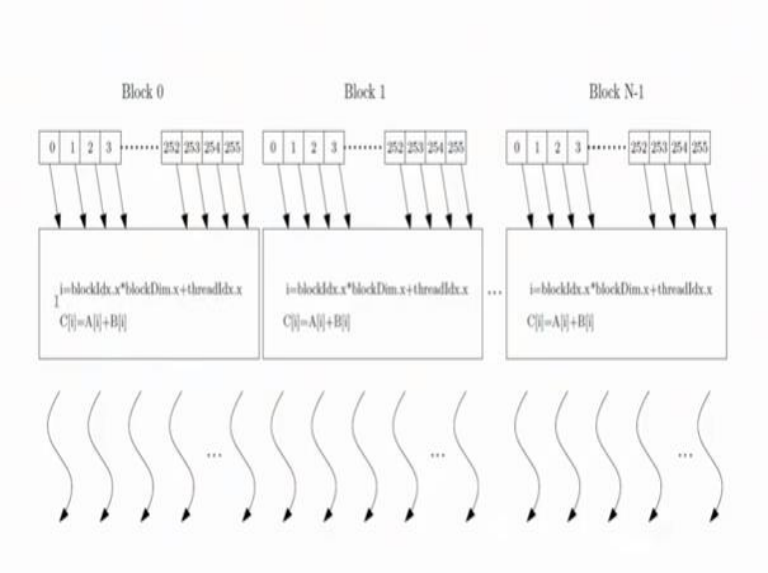
GPU Architecture and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 05
Lecture No # 10
Intro to CUDA programming (Contd.)

(Refer Slide Time: 00:28)



Hi so as we have discussed that there is a hierarchy of blocks and threads for blocks which defines the grid of threads so looking at how the arrangements can be thought of pictorially we have this n number of block each block is comprising 256 threads as you can see in the picture all of the threads are trying to progress their computation in parallel. And this is how the threads are indexed here. So now let us go back the code here the vector at kernel program once again

(Refer Slide Time: 01:13)

Examples : Vector addition CPU-GPU

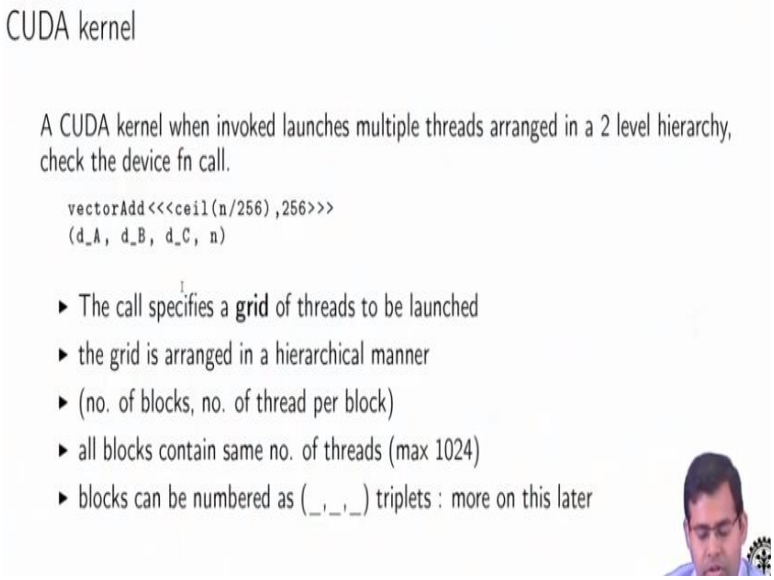
```
#include <cuda.h>
#include <cuda_runtime.h>
__global__ void vectorAdd(float*, float*, float*, int);
/*------------------------------------------------------------*/
__global__
void vectorAdd(float* A, float* B,
float* C, int n){ //CUDA kernel definition
 int i=threadIdx.x+blockDim.x*blockIdx.x;
 if(i<n)
         C[i] = A[i] + B[i];
}
/*------------------------------------------------------------*/
void vecAdd(float* h_A, float*h_B,
float* h_C, int n)
{//host program
   int size = n* sizeof(float);
   float *d_A=NULL, *d_B=NULL, *d_C=NULL;

   // Error code to check return values for CUDA calls
   cudaError_t err = cudaSuccess;
```

So we have a line here which says i = thread id x dot x and block dim x block dim dot x star block id x dot x. So this are again internally defined variables thread id x block dim and block id x and i is the local variable which is being used by this kernel to compute the global id of a thread of a CUDA thread. Now let us go back to this picture of this thread arrangement and try and understand what is happening.

