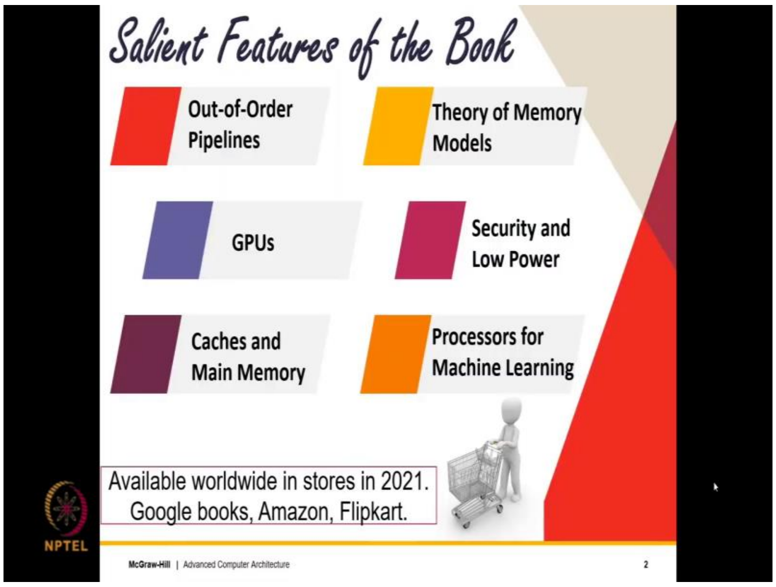
Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 18
Graphics Processors Part - III

(Refer Slide Time: 00:24)



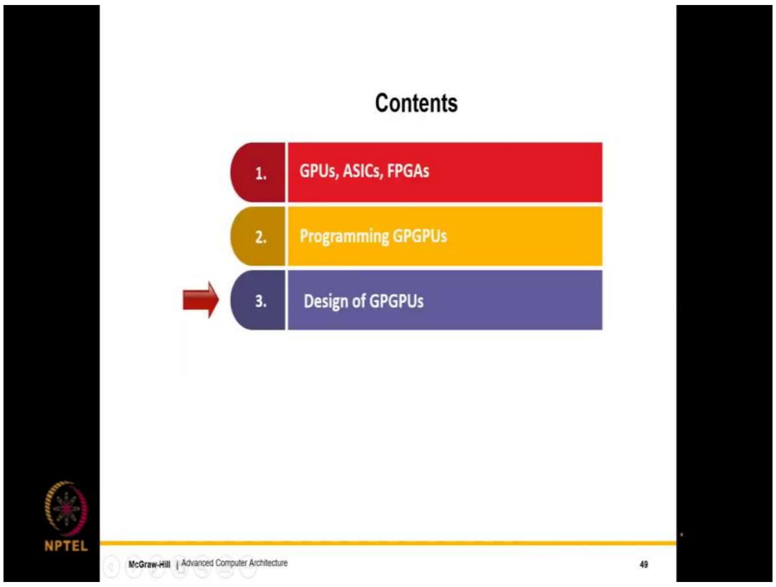
Salient Features of the Book

- Out-of-Order Pipelines
- Theory of Memory Models
- GPUs
- Security and Low Power
- Caches and Main Memory
- Processors for Machine Learning

Available worldwide in stores in 2021. Google books, Amazon, Flipkart.

McGraw-Hill | Advanced Computer Architecture 2

(Refer Slide Time: 00:38)



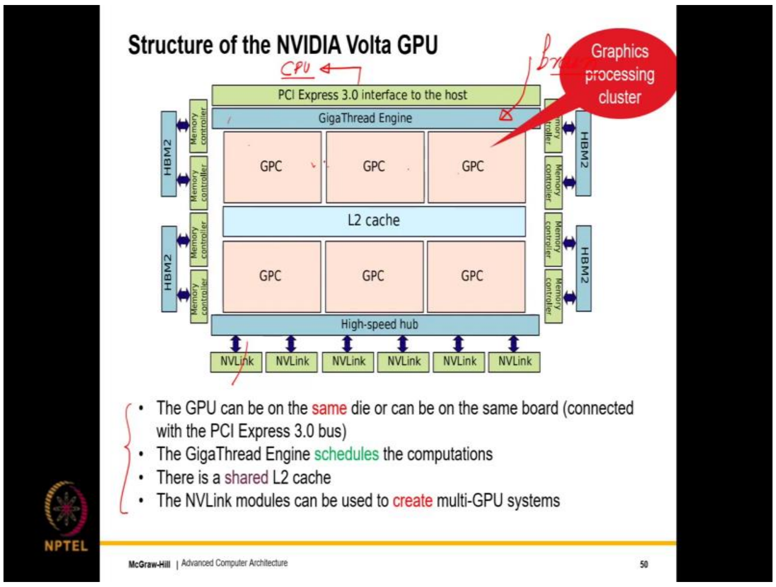
Contents

1. GPUs, ASICs, FPGAs
2. Programming GPGPUs
3. Design of GPGPUs

McGraw-Hill | Advanced Computer Architecture 49

Welcome to the 3rd lecture on GPUs. So, in this lecture, we will talk about the Design of general-purpose GPUs, design of GPGPUs.

(Refer Slide Time: 00:51)



Structure of the NVIDIA Volta GPU

- The GPU can be on the same die or can be on the same board (connected with the PCI Express 3.0 bus)
- The GigaThread Engine schedules the computations
- There is a shared L2 cache
- The NVLink modules can be used to create multi-GPU systems

McGraw-Hill | Advanced Computer Architecture 50

So, let us take a look at the structure of the NVIDIA Volta GPU. So, as you can see GPUs are typically large chips so, they have a large number of computing elements. So, the first green box that you need to look at is the PCI express interface with the host and the host in this case is the CPU. So, the CPU and GPU typically communicate by an external bus which is a PCI express 3.O interconnect.

Then, every GPU task regardless of whether it is a general purpose, or a graphical task is a thread. So, GPU has its internal thread manager and thread scheduler known as the giga thread engine. So, think of this as kind of the brain of the GPU. So, this does the job of the scheduling aspect, it is not much, it is a very simple brain, it is not a human brain, it is more like a bird’s brain, but it is a small brain.

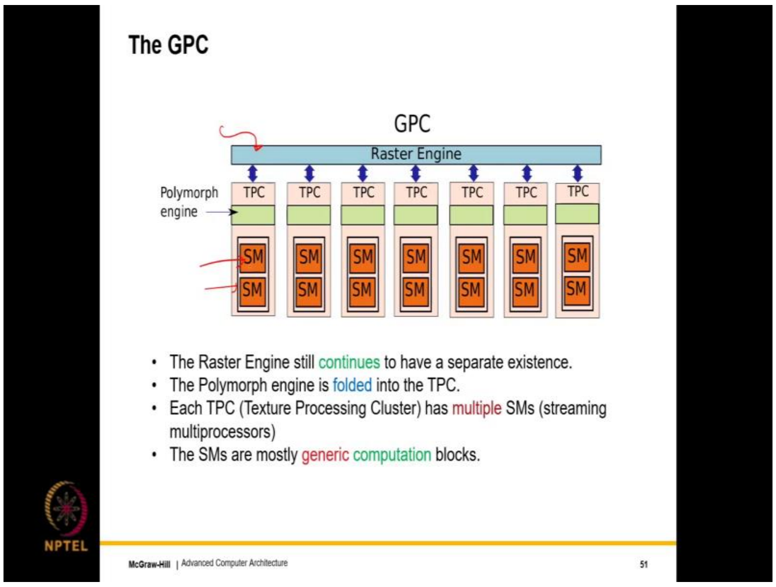
Then, we have these graphics processing clusters which is basically a large collection of small cores. So, the Volta GPU we have six of these, 1, 2, 3, 4, 5, 6, we have six of these and we have a large amount of on chip memory. So, as I said as opposed to as compared to CPU caches, GPUs are bigger, they have more resources. so, they have a big L2 cache.

And furthermore, NVIDIA also allows us to connect multiple GPUs together. So, if you have one GPU, they all can be connected, interconnected together. So, then, we have these high-speed hubs. So, with these high-speed hubs via the NV Link inter connects, it is possible to interconnect GPUs and then, we have a set of eight memory controllers.

The eight memory controllers are then connected to high bandwidth memory. So, we will study what is high bandwidth memory in chapter 10.

So, as I have said all these four points are covered. So, the broad idea or the broad takeaway point from this slide is that every GPU has a large amount on GPU processing clusters, and we have six of these which we shall see in as a deeply hierarchical structure, but we have six of these on a Volta GPU.

(Refer Slide Time: 03:26)



The GPC

GPC
Raster Engine
Polymorph engine
TPC
SM

- The Raster Engine still continues to have a separate existence.
- The Polymorph engine is folded into the TPC.
- Each TPC (Texture Processing Cluster) has multiple SMs (streaming multiprocessors)
- The SMs are mostly generic computation blocks.

