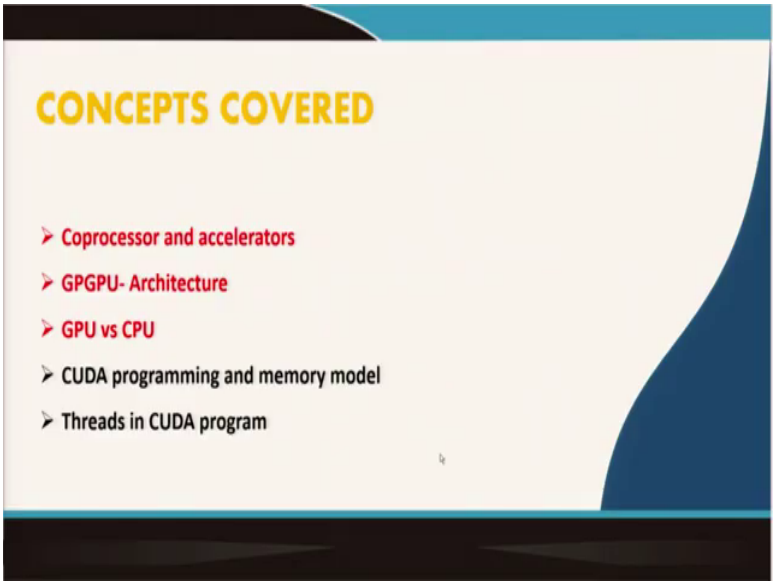
High Performance Computing for Scientists and Engineers
Prof. Somnath Roy
Department of Mechanical Engineering,
Indian Institute of Technology, Kharagpur

Module – 04
GPU Computing
Lecture – 32
Introduction to GPGPU and CUDA (Continued)

Hello. We are in the class of High Performance Computing for Scientists and Engineers and this is the 4th module on GPU Computing and we are discussing Introduction to GPGPU and CUDA.

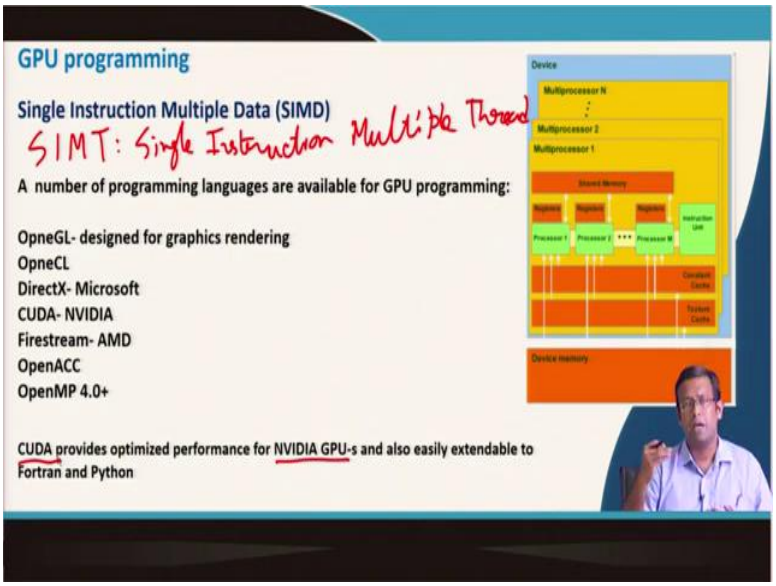
(Refer Slide Time: 00:37)



So, in the last couple of classes we have discussed coprocessors and accelerators. We looked briefly into GPGPU General Purpose Graphics Processing Unit architecture. We understood that GPU has a different architecture and programming GPUs needs an awareness about that architecture also.

We looked into the difference between GPUs and CPUs both from the architecture perspective as well as from the programming and execution perspective and we will start discussing GPU programming now.

(Refer Slide Time: 01:19)



So, essentially we use a single instruction multiple data model in GPU programming. This is a schematic of the GPU architecture. There is a device memory,streaming multiprocessors are inside each multiprocessor.