So for every thread I have valve of its block number so for example any of this threads whose global thread if I want to compute so for block 0 I have threads numbers from 0 to 255. So for all this threads the block id is 0 the block dimension is 256 and thread id if any of the possible values from 0 to 255 okay.

(Refer Slide Time: 02:30)



Now coming back to the earlier slide we discussed that blocks can be numbered as triplets that is why we have this facility of block id x dot x. In this we are only doing everything 1 dimension so block id's are all linearly numbered so I only have defining block id's doing x dimension. So for all the threads inside this block we have them with the block id 0 for all the threads in the next block we have them with the block id 1 all the threads in the last block have the block id N -1.

The block dimension as already said as 256 now again for block dimension I can have a hierarchy block can be multi-dimensional only in that case the block id's are actually triplets but here it is only in the single dimensional. So it is basically 256 here I have block id 0 and whatever is the actual thread id comes here. So for the first block the value of i evaluates to be equal to the thread id right.

For the second block the value of i evaluates to its own thread id plus since the block id is 1 so the own thread id plus a 256 right. So essentially what is this i? If I can think of all this threads arranged together and I am making an arrangement of their ordering so then their thread id would be 0 to 256 here followed by 257 sorry 0 to 255 here this should be 256 this should be 255 + 256 like that right.

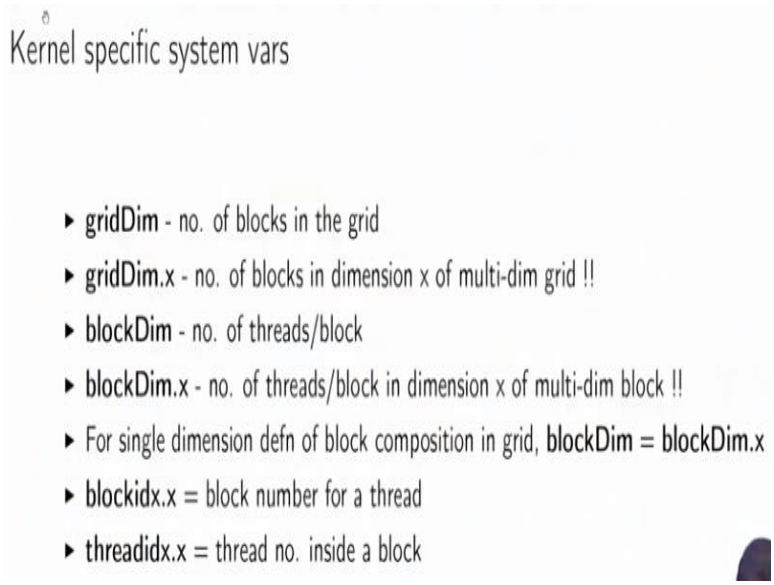
So that is the value that is get stored in 9 so it is the global thread id so that gives me the actual if of the thread and with that value of i I can decide that this thread will act on which element of the arrays. So using this notion of blocks and threads per blocks I am dividing the arrangement of

threads hierarchally. Finally when i do I have to give the actual computation I have to compute from this hierarchy what is the exact ordering of this thread and that is defined by this global thread id that I do compute with this kind of access expression.

And then this thread id is used to decide this corresponding thread will act on which elements of the data points in A and B. So as we can see that let us say I pick up thread 1 here of block 0 if I take thread 1 of block 0 so that will give me i as 1. So this will operate on C1 I mean it will write to C1 the value A1 + B1 if I pick up something let us say thread id x 1 from block 1. So essentially that is 1 + for block id x have the value 1 times block dimension 256.

So 256 +1 that would be the value of i here and accordingly suitable locations in AI and BI will be used to do the addition right. So in that way every thread maps to a different unique id and using that unique id it is decided which data elements with the thread warp 1 and which data element will be thread update. So with this back ground we have possibly a clear understanding of how vector addition is done in parallel using a simple CUDA program. Now let us try and understand this notion of thread and block definition I mean I a more detail with respect to CUDA syntax.

