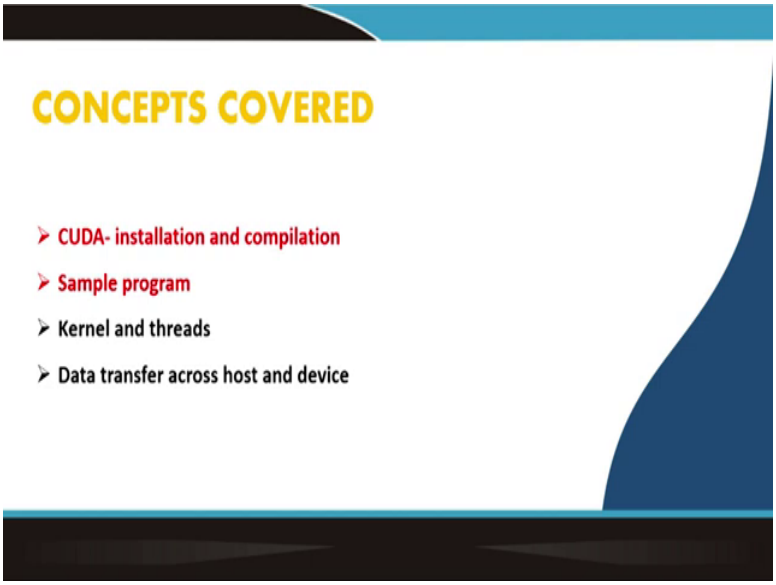
High Performance Computing for Scientists and Engineers

Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module – 04
GPU Computing
Lecture – 35
Introduction to CUDA Programming (Continued)

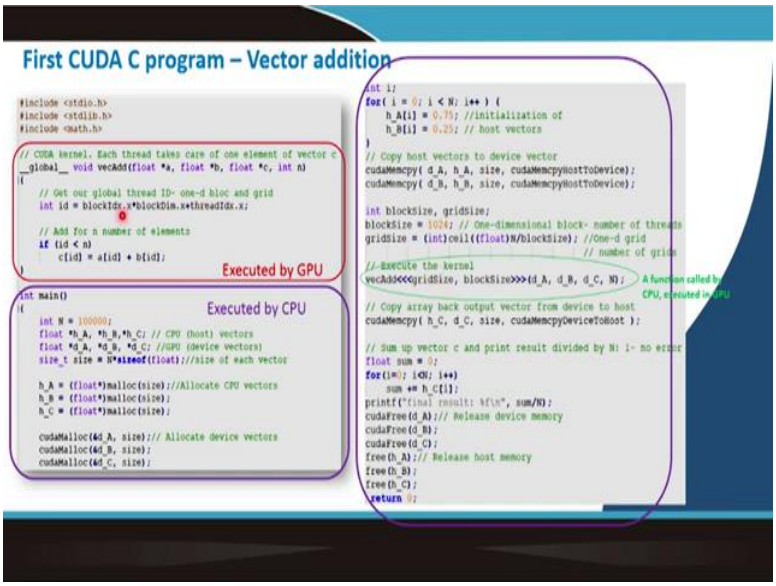
Hello everybody, we are discussing GPU computing modules in the course of High-Performance Computing for Scientists and Engineers and this is the lecture on Introduction to CUDA Programming, which we are continuing from last class.

(Refer Slide Time: 00:37)



In the last class we discussed a few aspects of CUDA programming; we first discussed how to install and compile CUDA and execute a CUDA program and then we looked into a sample program, we will continue from there.

(Refer Slide Time: 00:50)



This is the sample program which is our first CUDA program, we have looked into last class. So, this is a program for vector addition; the idea of this program is that we will take two vectors of relatively large size. So, each of these vectors have hundred thousand elements in our example, and we will write a CUDA C program to add these two vectors and get a resultant vector.

The addition will be done by CUDA kernels in a sense that each of the threads in CUDA program and will be responsible for adding each of the elements in the vector. So, in parallel, multiple threads will be launched and multiple elements will be added together.

This is quite similar to the concept of vector processing, at least this programming model which we have seen earlier when we are discussing fundamentals of parallel computing. So, if we look into the program it has, it is a single file with an extension dot cu. So, the program name, say, vecAdd . cu that is the program name.

If we look into that file it has two components; one is the portion which will be executed by CPU, it has the it is the main program and this is most likely your C code with some of the CUDA constructs inside it.

Another particular instance in this program, which writes and executes the kernel and this executes a kernel vecAdd. We write that, this is a function which will be called by the CPU

and that will be executed in the GPU. So, this kernel is a function which is called by the C program and this function will be executed by the GPU.

So, we will look into the function name and there is a specific syntax for this function we are calling; we will look into this syntax in more detail. If we look into the function name vecAdd and we will see that , the next part of the code is the vecAdd function which is CUDA kernel. It starts with an execution syntax __ global __ and then it is a function of void type. So, it cannot return anything, and this is executed by the GPU now.

So, this is the function which is called by the CPU and will be executed by the GPU. So, CUDA will launch this function in parallel in multiple GPU cores and different threads will be active then.

Well, this will typically return a void; because GPU and CPU memories are different, whatever we are executing in CPU or whatever a programmer is writing primarily, that code will primarily take information from CPU ram and will be executed in CPU. So, whatever is executed in the CPU will write to the CPU; cannot read from the GPU memories.

