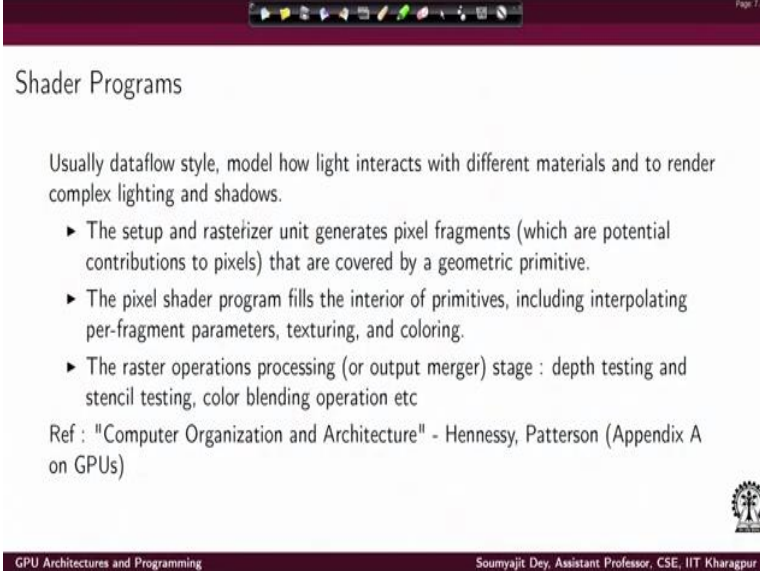


GPU Architectures and Programming
Prof R. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No. 6
Intro to GPU Architectures (Contd.)

Alright, so we will start from where we left off,

(Refer Slide Time: 00:27)



Shader Programs

Usually dataflow style, model how light interacts with different materials and to render complex lighting and shadows.

- ▶ The setup and rasterizer unit generates pixel fragments (which are potential contributions to pixels) that are covered by a geometric primitive.
- ▶ The pixel shader program fills the interior of primitives, including interpolating per-fragment parameters, texturing, and coloring.
- ▶ The raster operations processing (or output merger) stage : depth testing and stencil testing, color blending operation etc

Ref : "Computer Organization and Architecture" - Hennessy, Patterson (Appendix A on GPUs)

GPU Architectures and Programming Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Which was the parallelism available in shader programs, as we know that these are basic data flow style of computation, and the primary work of shader programs is to model how light interacts with different materials and accordingly to render complex lighting and shadows. So a typical shader program will will kind of interact with several specific units. That generate some specific components of output graphics.

So for example, there will be the setup and rasterizer unit that will generate the pixel fragments that are kind of potential contributions to pixels, and covered by a geometric primitive, the pixel shader program is the one that fills up the interior of the primitive. So the first one, generates the fragments to be covered by the geometric primitive the interior will be kind of painted by the pixel shader program.


So that also gives a hint that this is the block which will be more compute intensive. It includes interpolating power fragment parameters, textures, as well as coloring. And by raster operations, we mean the stage where things like color blending testing for stain style, as well as depth testing these are being done right? And.

(Refer Slide Time: 01:50)

GPUs : massive multi-threading

Design goals

- ▶ Cover the latency of memory loads and texture fetches from DRAM
- ▶ Support fine-grained parallel graphics shader (and general parallel compute) programming models
- ▶ Virtualize the physical processors as threads and thread blocks to provide transparent scalability
- ▶ Simplify the parallel programming model to writing a serial program for one thread



GPU Architectures and Programming Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So these are also specific units, if you look at the picture here.

(Refer Slide Time: 01:55)

GPUs

Ideas from parallel instruction handling by vector architectures, ILP techniques etc were borrowed to accelerate graphics processing

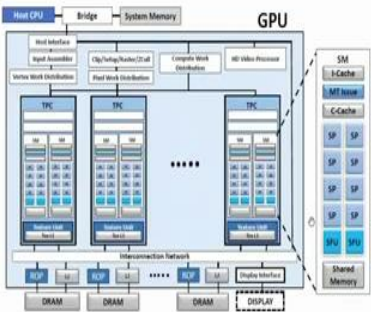



Figure: GPU systems (GeForce 8800) - Hennessy, Patterson (reproduced)



GPU Architectures and Programming Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So you have the input assembler the vertex world distributor, the pixel wall distributor. And this is units, which are present in this block diagram.

