6 Graphics Processors

Up till now we have discussed general purpose processors, which are used to run all kinds of code. However, they have their limitations. They can at best achieve an IPC of 4 or 6. However, this is the best case and in practice to reach an IPC of 4 or 6, we need a near-perfect branch predictor, a very accurate prefetcher, and copious amounts of ILP. It is very hard to find all of these highly desirable traits in regular integer programs; however, it is often easier to find them in numerical programs that use a lot of floating point operations. We also have issues with power consumption and complexity. Having large instruction windows, rename tables, and elaborate wakeup-select logic requires a lot of power, and since power is one of the largest bottlenecks in modern processors, it is often very difficult to scale the issue width beyond 6 instructions per cycle.

This means that we are naturally limited to an IPC of 4 or 6. However, there are programs (albeit in a fewer set of categories) that can sustain a far higher IPC. For example, if we are adding two matrices, then all the additions can be done in parallel. This means that if each matrix has 10,000 elements, we can in principle do 10,000 additions in parallel. Since most numerical/scientific programs such as weather prediction, temperature simulation, and earthquake modeling involve a lot of numerical algorithms that basically consist of matrix operations, the need for designing a new processor for such codes is there. These matrix operations (a subset of the broader area of linear algebra) form the crux of most scientific applications. Furthermore, they have a lot of operations that can be performed in parallel. If we can design a good branch predictor and prefetcher, we will be able to supply instructions to the parallel pipelines at a fast rate. Given the fact that we have high ILP, we should be able to achieve a high IPC.

However, the problem is that all of this is not achievable in a typical processor. A typical processor has a lot of overheads in terms of the decode logic, renaming logic, scheduling logic, the load-store queue, and commit logic. To increase the issue width we need more area such that we can place more functional units, and we also need much wider schedule and wakeup-select units. We are severely constrained by chip area and power, and thus there is a need to think of alternative solutions. Using general purpose processors to do specialized computations such as scientific programs, is thus not a wise idea. General purpose processors are not equipped to deal with such kind of programs.

We need a special kind of processor that is particularly suited for such programs. Before going further, let us understand the nature of such programs by taking a look at a very simple sample computation, where we are adding two $(N \times N)$ matrices: A and B.

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

There are two loops: one for rows and one for columns. These loops are used to traverse large matrices. In other words, the parameter N can be very large (let's say > 1000), and thus the sophistication of the branch predictor does not matter because most branches are very predictable in this case. Secondly, in a lot of cases N is known in advance (during compile time). Thus, the programmer can manually optimize the program, and even break a large computation into several disjoint parts such that these individual parts can be run on different cores of a multiprocessor. It is true that the portability of code is an issue: a piece of code optimized for one computer will not run efficiently on another computer. However, given the fact that most scientific and numerical programs are not made to run on general purpose computers with ubiquitous usage in mind, this can be an affordable luxury.

We further observe that we have N^2 additions, where there are no dependences between these individual addition operations. They can be done in parallel, and thus we have a massive amount of ILP in this program. If we were to run this program on an aggressive out-of-order(OOO) processor, we will get a very high IPC (close to the fetch or issue width (minimum of the two)). However, to get even larger values of IPC, we need to create a processor that has many adders that can operate in parallel. If we have a processor with 100 adders, then we can perform 100 additions in parallel. If we have 1000 adders, then we can do 1000 additions in parallel.

In principle, this sounds to be a very enticing idea. If we have a different kind of processor that has hundreds of simple ALUs, then we can perform hundreds of arithmetic operations per second. This will give us the required throughput to execute large scientific applications very quickly. As we can see from the example of matrix addition, such codes are relatively simpler as compared to normal programs, which have a lot of complex constructs, conditionals, and pointer arithmetic.

Way Point 4

1. *There are a class of programs that have a lot of numerical computations. They are relatively simpler in terms of the number and nature of operations as compared to traditional general purpose programs.*
2. *These programs have high ILP. Most linear algebra operations fall in this category.*

Now, for such programs, we would definitely benefit if we have a processor with let's say 100 ALUs. This means that at least in theory we can execute 100 operations simultaneously, which is way better than an aggressive OOO processor that can at best provide a throughput of 4-6 operations per cycle. There is a flip side, which is that once we dedicate all our resources to these ALUs, and math processing engines, we are left with little area and energy to implement traditional functionalities of a processor such as fetch, decode, schedule, memory access, and commit. This implies that we need to simplify all of this logic, and create a processor that can execute a limited set of operations very efficiently. We essentially need to constrain the scope of the programs, and provide extremely high instruction throughput for the programs in this limited set.

Let us thus list out a few basic principles from what we have just learned.

1. We need to design a processor with possibly hundreds of ALUs. This processor should be able to execute hundreds of arithmetic operations per clock cycle.

2. We need not support a very expressive ISA. We can have a small custom ISA explicitly tailored for numerical programs.
3. Conditional instructions, long dependence chains, and irregular memory accesses make conventional programs complicated in terms of their structure and the hardware that is necessary to execute them. We can have limited support for such instructions and programmatic constructs.

By now, the broad contours of a processor explicitly tailored for numerical operations is more or less clear. We now need to find the best way to design one such processor. We at this stage might not know what to do, but we at least know what not to do – we cannot use a general purpose OOO processor for this purpose.

6.1 Traditional Technologies

Let us look at some of the conventional alternatives that have been there for at least the last three decades.

6.1.1 ASICs and ASIPs

We can use a custom made silicon chip called an ASIC (application specific integrated circuit). This is a silicon chip that is specifically designed for a single class of programs. Here, the algorithms are hard-coded in RTL and the degree of flexibility is very low. For example, we can have an ASIC to add two matrices, or multiply two matrices. In this case, we need to provide the matrices to the chip, and their dimensions. The chip will add (or multiply) these matrices, and store the result in memory. Even though such processors can be tailor-made for a problem, they have extremely limited applicability. A circuit designed to multiply matrices will not be able to find the inverse of a matrix. ASICs at best can be useful in very targeted applications. For example, we can have an ASIC in a video camera to compress the captured video. This will be an extremely fast and power efficient circuit. However, the ASIC cannot be used to do anything else. There is a clear trade-off between efficiency and flexibility.

In general, ASICs are very useful for specialized applications. The ASIC market as of 2020 is hundreds of billions of US dollars. However, it should be noted that ASICs are not full fledged processors because they are not programmable. Hence, they cannot be used for solving generic numerical problems. We can try to make the ASICs more programmable by introducing an instruction set, and by bringing in some components present in regular processors. Such interventions will make an ASIC an ASIP (application specific instruction set processor). An ASIP provides a very restricted ISA. In terms of flexibility it lies in between an ASIC and a general purpose processor. We can view the range of options from an ASIC to a regular processor as a spectrum; ASIPs fall somewhere in the middle.

Our aim is to operate at a sweet spot in this spectrum where we can optimally balance flexibility, programmability, power and performance.

6.1.2 FPGAs

No text on application specific computers is complete without a reference to FPGAs (field programmable gate arrays). An FPGA contains a large set of programmable logic circuits (or blocks), and a reconfigurable interconnection network. We can program the reconfigurable interconnect to make specific connections between logic blocks. It is possible to realize almost any digital circuit using an FPGA. FPGAs are logical successors of programmable logic arrays. Recall that a PLA (programmable logic array) consists of a large set of AND gates and OR gates, and it can be used to implement any type of Boolean function. An FPGA goes one step further and incorporates a lot of state elements and SRAM memories as well. We can program all of these blocks, and their interconnecting network at run time to realize an actual processor. This basically means that we can take an unprogrammed FPGA and convert it to a working processor on the fly (at run time).

It is true that we can dynamically create sophisticated processing elements on the fly using FPGAs. However, given the fact that they are being created from generic processing blocks using a generic interconnection network, they have performance limitations. As of 2020, FPGAs cannot be clocked at a frequency more than 600 MHz (general purpose processors can be clocked at 3-4 GHz). Furthermore, for most of their logic, they rely on lookup tables (LUTs). Lookup tables are large arrays that are used to compute the values of Boolean expressions (logic functions). A naive version of an LUT for a function with 16 variables contains 2^{16} (65,536) entries. Each row represents a Boolean combination of inputs, and each entry in a row contains the result of the function that is being evaluated. We can use such LUTs to compute the values of Boolean functions.

The advantage of an LUT is that we can implement any Boolean function using it. We simply need to change its contents. Of course, there are limitations in the scalability of this approach. If a function takes two values – 64 bits each – then we cannot afford to have a single LUT. Instead, we need to break the computation into several sub-computations, and we can have an LUT for each sub-computation.

Having a network of such LUTs, and a generic programmable interconnect is a very good idea for creating a high throughput system for a particular set of algorithms such as gene mapping or image reconstruction. However, FPGAs again have limited applicability, and also require fairly sophisticated programming tools. We desire a piece of hardware, which is reasonably generic, can be programmed with a fairly general purpose programming language, and delivers great performance.

This is precisely where GPUs (graphics processing units) come in to the picture. Let us understand traditional GPUs before moving to their modern avatars – general purpose GPUs (GPGPUs). Traditional GPUs were exclusively used for accelerating graphics computations; however, modern GPGPUs are used for general purpose numerical codes.

6.2 Traditional GPUs

In the good old days of MS-DOS and other operating systems that primarily operated via the command line, computer graphics had a very limited role to play. Most personal computer users as well as business users primarily used text-only terminals and were happy with predominantly textual interfaces. Even popular computer games like Tetris (see Figure 6.1) did not use a great amount of animation and graphics.

However, the entire situation changed with the introduction of operating systems such as Windows® that did not rely on the command line, and highly graphics intensive games. To sustain the requirements for such applications, there was a concomitant increase in the resolution of monitors. We started seeing monitors with a resolution of (1024x768 pixels and more) that used displays with vivid colors. Current monitors support 16 million colors. Even the nature and quality of animation increased, and suddenly car racing games felt like actually racing a real car in a FORMULA 1™ race.

Let us figure out the cost of doing all of this computation. A car racing game requires a lot of computation; consider something simpler – minimizing a window. In a modern system, we see the window artistically minimizing itself and becoming an icon on the taskbar at the bottom of the screen. The space that is left behind is used to show the contents of other windows or the Desktop. This much of computation definitely does take a toll on the processor. Consider a relatively old display with a resolution of 1024x768 pixels. This makes a total of 786,432 pixels.