McGraw-Hill | Advanced Computer Architecture 51

So, coming to the GPC, the GPC has a raster engine. So, if you go back to the first chapter where we talked about rasterization, we did discuss that rasterization is not something which is amenable to parallelism. So, rasterization requires its separate unit and so, that is the reason is separate hardware for the raster engine has been provided.

And then, in this case, every GPC has seven TPCs, I will come to a second, what is a TPC? It is a texture processing cluster. So, that is like historical name, it is not that this only processes textures, but this is more like a historical name for this piece of hardware. So, in this piece of hardware, we have seven such TPCs 1, 2, 3, 4, 5, 6, 7 in a GPC and then, we have a little bit of extra graphics processing hardware.

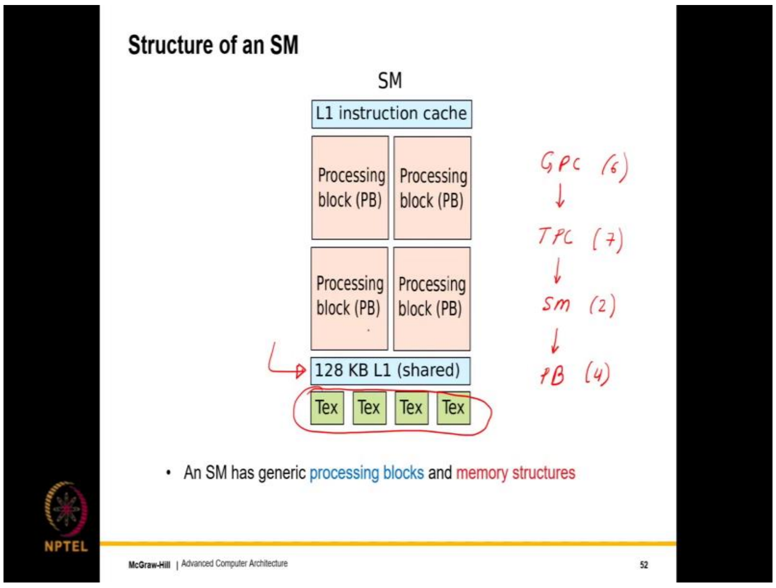
So, as you can see the philosophy was to limit the amount of graphics processing hardware to be as little as possible. So, the special hardware unit that we have over here along with the raster engine, we also have a special hardware unit called the polymorph engine.

So, the polymorph engine does the tasks that a regular polymorph engine is starts to do. So, which we have discussed in great detail in lecture 1 of this series on GPUs where the polymorph engine as we have seen was divided into five stages does tessellation and all of that. So, large amount of the computations is actually mapped to all the cores in these SMs and SM is a streaming multiprocessor which again has a large number of cores.

As you can see, this is a deeply, deeply, deeply hierarchical structure, but the key point to note is that there are two separate units over here for rasterization and the polymorph engine which are kind of dedicated for graphics, but the raster engine is more dedicated. The polymorph engine what it does in modern GPUs to a large extent is that it tries to map the computations to general purpose units as much as possible.

That is the best thing to do from an engineering standpoint such that you can reuse the GPU for regular computations as much as possible. So, now, let us drill down. So, where were we? So, we had six GPCs, eight GPC has seven TPCs and the TPC has two SMs. So, let us now further take one-level deeper and look at the structure of these SMs.

(Refer Slide Time: 06:16)


Structure of an SM

- An SM has generic processing blocks and memory structures

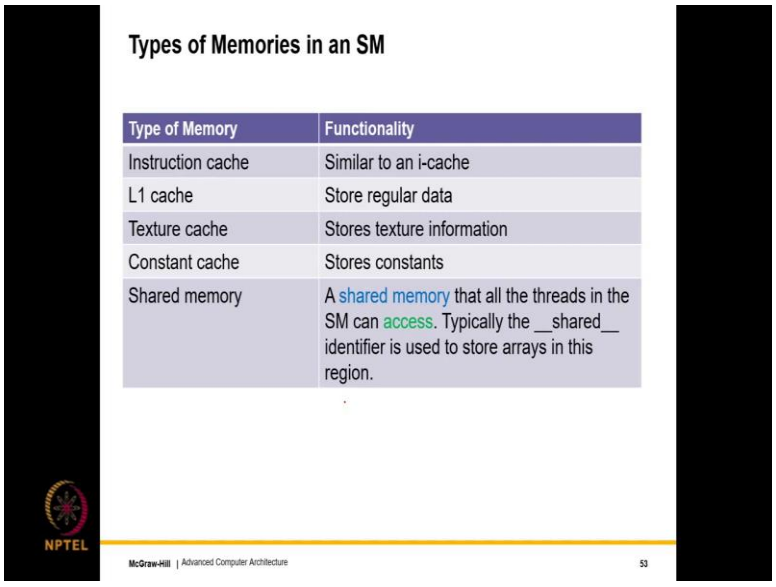
So, the structure of an SM is as follows that an SM has an L1 instruction cache, and it has four of these processing blocks. So, again you see one more level of hierarchy. So, just in case you have forgotten about it, let me remind you, it is a GPC, then a TPC, then an SM and then, the SM is now divided into four PBs. So, for the NVIDIA volta, we had 6 GPCs, per GPC we had 7 TPCs, per TPC where 2 of these SMs and per SM we had 4 of these PBs wow, that is a lot.

So, that is 6 X 7, 42 X 2= 84 and 84 X 4= 336 and we still have not reached the bottom level code, but we are getting there. So, the key points over here what all the processing blocks share in an SM is the L1 instruction cache. So, of course, one instruction in a GPU can actually do a lot because the same instruction is being executed by multiple threads in the kernel as we shall see.

So, that is the reason one instruction can actually translate to a lot of instructions across the thread block. Of course, the same static instruction but multiple dynamic instructions. Then, we have a 128 KB shared L1 cache for data and then, we have four dedicated units for texture, storing texture data.

So, these they are called the texture cache, and this is required primarily because in modern graphics such as games and so on, it is a much better idea to have a dedicated storage of texture caches which store only textured information nothing more. So, this is again one of those units that are dedicated for saving graphics-oriented information.

(Refer Slide Time: 08:26)



Types of Memories in an SM

Type of Memory	Functionality
Instruction cache	Similar to an i-cache
L1 cache	Store regular data
Texture cache	Stores texture information
Constant cache	Stores constants
Shared memory	A shared memory that all the threads in the SM can access. Typically the __shared__ identifier is used to store arrays in this region.

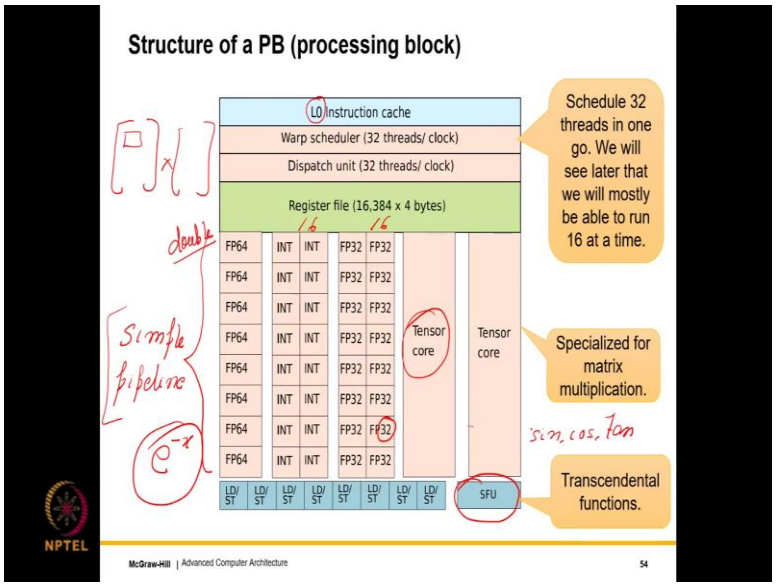
McGraw-Hill | Advanced Computer Architecture 53

So, there are four kinds of memories in an SM. So, what we have seen on the last slide is that we have seen the L1 instruction cache which is quite similar to our i-cache. We have seen the L1 cache, it stores regular data. Then, we have seen the texture cache also which stores texture information.

So, there are a few more types of caches so that there is a constant cache that stores constants and we have also discussed the shared memory directive when we discuss the CUDA processing language. So, this cache over here is actually shared. So, whenever we have the underscore shared underscore identifier, it is used to store arrays in this region which in this case does map to the shared L1 cache.

So, this is the reason that whenever we are writing a piece of code, we need to be mindful of the resources of the GPU for efficiency and let us say that if the size of our shared variables or shared arrays are more than the size of the shared cache, it is not really an error, but of course, things will get slowed down.

