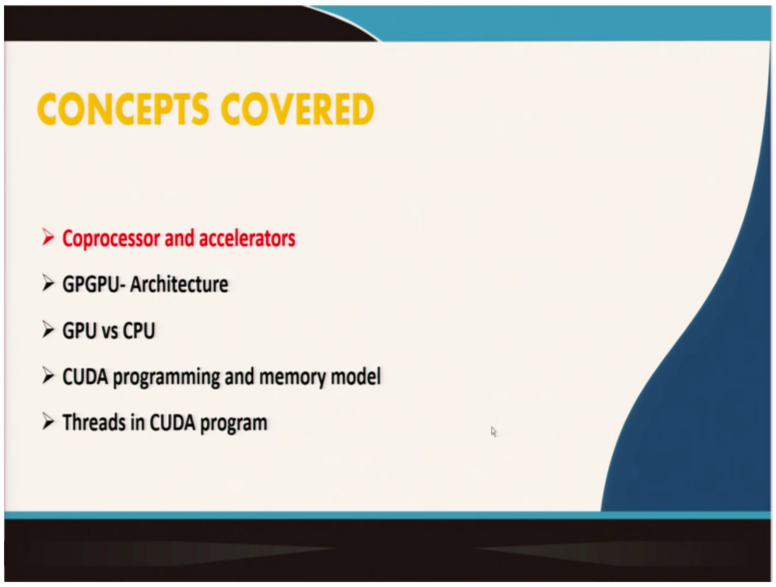
High Performance Computing for Scientists and Engineers

Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module – 04
GPU Computing
Lecture – 31
Introduction to GPGPU and CUDA (continued)

Hello, welcome to the class of High-Performance Computing for Scientists and Engineers. We are discussing the 4th and final module of this course which is GPU computing and we are continuing with our previous lecture introduction to GPGPU and CUDA.

(Refer Slide Time: 00:43)

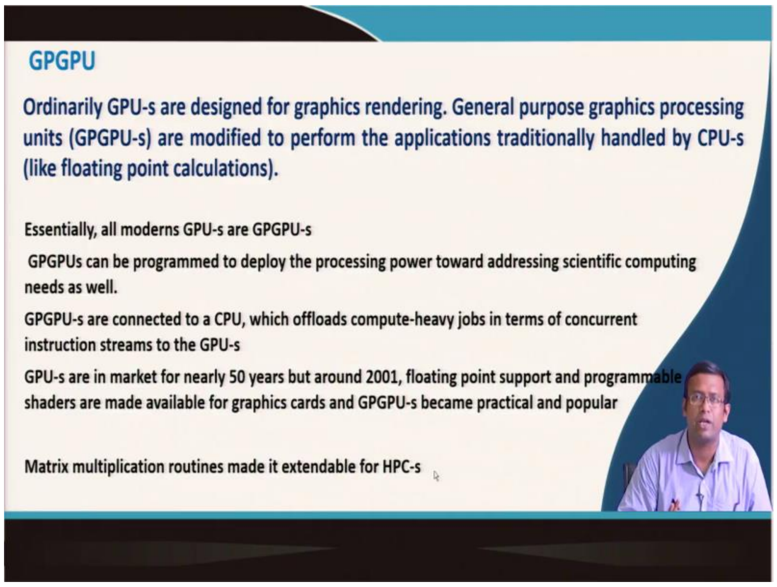


CONCEPTS COVERED

- Coprocessor and accelerators
- GPGPU- Architecture
- GPU vs CPU
- CUDA programming and memory model
- Threads in CUDA program

In the last class we have looked into coprocessors and accelerations and we started discussing what is a general-purpose graphics processing unit and what are the architecture for GPU and I mean what is special about GPGPU and then we are also discussing comparisons of GPU and CPU.

(Refer Slide Time: 01:04)



GPGPU

Ordinarily GPU-s are designed for graphics rendering. General purpose graphics processing units (GPGPU-s) are modified to perform the applications traditionally handled by CPU-s (like floating point calculations).

Essentially, all moderns GPU-s are GPGPU-s

GPGPUs can be programmed to deploy the processing power toward addressing scientific computing needs as well.