If we consider a 3 GHz processor with an IPC equal to 2 (floating point + memory operations), then we are only allowed to execute 7633 instructions per pixel per second. If we consider a 60 Hz monitor that displays 60 frames per second, it means that we can only run 127 instructions per pixel per frame. This might be enough for minimizing a window; however, this is clearly not enough for all the games that we play. Some such games have multiple characters shooting bullets, cracking windows, and characters jumping across ledges and crevasses. We need to calculate a lot of things such as scenes, animations, shadows, illumination, and even the way that the hair of the characters will bounce when they jump. Sadly, the resources of a general purpose processor falls short for the requirements of modern games,

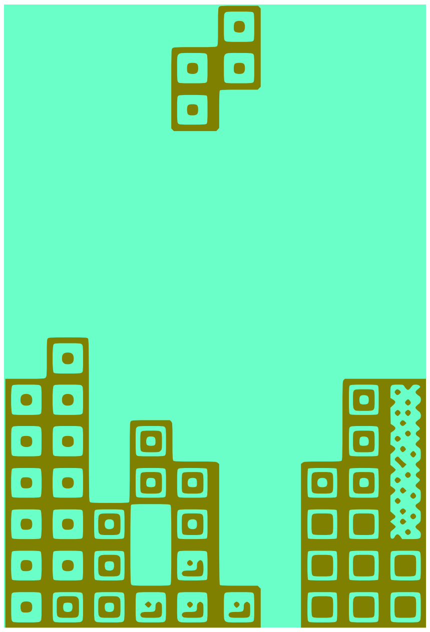

Figure 6.1: The DOS based Tetris game

which run on extremely high resolution displays (e.g. 3840×2160 UHD displays). Even if we can barely process the scenes in a game, we cannot do anything else like simultaneously browsing the web.

Highly aesthetic windowing systems, or games are not the only applications that require advanced graphics processing. Most of us regularly watch HD quality videos on our computers that require sophisticated video decoding engines, and we regularly visit highly interactive web pages. In addition, most engineers typically do their work using graphics intensive software such as AutoCAD®, which help us in designing complex 3D objects. Such diverse uses highly stress the processor, and most single or multicore processors simply do not support a large enough instruction throughput to run such applications. Furthermore, when we are playing a game, we would like to do other tasks in the background as well such as take backups, or run anti-virus programs. This requires additional computational throughput.

In response to such requirements, processor manufacturers increasingly started to ship their system with an additional *graphics processor* abbreviated as a GPU (graphics processing unit) along with regular processors. The job of the GPU was to process the graphics operations. This required support at multiple levels.

1. Programmers had to write their code such that a part of it ran on the CPU and a part of it ran on the GPU.

2. It was necessary to design new motherboards, where there was a fast and high bandwidth connection between the CPU and GPU.

3. Some new languages and compilers were created to write code for GPUs and optimally compile it for a given GPU.

It is important to note that till 2007 GPUs were predominantly used for graphics intensive tasks. There was no explicit thought of using GPUs for other generic tasks as well. This actually came gradually after 2010, when the community realized that GPUs could be re-purposed for general purpose computations as well. Thus, the idea of a GPGPU (general purpose GPU) was born. However, before delving into the details of a GPGPU, let us concentrate on traditional GPUs and understand them.

6.2.1 Early Days of GPUs

Graphics applications can trace their origin to the 70s when the main applications of computer graphics were in CAD (computer aided design) and flight simulation. Gradually, with an increase in computing power, many of these applications started migrating to workstations and desktops. By the late eighties, some new applications such as video editing and computer games had also arrived, and there was a fairly large market for them as well. This is when a gradual transition from CPUs to GPUs began. The modern GPU can trace its origin from early designs proposed by NVIDIA® and ATI® in the late nineties.

Before we proceed forward, it is important to define two terms here – *vector* and *raster*. These concepts can be explained as follows. There are two ways to define a rectangle. First, we can define it by the coordinates of its top-left corner, its height and width. In this case, each rectangle can be defined by 4 floating point values. This is a vector graphics system. We can optionally specify the color and width of the boundary of the rectangle, and maybe the color that is used to fill the inside of the rectangle. These will be a few more values. In comparison, in a raster system we store a rectangle as a matrix of pixels. Each value in the matrix represents the color of the pixel. This means that if a rectangle contains 10,000 pixels, we need to store 10,000 values.

Both the systems have their relative advantages and disadvantages. If we take an image (drawn using a vector graphics software) and enlarge it, then it will still retain its visual appeal. On the enlarged display, the system will still be able to draw the rectangle correctly. However, if we do the same with raster graphics, then it is possible that the image might actually look very grainy (see Figure 6.2 and 6.3). Also, for stretching and transforming an object, vector graphics is much better. In comparison, if we need to add effects such as illumination, shadows, or blurring, then raster graphics is more preferable. In general, we prefer raster graphics when we work with photographs.

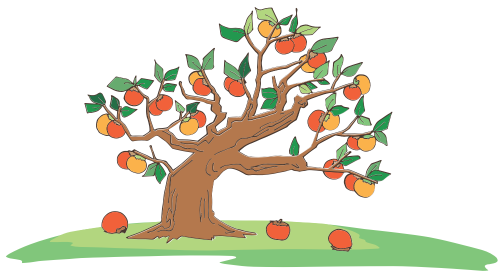
Figure 6.2: Vector graphics

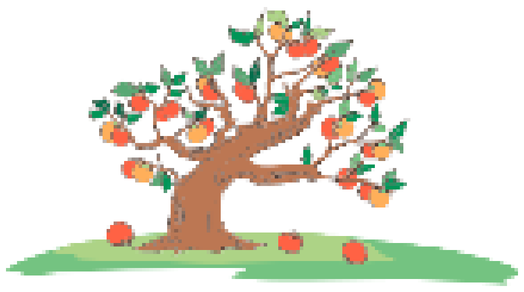
Figure 6.3: Raster graphics

Given the fact that there is no clear winner, it is advisable to actually support both the methods while creating a *graphics processing engine*. This engine needs to have multiple types of units for performing different kinds of tasks. Early systems were multi-pass systems [Blythe, 2008] – multiple computational passes were made on the same image. Each pass added a particular kind of transformation. This becomes particularly difficult, when images require a large amount of memory. Hence, a single-pass method is desired, where we can conceptually create a *graphics pipeline*. Images, scenes, or videos enter the pipeline at one end, undergo a process of transformation, and then exit from the other end. The end of the pipeline is typically connected to a display device such as a monitor.

By the beginning of this millennium the idea of creating a graphics pipeline with many specialized units, as well as many general purpose units that could be programmed, started to take shape. Also, by that time a lot of graphics applications had emerged, and the space of applications was just not limited to a few well defined tasks . *Fixed function pipelines* that consisted of a set of simple units that could just perform basic vertex and pixel transformation tasks were not powerful enough for these new classes of applications.

As a result, the best option was to provide users much more flexibility in terms of what they could do. Thus, the idea of a *shader* program was born. A *shader* is a small program that processes a fixed set of vertices or pixels. It is typically used to apply transformations to images, and add specialized effects such as rotation, shading, or illumination. The conceptual diagram of a shader is shown in Figure 6.4. Researchers further started working on custom languages for writing shaders. There was a need to make these languages independent of the underlying hardware such that they could run on different kinds of GPUs.

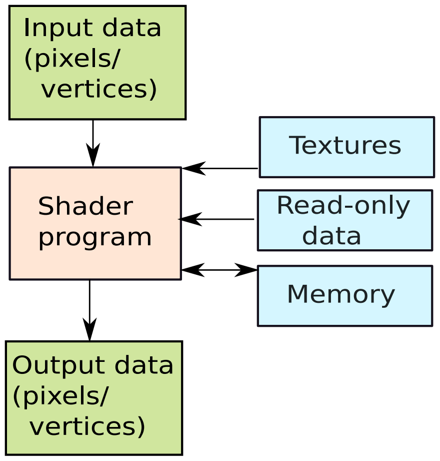

Figure 6.4: Conceptual diagram of a shader [Blythe, 2008]

Shaders have matured over the years. Starting from very rudimentary vertex and pixel processing operations, they have become significantly sophisticated. They are used for all kinds of applications: motion detection, adding texture, lighting, shadows, edge detection, and blurring. Much of today's research into the graphics aspect of GPUs is focused on designing and supporting more sophisticated shaders. Note that even though a shader is a program written in software, to run efficiently, it needs elaborate hardware support. This is where architectural techniques become important.

6.2.2 High Level View of a Graphics Pipeline

Before we discuss the high level view of a graphics pipeline, let us emphasize one thing. A graphics pipeline is meant to synthesize images, it is not meant to only display images. Most of the units in a graphics pipeline are dedicated to image synthesis. This process is also known as *rendering*.

Definition 32
Rendering *is a process of automatically creating an image (2D or 3D) from some rules and models. For example, a rendering engine can create a nice looking window on the screen based on some simple rules, models, and image files.*

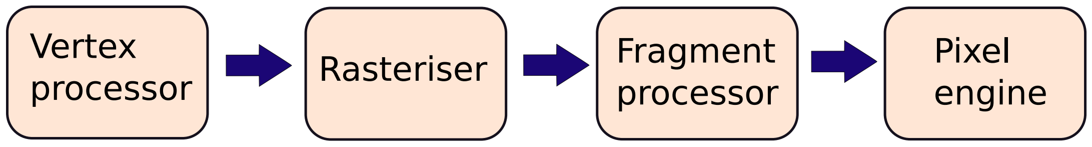

Figure 6.5: A basic rendering pipeline

A basic rendering pipeline is shown in Figure 6.5. We have four units. The programmer specifies a scene as a set of objects and a set of rules to manipulate those objects. These objects correspond to different distinct parts of a scene. These rules are written in high level graphics programming languages or APIs such as DirectX or OpenGL. Then, there are dedicated compilers that translate directives written in these high level languages to device specific commands. Graphics processors have their own assembly language and instruction formats. The NVIDIA family of graphics processors compile programs written in C/C++ (using the CUDA library) to a virtual instruction set called PTX (Portable Thread eXecution). PTX is a generic instruction set that is compatible with a broad line of NVIDIA processors. At runtime PTX is compiled to SASS (Shader ASSembler). SASS is a device specific ISA and is typically not compatible across different lines of processors, even from the same vendor.

Let us describe a generic pipeline that is broadly inspired from the NVIDIA® Tesla® [Lindholm et al., 2008] and NVIDIA Fermi [Wittenbrink et al., 2011] processors. For the exact description of the pipeline, please refer to the original sources. Our aim in the next few sections is to describe the main parts of the graphics processing pipeline. It is possible that individual processors might have slightly different implementations.

6.2.3 Vertex Processor

We start out by dispatching a set of instructions and a set of objects (to be manipulated) from the CPU to the GPU. The first box in Figure 6.5 is called a *Vertex Processor*. It accepts several objects as inputs along with rules for manipulating them. Even though it is more intuitive to represent objects as polygons; however, in the world of computer graphics it is often easier to represent them as a set of triangles. For each triangle, we simply need to specify the coordinates of its three vertices, and optionally the color of its interior. Note that more efficient storage mechanisms are also possible; however, this is beyond the scope of the book.

There are several advantages of using triangles.

- When we want to represent a 3D surface, decomposing it into triangles is always preferred. This is because the three vertices of a triangle will always be on the same plane. However, this is not true for a quadrilateral. Hence, even if the surface has many bumps, twists, bends, and holes, it can still be efficiently represented by a set of triangles.

- The technique of using triangles is the simplest method to represent a surface. Hence, we can design many fast algorithms in hardware to quickly process them.

- To generate realistic images it is often necessary to treat a light source as consisting of many rays of light. For each ray, we need to find its point of intersection with a surface. If a surface is decomposed into triangles, then this is very easy because there are very efficient methods for computing the point of intersection between a ray and a triangle.

