

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

**Module No # 02**  
**Lecture No # 09**  
**Intro to CUDA programming**

Hi let us get started with the third topic in this course on GPU architectures and programming. Now we are to introduce the notions of CUDA programming language one of the most popularly used GPU programming languages.

**(Refer Slide Time: 00:48)**

### Compute Unified Device Architecture

- ▶ CUDA C is an extension of C programming language with special constructs for supporting parallel computing
- ▶ CUDA programmer perspective - CPU is a *host* : dispatches parallel jobs to GPU devices

So what is CUDA the full form first of all is compute unified device architecture. So CUDA is essentially an extension of C programming language with special construct that support parallel programming. Now this is a programming language extension developed by nVIDIA primarily for their GPU's as we know that nVIDIA GPU's are very popularly used as accelerating a device in conjunction with I mean high performance CPU's.

So for the CUDA programmer the perspective is that the CPU is the host that means in CPU you have an (()) (01:35) program which is running and it is dispatching parallel jobs to GPU devices. Well they are may be more than one GPU device attached with the CPU. So generally, it can dispatch multiple parallel jobs to this GPU devices. This job will execute in the devices and get back the results to the host.

(Refer Slide Time 01:57)

CUDA program structure

- ▶ *host code* for a host device (CPU)
- ▶ *device code* for GPU(s)
- ▶ Any C program is a valid CUDA host code
- ▶ In general CUDA programs (host + device) code cannot be compiled by standard C compilers

NVIDIA C compiler (NVCC)

GPU Architectures and Programming | Soumyajit Dey, Asst. Prof.

So the way a basic CUDA program is structured is as follows. So there is a host code which is resident that means it executing on a host device that is the CPU. And there is a part that we called as device code which is supposed to execute on the GPU's. Technically speaking any C program is a valid CUDA host code. Essentially it is a code that it can execute on CPU. So in general CUDA programs constitute of host plus device code and they cannot be compiled by any standard C compiler and for this purpose we require this specific compiler from NVIDIA that is the NVCC compiler. That is the NVIDIA C compiler.

(Refer Slide Time 02:43)

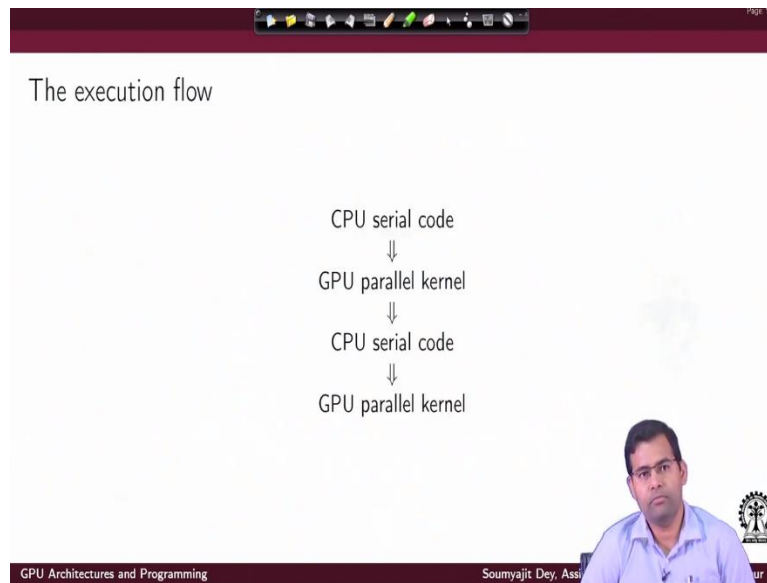
The compilation flow

```
graph TD; A[C program with CUDA extensions] --> B[NVCC]; B --> C[Host Code]; B --> D[Device code]; C --> E[Host C compiler / linker]; D --> F[Device JIT compiler]; E --> G[CPU + GPU]; F --> G;
```

GPU Architectures and Programming | Soumyajit Dey, Asst. Prof.

