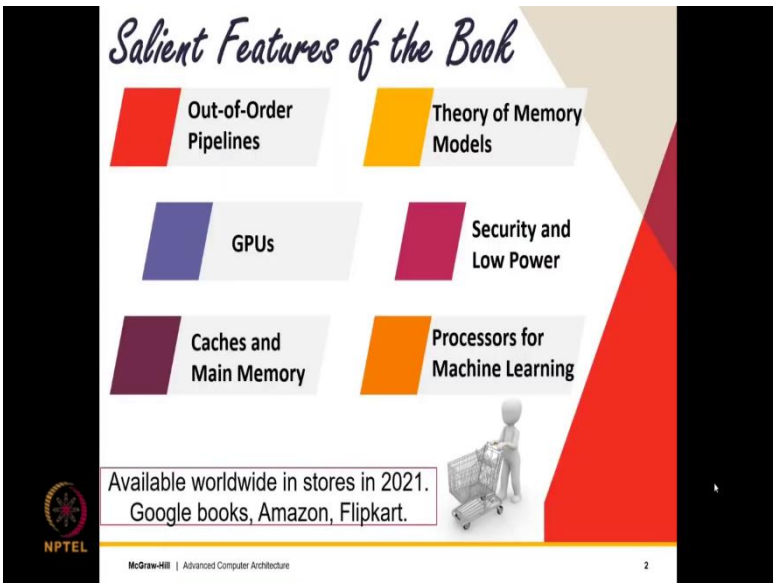
Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 17
Graphics Processors Part - II

(Refer Slide Time: 00:17)

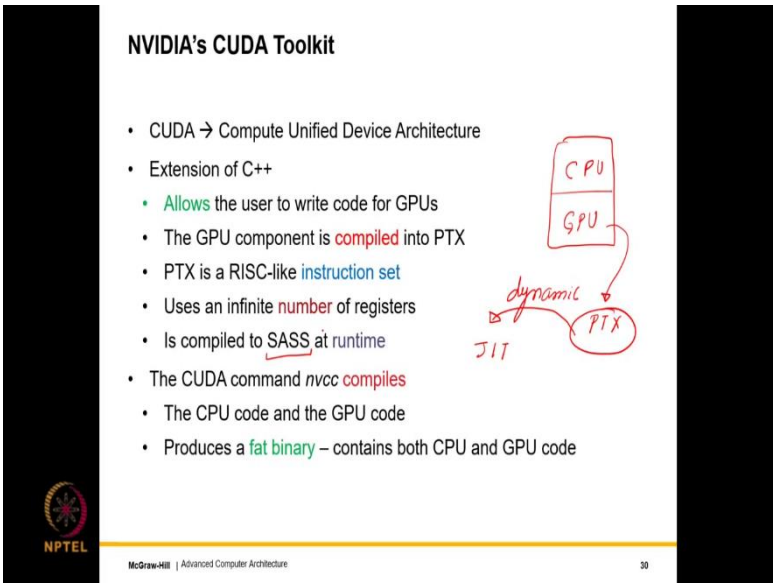
(Refer Slide Time: 00:23)



(Refer Slide Time: 00:37)

Welcome to the second video on GPUs. So, in this video, we will study the CUDA 10 language and figure out how to program GPGPUs. So, we will mainly be confined to the software interface.

(Refer Slide Time: 00:51)



So, the CUDA language was introduced way back in 2006-2007. CUDA stands for Compute Unified Device Architecture, which was NVIDIAs CUDA toolkit. It is primarily designed as an extension of C++, because C++ used to be the most popular language those days. And even if this were to be done now C++ would not be a bad choice, because a lot

of pointers are used in GPU programming. So, that is where the user C++ is required.

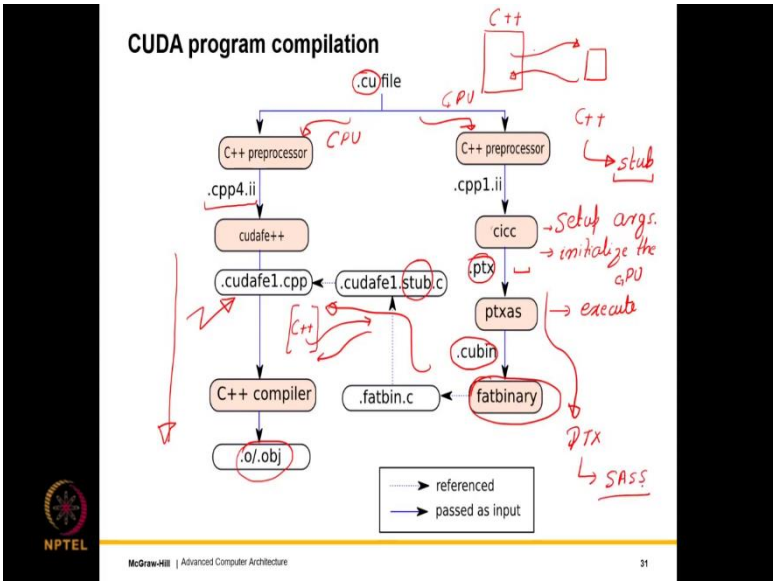
So, this allows the user to write codes especially for GPUs and again this primarily an NVIDIA GPU language. So, the GPU component of the fat binary, which we have discussed in a previous lecture; just to recap a fat binary has both CPU code as well as GPU code. So, the GPU component is compiled into a virtual instruction set called PTX. PTX is basically a RISC-like instruction set and it is assumes that there are a infinite number of registers.

So, one advantage of this is that let us say if there is a piece of code and clearly if there is a lot of intellectual property in the piece of code, we do not want to release the piece of code. So, what we would like to do is, we would like to compile it to an intermediate language, such that the original code is not recoverable and then they release. So, then what we do is, we release the piece of code, such that the users can run them.

But the point is that, there are different kinds of GPUs with different sizes. So, there are GPUs, small GPUs, bigger GPUs, less cores, more cores and so on. So, there is a need for dynamic recompilation. So, dynamically let us say at runtime, we have a JIT compiler; JIT means Just in Time which means that, whenever there is a requirement, the code is compiled and it is compiled to a binary known as shader assembly or SASS.

So, this is specific to a GPUs. Every GPU has its own low level instruction set, which is SASS; so the dynamic PTX to sass conversion happens at runtime. So, CUDA has a dedicated compiler called NVCC. So, you can think of NVCC kind of released on the same lines as ECC; say NVCC basically compiles the CPU code and the GPU code. So, it produces a fat binary; this contains both the CPU code as well as the GPU code, it has both.

(Refer Slide Time: 03:45)



So, here is the flow of the compilation of a typical CUDA program. So, we create a dot cu file. So, the .cu file is set in both directions towards the C compiler and towards the CPU compiler, towards the GPU compiler. So, let us look at the CPU side first. So, we have the regular compilation stages.

So, we have the C++ preprocessor over hear. So, the C++ preprocessor creates a file with this extension .cpp 4.ii. Then we have a regular C++ compiler called cudafe ++, which creates the extracts the C++ code and we will see that here there is a connection with the GPU, so we will come back to it.

But basically what happens is that, in the C++ code, it will generate kind of special code; the reason is that a lot of the C++ code what that, would is that would invoke a function within the GPU and get back its results. So, the way that CUDA works is that, we have a regular C++ program, which is the entry point; then what we do is that we invoke a function within the GPU.

So, we transfer data first to the GPU, then we invoke a function within the GPU and then we come back. So, after executing something on the GPU, we come back and then we get the data back. So, this is the typical process. So, what is done is that, when the user is writing the code, users simply write C++ code for the CPU and the GPU.

But all of this internal plumbing, which means set up the arguments, set up the data,

actually call the function on the GPU which involves many low level calls and again get the data back. So, all of these, all of this code also has to be injected automatically into the C++ code. So, that is what this file stands for.

And also of course, a bunch of files are generated, so it is important to understand what is a stub function. A stub function basically means that if let us say in the C++ code, I am calling a function to be run in the GPU; then instead of calling the real function which I cannot, because it is written in a separate isa and so on. I actually call a stub function, the job of the stub function is to do a bunch of things for me. So, these include setting up the arguments.

Then basically set the state of the GPU essentially initialize the GPU, send the arguments to it, say that a new functional execute. So, there are a lot of commands which need to be sent to the GPU with we just initialize it; then basically execute the function. So, all of these things are managed by the stub function and the and this is kind of library code API code, which is automatically generated and once that is done, then the C++ compiler compiles the program and we generate at .o or object file.