There are multiple processors or SPS cores streaming processors. They have one instruction unit. So, they basically take one particular instruction and many threads operate on that instruction. They take memory from the device ram each processor has a register only, but all the processors combinedly in one multiple processor share a shared memory issue.

So, if you are not using the shared memory then the from device memory directly variable is loaded into the register and processor uses that and each one is operating on the different data. So, the cache issues are not that important in case you are not using shared memory.

In case you are using shared memory cache coherency false sharing all these issues will come in place. If you have to write an efficient program you have to use shared memory and these issues will be there. We will look into it in detail later. There are some other caches like constant and texture caches we will see about them later.

But, most importantly, there is one instruction unit which offloads the same instruction to different processors and different threads are launched there. We also call a GPU programming model a SIMT model or Single Instruction Multiple Thread model. The threads follow the same instruction exactly essentially the same instruction, but there can be a number of threads

following that instruction.Now while doing the GPU programming and a number of programming languages and APIs are available for that.

Initial programming language the first programming language for GPU programming was this the languages rather they were mostly looking into graphics based programming and OpenGL or Open Graphics Language that was designed for graphics rendering, was one of the dominant languages in early GPUs.

Then open compute language OpenCL came for GPU programming and there are programs developed by different vendors like Microsoft NVIDIA and AMD for GPUs, DirectX for Microsoft CUDA from NVIDIA and fire stream from AMD.

These programs are developed by the vendors specially NVIDIA and AMD. They are GPU manufacturers, they develop the programs, they have optimized it for their particular GPU device.

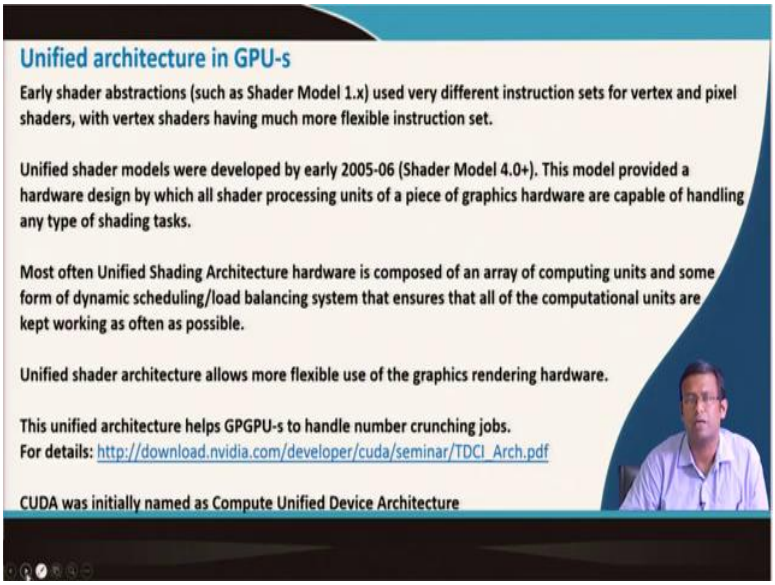
The version of the API and the GPU device version they usually match and this gives us very efficient this gives us very efficient solution also helps programmer a lot to because the vendor itself has looked into certain low level things within the hardware and the scheduler as well as the compiler and the programmer gets little more flexibility while using these programs.

There are open source programs which follow something close to OpenMP. One is OpenACC another is OpenMPs recent version versions after 4.0 they are also capable of GPU programming. However, the vendor developed programs that give optimized performance. I mean over a number of research papers are there that CUDA and AMD can give us much better performance compared to simple implementations using the other programming languages.

Therefore, we will look into the vendor developed program and here for NVIDIA GPUs we will look into CUDA.In this particular course we will mostly focus on CUDA programs and try to understand different features of CUDA programs, try to understand the semantics and syntaxes called by CUDA. Mostly we will consider C programs, but CUDA is extendable to other languages like Fortran and Python also. As I told you as the hardware vendor gives support for this API we can give more efficient and optimized performance using this programming language.