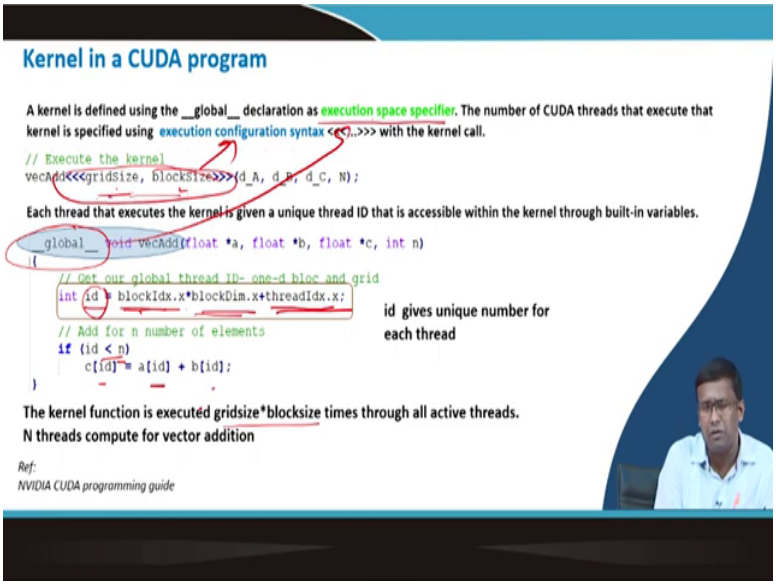
Therefore, when a function is called from the CPU to run in the GPU, it must return a void, it cannot return a memory. Memory can flow from CPU to GPU only through cudaMemcpy. So, if we look into this function int id gives the particular id of each thread. Based on the id, if the id is less than n; n is the size of the vector, based on that id elements of a and b are added and element of c is obtained.

So, once this kernel is executed on the specific number of threads and the number of threads is specified by grid size and block size; then that will write to the vector c which is passed as d _ C that is in that CUDAs in the GPUs memory d _ C.

Then d _ C or device C will be copied to host C or CPU C and then only we can read it from the C program that we are executing through the CPU. So, the first part will be the vectors which we are allocating in the CPU, assigning in the CPU initialization of the vectors; these vectors will be copied to the GPU, then function will be called by CPU which will be executed in the GPUs in multiple threads. Then the function will return a void, but it will make certain changes in the GPU memory which is device _ C, d and _ C that lies in the GPU memory. And then data will be copied from GPU memory to CPU memory and then if we want to do some processing with that data, we can do that.

We have discussed this program in the last class also. Now, we can identify that there are two important instances in this program; one is that kernel calling that this function is called as a kernel which will be executed, which will be executed by the GPU and called by CPU. Then this kernel function itself is executed on GPU not in CPU. We can identify that there is a specific syntax for kernel calling and there is a specific syntax for identifying the kernel, we will look into these things. Also, a kernel takes some of the arguments which are memory locations in the GPU and some arguments are given within a brace, which gives the number of blocks inside a grid and number of threads inside a block. So, in total how many threads will be launched that is also specified here.

(Refer Slide Time: 06:36)



So, a kernel is defined using _ global _ declaration and this is called an execution space specified. It specifies what is the execution space, this declaration specifies as what is the space in which this function will be executed; will it be executed in GPU, will it be executed in CPU, etcetera. This particular statement specifies that execution. The number of threads that we execute the kernel that is specified using execution configuration syntax, and what is execution configuration syntax; which is present inside the brace with the kernel call.

So, if we look into the program when we call the function vecAdd and within grid size, block size and pass the variables; then this becomes the execution configuration syntax.

So, this becomes the execution configuration syntax which is; so, it specifies how many threads will be running there. Each thread that executes the kernel will get a unique thread id which is

accessible in the kernel through built in variables. So, when we call the kernel, it is identified by the execution space specifiers.

Execution configuration syntax is specified with the kernel call; execution space special specifier is specified before we write the kernel function at the beginning of the kernel function. Once we look into the kernel function, it knows what is the size, number of threads in a block; and what is the number of blocks in a grid.

So, what is the total number of threads it knows; based on that it finds the global thread id and each kernel has its own global thread id, which is obtained by which where the id of the block, id of the thread and dimension of the block. Dimension is what is the total number of threads in the block. Then based on the particular thread id a thread operates on a certain part of the memory location of the c vector . So, we need to look into detail what is the execution space specifier, and this is our execution space specifier _ global _ which is present in the kernel function, and what is execution configuration syntax, which is called which is written during the kernel call.

When the main program calls kernel, the execution configuration syntax is given that these two things have to be looked into. And everything we are discussing here is taken from NVIDIA CUDA programming guide; because CUDA is developed by NVIDIA, so whenever we look into details of CUDA details of syntax and semantics of CUDA, we have to look into NVIDIAs CUDA programming guide.

This id gives a unique number for each thread. So, id of the thread which is executed by one particular GPU code during the kernel execution is obtained by the built-in functions of the CUDA kernel and based on blockId, local threadId and the dimension of the block. I will look into how to find out the kernel id’s in more detail when you look into thread algebra.

The kernel function is executed into grid size *block size time; because grid size is the number of blocks inside the grid, block size is the number of threads inside a block. So, how many threads are launched? Each block launch block size number of threads and grid size number of blocks are launched. So, total grid size* block size number of threads are launched and each thread calls a kernel.