On the same lines if we go to the GPU side, again we have a preprocessor and as we know preprocessors job is just to replace macros; is just to replace the contents of macros. Then we have the compiler cicc, which takes the C++ code that is to be executed on a GPU and converts it to ptx.

Then the ptx code, we have a ptx assembler. So, the ptx assemblers job is to take the ptx instructions and kind of generate the machine code which is ptx machine code and generate a .cubin file, which is essentially a GPU binary file, but of course, the isa is ptx.

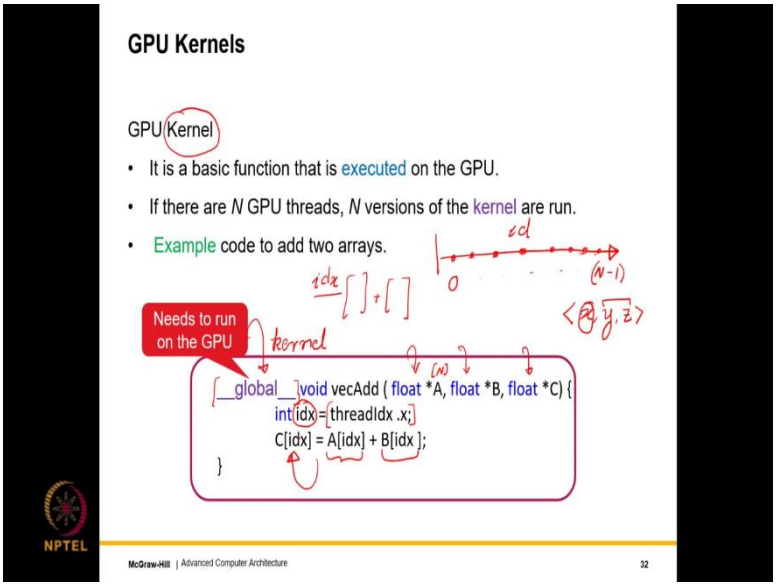
So, then the object file and the ptx they are combined into what is what we have been calling a fat binary. And along with that once the GPU code has been generated references to it via stub functions are passed to the C++ code, such that it can link to the GPU functions, it knows how to call them? And all the additional internal plumbing that is what we call it programming language parlance is automatically added.

So, the key point is that there are two parts, there is the C++ compilation part, which primarily incorporates the effect of stub functions and there is a GPU part where what we do is, that we take the preprocessed macros, generate ptx, generate ptx binaries and

combined both into a fat binary and that is kind of what is distributed. So, these dotted lines are kind of references which we have discussed.

So, once a fat binary is generated, we are ready to execute it. When we execute it, the runtime will basically figure out what to run on the CPU, what to run on the GPU. And furthermore some JIT compilation is done to convert from PTX to SASS and do many GPU specific optimization; these are very important, many of these GPU specific optimizations have to be done.

(Refer Slide Time: 09:38)



So, now let us come to the code that runs on the GPU. So, we will gradually work on this example and keep on expanding it. So, the aim over here is to divine, define a function that will execute on the GPU; a function that executes on the GPU is known as a kernel. So, this is known as a GPU kernel, it is a basic function that is executed on the GPU; if there are N GPU threads, N versions of the kernel are run.

So, basically this is something that is meant to execute in parallel. So, think of it as a parallel function. So, I am showing the example code to add two arrays. So, let us look at this step-by-step, piece-by-piece. So, whenever we have the identifier underscore underscore global underscore underscore.

This basically refers to or their directive which tells the CUDA compiler that this is basically a kernel and this is supposed to execute within the GPU and that too in parallel.

So, the name of my function is vecAdd and does not written anything.

So, it takes three floating point arrays as input A, B and C. So, these are pointers. So, these are pointers to which memory space we will discuss that later but if it is running in the GPU, think of it as a pointer to the GPU memory space, but all of this will be described later in a reasonable amount of detail.

So, what we do over here is that, let us say in this case; but we are arranging all the threads in a one dimensional access. So, let us say if there are N threads, so we assuming a simple case whether N threads and each of these arrays have N elements that is the simplest case.

So, we are assuming that all the N threads are conceptually arranged or numbered along a single x axis and each of them has a number from 0 to (N – 1). So, this is a standard technique that is used in almost all parallel programming, where every thread has a number. So, in almost all other frameworks, every thread has a unidimensional lumber; but in the case of GPUs, every thread is assigned a multidimensional number or id.

So, we are first looking at the unidimensional case, where the id can be between 0 and (N – 1). So, in this case what we do is, we get the id of the thread by reading the variable thread idx. x. So, I have already indicated to you that, we can have a multidimensional id for a thread; which means that there could be three components x, y and z. So, we will maximum go till three dimensions.

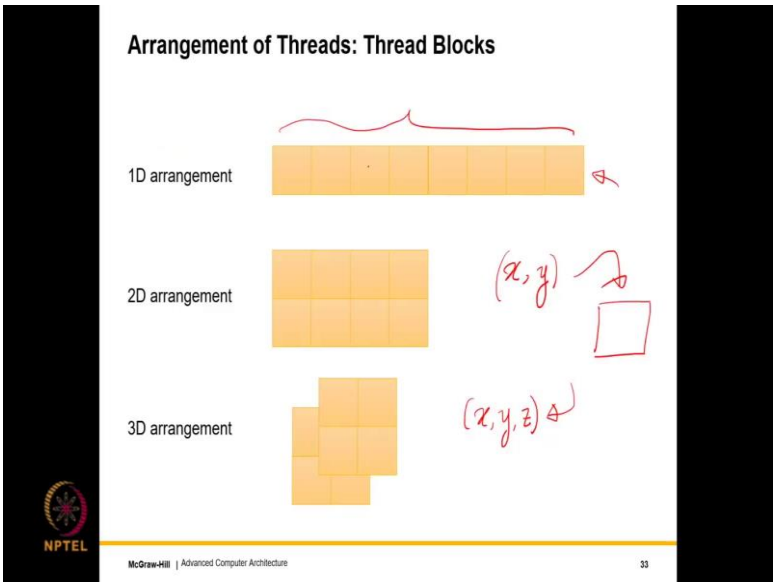
So, in this case, we do not have any y and z component. So, we will see couple of minutes, how that is ensured. But let us assume we have no y and z components, you only have the x component; so the x component varies from 0 to (N – 1). So, every thread gets its number. And once it does that, it just does the addition that is assigned to it; because what is the task, the task is to take two arrays and add them element by element.

So, if this is the problem of adding two arrays element by element, we simply take the index. And we find the relevant index. So, we find A idx the value from the idx index for A similarly for B, the add them and assign them to the corresponding entry for C. See if the array has let us say thousand elements and we run a thousand such threads in parallel, then of course we are done.

So, as we have discussed the global directive makes it run on the GPU and the rest we

have discussed in fair detail.

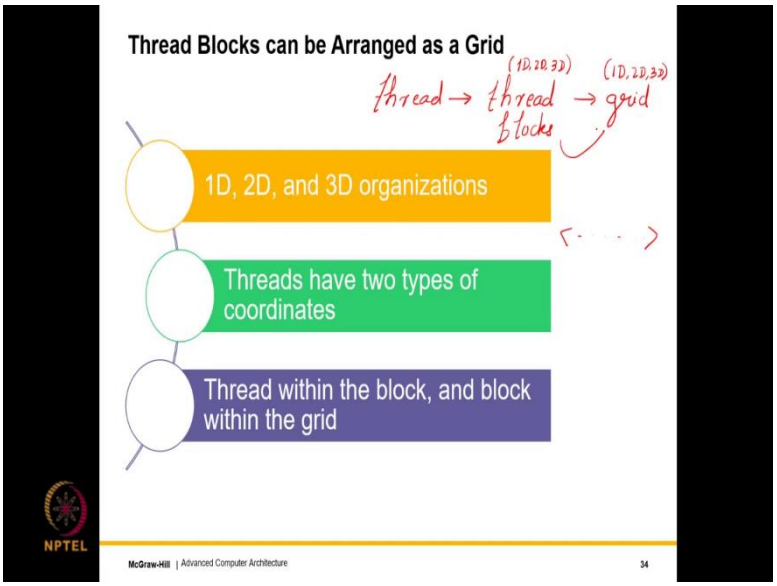
(Refer Slide Time: 13:57)



So, as I discussed in the last slide threads, GPU threads have an arrangement. So, if we want, we can arrange them linearly on a single axis as we just saw in the last slide. We can have a 2 dimensional arrangement, where every thread will have an x coordinate and a y coordinate or we could have a 3 D arrangement where the threads would have an x, y, and z coordinate.

