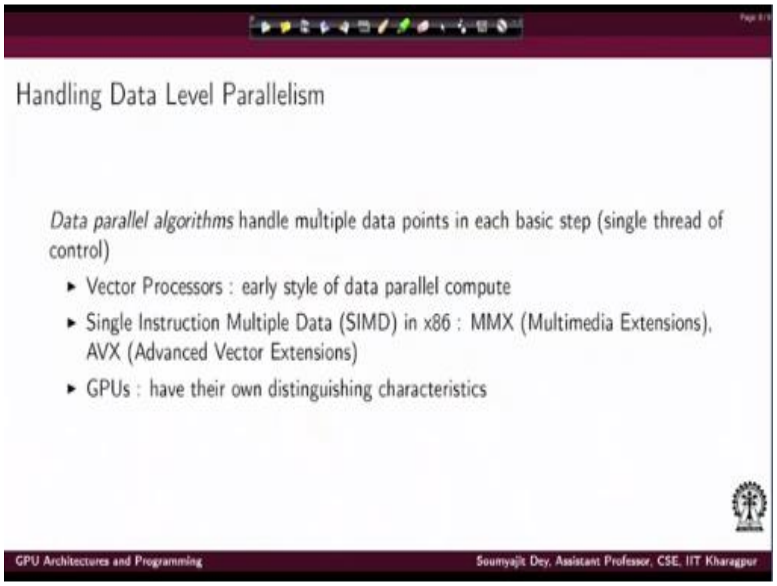
GPU Architectures and Programing
Prof R. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No. 5
Review of Basic COA w.r.t. Performance (contd.)

Hi, so after our basic introduction to come I mean, We have done a basic review of general computer architecture trends, more specifically, some background on risk architectures, with that background we are slowly going to delve deeper into how GPU supported, in terms of the architectural details.

(Refer Slide Time: 00:47)



Handling Data Level Parallelism

Data parallel algorithms handle multiple data points in each basic step (single thread of control)

- Vector Processors : early style of data parallel compute
- Single Instruction Multiple Data (SIMD) in x86 : MMX (Multimedia Extensions), AVX (Advanced Vector Extensions)
- GPUs : have their own distinguishing characteristics

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

So, we have always fixed on the fact that while we discuss about the risk pipeline. We have always say that how our risk processor will execute a single instruction, Using the standard phase decode, execute cycle. And at the end of that we also discussed how parallelism is handled in general purpose processors. And that also leads to some specific kind of processing that we know as vector processors.

And as we know that GPUs represent as more specialized kind of vector processor. Which have been found to be very useful for general purpose as well as graphics compute loads. So, the question is how is this data level parallelism handled? So, when we talk about data parallel algorithms. Essentially, these are algorithms that handle multiple data points in each basic step. So there's a new term we have been talking about this data parallel algorithms.

When we say there is parallelism in execution of a program. As we know that there can be parallelism resident at different levels. There can be instructional level parallelism, there can be thread level parallelism. But when we talk about data parallel algorithms, the specific thing that we mean is the algorithm has a sequential thread of processing, but in each step of the processing. The work will be done, on multiple data points simultaneously.

So it is like doing an add of multiple data points followed by doing a multiply of multiple data points followed by doing some other operation on again multiple data points like that. So, if this kind of instances of computation is found very frequently in some algorithm we like to call it a data parallel intensive or in general data parallel algorithm. So, typically, as we can see that there can be several examples of graphics workloads.

And several other kinds of processing, multimedia computation, or there are several such example workloads, which fit nicely into this notion of the data parallel algorithms and coming to the way they can be handled in hardware, So of course to handle such algorithms we require hardware processors that have the capability to exploit this parallelism right? Now, as we understand that parallelism in at the instruction level.

Can be extracted in two ways like we discussed earlier, it can be done at the compiler level, it can also be done at the hardware level. So, in a good case, it should be a combination of two that the compiler is able to extract some parallelism from a piece of code. That has been recreated sequentially, and also the hardware while executing instructions is able to mark out. Which of instructions that can execute in parallel.