But, you can get optimized performance using something like OpenACC or OpenCL, but that requires more expertise and effort from the programmer side, but here in this course we will try to learn in little detail about CUDA programming and if you are using some other API or some other programming language based on the concepts of the CUDA you can explore that language.

(Refer Slide Time: 06:53)



One important leap in GPU technology was unified architecture in GPUs. We have discussed it earlier that the concept of GPU programming has arisen from graphics rendering.

Now, while looking into the graphics there are 2 different aspects of one particular graphic cell that has some vertices. So, the graphics card has to identify the vertex and put them in the right place and there are pixels. What will be the pixel? How many pixels will be there and what will be the colour levels inside each pixel?

So, there where shading abstractions that these shaders are you can think of shaders are putting colours or grey scales inside each pixel which you are looking into vertices and pixels separately. There are processing units for vertex identification; there are processing units for pixel colouring. Earlier shaders were looking into them looking at them separately. There are different instruction sets for vertex and shaders and so vertex shader has a more flexible instruction set and more region instruction set is there for pixel shaders.

Then around 2005 to 2006 there was a development called Shader Model 4.0 came which provided a hardware design by which all shader processing units be it vertex shader be it pixel shader are capable of handling any type of shading task. So, there is no separate need specifically to look into vertices and specifically looking into pixels. How did it help our case? We are looking into precisely double precision floating point calculations using GPUs. Now, all the cores of the GPUs should be capable of doing similar work.

It happened when this unified architecture came into GPUs or unified shader models came into GPUs that the same that is the hardware design I mean provided the support. So, that all the shader units ,all the processing units can handle similar shading tasks.

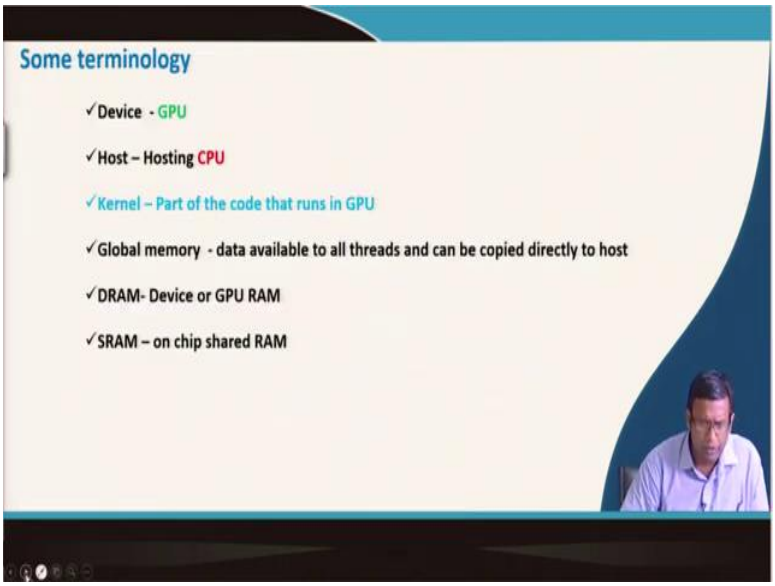
That helped that all the cores in the GPU can handle the same type of tasks. Now, there are certain developments like there are cores designed for double precision, there are cores designed for mixed creation, there are cores designed for single precision.

However, irrespective of the cores location each core can do the same type of work and the same architecture is applied for all the cores. They can do the same type of floating-point work. This unified architecture is called unified shading architecture. This is a hardware which is composed of an array of compute units and some form of dynamic load balancing systems,so that all of the computational units can work at the same instance. That gives us much more concurrency while executing the instruction streams and more flexibility in execution because the instruction schemes can be the same irrespective of the cores.

This gives us the more flexible use of graphics rendering hardware. At the same time in 2005 2006 Compute Unite Device Architecture or CUDA came here that this unified architecture had helped the GPUs to handle number crunching jobs or floating point calculations and using this unified architecture for doing computational jobs or floating point calculations is the role of CUDA.