So, all these three arrangements are possible with their advantages. But an advantages that look if we are working on volumes. So, let us say we are looking at an airplane wing which is a volume, then it this arrangement kind of naturally makes sense. If you are working on a 2D plane like solving the heat equation on a 2D plane, then maybe this arrangement makes sense. And for our simple example of adding to arrays, this arrangement makes more sense.

(Refer Slide Time: 14:53)

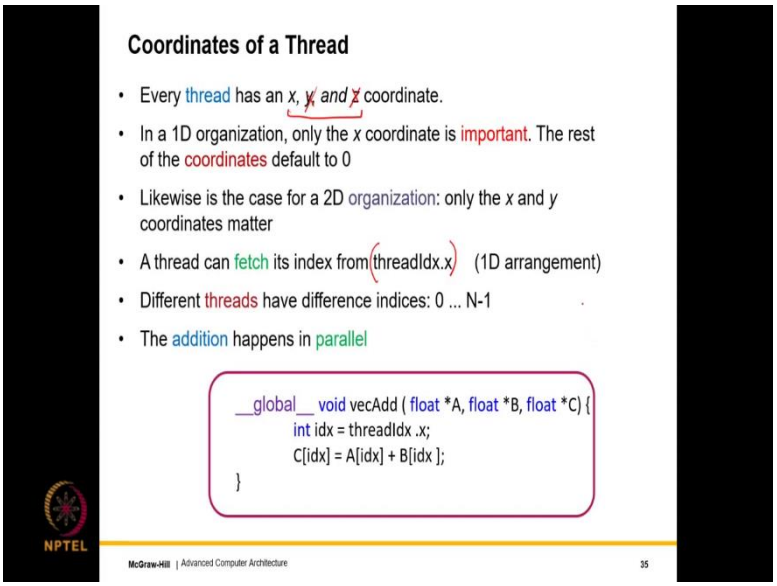


So, what we have done is that here we create this, this arrangement is a thread block. So, we have a 1D, 2D and 3D organization for a thread block and thread blocks themselves can be arranged in a 1D 2D or 3D fashion. So, that is known as a grid. So, where did we go? We had a thread, threads are grouped into what are called thread blocks. And thread blocks are again arranged in a grid.

So, again for thread blocks we have 1D, 2D and 3D; for a grid we have the same thing 1D, 2D and 3D same thing, exactly the same thing. Threads have two types of coordinates. So the first coordinate is the coordinate of the thread within the thread block, and the second coordinate is the coordinate of the thread block within the grid. So, thread within the block and block within the grid.

See if it in think about it, the universal coordinate of a thread is actually six numbers. So, first x, y and z is the coordinate within the thread block and a second x - y - z - is the coordinate of the thread block within the grid.

(Refer Slide Time: 16:13)

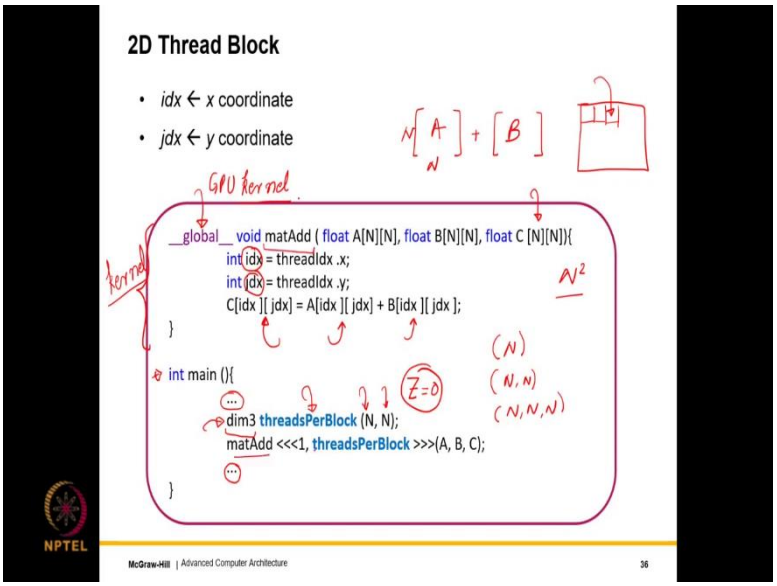


So, now let us come to the coordinates of a thread once again. So, this is an important concept and this is where the CUDA language differed somewhat significantly from its predecessors, which were traditional parallel programming languages such as open MPI and open MP and MPI.

So, which is that every thread was assigned at least a 3D coordinate within its thread block. And of course, we may choose to ignore a few of these coordinates in a sense set them to 0. For example, in us in array addition example x and y were set to 0, they are defaulted to 0; likewise, we can have a 2D organization, so this is possible to do.

In a 1D arrangement, our thread was fetching thread idx .x; but it could have also fetched thread index thread id x.y or z as well. The addition needless to say is happening in parallel and this is where the power of the GPU comes in the sense we create a lot of threads and the GPU does a fantastic job in scheduling and managing them.

(Refer Slide Time: 17:26)



So, let us now look at a 2D thread block and we will add some more code. So, we have kind of been stingy in showing our code but in this case we will add a little bit more code and show you more of it. So, in this case, we will consider a 2D thread block. So, the two coordinates are idx and jdx; idx is the x coordinate, and jdx is the y coordinate.

So, what we are doing, so we will discuss the main function later. So, incidentally now we have also gone into the C++ part, where we will discuss that later. So, let us again take a look at the kernel, which is this part.

So, in the kernel we have the underscore global directory, which again tells you that it is a GPU kernel. Then we have matrix addition. So, in this case what, we want to do is we can we want to take 2 2D matrices and we want to add the 2D matrices. So, we want to do a mat add a matrix addition of both of them.

So, let us say we want to add matrix A and matrix B and produce matrix C. So, since we are arranging the threads the 2D block and for the sake of simplicity let us assume that, there are N X N threads; the size of the thread block is N X N and the size of the matrix is also N X N. Well, then there is no problem; we get the x coordinate, we get the y coordinate.

So, we read thread id x. x, thread id x. y and then what we do is; we just add corresponding elements of both the matrices, similar to what we had done for arrays. As you can see it is

A idx jdx B idx jdx, we add them and we get C idx jdx as simple as that. So, of course, in this case we have N^2 parallel threads and they are transparently managed by the GPU.

So, of course, you can argue that, there are limits to the number of threads, in fact there are. So, there is a limit to the number of threads you can have in a thread block, but N can be very large. Of course that is fine, but then for a large N that is not a problem, we can take a large matrix and kind of break it into small tiles and assign each tile to a thread. So, that is easy to do.

Now let us come to the main function; we are seeing the C++ part first, so the dot . . indicates the code that we are not showing. So, gradually these dot dot dots will start opening up; but let us take a look at some data structures at this point. So, we want to create a data structure that will tell us the threads per block. So, we will define a variable threads per block.

This is of type dim 3, dim 3 means dimension 3, which means at the most it can contain three numbers; even though it is not restricted to you, it is not limited to containing only three numbers, it can contain one, two or three numbers. So, the number that is not initialized will default to zero.

So, in this case we are the z coordinate is defaulting to 0, that is the reason we are passing on the two arguments. And so, the argument that is not passed and assumed, that access is assumed to not exist. So, you are passing two arguments, which means N coordinates on the x axis, N coordinates on the y axis and total N^2 threads.

So, this is of object dim 3 and of course, it has many constructors; we can take a single argument, this would mean just N threads on the x axis; N , N would mean N threads on the x axis; I mean N X N threads basically on x or x and y axis and N N N would mean N cube threads x, y and z axis.