(Refer Slide Time: 09:45)



Structure of a PB (processing block)
L0 Instruction cache
Warp scheduler (32 threads/ clock)
Dispatch unit (32 threads/ clock)
Register file (16,384 x 4 bytes)
16
16
FP64
INT
FP32
Tensor core
Tensor core
LD/ST
SFU
Schedule 32 threads in one go. We will see later that we will mostly be able to run 16 at a time.
Specialized for matrix multiplication.
Transcendental functions.
sin, cos, tan
double
Simple pipeline
e^-x
NPTEL
McGraw-Hill | Advanced Computer Architecture
54

So, now, let us come to the structure of a PB which is the last level because PB is the one that contains cores and contains a lot of cores. So, if you look at the PB, the PB also has an instruction cache. so, it is an L0 instruction cache. So, if you would recall, we had an L1 instruction cache with each SM, but each PB has an L0 instruction cache and it can deal with or it can handle, it can manage 32 threads every clock cycle.

So, these threads are like a subset of a thread block, something that we had studied in the previous lecture. So, what we do is that we schedule, or we group 32 threads into one group or one block and they operate in lock step. So, this is known as a warp. So, we will discuss more about warps later, but what each PB has is that it has a warp scheduler.

So, the warp scheduler basically schedules an entire warp in one go. So, we will discuss more about what that means, later and then, there is a dispatch unit which dispatches a warp to all of these cores, we will see in a second what they are, but essentially, the entire warp which is a group of 32 threads is treated as one atomic indivisible unit.

So, the fun part with a PB or a GPU in general is the register file is very large. So, just look at how large the register file is. So, in principle, even though PTAs does assume that we have an infinite number of registers, it is not that far from reality because if you see that we have 16k 4-byte registers, it is a lot of registers roughly, 64 KB of space.

So, such a large register file is required mainly for data locality as well as to support the very high levels of parallelism that we have within a PB. So, now, let us count the number of cores. So, if you look at it, we have eight double precision cores. So, each core is a is like more of an embellished ALU, in a sense it has a very simple pipeline maybe a two or three stage pipeline.

In many cases, it is a single cycle pipeline as well, but the key point is that these are very very simple cores which basically enclose an ALU pretty much and there are few double question units because the idea is it is a bad idea to use double precision all the time, it is expensive. So, there are more of these single precision units. So, we have 16 integer units, and we have 16 floating point rates.

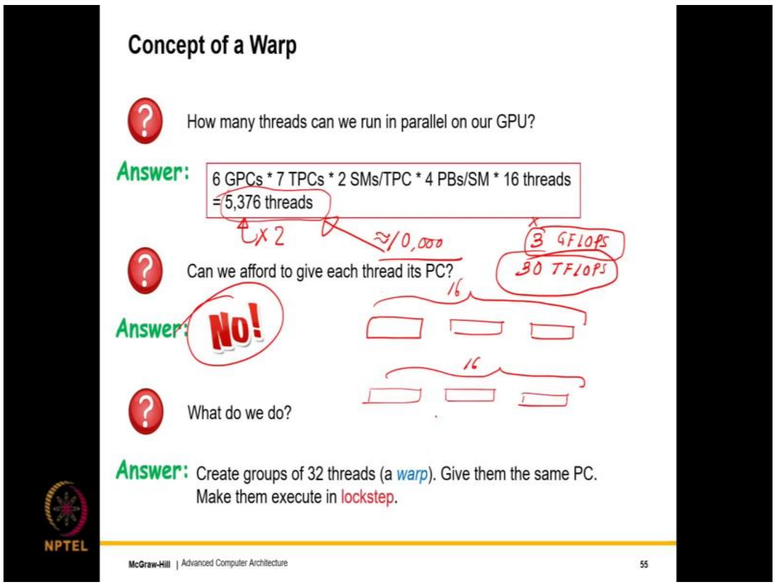
So, any warp if let us say there is an integer operation and there are 32 threads, it will take two cycles to process the entire warp in terms of throughput. The reason being that the first 16 threads go, they execute in lock step. So, we will discuss what lock step business means and then the next 16 threads. Similarly, if it is a floating-point instruction, then 16 threads execute FP 32, 32 is single precision, a float whereas, 64 is double precision or a double.

So, we execute 16 threads and then, 16 more. Finally, we have two tensor cores. So, what we will discuss in chapter 14 which is a chapter on deep learning is that most of the operations in CNNs and in deep learning are essentially matrix operations. Matrix adds, subtract, multiply, primarily matrix sublime operations.

So, the tensor core that we have over here is specialized for matrix multiplication and this is a direct support for any kind of a deep learning algorithm because the tensor core directly offloads the matrix operation and essentially, works in an entire matrix in one go. Of course, if we are multiplying two large matrices, then it is necessary to split them into smaller sub matrices, but that is done later and then, we have a bunch of load store units to talk to memory and a special function unit.

So, there may be several special function units, but I have just drawn one rectangle. So, the reason again is that if you look at many scientific cores, they use many trigonometric functions such as sin, cos and tan. A lot of deep learning actually uses the sigmoid function that is based on computing this value. So, such functions are computed in the SFU primarily using lookup tables.

(Refer Slide Time: 14:38)



Concept of a Warp

How many threads can we run in parallel on our GPU?

Answer: 6 GPCs * 7 TPCs * 2 SMs/TPC * 4 PBs/SM * 16 threads = 5,376 threads

Can we afford to give each thread its PC?

Answer: No!

What do we do?

Answer: Create groups of 32 threads (a *warp*). Give them the same PC. Make them execute in lockstep.

x 2
≈10,000
3 GFLOPS
30 TFLOPS
16
16
NPTEL

So, let us now discuss the concept of a warp right which was one of the key concepts that emerged from the previous slide, but we did not discuss it in great detail. So, first let us do a little bit of math of the number of parallel threads that can run on an NVIDIA GPU. So, we have 6 GPCs, we have 7 TPCs per GPC, 2 SMs per TPC, 4 PBs per SM and let us assume, we can multiply, we can process 16 threads in one go which means either all of this is being used or all of this is being used.

But we will then see that in modern GPUs, they can actually run two separate kernels at the same time and use both, but that is slightly more sophisticated, but the number will double. But if you just were to multiply this 6 X 7 = 42 X 2 = 84, 84 X 4 = 336 X 16 = 5376. So, just look at the shear parallelism within a GPU, we can run these many threads together in one bow.

If you assume that there are no memory stalls, then just look at the shear computational throughput of a GPU. It is true that the frequency of a GPU is slightly lower. so, it is not as high as 3.5 gigahertz, it is lower, it is around 1.5 or so, but still this degree of parallelism is huge, and this is pretty much like a supercomputer made of CPUs, we are bringing all of that within one GPU.

And then, if we go with the modern of that which is that we can execute both of these together LV by different kernels or let us say different warps, then this number gets multiplied by 2. so, we basically have approximately 10000 threads that we execute together and that is a huge huge huge increase in the total computational throughput.

So, if you look at let us say one CPU, for one CPU it will provide you around you know around best case let us say 3 gigaflops where a flop is a floating-point operation per second and just multiply this by 1000, this will become 3 teraflops by 10 more, this will become 30 teraflops. So, 30 teraflops is pretty much what one GPU can give you.

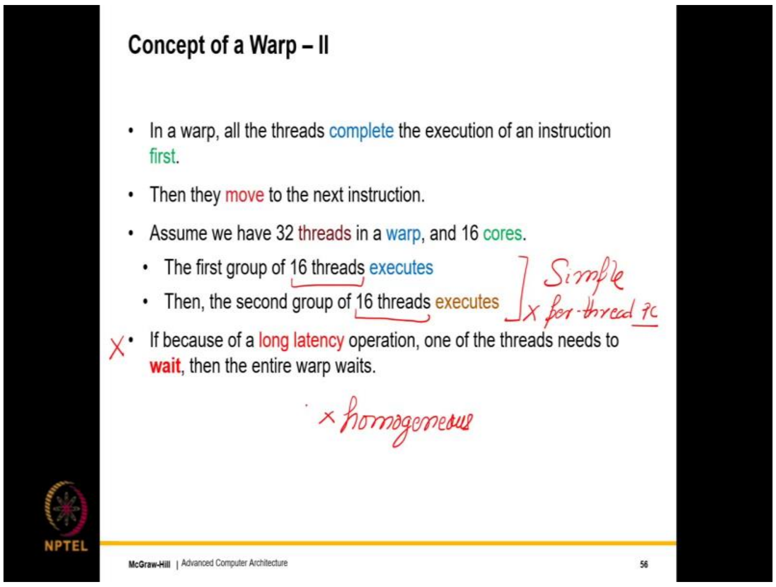
And If you have 30 of this GPUs 33, then you can pretty much realize the petaflop machine. So, it is actually not that hard to get a petaflop worth of computing just using GPUs. Of course, using CPUs, it is much much harder as you can see, but of course, if you have multicore CPU, then this number will get multiplied.

