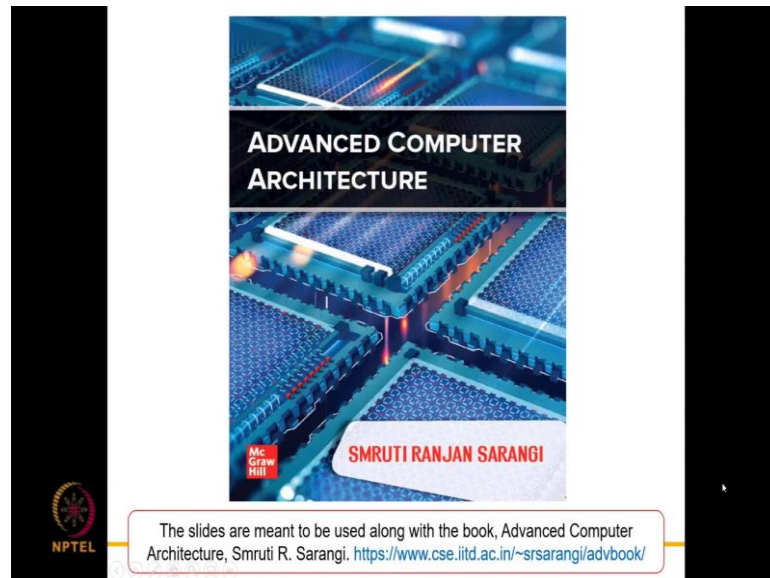


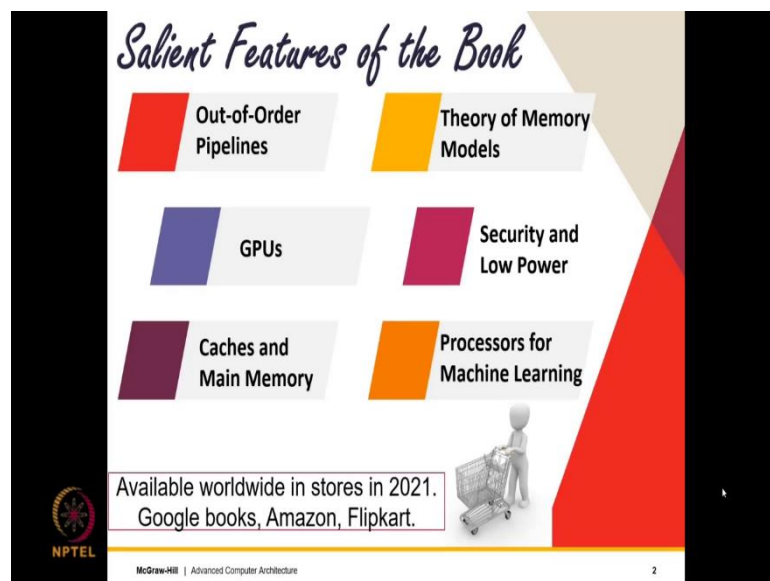
Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 16
Graphics Processors Part-I

(Refer Slide Time: 00:17)

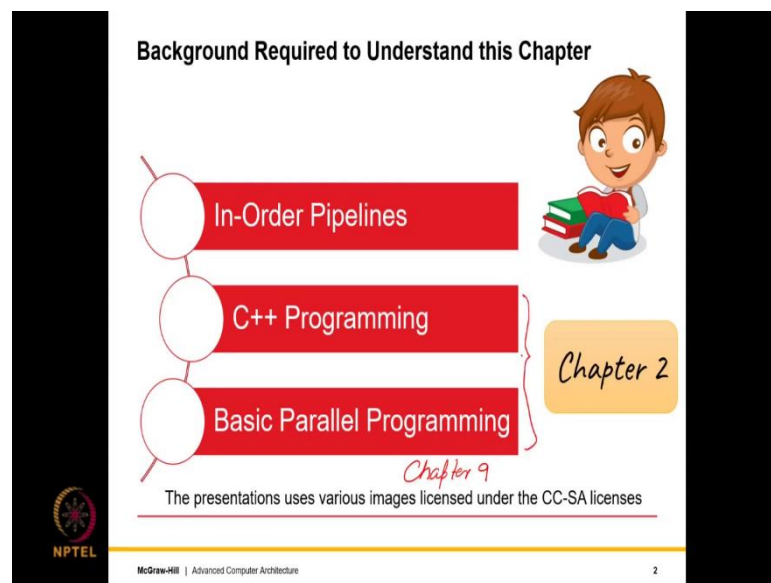


(Refer Slide Time: 00:23)



Welcome to chapter 6, the chapter on GPUs or Graphics Processors.

(Refer Slide Time: 00:37)



So, here is the background that we require for this chapter. So, we need three things specifically. We need in ordered pipelines. Now, basic knowledge of C ++ programming and some knowledge of parallel programming; so, for parallel programming you can also look at chapter 9 that will give you some idea of parallel programming or the early.

Or the first part of the two book series computer organization and architecture. So, that has something about parallel programming or you can consult the web or any other book. So, broadly speaking from this book we need chapter 2 and these two as a matter of pack up, nothing else is required.

(Refer Slide Time: 01:23)



So, there are three parts to these chapter 3 sections. 1st we will give a broad overview of application specific computing notably existing GPUs, ASICs and FPGAs then we will look at the CUDA 10 programming language. See even though as we speak CUDA 11 is the latest, but we will discuss CUDA 10 because that for us is complex enough and also the time of writing this book it was the latest version and finally, we will go to the design of GPGPUs.

(Refer Slide Time: 02:01)

Numerical and Scientific Programs

- Are we **satisfied** with an IPC of 4 or 6?
000 1.5-2.5 **No!**
- Many programs such as airplane wing simulation, weather prediction, image recognition have far more ILP and far higher performance requirements.
- Linear algebra **forms** a core of these methods.

Consider the algorithm to add two matrices:

```
for (i=0; i < N; i++) {  
    for (j=0; j < N; j++) {  
        C[i][j] = A[i][j] + B[i][j];  
    }  
}
```

independent

$O(N^2)$ additions can happen in parallel

NPTEL

McGraw-Hill | Advanced Computer Architecture

4

So, let us first start with a question. Are we satisfied with an IPC of 4 or 6? Even achieving this IPC is quite hard. So, if you look at a very aggressive out of order processor with all the tricks that we have done. So, with all the tricks with regards to advanced speculation, replay, pipeline optimizations all of that including memory system optimizations getting an IPC of 4 or 6 is very hard.

So, at the typical IPC for an out of order pipeline even if its 4 issue would be between 1.5 and 2.5 typically. So, even getting 4 or 6 is quite hard almost impractical, but it turns out that we are not satisfied with it. The reason being that if we consider very large programs such as for example, simulating an airplane wing, weather prediction, image recognition they have far more they have far serious ILP requirements and far higher performance requirements as well.

So, the reason I say that is because linear algebra particularly matrix algebra forms a core of these methods and consequently if let us say I were to add two matrices then the instruction level parallelism is huge and in that an IPC of 4 or 6 is actually too low. So, you consider this piece of code. So, in this piece of code we are just adding two matrices element by element. See if we look at this then we have two loops iterating from 0 to N loop iterators being i and j.

So, if we consider this then what we can see is that for different pixels these are completely independent computations totally independent. So, for different pixels these computations are totally independent. So, for large values of N we have truck loads of parallelism. We have a huge amount of parallelism right.

If N is a large value, the reason is we consider N square points and all N square points can be processed in parallel. All of these order N square additions can happen in parallel and that is exactly why we are not satisfied with the major IPC that we get. We need a different computing platform.

(Refer Slide Time: 04:25)

High-ILP Programs

- Many linear algebra-based programs have a **high** ILP
- Existing multicore processors are **constrained** in terms of their **fetch and issue width** (4-6)
- **Solution:** create a new processor that has 100s of ALUs for computing such instructions in parallel
- Method:
 - Have a very **simple** ISA
 - **Limit** irregular memory accesses, conditional branches, and

Handwritten notes: "hundreds of" with a box containing four small squares, and "small computing units" with an arrow pointing to the box.

Diagram: A lightbulb icon next to a box containing the text "Create a new processing framework for such parallel programs".

