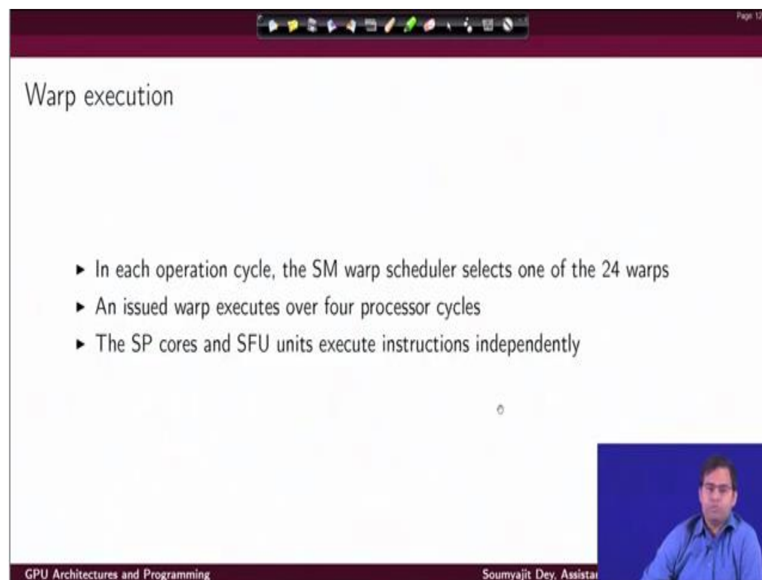
**GPU Architectures and Programing**
**Prof R. Soumyajit Dey**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Kharagpur**

**Lecture No. 7**
**Intro to GPU Architectures (Contd.)**

Hi, So, welcome back to our lectures on up we are hitting certain programming. So we have been discussing this SIMT model of computation. And based on that, how the different threads that are launched via kernell on a GPU kids can use. And as we have seen that threads, get packeted in the form of 32 parallel threads running executing together as what we own as warps and this warps essentially getting mapped to the circular process of course.

And the GPUs multi, I mean multi level scheduler is responsible for scheduling this warps into the processor architecture as we shall see.

**(Refer Slide Time: 01:10)**




And so coming to warp execution, as we have discussed earlier, that for this specific GPU, the SIMD warp scheduler selects, one of the active 24 warps and accordingly. This warp is executed. And, of course, at a time they are many many parallel warps that are active which also depends on the size of the SM And the size of the and the number of SMs in the system and all that. So, how does we warp really execute.

So, for the example system that we have taken here since one warp will have 32 parallel threads is and being executing. And there are essentially 8 SP course. So, an issued warp will execute over 4 processor cycles. And of course, there will be the SP cores feature comprising the interior ALU, and, and also the floating point units. And also there are the special function units.

Which are separate from the SPcores, and they are going to execute those the corresponding instructions independently. So, coming to the GPUs, ISA

**(Refer Slide Time: 02:18)**

ISA

- Support for floating-point, integer, bit, conversion, transcendental, flow control, memory load/store
- Floating-point and integer operations include add, multiply, multiply-add, minimum, maximum, compare, set predicate, and conversions between integer and floating-point numbers
- Transcendental function instructions include cosine, sine, binary exponential, binary logarithm, reciprocal, and reciprocal square root.
- Bitwise operators include shift left, shift right, logic operators, and move
- Control flow includes branch, call, return, trap, and barrier synchronizatio

GPU Architectures and Programming

Soumyajit Dey, Assistan

Like, what are the different kinds of instructions that are supported by the execution model of a GPU. So as we know that this is to be defined by the instruction set architecture of the graphics processing unit, And the instructions that architecture specifies what are the different classes of instructions that are going to be supported. So it happens that there is support for a lot of floating point operations, apart from standard integer and bit level operations.