GPGPU-s are connected to a CPU, which offloads compute-heavy jobs in terms of concurrent instruction streams to the GPU-s

GPU-s are in market for nearly 50 years but around 2001, floating point support and programmable shaders are made available for graphics cards and GPGPU-s became practical and popular

Matrix multiplication routines made it extendable for HPC-s

We will continue with that discussion in this class. So, as we are discussing in the last class GPU-s can give us extremely high speed up compared to CPU and what a cluster of 100 of CPU-s can do the same speed up we can probably get using a single GPU card of most recent capability. GPU-s were initially designed as the name suggests graphics processing unit, for graphics purposes. However, it is later realized that these GPU cards which come with multiple cores for processing simple instructions concurrently and their number of cores are very high of the order of a few thousands. They can be used for scientific computing purposes for parallel computing or with crunching numbers of arithmetic operations involving arithmetic operations and that is what is called a leap in the GPU technology.

General purpose graphics processing units or GPU-s used for other activities came here for this general purpose. Graphics processing units or GPGPU-s are the modified versions of GPU which are modified to handle the jobs which CPU-s are traditionally designed for. These are things like floating point operations. We will see if there are GPU utilities in deep learning also. So, these GPU-s initial roles were graphics rendering, number crunching doing arithmetic operations were CPU-s jobs.

But now GPUs are modified so that they can replicate some of the CPU-s jobs and can be used for arithmetic doing floating point calculations or arithmetic operations and these GPUs are called GPGPU-essentially all modern GPUs are GPGPU-s they can-do floating-point operations. However, we do not use any GPU for our scientific computing application, because when we do scientific computing it is important for us that what is the accuracy of the solution and therefore, we need to have a control over the numerical errors or the round of errors.

That can be controlled by using double precision floats and therefore GPU-s with good double precision performance are important for our use in scientific computing. So, though all the GPU-s can do double precision computing and can be used for scientific computing, but performance wise the GPU-s which have good double precision performance, that means, which has a greater number of FP 64 double precision cores can be used for our purpose.

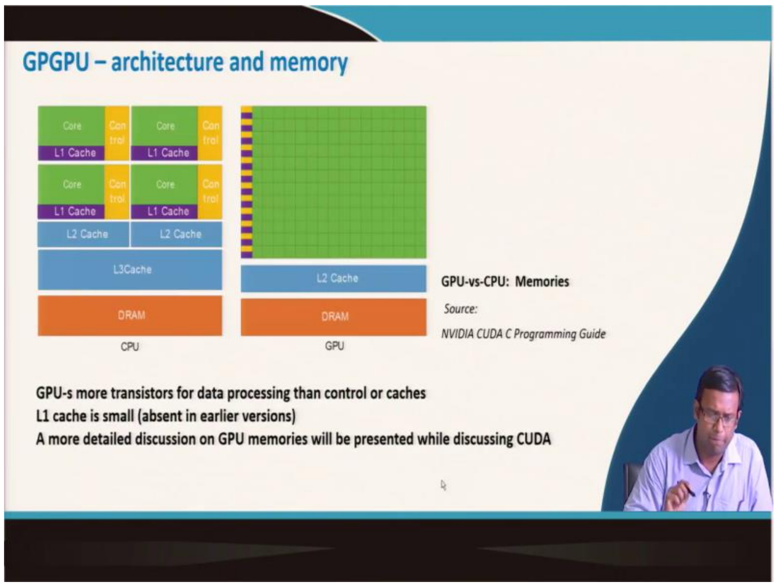
Now, GPGPU-s can be programmed to deploy the processing power toward addressing scientific computing needs as well and they can do matrix computing. We will look into it through our discussions. These GPGPU-s are accessories to the main computers they cannot be directly accessed; as their accessories you need a CPU which only can talk to the GPGPU user cannot directly talk to the GPGPU.

Therefore GPGPU-s are always connected to a CPU and this CPU-s offloads their compute heavy jobs in terms of concurrent instruction streams to the GPU and different cores of the GPU-s work and execute this instruction streams and that is how we get the GPU performance.

General GPUs have been in the market for nearly 50 years from 1970 there are GPU-s in the market. However, this floating-point support and programmable shaders; shaders again this term comes from shading which is a graphics-based utility. So, on the shading algorithm there are certain modifications through which this can be programmed and floating-point calculations can be done. That converted GPU to GPGPU-s and only from 2001 these things are available and GPGPU-s are in market.

It is seen that matrix multiplication can be done very efficiently using GPGPU-s and therefore, HPC-s or scientific computing applications can be addressed using GPGPU-s and HPC systems are using GPGPU-s. So, GPU-s were initially designed for graphics purposes. It is observed that by using the graphics capability of the GPU-s they can be programmed to do floating point operations, they can do matrix multiplications very efficiently and therefore we can use them for HPC purposes.