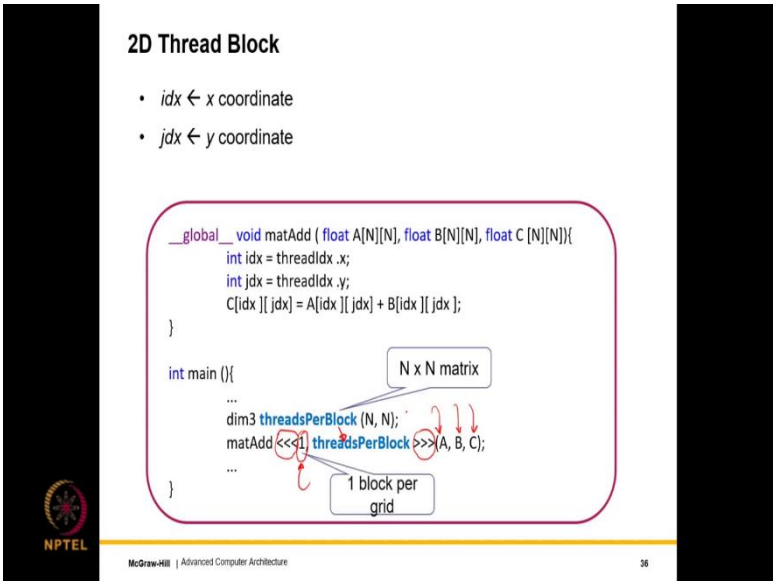
So, now what we do is, we call we invoke the mat add kernel over here and invoking the kernel has a special syntax. So, to highlight the special syntax, I will erase all the ink on the slide. So, we will have these two markers, kindly pay great attention to these two markers. So, these markers are essentially indicating the configuration in which we are executing.

So, the first is the structure for the arrangement of thread blocks within the grid; if this is 1, it means that look there is only one thread block per grid, so you need not bother, no problem. And the second one is the data structure that we just initialized, which is this one, which is saying we have N X N thread threads per thread block. So, the single thread block with an N X N arrangement.

So, this arrangement is very important and this is how the GPU initializes the kernels and the threads. Once that has been passed, then we send the arguments. So, the arguments in this case, well the CPU basically has to pass pointers to GPU memory. We will see that that is possible to do, but we have not shown the code, but the code is coming.

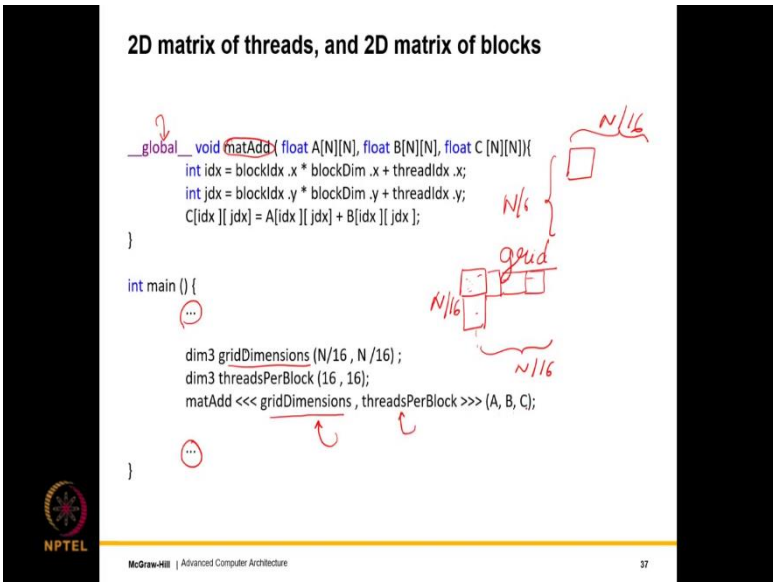
But the important take home point over here is that, you need to understand that the arrangement has to be specified within the 3 less thans and 3 greater thans. The first is the arrangement of thread blocks per grid, which in this case is defaulting to 1 and threads per block.

(Refer Slide Time: 23:13)



And we are creating basically an N X N matrix and 1 block per grid.

(Refer Slide Time: 23:23)



Let us now look at a 2D matrix of threads and 2D matrix of blocks. So, we will look at something quite similar. So, we will start with the main initialization, the main function; again we are not showing the code over here. So, we will define two data structures, two dim 3 data structures, the first one is the dimensions of the grid, because in this case what we want to do is, we want to arrange the thread blocks themselves in a grid and we want to do it in a 2D grid.

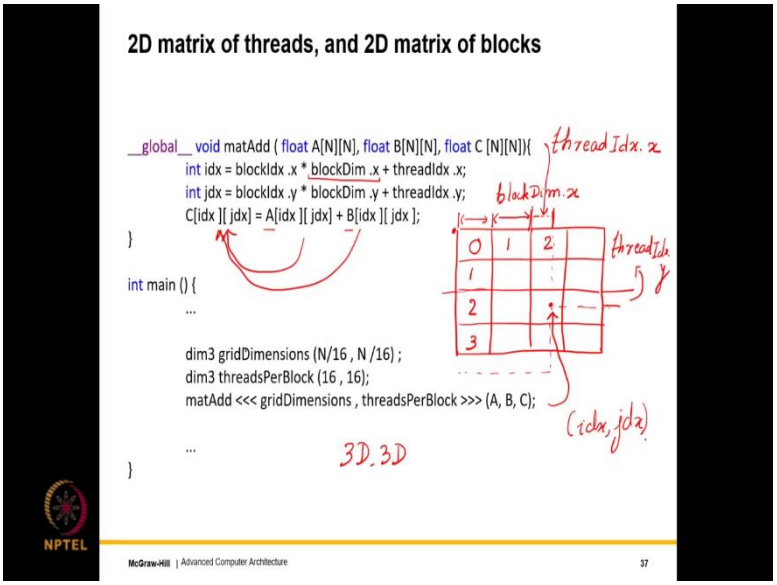
So, we will have N / 16, where N is the parameter that we have defined earlier right within these three dots. So, the dimensions of the grid are N / 16 X N / 16. So, we have N^2 / $(16)^2$ thread blocks within the 2 dimensional grid. So, this goes as the first parameter to the kernel invocation.

So, as you can see the kernel is over here, its name is matAdd and the first parameter to matAdd that is going is actually the dimensions of the grid. After that, we define the arrangement of the blocks within the thread block. So, the threads per block are arranged as 16 X 16.

So, we have 16 X 16 threads per blocks; if you actually look at it, it is a 2 dimensional 16 X 16 structure and there are N / 16 such structures and again N / 16 such structures. So, the total is still N^2, but in this case we have divided it in a different manner; we have arrange them first as separate tiles kind of think about this, as a separate thread blocks within a grid and within each thread block of course they are arranged in a square fashion.

So, with this we give 2 dimensions or two dim 3 data structures, no defaults over here and the three arrays A, B and C. So, it is interesting to see how this information is processed within the matAdd kernel. So, within the kernel we compute two indices idx and jdx; but idx is of course, computed in a special way, the same holds for jdx as well. So, to understand let us draw the great and a better fashion.

(Refer Slide Time: 26:00)



Each of these squares here is a thread block. So, what we do is that, we first take a look at this parameter, which is the block dim. x is the block dimension. So, it essentially says that, what is the size of a thread block in its x axis, so the block dim .x is basically this quantity over here. So, it is basically this quantity over here. Say if let us say the block index is 2, then this would be the first block, this would be the second block, this would be 0th block, first block, and second block.

Say the block index is 2 what we do is, we multiply two times blockDim . x, which is exactly what you see here; blockIdx . x times blockDim . x, so we multiply two times the block dimensions. And so, then we get the starting point of this. And then we add the thread index . x, which is this quantity over here. So, this quantity over here is a threadIdx .x. So, this quantity here is the thread index.

And so, this is how we get the x coordinate, we are assuming the x axis goes this way. So, this is how we are getting the x coordinate of the thread, which is exactly this point. So, the x coordinate is coming how? We take the block dimension multiply with the block

index, of course the x coordinates of them and we add the thread index. So, this is how they exactly come here. And let us say the actual thread is over here, we do the same; this time we go down the y axis, we are assuming the origin is over here and again these are numbered 0, 1, 2, 3 and so on.

