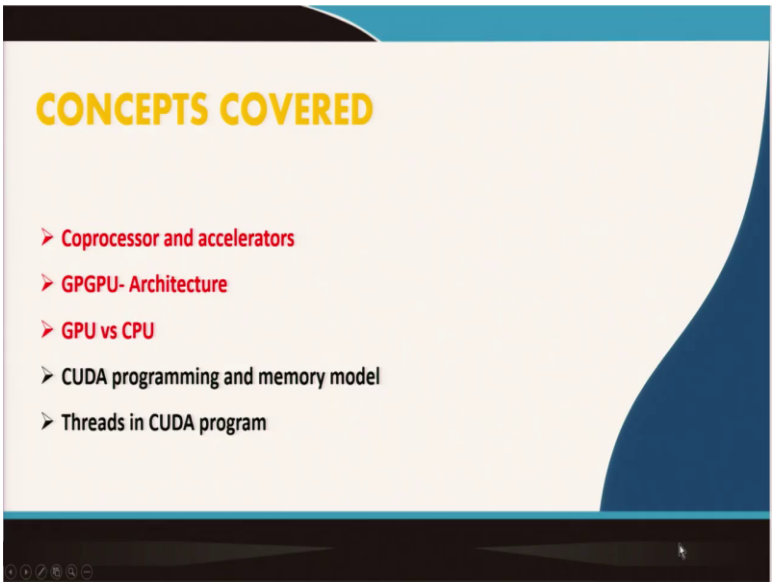
High Performance Computing for Scientists and Engineers
Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module – 04
GPU Computing
Lecture - 33
Introduction to GPGPU and CUDA (continued)

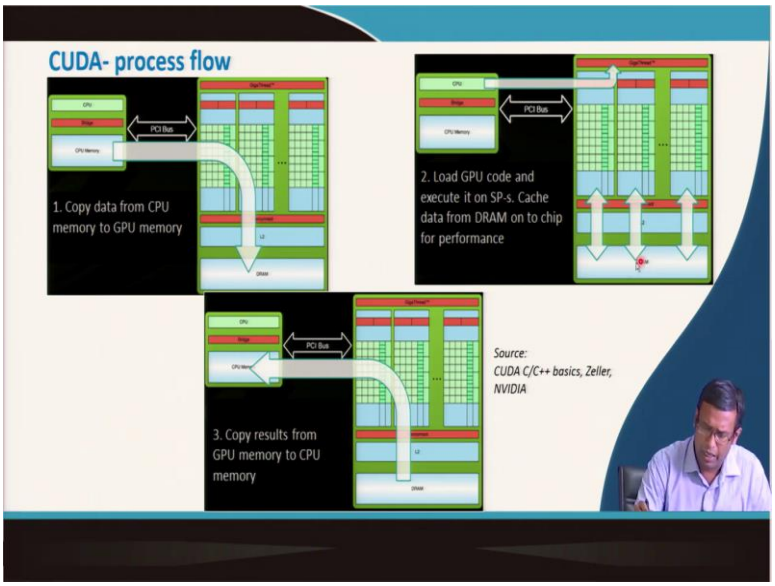
Welcome, we are discussing GPU computing in the course High Performance Computing for Scientists and Engineers and we are basically discussing the Introductory part on GPGPU and CUDA.

(Refer Slide Time: 00:42)



So, in the last few classes, we have introduced GPGPUS and their architectures and showed; what is the difference of the GPU architecture and CPU architecture and also the programming aspects and now we are looking into CUDA programming. CUDA is a programming model for GPUs developed by Nvidia; Nvidia is one GPU hardware vendor and we will discuss the CUDA memory model and threads in the CUDA program in this lecture.