NPTEL logo and McGraw-Hill | Advanced Computer Architecture | 5

So, what is the; what is the main big deal with these high ILP programs? Well, many of these linear algebra based programs that deal with matrices regardless of the matrix operation they are very high degree of ILP extremely high degree of ILP. Existing multi core processors such as the ones that we have designed possibly maybe with one or multiple out of order codes they are very general purpose.

In the sense they are designed to execute a very large array of programs. So, they are kind of constrained by the fetch and issue width, fetch is not much. It is limited to 4 to 6 and that also restrains the IPC to remain within this value. So, a solution here is to create a new processor that has 100s of ALUs. So, what we want to do is that we want to design a different kind of processor, where we will have very very small computing units, hundreds of small computing units.

So, we will have hundreds of small small computing units. And what this small computing unit will basically give us is that let us say for these simple pieces of codes where we have to do many additions or multiplications in parallel, we will be able to realize the massive amount of parallelism. So, for these small computing units we can either call them an ALU or a code they are often small one two three issue in ordered codes they have a very simple ISA.

Many of the advanced features of modern programming languages such as irregular memory accesses or conditional branches are not supported. So, what we need to do is we

need to create a processing framework for such kind of parallel programs that have a massive number of parallel yet simple operations. So, the key point is that the operations are simple.


(Refer Slide Time: 06:43)

ASICs, ASIPs, and FPGAs

- Create a **custom chip** with 100s of ALUs
- ASIC – Application-specific integrated circuit
 - Used in all kinds of **devices** such as video cameras, washing machines, etc.
 - They can be made more **programmable**
- ASIC → ASIP
 - Application-specific instruction set processor
 - Provides some **degree** of **limited flexibility**

image processing

ISA - support flexibility



NPTEL

McGraw-Hill | Advanced Computer Architecture

6

So, we will now discuss ASICs, ASIPs and FPGAs. So, what we want to do is we want to create a custom chip with 100s of ALUs. So, one of the options is to create an ASIC, an Application Specific Integrated Circuit. So, this is done only for a specific application, specific kind of application with the inputs outputs formats everything is fixed.

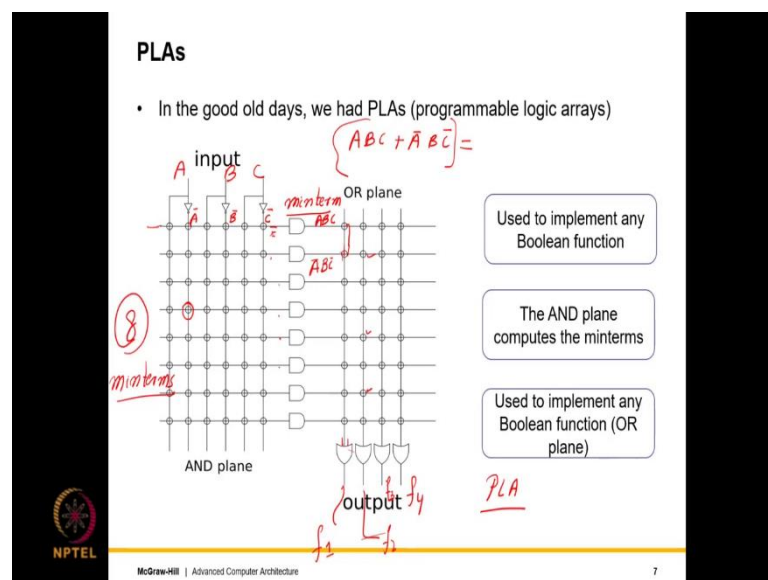
So, instead of using software we just create a chip for it, but needless to say there has to be a market for it because without a market, we cannot really design a chip. So, this will be used in all kinds of devices which include; well let us say that ASICs are used in all kinds of devices, but it does not mean that the same ASIC is used in all kinds of devices different ASICs are used.

For example, let us say for a video camera you would have to do image processing at a very fast rate. So, we can have an image processing ASIC with it. For washing machines, we will require a different kind of chip. So, we will fabricate a different ASIC. And given the fact that video cameras and washing machines have a huge market, what we can do is that well we have a large enough market to justify the creation of a specific chip for doing a specific task. So, these are in general not all that programmable.

So, as I said ASICs have a lot of value. In the sense that we need not have large processors that run programs, but they have a lot of value in specific scenarios. They can be made program more programmable if we move from ASICs ASIC to ASIP. An ASIP is an application specific instruction set processor which brings in a little bit more in terms of flexibility.

So, it provides a degree of flexibility in the sense there is some ISA support. So, you can think of it as a weak processor with very heavy ASIC like functionalities. There is a basic ISA support and there is definitely some flexibility including the flexibility to write small programs, but of course, the what you can do is quite limited in that.

(Refer Slide Time: 09:03)



Then we have PLAs. So, in the good old days we had programmable logic arrays or PLAs. See if you look at it this can be used to implement any Boolean function. So, for example, if let us say we have three inputs A , B and C . In the inputs at this point will become \bar{A} , \bar{B} and \bar{C} .

So, here the idea basically was that we choose a subset of this. So, clearly we cannot choose A and \bar{A} both together, but we choose a subset of this. For example, \bar{A} , B and C . So, what you are seeing in this wire? So, this is not a single wire, but it is actually a group of three wires that are going into this AND gate.

So, the first wire is carrying any one of A or \bar{A} ; the second one any one of B or \bar{B} and the third one any one of C and \bar{C} . So, essentially this the AND gate over here is computing a minterm. So, that is why this is known as the AND plane. So, just to remind you once again this is not a single wire, but a set of three wires.

So, here what we are doing is we are computing the minterm. And so, the advantage of computing the minterm over here is that we can compute different kinds of minterms. So, let us say we want to let us say compute ABC our function is of this type then the first minterm could be ABC . The second the one could be \bar{A}, B and \bar{C} then what we can do is we can take two wires like this and we can leave the rest. So, it does not matter.


So, we can choose in for this vertical wire. Again we do not have a single wire, but let us say we have multiple wires. So, we have 8 wires. So, out of this we can only choose these two and send them to an OR gate. So, then the output over here would basically be the output of this sum of these two minterms. So, similarly we have other OR gates. So, this is known as the OR plane.

So, we can then take another set of minterms and maybe like enable them enable this this and this and compute or of this. So, this will be another Boolean function. So, this is being used to compute Boolean functions. So, we are computing f_1, f_2, f_3 and f_4 . So, in this case as you can see we can generate 1, 2, 3, 4, 5, 6, 7, 8.

We can generate 8 minterms and then among the 8 minterms what we can do is that we can we are generating 4 functions. So, we can choose any of those 8 minterms, do an OR a logical OR of that and we can generate 4 Boolean functions similar to the way that they are generated.

So, as you can see that these selecting units over here are fully programmable that is the reason we call this a programmable logic array, where given a Boolean function and a set of variables we can implement it. In fact, we can compute several Boolean functions together. So, that is the reason even implementing something like an adder can actually be quite simple because one of them could be the Boolean function for the sum and other one for the carry. So, this as you can see is a quite flexible programmable logic array.

(Refer Slide Time: 12:47)



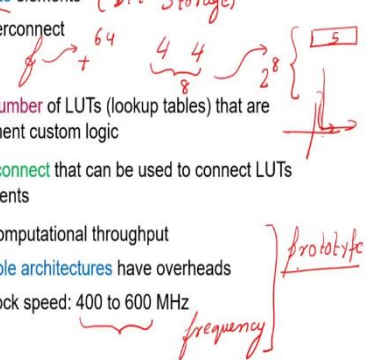
NPTEL

FPGAs

- A PLA is used only for combinational logic functions
- They cannot be used to implement a simple processor
- We need state elements (bit storage)
- A flexible interconnect

Use an FPGA

- Uses a large number of LUTs (lookup tables) that are used to implement custom logic
- A flexible interconnect that can be used to connect LUTs and state elements
- Provide high computational throughput
- Reconfigurable architectures have overheads
- Maximum clock speed: 400 to 600 MHz



64 4 4 8 5
frequency

McGraw-Hill | Advanced Computer Architecture

8

This of course, PLAs have been superseded by FPGAs because PLA is only for combinational logic functions. There are no state elements, but to implement something similar to a processor we need state elements. In the sense elements like flip flops and registers which can store values. So, we need some sort of a bit storage mechanism something similar to a state element and we need a flexible interconnect.