Also, specific operations like transcendental operations are supported. And there are also specific instructions which control the flow of execution into the GPU is some topic that we will get into in more detail later on. Of course there are also instructions for doing memory load and store of data points. Now, what are really the floating point and integer operations that get executed of these are pretty similar to standard processor instructions.

With respect to addition, multiplication. There are also fused multiply add units. So you have one instruction which will do the multiply add. There is instructions for performing. I mean, minimum, as well as maximum value extraction comparison instructions and set predicate instruction. So there is something also which is very important with this the two instructions flow control, that is also something we touch upon in more detail later on.

And also there are instructions which do the conversion between integer and floating point numbers. And as we have discussed earlier, that we have inside the GPU Apart from this scalar process we also have the special function units which take care of executing the transcendent and functions. That means functions for which you do not have a nice codes on, but there are some. I mean, this our standard numerical algorithms.

We could actually implement their approximate versions, for example. Cosine Transform, sine , binary exponential binary logarithm as we know that in terms of algebraic expressions that exact values will equal to compute an infinite series of terms. But inside the standard ecosystem, the way they are implemented is you have a numerical algorithm which does an approximate computation which is good enough.

In terms of the number of pieces that are provided to the represent the value. So, you have transcendental constant instructions for all the trigonometric functions, then binary exponential binary logarithm computing reciprocal as well as reciprocal square root. With respect to bitwise operators, you have the standard shift left shift right logical operations, and also move instructions. You can also do control flow execution, you can actually have instructions.

Of course you need them in any standard barrier by saying you need instructions which will manage the control flow that is managing branches managing function calls, managing returns the, trap, and also something known as barrier synchronization that is very useful for handling parallel threads in, in any kind of parallel programming interface, it's may be GPU or its may be MPI or something else.

**(Refer Slide Time: 05:45)**

So what about the register file, like we know that in any processor you have a register file which contains the registers the registers called the data, which immediately needs to the ALUs, and so that for doing any kind of a new operation. The operands have to be present in the register, otherwise they have to be brought from the main memory to the cache. So, every SM has a large vector register file.

So it's like a vector processor is registers are divided logically across the SIMD lanes, that is a SPs. So as we have seen earlier, in our small discussion on Vector processors you have Vector registers, that means a register which can hold and array of values, the same type. So similarly like that. I also have a big register file inside on SM, and the register certainly divided logically across the assembly lines.

So that SPs, kind of, present the individuals scalar computer elements of a large vector. And then, what are the values of the registers, that is something that keeps on getting across architecture families with the older GPU architecture we had smaller numbers. But with the newer GPU architectures definitely the number of registers part of them, keep on increasing we will have some figures on this later on.

**(Refer Slide Time: 07:07)**

Fermi GTX 480 GPU

Has

- 16 SMs, total 512 CUDA cores
- Each SM has 32 SPs, 32,768 32-bit registers divided logically across executing threads
- Each SIMD Thread is limited to no more than 64 registers
- A warp has access to 64×32 registers which are 32 bit,
- Alternatively, considering double-precision floating-point operands, a warp has access to 32 vector registers of 32 elements, each of which is 64 bits wid

GPU Architectures and Programming    Soumyajit Dey, Assistan

Now coming to the second architectural example so we move from Tesla, to the fermi family of NVIDIA architectures, which provided some new facilities in terms of processing. So coming to an example. We decided to Fermi  GTS 480 GPU. These are representative example of the Fermi family one of the earlier examples. It has got 16 SMs. Together they can process, 512 CUDA cores.

So you have these 512 CUDA cores inside the system itself segments each of them has got 32 of these CUDA cores, 32 scalar process. And you have got this 32 768 number of 32 bit registers divided logically across the executing threads. Inside each SM. So, I have each SM is comprising this 32 SPs. And these many 32 bit registers. So, if I look at the system from the perspective of a single SIMD thread. It is limited to no more than 64 registers.