(Refer Slide Time: 06:31)



GPGPU – architecture and memory

GPU-vs-CPU: Memories

Source:

NVIDIA CUDA C Programming Guide

GPU-s more transistors for data processing than control or caches

L1 cache is small (absent in earlier versions)

A more detailed discussion on GPU memories will be presented while discussing CUDA

Now, if we look into the architecture of the GPU, if you see a CPU architecture this is a multicore CPU or quad core CPU. Each core is connected with one L1 cache, L1 cache is the cache which is sitting next to the core on the chip core; each core has a large control unit. So, if else loops, different complex operations can be operated by the core, there can be scheduling in the core, some active jobs are working on that core, suddenly there is a latency there so some other job can be pushed, it can be hyper threaded shared across many tasks etcetera. Then multiple cores are there say CPU typically have 4, 8, 16 cores it is not a great number of cores can be there, but some cores can be there. Then few CPU-s are connected with the same L2 cache, few cores are connected with one L2 cache all the cores are connected with one L3 cache and then finally they are connected to the dynamic RAM DDR.

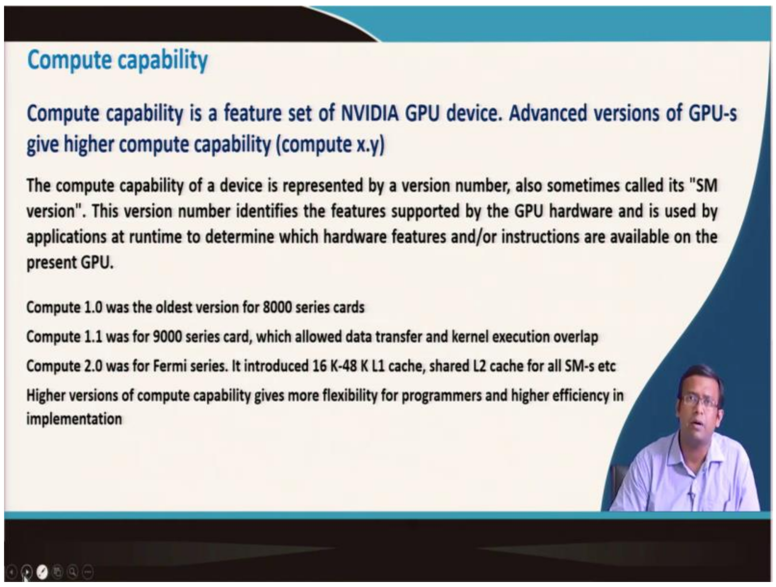
If we look into GPU-s the GPU-s have a DRAM and then L2 cache and there are a large number of cores, but with a huge number of cores there is one small control unit under small L1 cache. In the earlier version of the GPU there was no L1 cache. So, it was taking data directly from the RAM by L2 cache and working on that there is a high latency on the data.

So, whatever GPU-s are working they are relying very small on the control and the cache, therefore they cannot execute complex instruction streams, they can execute simple instruction streams, because they have less control. They cannot work on a large data set or rather they can work on a very small amount of data set.

Because they are mostly relying on the registers for working, if they have to take something which is not fitting in their register, they have to take the data from the L2 cache and dynamic RAM or device RAM which is quite away from the processor. So, this is one difficulty in GPU programming that the most of the transistors in GPU-s are dedicated for data processing for the processing units or cores. Very few transistors are given for control and L1 cache and therefore they have a huge compute potential they can do a lot of computing most of the transistors are actually for doing additional subtraction operations.

However, their data access has high latency and they cannot access large amounts of data, they cannot also perform well complex logical operations because they have a small control unit. So, the number crunching or operating on data using the registers or taking a small amount of data and doing calculations over that is typically what a GPU can do. We will have a more detailed discussion on this while we will discuss CUDA.

(Refer Slide Time: 10:02)



Compute capability

Compute capability is a feature set of NVIDIA GPU device. Advanced versions of GPU-s give higher compute capability (compute x.y)

The compute capability of a device is represented by a version number, also sometimes called its "SM version". This version number identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.

Compute 1.0 was the oldest version for 8000 series cards

Compute 1.1 was for 9000 series card, which allowed data transfer and kernel execution overlap

Compute 2.0 was for Fermi series. It introduced 16 K-48 K L1 cache, shared L2 cache for all SM-s etc

Higher versions of compute capability gives more flexibility for programmers and higher efficiency in implementation

Now we have seen that earlier that with the recent GPU versions the computing power of GPU-s, their flexibility is increasing. This is increasing with the hardware of the GPU, initially GPU-s were for graphics purpose now vendors NVIDIA or AMD are designing GPGPU-s where GPUs are for HPC purpose.