So, basically why a flexible interconnect? Because we will have to store different logic elements sorry connect different logic elements to different state elements and that is where we have the opportunity of creating a custom processing unit. So, this is why we use an FPGA which has these things. So, you can think of an FPGA as an extension of a PLA at a state elements and the flexibility of connection.

So, what it does is that along with programmable logic. So, incidentally now the amount of programmable logic in FPGAs has gone down quite substantially. So, given that the presence of PLAs in FPGAs has reduced quite substantially. We have moved to lookup tables or LUTs.

So, here the idea basically is there assuming that we want to add two 4 bit numbers, so, what is the total size of the input. It has 8 bits and the total size of the output maximum could be 5 bits. So, what I can have is that I can have a table with $(2)^8$ entries, which is like all combinations and for each combination I can store the output. So, let us say it is a

5 bit output for each combination. I just store the value of the output whatever it is. So, that will give me the sum as well as the carry.

So, I can basically take any function. So, we have a lot of customized software extremely sophisticated software these days that can take any function and basically map it into a set of simple lookup table accesses even if we are adding two 64 bit quantities. So, we are adding two large 64 bit numbers. We can still break it into a large number of smaller computations such that we access these lookup tables.

So, with lookup tables and flexible logic and flexible interconnects we can implement anything; addition, subtraction, multiplication, large processors everything. So, a flexible interconnect can be used to connect LUTs and state elements together. So, this will provide us with high computational throughput. So, note that reconfigurable architectures such as FPGAs. So, this method of using PLAs LUTs interconnects and so on.

So, even flexible interconnects are done with a PLA like logic only where let us say you know we want a packet to go this way. So, all that we need to do is we will have a set of multiplexers over here and we will then configure it to route all the signals to go that way. So, FPGAs also use something called an anti fuse. So, I will, so that can be seen on literature on FPGAs.

So, basically that is more like a persistent element where once programmed it maintains its state and it ensures that whatever signals from this will get routed that way. But, we will not discuss that in great detail, but what we will discuss is that FPGAs are extremely flexible and they have a very large number of logical limits.

So, you can implement massive parallelism. The only thing is that their clock speed is quite limited 400 to 600 megahertz and it is also not the case that one clock cycle achieves a lot of things. So, even for a simple operational that would take one cycle on an ASIC it may take multiple cycles on an FPGA. So, FPGAs are limited in terms of their frequency in terms of their area overheads all everything.

So, they are fairly you know they are extremely used they are fairly used frequently when creating hardware prototypes, but nowadays they are also used to implement production quality systems because both the efficiency of FPGAs and the software used to program FPGAs that has matured significantly. So, for many tasks like DNA sequencing and linear

algebra and so on FPGAs are not bad candidates, but they are quite expensive. So, one FPGA is almost as expensive as a state of the art server.

(Refer Slide Time: 17:21)

Traditional GPUs

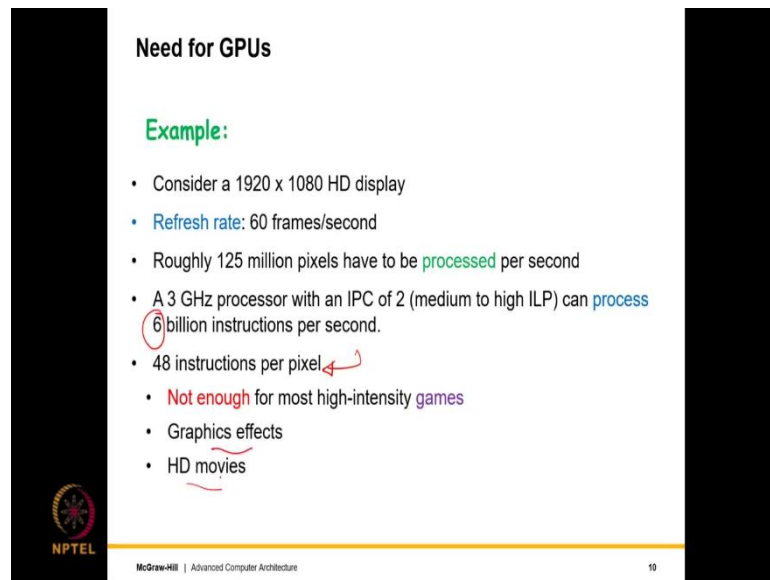
- 1 They were traditionally used to only handle graphics-intensive tasks. These are numerical programs with high ILP.
- 2 GPU \rightarrow GPGPU
Later they were repurposed to implement general numerical and scientific programs.

NPTEL McGraw-Hill | Advanced Computer Architecture 9

So, given the fact that ASICs are very good, but again ASICs are extremely specialized and furthermore if I have a small application I am not really going to make an ASIC for it, it is too expensive. Fabricating a chip when the market demand is not there or is not that much to justify the huge costs and given the overheads of FPGAs particularly for graphics intensive tasks which is almost everything we do these days our windowing systems games and so on GPUs were designed.

So, initially they were handling graphics intensive tasks like mostly handling the windowing systems or modern operating systems and also games, but later on they were repurposed to run numerical programs and scientific programs. So, this is how the GPU the graphics processor unit became a GPGPU, a general purpose GPU.

(Refer Slide Time: 18:25)



Need for GPUs

Example:

- Consider a 1920 x 1080 HD display
- Refresh rate: 60 frames/second
- Roughly 125 million pixels have to be processed per second
- A 3 GHz processor with an IPC of 2 (medium to high ILP) can process 6 billion instructions per second.
- 48 instructions per pixel
- Not enough for most high-intensity games
- Graphics effects
- HD movies

NPTEL

McGraw-Hill | Advanced Computer Architecture

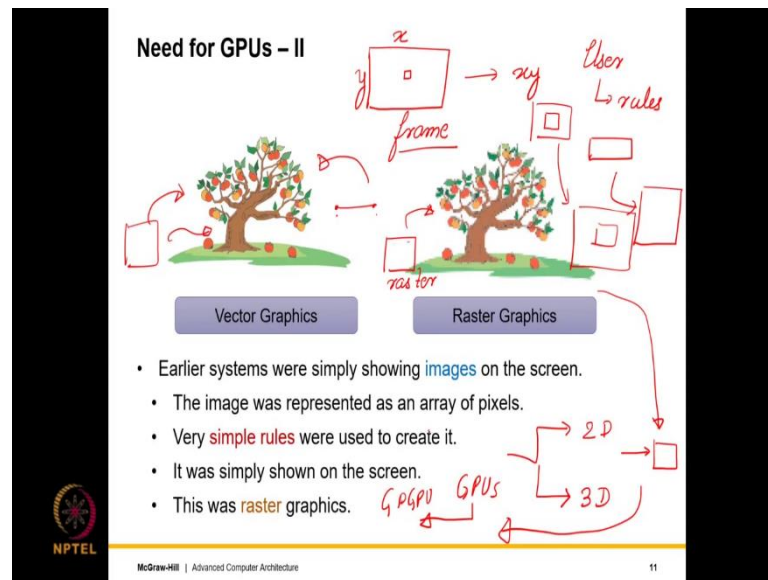
19

So, what is the need for a GPU? Let us do a little bit of math. So, consider the display on which I am recording this video. So, it has 1920 pixels on one axis and 1080 pixels on other axis. So, it is a high resolution, it is an HD display. The refresh rate which is the number of frames that I show per second is 60, 60 frames per second. So, this means that roughly 125 million pixels have to be processed per second.

If I consider a 3 gigahertz processor with an IPC of 2, which is medium to high it will be able to process 6 billion instructions per second roughly and so, which basically means 48 instructions per pixel ok. So, this is roughly what we will have to process. So, 48 instructions per pixel even though it sounds to be a lot, but actually it is not a lot if I look at this high intensity games that we have all of the effects that we have all the video that we watch.