(Refer Slide Time: 02:10)

Early GPUs

Early GPUs accelerated the logical graphics pipeline

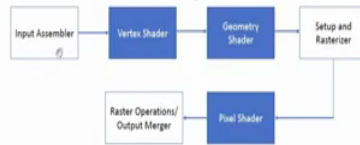


Figure: Graphics logical pipeline

And they also nicely map to the graphics logical pipeline containing the input assembler vertex shader geometry shader rasterizer, the pixel shader as well as the output module phases. So we want to live, I understand that for a specific shader program. We have the setup rasterizer unit in the pixel shader program, and the output merging, or the output processing phase.

(Refer Slide Time: 02:38)

GPUs : massive multi-threading

Design goals

- ▶ Cover the latency of memory loads and texture fetches from DRAM
- ▶ Support fine-grained parallel graphics shader (and general parallel compute) programming models
- ▶ Virtualize the physical processors as threads and thread blocks to provide transparent scalability
- ▶ Simplify the parallel programming model to writing a serial program for one thread

So we will see that how this kind of graphics processing, nice map this, they will kind of nicely map, two GPUs which support this massive multiple threaded computation. So for GPUs. The primary goal at its early age was to provide a solution, which was partly programmable. It was, it should be able to accelerate the working of this kind of a graphics pipeline. The reason being that you want the graphics to be rendered.

With as much resolution as possible as fast as at the highest rated possible the higher rate at which you can render the graphics, the images would seen the movements have seen that many that much realistic right? So the overall goal was to cover the latency of memory loads and texture features from the DRAM. So in order to activate the graphics pipeline very frequently, you would like to fetch this data from the DRAM with that much high frequency as possible.

Or, you have to fetch in a large width, that means in each transaction use flatfish lot of data points and work of them work on them in parallel. And that is how you can cover the latency of memory loss is what it means by covering the latency. And then comes the point that, well, you have fetched a lot of data points for the graphics pipeline to work or the graphic should have programs to work.

You should be able to work on them in parallel and to their compute right? So this is, this helps to accelerate also general parallel compute programs, but the primary goal was to accelerate graphics shaders in parallel. The third goal was to virtualize physical processes threads. And thread blocks to provide transparence scalability. Now this is an interesting point. We are trying to say that, see parallelism is not a specialized thing.

Rather let us look at every operation as paralyzed think that that is the general case and try to accelerate parallel operations. That means, you always think in parallel, you start thinking that you have a thread of computation, where every computation has the ability to work on parallel number of on upon a set of vectors instead of scaler points. So with this design goes, this, this idea of GPU design started.

And also the overall objective comprise that you should be able to simplify the parallel programming model by simply making a programmer write a serial code that says there's about what one thread of competition is going to do. And it's just that you write a sequential behaviour of a single thread, and it's just a case that the thread works on multiple data points in parallel, for doing all the operations sequence theory.


(Refer Slide Time: 05:45)

Page 7/10

First generation GPUs

- ▶ GeForce 256, introduced in 1999
- ▶ Contained fixed function vertex, pixel shaders programmed with OpenGL and the Microsoft DX7 API
- ▶ GeForce 3 - the first programmable vertex processor executing vertex shaders

- Ref for contents and here and subsequent places : "NVIDIA Tesla: A Unified Graphics and Computing Architecture" by Erik Lindholm, John Nickolls, Stuart Oberman, John Montrym, (NVIDIA) IEEE Micro, Volume 28, Issue 2, March 2008



GPU Architectures and Programming Soumyajit Dey, Assistant

So, what were the first generation GPUs. So these Geforce 256 was introduced in 1999. So one of the first GPUs. And the content fixed function vertex pixel shaders, which were programmed with OpenGL and model Microsoft DX7 API. So, the graphics pipeline for these GPU, still content fixed function vertex and pixel shaders that means they were not software programmable in the sense that they can be.

These are hardware blocks which understood the lower level API calls from OpenGL and Microsoft DX7. Then came Geforce 3, and this was the first programmable vertex processor was which was executing vertex shaders. So what happened in with them was this blocks of the graphics pipeline, the blocks highlighted in blue. That is a vertex shader the geometry shader and pixel shader.

They instead of being fixed function blocks on the hardware, they became programmable blocks, that means the GPU will be having this general purpose programming interface through which it can run a vertex shader also a geometry shader problems also opens, these pixel shader programs. So for this, and all subsequent many contents are many places, we will be referring to this nice paper that came up in IEEE micro in the year 2008.

The paper is titled NVIDIA Tesla unified graphics and computing architecture. So one of the position papers on this large initiative that was published at that point of time.