(Refer Slide Time: 01:09)



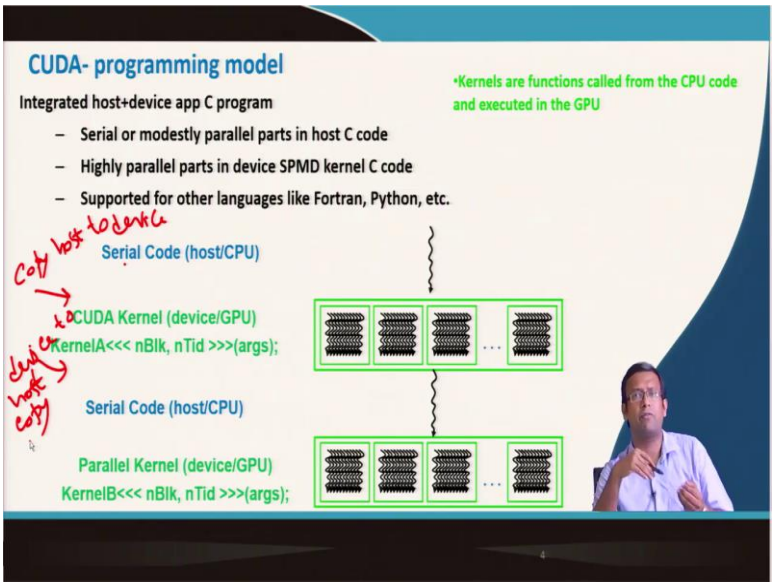
We are looking into in our last lecture how the process flows in CUDA. We understand that we as programmers cannot directly communicate with the GPUs. We cannot ask our programs to directly communicate and talk to the GPUs directly put the instructions on the GPUs.

Whatever programs we are writing that will be preliminary hosted by a CPU and CUDA will help the compiler to take part of that job and offload it into the GPUs.

Also, the memory of the GPUs is; not directly visible to the programmer. So, when he writes a program, he asks CUDA to take the relevant memory from the CPU ram and copy it to the GPU that CUDA uses. Then CUDA asks the GPU threads to be active when a kernel is launched. The kernel is a function which is launched by the main C program; CUDA enables C program here and that part that function should be executed in the GPU.

So, CUDA asks GPUs to launch threads to execute that kernel and once this is done, then the output data is written into the device stem or the ram of the GPU. Then CUDA copies this data into the CPU ram and now CPU ram data is readable by the programmer. It can be written to the hard disk or it can be written out to the screen. GPU RAM is not visible by the hard disk or by this or by the file systems. So, GPU ram data is device terms data is computed to the CPURAM. This is the work process flow in the CUDA.

(Refer Slide Time: 03:03)



Now, we will see who is the programming model. Any CUDA program is a host plus device application C program say C, it can be C ++, it can be FORTRAN or some other language Python, Java. So, it is basically a CPU language program with CUDA function calls and this is integrated with the host part of the program which will be executed by the host or the CPU and the device part which will be executed by the GPUs.

The serial or modestly parallel; modestly parallel pa rt means it can be parallel with 4 threads,8 threads,20 threads etcetera that you run in the CPU that is part of the host C code. We are discussing c code here, but it can be in java, Fortran or some other language.

The highly parallel parts where thousands of threads can concurrently work that will be following a single program multiple data kernel code that will go to the GPU part that is also like a c program. But it is called as a kernel function and there are certain seven syntax in CUDA by which this kernel function is identified that this will run in the GPUs and it is supported in for the language like Fortran and C.

So, again we will look into little detail that says this is the serial part which will run in the host or in the CPU. Then the CUDA kernel will be there. The kernel is called like the kernel name and in the number of blocks and number of threads and the argument.