So, it is not that simple. So, basically this number that I gave is for a single core and it is clearly not that simple because multicore CPUs are also better, but that said and done, the shear computational throughput that a GPU offers is huge, is very very large and it is quite hard to beat that.

So, the question is that can we afford to give each thread its PC? So, can the threads be executing different instructions? So, this is something the clear answer is no, this would involve too much or overhead. So, what do we do? So, what we do is that we create groups of 32 threads in the warp, I call them a warp.

So, think of this as a subset of a thread block, they give them the same PC so, because they execute the same code. So, what happens is that all the 32 threads execute in lock step which basically means that we execute the same instruction for all the threads. So, even if let us say there are 16 units so, for the same instruction, we first execute 16 threads and then, we execute the next 16. Once that instruction is done, we move to the next instruction. So, this is what we mean by lock step.

(Refer Slide Time: 18:52)



Concept of a Warp – II

- In a warp, all the threads complete the execution of an instruction first.
- Then they move to the next instruction.
- Assume we have 32 threads in a warp, and 16 cores.
  - The first group of 16 threads executes
  - Then, the second group of 16 threads executes
- If because of a long latency operation, one of the threads needs to wait, then the entire warp waits.

Simple
× per-thread PC
×
× homogeneous
NPTEL
McGraw-Hill | Advanced Computer Architecture
56

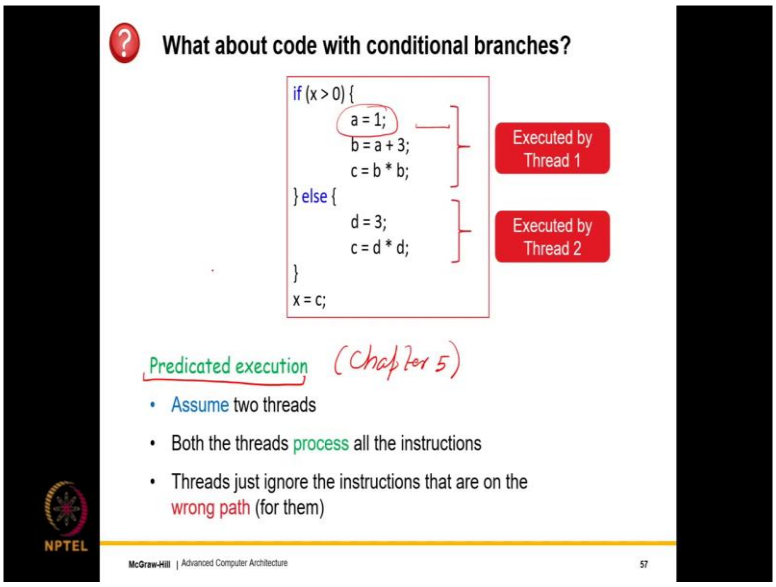
So, let me further explain. In a warp, all the threads complete the execution of an instruction first and then, they move to the next instruction. So, just to reiterate, if we have 32 threads in the warp and we have 16 cores, the first group of 16 threads executes and the second group of 16 threads executes and if let us say that we have only 8 functional units, then instead of this taking two cycles, it will take four cycles.

First group of 8, second group of 8, third and fourth groups of 8. So, there are several good things about it, the first is that it is simple, no doubt. We do not have to maintain a per thread PC. So, if let us say because of a long latency operation, I am not talking of a disadvantage, because of a long latency operation, one of the threads needs to wait, then an entire warp will wait.

So, warp essentially does not leave a thread and go forward, the reason is that we are not maintaining a per thread counter or a per thread PC. so, because we do not have a per thread program counter, what we do is that if one of the threads stops, the entire warp stops.

his may sound inefficient but given the fact that GPU cores are incredibly overgenerous, in a quite predictable and quite homogeneous. This normally does not happen, normally it is not the case that one thread gets delayed substantially, the reason being that also the data in memory is co located so, normally this does not happen. So, that is the reason its typically not that concerning.

(Refer Slide Time: 20:42)



What about code with conditional branches?

```
if (x > 0) {
        a = 1;
        b = a + 3;
        c = b * b;
} else {
        d = 3;
        c = d * d;
}
x = c;
```

Executed by Thread 1
Executed by Thread 2
NPTEL

Predicated execution (Chapter 5)

- Assume two threads
- Both the threads process all the instructions
- Threads just ignore the instructions that are on the wrong path (for them)

McGraw-Hill | Advanced Computer Architecture 57

So, before you ask the question, let me answer what about code with conditional branches? If we have code with conditional branches, then of course, as you can see there is an if part and there is an else part, the notion of lock step execution does not hold because maybe out of 32 threads, 10 threads will execute the code of the body of the if statement and 22 threads will execute the code in the body of the else.

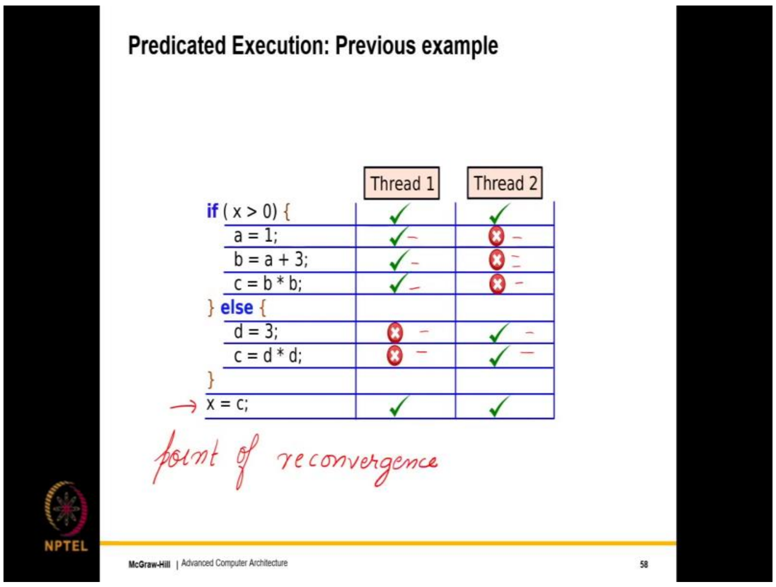
So, then, how do we execute by execute in lock step? So, here we do predicated executions, if you would go back to chapter 5, the lecture on Itanium, then we will have the notion of predicated execution. So, predicated execution is like this. Let us assume two threads.

So, then, what will happen is thread 1 and thread 2 will process both the instructions, but for thread 1, the predicate will be true which means that this is on the correct path so, it will both process, execute and commit the results whereas, thread 2 will process in the sense it will just get the instruction in, but will ignore it, it will not do anything with it.

So, they will still proceed in lock step. So, the key point that I am trying to make here is that all the threads in a warp will still proceed in lock step, it is not the case that lock step execution will not happen. It is just that for every instruction, a thread needs to assess whether the instruction is in its correct path or not.

If it is on the correct path, then so, what it will do is that it will basically process the instructions, compute the results and update the state otherwise, it will ignore it and for more on predicated execution, you can go to the chapter on Itanium, chapter 5, the lecture on Itanium, the last lecture of chapter 5.

(Refer Slide Time: 22:49)



Predicated Execution: Previous example

	Thread 1	Thread 2
if (x > 0) {	✓	✓
a = 1;	✓	✗
b = a + 3;	✓	✗
c = b * b;	✓	✗
} **else** {		
d = 3;	✗	✓
c = d * d;	✗	✓
}		
→ x = c;	✓	✓

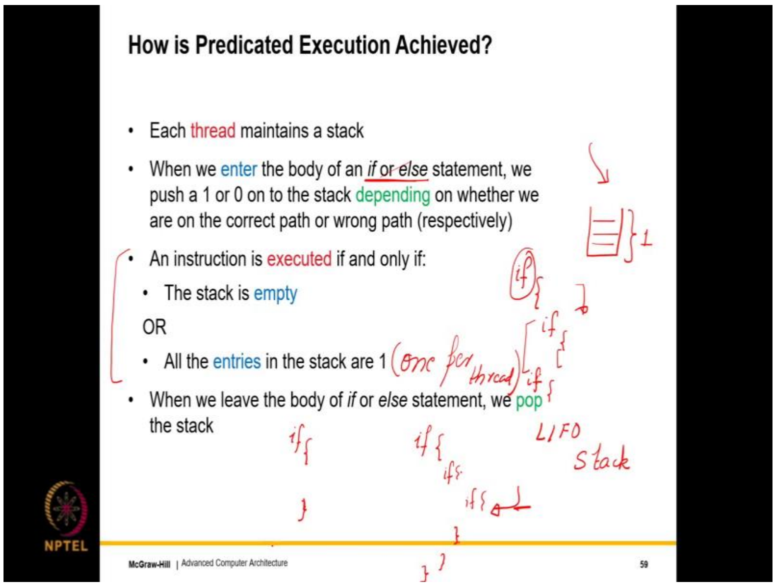
point of reconvergence

McGraw-Hill | Advanced Computer Architecture 58

So, let us look at predicated execution, the previous example. Assume there is thread 1 and thread 2. So, both will execute the if statement because here they are evaluating whether x > 0 or not. So, of course, x is a thread, specific variable here, that is why this instruction falls in the correct path of both threads 1 and 2. Now, if we look at the body of the if statement, then the body of the if statement is in the correct path of thread 1, but for thread 2, it is on the wrong path.