The compilation flow for NVIDIA C compiler is as follows. That you write the CUDA program essentially C program CUDA extension you compile it with NVCC what you get is the host code and the device code that is the part of the code which will be compiled further by the host C compiler and linker. And that device code will be JIT compile for execution on the device. So overall these are the two different segments and code that you get one to execute on the CPU and the other to execute on the GPU devices.

**(Refer slide Time 03:19)**



The execution flow is as follows. In the host code is the code that is executing in the CPU serial. The host code will launch the device code that is the GPU parallel kernel. The GPU parallel kernel will execute in the GPU this is what we referring to the device code. It will return results to the host code with those results the host code may again execute do some functionality and then again launch some parallel kernels for the GPU devices. And this computation can go on back and forth.


I can have a simple CUDA program which will launch one kernel per GPU device get back the result and print me the result or I can a sufficiently complex program which will launch some kernel get some result do some processing in the CPU and then again launch another kernel and so on so forth.

**(Refer Slide Time 04:16)**

Page 1/1

Examples : Vector addition CPU only

```
void vecAdd(float* h_A, float* h_B,
float* h_C, int n)
{
    for (i = 0; i < n; i++)
        h_C[i] = h_A[i] + h_B[i];
}
int main()
{
    float *h_A,*h_B,*h_C;
    int n;
    h_A=(float*)malloc(n*sizeof(float))
    h_B=(float*)malloc(n*sizeof(float))
    h_C=(float*)malloc(n*sizeof(float))
    vecAdd(h_A, h_B, h_C, N);
}
```



GPU Architectures and Programming Soumyajit Dey, Assis

So before discussing our first CUDA program let us start with an example vector addition code that is executing in a CPU the very simple C program. So you have a main in which you have this float types define. So this are the pointers to floating point arrays which I mean which you dynamically allocate with the malloc calls. Once this dynamically allocation is done you call this vector add function. Inside this vector add function of course I am just trying to show you an example we have not written any code for initialization of arrays and all that assume those are there.

So once that is done the vecAdd will be called and after vecAdd is called inside a loop I am just trying to add the 2 vectors and return the result in the h C h underscore C array. So in that way this vecAdd with three arguments. The first two are kind of the input argument and h underscore C is a dynamically allocated array which we contain the output argument. Of course, the initialization code is missing here as discussed earlier. So this is how the typical vector addition will execute in a CPU.

**(Refer Slide Time 05:48)**


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;

```



GPU Architectures and Programming Soumyajit Dey, Assista

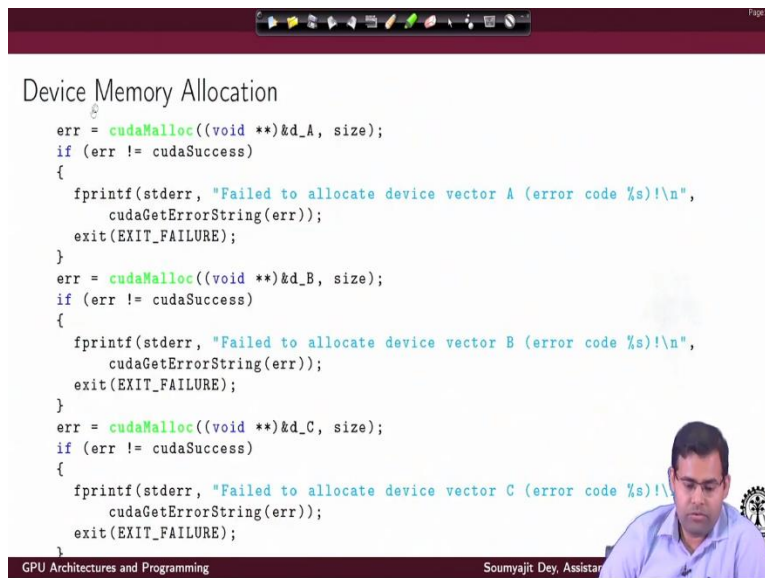
How it will work on CPU GPU system? So here comes the first CUDA program. First of all you observe the hash includes that we have well we have hash include CUDA dot h and also we have a hash include of CUDA underscore runtime dot h. This are the header files which contain the required CUDA functionalities that will be using in our code. In this first CUDA program as we have discussed earlier there will be a piece of code that will execute in the CPU it will launch the parallel code or the device code for the GPU. This is what we called a kernel.

Now what is the code that will execute in the CPU. So that is essentially this vecAdd program. This is the program that is commented as a host program as you can see this similar to our earlier CPU only vecAdd program. So essentially this is a program that is called from main, it is expecting to be past three pointers right. So this pointers that containing this addresses for three arrays which has been dynamically allocated before the call has been made and also suitably initialize that is the input arrays.

And they are added as vector and output array is return. That was my original vecAdd code. Here I can have the VecAddcode which is the CPU side host program. Similar input arguments are there. Now what we are trying to do here is we do not want the addition to be done in this CPU. This host code is supposed to launch something called a GPU kernel that is the code that will be executed in the GPU right. So let see how it is done. So we will just watch through this program step by step.

First thing we will do is inside this host program `vecAdd` we declare some pointers and then we also declare a specific enumerator data type `cudaError_t` and we initialize it with one of the types that is `CUDA_SUCCESS`. So this is the error code to check return values from CUDA calls. Now these are all defined in the header file that we have defined earlier. These are not our own design they are already defined.

**(Refer slide Time 08:25)**