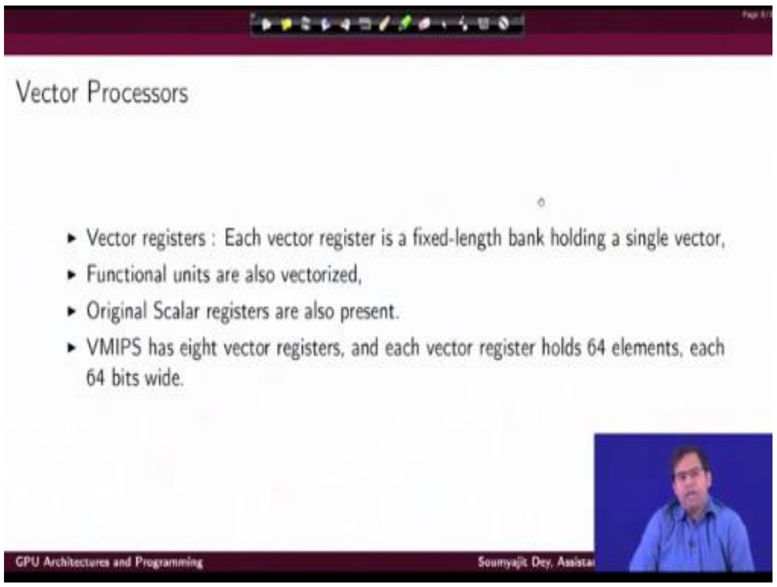
These are the two fundamental types we have already explored. Coming to handling of data, parallel algorithms. There are three popular techniques. I mean, the third one of course is the one that is general, I mean this graphic is processing units. The first one was vector processors. So, This was the earliest style of data parallel compute, like if you are trying to do operations on a parallel set of data points.

The same operation. Then, the solution that was proposed was vector processors will learn a bit about them. Then came the notion of specific instructions, which will be part of normal CISC processors like x86. For example, some MMX instructions, There are some multimedia extension or AVX instructions which came later on the advanced vector extensions. So these were kind of SIMD instructions.

Or single instruction multiple data type instructions that became a part of inter access architecture, and they are essentially instructions which perform the same operation on a vector data type, essentially, a set of data type, which are homogeneous. So, there's something that came next. The earliest style of data parallel compute was vector process. And as we know in recent times, that the idea of graphics processing units, has caught everybody's imagination.

And they represent the most modern technique of doing a lot of parallel computations in hardware together.

(Refer Slide Time: 05:24)



Vector Processors

- Vector registers : Each vector register is a fixed-length bank holding a single vector,
- Functional units are also vectorized,
- Original Scalar registers are also present.
- VMIPS has eight vector registers, and each vector register holds 64 elements, each 64 bits wide.

GPU Architectures and Programming

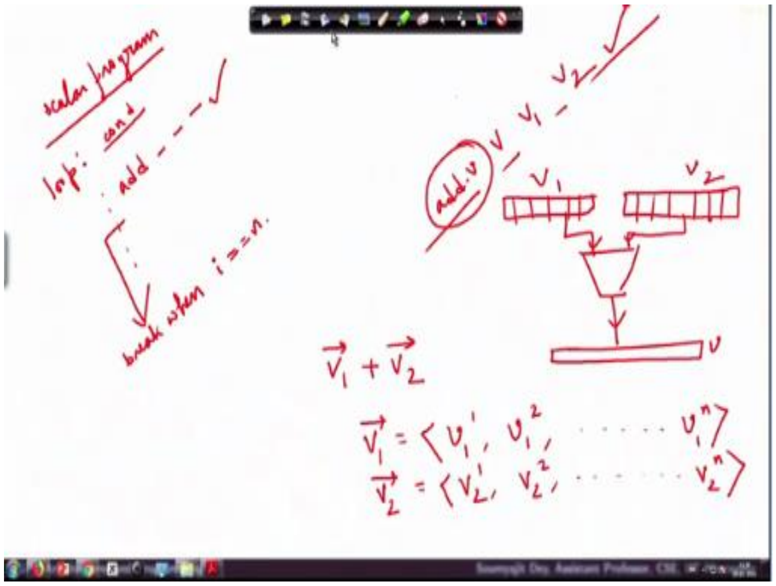
Soumyajit Dey, Assista

So coming to vector processor so what essentially is the design philosophy. Behind a vector processor is fundamentally like this that you have a register file processor. Apart from having normal by normal I mean scalar registers. You also start defining what we call a vector register. So what's the vector register, so each Vector Easter is a fixed-length bank holding a single vector.