(Refer Slide Time: 07:37)

Trade-off

- ▶ Vertex processors were designed for low-latency, high-precision math operations
- ▶ pixel-fragment processors were optimized for high-latency, lower-precision texture filtering - typically more busy (considering large triangulation)
- ▶ if these are fixed function blocks - difficult to select a fixed processor ratio
- ▶ Primary design objective for Tesla architecture - execute vertex and pixel-fragment shader programs on the same unified processor.
- ▶ Unification helps in 1) dynamic load balancing of varying vertex- and pixel-processing workloads, 2) introducing other shaders



GPU Architectures and Programming

Soumyajit Dey, Assistant

Now, what was that trade off that the designers of this Tesla, architecture, or the Tesla family of GPUs were really looking at. So, if you look at what is the activity of a vertex shader program, which was to be run earlier by vertex processors, as I said, the fixed functional blocks. Which would be part of the early GPUs or early, early graphics systems that there will be a specific fixed function hardware block which will do vertex processing it will be a fixed function block which will do the pixel shader processing and all that.

So for them. They have quite well defined objectives the vertex processors will operate on coordinates, but they will implement high precision math operations at low latency. So that was the design objective of a fixed function vertex processor when it came to the pixel fragment processing, they were optimized for high latency, but low precision texture filtering. They are typically more busy, considering large triangulations.

So, if I consider large triangulation of an image, there will be lot of pixel points inside. Specific triangle. So processing at the pixel level will be having a lot of compute load for that. There should, need not be done at a high precision level, but the objective is to do more of them in parallel right? Now, if this vertex and pixel processors are implemented as fixed function blocks. It is difficult to select a fixed processor ratio.

That means how much throughput should be allowed for the vertex processor and how much throughput should be allowed for the pixel processor, because that also. Because overall how much this pipeline will work at what is overall throughput that also depends on the input data input images triangulation scheme. So, if the, if it's really as has been thought here that is large triangulation, then the vertex processors will be operating on lesser number of points.

So this idea would hold that yes it can work on lesser number of points but compute in high position, whereas the next level will have lot of things to do part triangle. So they work on low position. But how about the triangulation is considered in input image as small, so then the vertex processor will have a lot of things to do, because there are a lot of triangles in the input lot of purchases to process, but the for the pixel process and part triangle job is reduced, right?

So, if they are implemented in hardware as fixed function blocks. What is an ideal image scenario that I mean, the overall throughput of this sequence of fixed function blocks would actually be a function of how the input image has been triangulated and how it has been provided the common case may get accelerated, but the cases which are not common with suffer from performers from performance in video, right?

So, the primary design objective for the GPUs. The Tesla architectures was that they should be. They should be these different things pixel fragment shader programs. They need not be implemented as fixed function blocks but they should be implemented uniformly on a program on a programmable processor, such that the same, the same processing fabric can be used for both of them.

So that, depending on dynamically, what is the input triangulation, it can be decided how much? How many, how many threads will be doing the computation for vertex processing and how many threads will be doing the computation for pixel processing, so that overall we have a very high throughput. Now this unification helps in. As we have discussed the dynamic load balancing of these heading throughputs of vertex and pixel processing.

And it also helps in introducing many other possible shaders. That can be part of the time for part of the graphics pipeline. As we know this is always the case when you move some hardware functionality to a software functionality with increased programmability. You can do more dynamic load balancing of things. So essentially, you move out from this fixed fixed function blocks. Make them all part of a programmable parallel processor.

That is the GPU, and you just dynamically decide how many threads will be computing for the vertex or the pixel, what is good for the overall high throughput functionality required.

(Refer Slide Time: 12:18)

Tesla architecture

We come back to GeForce 8800 GPU with 128 SPs organized as 16 SMs

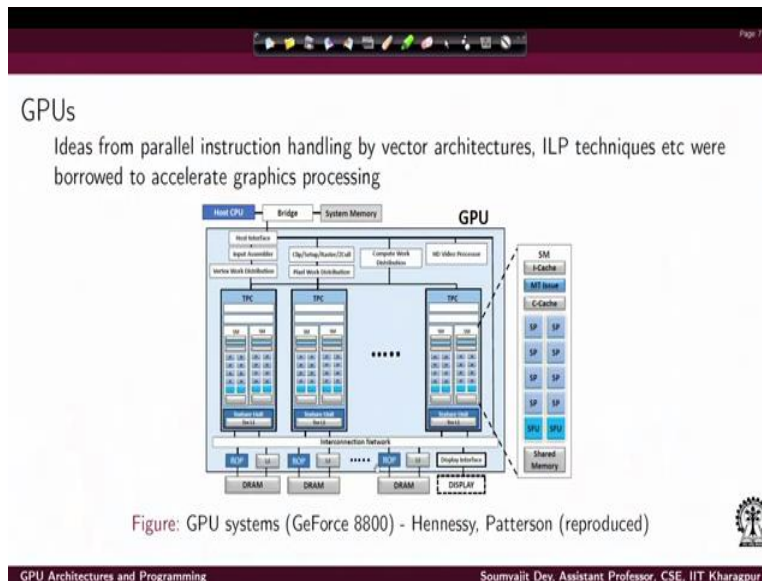
- ▶ external DRAM control and fixed-function raster operation processors (ROPs) perform color and depth frame buffer operations directly on memory
- ▶ The interconnection network carries computed pixel-fragment colors and depth values from SPs to the ROPs
- ▶ The network also routes texture memory read requests from the SP to DRAM and read data from DRAM through a level-2 cache back to the SPs