While doing so they are looking into more hardware capability for doing HPC computing using GPU-s and similarly the software the support system the schedulers are also putting there. Combining the features of all both hardware and system level software features of the GPU-s how we can use them for scientific computing purposes. There is a from NVIDIA there is a feature set called compute capability.

Compute capability is given as compute x. y, x is the major version which comes from the compute capability which comes out the newer newer version of the GPU. Why are the minor changes in the feature set? So, with increased value of x we are getting more flexibility and better performance in GPU-s, we can quickly look into the compute capability in detail.

The compute capability of a device means the GPU device here is represented by a version number that is also sometimes called the SM version. This version number identifies the features supported by the GPU hardware and is used by applications at run time to determine which hardware features or instructions are available on the present GPU-s.

So, once we identify what is the compute capability of the present GPU accordingly, we can write the programs and modify the applications based on that. What are the computer capabilities? For example, the first compute capability version compute 1.0 was the oldest version of GPU for 8000 series cards. Then compute 1.1 came for the 9000 series card; it allowed data transfer and kernel execution overlapped; the kernel is the instruction stream that is operating on the GPU.

While a GPU is processing some instruction, it requires data for the next instruction set it will require some other data and this data transfer because the GPU is taking data from the device RAM and also the device GPU RAM is getting data from the CPU RAM mapping data from CPU to GPU. So, there is a certain amount of latency involved in data transfer. So, when compute 1.1 came the hardware allowed overlap between kernel execution or processing and data transfer.

Similarly, when compute 2.0 came it introduced L1 cache before that then GPU there was no L1 cache. So, the Fermi architecture which came with compute 2.0 introduced L1 cache; they put an L2 cache which will be shared by all the streaming multiprocessors.

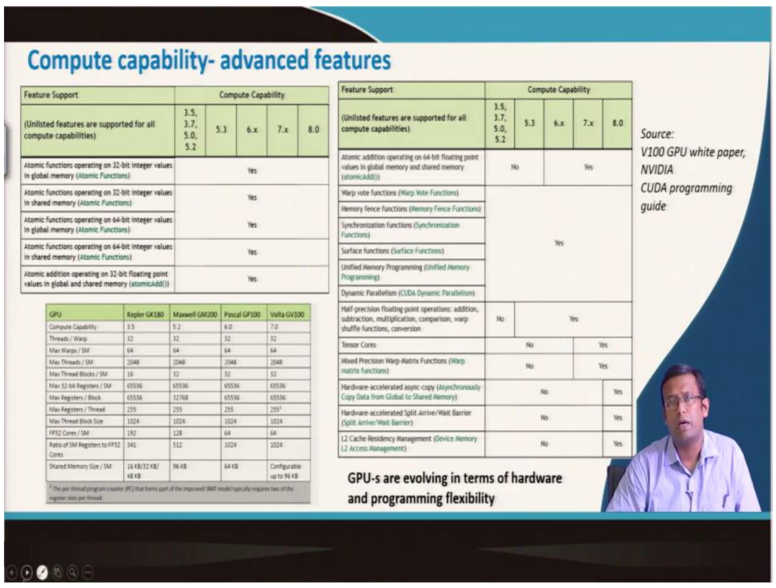
So, this is in between the device memory and the streaming multiprocessor so they put a shared L2 cache. But earlier there was no L1 cache, so when compute 2.0 version is invoked it is a version of GPU which has L1 cache. Why L1 cache is put there for graphics rendering purpose L1 cache was not important, but for number crunching purpose for floating point operations L1 cache is important.

So, as GPU-s are moving more towards HPC applications development in hardware architecture is coming and that development in terms of the computing capability of the GPU is named as compute capability and coming to the newer compute versions.

Higher versions of compute capability give more flexibility for the programmers, you do not have a cache, you have to completely rely on registers, it is more difficult to program. It is also because the program will run less efficiently because the latencies are high.

But as the cache is introduced, programming is easier and more flexible for the programmers, also the efficiency will be better because latency will be reduced. So, with higher versions of compute capability we are getting more flexibility and more efficiency in the implementation.

(Refer Slide Time: 15:01)



Compute capability- advanced features

Feature Support	Compute Capability				
(Unlisted features are supported for all compute capabilities)	3.5, 3.7, 5.0, 5.2	5.3	6.x	7.x	8.0
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	Yes				
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)	Yes				
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)	Yes				
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)	Yes				
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())	Yes				

GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6.0	7.0
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	255[1]
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
Ratio of SM Registers to FP32 Cores	341	512	1024	1024
Shared Memory Size / SM	16 KB/32 KB/ 48 KB	96 KB	64 KB	Configurable up to 96 KB

[1] The per thread program counter (PC) that forms part of the improved SIMT model typically requires two of the register slots per thread.