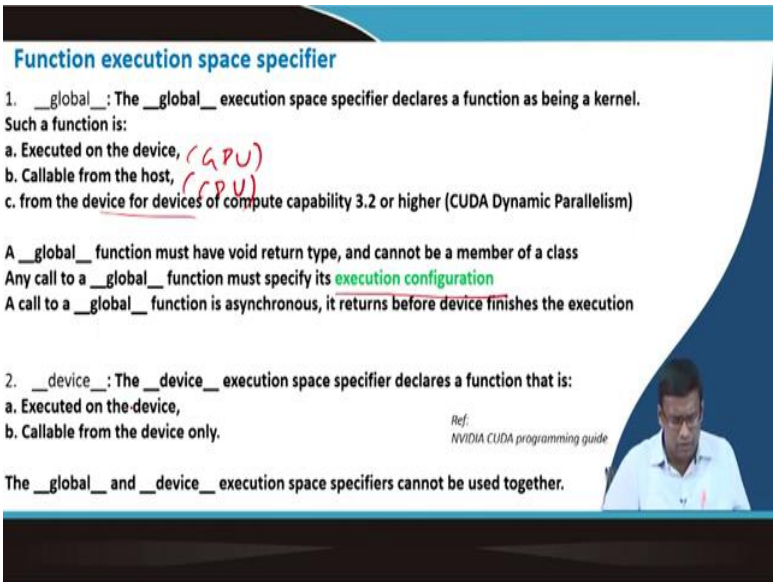
So, kernel function is executed in grid size * block size times. However, the given vector has a dimension n. So, n can be less than this number; because we cannot call irregular patterns of threads, each block should have the same number of threads.

So, if you have 1000 threads and 50 blocks, there will be 55000 threads which will be launched; say now in case you have 49990-dimension vectors. So, there will be 10 threads that will basically do nothing.

So, you have to identify that if id is less than n; then only the threads will operate, otherwise those threads will not operate. So, N threads will be there which will do computation for the vector addition; there will be at least one block in which some of the threads are inactive. In case the number of blocks block into number of threads is not same as the, is greater than the size of the vector.

So, there will be some threads which will be inactive in that part, but the end of these threads will compute further vector addition and n different locations.

(Refer Slide Time: 12:01)



So, we will see what is a function execution space specifier, that is _ global _ in our previous context? So, the global execution space specifier declares that a function is a kernel.

What is global? That this function is executed on the device and device means GPU; this function is callable from the host. So, a function which is called from the CPU and executes in

the GPU that is identified by a function which is executing in the global space. So, it is called from CPU; it is called from CPU, it is executed in GPU.

So, it spans over the entire space combining CPU and GPU; therefore, it has a global execution space specifier. Also, in some cases when you do dynamic parallelism, that means CUDA threads are further calling the next set of CUDA threads; one GPU is calling another GPU or one GPU is asking some of the jobs to be executed again in the GPU as a function.

Then, we can use this for compute capability greater than 3.2, we can use device to device call also. So, from one device another device is asked to execute a function that also can have a global execution space specifier.

A global function must have a void return type and we have discussed that; because we cannot write data in the CPU memory directly from the GPU memory, it always has to be copied back. So, it cannot return any value, it cannot return anything which will be a value and return to the CPU memory. So, it will always return a void return type. Global function, any call to global function must specify its execution configuration.

If you remember in the last slide, we discussed execution configuration, execution configuration tells what is the grid size and what is the block size. How many blocks and how many threads per block will be there, that has to be specified once a global function is called.

So, global function is a function which will run on a GPU called from a CPU or in case of dynamic parallelization, call from a GPU and run on a GPU. But as the global function will run on a GPU, it will run in a single instruction multiple thread architecture; therefore, the total number of threads are to be specified, which can be specified by specifying number of blocks in the grid that is grid size and number of threads per block that is block size.

So, that has to be specified through execution configuration, when the kernel function is called. A call to global function is asynchronous and that is important . So, when the function is called; an asynchronous function will ensure that all the threads are executed, then only the next statement will be executed by the program.

But in case of a kernel call, it is not ensured that all the threads are executed it launches the kernels; depending on the GPU scheduler, the threads will be active and they will do the computation. However, as soon as it asks the GPU to run the program in multiple threads, the

function returns back; I mean the control instruction instance returns back to the CPU and CPU keeps on working on the next set of instructions.

So, this is an asynchronous call, it does not satisfy the criteria that all the threads are done and then only the next job will be processed. The CPU can keep on processing the next job, only once the kernel function is informed to the GPUs.

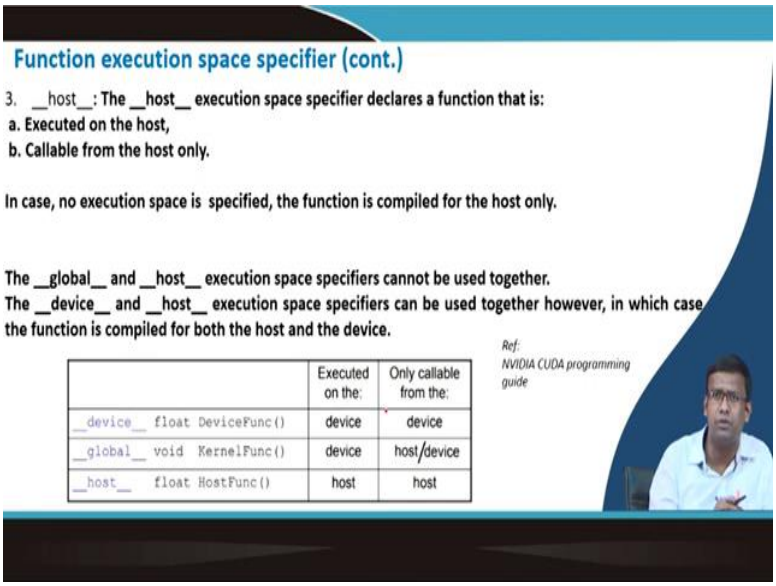
So, in case we need a synchronization, in case we need that all the threads are executed, then only the next step will be done; then we have to put some synchronization barrier, one of them is CUDA sync threads in CUDA, we will see at some point of time.

The other execution space specifier is device and this is for the function which is executed on the device and the GPU and callable from the device only. Say for example, you take a kernel function itself, within the kernel function it needs to call another function. So, it is a function which we call by the GPU and executed by the GPU, in the same stream of instructions. So, this is identified as the device execution space specifier.