GPU Architectures and Programming Soumyajit Dey, Assistant

So coming to this Tesla architecture. We, we speak, we speak of, again this GeForce 8800 GPU with 128 SPs that are organized as 16 streaming multiprocessor for SMs. So in this specific GPU, you have an external DRAM control. You have fixed function raster operation processors are opis, which perform color and deep from a buffer operations directly on the beam memory. So, given that you have this ROPs is present.

It's possible that to do this specific operations on color and framework for depth directly on memory, instead of bringing them for doing some compute on the main processor pipeline, the interconnection network carries these computed pixel fragments colors and depth values from the SPs to the ROPs. So, just to get a feeling of the connectivity. If we look back here into the picture that we had for our GPU.

(Refer Slide Time: 13:29)



So we have this ROPs being connected with the interconnection network. And also, so the interconnection network is here connecting the different SPs, which contain the SMs and also the species. And this interconnection network also connects the ROPs piece, and these connect to DRAMS through this hierarchy of level two caches. Now coming to our point here. So, this external DVM control, and the ROPs.

They perform color and depth depth framework for operations directly on the memory, the interconnection network carries the computed pixel fragment colors, so they carry the output pixel fragment colors and depth values from the from the scalar process to the ROPs for doing the fixed function raster operation. And then they get stored back to the memory. The network also routes texture memory read requests from the scalar process to the DRAM,

And they read data from the DRAM through a level two cache back to the SPs. So, this auto piece that is the raster operation for us. They still remain as fixed function blocks. And they did they do their operations and they stole back the data to the memory of back and forth to the hierarchy of L2 to caches. And the interconnection network carries this computed pixel fragment colors and depth values. We are from from SPs to the ROPs back right?

(Refer Slide Time: 15:26)

Graphics in Tesla

- ▶ The input assembler collects vertex work
- ▶ Vertex work distributor distributes vertex work packets to the various TPCs
- ▶ The TPCs execute vertex/geometry shader programs
- ▶ output data is written to on-chip buffers
- ▶ buffers then pass their results to the viewport/clip/setup/raster/zcull block

We continue from here to general purpose processing

GPU Architectures and Programming Soumyajit Dey, Asstistat

So coming to the graphics in the Tesla what are the different blocks that are present in the architecture. So, first of all there's the input assembler which collects the vertex work to be done. And then you have the vertice work distributor, that distributes this work packets to the different TPCs that are present. And this TPCs that are going to execute the final shader programs. Both the vertex as well as the geometry shaders.

Now this is where as we already spoken that the programmability comes in earlier vertex and geometry shaders also used to be fixed function blocks, but now they become a part of the programmable pipeline. So, with this. Just to finalize once again, you have this raster operation still in fixed function blocks the data flows for between these, these, these raster of these ROPs and the end the DRAM through this level to hierarchy.