- A lot of algorithms to add color and texture to a surface assume a very simple surface that is easy to work with. The surface of a triangle is very easy to work with in this regard.

- Moving, rotating, and scaling triangles can be represented as matrix operations. If we can quickly process such matrix operations, then we can quickly do many complex operations with sets of triangles.

Since complex rendering tasks can be achieved by manipulating the humble triangle, we can design the vertex processor by having a lot of small triangle processing engines. Each such engine can further support primitives such as translating, rotating, scaling, and re-shaping triangles. Such geometrical operations can be done in parallel for different objects and even for different triangles within an object.

Each triangle can additionally be augmented with more information such as the color associated with each vertex, and the depth of the triangle (how far is an object from the eye in a 3D image).

6.2.4 Polymorph Engine

In modern GPUs [Wittenbrink et al., 2011], the Vertex Processor has been replaced by a more sophisticated Polymorph Engine. This is because the demands of modern applications such as 3D scene rendering and virtual reality require complex 3D operations that are well beyond the capabilities of traditional Vertex Processors. Hence, there is a need to create a new pipeline with new functional units.

The key idea of this engine is to focus on what is called "geometry processing". We divide the surface of an object into a set of polygons (often triangles). This process is known as *tessellation*. Subsequently, we perform very complex operations on these polygons in parallel. Figure 6.6 shows the 5-stage pipeline of the Polymorph Engine (source: NVIDIA GF100 [Wittenbrink et al., 2011]). Let us proceed to describe the Polymorph engine that is a part of most modern GPUs today. In specific, let us look at the pipeline of the NVIDIA Fermi processor [Wittenbrink et al., 2011].

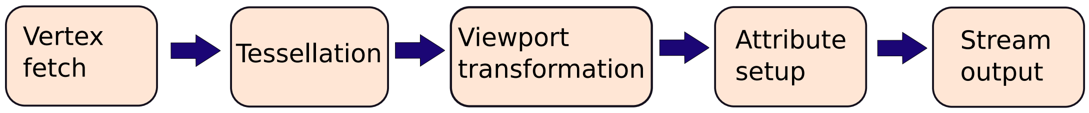

Figure 6.6: Stages in the Polymorph Engine

Vertex Fetch

The input to this stage is a set of objects with 3D coordinates. The coordinates are in the *object space*, where the coordinates are local to the object. At the end of this stage, all the vertices are in *world coordinates*; this means that all of them use the same reference axes and the same 3D coordinate system.

We start out with fetching the vertex data from memory. Subsequently, we perform two actions: vertex shading and hull shading. We shall see in Section 6.4.2 that GPUs consist of groups of cores known as streaming multiprocessors (SMs). The Polymorph Engine delegates a lot of its work to different SMs. Specifically, SMs perform two tasks in this stage: vertex shading and hull shading.

Vertex shaders are particularly useful in 3D scenes. They are used to add visual effects to a scene. For example, a vertex shader can be used to compute the effect of lighting a surface, or to simulate bones in a lifelike character. In the latter case, we need to compute the new position of each vertex in the bone as the arm that contains the bones moves. This can be done by the vertex position translation feature of a vertex shader. Thus, to summarize, the vertex shader works at the level of vertices, and can primarily change the coordinates, color, and the texture associated with each vertex.

The hull shader divides polygons into several smaller polygons. This is because we want different degrees of granularity at different points in the generated image. The objects that are closer to the viewpoint need a finer granularity as compared to objects that are far away.

Tessellation

The process of tessellation involves breaking down every polygon in the image into several smaller structures: triangles and line segments. The tessellation stage uses inputs from the hull shader. The main reason for doing tessellation is to create more detail on the surface and to also enable later stages of the pipeline to create an elaborate surface texture.

Viewport Transformation

In general, when we create a scene we are more interested in the objects and the rules that govern the interaction between them. For example, if we are rendering a scene in a game, we care about the position of the characters, and how they interact with their environment. However, we do not want to show the entire scene on the screen. It might be too large, and also all the parts of the scene may not be relevant. Let us refer to the scene that we have worked with up till now as the *window*.

Let us define a *viewport*, which is a portion of the coordinate space that we would like to show. There is thus a need to transform the 3D scene in the window to the scene that will be visible in the viewport. We first need to clip the scene (select a portion of it) and then perform scaling if the aspect ratio (width/height) of the viewport is different from that of the display device.

Attribute Setup

Now that we have created a scene in the viewport, we need to ensure that it renders correctly, particularly when we create the final 2D image. We do not want the backs of objects to be visible. Furthermore, if there is a light source, the illumination depends on the direction of the light rays, and the outward normal of the surface at each point. The dot product between the outward normal and the light rays determines the degree of illumination.

For each constituent triangle in the image, we compute the image of the plane (known as the plane equation) that it belongs to, and annotate each triangle with this information. This attribute will be useful later when we want to compute the visibility of different sides of an object and the degree of illumination.

Stream Output

The list of triangles is finally written to memory such that it can be used by subsequent stages. We typically do not have sufficient storage on the GPU to store all this information.

6.2.5 Rasterization

This process converts all the triangles to sets of pixels. Each such set of pixels is known as a *fragment*. This can be achieved by overlaying a uniform grid over each graphical object. Each cell of this grid consists of multiple pixels and can be considered as the fragment. In this stage, we can optionally compute a color for the fragment by considering its center point. We can interpolate its color by considering the colors of the vertices of the triangle, which this point is a part of.

Note that we do not discard all the information about triangles that comes from the Vertex Processor. Often all of this information is passed to the subsequent stage (Fragment Processor). Since the process of rasterization typically is not associated with a lot of flexibility, we can have a dedicated unit for rasterization. It need not be very programmable.

Furthermore, there is some degree of variance in the rasterization stage among different processors. In earlier processors such as NVIDIA Tesla this stage was relatively smaller. However, in NVIDIA Fermi and beyond, this unit does visibility calculations as well. This means that we compute which parts of objects are visible in a scene. There is no hard and fast rule on which action needs to be performed in which stage as long as all the actions that it is dependent upon are done. Readers should thus interpret the design presented in this section as broadly suggestive.

6.2.6 Fragment Processor

Subsequently, we need to compute the final color value of each pixel fragment. We need to take all kinds of visual effects and textures before computing this value. The job of the fragment processor is to perform all these computations.

The simplest method is to use the interpolated color value of the centroid of the entire fragment. However, this produces fairly grainy and uneven images. Instead, we can use a more elaborate process. This often requires solving complex equations and performing a lot of linear algebra operations. Let us look at some common operations in this stage.

Interpolation How do we compute the value of the color at each pixel? There are several interpolation techniques that allow us to do this. Some common techniques in this space are Goraud shading and Phong shading. Goraud shading is a simple linear interpolation based model where we can compute color values based on the colors of the vertices, the nature of the ambient light source, and a model of reflectivity of the surface. It assumes that a triangle is a flat surface, whereas Phong shading, which is a more involved technique does not make this assumption. It assumes a smoothly varying normal vector (perpendicular to the surface) across the surface of the triangle, and has a much more complex model for reflectivity.

Texture Mapping Consider a realistic image. It is very unlikely that its surface will be a single color, or even be a gradient. For example, the color of a tree's surface is not exactly brown, neither does the color uniformly vary between different shades of brown. A tree's color has a *texture*. Refer to Figure 6.7 for examples of different kinds of textures.

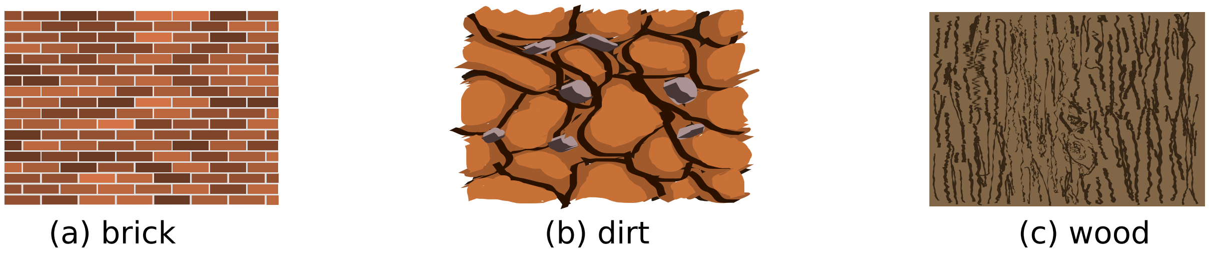

Figure 6.7: Different kinds of textures

Based on rules specified by the programmer, the GPU maps textures to triangles such that the surface of a tree looks realistic. We show the effect of adding a wooden texture to an object in the following figure.

In modern graphics processors it is possible to apply several textures and nicely blend them to produce a combined effect.

Fog Computation Distance Fog is a 3D rendering technique where pixels that have a greater depth (further away from the eye) are shaded (colored) differently to give a perception of distance. Recall that the vertex processor computes the distance information. This information can be used here to color objects farther away slightly differently.

6.2.7 Pixel Engine

The job of the Pixel Engine is to take the output of the Fragment Processor and populate the frame buffer. The frame buffer is a simple 2D matrix that holds only the color information for each pixel on the screen. The frame buffer is directly sent to the monitor for display.

To populate the frame buffer it is necessary to take all the fragments, and create a 2D image – image that is seen on the screen.

Depth and color buffering Fragments have different depths (known as the *Z-depth*). While rendering 3D images, one fragment might block the view of another fragment. It is possible to find out if this is happening by comparing their coordinates and Z-depths. Once this computation is done, we can find the fragments that should be visible and the fragments that should not be visible. We can then look at their colors (computed from the previous stage) and use them to create a 2D image out of the visible fragments.

Transparency effects Modern coloring systems are based on three colors: red, green, and blue (RGB). In addition, they take an additional value called *alpha* that specifies the degree of transparency. It varies from 0.0 (fully transparent) to 1.0 (fully opaque). If a translucent object (semi-transparent) is in front of an opaque object, then we should actually be able to see both. This part of the graphics pipeline ensures that this is indeed the case.

Once the frame buffer is populated, it is ready to be sent to the display device.

6.2.8 Other Uses of a GPU

Till now, we have discussed the different structures inside a traditional GPU. The crux of the discussion centered around the concept of breaking complex structures into triangles and then operating on sets of triangles. This was achieved by converting operations on triangles to different kinds of matrix operations. Note that many such linear algebra based operations form the backbone of a lot of scientific code starting from weather simulation to finding the drag experienced by a wing of an aircraft. The users of such simulation software were traditionally using expensive workstations and large supercomputers.

Starting from 2006, the scientific community gradually woke up to the fact that graphics processors are almost as powerful as large servers if we simply compare the number of floating point operations that they can execute per second (FLOPs). For example, in April 2004, high-end desktop processors had a peak throughput of roughly 20 GFLOPs (giga FLOPs), whereas some of NVIDIA's GPUs had a peak throughput of 50 GFLOPs [Geer, 2005]. By May 2005, the peak GPU throughput had increased to 170 GFLOPs with the CPU performance remaining more or less the same. This is where programmers sensed an opportunity. They started thinking on how to commandeer the resources of a GPU to perform their numerical calculations.