A warp has access to 64 times 32 registers, of course inside a warp, I will have 32 such threads. Since each thread has got an access to 64 registers. When warp is executing inside a GPU. It has got access to this overall number of registers. Of course, each of these registers a 32 bit. So, just to get to them values again, overall I have this many number of registers for a single warp. I have access to this 64 cross 32 registers, each of which are 32 bit.

However, they can hold different kinds of data types right? So I can use the 32 bit registers to hold 32 bit data. Also, if I am doing operations on double precision, floating point operations. I

need to 64 bit data values. So we might consider this kind of double precision floating point operations, then I would start saying that warp has access to 32 vector registers. And each of these vector registers have 32 elements.

And each of these elements have 64 bit wide, so essentially this figure of 64 cross 32 registers which are 632 bit changes to 32 vector registers each of them are 32 wide. So, each of them is 32, the width is 32. So, essentially 32 cross 32 registers which are 64 bit while the same register file can operate in both the modes.

**(Refer Slide Time: 10:03)**

So this is an example picture of our Fermi family SM. So, this is a streaming multiprocessor example from fermi family. As we can see, the cores are all, the SP course can be seen here. So, as we say that each SM has 32 SPs. In total there are 16 SMs, we are having a deeper look into one of the SMs right? So inside one SM, I have this kind of 32 SPs. I have got this 32 SPs. And there are 16 of the load store units.

We do memory load and store via the cache to the DRAM. So each SM has got the 16 load store units, essentially, is contributes to 16 SIMD lanes. And each lane has got these 2048 number of registers. So essentially you divide these many registers across this 16 SIMD lanes. So you will laned up with 2048 registers for SIMD lane. Each lane has got one low storage unit loading data points on the, on the DRAM.

Now, if I look at to the other functional units that are present in the SM, then I have got 4 special function units as we can see that they are not directly integrated into this pipeline. So I can see that if I can have an alternate loop into this figure from a horizontal way, then I can start saying that okay for every load/store unit. I have got 2 cores there. That is like an one SIMD lane. I have got 16 SIMD lanes.

These issues are sitting there, they are not directly integrated into each of these lives, but of course for operations. They are available. And I can have in the best case 4 possible special function operations going on. If I take a closer look into each of the scalar processors. So there is the figure we have here. So in this bigger as you can see that you have got inside it so there is the dispatcher, there is a result queue.

And inside this CUDA coree that is the scalar processor. You have one floating point unit, and one integer ALU. So these operations are supported by these units, you can have an integer operation. You can also have a floating point operation here inside the core. The ALU also support standard Boolean, shift ,move, compare convert kind of values. Extracting specific bit fields from an input value in a register and all that.

So that is about the computation part in the SM right? So I have these many compute cores. Inside the SM. Just to summarize, there are total 16 SMs each SM was got 32 SPs, there are 16 load/ storage units, 16 SIMD lanes and this is a over all register file inside the SM. And this register file gets divided across the SIMD lanes. So, each lane gets 2048 number of registers. And if I start looking at the point from the execution view of a warp.

Then the warp has access to these many registers which are all 32 bit. I mean, it has got access to this many registers out of this total. But they can also be configured in a different way and I can say that it also that says to 32 vector registers. Each holding 32 elements which are 64 bit wide.

**(Refer Slide Time: 13:57)**

Memory Hierarchy

- Local memory for per-thread, private, temporary data (implemented in external DRAM)
- Shared memory for low-latency access to data shared by threads in the same SM
- Global memory for data shared by all threads of a computing application (implemented in external DRAM)

GPU Architectures and Programming Soumyajit Dey, Assistant

Now coming to the memory hierarchy of the SM. So as you can see, here I have the organization of the compute units. Also, there is something called a shared memory, but it's also written as an L1 cache with an oblique, and the amount of it available is 64 kilobytes. So what is the shared memory. So, in a memory hierarchy. The memory is organized from the programming point of view into the following parts. So there is a local memory par-thread.