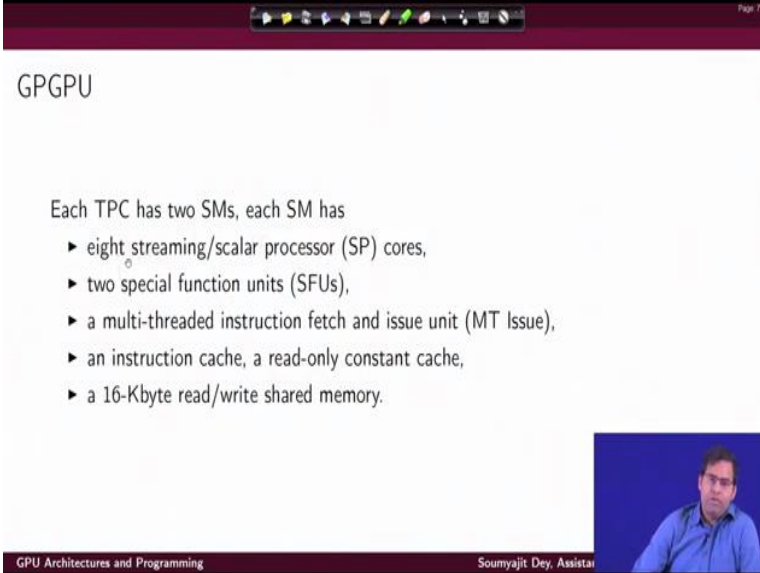
And the interconnection network will carry the fixed fragment colors and depth from the SPs to the ROPs for doing the rester operations, of cores the SPs are there is to execute the different shader operations and coming back here, the TPCs is will execute this sheder operations and output data will be returned to the on-chip buffers, Which will further be passing the result to the viewport, or the rest of law.

So this is how as we can see that the original graphics pipeline gets represented in the Tesla architecture with the shader operations, being done in the scaler process, whereas you have the

raster operations, being done in the fixed function blocks, and also the buffers. The over all outputs are written to achieve buffers and they pass the result to the viewport. So, this gives us a basic idea of how graphics processing changed from fixed function pipeline to a GPU pipeline.

And from here we will continue to how GPUs in general handle instruction processing in more and more in the SIMT style. And we will talk about general purpose programs.

(Refer Slide Time: 18:03)



The screenshot shows a presentation slide with a dark red header and footer. The title 'GPGPU' is centered at the top. Below it, the text reads 'Each TPC has two SMs, each SM has' followed by a bulleted list of components. A small video inset in the bottom right shows a man speaking. The footer contains the text 'GPU Architectures and Programming' and 'Soumyajit Dey, Assistant'.

GPGPU

Each TPC has two SMs, each SM has

- ▶ eight streaming/scalar processor (SP) cores,
- ▶ two special function units (SFUs),
- ▶ a multi-threaded instruction fetch and issue unit (MT Issue),
- ▶ an instruction cache, a read-only constant cache,
- ▶ a 16-Kbyte read/write shared memory.

GPU Architectures and Programming Soumyajit Dey, Assistant

Coming back to the way the different functional areas in the GPU. So you have each of these two pieces. They contain two streaming multi processors. Each of the streaming multi processors have got eight streaming or scalar processors. So this is scalar processors, along with them. I mean this SMs will also have two special found function units or the SFs. And there is a multi threaded instruction fetch and issue unit.

So again if we just go back to this figure. So, this is the MCs we are already back so as you can see, we have this input assembler like we discussed from the graphics pipeline, then the vertex world distribution, which will distribute work, and also the pixel distribution that is to the workers and pixel works to the different TPCs. And finally we have the auto pieces fixed function props.

Also, the texture units as fetched from blocks and outputs finally go to the viewport. Now, coming here to the core, the internal content of each of the streaming multiprocessors. So you have the I-cache for the instruction fetching instructions. And then you have this multi threaded issue unit, which is going to issue instructions in parallel to each of this scalar process or streaming process.

As we have discussed this SM each SM has got eight scalar processors. And also, two special function units or is a fuse. So, there is a multi threaded instruction fetch and issue unit is supposed to issue the instructions that are going to be executed by the scalar cores

And also, as we have seen in that picture we have a 16 kB read write shared memory. As part of streaming multiprocessor. So, what is there inside each of these streaming multiprocessors In summary, each SP core will contain a scalar, multiply and add unit. Does that mean inside each streaming multiprocessor you have got eight scalar, multiply and add units. Also the SM. As we have seen, it has got to have this special function units.

And this special function units can be used for doing some transcendental computation. Also, each of this is a fuse content four floating point multipliers. So for all I can say that inside each SM, the streaming multiprocessor I have got eight multipliers and add units. And also, eight floating point multipliers, I can do eight floating point multiplication operation in parallel. And also, eight scalar multiply operations in parallel.

(Refer Slide Time: 20:39)

The image shows a presentation slide with a dark red header and footer. The slide title is 'SIMT'. Below the title, the text reads 'GPU execution model' followed by three bullet points. The first bullet point states that SIMT architecture is similar to SIMD design, applying one instruction to multiple data lanes. The second bullet point notes that SIMT applies one instruction to multiple independent threads in parallel, not just multiple data lanes. The third bullet point explains that while a SIMD instruction controls a vector of multiple data lanes together, a SIMT instruction controls the execution and branching behavior of one thread. In the bottom right corner of the slide, there is a small video inset showing a man in a blue shirt. The footer of the slide contains the text 'GPU Architectures and Programming' and 'Soumyajit Dey, Assistant'.