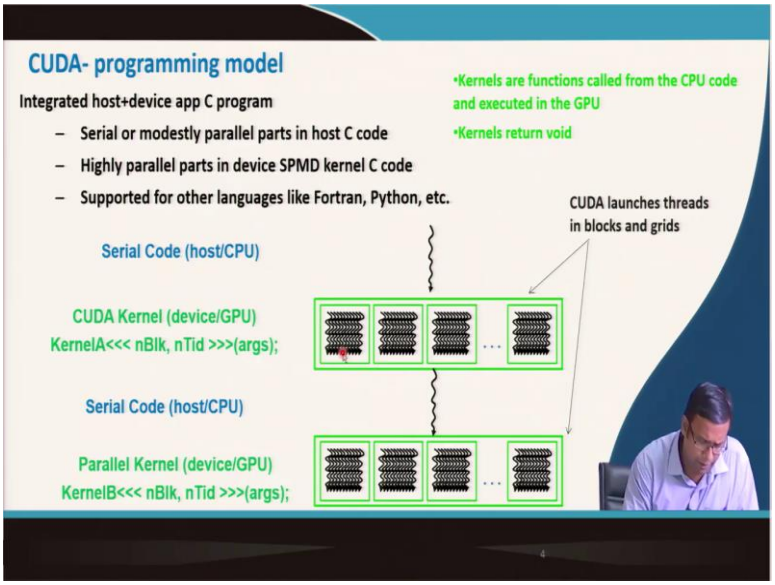
So, when the kernel is launched, multiple threads are launched. Block of threads are launched and inside each block there are certain number of threads that gives us the total number of

threads and when this kernel function is called, then this massively large number of threads parallel threads are launched

Again, once the kernel is over, then the program goes back to the serial mode. In this, there is a barrier of the threads and all the threads converge and only one thread which is acting only in the CPU that works. Again, if there is another kernel launch, then the program control goes to the GPU and GPU runs multiple threads there.

In between them, there are copy host to device; data is copied from CPU to GPU after the kernel functions are over, then device to host copy. The data on which the CUDA kernels are working; that data is copied from host to device after the CUDA kernels have done their job. Then again it is copied from device to host. So, in between these copying's are done.

(Refer Slide Time: 06:33)



These kernels are the functions that are called from the CPU code and executed on the GPU and kernels return void. GPUs cannot directly return any value to the CPU. Only the GPU kernels or GPU cores can write back to the memory of the GPU. Anything which can be read by the CPU has to be copied using CUDA from the device stream to the system CPU ram.

So, kernels, the functions that are being called by the main C program and running on the GPUs cannot give any output, cannot write anything back, cannot send anything any value back to the CPUs

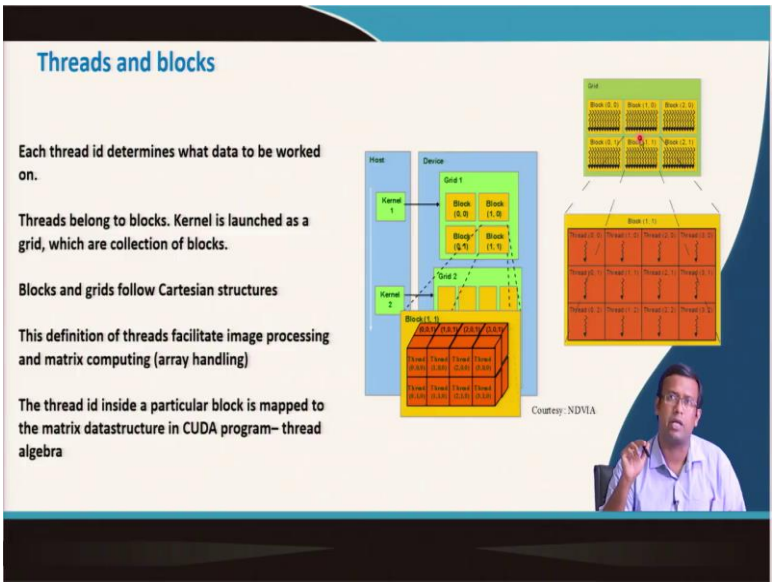
What are they doing? They are only writing in the GPUs. Therefore, kernels should always return void. What are they doing? In the kernel, they are updating some memory locations in the device time and this updated memory has to be copied back from the device to the CPU. Kernels cannot return anything; kernels will always return void. The output of the kernels will be in the memory of the device ram.