So, when in the program, a function is given with device execution space specifier; it is a function which is called by the kernel and will be executed on the GPU only. When global is given, it is a function which is called by the CPU and will be executed on the GPU; therefore, it is a kernel function. The global and device execution space specifiers cannot work together; I mean a function in together cannot be a global function as well as a device function.

Everything is taken from NVIDIA CUDA programming guide.

(Refer Slide Time: 17:15)



The other specifier is host, what is host? We can understand that when a function is executed at host and call from the host only. So, CPU calls any function that will be executed on CPU; it is not a kernel as CPU c code part calls a function and this function itself works on the CPU this is called a host execution space specifier.

We can understand that by default any function is a host function; if it is a global function, we have to mention it as a kernel. If it is a device function, we have to mention it as with a device specifier. But if it is a host function, it is by default if you do not specify anything, we just call the function, it should be a host function.

So, in case no execution space is specified, by default it is compiled for the host and it is called from the host and runs in the host only. The global and host execution specifier cannot be used together; I mean the same function cannot be asked to run on both CPU and GPU called from a CPU. So, they cannot be used together.

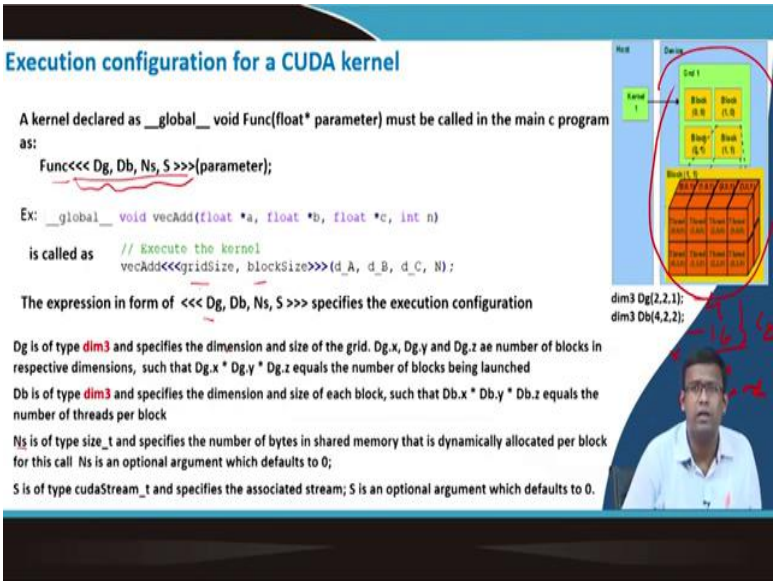
But device and host specifiers can be used together; there are certain cases when we can run a function in both CPU and GPU. So, CPU calls the function, runs in the CPU; GPU calls the function runs in the GPU, it is possible in certain cases. So, if we summarize it, the execution space specifiers give; if it is a device and it can return anything for device and host function, it will be executed on the device called from the device.

So, it is called from the GPU, it is executed in the GPU and it can GPU can write to GPU memory, GPU function can always write to GPU memory. So, it can return anything float, double, integer, void it can return anything. A host is called from host executed in the host, it also can return any value; but a global is called from a host or in certain cases called from a device under dynamic parallelism and executed on the device.

Now, GPU cannot write to CPU memory; therefore, it should always return a void, these are the three types of execution space specifier. So, whenever we look into a CUDA program, we look into the functions and see what is the execution specifier mentioned before that function.

Based on that we identify what type of function it is; is it a simple function called by the CPU program to run in the CPU, is it a kernel called by the CPU to run in the GPU or is it a function with GPU kernel calls and is executed on the GPU. Then these three types of functions are important; these three types of functions are present in the code and they are compiled separately, the compilations branch out differently.

(Refer Slide Time: 20:07)



Well, the next part is the execution configuration. If we see the kernel function; when the function is called, we have seen that the function is called syntax within a brace. So, we write the function name when a kernel function is called; function name kernel function means, the function which is declared as global when we write the function.

So, function when it is called is a written function, then this brace three <<<,three >>> signs within these certain parameters are passed and then some parameters which are passed for this function.

Now, an example is that for the vecAdd of the kernel we have looked into this example; when we call it, we write vecAdd, then grid size, block size and we pass the parameters. The parameters that will be passing are again parameters on the GPU memory only. So, d _ A, d _ B and d _ C are the variable locations in the GPU memory that we pass here.

Well, so these are the arguments inside the execution space specifier and we can see there can be four arguments here; D g, D b, N s, S. What are them? The expression in the form of D g, D b, N s, S specifies the execution configuration. D g is type dim 3, dim 3 is one particular type of variable in CUDA programming and it is basically a variable with three-dimensional non-negative integer value. It specifies what are the number of blocks or threads in each dimension of that.

So, D g is space dim 3 and specifies the dimension and size of the grid. D g x, D g y the first component, second component and third component are the number of blocks in respective dimensions. In x direction dimension what are the number of blocks; y dimension what are the number of blocks; z dimension what are the number of blocks that will be specified by dim 3 D g.

They say we can write dim 3 ,D g and these three numbers that will specify that, this is the grid size here. Then this D g is passed as grid size in the kernel call. Therefore, the total number of blocks will be in x direction and the number of blocks is D g. x , D g. y, D g. z their multiplication will be the total number of blocks.

Similarly, D b is also a similar variable, variable of dim 3 and it specifies the dimension of size and each block. So, D b .x, D b. y, D b. z will equal the total number of blocks. So, we can see that if we launch a kernel, which will launch a grid; each kernel will launch an associated grid. Inside the grid we write, for to specify the grid size we write D g 2, 2, 1; therefore, 2 blocks in x direction, 2 blocks in y direction and z direction only 1 block. So, total 4 blocks will be there and D b that is the block size we write 4, 2, 2.