So, thread 1 will process the instructions and thread 2 will just ignore. Finally, the body of the else well for thread 1 will not process, thread 2 will process and this is known as the point of reconvergence because this is when thread 1 and thread 2 start processing, both of them start processing the instruction. So, you can say that this is where the if statement kind of ends so, this is the point of reconvergence.

(Refer Slide Time: 23:53)


How is Predicated Execution Achieved?

- Each thread maintains a stack
- When we enter the body of an *if* or *else* statement, we push a 1 or 0 on to the stack depending on whether we are on the correct path or wrong path (respectively)
- An instruction is executed if and only if:
  - The stack is empty

  OR
  - All the entries in the stack are 1 (one per thread)
- When we leave the body of *if* or *else* statement, we pop the stack

McGraw-Hill | Advanced Computer Architecture

59

So, how is predicated execution achieved? Well, each thread maintains a stack. When we enter the body of an if or else statement, what we do is we push on a 1 or 0 onto the stack depending on whether we are on the correct path or the wrong path respectively. So, let us say I am entering the body of an if statement or an else, what I do is that I push a value onto the stack. So, I will to come to a second why it is a stack.

So, if let us say I am on the correct path, it push a 1 otherwise, it is a 0. Now, coming to the fact why it is a stack well, because we could have nested if statements. See if that is the case, it is very well possible that I may be on the wrong path or the first if statement, then after that I come and then you know maybe I do a comparison and I see that I am in the correct path of a if statement that is nested in the outer if statement, but this does not matter because I would have not entered this piece of code anyway.

So, what I basically do? or what is supposed to be done is that the stack maintains the fact if you see there is a last in first out behavior, the last in first out behavior comes because if I have one if statement, I have one more, I have one more and so on, then I enter this if statements body at the end and also exited the first so, it is last in and first out again, last in and first out again, last in the first out.

So, the moment we have a last in first out behavior, we use the stack data structure. The stack data structure is basically used to store the outcomes of the branches or whether I am on the correct path or on the wrong path. So, to ensure that I execute an instruction, all the entries in the stack assuming that even if I am in the wrong path of an if statement, at least I do the comparison.

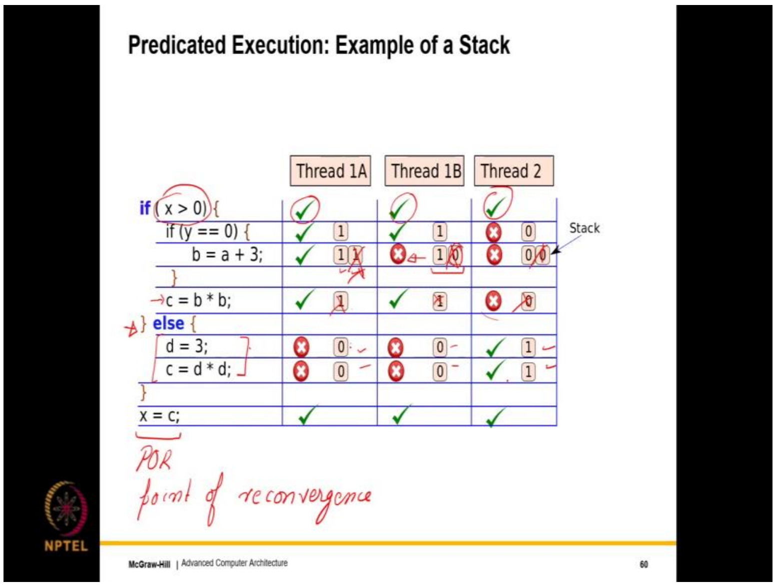
All the entries of the stack have to be 1, then only I can say that a given instruction is on the correct path, it should be executed, its result should be computed and committed to permanent state. otherwise, if any one of these entries is 0, then I will say that I am on the wrong path ok, and the instruction should not be executed.

So, there are two conditions over here, the first is an instruction executed if the stack is empty yes, that is obvious or as I have just described that the last in first out stack if all the entries in the stack are 1 so, this is so, we have a separate stack for each thread, this is not the programming stack, it is a separate stack which only stores the results of predicated execution.

So, if all the entries in this predicated stack are 1 so, we have 1 stack per thread, 1 per thread. So, we maintain 1 stack per thread in the warp. So, if all the entries are 1, then the instruction is on the correct path, and it is executed. So, when we leave the body of the if or else statement, we pop the step.

So, this is what is done that we push an entry when we enter the if statement and when we leave it, we pop an entry of the stack. So, who adds all of these codes? Well, all of this code is added by the CUDA runtime in terms of special instructions or directives or bits that are sent to the hardware and the hardware then picks up these queues and maintains the stack for the thread. So, the stack is maintained in hardware, but of course, the directives or when to push and pop are given by software.

(Refer Slide Time: 27:58)



Predicated Execution: Example of a Stack

	Thread 1A		Thread 1B		Thread 2	
if (x > 0) {	✓		✓		✓	
if (y == 0) {	✓	1	✓	1	✗	0
b = a + 3;	✓	1 ~~1~~	✗	1 ~~0~~	✗	0 ~~0~~
}						
c = b * b;	✓	~~1~~	✓	~~1~~	✗	~~0~~
} else {						
d = 3;	✗	0	✗	0	✓	1
c = d * d;	✗	0	✗	0	✓	1
}						
x = c;	✓		✓		✓	

Stack

POR
point of reconvergence

McGraw-Hill | Advanced Computer Architecture 60

Let us now look at a slightly longer example with three threads. So, in this case, we have nested if statements. So, the first if statement is executed by all three threads as you can see over here. Then, we have our first if statement. So, the first if statement is executed only by the first two threads and not by the third one which is thread 2. The reason being that this predicate for thread 2 is false, x > 0 for instructor. Then, we execute the body of this if statement.

So, as you can see the body of the if statement for thread 1 both the predicate bits are 1 and 1 which essentially indicate that it is on the correct path of thread 1A, but for thread 1B, even though it passed the first if statement, it was not able to pass the second if statement so, that is why for thread 1B, this predicate here is false so, that is the reason instead of put pushing a 1 onto the stack, we push a 0 and because we have 1, 0 valued entry, this instruction is on the wrong path of thread 1B and that is why this instruction is not going to be executed and it will be ignored.

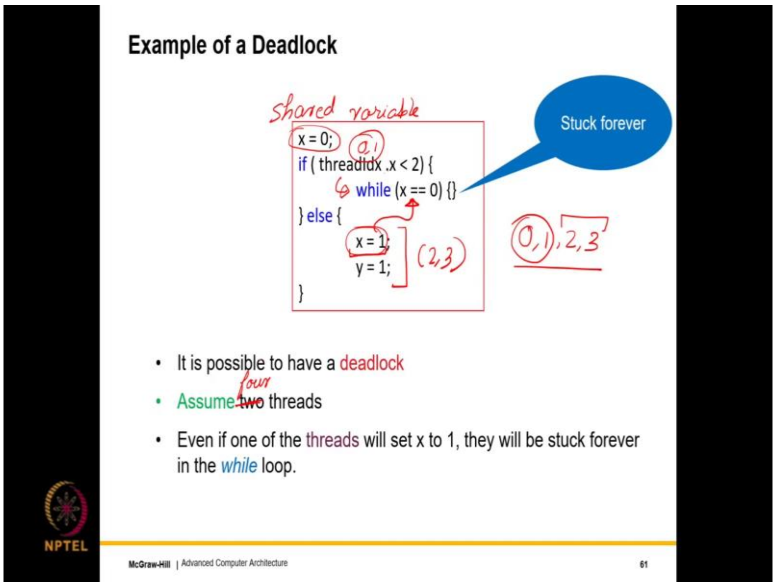
Similarly, since we are already on the wrong path, as I said we may decide to do a comparison, we may decide to omit it, in this case, we do not do a comparison for thread 2 because we are on the wrong path anyway so, we again push 0 onto the stack and so, the thread 2 also this instruction is not executed. At this point, threads 1A and 1B reconverge so, we pop the stack so, this entry is popped for all 3.

So, you are left with 1, 1 and 0. So, again for thread 2, this instruction is not executed whereas, for threads 1A and 1B, this instruction is executed. Finally, we have two instructions over here. For thread 1A and 1B, the else path for both of them is they are on the wrong path. So, how did we arrive from this state to this state? Well, the moment we exited the if statements body, you pop the stack, it became empty and then, we pushed 0 onto the stack because we are on the else path for thread 1A and 1B so, it is a wrong path.