So this is the private temporary data, which is booked par-thread in the external DRAM outside this setup SMs I have one big DRAM here, to which is things are connected through a interconnect. For computation by each thread in the SPs. We have a specific set of registers available as we have discussed that there is a specific set of registers available for warp inside, the warp I have gotten a state of history cores that are engaged.

But of course, those registers may not be enough to hold all the computations values, the intermediate values that are getting computed in a par-thread This is, in case a thread is very computation heavy, and it is generating lots of intermediate data, which needs to be stored somewhere, and the register files allocated set of registers for the thread and not enough, then definitely you need to have space in the DRAM, which can be used right, for the storage of data.

So this is known as the local memory per-thread, essentially insight is in every thread, will have access to some segment of the external DRAM, definitely uses the DRAM the access is slow. So

this will happen, the storage or the access to the local memory of the thread will happen when it is doing some local computation and the values cannot be cannot be stored in the register there are too many things to store for the registers available to this thread.

Then we have shared memory for low latency access to data shared by threads in the same SM. So what is this shared memory. So as we can see that the register file debuggers divided in a bar warp basis but suppose the thread across the warp needs to collaborate among each other. They want to collaborate on the computation that is going on. For that, you need something which is available here, was the shared memory.

So this is the memory segment, which is transparency visible to all the cores. So, somebody's updating some data in this memory segment is visible to other computing thread in any of the other cores. So, the good thing is the shared memory sitting inside the SM. So, the access of the shared memory is very fast. If I compare it with the access of that DRAM. So, if multiple threads, executing across cores you want to do some collaborative computation.

The good thing for them to do, would have the certain data points defined as shared memory type data and access them. Of course, register the fastest access, but next to be access time for the shared memory. So, the shared memory is useful for low latency access to data shared by threads inside the same SM. And then of course you may need a lot of data for the threads to warp on, which is basically sitting in the global memory.

And it is being brought into the system that is inside the SMs hierarchy of shared memory L1 cache and the registers as and when required. So this global memory for data is shared by all threads of a computing application. And this is again implemented in the external DRAM chip of the GPU. So. So just to summarize, you have the global memory available for a CUDA program, defined as a space in the external DRAM chip of the for par-thread.

If there is something that has to be stored beyond the register, because the registers is already loaded. And then you also have some access to the local memory per-thread, which is basically prior to the thread, and that the physical location for that would also be the DRAM. So, we can

understand what is the difference between DRAM segment which is a global memory and a DRAM segment is a local memory.

The global memory is again shared by all threads so any update to any data point on the global memory is visible to all threads. But the local memory is also something that is implemented in the external DRAM, but it is defined in a per-thread basis. So any update done by a specific thread is going to be used by that third only it's not going to be visible to the other fits. So, essentially, how this memory model for CUDA programs is organized.

You have a very high level DRAM. There is a physical location where you have a global memory that is defined that is shared by all the threads, and that is to be used for collaborating computation across SMs, because as you can see, the shared memory is also sitting inside SMs, and this way I have 16 more SMs, and all day sales update hulu's finally to the global memory. So, if I have to do some collaborative computation across SMs threads.

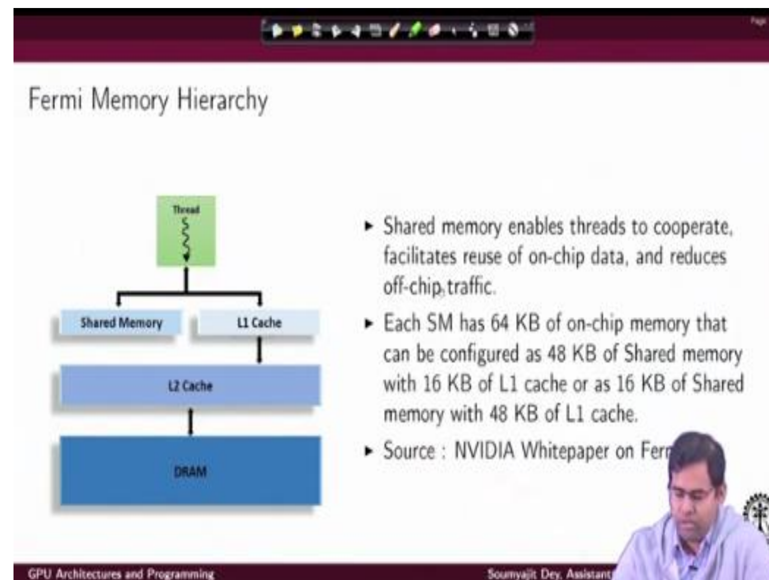
Across SMs in that memory update has to happen to the global memory segment that is defined in the DRAM. Again, I will repeat this part that local memory is also something defining in the DRAM, but is define in a part-thread basis it is used in a part-thread basis used for computation and holding of temporary values for that specific thread. And this is, this is true for every thread individually.