Feature Support	Compute Capability				
(Unlisted features are supported for all compute capabilities)	3.5, 3.7, 5.0, 5.2	5.3	6.x	7.x	8.0
Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())	No		Yes		
Warp vote functions (Warp Vote Functions)	Yes				
Memory fence functions (Memory Fence Functions)					
Synchronization functions (Synchronization Functions)					
Surface functions (Surface Functions)					
Unified Memory Programming (Unified Memory Programming)					
Dynamic Parallelism (CUDA Dynamic Parallelism)					
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No	Yes			
Tensor Cores	No			Yes	
Mixed Precision Warp-Matrix Functions (Warp matrix functions)	No			Yes	
Hardware-accelerated async-copy (Asynchronously Copy Data from Global to Shared Memory)	No				Yes
Hardware-accelerated Split Arrive/Wait Barrier (Split Arrive/Wait Barrier)	No				Yes
L2 Cache Residency Management (Device Memory L2 Access Management)	No				Yes

Source: V100 GPU white paper, NVIDIA CUDA programming guide

GPU-s are evolving in terms of hardware and programming flexibility

There are some examples of advanced features with compute capabilities, starting from compute version 3.3 and we look up to compute version 8.

So, atomic functions were not there. We understand that atomic operations are extremely important in terms of GPU programming. Because multiple cores are there, they are working on the same data set. So, if one core tries to operate on a certain part of the data set, that part and has to be unaffected by the other cores.

Therefore, the atomic operations are important here. But initial compute versions did not have all the atomic features and we can also see that even from version 3.5 for 32-bit integer value in the global memory global memory is the device GPU RAM, and up to atomic addition of 32 bit or single precision floating point operation was there from compute capability 3.5.

When we talk about double precision of atomic addition, atomic addition with 64-bit floating point it was not there up to the compute version 5. Only from compute version 6 double precision atomic additions; atomic functions we know that a memory location is being modified only by 1 thread, it is not being interrupted or hindered by the other threads.

There is no contention among the threads if the atomic version is not their multiple threads trying to access the same location, there will be contention therefore the performance will fall down. There can be garbage output because the threads are operating on the same variable at a random order and thus output can be garbage.

So, in certain cases atomic operations are important, if atomic capability is not given to the threads then the programmer has to be very cautious and include complex logics which will take care of the contention. For example, padding increases the data set size, so that there is no fault sharing which we have done in openMP programming; this type of logic has to be brought into.

So, programmers' jobs become more restricted as well as performance endurance happens. But when up to version 5.3 we can see atomic addition was given for single precision 32-bit variables. For 64-bit floating-point operations only after 6 atomic operations came, so programmers got more flexibility and performance also increased.

Similarly, something called half precision operation, tensor cores came with higher versions, one of the important is L2 cache residency management that how device memory to L2 memory mapping and this memory management will be done. This was not there up to version 7, that means up to and version 7 is associated up to the Pascal version of the GPU. So, up to Pascal it was not there from the Volta V version it came here.

There are many other aspects so we can see that some of the features were not present for us previous compute capability version in an advanced compute capability version. However, we must not confuse compute versions with CUDA versions, CUDA version is something else CUDA version is associated with the software capability of the CUDA software in which we will by which we will develop the GPU programs. Computes are more with the hardware capability of the GPU.

There is a comparison between Kepler GK 180, Maxwell, Pascal and Volta GPU and the computer versions are different. Kepler has computed version 3.5. So, if we think of atomic addition of double precision that was not possible in Kepler, but in the newer GPU it is possible. Also, this is one of the hardware features that certain memory access operations and some other operations in terms of synchronization are possible in modern GPUs.