(Refer Slide Time: 06:44)



So as we have discussed that the packing of threads together is defined as the grid the grid is divided into blocks and inside each blocks I have the number of threads are threads per block. Now thinks can be upto 3 dimensions here that mean I can have a grid defined in 3 dimensions.

So if I say grid dim that is the number of blocks in the grid if I say grid dim dot x that means the number of blocks in dimension x considering that the grid is multi-dimensional.

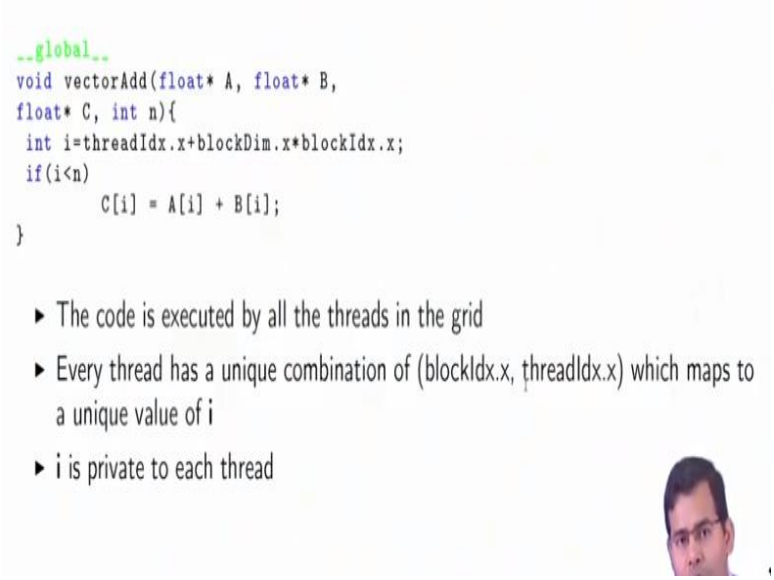
If I say block dimension that is the number of threads per block if I say block dimension dot x that means number of threads per block in dimensions x of a multi-dimensional block. So or single dimension that definition of I mean block and block dim dot x is same. So whenever we are talking about a single dimension of block if I say block dim it is essentially meaning block dim dot x.

If I say thread id x dot x or block dim or block id x dot x essentially they are talking about the single linear dimension. So by default I mean using a single value instead of a triplet in definition of thread id's or the block id's I can access them using block id x dot as well as thread id x dot x variables. So in that way for a single dimension I have block id x dot meaning the block for a given thread and thread id x dot x meaning the thread number for the thread inside a block.

So for any thread it has got it is own values of this variables thread id x dot x and block id x dot x considering a single dimension. So if I have a hierarchal arrangement of the grid divided into blocks and threads per block both in single then for any thread I will have a value of block id x dot x and thread id x dot x. This along with block dimension can be used to find out the global thread id that is the summary.

If I am considering to be packed in multiple dimensions 2 dimension or 3 dimensions then the blocks will be arranged as per following as triplet or duplet arrangement accordingly we will definition for block id x dot x block id x dot y. Similarly threads can be packed inside the block in 2 dimension or 3 dimension accordingly we shall have values for thread id x dot x thread id x dot y thread id x dot z like that.

(Refer Slide Time: 09:33)

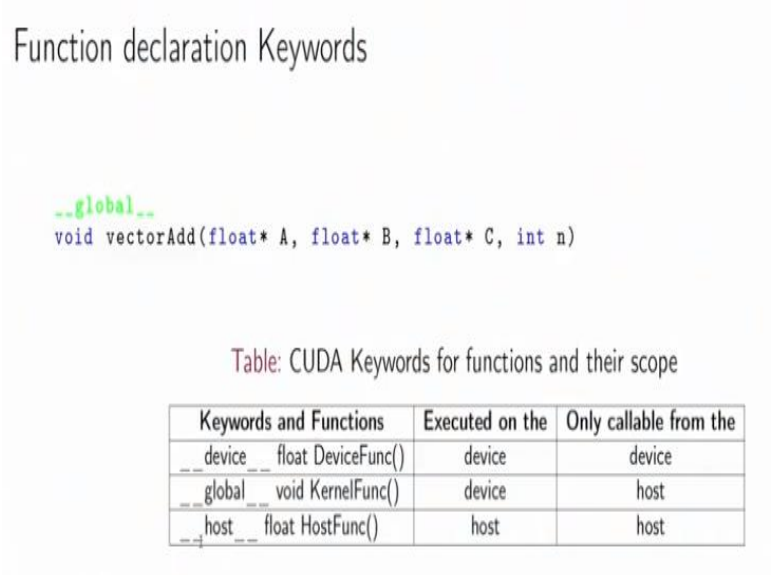


So for this vector add code the code is executed by all threads in the grid every thread as a unique combination of block id x and thread id x essentially for no thread I will have all this values same I mean block id and thread id all of them will not match. Hence we will evaluate to the global thread id that is the unique value of i which is private for this thread I mean (()) (10:01) to that thread.