Some early pioneers started mapping their calculations to scenes and textures. Then they used vertex and fragment processing operations to perform operations on their matrices as described by Rumpf et al. [Rumpf and Strzodka, 2006]. This is like using the engine of a car to run a boat.

Gradually, GPU designers understood that it is better to make the GPU far more flexible and programmable. This is the most beneficial: it opens up new markets because now a GPU can be used for many other purposes other than rendering scenes. Vendors of GPUs also realized that many of the operations in vertex and fragment processing are in reality massively parallel computations mostly based on linear algebra. To increase the degree of programmability, it is wiser to make these units more generic in character. Furthermore, since engineers working on high performance computing also require such capabilities, it is advisable to create features such that they can write and run their algorithms on a GPU. Anticipating such trends, NVIDIA released the CUDA (Compute Unified Device Architecture) framework in February 2007, which was the first widely available software development kit (SDK) that could be used to program a GPU.

Thus, the modern GPGPU (general purpose GPU) was born. It is a very programmable and flexible processor that can be used to perform almost all kinds of high performance numerical computations. GPUs today are no more limited to just processing and creating computer graphics, instead, graphics is just one of the applications that is supported by a GPU.

6.3 Programming GPGPUs

As we saw in Section 6.2 graphics processors have elaborate structures to perform graphics intensive tasks. However, they can also be effectively used for regular scientific code such as weather simulation, or computing the distribution of temperature in an object using finite element analysis techniques. Such computations more or less share the same characteristics. They involve complex linear algebra based operations and a lot of predominantly numerical operations.

Using GPUs for numerical and scientific computation might sound as an obvious idea particularly after we have described the motivation for doing so; however, for its time, it was a very revolutionary idea. Your author recalls a fair amount of skepticism among people from both industry and academia when this idea was first proposed by some GPU evangelists. It took time to sink in. Now, nobody bats an eyelid if they hear that GPUs are being used for high performance computing. In fact, today GPUs can do most of the heavy lifting when it comes to computational work.

In this section, we shall introduce the CUDA programming language, analyze its constructs, and briefly cover some advanced features. Note that this section just provides a cursory introduction to the CUDA language and runtime. For a much more detailed treatment, readers should consult books on the CUDA programming language [Farber, 2011].

6.3.1 GPU ISAs

As mentioned in Section 6.2.2, the process of compiling a program on a GPU is a two-step process. The user writes her program using a variant of C++which can be one of the popular GPU programming languages: CUDA or OpenCL. Let us describe NVIDIA's CUDA toolkit. CUDA is an abbreviation for *Compute Unified Device Architecture*. It is an extension of C++ where a user writes a program for NVIDIA's entire line of GPUs. A typical CUDA program looks almost like a C++ program with some additional directives that specify which part of the code needs to run on the CPU, and which part of the code is meant for the GPU.

NVIDIA provides a dedicated compiler called *nvcc* that can be used to compile CUDA code. It uses separate tools to compile the CPU code and the GPU code (see Figure 6.8). The GPU code is first processed by a C++ preprocessor that replaces macros. Then a compilation pass compiles the code meant for the GPU into the PTX instruction set, which is a virtual instruction set. PTX stands for Parallel Thread eXecution. It is a RISC-like ISA with an infinite set of registers, where the compiler generates code mostly in the single assignment form – each variable is assigned a value exactly once. Using a virtual instruction set is preferable because there is a large diversity in the underlying hardware, and if we generate code for one kind of hardware, the code will lose its portability. Subsequently, we generate a fat binary, which contains the PTX code, and also contains the machine code for different popular models of GPUs. In other words, a single binary contains different images, where each image corresponds to a specific GPU ISA. Simultaneously, we compile the C++ part of the code using standard C++ compilers, embed references to GPU functions, and link functions provided by the CUDA library. *nvcc* finally produces one executable that contains both the CPU and the GPU code.

When the program is run, the runtime dispatches the PTX code to the GPU driver that also contains a compiler. If we are not using a pre-compiled binary, then this compiler performs just-in-time (JIT) compilation. The advantages of just-in-time compilation is that the code can be optimized for the specific GPU. Given that PTX assumes a virtual machine, specific optimizations need to be made at a later stage to generate the final machine code. Furthermore, unlike general purpose processors, GPGPUs are still not completely standardized; fairly invasive changes are happening every generation. Hence, to ensure that code written in the past runs efficiently is a challenge, and this necessitates compilation at runtime. The PTX code is compiled to SASS (Shader ASSembler) code, which is native to the machine. It can be generated separately from the PTX binary using the CUDA utility *ptxas* as well.

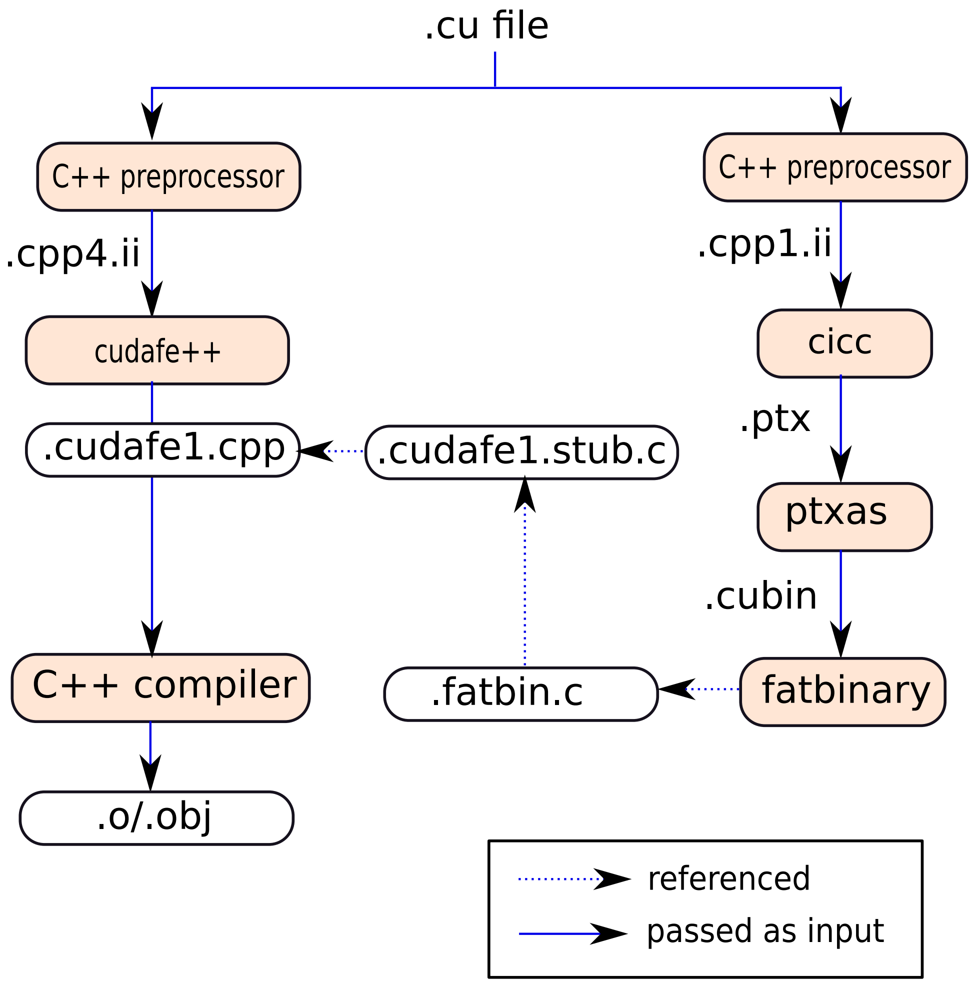

Figure 6.8: Compilation flow of *nvcc* (source [NVIDIA Inc., 2020])

6.3.2 Kernels, Threads, Blocks, and Grids

Let us now look at the basics of writing CUDA code. In this section, we shall describe some basic concepts. The source of this material is NVIDIA's documentation for the CUDA 10 toolkit [NVIDIA, 2018].

A basic function that is to be executed on a GPU is known as a *kernel*. It typically represents a function that needs to be invoked for each item in a list of items. For example, if we are adding two vectors, then the kernel can be a simple add function that adds two elements in a vector. Each kernel is called by a CUDA thread, where a thread is defined as a process (running instance of a program) that can share a part of its address space with other threads. In a GPU we can think of the threads as separate programs that execute the same code, share data via shared memory structures, and execute in parallel. Each such thread has a unique thread id, which can be obtained by accessing the *threadIdx* variable. To better explain these concepts, let us write our first CUDA code that uses multiple parallel threads to compute the sum of two vectors. Let us show only the function that will run on a GPU. We will gradually reveal the rest of the code.

```
__global__ void vecAdd (float *A, float *B, float *C){
    int idx = threadIdx.x;
    C[idx] = A[idx] + B[idx];
}
```

The *__global__* directive precedes every kernel indicating that it should run on a GPU. Let us now explain the built-in *threadIdx* variable. In the CUDA programming language threads are grouped into blocks of threads. A *block* of threads contains a set of threads, where each thread operates on a set of variables assigned to it. The threads in a block can be arranged along 1D, 2D, and 3D axes. For example if we are working on a cube, it makes sense to arrange the threads as per their x, y, and z coordinates. *threadIdx* in this case has three components: $threadIdx.x$, $threadIdx.y$, and $threadIdx.z$. For the code that we have shown, threads are arranged along a single dimension, hence we only use $threadIdx.x$ to get the index of the thread. We further assume that if we have N threads, then each vector also has N elements: assign one element to each thread. For each thread we read the corresponding array index, get the data values, add them, and write the result to the array C. If all the threads work in parallel, then we can potentially get an $N\times$ speedup.

The main advantage of arranging threads as a 1D chain, 2D matrix, or a 3D cuboid is that it is easy to partition the data among the threads because the arrangement of the threads mimics the structure of the data. Thread blocks typically cannot contain more than 768 or 1024 threads (depending on the architecture). On similar lines we can group blocks of threads into a grid.

Let us now show an example of a matrix addition kernel that uses a 2D block of threads. In this case we shall show a part of the *main* function that invokes the kernel.

```
__global__ void matAdd(float A[N][N], float B[N][N], float C[N][N]){
    int idx = threadIdx.x;
    int jdx = threadIdx.y;
    C[idx][jdx] = A[idx][jdx] + B[idx][jdx];
}

int main(){
    ...
    dim3 threadsPerBlock(N, N);
    matAdd<<<1, threadsPerBlock>>>(A, B, C);
    ...
}
```

In the *main* function we define an object called *threadsPerBlock* of type $dim3$. $dim3$ is a built-in type, which is a 3-tuple that can contain up to 3 integers: x, y, and z. If any integer is unspecified, its value defaults to 1. In this case, we are defining *threadsPerBlock* to be a pair of integers, which has two elements: N and N. The third element has a default value of 1. Thus, the value of the variable *threadsPerBlock* is $\langle N, N, 1\rangle$. Subsequently, we invoke the kernel function *matAdd* that will be executed on the GPU. Between the angle brackets, $<<<$ and $>>>$, we specify the arrangement of blocks in the grid, and the arrangement of the threads per block. In this case we have a single block in the grid, hence the first argument is 1; however, we arrange the threads within the block as an $N \times N$ array. This is because the second argument is the *threadsPerBlock* variable, which we set to $\langle N, N, 1\rangle$. The GPU subsequently creates N^2 threads.