So, 4 threads in each block there will be 4 threads in x, 2 threads in y and 2 threads in z dimension. So, once we declare this dim 3 D g and D b value; when we pass D g and D b, this

particular grid block arrangement will be passed to the kernel. So, how many threads will be launched by the kernel? 4 blocks will be launched by the kernel and each block will have 16 threads. So, 64 threads will be launched by the kernel.

Then there are also certain arguments N s and S. N s is of type size, it specifies the size of the shared memory variable that will be dynamically allocated per block for the call. So, you know that shared memory and L1 cache both are on chip memories and they share the same space in modern GPU architecture.

In case we need some shared memory variable; shared memory means, this shared memory is little different than what we discussed about shared memory and distributed memory architectures.

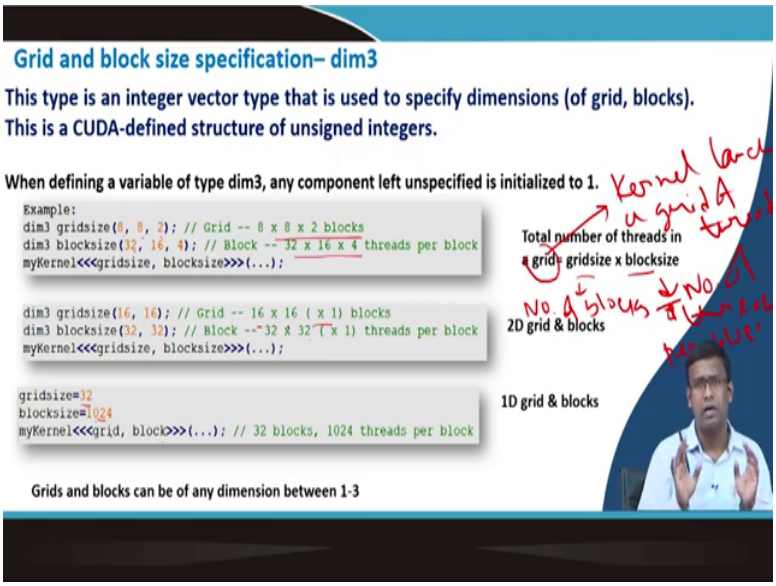
This shared memory is the on-chip memory, a memory which is shared by the cores present on a streaming multiprocessor. So, when a block will be active on a particular streaming multiprocessor, it can utilize some of the on-chip memory which will be very fast access, we will look into shared memory programming later.

So, Ns say tells what is the size of the shared memory in byte that this programmer can specify. If nothing is specified for this case, only D g and D b is specified; Ns is not specified, it takes it to be 0, there are no shared memory variables that can be declared here.

S is the associated stream, identifies the stream; this will not discuss at this stage, but in a CUDA kernel you can launch multiple streams of instruction. And how many streams of instruction that will be launched that can be specified by S and that many streams will be launched. But if we do not specify anything; S will be 0, only one stream will be active. So, if we do not specify any of these numbers, they will be assigned to this default value.

So, let us see about the dim 3 variables.

(Refer Slide Time: 26:02)



The dim 3 is an integer vector type which is used to specify dimension of grids and blocks. This is a CUDA defined structure of unsigned integers. When defining a variable of time dim 3, any component we do not specify is initialized to the value 1.

So, dim 3 typically will take 3 in non-negative integer values; say for grid size it will give us the number of blocks in grade in x, y and z direction. If we do not specify all these three values; if we only specify one value, it will take that in x direction and y and z will be initialized to one will not, it will consider that there is no other block in y and z direction.

So, if nothing is specified, it will stick to the initialized value of 1. So, we can see some of the examples; if I write dim 3 grid size 8, 8, 2; dim 3 block size 32, 16, 4; then grid size means that there are 8 grids in 8 blocks in x direction, 8 blocks in y direction and 2 blocks in z direction. So, the total number of blocks is 8 *8* 2. In each block there are 32 threads in x, 16 threads in y and 4 threads in z.

So, when we call the kernel it is launched by 8 * 8 *2 blocks and each block launches 32 *16* 4 threads. We can see another example for a 2 D grid and block; we write that dim 3 grid size 16 *16. So, each grid launches 16 into 16 blocks and in z direction nothing is specified; so, by default there is only 1 block in z direction, no other block in z direction. So, 2 D arrangement of blocks.

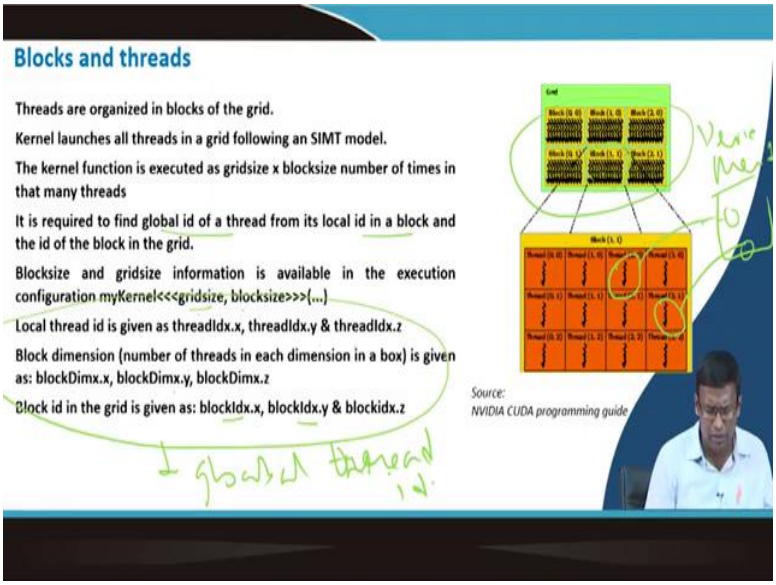
Similarly, it launches 2 D arrangements of threads per block. If only grid size is specified as only one scalar, block size is also a scalar. So, it launches 32 blocks and then block size is 1024. So, each block says 1024 threads. So, it is as simple as that. So, you have to define these two arguments grid size and block size before calling the kernel and pass them in the kernel.