So, nowadays what happens? Nowadays even if I am writing a book for making a PowerPoint and keep on listening to some music on YouTube or keep watching a YouTube video at the corner. So, then you know my processor has to do both. It has to run PowerPoint as well as process the sound of a YouTube video and the video of a YouTube video. So, typically this much is not enough. So, for more high definition and high intensity graphics we need better platforms for the graphics effects and HD movies.

(Refer Slide Time: 20:09)



So, this is where the need for GPUs comes up. So, GPUs are used for many things. So, one thing we need to understand before we actually look at GPUs about in modern graphic systems. So, in the modern graphic system as compared to what things were 30 years ago, where we thought of the screen just as an array of pixels, so, I am talking of things 30 years ago.

So, we thought that you know screen has x pixels and y pixels. So, we have xy pixels and this can be represented as an xy element array. Each element of the array is assigned a color right and all that we need to do is we need to compute these arrays and show them. And monitors of those days that worked at roughly 30 to 50 frames per second, so incidentally this array is called a frame.

So, you just have to recompute this array 30, 40, 50 times every second and most of the time the array remained constant you know there was no change. Only when there was a change this was updated, but modern graphics is quite different. So, in modern graphics we have to construct the image. So, what the user actually writes is a set of rules that is what the user writes. So, these are a set of rules.

So, let us say there is a character running in the middle of a city with guns and police cars after him with sirens glazing as you have in a game. So, what the user writes is the user defines the objects, defines the interactions between the objects, puts effects such as shadows and elimination and so on, but the final scene has to be constructed from the rules

and furthermore, if its small monitor like the monitor of a mobile phone then of course, rectangles will appear smaller, but you will still see the scene.

If let us say move it to a big monitor, we will still see the scene. So, this means that whatever the user does is not really specified in terms of coordinates, but rather in terms of rules. So, the user basically says that look in this frame this is the starting and end point of a rectangle and let us say that we move it to a bigger monitor the rectangle will get scaled appropriate.

So, when this method is used it is called vector graphics where for example, if you are storing a line we just store its first coordinate and the second coordinate. Now, if you move to another monitor with the line needs to be lengthened, it is very easy to lengthen it because we are not storing each pixel of the line, but rather we are storing the start and end. So, it can be lengthened.

So, you can have a library to lengthen it and render it. Render it means show it, show it correctly. So, we will discuss about rendering, but rendering basically means creating the graphics object on the screen. So, when such a rule based system is used as you can see over here where a rule based system has been used to create this tree full of fruits.

So, the image was much smaller, right, it maybe the image was this side, size, but when I expanded it, it became this size, but it still looks like a good image, it is not blurry. In comparison I took another image where only the pixels were stored as opposed to vector this is called a raster rasterized image. So, the rasterized image when I actually made it bigger you can see the blurriness in this image. So, this is called raster graphics.

So, the two paradigms that we have in GPUs particularly is that in the olden days we mostly had raster graphics because we only cared about arrays of pixels, but after that very soon we came to a point where we represent an object as a set of polygons. So, you will see mostly it is triangles, but let us say polygons and shapes of all sizes and any image is generated by the interaction of these shapes.

Here again there are two generations. The first generation was 2D that the tom and jerry kind of cartoons and the current generation is 3D where you can you have these three d characters in games and movies and so on. So, that is like 3D. So, movies of course, we

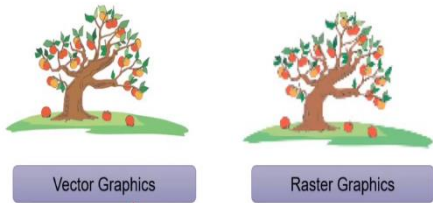
do not generate anything, but movies there is a separate problem that movies come encoded and they need to be decoded by the graphics card. Why encoding and decoding?

Because encoding means like heavily compressing a movie and decoding means uncompressing it with as little loss of quality as possible. That said and done most graphics tasks be it power point be it a game or be it just your windowing system specify the interaction in terms of objects and the GPU's job to pretty much convert that into a scene which you see on the monitor.

Raster graphics is still used in the sense many objects are still specified as arrays of pixels, but that is not all that common. So, given the fact that these GPUs were doing so much of geometry. People observed that this could be the GPU could be easily repurposed to create a GPGPU that also does general purpose computation without changing the design of a GPU very much.

(Refer Slide Time: 25:25)

Need for GPUs – II



- In modern systems, images are **created** from basic rules
- The programmer **creates** high level objects: shades, textures, characters, worlds, etc.
- She specifies the **rules of interaction**
- The system generates the images: **vector graphics** \Rightarrow GPGPU
- If we **zoom** the image, clarity is not lost

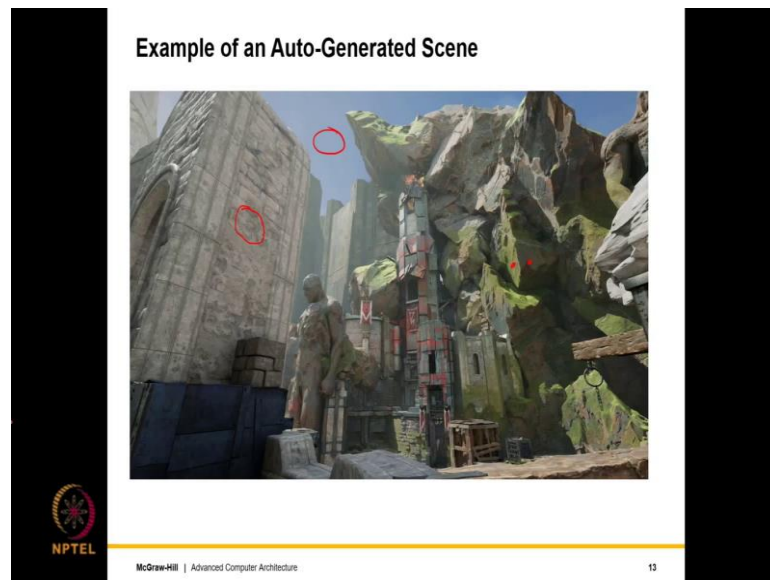
NPTEL

McGraw-Hill | Advanced Computer Architecture

12

So, if we further look at this, if we further look at this need for GPU slide you will see that in vector graphics as we have said I am just repeating some things where the programmer has created a high level set of objects with shades, textures, characters, worlds etcetera with the rules for interaction. And this vector graphics capability of the GPU was later on used or leveraged to create a GPGPU where a GPU continues to do what it was doing, but additionally it can also be repurposed to run absolutely general purpose numerical and scientific ports.

(Refer Slide Time: 26:07)



So, this for example, would be an example of an auto generated scene from a game where you can see a lot of effects. You can see this complex object over there. You can see the effect of light, shading, elimination. Just look at this point and look at this point. So, this point you can clearly see the effect of the face of the rock pointing towards the sun and you can see the shade over here.

So, all of this has been automatically generated after the main after the design was given to the GPU and you can also see the effect of texture. So, just look at the texture of this wall like you say feeling that this is the wall, but you this area gives you the feeling. It is the sky. So, the texture etcetera was all added.

So, lot of these things are added and you will see that if let us say I play the same game on the mobile phone. You will pretty much have a very similar visual experience because of the fact that the GPU of the mobile phone can interpret the rules differently as per its screen size.

(Refer Slide Time: 27:11)

Processing Requirements

- The **processing** requirements for games, high-intensity graphics, movies, etc., is huge
- **Aggressive** OOO processors (even multicore ones) are insufficient.
- We need **shader programs**.
- Custom language to **work** on objects, vertices, and pixels
- Apply **transformations** to images: rotation, skewing, etc.
- Apply **effects**: textures, shading, and illumination

Handwritten diagram in red ink:

ASICs
FPGAs
Trad. GPUs
GPU
1. Graphics
2. General Purpose

The diagram shows 'ASICs' and 'FPGAs' with arrows pointing to 'GPU'. 'Trad. GPUs' has an arrow pointing to 'GPU'. To the right, a list shows '1. Graphics' and '2. General Purpose'.

NPTEL
McGraw-Hill | Advanced Computer Architecture
14

So, this is where our GPUs have come to. So, now, the thinking is that well ASICs are very good, their but their usage is very limited and they are very expensive to at least create. So, there is a very high startup cost. So, unless the volume is large ASICs make no sense.