In the code of the kernel (function *matAdd*), we find the x and y coordinates of each thread by accessing the variables *threadIdx.x* and *threadIdx.y*. They are stored in the variables *idx* and *jdx* respectively. Since the threads are arranged the same way as the data, we can use *idx* and *jdx* to find the elements in each matrix that correspond to a given thread. Then we add the elements and set the corresponding element in the result matrix C to the sum. We thus have parallelism at the level of elements, which is elegantly exploited by arranging our threads in the same way as the underlying data.

We can subsequently group blocks into a grid using a similar mechanism. A grid is in principle a 3D structure of blocks, where every block has an x, y, and z coordinate. However, by setting a subset of these coordinates to 1 we can think of a grid as a 1D chain or a 2D matrix of blocks.

Let's say we want to add two $N \times N$ matrices, where $N = 1024$. Furthermore, assume that we cannot create more than 768 threads. In this case let us limit ourselves to 16 threads per dimension (assuming 2 dimensions). We can then create $N/16 \times N/16$ blocks, where each block's dimensions are 16×16. The resultant code is shown below.

```
__global__ void matAdd(float A[N][N], float B[N][N], float C[N][N]){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int jdx = blockIdx.y * blockDim.y + threadIdx.y;
    C[idx][jdx] = A[idx][jdx] + B[idx][jdx];
}

int main(){
    ...
    dim3 blockDimensions (N/16, N/16);
    dim3 threadsPerBlock(16, 16);
    matAdd<<<blockDimensions, threadsPerBlock>>>(A, B, C);
    ...
}
```

Similar to *threadIdx*, *blockIdx* stores the coordinates of the block. The variable *blockDim* stores the dimensions of each block. It has an x, y, and z component, which are represented as *blockDim.x*, *blockDim.y*, and *blockDim.z* respectively. Blocks of threads are meant to execute completely independently on the GPU. They can be scheduled in any order. However, threads within a block can synchronize between themselves and share data. For synchronizing threads, we can call the *__syncthreads*() function, which acts as a barrier for all the threads in the block. Let us define a barrier. A *barrier* is a point in the code where all the threads must reach before any of the threads is allowed to proceed past it. This is graphically shown in Figure 6.9.

Definition 33
A barrier *is a point in the code where all the threads must reach before any of the threads is allowed to proceed past it.*

Definition 34

- *A kernel is a function in CUDA code that executes on a GPU. It is invoked by the host (CPU) code.*

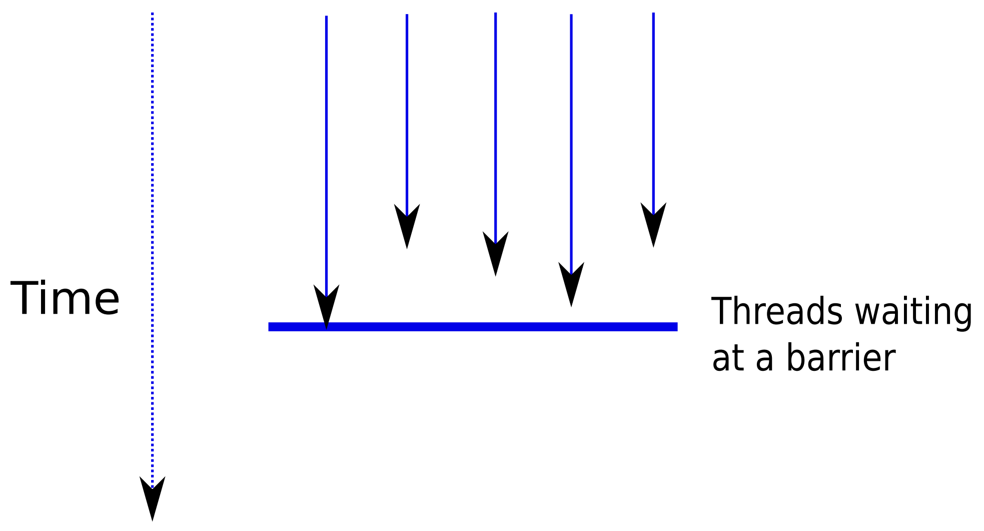

Figure 6.9: A barrier in the code

- *A thread in the context of a GPU is a process running on the GPU that is spawned at runtime. Similar to CPU threads, different GPU threads can share data among each other. However, the rules for sharing data are far more complex. While invoking a kernel we typically specify the number of threads that need to be created. They are created by the runtime. Each thread is assigned some data for performing its computation. This depends on its position within the* block*. In general, the threads execute in parallel, and this is why GPUs provide very large speedups.*
- *A block is a group of threads, where the threads can be organized in a 1D, 2D, or 3D arrangement. The threads in a block can share data, and can synchronize with each other.*
- *Similar to threads in a block, blocks are arranged in a grid in a 1D, 2D, or 3D arrangement. Blocks within a grid are supposed to execute independently without any form of synchronization.*

6.3.3 Memory Access

In GPUs there are two separate memory spaces: one on the CPU, and one on the GPU. We need to explicitly manage the flow of data between these spaces, and ensure that the process of transferring data is overlapped with computation as much as possible.

GPUs have many kinds of memories. Each thread has a per-thread local memory, then it has a shared memory that is visible to the rest of the threads in the block, and the lowest level is called the global memory, which is visible to all the threads. Additionally, many GPUs provide access to different read-only memories. Two of the popular memories in this class are the constant memory (for read-only constants) and the texture memory. The latter is used to store the details of textures in graphics oriented tasks.

In CUDA while allocating a region in memory, we can optionally add a memory specifier, which indicates where the bytes will be stored. The list of valid specifiers is as follows.

Specifier	Meaning
__device__	The region resides in the GPU (device)
__constant__	Resides in the constant caches of the GPU
__shared__	Resides in the per-block shared memory
__managed__	Can be used from both the host (CPU) and the device

Let us now show the complete code for vector addition, which includes statements to allocate memory, transfer data between the CPU's memory and the GPU's memory, and free memory on the GPU.

Listing 6.1: CUDA code to add two vectors

```
 1  __global__ void vecAdd (float *A, float *B, float *C){
 2      int idx = threadIdx.x;
 3      C[idx] = A[idx] + B[idx];
 4  }
 5
 6  int main() {
 7      ...
 8      /* Allocate the arrays A, B, and C on the CPU */
 9      int size = N * sizeof(float);
10      A = (int *) malloc (size);
11      B = (int *) malloc (size);
12      C = (int *) malloc (size);
13
14      /* Initialize the arrays */
15      ...
16
17      /* Allocate A, B, and C on the GPU */
18      float *g_A = cudaMalloc (&g_A, size);
19      float *g_B = cudaMalloc (&g_B, size);
20      float *g_C = cudaMalloc (&g_C, size);
21
22      /* Copy vectors from the host to the device */
23      cudaMemcpy (g_A, A, cudaMemcpyHostToDevice);
24      cudaMemcpy (g_B, B, cudaMemcpyHostToDevice);
25
26      /* invoke the kernel */
27      vecAdd<<<1,N>>> (g_A, g_B, g_C);
28
29      /* Transfer from the device to the host */
30      cudaMemcpy (C, g_C, cudaMemcpyDeviceToHost);
31
32      /* Free the space on the GPU */
33      cudaFree(g_A);
34      cudaFree(g_B);
35      cudaFree(g_C);
36
37  }
```

First we allocate memory on the host for the three arrays (Lines 10 to 12). Then we create three arrays on the device (GPU) with the same dimensions. These arrays need to be allocated space on the GPU (Lines 18 to 20). We use the function *cudaMalloc* for this purpose, which allocates space in the global memory. Then, we need to copy the contents of the arrays from the host's memory space to the device's (GPU's) memory space. We use the *cudaMemcpy* function to copy the arrays; this function takes three arguments: destination array, source array, and the direction of the transfer. The third argument (direction of the transfer) specifies whether we are transferring data from the host to the device or from the device to the host. It thus can have two values: *cudaMemcpyHostToDevice* and *cudaMemcpyDeviceToHost*.

Then we invoke the kernel in Line 27. We are using N threads and 1 block. Furthermore, this is a synchronous call, which means that we wait till the GPU has computed the result. Once this is done, we transfer the contents of the array g_C from the device to the host. We again call the *cudaMemcpy* function; however, this time data is transferred in the reverse direction. Finally, we free the arrays on the device in Lines 33-35 using the function *cudaFree*.

For multi-dimensional arrays we can use the function calls $cudaMallocPitch()$ and $cudaMalloc3D()$, which are used to allocate 2D and 3D arrays respectively. It is recommended to use these functions rather than using $cudaMalloc$ because these functions take care of the alignment issues in data. Additionally, we have similar functions to copy data from the device to the host and vice versa: $cudaMemcpy2D$ and $cudaMemcpy3D$.

6.3.4 Streams, Graphs, and Events

Streams

The code in Listing 6.1 is to a certain degree inefficient because it has three well defined phases – transfer the data to the GPU, perform the computation, and transfer the results back to the CPU – that execute sequentially even when they are not required to. It would have been much better if we could pipeline these phases such that the GPU's compute engines are not idle when the data is being transferred to them. Additionally, if we consider multiple kernels, then one kernel needs to wait till all the outputs of the previously executed kernel have been transferred to the CPU. This is again inefficient.

To solve problems of such a nature, the designers of CUDA introduced the concept of streams. A *stream* is defined as a sequence of commands that are meant to execute in order. The commands can be initiated by different threads on the host.

We can create different streams. There is no ordering of commands across the streams – they are independent. Depending on the compute capabilities of the device it might allow different kinds of overlaps to occur, or it might preclude them.

Let us explain with an example. Consider the code in Listing 6.1 once again. It has three distinct phases: transfer to the GPU (ToGPU), compute the result (*Compute*), and transfer to the host (*ToCPU*). Without any streams, their execution is as shown in Figure 6.10(a). There is no overlap between the phases. Let us now create two streams where we partition the arrays into two parts. If the arrays have N elements each, let the two parts cover the indices $[0, \lfloor N/2 \rfloor]$, and $[\lfloor N/2 \rfloor + 1, N - 1]$ respectively. Stream 1 is assigned the first half of indices, and stream 2 is assigned the second half. In our code that performs simple addition, stream 1 and stream 2 do not have any dependences between them.

Figure 6.10(b) shows the timeline of the execution with streams. We divide each phase into two parts, and distribute it among the two streams. Let us quantitatively compare the difference in the execution time. Assume that the three phases take 1 unit of time each. In Figure 6.10(b), each phase takes 0.5 units of time. Stream 1 starts at time $t = 0$ and ends at $t = 1.5$. Stream 2 starts at $t = 0.5$, and ends at $t = 2.0$. We thus observe that by using streams, we speed up the execution from 3 units of time to 2 units of time. We reduced the time of execution by one-third. Depending on the amount of resources, and the nature of overlaps that are allowed, we can increase the speedup even more by having more streams. For example, some devices allow the concurrent execution of kernels and some allow concurrent data transfers.