On the CPU ram CPU memory kernels cannot send anything back and this output which is in the device stem will be copied from the device to the host. That is the main part of a CUDA programming model. That first there is a copying from CPUs ram to the GPUs ram, GPU does certain work using multiple threads in different blocks and once the GPU our threads work is done, it writes back to the GPU memory and then CUDA copies the data from the GPU memory to the CPU memory and now programmer can access that data

Gpu memory is not directly accessible to us, we need something like CUDA which will copy the values from GPU memory to the CPU memory. Then CUDA launches threads in blocks and grids. So, the entire set of threads that is launched by CUDA kernel is called a grid. Inside the grid there are multiple blocks and inside each block there are a number of threads.

Why is this so? There is definitely a certain reason behind that, we will explore as we will learn it in more detail. But two important factors are that scheduling these threads to get the massive parallel thread core architecture of GPU is one factor. Also, another factor is that when we are writing CUDA programs, we are doing scientific computing types of jobs in which we are mostly interested in matrices. So, some mapping with the threads with the matrix rows that is also important

(Refer Slide Time: 09:25)



Each thread id determines what data to be worked on and this is a very important part in CUDA programming. We will later learn in detail about thread algebra that each thread is supposed to work on one particular data and the threads are mostly relying on registers. They have very small L1 cache and shared memory.

So, they are working on small amounts of data, but each thread is executing essentially the same instruction on a specific data location. So, the thread id should be well mappable to the data location. So, each thread id will determine which data will be worked on.

Now we can see that when a kernel is launched, it launches a grid and inside that grid there are multiple blocks. Inside each block, there are multiple threads. So, when a kernel is launched, it launches thousands of threads, but these threads are not launched as a contiguous set of threads. These threads are launched as parts of different blocks. Each thread is a member of a block and there are a number of blocks and they follow something like a Cartesian arrangement.

We can see here. This is a combination of 3D combination of threads that gives us a block here and a 2D combination of blocks gives us a grid here. So, all this block combinedly is a grid. And when we launch a kernel, we launch a grid of blocks. Inside these blocks there are multiple threads. So, that is how we launch the large number of threads.

Threads belonging to blocks kernel are launched as a grid which are collections of blocks. So, when we launch a kernel a grid is launched where there are many blocks. Inside each block inside one particular block, say, we took this block 1, 1.

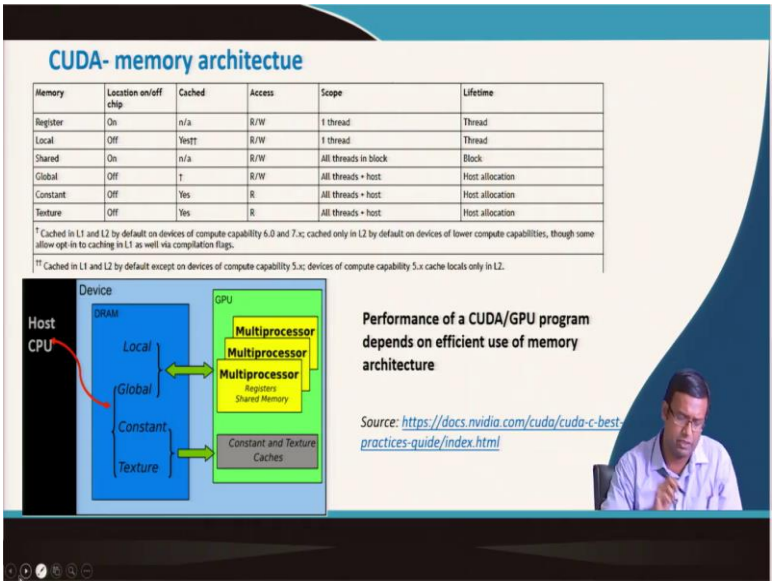
There are Cartesian arrangements of threads and this can be 2D or 3D; we can specify how many dimensions and each thread is therefore identified by the coordinate of the thread inside the block. Each block is identified by its coordinate inside the grid.