For example we already have here like suppose I pick up thread which ID 253 a thread with ID 253 in the block 1 then I can see that for this thread the block id is 1 block dimension is on 1 dimensions of block dim dot x is 256 and thread id x dot x is 253 so that will give me a value of i and this will decide exactly 256 + 253. So I have 509 so that is the value of I it decides exactly which of the A and B array positions are to be added by specific thread.

So that is kind of summary of how thread are getting declared and arranged inside a CUDA data space and how the threads get to know what is that index in the overall data space and how that index can be used to decide what are the data points on which the thread is going to act on.

(Refer Slide Time: 11:18)



Following this we have some discussion on the different keywords that are to be used for declaring CUDA functions. Now if you noticed that for this vector add we have used a keyword here highlighted in green that is global. Global is a key word followed by standard definition or standard function like definition written type is void because of course this is the device type device function I mean this is kernel it will execute in the GPU device it has its read and write parameter the device memory locations which are to be passed to this pointers.

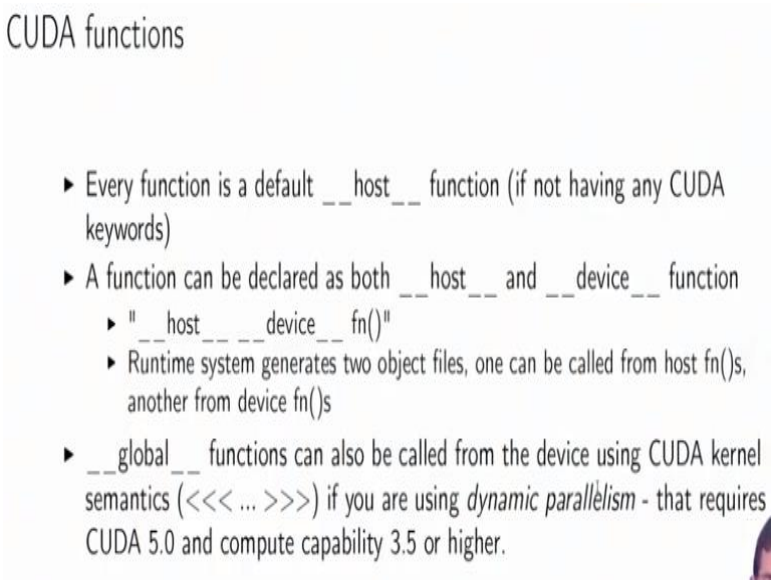
And this keyword global actually it is indicator of the nature of this function so these are CUDA keyword and this keyword helps to define what is the scope of the function overall for such functions we have 3 possible keywords global and host. So when I say the keyword is device for a specific function there meaning that it is an function that can execute on a GPU device it is callable on from the device.

Note that the function we used where a keyword global which mean it is function can execute on a GPU device but it is callable only from the host that is why we add host side vector addition program which was actually calling the kernel or launching the kernel the GPU device. So that is like functions with global keyword there can be function which are device keyword level there mean they are kind of kernels that can be launched from other kernels.

Basically some other device function so that is why is only callable from device function and we also execute on a device function. Now if I use this keyword host so that is a normal program

execute on the host side that is a CPU and they are only callable from another function execute on the CPU that is essentially a host type function can call another host type function and they are to be level by keyword host. So this are the 3 different possibilities of CUDA functions with respect to their scope.

(Refer Slide Time: 13:39)



Now by default every function is definitely a host function if nothing is specified that means the default is host I mean it is a normal C function a function can be declared as both host and device now when will you like to do that and the definition will look like this. I have the declaration with the host followed by device then the functions normal declaration. If it is given like this then the run time function we will generate 2 object device.