Graphs

In the CUDA framework, the costs of launching a kernel and managing the data transfers between the CPU and GPU are high. This cost is even more pronounced when we have a lot of kernels that run for a short duration. Even if we group kernels into streams, the static overhead of setting up the kernels in

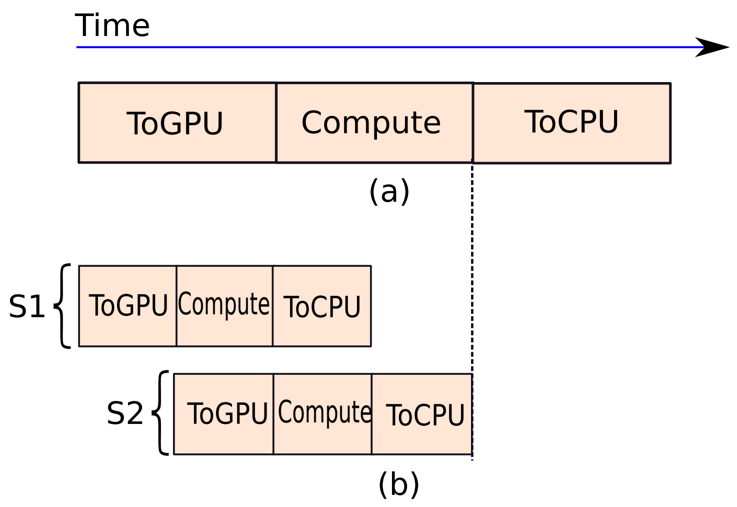

Figure 6.10: A sample execution with two streams

the GPU, loading their instructions, and initializing the GPU's hardware structures, does not reduce. In such scenarios CUDA graphs are useful.

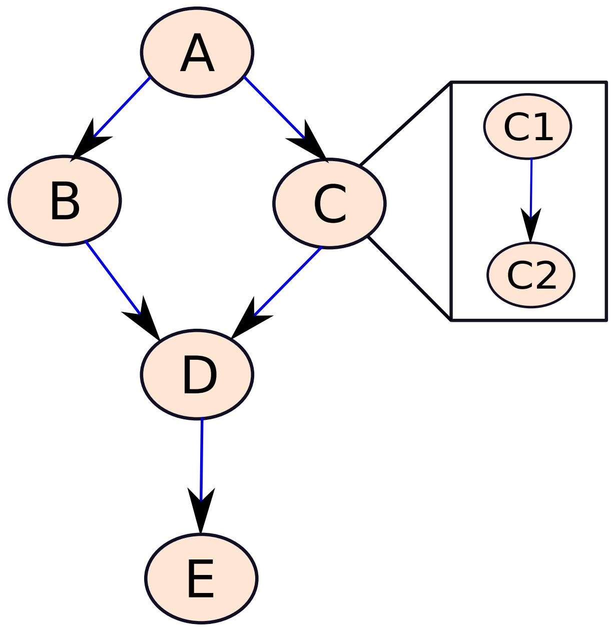

Figure 6.11: A graph in CUDA (the example shows a graph where node C is a subgraph)

A graph is a data structure that contains a set of vertices (nodes) and edges (see Definition 11 in Section 2.3.2). As shown in Figure 6.11, each edge joins two nodes. In CUDA, a graph can contain the following types of nodes:

1. Kernel (runs on the GPU)
2. Function call on the CPU

3. Memory copy and initialization

4. Another child graph

The programmer creates a graph by specifying the kernels, the nature of the data transfer, the CPU function calls, and the dependences between them. If there is an edge from node A to node B, then it means that task B starts only after task A executes completely. This graph is first validated and then the CUDA runtime prepares itself for running the tasks associated with the graph. The advantage of this approach is that when an ensemble of tasks is presented to the CUDA runtime, it can reduce the cost of launching kernels, and setting up data transfers, significantly. This is done by pre-computing the schedule of actions, prefetching data and code, and allocating resources for intermediate values and prefetched data.

Events

Programming a GPU is a skillful job. Unlike programming regular multicore processors, a GPU is a very complex framework. Programmers need to be aware of the details of the hardware, the way in which CUDA programs leverage the features of the hardware, and have an accurate knowledge of the performance bottlenecks in their program. Let us focus on the last point. CUDA programmers should know how much time different portions of their program are taking to execute. They can use CUDA events for this purpose. The programmer can create CUDA events that record the time at which a stream of commands started, and when it ended. The function *cudaEventElapsedTime* can be used to find the time between the two events: start of a stream's execution and end of a stream's execution. This information can be used to optimize the CUDA program.

6.4 General Purpose Graphics Processors

Over the last 10-15 years (as of 2020) many new designs of GPUs have come up; however, the broad architecture of a GPU has more or less stayed the same barring small modifications and improvements made every new generation. We shall describe a representative architecture in this section, which is broadly similar to NVIDIA Volta, even though it is not exactly the same.

6.4.1 Overview of the Architecture of a GPU

Figure 6.12 shows the reference architecture of a GPU. Let us start with the interface. There are several ways in which the GPU can be connected with the CPU. It can be on the same die as the CPU like Intel Sandy Bridge, or it can be housed separately. In the latter case, the CPU and GPU need to be connected with a high bandwidth interconnect such as PCI Express.

The interconnect is connected to the Thread Engine (GigaThread Engine in Figure 6.12) whose role is to schedule the computation (graphics and general purpose) on different compute units. Given that a GPU has a lot of compute units, programmers typically run a lot of multithreaded code on the GPU. It is too expensive to let the operating system or other software units schedule the threads as is the case in a normal CPU. It is best to have a hardware scheduler that can very quickly schedule the threads among the compute units.

In the architecture shown in Figure 6.12, we divide the compute units in a GPU into six separate clusters – each cluster is known as a GPC (Graphics processing cluster). Since a GPU is expected to have a lot of cores, we shall quickly see that it is a wise idea to divide it into clusters of cores for more efficient management of the tasks assigned to the cores.

Finally, in the high level picture, we have a large L2 cache, which as of 2020 is between 4-6 MB. The GPU has multiple on-chip memory controllers to read and write to off-chip DRAM modules. In this case the High Bandwidth Memory 2 (HBM2) technology is used.

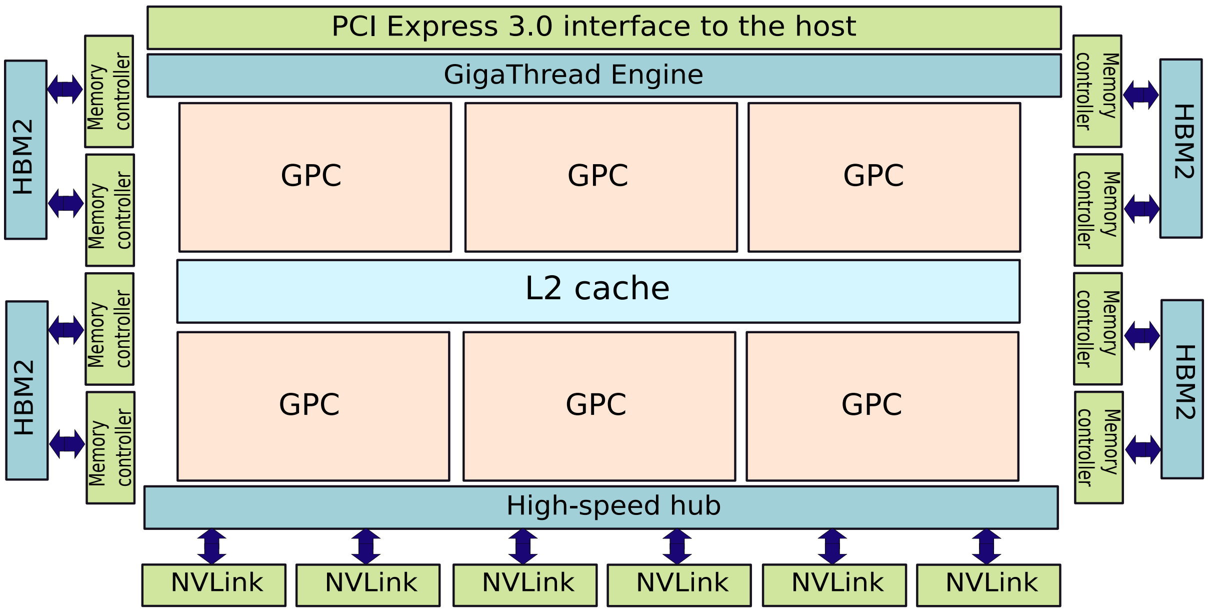

Figure 6.12: Architecture of a GPU (adapted from [NVIDIA Inc., 2017])

Most high performance systems today are multi-GPU systems. A large problem is split into multiple parts and each part is assigned to a separate GPU. The GPUs need to coordinate among themselves to execute the problem. As a result a very high bandwidth interconnect is required to connect the GPUs. NVIDIA created the NVLink interconnect that can be used to create such multi-GPU systems. The architecture shown in Figure 6.12 has six NVLink controllers that can be used to communicate with other sister GPUs.

6.4.2 Structure of a GPC

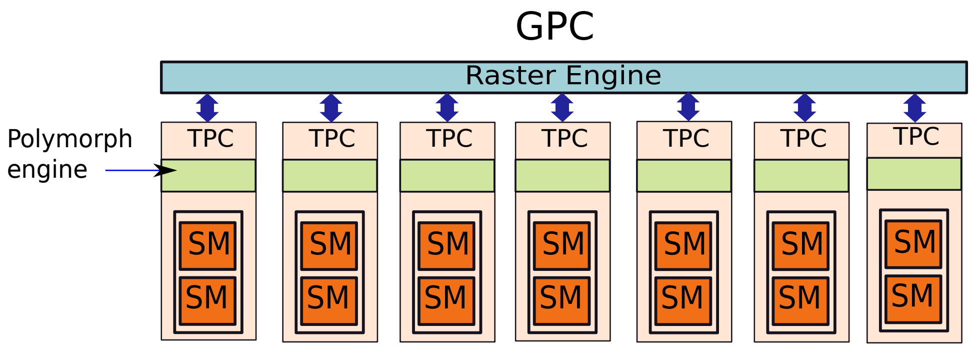

Figure 6.13: Architecture of a GPC (adapted from [NVIDIA Inc., 2017])

Let us now look at the structure of a GPC in Figure 6.13. Each GPC has a rasterization engine (referred to as the *Raster Engine*), which does the job of pixel rasterization (see Section 6.2.5). This unit is connected to seven Texture Processing Clusters (TPCs). Even though the TPC has maintained its historical name – as of today it consists of two distinct parts. The first is a vertex processor called the

Polymorph Engine (see Section 6.2.4) and the second is a set of two compute engines called Streaming Multiprocessors (SMs).