Therefore, from the block id, block coordinate and grid coordinate; we can identify the pointer to the thread and that pointer can be related with the data on which it is operating and that is called thread algebra.

Blocks and grids follow a Cartesian structure. The definition of threads facilitates image processing because it is a pixel-based geometry in which the entire display is differentiated into multiple layers and arrangement of pixels on each layer or matrix computing or array handling which we will do in the scientific computing case.

The thread id inside a particular block is mapped to the matrix data structure in a CUDA program. So, you know the thread id inside the particular block, you know the block id combining them, you can find a global id of the thread which can be mapped to the matrix data structure. Now as threads have blocks of multiple coordinates, we can consider matrices of higher dimensions also. This is known as thread algebra. While looking into matrix manipulations, we have to look into thread algebra also

(Refer Slide Time: 13:20)



Then it is important to look in the memory architecture of the for CUDA program. So, there is a host CPU which writes data to the device ram. Where it writes data to the device ram? It writes to the global memory. There are some other memories called constant and texture. Remember CUDA, GPUs have been developed initially for graphics purposes. So, terms like texture are there which are given to the GPU rams for the graphics rendering purpose.

However, there are two different types of memory constant and texture. There is another memory on the device ram which is known as local memory

Now, we see where this memory exists and then we will use this device ram while accessing and device ram is connected to the main GPU part, the TPCs and multiprocessors to a bridge through an interconnect. Inside each multiprocessor there are registers and shared memory each. Inside a multi-processor, we have the computing cores who are accessing registers and shared memories.

So, in terms of latency, registers and shared memory variables have less latency because they are residing exactly on the multiprocessor. Register, we know it is there on the processor almost. So, multiprocessors can very quickly access data from there.

But the main global memory as well as what is called local memory; local memory is memory associated to each thread, but this does not reside on the multiprocessor. So, this is on the device time. So, this has high latency accessing the local and global memory.

The constant and texture though the name suggest by constant that this value will not be changed throughout the kernel operation. So, once the kernel is launched, the constant can be copied to the GPU chip close to the multiprocessors because we are not changing it will be fixed there. If it is the multiprocessor if the codes need to read this value, they can read it from there. So, there is a cache for constant and texture which sits on the chip.

So, if we look into the GPU, the GPU has two part the devices; one is the GPU chip, another is a device frame attached with the processors or TPCs texture processing clusters with an interconnect bridge.

Now, if the memory we are considering is on the chip, on the GPU chip, on the TPC or on the streaming multiprocessor, it can be read very fast and thus utilizing that memory gives us better efficiency. So, it is important to see which memory is where. The register memory is on the chip, it is not important whether it is cached or not because it is on each core that is directly accessing the register.

The thread remembers this register once the thread is over the register value, it cleans up, it washes off as the thread is there. The local memory is cached. Local memory resides on device ram. It is cached in L1 and L2 is cached in L1 and L2 and for compute capability 5, it is only cached in L2, but for other devices; it is cached in L1 and L2 ram. So, it is an off-chip memory, but it is cached in between the ram and the GPU chip. It's a lifetime thread.

So, what is what a thread is operating on is either the register, the private memory to the thread is the register or, if it requires more memory, it is the local memory that is private memory. This is not seen in the other thread, this is what we talked about private memory in open MP, local memory or a register memory is the same register. We know already it is in CPUs also. Local memory is part of the private memory. It is residing in the device stem. So, it has slow access, but it is cached in L1 and L2ram.