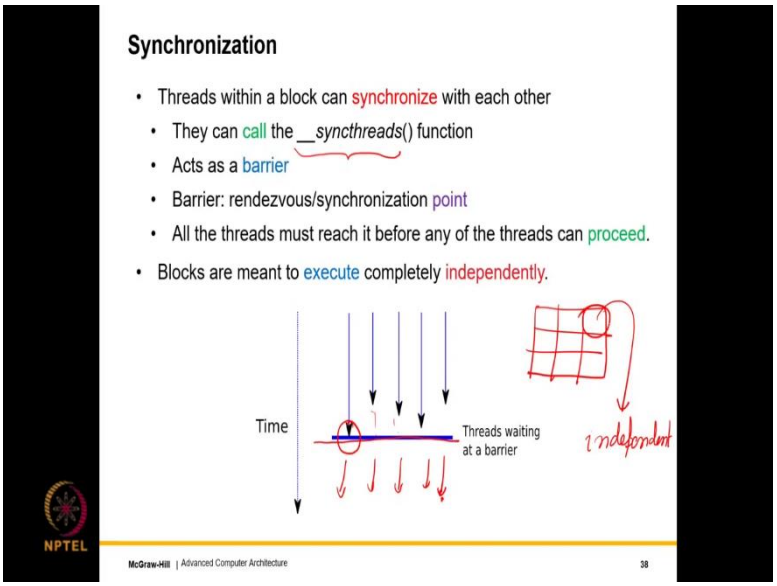
So, we do the same, for y also we multiply with the y-th component of the block index multiplied with the block dimensions. And then what we do is that, we arrive at the starting and then this displacement, this displacement is threadIdx. y. So, this displacement over here that you are seeing is threadIdx. y. So, that is how we get the coordinates of this thread over here on the grid, which is basically idx and jdx,.

So, the computation of the coordinates of the thread is slightly more difficult over here as compared to what we have seen earlier, but that is ok because in this case, we have a hierarchical organization, we have we are organizing or arranging threads within a block and we are also arranging blocks within a grid. So, because of that we have this two level thing.

And. So, what we are doing now is that, we simply take the components of A and we take the components of B and we add them and the moment we add them, we get the equivalent component of C. So, this is CUDA programming and all of it is glory. So, of course, we here we have considered a 2D matrix of threads and a 2D matrix of blocks; so we could always make it 3D and 3D. So, this would slightly become, this will become slightly complicated but to a certain extent that is ok.

So, we will then know what is to be done. So, what we will do in that case is that, we will also have to take care of the third component, which is also the z axis; we will have a similar formula and we can work out.

(Refer Slide Time: 30:36)



So, then if you have so many threads, how do they operate? So, how are they synchronize? So, that is an important question, because the point is that if let us say that there is a small computation; in a large computation we would want all the threads to arrive at a barrier, which is basically a common point.

So, all the threads arrive over here and once all the threads have arrived over here; so here you can see one thread is fast, it has arrived quickly, but the other threads are lagging. But once all of them arrive at the barrier, all of them can be released at once and they can go. So, this ensures that if thread is over here, it has some information about the rest of the threads that they have actually crossed the barrier.

So, the barrier can be thought of as a rendezvous or a synchronization point and all the threads must reach it before any of the other threads can proceed. And there is a CUDA function called the syncthreads underscore underscore syncthreads function. So, in fact you will find a lot of CUDA functions with underscore underscore syncthreads. And so, this is this can be used to implement a barrier.

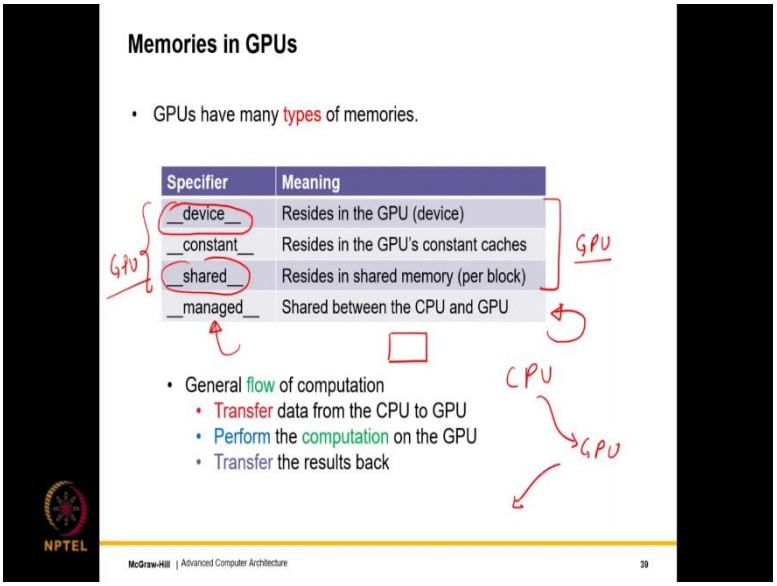
So, the idea over here is that, blocks thread blocks are meant to execute completely independently. So, the way that we take a computation is that, of course we divide the entire computation is a grid; we divide the grid as we have seen into thread blocks, each thread block is meant to be completely independent of the other thread blocks.

So, most of the functions of CUDA, sharing, synchronization, most of them, of course there are global ones which hold for the entire grid, but most of them hold within a thread block because the idea is that a thread block works independently of other thread blocks. And within a thread block, you do as much of fine grained synchronization as you want. So, the also the, this is in a key implication with the hardware and because that is the way the hardware will schedule threads.

So, in an interesting divergence due here; so what I am the way I am recording this video is that, I am using Camtasia studio. And so, then I record the raw video, but when the final video is produced it actually uses my NVIDIA GPU. And the NVIDIA GPU is usage you can see that entire laptop like they are having really hot and the GPU is used that the usage is between 80 and 90%; but if I do not use a GPU, it actually takes hours.

The reason that this completes so soon is only because of my GPU.

(Refer Slide Time: 33:28)



So, after that we recon verge here. So, we are discussing the computation part to a large extent; of course we have not discussed most of the nitty-gritties, but at least the basic idea has been provided. Now, let us look at the memories. So, GPUs have many types of memories. So, we have been primarily looking at the device memory.

So, again in a note these underscores on the beginning and end. So, this is a constant feature, it will be a recurring theme of the CUDA language. So, the device here memory

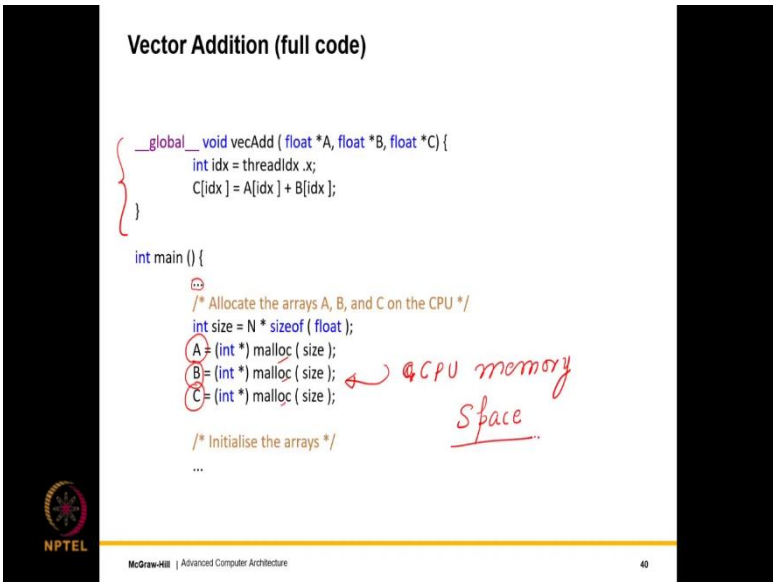
means, it resides in the GPU or the device. The constant means, it resides in the GPUs constant caches, shared means it resides in shared memory.

So, shared memory is something that is there per block, which is per thread block; we assign a we assign some memory which is shared only by the threads within a thread block, this is also within the GPU, all three of these are within a GPU. We have managed. So, the managed is an interesting memories is not really high performance memory space, all of these memory spaces.

But of course, this memory space is both a virtual space and a physical space, but both are within a GPU. The managed is a space both virtual right and it has a physical connotation, but let us look at it from the programmers point of view. This is shared between the CPU and GPU in the sense that, GPU pointer can be used in the to access CPU memory and vice versa as long as it is there within the managed space.

So, you can think of this as kind of like a shared memory between CPU and GPU. So, there is a general flow of computation when we transfer data from the CPU to GPU, perform the computation on the GPU and the results come back.