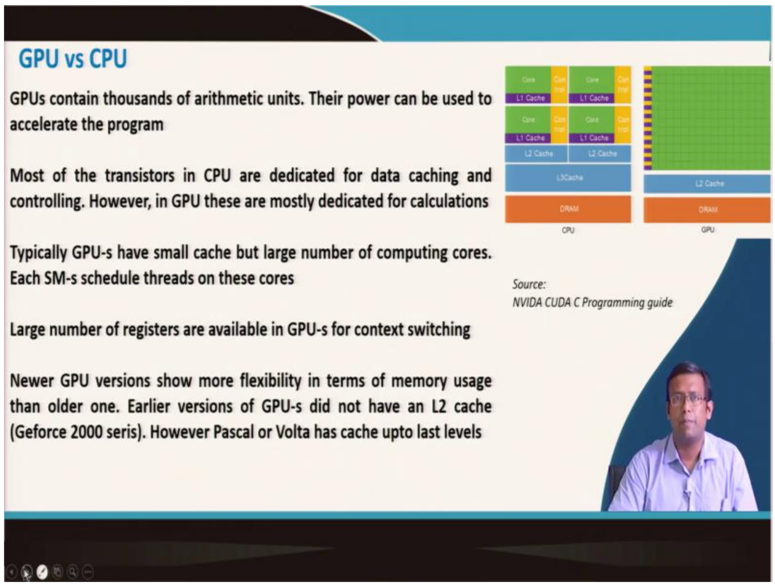
But also, we can see that in terms of the number of threads, in terms of the number of registers in terms of the number of the, in terms of the shared memory size, in terms of the number of cores the modern GPU are making some advancement.

But this is another thing to look into that Kepler has more single precision core, but Volta has less single precision code FP 32 core. Because we have seen previously that the double precision cores have increased in Volta and journey from single precision to double precision again one of the driving factors is scientific computing.

So, modern GPUs are getting better equipped to run scientific computing applications and they are serving our purpose better. GPUs are evolving in terms of hardware and program programming flexibility. So, this is one very important point that learning GPU programming is important in a futuristic sense also, we understand that programming with GPU-s is more difficult than programming with CPU-s.

But GPUs are evolving very rapidly and they will bring more new features and the programming flexibility will increase as well as efficiency will increase. It is the way the market and technology are moving; we can see that these are going to increase in the future. So, if we equip ourselves in GPU programming, we will get the benefit in a very soon.

(Refer Slide Time: 21:24)



GPU vs CPU

GPUs contain thousands of arithmetic units. Their power can be used to accelerate the program

Most of the transistors in CPU are dedicated for data caching and controlling. However, in GPU these are mostly dedicated for calculations

Typically GPU-s have small cache but large number of computing cores. Each SM-s schedule threads on these cores

Large number of registers are available in GPU-s for context switching

Newer GPU versions show more flexibility in terms of memory usage than older one. Earlier versions of GPU-s did not have an L2 cache (Geforce 2000 seris). However Pascal or Volta has cache upto last levels

Source:
NVIDA CUDA C Programming guide

Let us keep on comparing GPU-s and CPU-s, GPU contain thousands of arithmetic units and their power is used to accelerate the program. So, this is a GPU with a number of arithmetic units, but in CPU there are a smaller number of arithmetic units.

Most of the transistors in CPU-s are dedicated for data caching, there is a huge cache in CPU-s 64 kilobyte cache is very common in CPU-s. We have looked into CPU caches when looking into open MP programming specifically and there are many transistors which are controlling and scheduling capacity into the CPU-s.

However, in GPU these transistors are not there, rather if you can put more transistors in a GPU card you will deploy it for adding more cores to the GPU for increasing the computing power. So, the idea is that in CPU you have a smaller number of computing cores, but the computing cores are supported by cache and control units. In GPU you have a large number of computing cores, but they have less support in terms of cache memory management, computing scheduling etcetera.

So, what can you do in the CPU? You can run a complex program involving a huge data set, but you can run fewer concurrent instructions. So, what can you do with the GPU? You can run simple instructions over a small amount of data set, but you can run a large number of concurrent instructions.

That is the main difference in the features of CPU and GPU. Therefore, programming with GPU requires more attention because the support and control and the memory access issues are more difficult in GPU compared to the CPU-s,and therefore if you do not write the program well you can get pathetic performance.

Typically, GPUs have small cache but large number of computing cores and each of this there are multiple streaming multiprocessors that is also important. That in GPU there are many SM streaming multiprocessors inside which many cores are there each streaming multiprocessor can independently schedule the threads into it.