Page 7/10

SIMT

GPU execution model

- ▶ SIMD architecture is similar to SIMD design, which applies one instruction to multiple data lanes.
- ▶ The difference is that SIMT applies one instruction to multiple independent threads in parallel, not just multiple data lanes.
- ▶ A SIMD instruction controls a vector of multiple data lanes together, a SIMT instruction controls the execution and branching behavior of one thread.

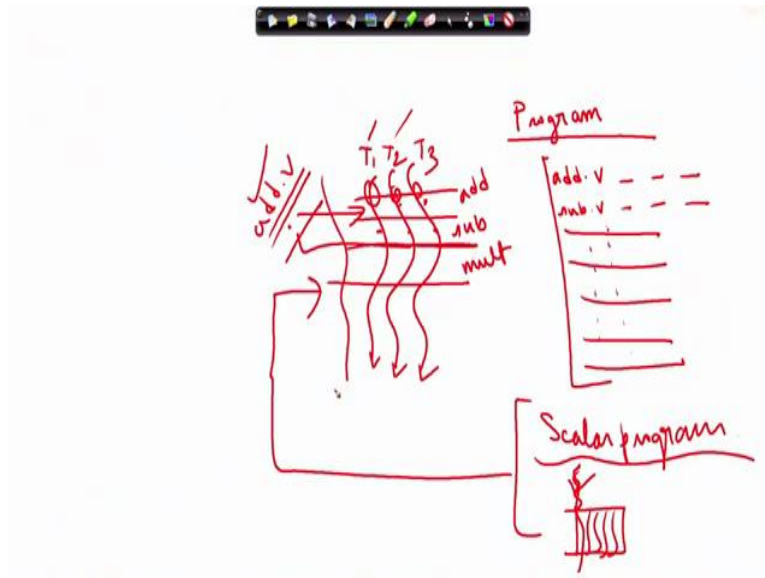
GPU Architectures and Programming Soumyajit Dey, Assistant

Now, coming to the execution model. So, as we have seen this idea of data parallel computation by instructions is well known as a single instruction multiple data. This is the idea. I mean, which amended it also from vector processing. And this idea, gives much more generalized. When people thought of the execution model for GPUs. So what people started talking about is size to a single instruction multiple threads.

So that would mean that you have multiple threads of computation, or in a different way, I would say that I have a single thread of computation. It's executing a sequence of instructions, each instruction is working on multiple parallel vectors vector, vector type data points. So looking at the other way I can just say that. Okay. Since I have a single instruction processing, but is doing the same job for different parallel threads of execution.

So that gives me that idea of what is SIMT that is instead of saying single instruction on multiple data. If those data points, all belong to parallel threads which are making progress in parallel, I would start calling them single instruction multiple threads. So, with single instruction multiple threads. I have a more generic model of instruction processing, because essentially what I am saying that, okay. To let us let us just take a simple example here.

(Refer Slide Time: 22:21)



So, when I am executing a sequence of vector instructions. In general, they may or may not, they may be unrelated. That means. Well, they can be doing computation towards a similar objective, or they may not be doing so right? I mean, it depends on how the processor has figured out the parallelism, and it has found that Okay, I have multiple the thread instructions I can process them in parallel and all that.

But now suppose I say I there. Okay, I have multiple threads of execution. So this is thread one thread, two. And this is three three. And I say that all the threads are going to do an add operation here. And then all the threads are willing to do some subtract operation here. And then, all the threads are going to do some mult operation here. This is the demand of the program. When I say that, looking at it from another way I can start saying that okay.

For this unit and assembly instruction. Because will. we are looking to do an end not be processing, this a vector processing. But when can you do that, only if the warp for each of the threads are such that they are similar, this thread also needs to do add this thread also needs to do an add this thread also do an add. Once that is done, this thread will do a sub this thread will do a sub.

And not only that, when, when this thread is doing the add, and the next thread is also doing the add. They are doing the add on consecutive items they are all doing the work on specific factors.

So, just when I start looking at a program where all instructions are vectorized is not that I have a program, which is basically a scalar program. And for extracting parallelism purpose. The compiler had figured out that okay here, something is done in parallel.

And then again, there is some vector processing that can be done. It's not like that. We are going to generalize this concept. And we are starting to say that no, the entire program is like a vectorized operation. Just start looking at it in a way that with each operation you have a set of threads that are making progress. Then I can say that I have a sequence of these kind of instructions or single instruction, which is operating on multiple threads.