CUDA was initially named after that Compute Unified Device Architecture, but later because all the GPU architectures are now unified device architecture, later NVIDIA dropped the full term full Compute Unified Device Architecture word or they use of CUDA as an acronym rather they kept CUDA as the name of the API which will be used for programming the GPUs. If you want to know about details about the unified architecture in NVIDIA site you can find out a seminar on CUDA and how unified architecture came into GPUs.

(Refer Slide Time: 12:08)



Now,we will start looking into the features of CUDA. Some of the terminologies will be important that if we talk when we talk about devices we will talk about the GPUs. GPU is the device.

Host is the hosting CPU because GPUs cannot be accessed directly. There is always a CPU which talks to the GPU. So, that is called the host.So, the programmer is communicating with the CPU only. Part of the CPU jobs are running in the GPUs which are the massively parallel part that part is known as kernel.

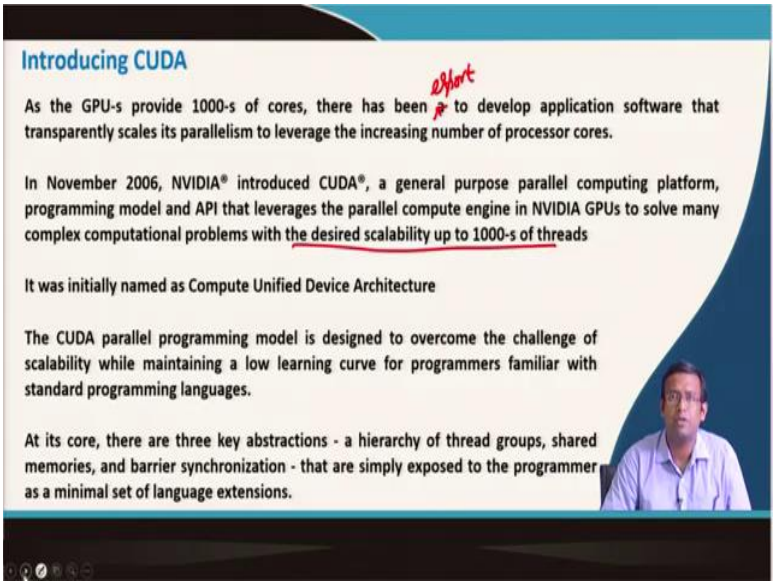
Kernel is a function which is called by the CPU program and that function is executed at the GPUs. Global memory data available to all threads and that can be directly copied to the host or the memory that resides in the RAM of the GPU which is available to all the threads and this is called global memory. This global memory is in the RAM of the GPU. So, we cannot directly access the global memory, but this global memory is the memory which is copied from the host memory. From the host memory what is copied to the device memory is known as the global memory.

This memory is uniformly accessible by all the threads. Now, where it resides physically that resides on the device RAM or the RAM of the GPU and on the TPC of the GPU there is some memory which is known as the shared memory and we call that SRAM or Shared RAM. This is on chip memory ,global memory is not on chip memory. We have looked into the GPU

architecture that there is an interconnect bridge on which all the TPCs are connected and the device RAMs or the global memories can be accessed only through that interconnection bridge.

But, the on-chip memories are the Shared RAM is (Refer time: 14:18) on the streaming multiprocessor itself. So, the GPU cores can directly access that memory with less latency. So, these terminologies we have to keep in mind for the subsequent discussion.

(Refer Slide Time: 14:34)



As the GPUs provide 1000s of cores there has been an effort to develop an application software that transparently scales parallelism to leverage the increasing number of processing cores. There are a large number of cores. So, when we have to develop an application software ,that must scale up to that large number of cores.

Well, the point is that thousands of cores are there, if we use any parallel program, the number of cores will be large and they are essentially operating on SIMD type of architecture. The overheads will be large as we increase the number of processors overheads are large.

So,; that means, there will be a drop down in scalability.Such a large number of cores can be so large that will not get any benefit from such a high number of cores. Therefore, when we are developing an application software it is important that good scalability is obtained as the number of cores are high. The way the jobs will be scheduled, the way the load balancing will be done; that the way the data will be accessed combining everything the scalability will be should be high.