```
Device Memory Allocation
err = cudaMalloc((void **)&d_A, size);
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to allocate device vector A (error code %s)!\n",
            cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
err = cudaMalloc((void **)&d_B, size);
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to allocate device vector B (error code %s)!\n",
            cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
err = cudaMalloc((void **)&d_C, size);
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to allocate device vector C (error code %s)!\n",
            cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
}
```

GPU Architectures and Programming Soumyajit Dey, Assistant

Now coming to the code, the first we are trying to do is just like we do `malloc` for dynamically allocating memory in a CPU we fire a function `CUDA Malloc`. Now again these are all defined in the `CUDA runtime` library the `CUDA headers` and what `CUDA Malloc` will do is it will allocate memory not on the CPU side but rather the memory resident on the GPU device. It will allocate a specific amount of the memory of size equal to this size and return a pointer for that.

And this is the pointer `d underscore A`. So as you can see we got we have already declared this `d underscore A` so now after this `CUDA Malloc` call `d underscore A` address is containing a generic pointer and this pointing to an amount of size amount of memory resident on the GPU device. In case this `malloc` call is not going perfectly then this next `if` condition will not be satisfied. So error will not be equal to `CUDA_SUCCESS`. Again, `CUDA_SUCCESS` is again another enumerator data type which should match provided the `malloc` call goes perfectly fine.

If it is not so then this condition will fire, and we will have the `if` printer a standard error where this message will be printed with the suitable error code. Now how is the error code coming in?

The error code is what is the reason can be figured out by this directive `CUDA Get Error String`. So `CUDA Get Error String` will take as a argument with the error of type `CUDA Error underscore t` and from that error code it will figure out the suitable error string which will be present which can be printed here using the `if printer` command.

Again I am assuming that you are very much conversant with `if print` if printing strings to standard error and everything. So please get acquainted with it if you have kind of forgot and all that. So and hence forth and exit will happen with this suitable code. So following this (()) (10:55) we will do memory allocation for the other pointers that is `d B` as well as `d C`. So as we can understand there is a difference between the original pointers that have been passed to the `vecAdd` host program this were `h A`, `h B` and `h C`.

So they point to memory location on the CPU memory they point to resident arrays on the memory location which have been dynamically allocated and initialize before the call the has been made for `vecAdd`. Internally `vecAdd` defines more pointers and make the `CUDA Malloc` call so the and assign `dim` assign these generic pointers so that they can point to memory locations not on the CPU side but on the GPU side.

We will soon see why this necessary so as you can see in this line we have a kind of repetitive code of `CUDA Malloc` in `d A`, `d B` and `d C` and incase of error suitable error commands will trigger as has been written down here.

**(Refer Slide Time 12:02)**

Host to Device Data Transfer