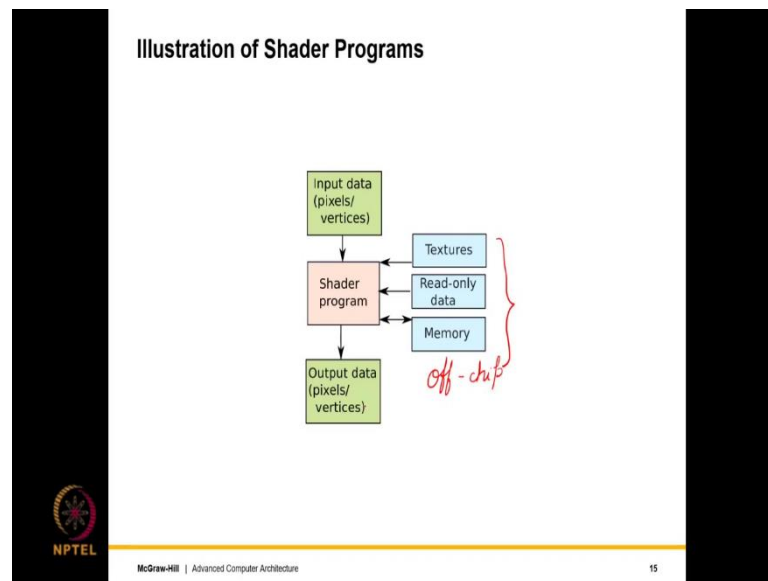
FPGAs again are great for prototyping and maybe a small set of applications. Traditional GPUs great for graphics, but if let us say I can kind of combine all of these three kind of get the power of traditional GPUs to do graphics power of ASICs to do general purpose computing. And power of FPGAs in the sense I have a C of computational units, I will arrive at the modern general purpose GPU. This is a broad idea.

So, this can look at games high intensity graphics movies of course, far more powerful in an out of order processing. So, we will first focus on the graphics aspect of a GPU and then we look at the general purpose that is fine. So, first we focus on the graphics aspect. Next we focus on the general purposes ok.

So, the graphics aspect what we need is we need a we need something called a shader program. This is a master program which does all the heavy work on objects, vertices, their pixels. It looks at transformation rotation skewing and so on. So, it applies all the transformations that are needed to images, it applies the effects.

So, as we have seen in the previous image it applies the effects of texture, shading, elimination. It does all of that automatically which also translate to linear algebra operations primarily, but we will discuss how having a general purpose language for these actually translates into something bigger something much bigger than the scope of GPUs themselves.

(Refer Slide Time: 29:23)

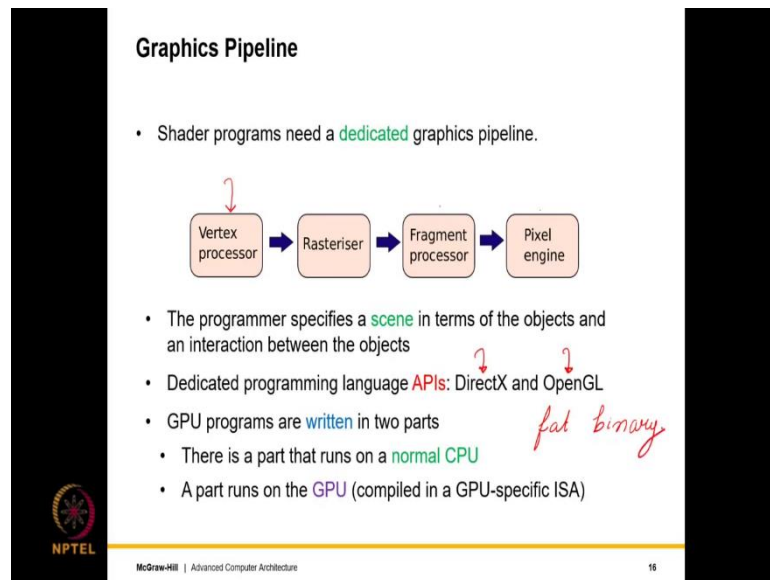


Let us now show the basic structure of a shader program. So, in a shader program we take an input data which can be in the form of pixels or vertices then we take in these three inputs. So, the first input actually comes from a dedicated memory structure known as a texture cache or a texture memory.

So, this provides all the textures for the image in the image region then we have a bunch of read only data think of them as constraints for the model. So, many GPUs particularly in the early days used to have; used to have a cache for read only data. So, maybe much of that does not exist today, but something of this nature used to be there. It used to be called the constant cache and then of course, GPUs regularly communicate with memory.

So, when we are talking of memory here it is mostly off chip memory. So, the GPU and the CPU do not share caches in general they speak they communicate via the off chip d ram memory and finally, the output is in terms of pixels and vertices. Some of it is stored in the caches inside the GPU and of course, if more data is generated which is not subsequently used then it is written to the off chip memory.

(Refer Slide Time: 31:01)

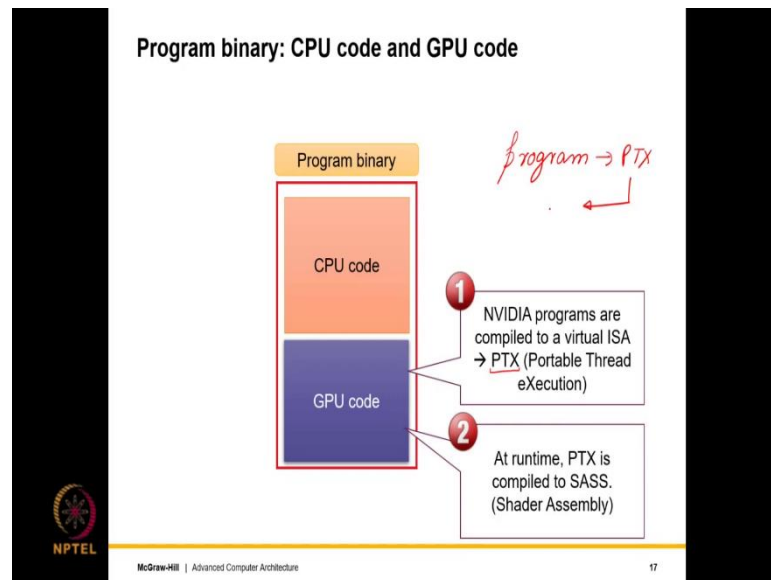


So, the graphics pipeline of a typical graphics processor this is the way that it looks, but there are some changes. So, one of the stages has changed. So, I will describe that later, but the quintessential graphics pipeline that shader programs use it is a dedicated graphics pipeline that has these four stages. The first is the vertex processor, then the rasteriser, the fragment processor and the pixel engine.

So, what the programmer does is that she specifies a scene in terms of the basic broad objects and the interaction between the objects, the light sources, elimination shading, the normal vectors of the surfaces and so on. The rest is automatically computed in hardware and in software. So, there are dedicated programming language APIs that allow you to write all of this code and a lot of it at runtime is converted into PTX code and what we shall see SASS code. So, we will discuss that later, but essentially in native code for the GPU.

So, there are two very common APIs Application Programming Interfaces that are used. Windows systems often use DirectX and OpenGL is quite platform independent. The GPU programs are typically written in two parts. There is a part that runs on a normal CPU and there is a part that runs on the GPU. So, this is compiled in the GPU specific ISA. So, basically you can think of it as a fat binary. A fat binary basically contains code of different instruction sets.

(Refer Slide Time: 32:53)



So, the program binary as we discussed is fat in the sense that it contains both CPU code as well as GPU code. Say, NVIDIA programs are first compiled to a virtual ISA. So, we will discuss in detail in later lectures what is exactly the need for a virtual ISA, but that is what NVIDIA programs are compiled to.

And the virtual ISA is called PTX or portable thread execution that is a virtual ISA. And at runtime the PTX virtual ISA is actually compiled to actual ISA. So, this is something like Java where let us say if you take the code of an of a program you first compile it to PTX and then you distribute that.