Once that is done, again I have another surge instruction which will again operate on multiple threads. And that is how the computation progresses. So coming to SMIT Instead of starting to think just to just doing a basic summary again, because this is the most important thing here. Instead of saying that, okay. It's not that you are doing some program execution, where some part of the code is scalar.

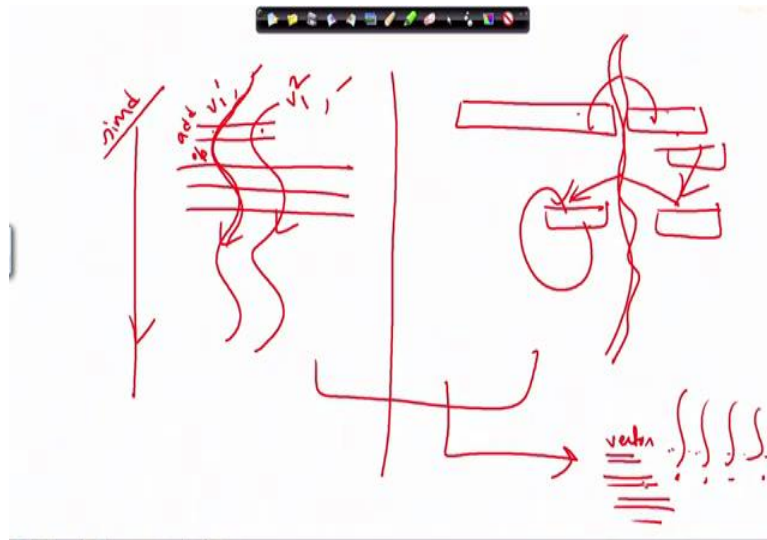
Some part of the code, the compiler has identified to the data parallel and hence this as executed, it has immediate some written instructions. Let us generalize that. let us think that every instruction execution is of some type vector. And not only that, the entire sequence of instructions is such that they represent the progress together by multiple parallel threads of computation.

When we start talking like that with mean that okay, we are thinking of a programming model, where I have a sequence of instructions, which is such that every instruction is like an SMID instruction. Because, and. And not only that, every instruction is doing an SMID operation which is part of some set collection of threads. Then again, there is another assembly instruction which is again making progress with the operation for that specific collection of threads.

And in that way things are going on. So in that way, what we have is a nice data parallel program. And that data parallel program has got multiple threads of computation. And they are all being executed synchronously by a sequence of instructions. Alternatively, I can look at it,

just like this, that, okay I have a single thread of execution, but in each point of execution is working on multiple different data points. So just to make things very clear

(Refer Slide Time: 27:55)



And not to get confused. I can say that these two pictures are quite equivalent. I have two threads, which are executing like this. But whatever they do here. There is an add on vector's so comfortable, so there is vector component v_1 vector component v_1 . I mean, sorry v_{11} and v_{12} . And then again, so there is the first operand, and then there is some second operand. They're doing an add. Then again, they are doing some other operation, like that.

So I can say that I have a sequence of SIMD operations going on on multiple different threads in parallel. Alternatively, I can say that I have data, sitting in vector datatypes is one thread going on which is going on, and doing a computation on them, producing a result. And then again it is doing a computation on some other vectors. And again, producing a result, while there may be dependencies and all that.

So there's less like thinking that I have a single thread, which is making progress, doing operations on parallel data vectors, or I can think that I have in the in that way. I have multiple threads, which are making progress in parallel. But both of these pictures at the instruction level maps to a scenario that I have a vector instruction, which is doing computation for different data points, belonging to these different threads

Followed by another vector instruction, doing some other computation to this same set of threads, and they are overall making some progress. So just to summarize, that when I speak of SMIT, I can say that it's about one instruction, getting applied to multiple independent threads in parallel, not just multiple data lanes. And one SIMD instruction controls of vector of multiple data elements together,

Whereas an SIMT instruction controls the execution and branching behavior of one thread. So, this may sound confusing, but this is what it means, when I talk about SIMD instruction. It controls a vector of multiple data lanes together. So, you have rectilinear type sitting there the type of data sitting on multiple data lanes and the same assembly instruction is operating on them right.

When I say SIMT, it controls the execution and branching behavior of one thread. Because when there is a branching happening. These different threads may have different behaviors. That is why you have this instruction, which would mean something for one of the threads, but it may mean something else, it may be. I have one SIMT instruction which will mean something for one thread.