```
printf("Copy input data from the host memory to the CUDA device\n");
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to copy vector A from host to device (error code %s)
    \n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to copy vector B from host to device (error code %s)
    \n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
```

GPU Architectures and Programming Soumyajit Dey, Assistant

Following this we have this printer statement which is saying that copy input data from the host memory to the CUDA device. So if this is printer that means all the previous malloc operation went fine so we have 3 locations in a memory of size equal to size allocated on the GPU memory side and they are being pointed two by dA, dB and dC. So the next thing that I need to do is I need to copy the input arrays from the host side memory to the device side memory.

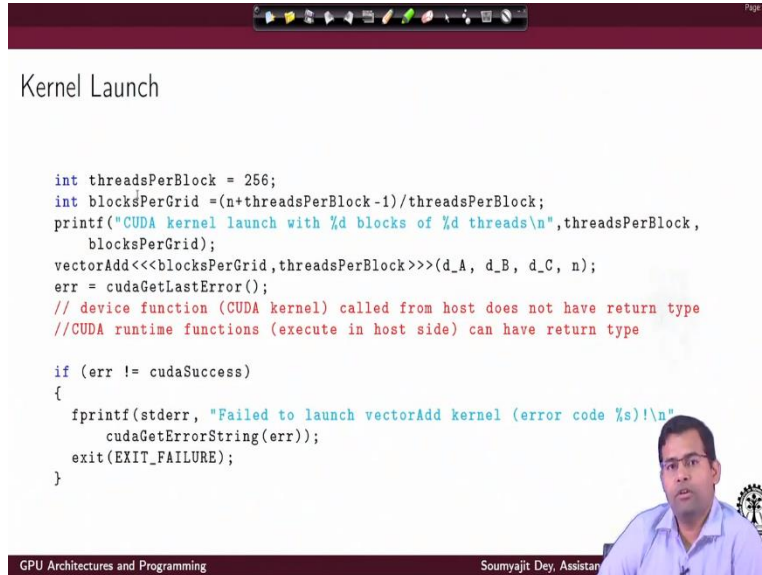
From the CPU memory to GPU memory right because essentially, I want to do the vector addition on the GPU in parallel not on the CPU. That is the basic idea of this program which is vector addition on a CPU GPU system. So now for doing this transfer of data from the host side to the device side we have this command CUDA Mem copy. So what it does. So this is the device side memory dA, this is the host side memory hA.

So there is a copy from hA to dA the copies size is dictated by this parameter size and the type of copy is CUDA Mem copy Host To Device. Following this directive all the data resident in hA up to size amount of space gets severely copied to the location pointed to by a dA in the GPU side memory fine. Now incase this is again not successful suitable error comments will get printed as has been already discussed. Similarly we also do the CUDA Mem copy for the array hB to the device array dB.



Once both of these things get done as you can see that the codes are again kinds of similar. Once both these operations get done, we have the 2 input arrays containing the 2 input vectors ready for addition in the GPU side memory.

**(Refer Slide Time 14:26)**



```
Kernel Launch

int threadsPerBlock = 256;
int blocksPerGrid =(n+threadsPerBlock-1)/threadsPerBlock;
printf("CUDA kernel launch with %d blocks of %d threads\n",threadsPerBlock,
      blocksPerGrid);
vectorAdd<<<blocksPerGrid,threadsPerBlock>>>(d_A, d_B, d_C, n);
err = cudaGetLastError();
// device function (CUDA kernel) called from host does not have return type
//CUDA runtime functions (execute in host side) can have return type