So say you have length vector. Which is holding the different length values for the different dimensions of a house. So that can be stored as a single type is vector type of length, will now start saying that is not located in different registers the different components of this type are not located in different registers. But they are all stored in a single register which is having a big much bigger size, but it also has these compartmentalized parts.

Each of which has. is kind of holding out some specific component of the vector. So each vector register is a fixed-length band. It holds the single vector . So when I say that there is an operation I am doing an operation like vector addition. So there are two vectors v1 + v2. I am having the content of vector 1 in one register and the having the content of vector 2 in the second register and they would get at it. Just as an example.

(Refer Slide Time: 06:58)



scalar program
lop: cond
add
break when i == n
add.v
v1
v2
v
$\vec{V_1} + \vec{V_2}$
$\vec{V_1} = \langle v_1^1, v_1^2, \ldots v_1^n \rangle$
$\vec{V_2} = \langle v_2^1, v_2^2, \ldots v_2^n \rangle$

So if I just say that I have two data points v1 + v2. So V1 is a vector. That means, it contains components both are n dimensional vectors and I am doing their addition. So, in our normal program, or let us call it a scalar program. So what I would ideally do is I will have a loop structure. And inside this loop structure. I mean, there will be a condition here right? And inside this loop structure.

I will have some added instruction, which is doing addition of each of the components right? And I break away from this loop structure. When the iterated variable of the loop features n right? Now when I say, I have this kind of Vector instructions, I'd like to think that instead of this. I have some code, which is like this. So I have a specific instruction let sat add.v, which represents its adding content of registers, v1, v2 are storing it at some v.

So, in a way, what's happening is, there is some functional unit. And it is getting data from two vector registers, v1 and v2, and output get stored in another Vector register v. Now the question is this function it also needs to be a vectorized unit. That means it should also be able to work on this input data types in parallel. Or maybe I may have lesser number of units for each word multiple times faster enough to operate them.

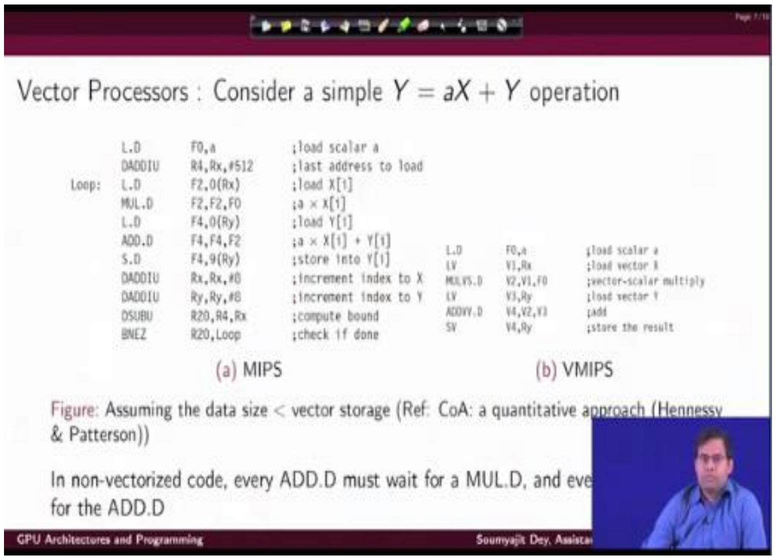
But the important thing is we will come to this pipelining idea, but the important thing is instead of having a single scalar instruction. I have a vector instruction. So basically, a hardware instruction which when executed. So there is a vector instruction which when executed. It operates on vectorize registers, and the vector is registers have got things dimensions, holding each of the components of the vector.

And in that way. The same operation gets done. Maybe at a higher speed up while we will come to that. So, the vector pricer will have a normal register file, as well as this kind of a vector register file. And functional units may also be vectorized essentially just like a explain that I can have an array of ALUs, which can do addition, subtraction, or other arithmetic operations in parallel. Of course, now original scalar registers are also present.

Now, what was the first proposition in this space. Well, one of the most popular initial academic propositions was the idea of vectorized MIPS, or VMIPS. So the VMIPS comprise the normal register file, along with 8 vector registers, each vector register holds 64 elements. Each of these elements were again 64 bits wide. That means. This VMIPS would have been able to operate on vectors of dimension 64.

And in each of the dimensions, the number of bits that were there for holding some value in that dimension will be 64.