(Refer Slide Time: 35:23)

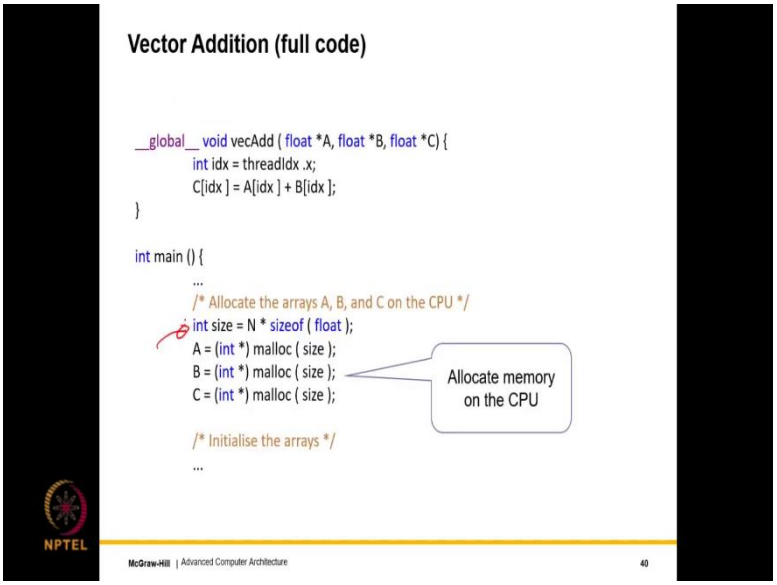


So, let us now look at the full code for vector addition. So, we have already seen the kernels over here when we take three arrays and we add them. So, given the fact that I have seen this, I am not discussing this; but in the main function, I will still have these three dots, but

I will still have I will show you more.

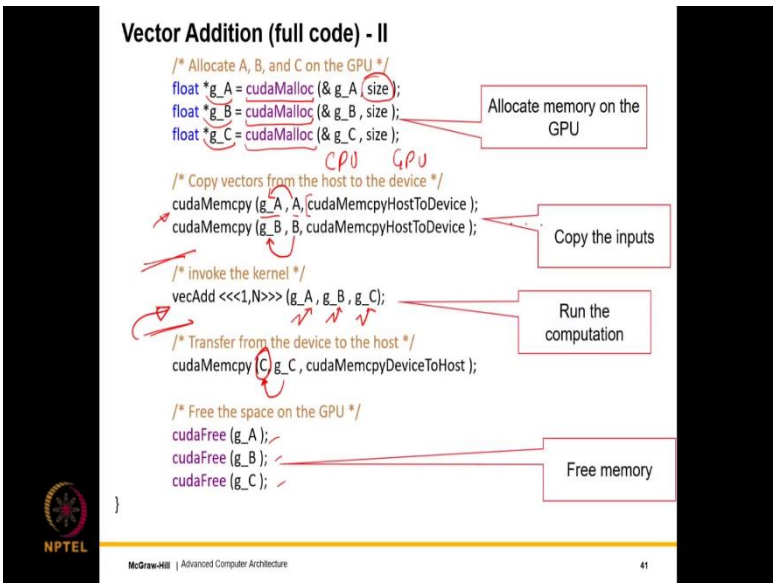
So, what we can do is that, let us first initialize the arrays A, B and C on the CPU. So, these are floating point arrays with N entries. So, we just use three malloc calls and then we initialize them and we have three arrays A, B and C. So, mind you they are in the CPU memory space.

(Refer Slide Time: 36:23)



Then we initialize the arrays. So, we allocate memory on the CPU, initialize the arrays.

(Refer Slide Time: 36:32)



Now, here is the fun part, the fun begins here that we allocate memory on the GPU and we get a pointer, which is as such useless on the CPU; but it is essentially pointed in the GPUs memory space. So, instead of the malloc function, we use the cudaMalloc function; the cudaMalloc function that we have over here what that basically does is that, it allocates an array on the GPU. And so, basically it is again of type float star and the size is specified over here. So, recall that we computed the size over here.

So, we create 3 such arrays on the GPU and once these arrays are there on the GPU we get the pointers; but of course as I said they are useless on the CPU, but they can be used to copy these vectors from the host to the device. What is the host? The CPU, the device is the GPU.

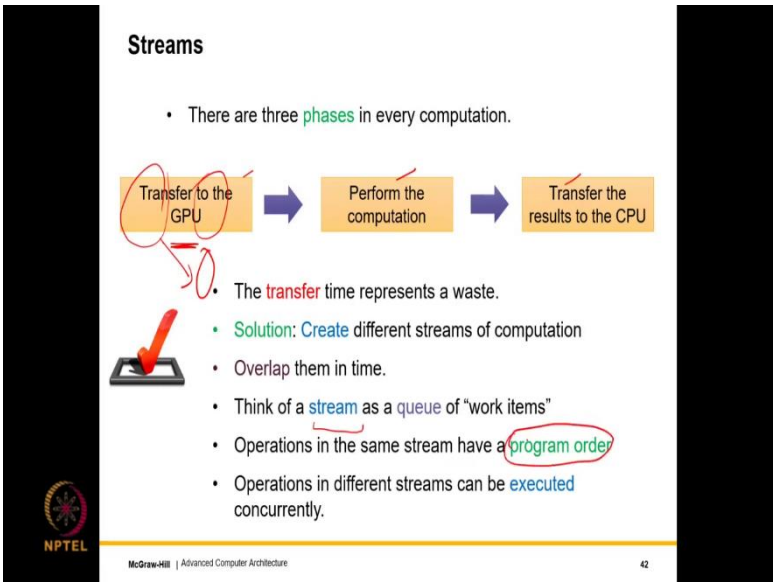
So, what we do is that we do a memcpy. So, first give the GPU pointer, then the CPU pointer. We then provide this flag; to which function; To the cudaMemcpy function, whose job is to transfer data between the CPU and GPU. We provide this flag cudaMemcpyHostToDevice, which computes data, sorry which copies data like this and then we launch the kernel; we have already seen this, previously the center part was under dot dot dot.

So, now, we provide the pointers, which are the GPU pointers; of course they cannot be the CPU pointers. So, we had done little bit of cheating over there. So now, we are not doing it I promise. So now, we are sending the GPU pointers to the arrays and we are launching the kernel.

Once the kernel is done, it will finish its execution. So, this is the only GPU part and this is what exactly the stub function will refer to in the compilation process. Finally, we transfer from the device to the host, no problem we do that. So, from the device you transfer it back to the host. So, which means the host the CPU has the contents in the array C.

And we frees the space on the GPUs. So, we do cudaFree g A, g B and g C, which basically. So, whenever we have a malloc, we should be doing a free this is board programming practice. And we do that in the sense we free the three arrays on the GPU, g A, g B and g C. So, just a quick recap, allocate memory on the GPU, copy the inputs, run the computation, free the memory.

(Refer Slide Time: 39:24)

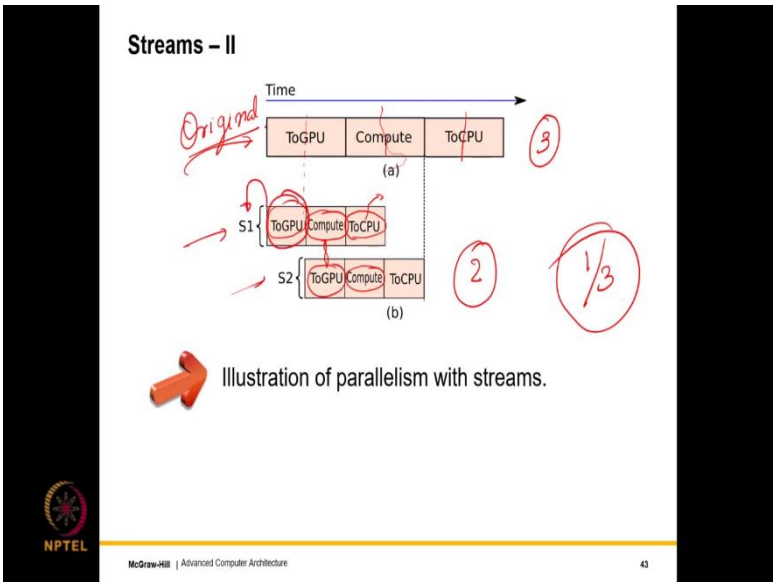


So, what we have seen is that, there are three phases in every computation which is that we transfer data to the GPU, perform the computation on the GPU, and transfer the results back to the CPU. Think about it this transfer time over here is kind of a waste, in the sense that we are not doing any active computation.