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to launch vectorAdd kernel (error code %s)\n"
              cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
```

GPU Architectures and Programming Soumyajit Dey, Assistant

At this point we are thinking of launching a program on the GPU which will access this array location in the GPU side memory do the addition and stored the result and transfer the result back to the CPU side memory. For doing that we declare certain suitable parameters known as threads per block and blocks Per Grid and we pass this parameter to function called vectorAdd.

Now as you can see the definition the call of this function is very different from our standard C function because apart from its argument being this dA, dB, dC and n there is something here called blocks Per Grid and threads Per Block. Now what this is we will discuss a bit later on for that time being just think that this is specific type of function with some extra parameters. Essentially this is how we are launching program for the GPU.

We are going to launch a function in the GPU which will do the computation of vector addition in parallel. This kind of function which does computation on the GPU side is traditionally known as kernel GPU kernel. GPU kernel launches follow this kind of parameter reiterations. Now the parameter definition is something we will speak a bit later on. Now just observe that the function has been apart from this threads Per Block and blocks Per Grid parameters which also been passed the GPU device memory location dA, dB and dC.

A and B are containing the 2 arrays which are the A and B are essentially pointing to the vectors which are to be added and the result is supposed to be stored in dC all of them are in device memory. Now let us look at the implementation of this kernel which is right here. So this is how a CUDA kernel can be defined. So first we have the declaration of the vector add and then we have actual declaration of the vector add.

So inside vector add we have a computation of an index that decides what is the part thread behavior and then we have the familiar code of the locations C getting the value for  $A_i + B_i$ . So  $C_i$  is being written with  $A_i + B_i$ . Now the difference with the CPU side is here if the GPU has got n number of cores inside it n number of scalar processor that many of number of threads will be launched which will do all this addition in parallel. And in that way all this vector addition will happen in parallel. How exactly is something which we will discuss.

So coming back here so this was the call for vector add at this point so with this vectorAdd call we land up here this addition is done. Now going back here the vectorAdd call we expect that dC this device memory location contains the added vector value right. Now some thing important to be noted here is this is a function which is executing on the GPU side for such a function it can take as operands value from memory does the GPU memory and write back values again on the GPU memory. So they do not return anything.

So that is what we are saying the device function CUDA kernel call from the host side does not have a return type here. Now this is unlike CUDA runtime directives like CUDA Mem copy or CUDA Malloc which can return on error code. This function cannot have a return type. But incase this function run into some issues while executing it will create a signature which can be caught by CUDA Get Last Error it is again a run time function.

It is a function which is a feature of the CUDA runtime system. The point I am trying to make here is the vector add does not itself directly provide the return but rather in case there was some issue in executing the function. This runtime function CUDA Get Last Error can provide a return a suitable error and if that is not a CUDA Success again we have a way to know that the launch of this vectorAdd kernel got into some issues ok.

**(Refer Slide Time 19:34)**

Page 1/1

### Device to Host Memory Transfer

```

printf("Copy output data from the output device to the host memory\n");
err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to copy vector C from device to host (error code %s
    )!\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
// Verify that the result vector is correct
for (int i = 0; i < n; ++i)
{
    if (fabs(h_A[i] + h_B[i] - h_C[i]) > 1e-5)
    {
        fprintf(stderr, "Result verification failed at element %d!\n",
        exit(EXIT_FAILURE);
    }
}
printf("Test PASSED");
} // End of Function

```

GPU Architectures and Programming Soumyajit Dey, Assistant

Now so we have the vector addition done and the result is there in the device memory hC which can be copied back again by CUDA Mem copy directive to the CPU side or host side memory hC. Now observe that earlier when we copied the values from host side to device side the directive inside CUDA Mem copy was CUDA Mem copy Host to Device. But now when we copy from device side to host side it is CUDA Mem copy Device to Host. So that is the slight alteration.

So with this directive we have the result back in the CPU. So once I have the results back in the CPU I can de allocate all the allocation that apart from the device side memory that is we can free of this dA, dB and dC using the CUDA Free directive. So essentially it is very much (()) (02:34) with some CUDA annotation. And then we have just written some small checking code. We are we which is just checking whether it is a absolute value of the sum of A + B subtracted from C within the error bound or not otherwise we will say that ok that there is some issue right. And and then we do the test pass printer.

**(Refer Slide Time 20:57)**

## Compile and Run