(Refer Slide Time: 11:50)



Vector Processors : Consider a simple $Y = aX + Y$ operation

```
       L.D      F0,a          ;load scalar a
       DADDIU   R4,Rx,#512    ;last address to load
Loop:  L.D      F2,0(Rx)      ;load X[i]
       MUL.D    F2,F2,F0      ;a × X[i]
       L.D      F4,0(Ry)      ;load Y[i]
       ADD.D    F4,F4,F2      ;a × X[i] + Y[i]
       S.D      F4,9(Ry)      ;store into Y[i]
       DADDIU   Rx,Rx,#8      ;increment index to X
       DADDIU   Ry,Ry,#8      ;increment index to Y
       DSUBU    R20,R4,Rx     ;compute bound
       BNEZ     R20,Loop      ;check if done
```

(a) MIPS

```
L.D      F0,a         ;load scalar a
LV       V1,Rx        ;load vector X
MULVS.D  V2,V1,F0     ;vector-scalar multiply
LV       V3,Ry        ;load vector Y
ADDVV.D  V4,V2,V3     ;add
SV       V4,Ry        ;store the result
```

(b) VMIPS

Figure: Assuming the data size < vector storage (Ref: CoA: a quantitative approach (Hennessy & Patterson))

In non-vectorized code, every ADD.D must wait for a MUL.D, and eve
for the ADD.D

GPU Architectures and Programming   Soumyajit Dey, Assista

So let us just consider a small example here of Vector SMP program. On the left hand side, we have normal MIPS code. On the right hand side, we have a vectorized MIPS code example. This is a standard example which you can easily find in the book on computer organization architecture quantitative approach by Hennessy and Patterson. For the time being we assume that the vector sizes are less than the vector storage.

That means we are working on data, where the data type is vector. And that means it is like an array, or I mean basically arrays, but the dimension on which I am working on those arrays. They are smaller than then the vector storage possible in the architecture. That means, let us say, the vector registers have got 64 bit, 64 width that means 64 dimensions are allowed in and each dimension has got some number of fixed number of bits to store the value.

So essentially, I am talking about working on arrays where the dimension of the array is limited to 64. So here we are trying to do a simple scalar operation. So you want to multiply a vector capital X with scaler, a, and then do an addition, with another vector Y right? So, with respect to that. If you look at the MIPS code, the non vectorize scaler MIPS code. There are a few important things to notice here.

So in the non vectorized code, you see there is this addition operation, which is essentially implementing this Addition of a x i, along with y i. So, this addition operation is going to take one of the operands from the register F4 and stall- back the result to F4. So, it has got an input dependency, with the multiply operation. Why, because the other operand of this addition operation is in F2.

And, F2 is going to be updated by multiply operation, previous to doing the addition. So, this multiply operation essentially does this multiplication of a and i th component of x has stored back to F2. And in addition operation, this has to be added with the content of F4 which is y right? Now as we know that this is going to be done in a pipeline fashion. So for the multiply operation to n it has to write-back the value to the register file.

And that would happen in the last cycle of its execution. And until unless that happens, add cannot really do the operation right? So, ADD.D the needs to wait for multiplied or D. There will be a pipeline stall, as we have seen earlier. And also, if you look at the instruction is S.D. So .D of course means in MIPS mnemonics it means I mean we are working on double data types, just to remember.

And so this again has to wait for the execution of ADD to complete. Now why is that, before we stored instruction is S.D the store instruction is going to store the content of register F4 to some memory location right. But this content to get by would get updated by add previous to that. and then so then again for add to finish the write-back to F4 has to happen. So, S.D has to wait up to the point when the write-back where F4 is completed for the ADD.D the operation.

So, as we can see in the normal scalar implementation. I need to have the loop with the loop iterate variable i. I mean, in each iteration, I am doing, practically a x i + yi. And then I am implementing i. In each iteration, I have the ADD, which needs to wait for the multiply, and also the stored instruction is to wait for add. Now let us look at the end and this wait of wait has to be done in each iteration of the loop, there's the whole point here.

If you look into the vectorized code. As we can see, this add instruction has to wait is vector ADD instruction has to wait for them with their multiply instruction to complete. But wait how many times. Once, the vector store instruction needs to wait before the add instruction gets completed again. Wait, how many times. Once, because there is no loop here, assuming of course the dimension of the data size is less than the vector storage.