Way Point 5

- *In our reference architecture the GPU has six GPCs.*
- *Each GPC has a large Raster Engine for rasterization, and seven TPCs.*
- *Each TPC has a vertex processor called the Polymorph Engine, and two SMs (Streaming Multiprocessors).*
- *We thus have a total of 84 SMs.*

6.4.3 Structure of an SM

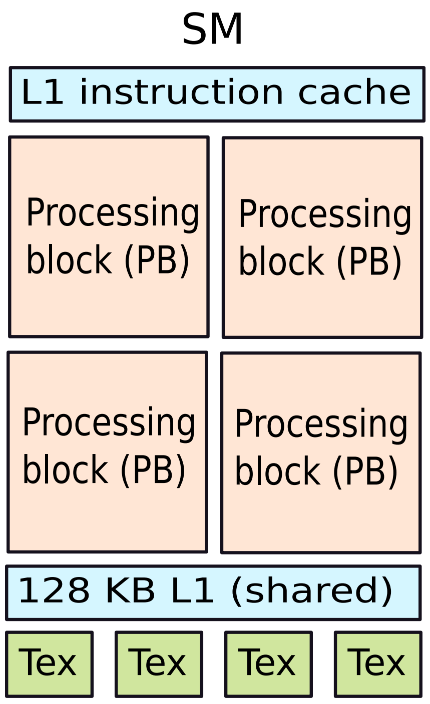

Figure 6.14: Layout of an SM (adapted from [NVIDIA Inc., 2017])

Figure 6.14 shows the structure of an SM. An SM can further be sub-divided into two parts: memory structures and groups of simple cores. Let us look at memory structures first. In GPUs, we have the following types of memory structures within an SM. Note that they need not be separate structures in every design. Some designs use the same structure for multiple functions.

Instruction Cache This is very similar to an i-cache in a regular processor. It contains the instructions that need to be executed.

L1 Cache This cache stores regular data that cores write (similar to a regular data cache).

Texture Cache This cache contains texture information. This information is provided to the texture units that color the fragment with a given texture.

Constant Cache Stores read-only constants.

Shared Memory This is a small piece of memory (64-128 KB) that all the cores in an SM can access. This can explicitly be used to store data that all the cores can quickly reference. CUDA programs can be directed to store arrays in the shared memory by using the *__shared__* specifier. The latency of this shared memory unit is typically far lower than that of the L2 cache, and other memory structures beyond it.

Some older GPUs had many kinds of such memories. However, there is a trend to unify them and have fewer memory structures. In our reference architecture each SM has an 128 KB L1 data cache that also acts as the shared memory. It is used to store all the data that the SMs need. Additionally, we have a single L1 instruction cache that is shared by all the cores in the SM.

Let us now focus on the compute parts of an SM. Each SM has four processing blocks (PBs) that contain cores and special computing units. They are used to perform all the mathematical processing in a GPU. Additionally, each SM has four texture processing units that process texture information (shown as *Tex* in the Figure). The job of each such unit is to fetch, process, and add textures to the rendered image.

The Compute Part of an SM: Processing Block

Figure 6.15 shows the detailed structure of a processing block (PB) in an SM. It is a simple in-order pipeline. Instructions pass through an instruction buffer (L0 i-cache), a scheduler based on a scoreboard (see Section 5.6.5), a dispatch unit, a large register file, and then they enter the compute cores. Other than the scheduler, the rest is similar to a regular in-order pipeline. We shall discuss the scheduling aspect shortly in Section 6.4.4.

In our reference architecture, a PB consists of 16 integer cores, 16 single precision floating point cores, 8 double precision floating point cores, and two tensor processing cores (for matrix operations). Each core is very simple: slightly more sophisticated than a regular ALU. When we have so many cores, they are bound to have a large memory requirement. We thus need 8 Ld/St units for accessing memory. In addition, scientific programs use a lot of transcendental and trigonometric functions. As a result, we need a special function unit (SFU) to compute the results of all of these functions. Additionally, SFUs have special support for interpolating the color of pixels. Recall that we had discussed in Section 6.2.6 that in the fragment processing stage, we need to interpolate the color of pixels based on the colors of adjoining pixels or the colors of the vertices of the triangle that the pixel is a part of. SFUs have special hardware to support these operations.

The cores are typically referred to as SPs (streaming processors). They contain fully pipelined functional units. Additionally, most GPUs support the multiply-and-add (MAD) instruction, which is of the form: $a = a + b * c$. Such instructions are very useful while performing linear algebra operations such as matrix multiplication.

Before delving into the details of the pipeline, let us understand the concept of a *warp*.

6.4.4 Concept of a Warp

We need to understand that we are not running a single pipeline. Instead, we are running a complex system with 40+ computing units. If we make this a free-for-all system, where any instruction from any thread can execute, then the entire system of threads will become very complex. We will need complex logic to handle branch statements, memory accesses, and dependences between the threads. This extra

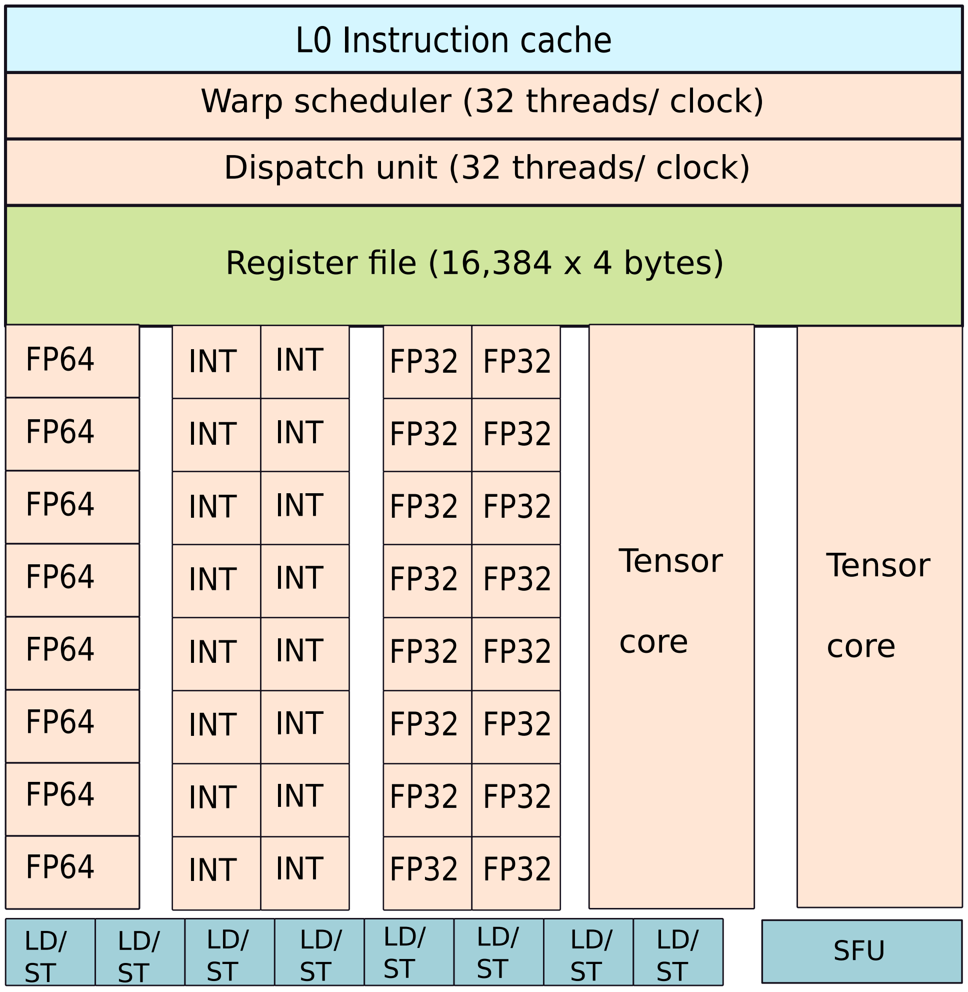
L0 Instruction cache
Warp scheduler (32 threads/ clock)
Dispatch unit (32 threads/ clock)
Register file (16,384 x 4 bytes)
FP64
INT
FP32
Tensor core
LD/ ST
SFU

Figure 6.15: Layout of a processing block (adapted from [NVIDIA Inc., 2017])

logic will increase the area of each core, and also increase its power consumption. In a system with hundreds of cores, we cannot afford this. As a result, some order needs to be imposed on the threads.

Modern GPUs (notably NVIDIA's GPUs) follow a SIMT model (single instruction, multiple thread) model. Here, we group a set of threads into *warps*, where a warp typically contains 32 threads. Each thread has the same set of instructions. When the threads in a warp execute, all of them start from the same point (same program counter). The scheduler maps each thread in the warp to an individual core, and then the threads start executing. However, note that the execution takes place in a special way. Threads do not run uncoordinated; they run in lockstep. This means that after a warp begins, the PB executes the first instruction in the warp for all the threads. Once the first instruction has finished executing for all the threads, it executes the second instruction for all the threads, and so on. The threads, of course, work on different pieces of data.

This is a very simple model of execution, and we do not need to have sophisticated hardware to ensure that all the threads are synchronized. The scheduler simply picks the next instruction in a warp and sends it to all the cores. Since all the cores execute the same instruction – albeit on different pieces of data – instruction fetch and dispatch are not difficult tasks. After all the cores finish an instruction, we send the next instruction. Now, this simple picture is riddled with corner cases. Let us see why.

Definition 35

- *The SIMT model – single instruction, multiple thread – is followed in most modern GPUs. Here, the threads run in lockstep. Conceptually, this is like all the threads executing an instruction, waiting till all the threads complete, and then moving on to the next instruction.*
- *The concept of the* warp *is integral to the SIMT model. It is a group of threads that are scheduled together on a PB and executed in lockstep.*

Varying Execution Times

First, instructions might take different amounts of time to execute. For example, it is possible that a memory instruction might not find its data in the L1 cache, and hence it needs to access the L2 cache. In this case it will take more time to complete than the rest of the instructions who possibly find their data in the L1 cache. Instead of letting the rest of the threads continue as in an OOO processor, we make all the threads wait till all the lagging instructions have completed. In general, this is not a very serious problem because all the data for graphical and numerical tasks is typically co-located. We normally do not deal with a lot of irregular memory accesses where data can be present at arbitrary locations. Nevertheless, it is always possible that accesses straddle cache lines and a few of these lines are not present in the cache. In such cases, the SIMT model will lead to a severe performance degradation.

Branch Divergence and Predicated Execution

There is another problem is even more pernicious and difficult to solve. What if we have branches? In this case, different threads will clearly execute different sets of instructions. It will be hard to maintain the SIMT and lockstep properties. Consider an if-else statement. It is possible that based on the inputs, one thread might execute the *if* portion of the code, and the other thread might execute the *else* portion of the code. In this case the lockstep property will be violated. Furthermore, we need to have a mechanism to wait for the threads to converge. Let us explain with an example. Consider the code in Listing 6.2.

Listing 6.2: If-else statements in GPU code

```
1  if (x > 0) {
2      a = 1;
3      b = a + 3;
4      c = b * b;
5  } else {
6      d = 3;
7      c = d * d;
8  }
9  x = c;
```

If we just consider two threads, it is possible that one thread enters the *if* portion and the other thread enters the *else* portion. Let us refer to the threads as threads 1 and 2 respectively. A normal processor would make thread 1 execute Lines 2-4 and then directly jump to Line 9. Similarly, it would make thread 2 execute Lines 6 and 7 and then proceed to Line 9. However, this requires complex logic to compute branch targets, and add offsets to program counters. The two threads will follow divergent paths till they reconverge at Line 9.