The dimensions of grids and blocks can be anything between 1 to 3; we can see there can be 1 degree, there can be 2 degrees, there can be 3 degrees. Similarly threads per block also can be arranged can be 1 D, 2 D and 3 D. The total number of threads launched in a grid will be grid size * block size. Number of blocks in a grid, this is the grid size is the number of blocks and this is number of threads per block.

What is the grid? Kernel launches a grid of thread. So, the total number of threads constitute a grid and this is arranged in blocks; a grid can be decomposed in multiple blocks and then these blocks can be multiplied and decomposed in multiple threads.

Total number of grids is grid size and block size and number of threads is grid size and block size, that is the number of threads inside a grid or number of threads when a kernel is launched.

(Refer Slide Time: 29:45)



We can quickly see an example that, threads when we launch a grid number and the number of blocks is launched, and when we launch in each block there are a number of threads. So, all of them are launched together; the total number of thread blocks are threads, the total number of threads that is the number of threads per block * the number of blocks they are launched.

Kernel launches all threads in a grid. So, threads are organized in blocks of grids; the kernel launches all threads in a grid following an SIMT model. Single instruction, the instruction is the same for all the number of threads.

The kernel function is executed into grid size into block size numbers in that many threads. So, each thread is executing the kernel function. So, the total number of threads is the total number of executions of kernel functions; but each kernel function is operating on a different thread location. Different thread is tagging with different locations of the memory, because it is again single instruction multiple threaded, single instruction multiple data model. So, the same instruction is being passed, but each instruction is looking into different locations of the memory. So, each of these threads has to be tagged with a particular location in the memory.

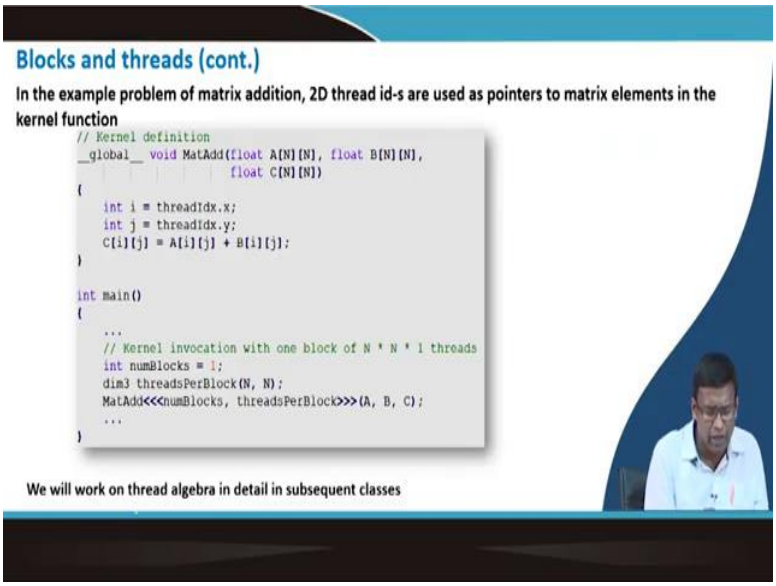
So, this is device memory, say, this will look into a particular location of the memory, this will be tagged to a particular location of the memory so on; each thread will take care of a particular location of the memory. So, it is important to find the global id of the thread from the local id; local id means that, local id of the thread is the id of the thread inside the block.

Global id is id of the thread inside the whole grid. So, it is important to find out the global id of the thread from the local id in the block and the id of the block in the grid, in which block it is residing and what is the local id of the thread within that block. Block size and grid size information are available in the execution configuration kernel. So, you always specify block size and grid size.

Now, local thread id is given, if it is a 3-D block distribution by thread id x. x , thread id x. y and thread id x. z. This is the id of a thread; this coordinate gives the id of the thread in the local block. The block dimension which is passed through the grid size argument blockDim x.x, blockDim x . y and blockDim x. z.

Block id is given by this. These are the built-in functions when we call from the kernel, it returns the values of the local block id, the dimension of the blocks and the local thread id.

(Refer Slide Time: 32:39)

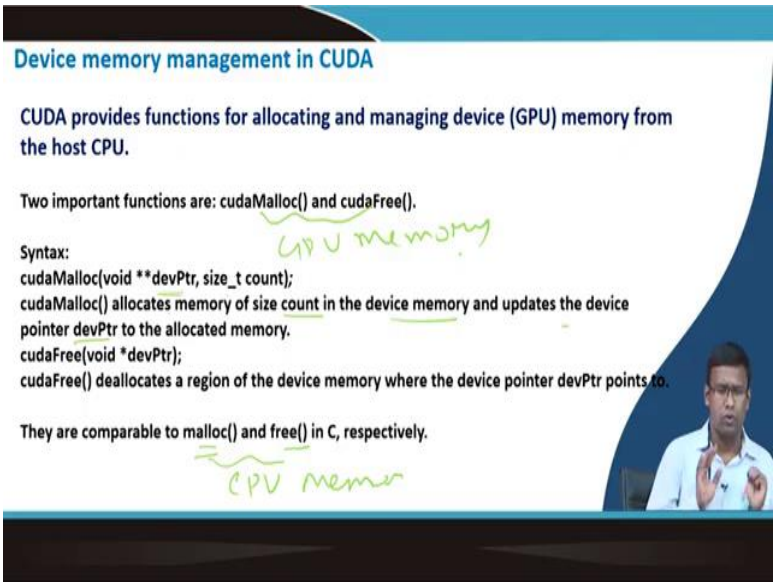


Using this we can constitute what is the global id of the thread. So, using this set of information, we can constitute global id. How global thread id is important, we can see an example of matrix addition. So, two matrices A and B will be added and we will get C. Now, we pass the number of blocks and number of threads per block. Then we said that the number of blocks is one only; so, one block is launched and all the threads are in the same block.