So, this ensures that nobody is seeing your code. Your code remains safe, your intellectual property remains safe and you assume a virtual machine and design an ISA for it and distribute it. But real time a lot of optimizations need to be done and these optimizations are nicely captured by the compiler for that specific GPU. So, that specific GPU at runtime it is dynamically compiled to SASS or shader assembly.

(Refer Slide Time: 34:19)

1 Vertex Processor *generic*

- Decompose a 3D surface into a set of triangles.
- Triangles have inherent advantages – all the vertices are on the same plane. Very *easy* to write code to process them.
- Perform *ray tracing* – trace the path of light rays and model their interaction with surfaces
- Add color and texture to a *surface*
- Move, rotate, and scale objects – basic *operations* on triangles
- Calculate the *depth* of the triangle (depth of the object in the image, measured from the eye) *Occlusion*

NPTEL
McGraw-Hill | Advanced Computer Architecture 18

So, the vertex processor this is the way that it looks which is the first stage. So, recall that there are four stages in the pipeline. So, we typically decompose a 3D surface into a set of triangles because it is very easy to work with triangles and triangles also have some inherent advantages which other representations do not have.

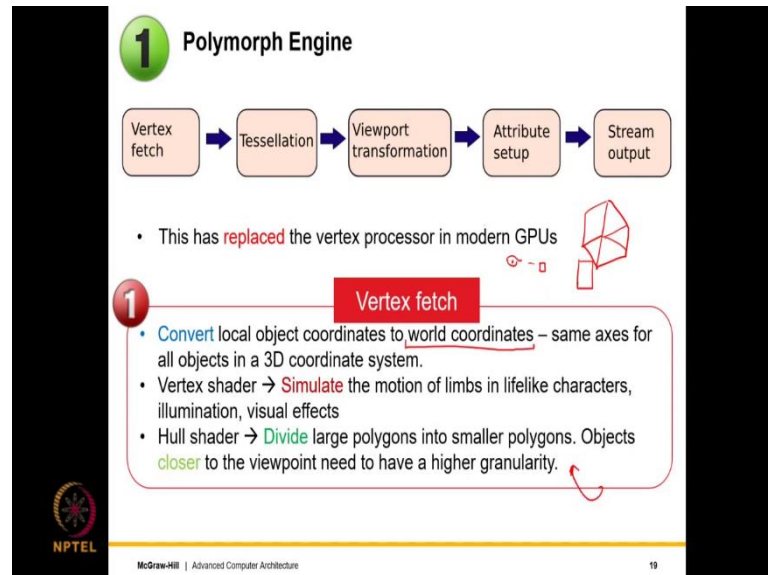
The most important being that, all the vertices are on the same plane. So, it is very easy to specify the plane of a triangle and its normal vector. We can then compute effects such as reflection very easily. And also there is a lot of literature and processing triangles and most triangle operations ultimately degenerate into matrix operations. We can perform ray tracing which is trace the path of light, light rays, see how they interact with surfaces. So, that will give you effects of elimination and reflection.

Finally, we can add color and texture to a surface. We can move rotate and scale the objects as is required and also calculate the depth of the triangle which is the depth of the object in the image as measured from the eye. So, with that what we can do is that things that are far away will appear to look slightly different. I also they will be occluded or hidden by nearby objects. So, this occlusion is important of which object hides the other object.

And so, much of this is computed here, but it is also computed later. So, the vertex processor you can think about as the stage that performs all the geometric operations on the shapes. So, this heavily uses linear algebra and previously this was using custom

hardware. Now, the vertex processors code is basically mapped to generic hardware and on generic hardware it just translates to tons and tons of linear algebra.

(Refer Slide Time: 36:25)



So, nowadays the vertex processor has been replaced by the polymorph engine. So, the polymorph engine itself is pipelined it has a five stage pipeline. So, if you think about it there is a pipeline within the pipeline stage. So, the polymorph engine has five stages; vertex fetch, tessellation, viewport transformation, attributes setup and stream output.

So, in modern GPUs they do not have the vertex processor again. So, I must underscore the fact that we are mostly talking about NVIDIA family of GPUs over here. We are not talking about other GPUs at the moment. So, in the vertex fetch stage what is what happens is we convert local object coordinates to world coordinates.

So, normally what happens is that in the programs in the shader programs that users write in OpenGL for example, they specify local coordinates, but all the objects are not in the same coordinate system. So, when all of them are in the same coordinate system this is referred to as world coordinates where you use the same axes same 3D axes x, y, z for all the objects.

And then the next part of this the vertex fetch stage is to also fetch the is fetch the vertices and then do geometric operations on top of them. So, this is where we simulate the motion of limbs, for example, in life like characters; move objects, you are shooting a bullet the

movement of the bullet and we also along with pre programmed motion we also look at other visual effects such as elimination, shading and so on.

So, we have a process here known as the hull shader. So, what this does is that let us say it takes a large polygon. So, a large polygon may be something like this. It divides this into a smaller set of polygons, could be triangles and there could be triangles could maybe triangles may not be triangles does not matter, but it is just that objects which are closer to the viewpoint which means closer to the eye they have a higher granularity.

In a sense the resolution is better and objects that are far away from the viewpoint. They have a lower granularity. This roughly corresponds to the way that we view objects that objects that are close to us close to our eye we tend to see them better. So, if let us say this is the eye and let us say an object that is close by we tend to see it much better as opposed to even a large object that is far away. We do not see that much of detail in it. So, this is also coming in over here.

(Refer Slide Time: 39:21)

The slide is divided into two main sections, each with a red header box and a list of bullet points. The first section, labeled '2' in a red circle, is titled 'Tessellation' and contains one bullet point: 'Break down the polygons into a set of smaller objects – triangles and line segments.' Below this text is a hand-drawn sketch of a rectangle with diagonal lines inside, representing a polygon being broken down. The second section, labeled '3' in a red circle, is titled 'Viewport Transformation' and contains three bullet points: 'The total scene is the window.', 'The portion that is shown on the screen is the viewport.', and 'This stage clips the scene and creates the viewport.' The slide also features the NPTEL logo in the bottom left corner and the text 'McGraw-Hill | Advanced Computer Architecture' and '29' in the bottom right corner.

2 **Tessellation**

- Break down the polygons into a set of smaller objects – triangles and line segments.

3 **Viewport Transformation**

- The total scene is the **window**.
- The portion that is shown on the screen is the **viewport**.
- This stage **clips** the scene and creates the viewport.

NPTEL

McGraw-Hill | Advanced Computer Architecture

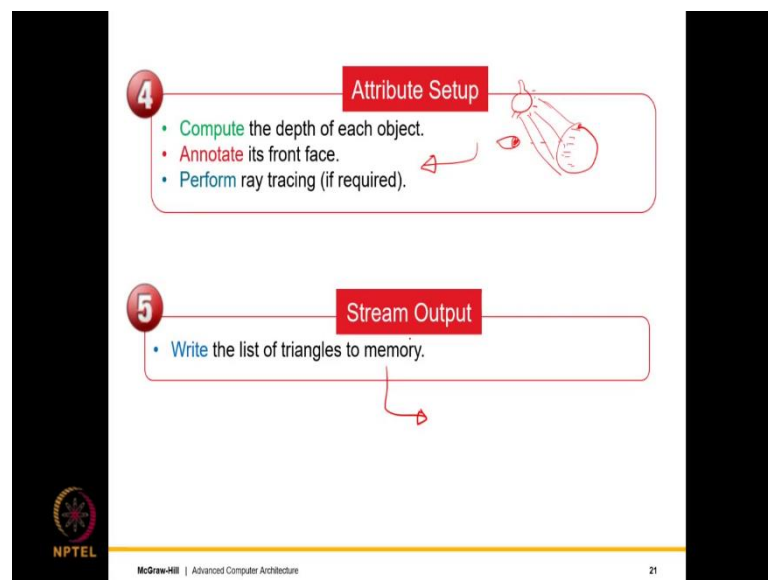
29

So, then we have tessellation. Tessellation basically means, once you have created the polygons once you have manipulated them to some extent we take them down into a set of smaller objects or which are triangles and line segments. So, we have already discussed the advantages of triangles. The advantages of triangles are twofold.