But then again there is a loop here, even if this assumption was not going to work. That means the data size was not less than vector storage. Still, then I would have a loop with much less number of iterations, the number of iterations would have decreased by a factor of vector dimension, so you can understand how many, cycles of waiting. You're going to save. If you have the vectorized code, and the vectorized hardware for executing the code.

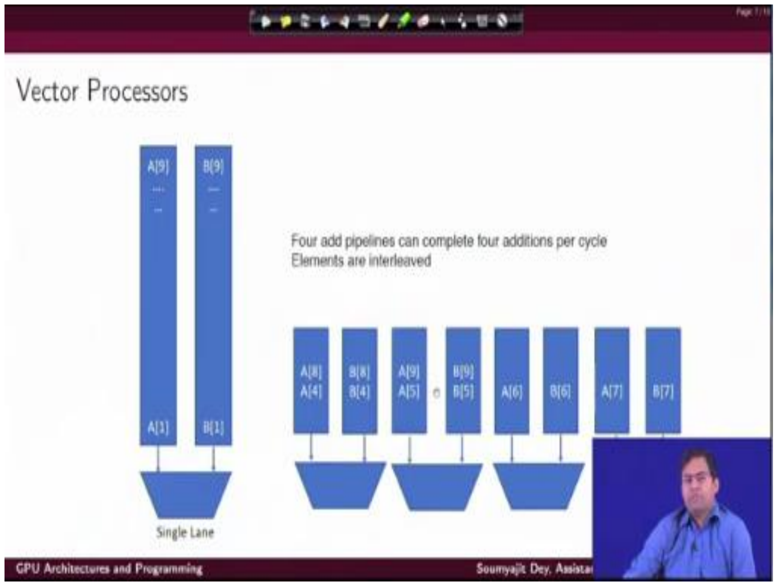
So, overall, the summary would be this the vector process, they will be executing scalar as well as vector instructions and vector instructions pass a lot of parallel work to the hardware. Of course, there has to be that much parallel hardware based compute capability, along with register banks which are vectorized. Now, what about the functional units, the hardware must have parallel functional units.

But it needn't be like this that okay my vector registers are of size of size 64. So I need to have 64 ALU, it need not be that that the functional units is can be either fully parallel or a combination of parallel and pipeline units. It depends on how the hardware is implemented. For example, if the clock rate of vector processor is halved doubling the number of lanes will retain the same potential performance.

These are standard optimization between pipelining and parallel processing. We will have an example for this. Now, the question is, if I have support in hardware for such better instructions as human that I have such support. Is there some overhead for the compiler oh yes there is, because then your compiler has to identify which are the segments of code that can be vectorized were to emit a vectorize multiplication instruction.

And where it cannot emit vectorized instruction. So this loop the vectorization and handling of dependencies is something that has to be done by compilers working towards vector processors.

(Refer Slide Time: 19:02)



Vector Processors

Four add pipelines can complete four additions per cycle
Elements are interleaved

Coming to vectorized hardware. So, how can you really implement vector processors, of course, as we understand that. the good thing is I have less number of instructions to execute. Now the instructions may take less amount of time. If I have a lot of parallel hardware. It may take, not so much little amount of time, if I cannot really execute all the vectorize instructions I mean all the, all operations in parallel.

There has to be some thread off. But this idea always holds that since I have much less a number of instructions to execute the number of weights in the pipeline would be less. So coming back to how Vector instructions can ideally be executed. So let us consider a thread off scenario here that suppose I am just trying to show that this a single lane of execution here. So I am assuming that I have this ALU nicely set up.

With the same port being connected to, to Vector registers, containing their of dimension, 10, each of them. And there is a single piece of ALU here. So I have been provided with an add.v kind of instruction. But then, since there is only one functional unit that adds happens seriously. in a serial manner. But the good thing is, there is not much of software assembly to execute only with that vectorize instruction.