So, solution is that, we create different streams of computation and we overlap them in time, such that at least the GPU can be used. So, what we can do is instead of transferring all the data at once, we transfer small chunks of the data and then we perform a computation on them. At the same time in a pipeline fashion, we transfer the next chunk. So, instead of like transferring all the data, we pipeline the computation.

So, of course we cannot do pipeline at the, pipelining at the level of bytes. So, what we do is that, we have a bigger granularity, we create a stream as a queue of work items, operations in the same stream are executed in program order. However, operations across streams may overlap, so there is no synchronization across streams per se.

(Refer Slide Time: 40:42)

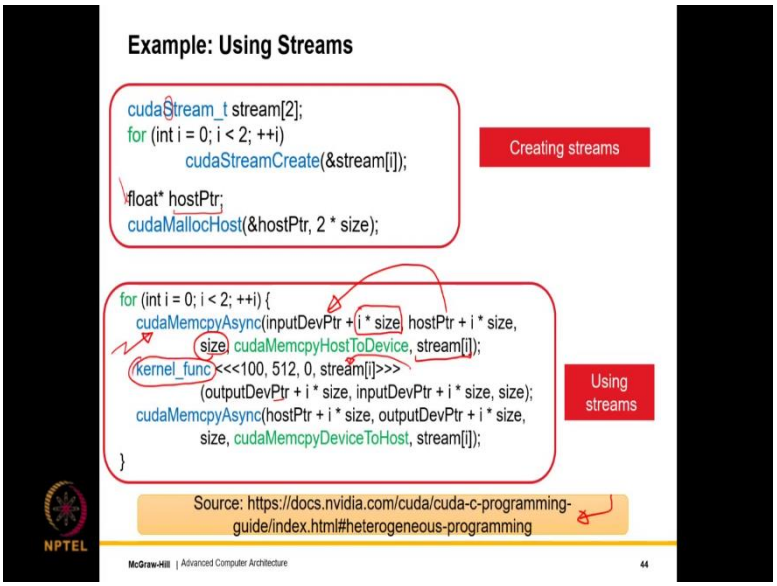


So, what I want to say is clearly visible over here, let us say that this is the original computation. See in the original computation, we transferred the data to the GPU. We do the computation; it goes back to the CPU. So, instead of that, let us do one thing. Let us transfer half the data to the GPU and then let us do the computation at the same time we transfer the other half, so you can clearly see the overlap here.

Then we transfer the results back to the CPU; at the same time the computation happens for the second stream, so this is stream 1, this is stream 2 and then we transfer it back to the CPU. So, what we can see is that, because of the higher degree of overlap instead of 3 time units, it takes 2 time units. So, we have a straight forward speed up of one-third. So, this is quite amazing, because we did not do anything, all that we did is we divided each of these into like 50%.

And what we basically did is that, after dividing them into 50%, we achieved an overlap. So, once we achieved an overlap, we were able to reduce idleness or wastage of time and this was done by having two streams, by having two separate streams this was achieved. So, this is an illustration of parallelism with streams.

(Refer Slide Time: 42:31)



Needless to say, streams are a method for improving the performance. I will now show you the CUDA code for streams; but of course, I will not go to the level of detail that I went in the previous case and, but you can refer to NVIDIAs official documentation shown here. In fact, the code is from there, the code is not mine; I am not claiming originality for this, there is the core is there is and I in good faith I am explaining it.

So, how do we create stream? Well that is simple. So, we have this cudaStream underscore t type. So, kindly note this nomenclature that NVIDIA is using. So, every conjugative word let us say cuda starts with a small c, but then S starts with a capital letter and underscore t denotes a type.

So, cudaStream ‘t’ the stream types, it creates a two streams. Then what we do is that, we have a for loop, does not go very far just 2. So, we create 2 streams. And also what we do is that, we create one floating point array and we just do a cudaMalloc on the host.

Now what we do is that, for both the streams; if you can see the for loop, this goes from 0 to 2, which is for both the streams, we do a cudaMemcpyAsync. So, cudaMemcpyAsync is just an asynchronous memory copy; which means that this is not tied to other actions. So, this is essentially what gives the streams the independence.

So, I am not showing the rest of the code, because the rest of the code is shown over here. Well, basically what I am trying to tell you is that, we are doing a memcpy from the host

to the device. So, this is the device pointer and the offset on the device pointer is i * size. So, i initially i = 0, so this is just going into device pointer. So, recall that we are the host pointers we have already malloc it and its size is two times size.

So, the first chunk gets transferred like this, the first set of size bytes get transferred from the host to the device. But of course in this case, the operation of transferring data from the host to the device is assigned to stream i. So, stream i initially will be 0 and 1; but let us call it stream i, this is the part of the stream.

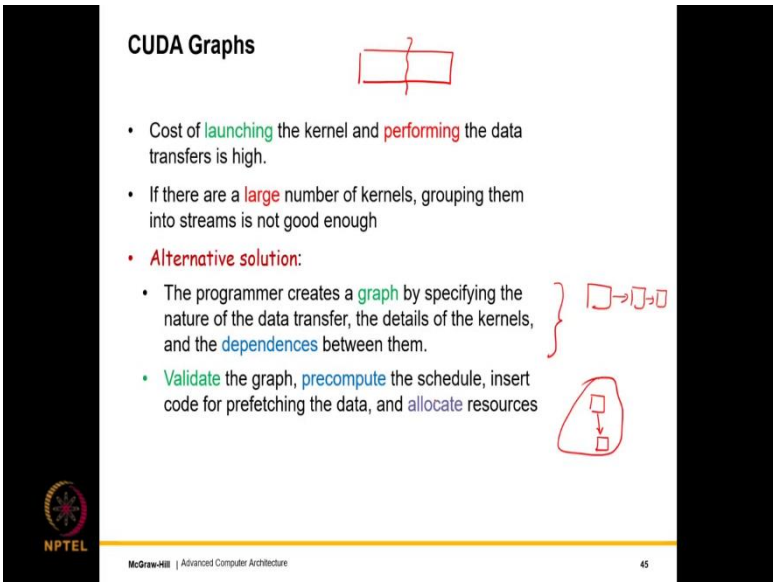
So, as you can see over here, these transfer operations are to be explicitly assigned to streams. So, this is being done right at the time of function invocation; that we are seeing that look you transfer it, but transfer it as a part of this stream. Then what we do is we invoke the kernel and in the kernel we are just a simple one dimensional kernel but what we are doing is that, we are also assigning the kernel to a stream.

So, every call basically is getting the stream id along with it and then of course, we are sending other arguments which are specific to the kernel. So, we are not worried about that. But the important point is that the kernel function is taking the stream id as an argument. I will slightly reduce the clutter and then what we have over here is, again we have a copy, which is the copy from the device to the host.

And again as you can see, we have the host pointer and output device pointer. But the most important thing is that, even this operation is being assigned to the stream over here. So, we are assigning all three operations to this stream and these are not synchronous, in the sense that they are not tying up the GPU.

And it is just that we are telling the stream that look these are the three operations you need to perform, within this stream as you have seen as we have just defined over here; that, operations in the same stream have a program order; which means that within the stream will have a program order, but across streams of course we can relax the order. So, this is captured by the async directive over here.

(Refer Slide Time: 47:02)



Let us now introduce an even more elaborate construct known as a CUDA graph. So, what we realize is that, the cost of launching the kernel and performing the data transfers is high; that is the reason it is a good idea to divide let us say large tasks into smaller ones and schedule them. See if there are a large number of kernels, we grouped them into streams, which did give us some benefits, but that is sadly not good enough, we can do better.

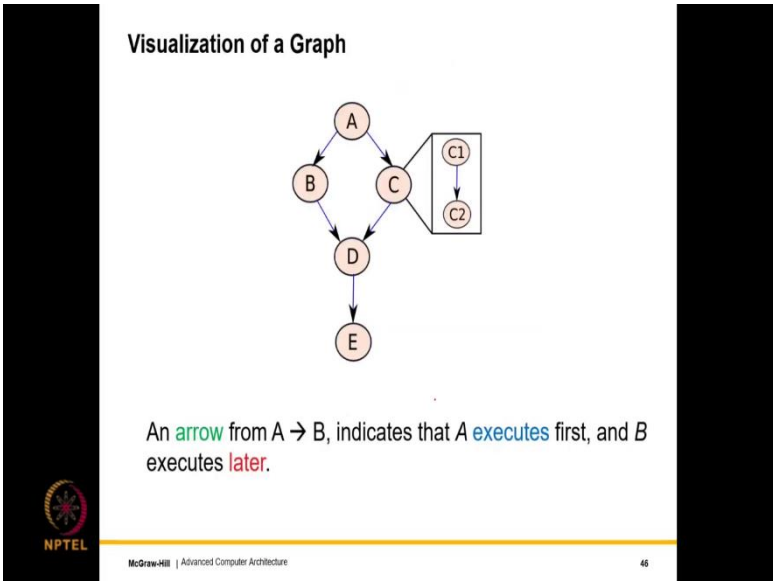
So, an alternative solution, particularly if you have a large number of tasks with a fair amount of heterogeneity is to use a CUDA graph. So, this will be used or is typically used to specify the nature of the data transfer, the details of all the kernels and the dependencies between them. So, you basically specify everything the same way we did in a stream; but of course a stream is like a link list and a graph is more like a tag or a directed acyclic graph.