So, you push 0 and 0 so, clearly for these two instructions, thread 1A and 1B will ignore them. But for thread 2, the else path is the correct path so, that is the reason we push 1 onto the stack so, thread 2 will execute them. So, as you can see in every stage, instructions are executed in lockstep.

So, basically the same instruction, instructions are executing quite well in fact, in lockstep and now, for the last instruction, which is x = c, this is where all the threads reconverts, it is called the POR or the point of the convergence. So, at this point, all the threads which are threads 1A, 1B and 2, they reconverge so, then what happens is that we execute the same instruction for all three threads.

(Refer Slide Time: 31:43)



Example of a Deadlock

```
x = 0;
if ( threadIdx.x < 2) {
        while (x == 0) {}
} else {
        x = 1;
        y = 1;
}
```

Shared variable
a,1
Stuck forever
(2,3)
0,1, 2,3

- It is possible to have a deadlock
- Assume ~~two~~ four threads
- Even if one of the threads will set x to 1, they will be stuck forever in the *while* loop.

McGraw-Hill | Advanced Computer Architecture 61

So, there is a good idea in the sense that using warps has made the operations of GPUs extremely efficient. However, such lockstep execution has a share of problems, and we will see why? So, assume that we have code like this. So, if we have code like, this let us assume that we take an internal variable x and we initialize it to 0. so, this is a shared variable and it is not a per thread variable, it is a shared variable, it is initialized to x.

Then, assume that we have four threads 0, 1, 2 and 3 so for the four threads, let us say for the first two threads, you will find the threadidx dot x < 2 why? Because it is 0 and 1. So; this x and this x are different so, this is the x coordinate of the thread index and this is the shared variables. so, they are different.

So, for thread 0 and 1, they will enter this if statement and because they are entering the other two threads, threads 2 and 3 will also enter because we execute this instruction where the same instruction in the warp is executed by all the threads. Of course, for the threads 2 and 3, this instruction is on the wrong path, but they will wait for their sister thread 0 and 1 to complete this. So, here, the fun begins.

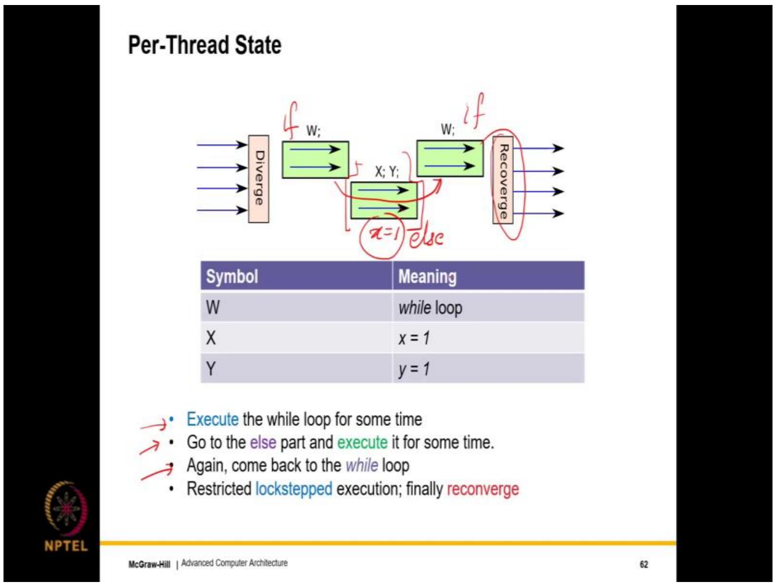
If we have a while loop over here which waits until x = 0, will be stuck forever. The reason will be stuck forever is basically because there is nobody who is going to set x to a value which is not 0, but if you look at this logic so, by this logic, threads 2 and 3 should be executing these two instructions which of course, are on the wrong path of 0 and 1, but for 2 and 3, they are on the correct path.

On the correct path, they will set x = 1 and this is going to release the while loop over here, but the point is that we will never come here. And so, basically the point is that we are never, never ever going to come here, and the reason is that the other two threads which are supposed to set x to 1 and release it, they will not come here because they will keep waiting for their sister threads which are 0 and 1 to exit and they are never going to exit the while loop.

So, we will be stuck forever even though we should not be. So, of course, we are not assuming two threads over here, we are assuming four threads. So, the point is that if you look at the simple logic as such there is no problem, but when you take a deeper look, you will realize that we are never going to reach this statement only because of our constraints on lock stepped execution within a warp.

So, this is of course, a slightly contrived example, but we will find many more such situations in real-life where such interactions do happen and many times, the code goes into a deadlock and the developer has no clue why. So, that is the reason in modern GPU design, such situations have been avoided.

(Refer Slide Time: 35:20)



Per-Thread State

Diverge
W;
X; Y;
W;
Recoverge
if
if
x=1
else

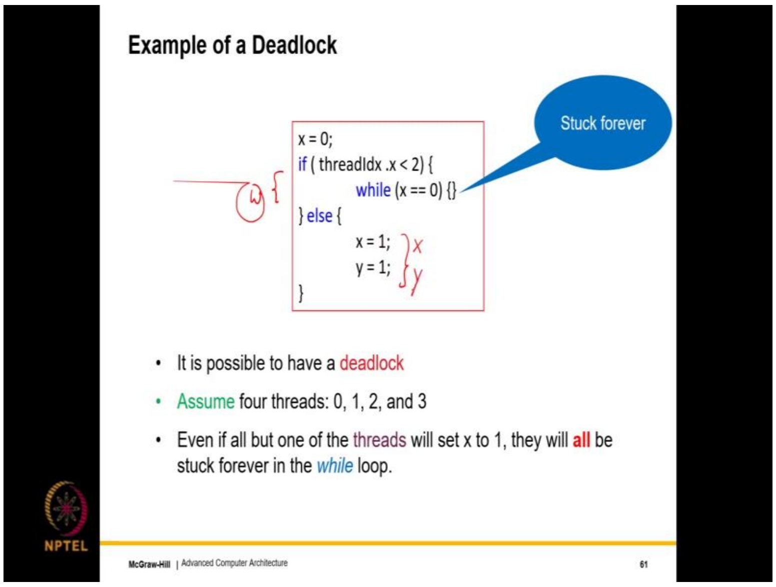
Symbol	Meaning
W	*while* loop
X	*x = 1*
Y	*y = 1*

- Execute the while loop for some time
- Go to the else part and execute it for some time.
- Again, come back to the *while* loop
- Restricted lockstepped execution; finally reconverge

McGraw-Hill | Advanced Computer Architecture 62

To precisely avoid such situations which may arise inadvertently in an execution, there is a notion of limited yet controlled thread divergence.

(Refer Slide Time: 35:41)



Example of a Deadlock

```
x = 0;
if ( threadIdx .x < 2) {
        while (x == 0) {}
} else {
        x = 1;
        y = 1;
}
```

Stuck forever
W {
X
Y

- It is possible to have a deadlock
- Assume four threads: 0, 1, 2, and 3
- Even if all but one of the threads will set x to 1, they will **all** be stuck forever in the *while* loop.

McGraw-Hill | Advanced Computer Architecture 61

So, let us consider the two code blocks. So, let this be the while loop, let us call it W and the other port block, let us just call it X for the one that is setting X = 1 and Y. So, what we can do is that we will not stop our basic lock step execution paradigm, we will go with what is called restricted lock step execution.

We will execute the while loop for some time which is the first point, we execute the while loop for some time, we see that we are not making progress. So, this can be seen that the while loop is just going on and on so, of course, we cannot detect an infinite loop, but at least we will see that we are there in one region of port for some time.

We can then go to the else part and again, execute it for some time. This again will follow the lockstep paradigm. So, all the threads will again execute the else part of course, all the predicate relations will be preserved so, only for all the threads whose predicates are all 1 in a stack, they will execute others will not, you have seen that. So, the X and Y blocks will be executed.

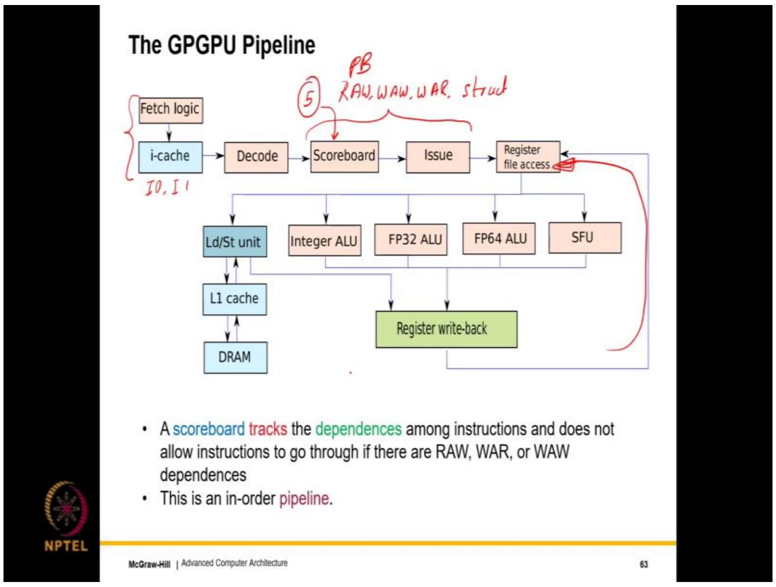
The advantage of doing this over here is that x = 1 will be set by some thread, does not matter who. Finally, we can come back to the while loop and the two threads which should have executed instructions in the while loop, they will execute it and basically, the others will ignore, but at least, they will exit the while loop.