First is a triangle lies on a single plane and second there are lots and lots of algorithms there is lot of research in computer graphics to work on triangles. The third stage is known as viewport transformation. See the total scene in the window if I create a large scene. So, it is a large scene with lot of objects characters and so on all of it will not fit in my screen. So, what will fit in my screen is a subset of this it that I choose known as a viewport.

So, what this will this state will do is this will kind of get rid of all the objects or parts of the objects that are over here and only focus on the viewport. So, the first stage was vertex fetching where primarily fetch the vertex divided into polygons smaller polygons do some geometric operations. Then tessellate which is to divide it into triangles such that we can do more operations on it then we focus on the viewport, the part of the scene that is visible.

(Refer Slide Time: 40:43)



Then this is a very important step because this step has further ramifications down the pipeline. In the sense other stages which are later on in the pipeline they tend to read data from this state. So, here what we do is we compute the depth of each object and so, let us say if there is a complex object like a stone. So, we find out how far it is from the eye ok, the distance.

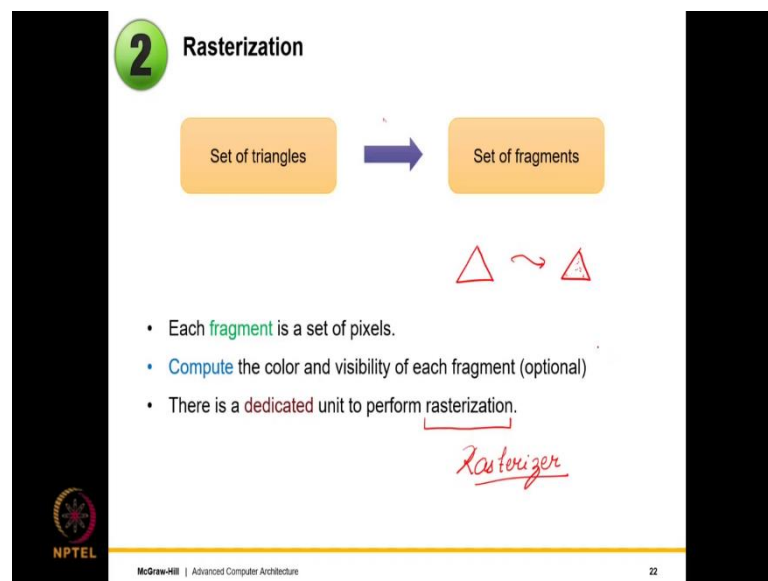
We annotate its front face. So, that so, if this is the front face. So, basically the part that is visible that part is annotated. So, this part is annotated and then if let us say there is a small light source over here then we look at different rays of light and see the way that they

interact with the surface and given the fact that different parts of the surface have different normal vectors.

There will be different degrees of reflection and scattering of light. So, all of that is computed in parallel of course. So, basically the list of triangles some degree of elimination information some degree of distance information; elimination and distance primarily distance from the eye right a hypothetical eye of course, and the elimination of the front face.

This information which is basically a huge list of triangles this is return to memory such that it can be passed to the next stage. So, in some cases it is return to the cache and it is kind of internally pipelined to the next stage. In some cases, there is a need to actually write it to main memory

(Refer Slide Time: 42:31)



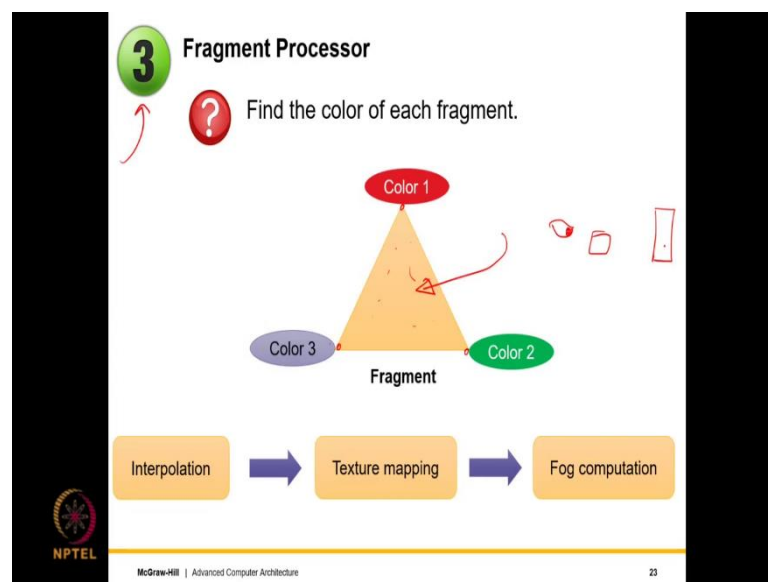
The next process; so, basically we finished the first stage of the big pipeline. So, in the first stage we looked at the vertex processor which has been replaced nowadays as I said with the polymorph engine. So, given a set of triangles, we convert the set of triangles into a set of fragments.

So, fragment basically the set of pixels and so, instead of looking at a triangular geometrical thing we do the same thing, but in this case since we are aware of the display we look at each pixel within the triangle. So, this triangle then becomes a fragment. We

here again optionally can compute the color and visibility of each fragment and given the fact that rasterization well of course, is required, but it does not involve a lot of geometric operations.

It is hard to derive any general purpose value out of a piece of hardware that does rasterization quite unlike the polymorph engine slash vertex processor. Consequently, there is a dedicated hardware unit even in modern GPUs that performs rasterization. This is known as the rasterizer hardware unit. So, this does the process of rasterization quite well in fact.

(Refer Slide Time: 44:05)



So, next we come to the 3rd stage which is known as a fragment processor. So, what does the fragment processor take from the previous stage which is the rasterizer is basically a set of triangles in kind of pixel format, but the triangles are not fully developed. So, there is a need to develop the triangles in the sense map it to the final scene.

So, to develop the triangles what we need to do is that we need to find the color of the fragment or color of the pixels within the fragment. The information that we have up till now is just the color of the three vertices of the triangle. We do not really have the information of the internals of the triangle.

So, in this case, the broad idea is like this that we find the colors of here and we somehow try to interpolate try to find out the colors from here just by applying a function on these

three colors and of course, the dimensions of the triangle. Then we add a texture over here right. So, we also superimpose a texture over here.

For example, if this is wood we will simple superimposed wood, if it is brick we will put in brick and finally, fog computation which is something similar to what I had said before that objects that are close by to the eye look slightly differently. So, if let us say this is the eye object that is close by looks differently in the sense the resolution is higher we see brighter colors.

Objects that are far away also appear to be slightly darker, but this also depends on the light sources. So, accommodating these effects is known as fog computation. So, we will discuss this. So, what are the three sub stages here in the fragment processor? Interpolation which is finding the color within the triangle, texture mapping and fog computation.

(Refer Slide Time: 46:03)

The slide is titled "Interpolation" and compares two shading methods: Goraud shading and Phong shading. It includes a list of characteristics for each, accompanied by hand-drawn diagrams. Goraud shading is described as assuming a flat surface, taking into account ambient light and surface reflectivity, and using linear interpolation of vertex colors. Phong shading is described as a smoothly varying surface (not necessarily a plane), using a complex model for reflectivity, and being more time-consuming. The diagrams show a triangle with light rays for Goraud shading and a curved surface for Phong shading.

Interpolation

- Goraud shading
 - Assumes that the triangle is a **flat** surface
 - Takes into **account** the ambient light and the reflectivity of the surface
 - Linear **interpolation**: compute color values based on the colors of the vertices
- Phong shading
 - **Smoothly varying** surface (need not be a plane)
 - Complex model for **reflectivity**
 - More time **consuming**

NPTEL

McGraw-Hill | Advanced Computer Architecture

24

So, for interpolation there are two shading stages. So, it is called Goraud shading and Phong shading. Goraud shading is simpler, Phong shading is more difficult. The idea over here is to find the colors of the pixels within the triangle. So, here we assume the triangle is the flat surface. So, this is the quintessential assumption that a triangle lies on a plane, it is a flat surface.

We take into account all the sources of ambient light. So, basically wherever there is light we take into account all the sources including in the reflections from other triangles. It take

all of the ray that are coming. All the rays are kind of let us say, hitting this triangle. The triangle has a normal vector oriented along some axis. So, if we compute the dot products we will find out the light that is hitting the triangle. The color of the light of course that is hitting the triangle.