```
nvcc kernel.cu host.cu -o output
./output
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Test PASSED
```

So if this code has to be compiled so we have to find this kind of a command we run the NVCC compiler with the kernel definition in kernel dot cu the host side code host dot cu will compile them to create the binary output. If you run it you get this sequence of print statements firing and this should be the output we would expect.

**(Refer Slide Time 21:25)**

## Observations

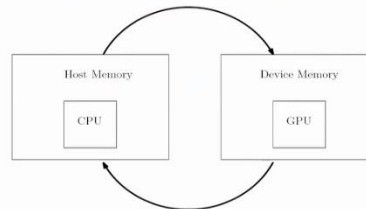


Figure: CPU/GPU Mem Layout

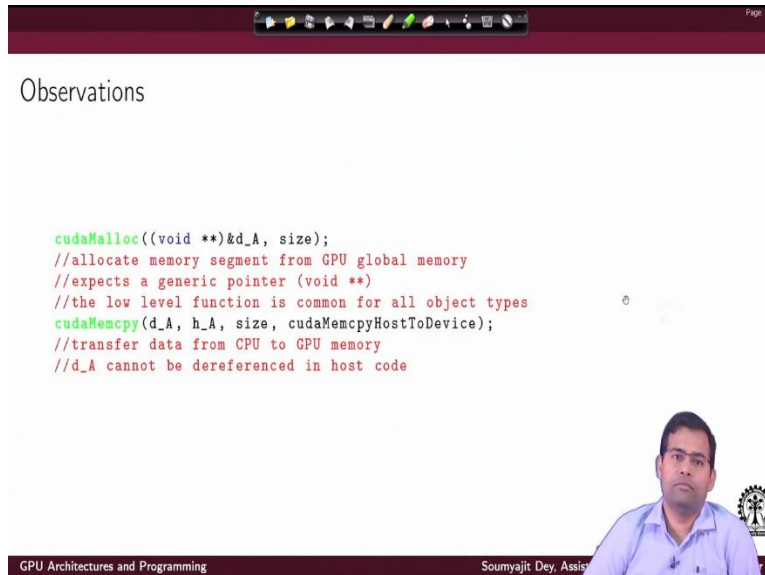
- ▶ `cuda.h` → includes during compilation CUDA API functions and CUDA symbols/variables
- ▶ `h_A, h_B, h_C` → arrays mapped to main memory locations

Ok now coming back to how really the execution is going on. So as we know that the GPU is a separate card which is a accelerator card which is kind of getting attached to the CPU and the GPU has got its own device memory. For doing the computation in the GPU from the last example we could figure out that suitable data points need to be transferred by the CPU to the GPU device memory.

Then the kernel is launched the GPU kernel is suitably launched by the first program. The GPU kernel executes on the GPU with input parameters taken from the device memory of the GPU. It writes data back on the device memory of the GPU which has to be again copied back to the CPU. So this is the overall execution flow in a bit more detail. This header file CUDA dot h includes the compilation CUDA API functions and the different CUDA system variables.

The once that we have been exhibiting with some examples inside the code and this as we also seen that there were the host code was using variable that were mapped in the main memory of the CPU whereas the device code wah I mean before executing the device code we needed to initialize pointers and allocate them suitable memory in the GPU DRAM or the device memory.

**(Refer slide Time 22:59)**



Observations

```
cudaMalloc((void **)&d_A, size);  
//allocate memory segment from GPU global memory  
//expects a generic pointer (void **)  
//the low level function is common for all object types  
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);  
//transfer data from CPU to GPU memory  
//d_A cannot be dereferenced in host code
```

GPU Architectures and Programming Soumyajit Dey, Assis

A few more observations so we have been using this functions supported by the CUDA runtime. So that there is CUDA Malloc now what was exactly doing it was allocating memories it was allocating memories segment from the GPU global memory which is physically different from the CPU's global memory. Now this expects a generic pointer whose type is void star star is a generic pointer it can point to any kind of data here.

Now this 2 level function is common for all object types that is the reason for why it is a generic pointer. Now the other things is the CUDA Mem copy comment as we have discussed earlier it will transfer data back and forth between the CPU and GPU. If it is a having the directive CUDA

Mem copy Host to Device. So it is copy from CPU to the GPU if it is CUDA Mem copy Device to Host then it is a copy from the GPU memory to CPU memory.

The important thing is this device memory pointers dA wants the CUDA Malloc is done with dA. This device memory pointer cannot be dereferenced in the host code. Simply for that reason they are pointing to a different physical location that is the GPU memory. So they can be dereferenced only inside the kernel code.

**(Refer Slide Time 24:18)**

Observations