Shared memory. What we talk about here is shared by all threads in the block. Shared memory resides in the multiprocessor. So, we understand that when we assign a block all threads in the block goes to one multiprocessor and shared memory resides as available to all the threads in the block. There is no requirement of caching this memory because it is readily available to all the threads there. It is readily available to the chip. So, it is very very fast; it can be accessed.

However, if we need to use L1 cache for some operation, we can assign some part of the shared memory to behave as a cache that can be done in CUDA programs. The global memory which resides on the chip. It is cached in L1 and L2 devices for compute capabilities 6 and 7, cached only in L2 for other compute capability memories.

It is visible to all the threads global memory as well as it is visible to the host. It can be that global memory can be copied back to the host and the lifetime is as long as the host has copied in the memory into the ram, it will still be there. As a host will copy some other memory to that some space, it will change. So, this memory is controlled by the CPU only.

Constant and texture; they cached to the GPU chip, but their main location is outside the chip and they are allocated by the host. They are read only by memory, the other memories can be read and written by the chips. This is the memory architecture here.

Now, we need to keep in mind which memory resides where and which memory is visited is how much latent from that. So, shared memory and registers can be accessed very quickly by the GPU chips. If it has to access anything local memory, though there is a cache, it will be slower access because again this local memory cache is only for 6 and 7, it is also L1 Cache. Earlier it was L2 cache only, but this can be cached and L1 caches can be very close to the shared memory, so, they can be read quickly, but once it is cached certain issues related to cache have to be looked into. Whether cache coherency issues will be there? No because local memory is a private memory. So, cache coherence is not there; false sharing is not there because local memory is a private memory

Shared memory is not a private memory. It is visible to all the threads in the block. It is not cached, therefore, the ideas like false sharing type of things might come in the shared memory. Though it is not cached, it is visible to all the threads and this is read write memory. So, all the threads can change this memory.

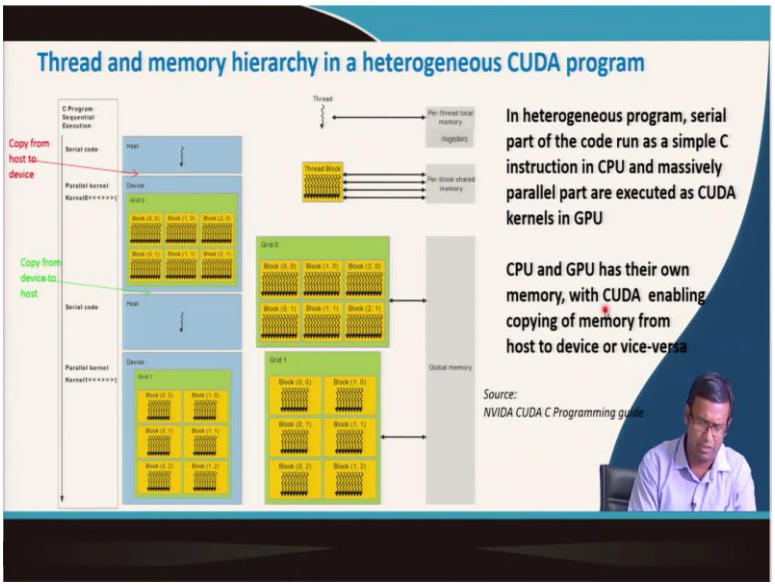
So, contentions can happen in the shared memory and idea of false sharing might come in the shared memory. We have to be cautious while sharing memory; however, the shared memory size is small.

So, when bringing data from global memory to shared memory, we cannot bring a large amount of data; only a small amount of data can go and sit in the shared memory. The global memory is accessed by all the threads and it is also cached. So, cache issues might come here. But again,

the cache coherency type of protocols, atomic operations are supported by CUDA; cache coherence protocols are established by CUDA itself.

Well performance on the CUDA and GPU program depends on efficient use of this memory architecture that we understand.