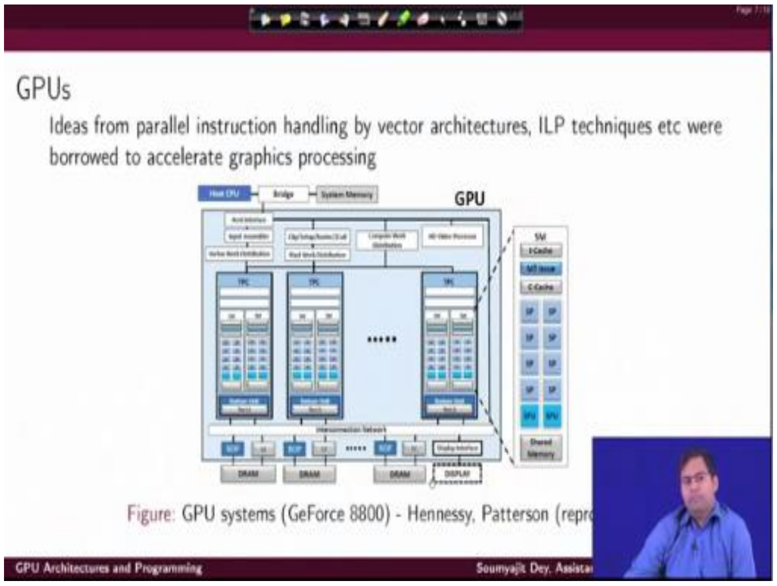
The entire pipeline understands that it has to operate A1+B1. And then it has to update the value somewhere. And then it will operate on A2+B2 update the value somewhere, it will do all the things, but in a serialized way, due to the absence of multiple vector processing lanes is a single lane execution. What is the other option. Now suppose I have Vector processing width just like earlier, I have 30 years to vectorized registers of size, like this, A1 to A9 is 9 size registers.

But I do not have that much width in terms in terms of vectorize lanes, I do not have that many parallel functional units. Let us say I have this kind of 4 functional units. Then what should ideally happen is the inputs will get distributed, like this. So, maybe they will be distributed in this kind of interleaving, in this kind of a interleaving, so the content of the Bs let say they can be distributed by A4 followed by A5,A6,A7 again A8, again A9 like that.

So across a different functional units. So I can say that, okay, although I have Vector width of 9, but the functional units are present here are only 4. So I have some speed up, but of course the entire addition will not be a single cycle operation, it will take multiple cycles. Of course, in this case it will take less number of cycles with it to switch to this, but it will not done well, it will not get done by a single cycle. again just to repeat the good thing here.

It is, I have a smaller assembly to execute the pipeline is being told that you have to execute a vectorized instruction. So the number of weights, are less, but how fast the instructions will execute depends on how many parallel hardware units are really present.

(Refer Slide Time: 22:35)



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 (repr

GPU Architectures and Programming   Soumyajit Dey, Assista

With this background on architectures. Let us just have a look on GPUs and their basic architecture. So this idea of parallel instruction handling by vector architectures, as well as the earlier, well known ideas like instruction level parallelism techniques. They were borrowed in the domain of graphics processing units. First, to accelerate specific graphics workload. And then, as we know that it has been found that

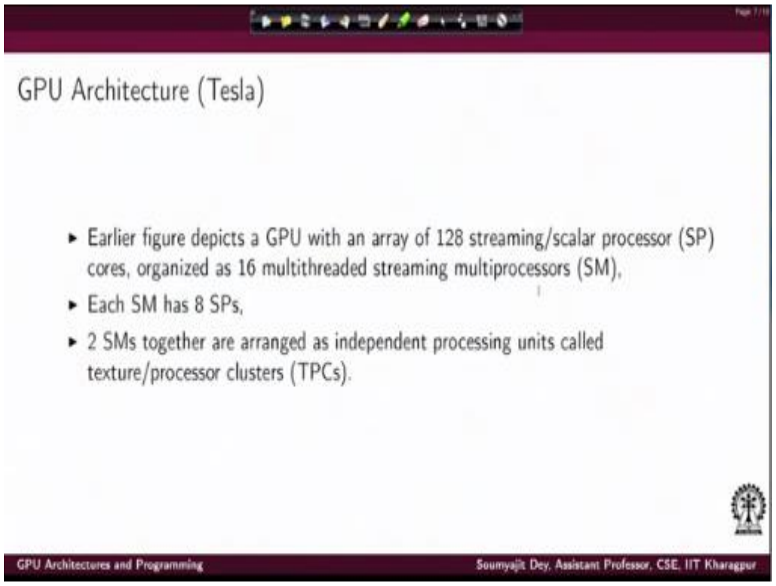
They can accelerate most general purpose data parallel workloads also. So this is a snapshot of our sample GPU system one of the earlier ones. This is a snapshot of the GeForce 8800 system process basic block diagram architecture is has been reproduced by me from the Hennessy Patterson's famous book. Again, which has also got a special chapter on GPUs. So, as you can see that the host CPU and the system's memory connect through a bridge.