So for holding the temporary data, apart from the register by segment which is assigned to that thread. They have some segment in the local memory, which is again physically located in the external DRAM. For faster computations, and collaboration among thread, inside in SM, you have this shared memory, which is allowing you low latency access. If I compare the access time with this the to the global memory.

And this helps for sharing data by threads inside the same SM. So sharing data by threads across SM has to happen to global memory, sharing data by thread aross SM has to happen to shared memory because it provides a low latency access for each thread in the SM the fastest access of data happens to the part of register assigned to it but if it needs more place for holding temporary

data, it has to access some segments. Inside the DRAM which is defined as that thread local memory.

**(Refer Slide Time: 21:23)**




If we look into the specific memory hierarchy Starting from this Fermi family of GPUs. Then there is something fascinating about the shared memory organization here. So if I look at the computation from the point of view of a thread. If I look at the memory organization from the computation of a view of a thread, then the nearest to me is the register file, not showing the bigger.

The next level is a shared memory or Ll cache then I have L2 cache. And then I have the DRAM. So, the good thing about shared memories as if you can look into the figure, it's inside SM. So, if I have to do the computation inside the SM two different course updating values and exchanging values across the themselves without crossing the boundary of the SM, then there is no point in updating value to the global memory and then going to the other SM. Rather, it can be done to the shared memory right? As we have discussed already.

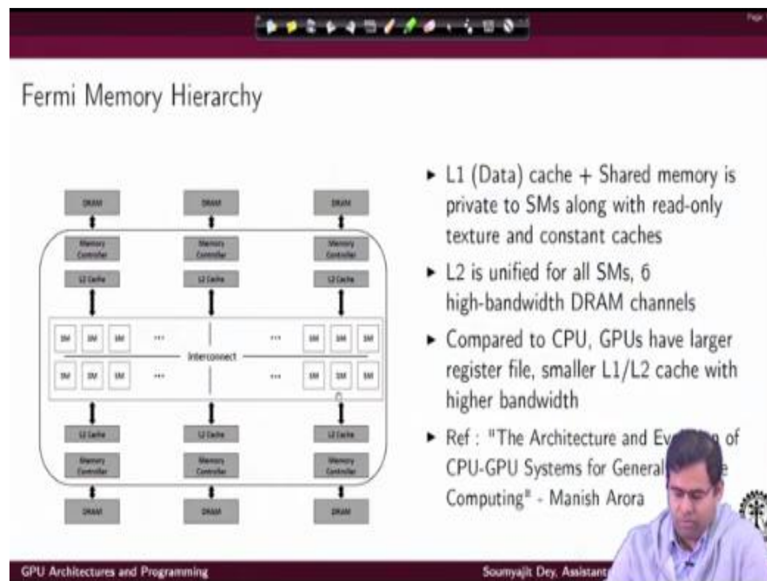
So that is what it enables it enables the threads to cooperate there to facilitate the use of on chip data and reduce of off chip traffic by off chip traffic we mean that access to things that are outside the GPU chip that is the DRAM. So thats outside the processor. Now, each SM will have

64kb of on chip memory. So this memory is configurable, when I mean this on chip memory I mean this shared memory and L1 cache apart so this is 64 kb.