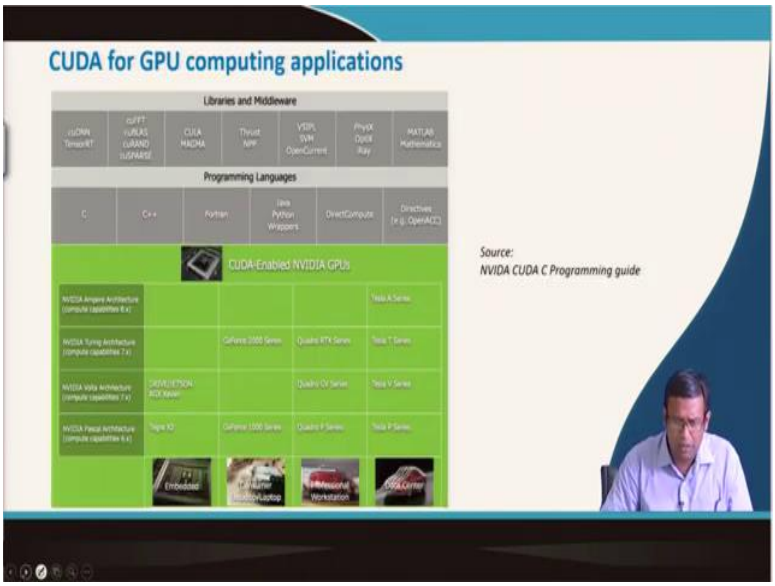
In 2006 typically when unified architecture came into the GPUs NVIDIA introduced CUDA. This is a general purpose computing platform programming model and API which helps the GPUs to solve complex computational problems with the desired scalability of 1000 of threads.

I told you earlier that CUDA gives very efficient solutions, well optimized and the speed up is high. That is because while developing CUDA it is kept in mind that the applications must scale up to these thousands of threads. This is the API as well as the computing platform and programming model which is used for writing GPU programs.

It was initially named as Compute Unified Device Architecture. Later, it is only called CUDA. NVIDIA is it developer NVIDIA do not pose it as an acronym this term CUDA rather it is it is the programming model and API for writing GPU problems. The CUDA parallel program is designed to overcome the challenge of scalability while maintaining a low learning curve for programmers familiar with standard programming language. The programmer who is being introduced to CUDA is most likely coming from a non-GPU background and knows C, C ++, Fortran, Python type of java type of programming knowledge. So when CUDA was developed it is kept in mind that although this scalability is important, learning for the programming language should be simple for the programmers.

In the core CUDA does 3 types of abstractions. One is it puts an hierarchy of the thread groups, how the threads will be organized.A hierarchy of the memories on the shared memories. The how the threads will share memory among themselves and how the threads will be synchronized where the barriers will be put and these are the 3 simple things a programmer needs to know when writing a CUDA program that how threads are organized, how memory is managed and how the barrier synchronization is done.

(Refer Slide Time: 18:44)



Now, we can see that CUDA is quite well popular in the GPU community and a large set of GPU applications in scientific computing are using CUDA. There are a number of libraries in CUDA for example, neural network libraries, FFT blast libraries then MATLAB mathematica.