```
//d_A cannot be dereferenced in host code
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
//transfer data from GPU to CPU memory
//can also transfer among different device mem locations
//can also transfer data host to host- we do not need that
//cannot transfer data among different GPU devices
cudaFree(d_A);
//free GPU global memory
```

GPU Architectures and Programming Soumyajit Dev, Assis

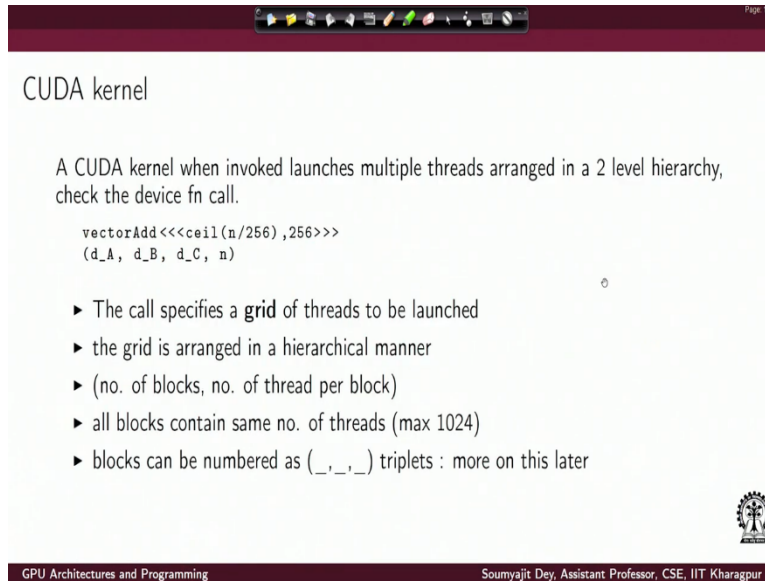
So with respective CUDA Mem copy we can have this kind of CUDA Mem copy from device to host, host to device and just like the transfer of data being supported among the GPU and CPU's with CUDA Mem copy. We can also do a transfer among different device memory locations. So even if I have two different device memory locations on the same device, I can do a CUDA Mem copy Device to device using a that kind of directive.

Also we can do a transfer from data from host to host but we do not need to do that right because since I had a normal CPU so that is normal movement of data inside 2 different location in the array. But I cannot transfer data among different GPU devices using a CUDA Mem copy directive and it is maybe very easy to understand right. Because what are the things that we can really do just to summarize.

I can copy data from host side memory to device side memory or from device side memory to host side memory. I can copy data between 2 different locations in the same device memory. I can copy data between 2 different memory locations in the same host side memory although that is not required because it is inside the CPU's DRAM. But what I cannot do is I cannot have CUDA Mem copy copying data from 1 GPU devices memory to another GPU devices memory.

Because in that case they are 2 again 2 different physical devices and CUDA Mem copy support star and it transfer between 1 host and 1 GPU. So in that case I have to copy data from device 1 to the host and then I have to copy from the host to the device 2.

**(Refer Slide Time 26:13)**



The screenshot shows a presentation slide with a dark red header and footer. The title is "CUDA kernel". The main text explains that a CUDA kernel launches multiple threads in a 2-level hierarchy and provides a code example: `vectorAdd<<<ceil(n/256),256>>>(d_A, d_B, d_C, n)`. A bulleted list explains the parameters: the call specifies a grid of threads, the grid is hierarchical, it consists of blocks and threads per block, all blocks have the same number of threads (max 1024), and blocks are numbered as triplets. A small logo is in the bottom right of the slide area.

CUDA kernel

A CUDA kernel when invoked launches multiple threads arranged in a 2 level hierarchy, check the device fn call.