As you can see its written here. Since we wrote this already right is shared memory, oblique L1 cache that isn't is this. It can be configured in two ways it can be configured as 48 kilobyte of shared memory with 16 kilobyte of L1 cache. Or, alternatively, it can also be configured as 16 kilobyte of shared memory with 48 kilobyte of L1 cache. So both things are possible. It depends on what really you want to do.

If you need to have more amount of collaborative execution than you actually across the inside the same across the different CUDA cores inside the SM you may like to have more amount of shared memory.

**(Refer Slide Time: 23:52)**




So, taking a more distributed loop into this memory hierarchy. So, earlier whatever we have been discussing was specific to one SM like this one, this is the picture of the architecture block inside the SM. If you have a loop into the memory hierarchy. From the point of view of the entire GPU chip, you have all these SMs arranged here, right? And they are connected to this interconnect network.

Each of the SMs contained inside them that shared memory or the L1 caches. The register files, the CUDA cores, which are functional units everything right? And you have all the systems here. So, this L1 data cache, or shared memory is private to the SMs, along with some other memory segments feature. The readonly texture and constant cache. So, these are specific cache types, which will also be located inside the SM.

So, the reason is, you can always have specific variables which will you, which your threads will be operating on in read only mode right? So you can put them in the constant cache inside the SM for faster access. And then you have the L2 cache for the L1 cache is private to the SMs. The next level of cache is not private to SMs, but L2 cache is unified for all the SMs. So, essentially, you can think that this L2 cache, its a unified thing.

So you have a common L2 cache to which all SMs can also collaborate. The access to the DRAM has to be done through memory controller from L2 cache, because of course if you do not get the memory element from the L1 you will access L2. If you if the L2 cache also give some miss then the memory controller will reference the DRAM. So the L2 is unified across all the SMs, and the access to DRAM is actually bank.

By bank, what we mean is that DRAM is not organized as a big junk of our physical memory, but is divided into multiple vents of memory, and all these banks can be accessed in parallel. So that is why we are seeing that 6 high bandwidth DRAM channels are present. So we saw here 6 possible access places for the DRAM. And we will of course come back to this topic of shared memory, bank conflicts. And how DRAM banks are access and all that later on.

So, if I compare this idea of CPUs and GPU architectures. It can be quickly observed that GPUs have a larger register file, much larger is to file with respect to CPU. The reason is very simple. GPUs have got more number of cores inside them. By core I mean the parallel processing elements, the basic computes the CUDA cores. And they do simple operations, but a lot of them together in parallel.

For sustaining that you need a very large register file to  provide them with a request requisite number of operants. However, if I compare the L1, and l2 cache size. They are much smaller. Of course the L1 cache is divided across, physically divided across a SMs while the L2 cache is unified. But they are much smaller. But again, they provide much higher bandwidth. They provide much higher bandwidth if I compare with CPUs

**(Refer Slide Time: 27:53)**

GPU ISA

- The instruction set target of the NVIDIA compilers is an abstraction of the hardware instruction set
- PTX (Parallel Thread Execution) provides a instruction set for compilers that remains same for different generations of GPUs
- PTX code gets translated to target hardware instructions while being loaded to GPU