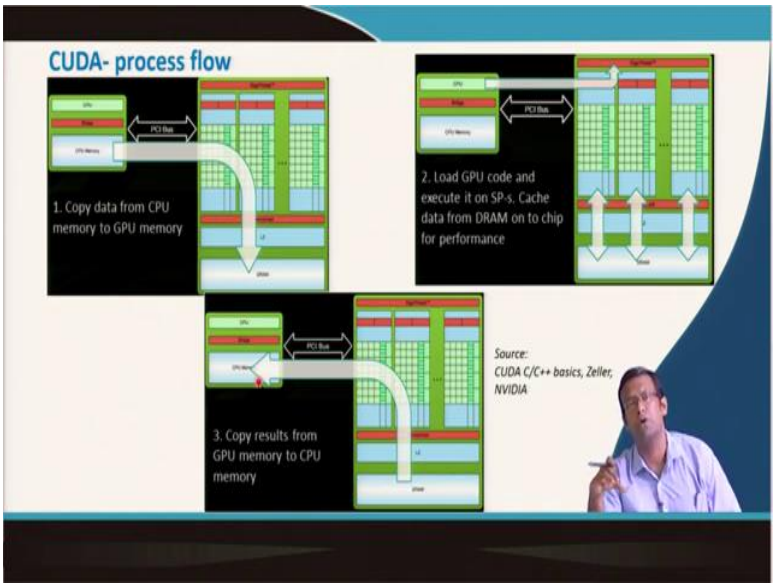
They have CUDA support and a number of libraries are available which have CUDA support ah. CUDA syntaxes are very similar to C plus plus syntax. So, CUDA is initially for C and C++, but now it is extended for other programming languages like Fortran, Java, Python or also in OpenACC directives CUDA can be used.

If you look into NVIDIA GPUs almost all the Tesla GPUs are CUDA enabled, some of the Quadro GPUs ,some of the GeForce GPUs are CUDA enabled and CUDA can be used for the embedded systems, CUDA can be used for consumer desktop, laptop. So, if you have a graphics card in your desktop or laptop it is probably a GeForce graphics card or a Quadro graphics card.

You can write CUDA programs and see about the GPU performance there, but for HPC applications for data centres CUDA is typically used with the Tesla series GPU cards like Tesla P and V series Even for lower compute capabilities CUDA is there, but right now. These are available starting from compute up to compute capability 8 in the data centre CUDA is available.

So, for a large number of GPU computing applications on NVIDIA GPUs ,CUDA can be utilized and around more than 50 percent of GPUs are NVIDIA GPU. So, therefore, a large number of GPUs can actually be programmed using CUDA.

(Refer Slide Time: 21:03)



Well, so now this one important part is how the process flows in a CUDA program. As we said 3 things are important in CUDA. One is how this memory management is happening, another is how the threads are being launched, how the threads are managed another is how the synchronization is done in between the threads, how the barrier is done.

Also if you look about programming a GPU, a GPU cannot be accessed directly. We need a host CPU from which a GPU can be accessed and if we look into the CUDA process flow the first part will be copying the data from the CPU memory to the GPU memory.

We cannot directly communicate with the GPU. We can put the data on which the application will work and we can use CUDA to copy the data from the CPU memory to GPU memory. Then instructions can go from cpu to GPU. This is the kernel function that is being called by the CPU and it can go to the GPU. Then the streaming multiprocessors in GPU will communicate with the device RAM in which data is already put there and all the streaming processors will take data from the device RAM into their chips and perform the operations and that is where the multiple threads will be activated that is also through a CUDA call.

Then once this operation is done with the data set we cannot read that because that is in the GPU data in the GPU RAM. So, that will be again copied back to the GPU CPU memory from the GPU memory CPU memory copying back of the program. Any CUDA program should essentially have these 3 steps.

Copying data from CPU on which we expect GPUs to operate that data initially assigned to CPU, copying this data from CPU to GPU. Now, GPU gets the data threads are launched via kernel call inside the GPUs each GPU takes the data from their device RAM to the chip and streaming multiprocessors or GPU cores operate over this data, change this data write it back to the device RAM. Now, this data is again copied back to the CPU.

This is the basic framework of a CUDA program or any basic framework of a GPU program while using CUDA. CUDA will help us to copy data from CPU to GPU to execute the job in this GPU, tell that this part of the job should be executed in the GPU and ask GPUs to execute that job. While doing that GPUs will read data from device RAM and take it to its registers or the on chip memories and operate on that and write it back in the device RAM and then CUDA will again ask the GPU device RAM to copy that data to the CPU RAM and once it is in the CPU RAM ,we can access this data. This workflow will be followed in all the CUDA programs.

Well, keeping this in mind we will look into the next important features: how the memories are managed in CUDA and how the threads are called in the CUDA problems in the next lecture.