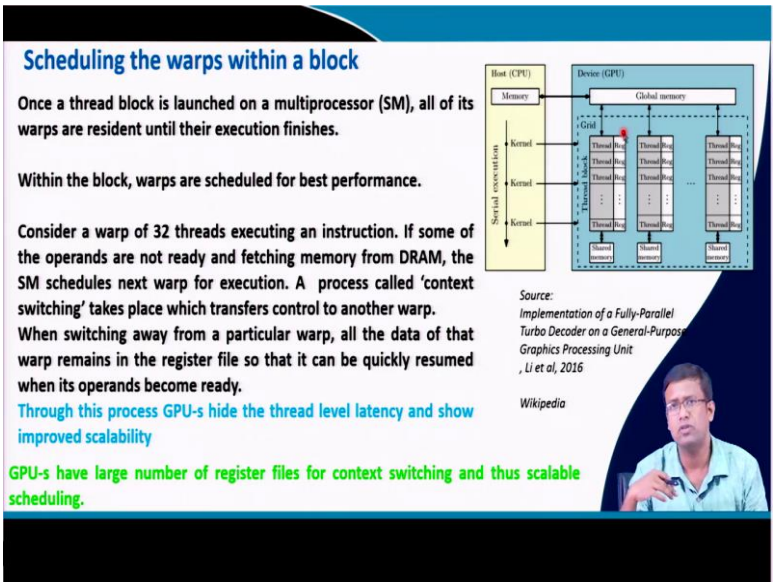
So, there will be some sequentially among different operations. If the warp is the given two different operations, they will be followed sequentially; it cannot be done parallelly. So, whatever is given in the warp that they will I mean that all the cores will do essentially the same warp. If you ask them to do multiple work using some if else statement that will be done serially and that will add to latency.

When a warp executes an instruction that accesses the global memory in the ideal case, it coalesces the memory access of the threads within the warp to make as few as transactions as possible. So, when the warp talks to the global memory, it is desired that it reads from the global memory in a coalesced manner.

A contiguous data from the global memory and written in a, because threads are threads within a warp are identified by contiguous thread Id similarly, the memory location must be contiguous and memory must be coalesced instead of 2 D arrays it should be a 1 D array as much as possible.

So, that as the threads will go, they can take a contiguous piece of memory and pull it to the register and operate accordingly. So, that is the desired thing and if you can make the memory coalesced and contiguous, you will get better performance.

(Refer Slide Time: 29:01)



Now, how this warps schedule within a block and this is extremely important. Once the thread block is launched in the multiprocessor, all of its works are resident until their execution is finished. So, once a thread block is associated with the multiprocessor, all of the warps are considered to be residing in the multiprocessor.

So, what happens to the residing warps? The threads are allotted to the multiprocessor and they are residing there. Therefore, each of these threads get one register some, one or few. Some register memories are assigned to the variables given to the threads and these register memories are on chip memories, memories which are attached to the streaming multiprocessor.

In case the threads require more memory than the register memory, it has to be decided allocated as local memory and it comes from the global memory and it becomes a slower process. But mostly we will try to ensure that that is one important part of finding out what will be the number of blocks, number of threads etcetera. That the threads get enough registers and interestingly GPU's have a very large number of registers. A streaming multiprocessor in GPU has a very large number of registers, so, whatever at the memory requirement that comes from the register memory itself.

Within the block warps are scheduled for best performance, how do these warps operate? Within a block there is not only one warp. One warp is scheduled first it goes in operation, but in later next warps will come warp has 32 core threads; that means, 32 cores will be populated by this warp. However, a streaming multiprocessor has said, V100 has 80 cores therefore, another warp can also be active. This scheduling is done for best performance, how? That is to consider a warp of 32 threads executing an instruction. If some of the operands are not ready and fetching memory from DRAM say, I have to read an array or some of the threads will read from an array which is in the main device stem. This reading is slow.

The streaming multiprocessor says this; so, what happened is the threads which are supposed to look into that memory, there are threads which are part of the warp. These threads now are not doing any processing on the GPU. These threads are looking from memory. They read memory from the device RAM; it takes some time.

What does SM do? It puts these threads into a latent stage. It hides the threads. It launches the next warp which will do some operation. By the time this memory fetching is done by the threads that time another warp is launched, and this process is called context switching; that instead of work working on this particular warp, streaming multiprocessor started working on

another warp and that is why warp size is much smaller than the number of cores in different perspectives.