GPU Architectures and Programming

Soumyajit Dey, Assistan

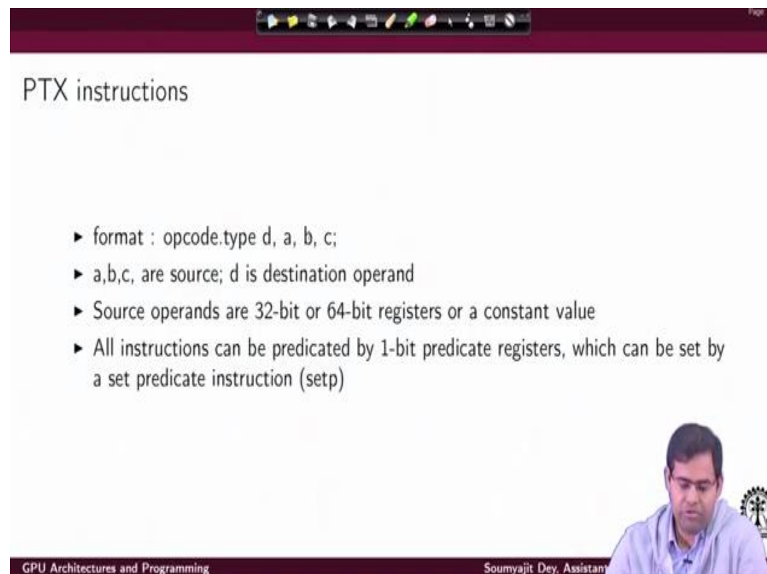
Coming to the instruction set target of the NVIDIA compilers. So, we have already discuss what are the instructions that are supported in general by GPU. But what exact instructions are going to be executed on the GPU. So that is specific set of instructions which are going to be executed in the GPU keeps on changing optimizing, and possibly being refined by suppliers like NVIDIA. So they do a different possible implement a different possible approach in terms of defining the instructions set, If I compare with CPUs.

So what they do is from the programmers point of view, the define an instruction set, which is the target of all NVIDIA compilers. And this is something that doesn't change. So, this is essentially an abstraction of the hardware instruction set, because we change in CPU heavily, the hardware instruction set goes to modification. But the NVIDIA compiler, or if somebody developed some other compiler.

They will, need not always upgrade themselves or conform to the ever changing actual hardware instruction set. But they just need to emit code in a well defined instruction set, known as PTX, PTX full form is Parallel Thread Execution. So, this is the abstract instruction set that is defined for generating code. And if you are trying to write a compiler for the GPU system, you like to generate code, which is in the PTX format will go into details later on today.

I mean, in his lecture, we are just introducing this idea of PTX. But then the question is finally it needs to be translated to hardware instructions. Well, PTX code gets translated to this instruction while its is actually loaded into the GPU. So when the compiler emit the code, you emit PTX code. Compiler optimizations. They are also defined on the PTX code.

**(Refer Slide Time: 30:07)**




The PTX code format is something like this. So you have a opcode, then you have a destination operand is followed by three possible source of operands. The source operands can be 32 bit or 64 bit registers are also a constant value. There is something interesting about this instructions, each instruction can be predicted by a one week predicate register. Which can be set by a specific instruction called a predicate instruction.

So this is a facility that helps to decide whether or not to execute an instruction. Based on some specific conditional that will be there in the program. So this is something that handles that plays a big role in handling branches in GPUs. And that is something that is a that is very important

because we can understand you are trying to execute multiple instructions in parallel. So, how realy you excecute a branch, depends.

Because if you are executing a warp, you have 32 threads progressing together in lockstep by lockstep I mean that you have. I mean, if I go to that older architecture example that we just kept. So there we defined a warp execution inside 4 clock cycles. Each clock cycle. Since because there are eight of the SP cores in that example, but of course you can understand things new things keep on changing the evolution of GPU architectures.

So, from a programmers point of view, it is a good way to think that, warp, all the instructions that are executing in a warp are executing exactly in a lockstep. So, all the threads execute the same instruction together. Next, they execute the next instruction together like that. when these threads face a branch instruction, it may so happen that some of the the threads, they are thread ids satisfy the branch construction.

For all the thread ids do not satisfy the branch constitute construction. So when that happens, then the GPU is to handle, which of the strip should make progress and which of the church should not make progress. And this is something that is decided by the set P says predicate instruction we will get into that later. Thank you. So we will end with this for the time being. And in the next lecture, we introduce something more in this regard.