And this is where the GPU will also be connected to. There is an interface with the bridge. And then there are certain functional units, which will be slowly progressing. What is important to note here is in typical in the GPU, you will have a hierarchy of processing elements at the high level. We have what we call as simultaneous multi processing units, the SMs right? And inside this SMs, you have specific

Processors, which are known as the scalar processors, the popular names for these are SM stands for streaming multi processors, and SP stands for. There are two well known terms that are generally used streaming processes. So they're not streaming multiprocessor streaming multiprocessor is SM, the higher level in the hierarchy, and at the lower level, each SM contains multiple instances of scalar processors.

Or in some books you will find the name streaming processors. So multiple instances of this streaming process. And of course there are other units like for the for the memory the cache the shared memory which will be covering one by one.

(Refer Slide Time: 24:48)



GPU Architecture (Tesla)

- Earlier figure depicts a GPU with an array of 128 streaming/scalar processor (SP) cores, organized as 16 multithreaded streaming multiprocessors (SM),
- Each SM has 8 SPs,
- 2 SMs together are arranged as independent processing units called texture/processor clusters (TPCs).

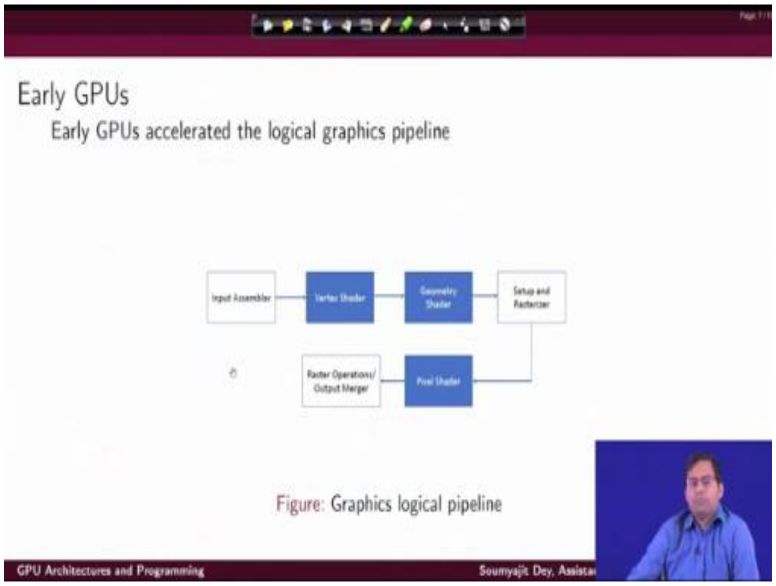
GPU Architectures and Programming   Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, coming to this idea of GPU. So, if we take an example GPU architecture as we know that there has been an evolution of GPU architecture in the last decade. And there are several well known families of GPU architecture. So we start with one of the earlier ones, was also very popular, the Tesla architecture. So, the earlier figure depicts a GPU, the GeForce GPU, with an array of 128 streaming processors, or scalar processors course, as we discussed earlier.

And they are organized as 16 multi-threaded streaming multiprocessors. So if we go back to this figure, as we said that there is a hierarchy here inside this hierarchy. I have 16 number of what we call us streaming multiprocessors SMs inside each of them. We have eight scalar processors or streaming processors, the SPs. Overall I have got 128 streaming or scalar processor. 2SMs together are arranged.

As an independent processing unit called texture /processors clusters texture processor clusters, or in some books is also known as thread processing clusters. So these are clusters. I have these clusters, eight of them. Each of them contain this to SPs, and inside. Sorry, I have got 16 of the SPS. 16 of the SMs, each of the SMs contain 8 SPs. So in total I have got 128 SPs the arrangement each, each SM contains eight SPs 2SMs together constitute one TPC.