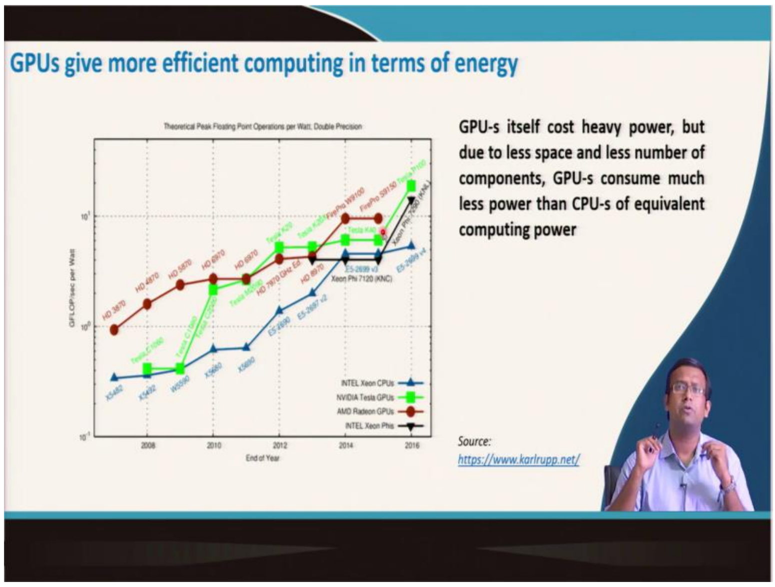
So, if you run a large number of threads few will be taken by one streaming multiprocessor and the number of threads can be much more than the number of cores available. But this streaming process multiprocessor will itself schedule the threads and hide the latency. So, one thread is searching from some data it will put scheduled the other set of threads and they will be active.

So, some way latency hiding is done also in between the threads and that gives more flexibility in the positions. A large number of registers are available in GPU, and these registers are because it has less control, but if you have to do a scheduling one some of the threads are available and now then a new set of threads will be there.

That is done by the registers: the registers take the memory directly on the C processing unit, they are the memory part of the processing unit. Take that load the data on the register and the processing unit operates over it in it.

So, some of the registers are working for some threads and some other registers will work for some other threads, that is possible in GPU-s and a large number of registers also have GPU-s to schedule in between them and switch the contexts. Newer GPU versions show more flexibility in terms of memory usage, for example earlier GPU did not have an L2 cache, the first time as the GPGPU-s came GeForce 2000 there is no cache between the dram and the shared memory. But in the Pascal or Volta there is up to the last level up to L3 cache is available there. So, GPUs are evolving in terms of cache usage and being more flexible for the programmer.

(Refer Slide Time: 25:56)



GPUs give more efficient computing in terms of energy

GPU-s itself cost heavy power, but due to less space and less number of components, GPU-s consume much less power than CPU-s of equivalent computing power

Source:
https://www.karlrupp.net/

There is another very important aspect of GPU programming. What we have shown here is a theoretical performance comparison between GPU, Xeon phis and INTEL Xeon CPU-s. This performance is not in terms of floating-point operations per second only but flops part what.

How much electricity is required to do the computing? It is well known now that HPC centres are energy sinks, they require huge amounts of energy for computing and we have already discussed PUE ,Power Utilization Efficiency.PUE is always greater than 1, 1.5 even more, which tells that if 1 unit of wattage is required to run the computer for one particular computing 0.5 unit will be required for air conditioning and other purposes.

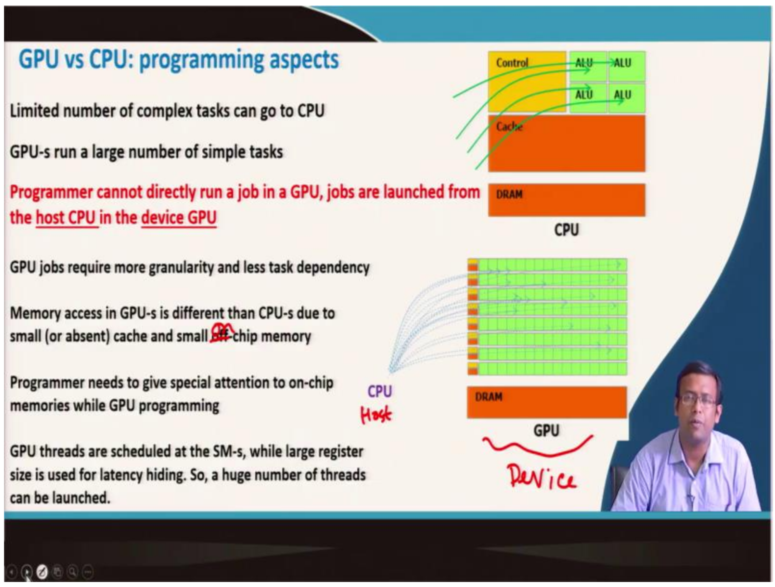
So, you require a huge amount of energy to do high performance computing, because the number of flops is high, your computers heat up you need to cool down and the power required by the transistors to do the floating-point operations is also high. Now GPUs cost power, if you compare to a single CPU many times GPU cost power. But they have less space and you really need a smaller number of components for a GPU which comes as an accessory to the main CPU.