```
vectorAdd<<<ceil(n/256),256>>>  
(d_A, d_B, d_C, n)
```

- ▶ The call specifies a **grid** of threads to be launched
- ▶ the grid is arranged in a hierarchical manner
- ▶ (no. of blocks, no. of thread per block)
- ▶ all blocks contain same no. of threads (max 1024)
- ▶ blocks can be numbered as (`_,_,_`) triplets : more on this later

GPU Architectures and Programming Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now coming back to the call of a CUDA kernel. How a CUDA kernel is called. So when a CUDA kernel is invoked launches multiple threads in a 2 level hierarchy. The threads are the busy computing units which engage the scalar processor in the GPU. Each scalar processor will execute one thread all the scalar processor executes threads in parallel. Now going back to our example CUDA kernel as we have discussed earlier that when this vector add kernel was launched there were 2 parameters blocks Per Grid and threads Per Block.

So let us now try to understand the significance of this parameters. So as we have already discussed that we are trying to do computation using multiple computing threads in a GPU that is the fundamental thing. So when I am doing vector addition I launch I want to launch a lot of

compute threads. All of them will add components of the vector in parallel. Now this arrangement of multiple threads follows a 2 level hierarchy.

So suppose I am trying to launch n number of thread. So what I can do is I can define their arrangement in 2 levels at the higher level I say that ok I have  $n / 256$  scaling number of element and then at the lower level is say that each element comprises 256 threads. So in total I have n number of threads being launched. So this call specifies the grid of thread to be launched. So that is the technical terms people use that you launch a grid of threads.

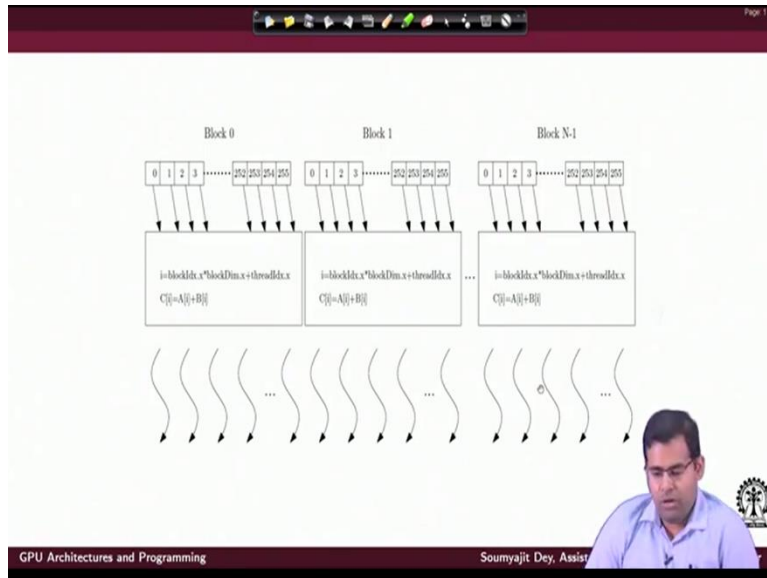
The grid is arranged in a hierarchical manner you have the number of blocks. So the first component here in the hierarchy is the number of block. So you have n by 256 number of blocks. And then you say that each block contains 256 number of threads that is the second parameter. So overall, I have a number of blocks and number of threads per block there is the specification which tells me that how many threads are launched.

Now if we go back to the definition of the vector add kernel. So we had threads Per Block defined as 256 and then we define the number of blocks Per Grid or just the number of blocks which was just  $n + \text{threads Per Block} - 1 / \text{threads Per Block}$ . So essentially we are looking at total n number of threads here. So you are if this print statement fire it will tell me how many threads are there per block and how many blocks has been launched essentially here we have the vector add being launched with the number of blocks in the grid, number of threads inside each block.

So of course, since I have a definition of threads Per Block that also means that all blocks contains same number of threads maximum 1024. Now I can even have a hierarchical specification of blocks that means I can number blocks in 3 dimensions using triplets. We will discuss this later on.

**(Refer Slide Time 29:33)**





So how are really blocks are arranged so when a CUDA kernel is launched I have this hierarchical arrangement of threads and suppose I have n number of blocks and each block is containing this 256 sets.