But, essentially also going to mean something completely else, a different branching style for someone that thread. This is something will explain explain in more detail later on, further time being, just remember those two pictures that were discussing that overall idea is very simple. When I talk about is SIMT is just like saying that will lives generalize this concept of single instruction multiple data and start thinking

That you have instructions operating across multiple programs are there lightweight versions that is threads and all the threads are making progress in log steps That means, fundamentally, all your instructions. Are SMIDs so you have one assembly instruction operating on different program threads. Again, another SMIDs instruction operating on different programs threads, so on so forth. And this is how it keeps on continuing.

(Refer Slide Time: 32:04)

The image shows a presentation slide with a dark red header and footer. The slide title is 'SIMT'. Below the title is a list of five bullet points. In the bottom right corner of the slide area, there is a small video inset showing a man in a blue shirt. The footer contains the text 'GPU Architectures and Programming' and 'Soumyajit Dey, Assistant'.

Page 12/13

SIMT

- ▶ In contrast to SIMD vector architectures, SIMT enables programmers to write thread level parallel code for independent threads as well as data-parallel code for coordinated threads
- ▶ SIMT - essentially a single thread of SIMD instructions
- ▶ Each SM's multithreaded instruction unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*
- ▶ Each SM manages a pool of 24 warps, with a total of 768 threads
- ▶ Each SM maps warp threads to the SP cores

GPU Architectures and Programming Soumyajit Dey, Assistant

So, in contrast with SIMD vector, SIMT will enable programmers to write thread level parallel code for independent threads, as well as a data parallel code for coordinated threads, this understand what we mean. So, as we have said this is SIMT, that means a single instruction working across multiple threads. But when you look at the behavior of the program, you have multiple threads.

But you need to specify the functionality of only one thread, how that thread works for its, its data points and all that right? So, I can say that it enables the programmers to write the thread level parallel code for one trip. If different threads are supposed to do different jobs, then there will be some more complex code to write, which we will see that how that works. But ideally you are going to write a part thread behavior.

And that is going to be replicated across the different parts of the different the different parts of the data segment. So in that way I can say that you said it is essentially a single thread of SIMD instructions so there's the alternate due taking on that I have essentially a single set of execution, but all the instructions are executing as a SIMT. So each of these streaming multi processors multi threaded instruction unit.

They are going to create, manage and schedule and execute threads. So they will spawn the threads, They will manage their execution, they will map them to the real hardware that is the

scalar processor or SPs, and the finally they will say that okay these are the threads that are executed they have committed and these are the threads that have to start and all that. So, this multi threaded instruction unit that what we showed in the last picture.

This is the thing that is going to do this is also known as a hardware scheduler we will talk about that again later on. But the important thing is this management of threads are done in blocks of 32 32 threads together are packed as something as called a warps. So this is a basic atomic unit of parallel execution in a GPU. So when I said it's a warps that. I mean, that I have 32 parallel threads there together and making progress in the space.

So when a warps makes progress from some is by executing and add instruction. That means all those 32 parallel threads have executed the right instruction. Each SM. Now this is specific to that architecture that we've been talking about it manages a pool of 24 warps with a total of 768 threads, again this is specific to architecture each SM maps warps thread to the SP cores. So, as you can see that there are not many threads to manage.

So essentially, there are 768 threads, they are, they are being distributed across 24 warps $768 / 32$, Where 24 24 warps, but each SM will map this warps to the SP codes, how that is done is also the job of the SM. Now of cores, and I will just repeat these are the numbers that are specific to the GPU architecture we've been talking about with different GPU cards, there are these numbers keep on changing and they become much bigger with modern GPUs.

(Refer Slide Time: 35:32)

Warp execution

- ▶ In each operation cycle, the SM warp scheduler selects one of the 24 warps
- ▶ An issued warp executes over four processor cycles
- ▶ The SP cores and SFU units execute instructions independently

GPU Architectures and Programming Soumyajit Dey, Assistant

So execution of warps you need operation cycle, the SM warps scheduler will select one of these 24 warps. This is the synthetic example I have 24 warps. One of them will be picked up, and four in each cycle of activation of the SMs warp scheduler. It will select one of these 24 and map it to SPs so that the SPS can process that warp. An issued warp executes over four processor cycles.

The SP cores, and the SFU units, execute instructions independently of cores that SP cores have got the scalar, multiply and add units, eight of them, And the SFU units, which are again eight of them which contain and that there are two SFU units in each of them, and they are contained in total, eight floating point multiply and add units. So they are operating independently. So with this will stop here and we will start again with more details of how the SPs execute the warps in the next lecture. Thank you.