So, since the if part and else part both the thread, all the threads have exited both, we can enter the point of reconvergence over here. So, what you are basically seeing is that we are again coming back to the while loop after executing the else part. So, this was the if part, this was the else part and this was the if part for the second time.

So, we are again coming back to the if part. So, this is an example of restricted lockstep execution where we are not really letting the threads diverge or go their own way. so, the threads are still executing together, they are still in lockstep mode so, we are referring to this as the restricted lockstep execution whereas, you can see there is this if else if kind of jump. So, this allows for reconvergence as we have seen.

And the deadlock situation that was arising here in the previous example, this is not arising over here. So, this was one of the big inventions that happened in NVIDIA Volta and has subsequently been kept and because this was required otherwise, there were many of these sticky kind of race conditions which were happening primarily because of our you know insistence target exists insistence on lockstep execution.

(Refer Slide Time: 38:46)



The GPGPU Pipeline

- A scoreboard tracks the dependences among instructions and does not allow instructions to go through if there are RAW, WAR, or WAW dependences
- This is an in-order pipeline.

McGraw-Hill | Advanced Computer Architecture 63

So, now, we are in the position to see the GPGPU pipeline kind of in full glory. So, this is let us say the pipeline of a PB. So, we have a fetch logic and instruction cache which uses both the I0 and I1 levels. Then, we have a decode. So, then the rest of the so, then, the scheduling part, the warp scheduling has.

So, the decode could be part of the fetch could be part of the warp scheduler, that does not matter, but the key aspect of the warp scheduler over here is that it uses a scoreboard. So, we have seen a scoreboard earlier so, that is the reason I am not explaining it once again.

We have seen it when we were discussing Itanium which was the last part of chapter 5. So, if you go to the video, the last part of chapter 5, you will find the videos on Itanium and the videos on Itanium have a discussion on the scoreboard. The scoreboard is nothing, but a hardware data structure that keeps an instruction stalled until there are no hazards.

So, the hazards are four types and structural hazards. Until these four hazards, there is no probability of any correctness problem happening because of these hazards, we just stole an instruction, after that we issue. So, since the scoreboard has been discussed in detail, I am skipping that part, but this was discussed in detail in Itanium and finally, we have the register file access.

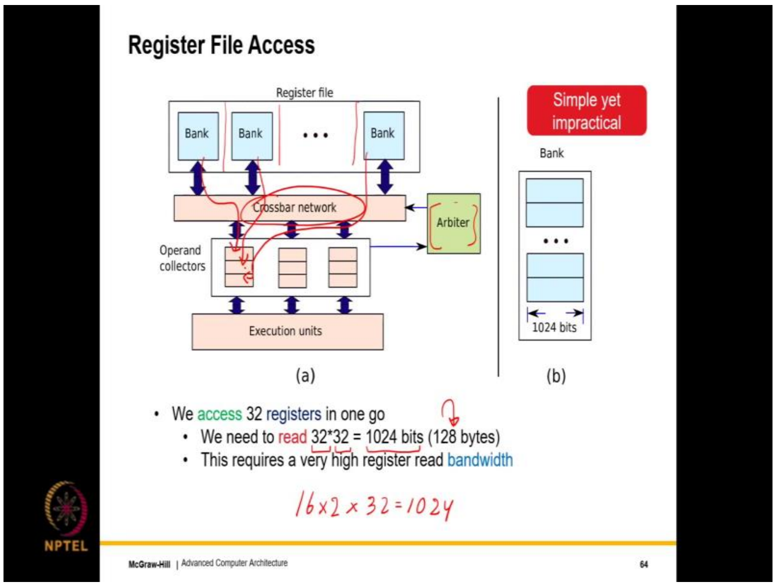
As you have seen the register file is an its looks like a memory in a PB and then, we have a bunch of these functional ALUs which could be the integer ALU, floating point, single precision, double precision ALUs, special function unit for sin, cos, transcendental functions, load stored units. Load store unit's stock to the L1 cache, the shared L1 cache and finally, to the DRAM.

So, I saw to add an L2 cache over here that should have been there, but still the GPU DRAM has a very large role to play in the GPU because a lot of the data because you are dealing with big data over here, a lot of the data ultimately ends up getting stored in the DRAM and for regular functional units, once the operands have been calculated, there is a register write back and we again write the results back to the register file.

This is as such a regular pipeline, the scoreboard is new, but the rest are not that great. The scoreboard basically tracks the dependencies and stalls instructions until the dependency is met and it is an in-order pipeline, it is not an out of order pipeline, it is primarily an in-ordered pipeline to a large extent, but of course, you will see in an ordered pipeline, WAW and WAR hazards do not come up.

So, basically, there is a very small amount of outer ordnance over here and which is sometimes if you have this kind of execution as I just showed on the previous slide, there is a limited amount of that, but in general, it is the job of the scoreboard to take care of these issues particularly, when we are executing threats in this fashion is the job of primarily the scoreboard to take care of it.

(Refer Slide Time: 42:21)



Register File Access

- We access 32 registers in one go
  - We need to read 32*32 = 1024 bits (128 bytes)
  - This requires a very high register read bandwidth

$16 \times 2 \times 32 = 1024$

McGraw-Hill | Advanced Computer Architecture 64

So, the important question that needs to be answered now is that if let us say we have 32 load instructions in warp or 32 store instructions pretty much one instruction per thread, what do we do, how do we access it? So, let us first count the number of bits that we need to read at the same time.

So, of course, same time need not mean the same cycle, but let us say in the window of two or three cycles. So, it is 32 X 32 well, why? Because we have 32 threads in a warp and each value is 32 bits so, this is 1024 bits or 128 bytes. So, this would require a very high register read bandwidth and a very high register write bandwidth as well.

So, if I were to argue in a different way so, you may say that no, instead of sending 32 requests per cycle, we may send 16 or we may send 8 well, that is fine. So, if we send 16, you should also factor in that fact that if let us say an instruction has two operands, we are not reading one operand, we are actually reading two so, this will become 16 X 2 = 32, again multiplied with 32, it will remain 1024 bits and let us see if we send 8 well, then it will become 512, but that is still a lot of bits.

A simple yet impractical solution could be that we just have a register file bank, a bank is nothing, but like a sub register file and then, each entry could be 1024 bits where all the entries are kept together, where all the information is kept together, this is impractical, this simple idea is impractical because we do not know in advance, which registers will access and whether they will all be kept at the same place or not so, this is rather impractical and does not work that way and also reading out so much of data takes a lot of time.

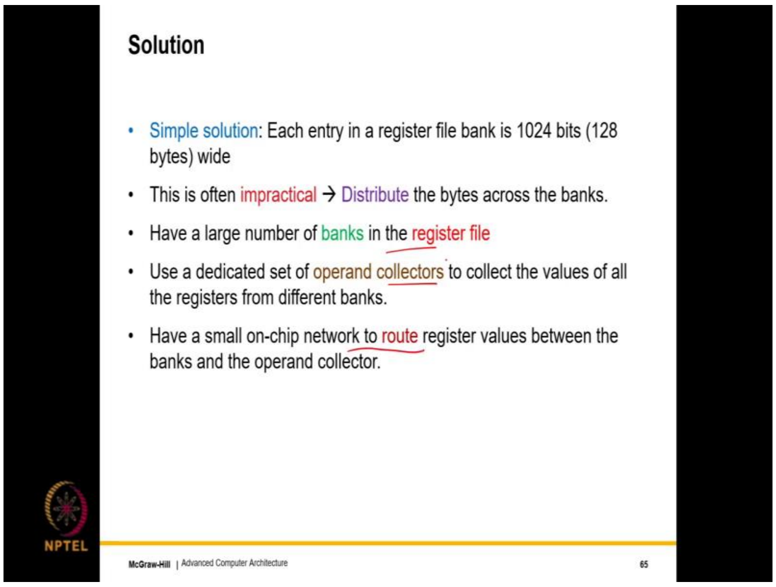
A far more efficient solution is to actually split the register file into multiple banks, sub register files and distribute these 1024 bits across these register file banks. So, one bank can keep let us say 64 bits or 128 bits. If let us say it keeps 128 bits, then across eight banks, the data will be distributed.