(Refer Slide Time: 21:41)



Now how are the thread and memory architectures in a heterogeneous CUDA program? Why do we call it a heterogeneous program? Because if we talk about a CUDA say CUDA C program, it is a program which will not only run in GPU or in CPU, it will run in both CPU and GPU. A part of the program will run in CPU a part of the program will go to GPU that is why you call it a heterogeneous program. Because the simple c instructions will run in CPU and the massively parallel part that will go to GPU.

The execution will be like that initially the host will run the serial part. Then the kernel will be called the device will launch the grid of threads and then the grid will come as a grid of blocks then, inside blocks there will be threads. So, a number of threads will be launched there.

Once this thread's scopes are over, it will go back to the host and the host will run another simple C program. Again, another set of grids will be launched when the next kernel will be called.

When host when host will ask the device to operate, it will copy the relevant data from host to device and again once the devices job, the GPU jobs will be done; the relevant data will be

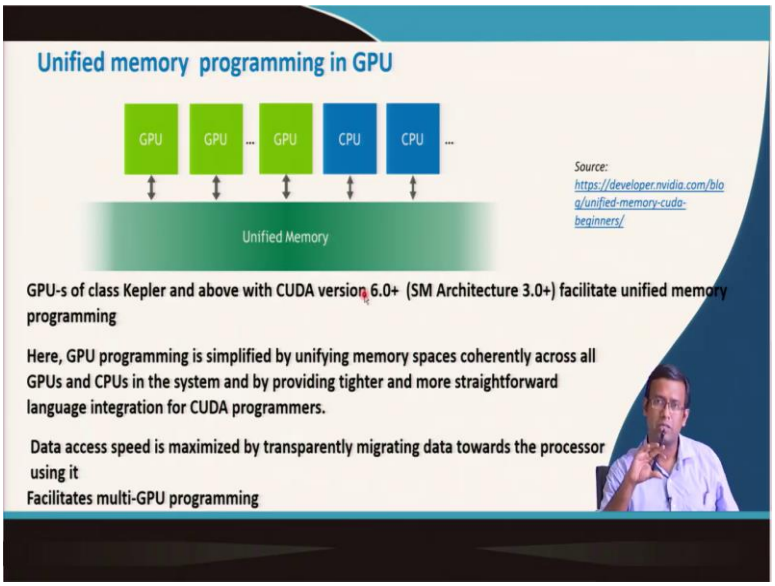
copied from device to host if copying is required; else the and the data is changed by the GPU device in the GPU ram only not on the CPU device CPU ram. So, if you have to read the changes, you have to copy it from GPU to CPU.

In the memory part when a thread is launched, it operates in the local memory on the register directly and the thread block operates on the shared memory. The entire set of threads or the grids they are communicating with the global memory. So, CPU and GPU have their own memory.

Threads first take data from registers, in a sense threads operate on a private data. They take data from a register if there is more data; they create a local memory and take data from there.

These blocks can use the on chip shared memory. Then you have to ask the blocks to operate on that memory and the entire set of grids, they are using the memory which is residing in the device ram. CPU to GPU memory transfer can be only done by allowing CUDA to copy data from CPU to GPU or vice versa.

(Refer Slide Time: 24:41)



Also, lately CUDA has come up with unified memory programming architecture. Unified memory eliminates the requirement of copying data from CPU to GPU and GPU to CPU rather makes the memory visible to all the CPUs and GPU. So, it somehow gives a virtual sense of connected memory in between the CPU and GPU.

GPU-s of class Kepler and above with CUDA version 6. 0 plus or compute capability or SM architecture 3. 0 plus facilitates unified memory programming. Programming becomes simple here because it is the same set of memory which both CPU and GPU is visualizing. Instead of looking into different memory and copying a unifying memory space with coherence across all CPUs and GPUs are seen here.