So, when you are comparing with the floating-point operations per second by the power utilization GPU perform much better than CPU-s and it can be seen that there is around 8 to 10 times improvement in flops per watt power from CPU to GPU. So, when we use GPUs using the same energy, we get more computing power.

So, they are more energy efficient compute computing systems also, because one I think we can clearly understand that this is simply a card attached to the main computer. If you think of a multi core system or if you think of a multi computer system you need a huge rack with much more space, many computers there you might need a data centre where the complete room will be dedicated for keeping the computers and you require cooling for the computers when the computers are working. For one CPU you need to operate the motherboard you need to put current to the computers SMPs etcetera.

So, overall power requirement is much higher in CPU-s, but GPU-s are much greener GPU require less power compared to the CPU-s and that is another advantage of GPU that with less energy you can get better speed.

(Refer Slide Time: 29:10)



GPU vs CPU: programming aspects

Limited number of complex tasks can go to CPU

GPU-s run a large number of simple tasks

Programmer cannot directly run a job in a GPU, jobs are launched from the host CPU in the device GPU

GPU jobs require more granularity and less task dependency

Memory access in GPU-s is different than CPU-s due to small (or absent) cache and small on-chip memory

Programmer needs to give special attention to on-chip memories while GPU programming

GPU threads are scheduled at the SM-s, while large register size is used for latency hiding. So, a huge number of threads can be launched.

Now we look into the programming aspects of GPU-s and CPU-s. Limited number of complex tasks can go to the CPU, whereas GPU-s can run a large number of simple tasks. So, if we see a CPU there are 4 cores so 4 jobs can be running in the CPU.

If we see a GPU there are multiple cores and multiple jobs can run in the multiple concurrent instructions can run in the GPU and this can be of the order of thousands and there is some scheduler inbuilt inside the GPU. So, the threads can be scheduled in between them. So, if there are 1000 cores there can be more than 1000 threads which are operating over the GPU-s in an auto scheduled manner.

However, GPU-s cannot work independently; they always require a CPU to be connected with them, programmers cannot directly run a job in a GPU. Jobs are launched from a host CPU to a device GPU and we always use this term host and device; GPU is a device the CPU which is offloading the jobs into the GPU is the host. So, we use this term, this is host and this is device.

GPU-s require more granularity and less task dependency, we can understand because there will be 1000 of threads or even 10000 of threads active for doing one particular part of the job, it has to be extremely granular. Also, because these threads are independently working among themselves, there must not be any task dependency among them. So, whether the program can be given off loaded to the GPU requires that it must be extremely granular and should have less task dependency.

Memory accessing GPU is different from CPU-s. We understand that, due to the small or earlier version there was no cache, small cache and small on chip memory. On the streaming multiprocessor chip itself the amount of memory is very small.

So, that is why most of the memory has to be taken from the device RAM, and in the device RAM it cannot operate on it. I mean cannot you cannot directly give anything assigned to the device RAM or the GPU RAM. It has to take the copy it has to copy the memory from the device ram. The device RAM has to copy the memory from the host or CPU ram.

So, these memory issues are there so memory access in GPU is different than in CPU and we have to be very cautious about the memory management when looking into a GPU program, we will look into these things in detail. Programmers need to give special attention to on chip memories while GPU programming on chip memory. What is on chip memory? On chip memory is the memory given on the streaming multiprocessor or on the TPC Texture Processing Cluster which is very quickly accessible from the cores, is less latency and higher bandwidth.

So, what is on chip memory and how can this on chip memory be accessed and even if we think about L1 cache that is on chip memory in a GPU ram. So, how these memories can be accessed programmers have to keep their attention on that. GPU threads are scheduled in the SMs because the streaming multiprocessors themselves can schedule the threads. So, it can take more threads than the number of cores available and schedule them accordingly.

Because there is a large register file which is given with the GPU it can do a latency hiding using the registers. For example, if one thread is working one particular variable and sends for that variable from the device RAM it is not available on the chip. The other thread can be active at that time, it can load the variable on another register and start working there.

So, there is a scheduling based on the register usage in the streaming multiprocessors of the GPU. That scheduling helps us to launch a huge number of threads much more than the available. The number of cores is also very high, but even so we can launch much more threads, but get very good performance in the GPU-s.

So, now we need to look at the programming on the GPU and we will discuss the programming APIs, which helps us to write a program which will offload some of the jobs from the CPU-s. and these jobs will be executed in the GPU we will look at. We will discuss these APIs in the next class.