So, we will need to have an elaborate network so, that is what the crossbar network is which is controlled by an arbiter circuit. To access all those banks which have the data, read them or write them so, let us look at read because read is more performance and sitting and we put all of that data in a dedicated hardware structure called an operand collector.

So, the job of the operand collector over here is to basically work like a temporary buffer till the data is completely read from the registered file banks. So, clearly, we are accessing multiple banks. So, from each bank, we are reading a lesser amount of data. So, what is happening is that this is a fast access and also there is parallelism.

We have exported the complexity to the network that is the reason we need an intelligent arbiter, but once that is done, all the 1024 bits can be read and they can be stored in the operand collector and from the operand collector, the data can then seamlessly pass through to the execution units and the execution units can then execute on the data, execute the function whatever it is add, subtract, multiply on the data. So, the key innovations over here are the crossbar network, the arbiter and the operand collectors.

(Refer Slide Time: 46:24)



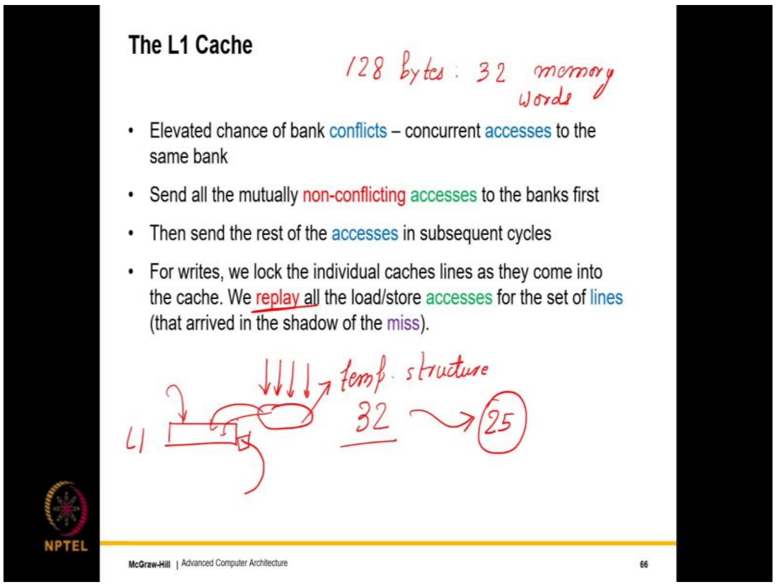
Solution

- Simple solution: Each entry in a register file bank is 1024 bits (128 bytes) wide
- This is often impractical → Distribute the bytes across the banks.
- Have a large number of banks in the register file
- Use a dedicated set of operand collectors to collect the values of all the registers from different banks.
- Have a small on-chip network to route register values between the banks and the operand collector.

McGraw-Hill | Advanced Computer Architecture 65

So, as we have discussed a simple, yet impractical solution is to just make a registered file bank entry wide, 1024 or 120 bits wide, that is not practical. You have to have a large number of banks in the register file, use the operand collectors to collect the values and then, use the on-chip network to route register and file values to the operand collector as we have seen.

(Refer Slide Time: 46:51)



The L1 Cache

128 bytes : 32 memory words

- Elevated chance of bank conflicts – concurrent accesses to the same bank
- Send all the mutually non-conflicting accesses to the banks first
- Then send the rest of the accesses in subsequent cycles
- For writes, we lock the individual caches lines as they come into the cache. We replay all the load/store accesses for the set of lines (that arrived in the shadow of the miss).

McGraw-Hill | Advanced Computer Architecture 66

Even in the L1 cache, when you are accessing the caches, not L2 to that extent, but L1 definitely. There is a far elevated chance of bank conflicts, the reason is if you are writing data, we are writing 128 bytes, or it is maybe easier to say 32 memory words for a word is 4 bytes. So, clearly, if you are accessing multiple banks, there is a chance of bank conflicts in the sense that a single bank may contain two of these words. So, if you have concurrent accesses to the same bank, that is a problem.

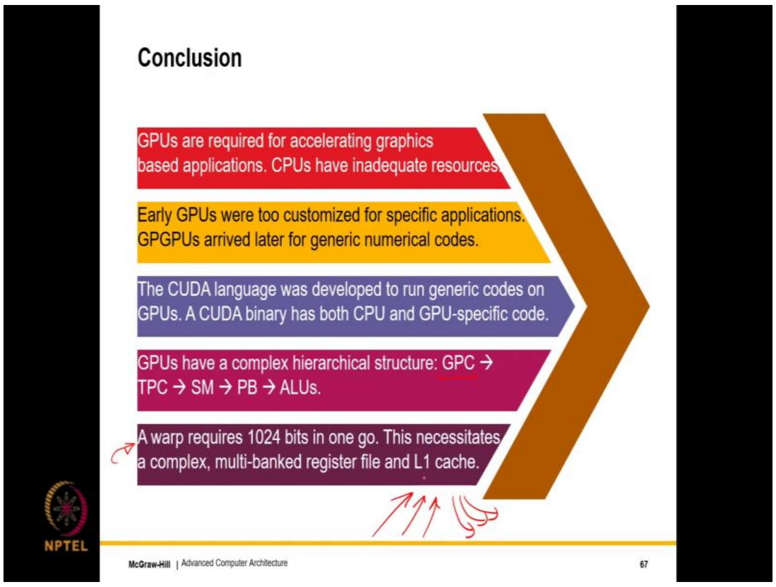
So, what we can do is we can have a dedicated piece of hardware that will send all the mutually non-conflicting accesses to the banks first, sort of let us say all the accesses, let us say 32, we might find 25 accesses that are mutually non-conflict, we send them first. Then, in the rest of that, then the rest of the accesses are sent to subsequent cycles, are sent to the banks and subsequent cycles were gradually drained up.

For writes, what we can do is that let us say there is a cache line, then what will happen is the data will either gradually come in because of bank conflicts and so on so, whenever we bring in a line from a lower level, we can lock the line in the cache and let us say we will have an MSHR kind of structure so, we will discuss that in great detail in chapter 7, but what we can do is that let us say, I will give a realistic scenario.

So, let us say that in the L1 cache, there is a write miss so, the clock has to come from L2. In the time being, a lot of reads and writes would have arrived at the L1 cache so, they are buffered in a temporary structure, they will be called an MSHR in the next chapter, but in this chapter, let us call it a temporary structure.

So, the temporary structure will hold on to all of the outstanding reads and writes and the moment that the block comes from the lower level, will basically replay all the load store accesses for the set of all the lines that arrived in the shadow of the miss which means when the miss was being serviced from the lower level, whatever reads and writes we got we immediately apply them to this block over here.

(Refer Slide Time: 49:18)



Conclusion

GPUs are required for accelerating graphics based applications. CPUs have inadequate resources.

Early GPUs were too customized for specific applications. GPGPUs arrived later for generic numerical codes.

The CUDA language was developed to run generic codes on GPUs. A CUDA binary has both CPU and GPU-specific code.

GPUs have a complex hierarchical structure: GPC → TPC → SM → PB → ALUs.

A warp requires 1024 bits in one go. This necessitates a complex, multi-banked register file and L1 cache.

McGraw-Hill | Advanced Computer Architecture 67

So, now, we have reached the end of the chapter. So, let me quickly conclude, you always have five concluding points. GPUs are no doubt a vital requirement a modern numerical scientific and high-performance templated. They are required for accelerating graphics-based applications, no doubt. CPUs in comparison has severely inadequate resources.

Early-stage, GPUs were too customized for specific applications, but then GPGPUs arrived on the scene, and they had a very generic application. In the CUDA language was developed to basically run generic codes on GPUs. A CUDA binary is a fat binary that has both GPU and GPU specific code, CPU and GPU specific code and GPUs have a complex hierarchical structure with 5 to 10000 cores these days which is massive, several teraflops of computational power is there within one single GPU.

So, here, we have a GPC, then a TPC, SM, PB and a bunch of ALUs. A warp requires 1024 bits in one go. This necessitates a complex and a multi-banked resistor file and a complex multibank L1 cache as well. So, we will discuss in the next chapter on caches, a lot more about the way that these caches are designed because what you would have realized by this time which is at this time meaning that we have completed the first part of the book where we have discussed out of order pipelines and GPUs.

And it is possible to embedded c of computational units and increase the IPC for some codes not for all, for the memory system can be bottle because the memory system also needs to provide data at a high-rate and also read out results at a high rate correct, the memory system could be a part.

Given the fact that the memory system is a bottleneck or can be one, we should invest a lot of our resources in designing a very fast and efficient memory system. So, this will be the subject of study in the second part of the book which is from chapter 7 to 10 and the lectures for this you will find on the website for chapters 7 to 10. so, kindly follow.