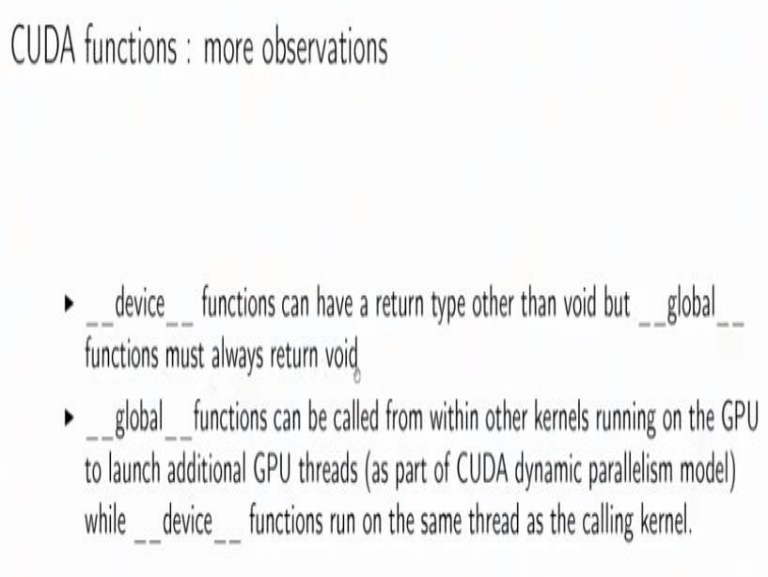
By default as we have seen that the host side function is compiled by the normal CPU's C compiler and (()) (14:14) pipeline and gives a host side object code. Whereas the device side function from compile from GPU to device to create a separate object codes. But if for a function we have both this host and device stag then the run time system will generate 2 object files. One can be called from host functions and another can be called from device functions.

So since it may be required that the function is such that sometimes a host code calls it and it will execute in a GPU or it can be also such that at the run time some other GPU device function calls it so having that kind of facility I may like to have 2 of this object files ready and available with

me. Now another important thing is this functions which are declared as global by default if you look into the definition.

So this global tagged functions are essentially device functions which can only be called from host side functions but if you are using CUDA 5.0 with compute capability 3.5 or higher then a global function can also called from the device using the CUDA kernel semantic that is this location. If you are dynamic parallelism so in case I want the global function to be also called from other (()) (15:45) other I mean from a device side functionality that requires this support of dynamic parallelism.

(Refer Slide Time: 15:52)



So to break it down what will that mean let us just explain first of all what is the device function it can have a return type other than void. But global functions must always return void this is a because a device function is basically called by another function that is executing already in the GPU. As you can see in the device qr is function which will be executed on the GPU and it is called by another function from the GPU side only it is called by device side program.

So definitely in that case this function executing on the GPU and return back to the calling which is also resident on the GPU. But that is not allowed when I have a device function executing and it is being called from the host side. Basically a device function with global qr because if the calling is resident on separate processor whereas this function is executing on GPU card physically.

So that is why the device function that can have return type other than void a global function must always return void then we have to use CUDA mem copy directives to copy back the values to the host side calling. But as we have discussed earlier that global functions can be called from device type functions in case I have this higher levels of CUDA and this is known as dynamic parallelism.

Let us understand what we happen in this case so suppose a CUDA kernel is executing so that means it as launched multiple kernel of a multiple thread. So let us say there are n number of threads that are launched and in case I have this support of CUDA dynamic parallelism then this device side function can also launch a global function and that would happen and suppose the global function is going to launch n number of threads in each location.

So overall I will have effectively n times m number of thread launches because this m number of threads of the global function will be launched for each of the n threads that already executing by the (()) (18:12) device function. So this is known as an example of dynamic parallelism in CUDA which as come into existence to CUDA 5.0. So with this we have discussed the classification of different possible types of CUDA functions that we have based on from where they are called and we have provided a discussion on how a basic CUDA function can be return I mean how a CUDA kernel operates I mean in conjunction with the host program.

How the host program can set up memory elements which are to be copied for as the necessary arguments for execution of a CUDA kernel. Following that copy how a CUDA kernel is actually invoked and once the CUDA kernel is execution is done how the computed values can be copied back. So with this we would like to end this lecture and in the next iteration we would like t go to bit more advanced concepts of basic CUDA programming thank you.