So, then we have an automatic engine which does all of this that, validates the graph; pre computes a schedule for it, schedules all the activities in the GPU, on the GPU. And it let us say there is a dependency between one task and the next; then needless to say, the first task executes first and the dependent task executes after it.

So, unlike program order, we have a task dependence order and that is respected. It has more, it is way more intelligent than that. So, basically it adds more to prefetch data. So, when the previous task is running, some code is added to prefetch the data for the next task and of course, allocate resources; particularly if we have multiple such graphs running on

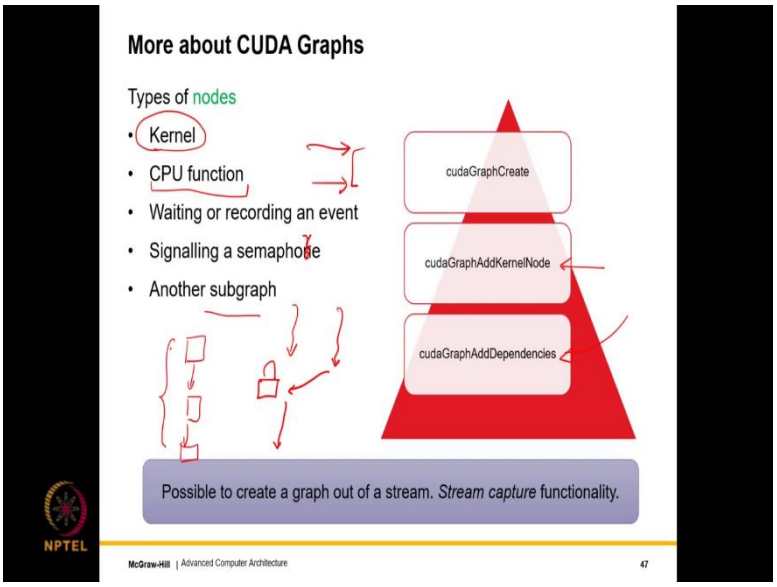
multiple applications running as is possible and modern GPUs that, we schedule resources appropriately.

(Refer Slide Time: 49:13)



Visualization of a graph; well we can have a graph like this, each of them has a task, and each node in fact can be a subgraph on its own, it can be a subgraph. So, an arrow from A to B indicates that, A executes first and B executes later. So, the arrow from let us say task A to task B indicates the dependence.

(Refer Slide Time: 49:39)



So more about CUDA graphs, well CUDA graphs can have all kinds of events. So, we can

have different kinds of nodes. So, we can have kernels of course; along with kernels which are GPU functions, we can also have CPU functions. We can wait or record an event; so what is an event I will discuss in the next slide. But an event is basically something like a timing event or something like a performance counter, where we say that look start recording the time from this point and end over here.

So you can understand how long an execution took. Then of course, signaling a semaphore. So, signaling a semaphore is basically a signaling mechanism, where it is a more sophisticated version of locks and unlocks, where the idea is that if one thread is kind of waiting here on the lock, another thread can come here, it can signal it to unlock and this thread will start. So, that is signaling a semaphore.

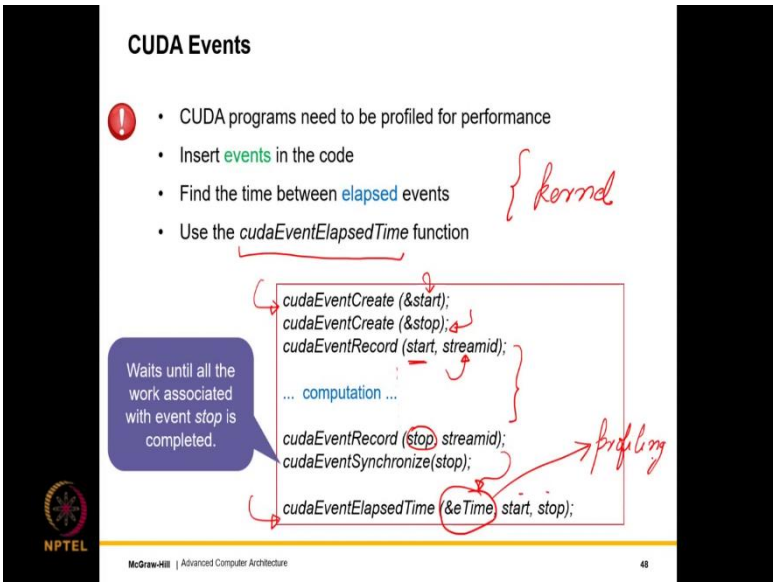
So of course, locks can have two states, locked and unlocked; but a semaphore can have multiple states, so it is a generalization of a lock where a lock is called a binary semaphore, but semaphores may not be binary. For example, we can have a counting semaphore, where let us say anytime we signal it, we increment it; anytime we wait on it, we decremented and we always want that the count of the semaphore more than 1 to release a thread.

So, this is an a OS concept, but do you know this can be looked at. So, you can think of this as a generalization of a lock. And of course, a node can contain another subgraph. So, these are the main functions that we have CUDA graph create, see the same naming scheme, add kernel node, similarly, we can add other kinds of nodes and add dependencies. So, we can add other; we can add dependencies between all of these nodes, so that will create a complex graph.

And as we have discussed, it is a hierarchical recursive structure, where one graph can be subsumed within a node. It is also possible to create a graph out of a stream. So, this is known as a stream capture functionality, where we take an existing stream and we create a graph out of it.

Then we get the additionals of this advantages of a graph, but given the fact that a graph is a heavyweight thing, in the sense that a lot of additional code including prefetching and validation etcetera is added along with a graph; it is typically preferred when we have a fair amount of computation, which is reasonably heterogeneous and unstructured.

(Refer Slide Time: 52:31)



So, finally, as we had promised, we come to events. So, CUDA programs have to be profiled for performance. So, we need to insert events in the code, find the time between elapsed events. So, that will tell us for example, how long a kernel took or how long let us say a piece of code took. So, we have a CUDA elapsed time function. So, then this will tell you how long it took. So, I will explain an example.

So, let us say we have this function CUDA event create, where the start event is created. And similarly, we have another event that we create called stop. We create these two events, start and stop and we pass their pointers, we start recording. So, we start the timer for the start event and of course this can be associated with a stream id. So, for that stream we start recording, the computation keeps going.

Then we again record one more event, which is called stop. And so, of course again for that stream id; so at this point what happens is that, we need to synchronize, which basically means wait until all the work associated with the event stop is completed. So, what can happen is the event itself processing the event is a multistage operation and so, we will at least want that all the work associated with the stop event, which is recording exactly at what point in the stream is at that completes.

So, that is why we do, we call the synchronize function, which is highly recommended; if you want to get an accurate time, which is also capturing a point in the stream a relevant point in the stream. And then all that we can do is that, we can call the function

cudaEventElapsedTime between the start and stop functions. So, that will be stored in the variable e time, e times for elapsed time. So, that will tell us between the start and stop functions, which captures all of this computation, how much time did it take.

This information is basically profiling information, which can be used by programmers and application developers to tune and make that code more efficient.

(Refer Slide Time: 54:55)

Good. So, we have finished the video on programming GPGPUs. So, we have completed the first two parts, we have a reasonably good understanding of the CUDA 10 language. So, as we speak in 2021, the latest version of CUDA is CUDA 11, that viewers are most welcome to look at.

In the next video, we will look at the design aspect of GPGPU, in the sense how are the GPGPU is designed and what are the design choices and also look at the design of popular NVIDIA GPUs in this space.