We need to take into account the absorptivity and the reflectivity and the transmissivity of the surface, in a sense that this can be a translucent object as well, but let us say if translucency for the time being. So, if you look at the reflectivity of the surface then what we can do is we can take this into account and we can also look at the colors of the vertices over here all three.





Use a complex function which is of course, easily computable in hardware to find the colors of all the pixels within the triangle. So, this is known as Goraud shading. The other approach is known as Phong shading which is more complicated. When we assume that the triangle need not be a plane need not necessarily be a plane, it can be a smoothly varying surface.

So, it could be a surface like this. So, let me just draw a bigger version of it. It just could be a surface like this ok and this will have a very complex model of reflectivity which will of course, be far more time consuming, but this may be able to model a smooth surface quite well.

So, we are of course, making the assumption that a triangle lies on a single plane, but this may be able to model the surface of a kind of an complex sloping object much better. So, this was all about Goraud shading and Phong shading. Both fall in the class of interpolation algorithms for pixels within the triangle.

(Refer Slide Time: 48:49)

Textures



- Apply a texture to each fragment.
- Most GPUs have a dedicated cache to store texture information.

NPTEL


McGraw-Hill | Advanced Computer Architecture

25

Next coming to textures: as you can see there is a brick texture here, dirt, wood and so on. So, we are applying a texture to each fragment. Most GPUs will have a dedicated cache to store texture information.

(Refer Slide Time: 49:03)

Fog Computation



- Distance fog
- 3D rendering technique that colors pixels further away in the viewport differently
- Provides the perception of distance
- The information from the vertex processor can be reused here.

NPTEL

McGraw-Hill | Advanced Computer Architecture

26

And the last is fog computation. So, fog you think of it as distance fog. So, you look at the pixels that are close by. You see a better granularity they appear clearer, but look at the distances far away. They appear blurred. So, there is lesser of a granularity. Also see the

lines, even though the railway line is parallel. Do not you see at the end it appears to kind of converge? It does.

So, these effects that provide the perception of distance are known as distance fog. And since the vertex processor is already giving us this information about which object lies where in the sense that from the eye if I were to measure the distance then which object is how far, this information can be used here to render the scene. Render means, create the scene from the rules. So, I said the effect of distance is clearly seen. So, this is known as fog computation which is the third substage in this stage.

(Refer Slide Time: 50:05)

The slide is titled "4 Pixel Engine" with a green circle containing the number 4. It features two main sections: "Depth and color buffering" in an orange box and "Transparency effects" in a purple box. The "Depth and color buffering" section contains two bullet points: "Different fragments have different depths (Z-depth)" and "Based on the Z-depth, compute the visibility of the pixels". The "Transparency effects" section contains three bullet points: "32-bit color is represented as RGB and alpha", "RGB → Red, Green, Blue" (with a handwritten red arrow pointing to the first bullet and a bracket under "8 8 8"), and "Alpha → Degree of transparency" (with a bracket under "8"). A small diagram of two overlapping rectangles is in the top right. The NPTEL logo is in the bottom left, and the footer text "McGraw-Hill | Advanced Computer Architecture" and the page number "27" are at the bottom.

4 Pixel Engine

Depth and color buffering

- Different fragments have different depths (Z-depth)
- Based on the Z-depth, compute the visibility of the pixels

Transparency effects

- 32-bit color is represented as RGB and alpha
- RGB → Red, Green, Blue (8 8 8)
- Alpha → Degree of transparency (8)
- Display translucent objects correctly.

NPTEL

McGraw-Hill | Advanced Computer Architecture

27

Finally, we come to the last stage of the traditional GPU pipeline, it is known as the pixel engine. So, the pixel engine by and large does two things. It does depth and color buffering. So, different fragments have different depths. So, we call them as it is called the Z depth or the Z depth. So, let us call it the Z depth.

Based on the Z depth, we compute the visibility of the pixels because what could happen is we could have two objects one object could be hiding a part of the other object. So, this is known as occlusion. So, these effects of occlusions where one object is hiding a part of the other object, this needs to be computed. How will it be computed?

Well, the way that it will be computed is like this that we know the coordinates the N coordinates of all the objects. So, we know where they stand with respect to the eye, we

know the Z depths. So, we just compute the visibility of each of the fragments, each of the pixels if you want to go one level lower, but typically the fragment level is fine. So, then we can deliberately hide or occlude parts of the objects.

Furthermore, we can look at translucency. So, we had discussed translucency earlier also. So, given that we had promised to come back to translucency here it is. So, the pixel color value is represented using 32 bits normally. There is a RGB channel. RGB is red, green and blue. So, we have 8 bits each to indicate the intensity of the color for R, G and B and alpha is a degree of transparency.

So, one end is fully transparent and other end is fully opaque and anything in between is translucent. So, these values are also computed and the translucency effects are taken care of over here in the pixel engine where we display the translucent engines in this translucent objects correctly.

(Refer Slide Time: 52:25)

Other Uses of GPUs

- By early 2004
 - High-end processors had a **peak** throughput of 20 GFLOPS
 - GPUs were **exceeding** 50 GFLOPS. Reached 170 GFLOPS in a year.
- Repurpose the GPU's resources
 - Take linear algebra calculations → **Map** them to scenes
 - Let the GPU **process** the scenes
 - From the **computed** scene extract the results of the calculations.

Handwritten annotations: **GP** (circled), **Prog**, **APU**, **PLA**, **SASS** (all in boxes, connected by arrows).

Just design more **flexible** GPUs.
They can **perform** graphics-oriented tasks.
Can be used to **perform** general purpose tasks as well.

NPTEL
McGraw-Hill | Advanced Computer Architecture
28

And finally, once the entire scene has been created it is sent to the display device, whatever it may be the monitor or you will large screen display projector whatever it may be. So, the early 2004, high end processors were approaching a peak throughput of around 20 gigaFLOPS.

But even those days GPUs were exceeding 50 gigaFLOPS. It is just that GPUs were doing their own thing. They were doing graphics computations and there was no way to

repurpose or use a GPU to do general purpose computations. But, the processing power of a GPU was increasing by leaps and bounds. So, it became 16 170 gigaFLOPS in just within one year and then just kept on increasing right. So, it reached a teraFLOP very soon.

So, now evil processor designers and programmers laid their eyes on GPUs. So, they thought it is a good idea to repurpose the GPUs resources in the sense that a large part of the computation that you have just seen involves linear algebra other than the rasterizer to some extent. Most of it is generic linear algebra and if that is being done for a graphical scene it can be done it can be done for other high performance and scientific workloads as well.

So, what we can do is that we can have a more generic design of a GPU where let us say the programmer writes in a certain layer call this the programmer layer. Then there is a layer of software which provides a lot of APIs and libraries and resources to kind of convert it to another lower layer. And again a bunch of other layers, you know we have already discussed PTX and then it is a virtual instruction set and SASS shader assembly.

So, with this set of layers a simple graphical operation can sort of be mapped to a large number of linear algebra and numerical operations. And if a graphical program can be done we can take any other program and just map it to PTX. PTX should ideally not care and this can be even a general purpose program. So, this is how the idea of a general purpose GPU was born, where the GPU processes scenes, that is fine and then we can repurpose the hardware.

So, before repurposing the hardware an initial solution was to repurpose the software. In the sense what we do is that instead of directly making the hardware generic what people initially did is that they took linear algebra calculations for other problems map them to scenes allow the GPU to process the scenes and extracted the results from the computed scene whichever think about it is a really indirect way of doing things it is much better to repurpose the hardware.

That is exactly what they did. They designed more flexible GPUs instead of repurposing the software right. Make the hardware generic such that it can do graphics oriented tasks as well as non graphics oriented tasks quite well. So, now we have finished the first part which is discussing traditional GPUs, ASICs and FPGAs and what we have realized is that traditional GPUs have a lot of power.

It is just that initially it was not realized, but later on once it was realized a more generic substrate was created. So, the next two sections we will talk about the software aspect which is programming the GPGPUs and then the design aspect of how do we exactly design them. So, this lecture which is the first part will end over here and the second part will be there in the next lecture.