So, the question can be I mean; if multiple blocks are launched, then finding the global id is difficult. Why do we still launch multiple threads? This is due to the fact that there are multiple streaming processors in the GPU and one SM can launch one block at a time; if there are multiple blocks assigned to one streaming multiprocessor, there will be some sequentiality between them.

So, we launch multiple threads, because all the streaming multiprocessors can be active which can launch one on its own, more parallelization can be given. But let us see the example: one block is launched and each block has N by N thread structure. So, we find out what is the x id of the thread, what is the y id of the thread; and this i and j we put as a pointer of the memory of the matrix location, and use this to find the matrix addition. This is called thread algebra; we look into this thread algebra in detail in the next class that, how we can use the local ids of the thread for multi-dimensional block and thread structure and use them efficiently to point to the memory of the array or matrix which will be operated by the kernel call.

(Refer Slide Time: 34:38)



The next important part we understand that, when we write a GPU program; we cannot directly talk to the GPU memory, it is CPU who allocates GPU memory through cudaMalloc and copies data from CPU to GPU and copies back data from GPU to CPU through CUDAmemcpy commands.

So, GPU memory in CUDA is managed from the CPU only; the only thing is that the operations in the memory is done through GPU code. But memory allocations and reading from that memory is done by the GPU only. So, device memory management is an important part of a CUDA code; CUDA provides functions for allocating and managing device or GPU memory from the host CPU.

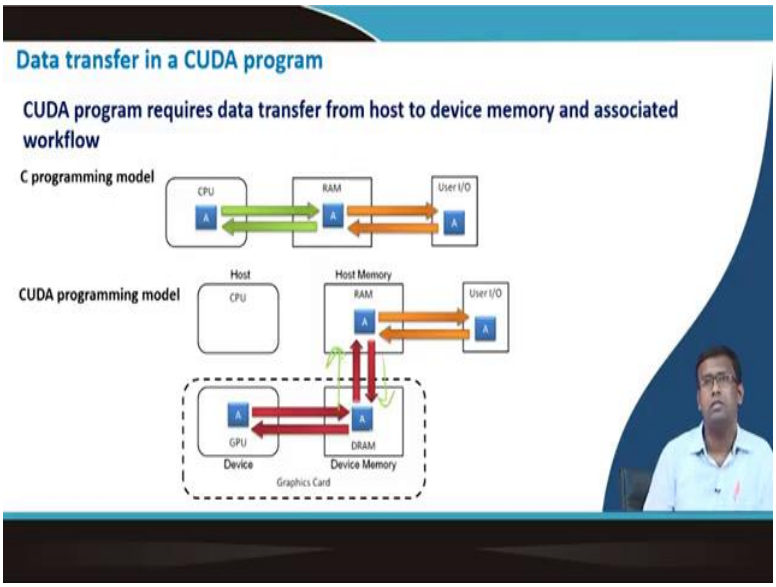
Two important functions are cudaMalloc and cudaFree; syntax is cudaMalloc, then the void location address of the memory in the device and the size of that memory. When we do that, cudaMalloc is being executed by the CPU, but it allocates the memory in the GPU. It allocates the memory of size count in the device memory and updates the device pointer that device pointer location with that allocated memory. So, it allocates some memory space in the device memory.

Malloc allocates the memory in the CPUs RAM sorry, cudaMalloc allocates in the GPU RAM. Once the operations are done; if you have to free the memory that is cudaFree; cudaFree of the memory location it deallocates the region of the device memory. They are compatible with

malloc and free; these operate on the CPU memory and these are on the GPU memory, but both are executed by the CPU.

So, CPU manages GPU memory through CUDA command.

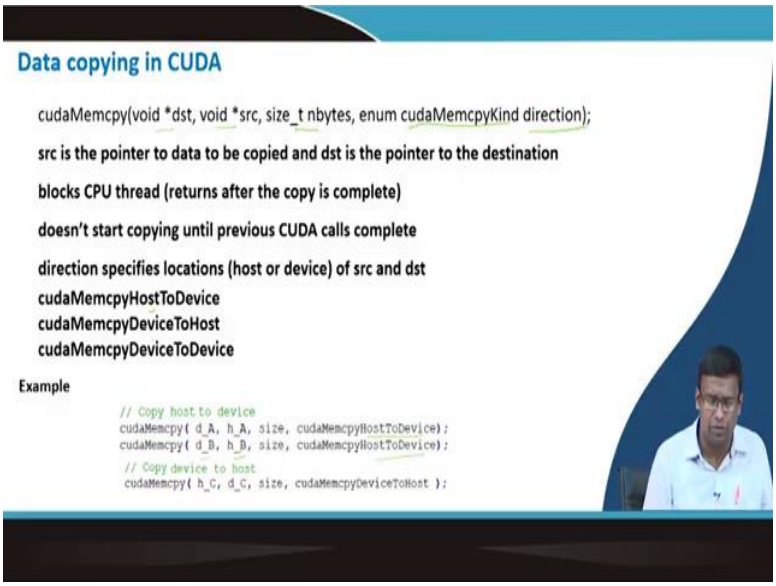
(Refer Slide Time: 36:50)



CUDA program requires data transfer from host to device and device to host and the associated workflow, because CPU cannot directly write to the GPU memory or GPU cannot write to the CPU memory. It has to be a data transfer; they are two different physical and virtual memory locations, therefore there has to be a data transfer across these memory locations.