One perspective is that it can do the context switching. How is this context switching done? That is the threads in these warps have their own register memory. Streaming multiprocessors have a large number of registers and other registers are associated with the next warp and then this next warp starts operating.

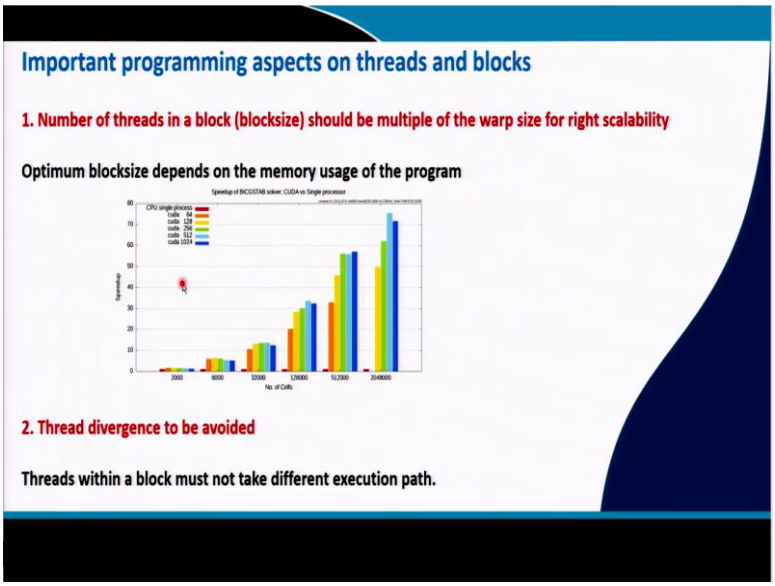
Now, once the next warp does some memory search, by that time this particular warp which was launched first has looked at its memory and loaded the registers. It has become active again and there is a switching away from the warps, but the register data always remains on the warps. Register of a particular thread, once a thread is resident, its register is active.

So, as soon as it is ready, its registers are populated from the device from memory. The other warp which threads are searching from memory goes into a, into background and switch comes into the previous one and it is activated.

In this way, a number of warps can concurrently work hiding their latency. Once one is looking for memory, the other warp will be active and it will search for the memory. Through this process the GPU's hide their thread level latency and show improved scalability. GPU's have a large number of register files for context switching and therefore, they have scalable scheduling.

So, this is the way by which warp has only 32 threads, but actually you get a parallel performance which is more than 32 threads, because some of the latency of these 32 threads in terms of memory fetching and or some other activities is hidden by activation of the next warp and this is done by using large number of register files.

(Refer Slide Time: 34:43)



So, few of the important programming aspects what we understand from the behavior of the from the its pattern of the execution of the threads; number of threads in a block or the block size should be multiple of warp size for right scalability, because it is warps which will be activated as a group of threads at one go. So, if your block size is not a multiple of warp size or less than the warp size, some of the cores will be inactive, some of the cores will be idle and you will get worse performance.

What is the optimum block size the question comes here? This is also considering how much memory utility has been done, because there is memory fetching. So, one has one warp looking for the memory.

In an ideal case for example, say there is no context switching inside a streaming multiprocessor, then one warp is good enough as the size, one warp as the block size. One warp is actively done, but once one warp will be finished, then the next warp will be taken, but now as context switching is there, so, how much memory they are searching for or if they are using on chip memory how much memory is being used that is also another part.

So, depending on that we can see that optimum block size is a multiple of warp size, but how much is this that has to be that depends on the programs and execution etcetera, but it is seen that maximum number of blocks is 1024, but for large problems we can see that 512 block size in case of a bi CG stab solver gives us best performance. So, something like 256 ,512 multiples of the warp size give us better performance.

Thread divergence has to be avoided and this is a very important issue. We look into it later the threads within a block must not take different execution path. All the threads should do the same work, there should not be branching of work among the threads, because again it is a warp which is being activated. So, all the threads in the warp will finish their job. If there are multiple jobs if the different threads in the warp are supposed to do different things, there will be certain issues. Some sequentially and it will kill the performance. We will look into thread divergence in detail and also look about memory issues in the next class.