(Refer Slide Time: 26:43)



Early GPUs

Early GPUs accelerated the logical graphics pipeline

Figure: Graphics logical pipeline

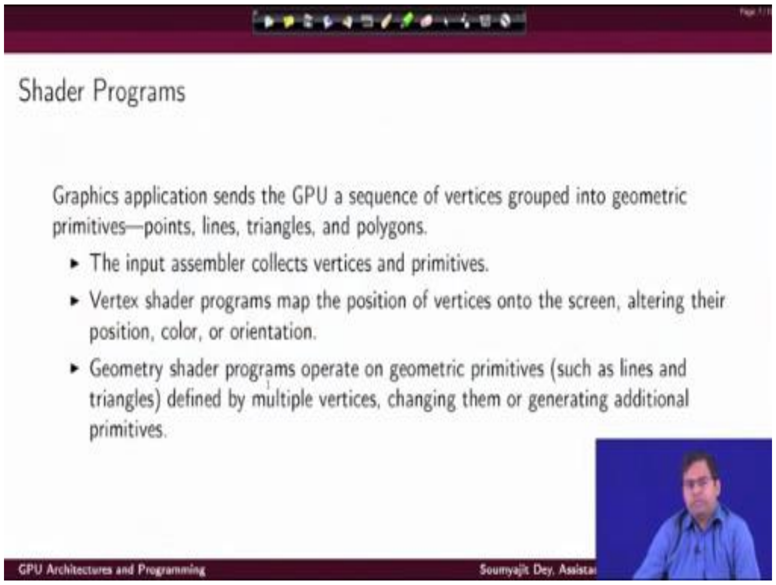
GPU Architectures and Programming

Soumyajit Dey, Assistan

So, what really was the function of earlier GPUs. So, earlier GPUs were developed with a specific idea in mind that they should be there to accelerate the logical graphics pipeline. So what's the logical graphics pipeline it represent the standard API, which is used for generating graphics for a computer. For that, there are a sequence of functionalities that need to be executed. And this is what the picture says, so we have the input assembler.

Followed by the vertex shader block, followed by the geometry shader block setup and rasterizer, and then the pixel shader and raster operations or, in some processor, also known as output merger. Now, these are the well defined significant components of the graphics pipeline, which together provide a standard pipeline through which computer graphics is usually render.

(Refer Slide Time: 27:48)



Shader Programs

Graphics application sends the GPU a sequence of vertices grouped into geometric primitives—points, lines, triangles, and polygons.

- The input assembler collects vertices and primitives.
- Vertex shader programs map the position of vertices onto the screen, altering their position, color, or orientation.
- Geometry shader programs operate on geometric primitives (such as lines and triangles) defined by multiple vertices, changing them or generating additional primitives.

GPU Architectures and Programming Soumyajit Dey, Assista

So, the idea is the software programs which really do that, they have a well known name that is called shader programs. Graphics application in the GPU, a sequence of what he says, which are grouped into specific geometry primitives points, lines, triangles and polygons. So, if you have an application and the application will try to render the output graphics on the screen. And for that it has to do the painting essentially.

It is to execute the computer graphics algorithm to create the bitmap image of that for that. It will require to execute this kind of specific graphics pipeline, which has got the six logically divided the parts for that. There are certain parts, which represent specific programmable parts, this is these are, these are the ones that we have highlighted here in blue. These are the vertex shader. The geometry shader, And the pixel shader.

So what are what is a Why is this sequence there since we have discussed that the graphics application will send this sequence of vertices. Now, these vertices needs to be processed by the GPU, the vertices may be grouped into specific different geometric primitives, they can either be in the form of specific discrete points. They can be in the form of mathematical objects thus lines or triangles, or polygons.

The input assembler is the block, which is going to collect these vertices and the corresponding primitives. The vertex shader is a software program, which will map the position of the vertices to the screen. Because finally is the screen where they need to be rendered? So this would mean a specific change in a specific change in the values, there will be a scaling and all that. And it will also mean altering their position color or their orientations.

After these vertex shader programs execution, comes the next colored block here, which is the geometry shader. Now the geometry shader program will operate on the geometry primitive. That means. I mean, it is going to operate on the specific graphics objects like lines and triangles, which are going to be defined again of course as we know by multiple parties as together, and they will change them or generate the original primitives for them.

So usually, what is these shader programs I mean, how are they written and all that because as we can see that they are the sequence of three shaders, vertex geometry and the pixel shaders, right? So essentially the shader programs that data's for style of computation that means multiple Parallel threads can work on different data points and do some computation in parallel. So we will let us do something like we stop here for this lecture and from this shader program and this Tyler computers and is from where we will pick up in the next part. Thank you