So, when we run a simple CPU program, the CPUs RAM reads from the main file system and then CPUs registers read from the RAM and then operates on the variable and again writes back on the RAM. But when we look into a CUDA program model, the CPUs RAM reads from the main file system I O system and then it copies the memory to the GPUs RAM. GPU processors read from the device RAM ,GPUs RAM and changes that memory and then it is copied back. So, there is a copying to the device and then again it is copied back to the CPUs RAM that is the process.

(Refer Slide Time: 37:57)



The copying command is cudaMemcpy; cudaMemcpy says that, what is first you have to give the destination address, then source address, then size. Then the argument of cudaMemcpy kind which gives the direction, it is copied from CPU to GPU or GPU to GPU, device to host or host to device.

This direction is which direction this copy is done; is CPU copying to GPU or GPU is coping back to CPU. So, src is the pointer to be the data to be copied and the dst is the pointer to be destination; first comes destination address, then comes source address. Destination and source do not lie on the same device; because then you do not need cudaMemcpy, destination either lies in CPU source in GPU or destination in GPU and source in CPU. So, destination and source lie in the different devices.

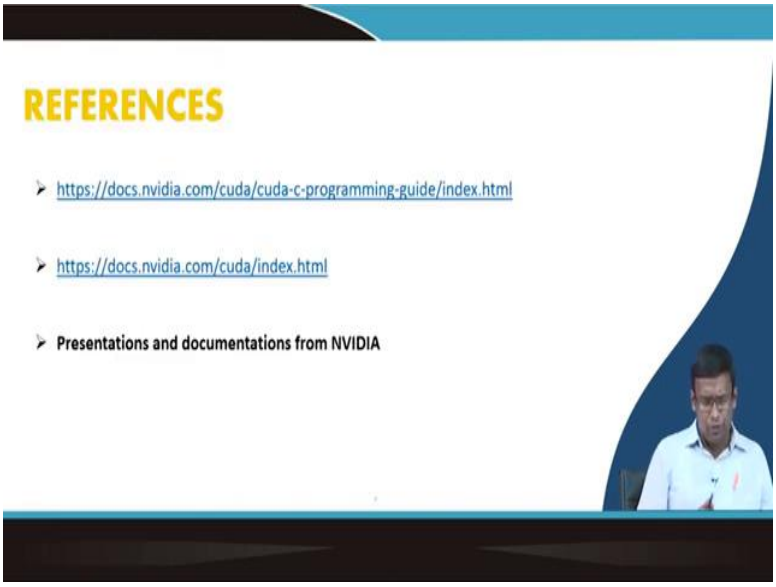
Interestingly this is a blocking thread, so this can act as a synchronization step; after your CUDA kernels are over, unless cudaMemcpy is executed, none of the executions can flow . CPU cannot do anything else, none of the executions can flow. So, this will be seen later this is acting as a thread synchronization or a barrier after the GPU calls. It does not start until previous CUDA calls are complete. So, all the threads have to come into an end; then only it will start.

This direction can be host to device, CPU to GPU, device to host, GPU to CPU and in certain cases device to device; for dynamic parallelism cases one GPU to another GPU, device to device.

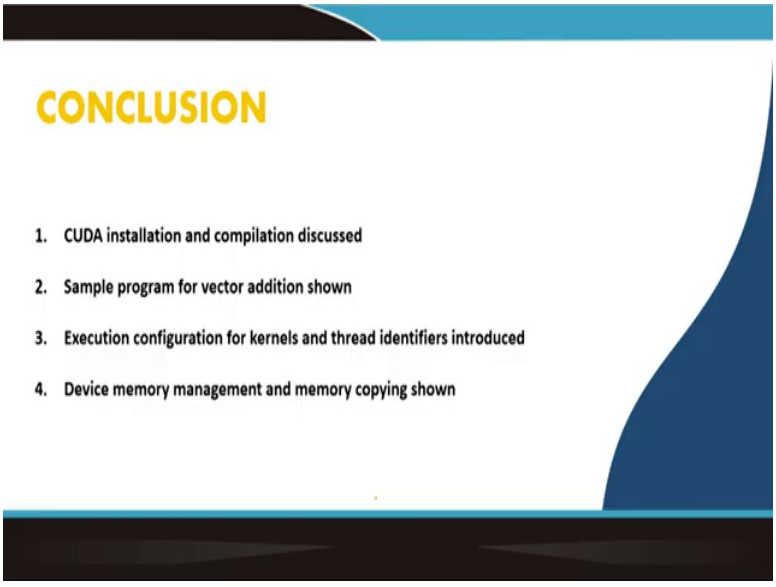
An example is that, in our code we have copied host memory A to device memory A and the direction is host to device; host memory B to device memory B direction host to device. Then after we calculated C is equal to A plus B, the device vector C is copied to the host vector C and the direction is device to host.

That syntax is given and this is extremely essential; because unless you are not using a unified memory access pattern, these memories are different and we always have to copy from CPU to GPU and GPU to CPU.

(Refer Slide Time: 40:15)



(Refer Slide Time: 40:20)



Well, these are the references, mostly it is taken from in NVIDIA's CUDA lecture manuals. We have looked into CUDA installation and compilation. A sample program for vector addition is shown.

Execution configuration kernels and thread identifiers are introduced; how to do very simple rudimentary form of thread algebra that, how global thread ids are to be found out and how a thread can be related with a memory location that is discussed, and then device memory management and memory copying is shown.