It gives a tighter association of CPUs and GPUs and more straightforward implementation because you really do not need to think about copying back and copying out here. The data access is maximized by migrating data towards the processor using it. So, data will reside in one particular location in the CPU ram. If the CPUs are working, then it is visible to the CPUs.

If the GPUs are working, the data will be migrated to the GPU ram. That support is provided by the newer architecture of GPUs which are started from CUDA class Kepler SM architecture 3. 0 plus and the newer version of CUDA compatible with CUDA version 6. 0 plus.

So, this gives more flexibility to the programmers as I told you earlier that GPUs are evolving in terms of flexibility and performance. So, this gives more flexibility to the programmers and the performance tuning is also better possible here.

However, we will stick to the legacy version of CUDA programming that is a non-unified memory or programming approach or a heterogeneous memory approach where we will consider GPU memory and CPU memory separately in our programs. Because we are specifically interested here in seeing how different elements of GPU memory are connected and how we can operate over chip and off chip memory efficiently to get right performance.

The one extremely important advantage of a unified memory program is that if there are multiple GPUs as we can see here, the multi GPU programming is facilitated much easily in unified memory architecture. So, we will at some point of time try to discuss some aspects of multi GPU programming and then we will see the effects of unified memory approach.

But now while learning CUDA program we will stick to the heterogeneous memory approach and look into CUDAmemcpy between device to host and host to device considering them as separate memory entities.

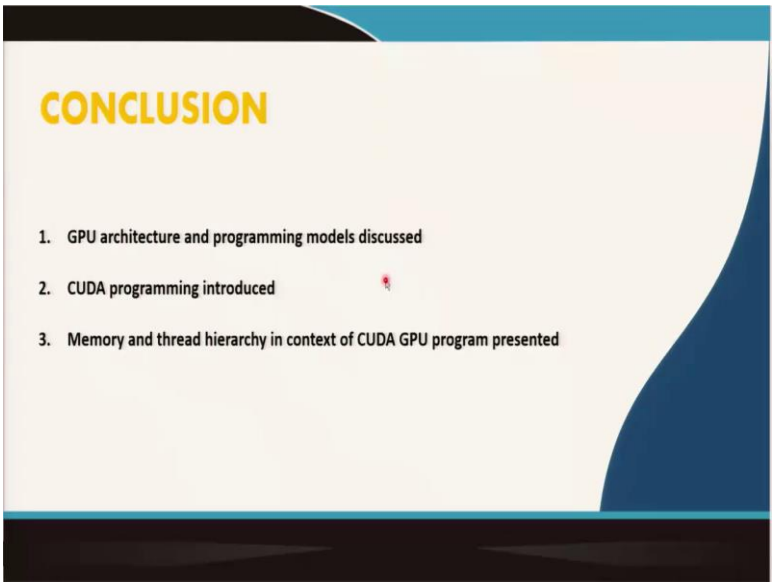
(Refer Slide Time: 28:09)

One of our references Shane Cooks, Developer's Guide to Parallel Computing with GPU. NVIDIA has a number of releases documents and presentations on GPU and CUDA because we are looking into CUDA programming which is proprietary of NVIDIA. I am heavily relying on materials provided by NVIDIA and two of the important reference manuals are CUDA programming guide from NVIDIA and white paper on Volta or V100 architecture.

These two I have consulted heavily and it is important for any CUDA program developer that you go through CUDA C program guide if you want to learn CUDA programming or if you even if you learn CUDA programming, if you want to optimize your applications, you must go to the CUDA programming guide.

If you are a Fortran programmer, we will discuss it in next class. There is a similar Fortran CUDA Fortran guide which is provided by both NVIDIA and PGI. PGI which gives Fortran compilers.

(Refer Slide Time: 29:22)



Well, the conclusion is we have discussed GPU architecture and programming modules, introduced CUDA while discussing GPU programming model and the memory and thread hierarchy in CUDA and GPU program presented.

In the next class, we will look into CUDA detail and we will start looking inside the CUDA programs.