Given the fact that thread 1 needs to execute one more instruction as compared to thread 2, we need to make thread 2 wait for the time it takes to execute one instruction. Subsequently, both the threads can start to execute the instruction at Line 9 (point of reconvergence) at the same point of time in lockstep. This is a very complex mechanism and is expensive. Furthermore, it is possible to have nested branches within the *if* portion or the *else* portion. This will further complicate matters.

To keep things simple, GPUs use predicated execution. In this model, all the threads follow the same path. This means that thread 1 processes all the instructions – Lines 2 to 9 – and so does thread 2. However, processing an instruction does not mean executing it. Here is what we mean.

- Thread 1 executes the instructions at Lines 2 – 4.
- However, when it comes to the instructions at Lines 6 and 7, it ignores them. Nevertheless, it waits for other threads to finish executing them, if they have to.
- On similar lines, thread 2 ignores the instructions at Lines 2 – 4. It is not the case that it ignores these instructions and moves ahead. Instead, it waits for thread 1 to finish executing these instructions. **The threads still move in lockstep.**
- Thread 2 executes the instructions at Lines 6 and 7.
- Finally, both the threads reconverge at the instruction in Line 9.

Definition 36 summarizes this discussion and formally defines predicated execution in the context of GPUs.

Definition 36
Predicated execution *refers to a paradigm where a thread executes (processes) instructions belonging to both the paths of a branch instruction. The instructions on the correct path are fully executed, and they are allowed to modify the architectural state. However, the instructions on the* wrong path *are discarded, and not allowed to modify the architectural state.*

In the context of GPUs, predicated execution is heavily used in the SIMT model. All the threads execute all the paths of branches in lockstep. They pretty much treat the instructions in a warp as a sequential piece of code. However, some of these instructions are on the wrong path. The hardware keeps track of these instructions and does not allow them to modify the architectural state. However, this does not mean that the thread moves ahead. The thread still follows the lockstep property, and waits for all the other threads to finish executing that instruction. The instruction scheduler maintains a mask for threads in a warp. Only those threads whose bit in the mask is 1 execute the instruction and the rest ignore it. Alternatively, we can say that if the i^{th} bit in the mask is 1, then it means that the current instruction is in the correct branch path for thread i.

For our running example (Listing 6.2), Figure 6.16 shows a graphical view of the predicated execution. The cross marks indicate that a given instruction is being ignored by a thread.

Using this method threads execute instructions in a warp very easily. Let us now briefly look at how predication is implemented. Let us associate a stack with every thread. Every time we enter the code of a branch (branch path) we push an entry on the stack. If we are on the correct path we push a 1, otherwise we push a 0. If we have nested branches (branches within branches), we do the same. Similarly, when we exit a branch path, we pop the stack. This means that for every line of code in a warp, we maintain a small stack with 1-bit entries. We execute a given instruction and commit it if all the entries in its associated stack are 1. This means that we are on the correct path of all the branches encountered so far. However, if this is not the case, then we ignore the instruction because we are on

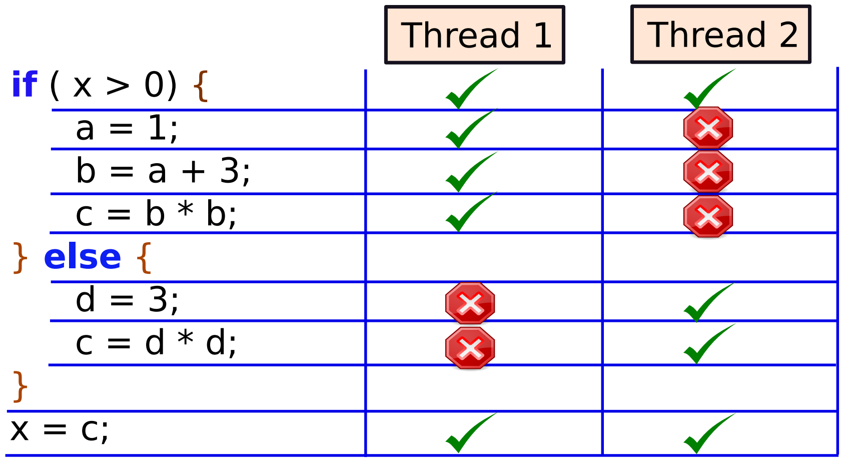

Figure 6.16: Graphical view of predicated execution

the wrong path of at least one branch. Note that if the stack is empty, then we execute and commit the instruction because this corresponds to the case, where the code is outside the scope of any branch. Before executing every instruction it is not possible to read the contents of the entire stack and compute a logical AND of the bits. We can succinctly store this information in a bit mask that contains 32 bits – one for each thread. If the i^{th} bit is 1, then it means that thread i can correctly execute the instruction. This bit mask is updated when we either enter the body of a conditional statement or exit it.

We show an example in Figure 6.17. Here, we consider three threads: 1A, 1B, and 2. We modify the code in Listing 6.2 (our running example) to add another nested *if* statement in the body of the first *if* statement. Threads 1A and 1B execute the body of the first *if* statement, whereas thread 2 does not. Inside the body of the first *if* statement, thread 1A executes the body of the second *if* statement, whereas thread 1B does not. The stack associated with the branch paths is shown in the figure beside the tick/cross marks. Please note how we push and pop entries into the stack as we enter and exit a group of conditional statements.

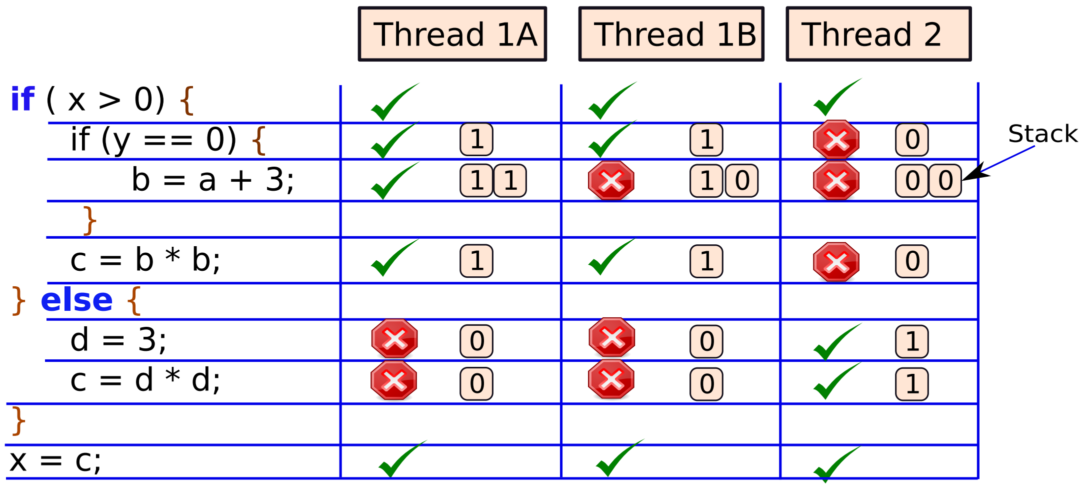

Figure 6.17: Using a stack for predicated execution in GPUs

Finally, we reach the point of reconvergence for all threads.

Definition 37
A point of reconvergence *is an instruction (point in the program) that is executed by all the threads in the warp and is just outside the scope of all previous conditional statements.*

Warp Scheduling

We need to appreciate that it is necessary to group computations into warps in a GPU. This keeps things simple and manageable. Otherwise, if we schedule every instruction independently, the overheads will be prohibitive; it will simply be impractical to do so. Hence, we have the concept of warps. However, we need to schedule warps and this requires a scheduler.

It is the job of the warp scheduler (typically a part of the PB or SM) to schedule the warps. It keeps a set of warps in a buffer. Every few cycles it selects a new warp and executes a few instructions from it. For example in the NVIDIA Tesla [Lindholm et al., 2008] GPU, the warp scheduler stores a maximum of 24 warps. Every cycle it can choose one of the warps and make it run. Later designs have modified this basic design, and have made the warp scheduler more complicated. For example, the NVIDIA Fermi GPU can select two warps at a time, and execute them simultaneously – each warp has 16 cores, 16 load/store units, or 4 SFUs at its disposal [Wittenbrink et al., 2011]. Later designs such as NVIDIA Kepler [GTX, 2014] have four warp schedulers per SM. In our reference architecture inspired by NVIDIA Volta [NVIDIA Inc., 2017], we divide an SM into four PBs, and we have one warp scheduler per PB.

The simplest strategy is to run a single warp at a time. However, running multiple warps at a time has some inherent advantages. Let us explain with an example. Consider an SM with 16 load/store units, and 16 ALUs. Furthermore, assume that a warp has 32 threads. Given that we execute the instructions in lockstep, all the instructions will be of the same type. Let us assume that we can either have memory instructions (load/store) or ALU instructions. This means that we can keep only half the number of functional units busy: either 16 ALUs or 16 load/store units. However, if we are able to schedule two unrelated warps at the same time, then we can do better. It is possible to make full use of the resources if we can overlap the execution of ALU instructions of one warp with the memory instructions of the other warp. In this case, one warp will use 16 ALUs, and the other warp will use the 16 load/store units. We will thus have 100% utilization.

Another possible option is to have a single warp scheduler with 32 ALUs and 32 load/store units. From the point of view of execution latency, this is a good idea; however, this is wasteful in terms of resources. It is best to have a heterogeneous set of units, and have the capability to schedule threads from multiple unrelated warps in the same cycle. If we have a good scheduler it will be able to ensure a high utilization rate of the functional units, and thus increase the overall execution throughput.

We can go a step further and schedule warps from different applications together, which is done in the NVIDIA Pascal architecture and later architectures. It does not strictly separate resources as previous architectures do. It allows a flexible allocation of functional units to threads of different warps depending upon the warp's criticality. Furthermore, NVIDIA Pascal also supports quick preemption. It is possible to quickly save the work of a warp, and switch to another task. Once that task finishes we can quickly resume the work of the unfinished warp.

Independent Scheduling of Threads in a Warp

Till now, we have treated a warp as a group of 32 threads that have a single program counter. Furthermore, we have a 32-bit mask associated with each warp. For an instruction, if the i^{th} bit is set, then only thread i executes it and commits the results to the architectural state. Otherwise, for thread i this instruction is on the wrong path, and it is not executed. This architecture unfortunately has a problem. Consider the code in Listing 6.3 in a system with a hypothetical 4-thread warp.

Listing 6.3: Deadlocks in a CUDA program

```
1 x = 0;
2 if (threadIdx.x < 2) {
3     while (x == 0) {}
4 } else {
5     x = 1;
6     y = 1;
7 }
```

We have four threads with ids 0, 1, 2, and 3 respectively. Two of the threads will execute the *while* loop (Line 3), and two threads will execute the code in Lines 5 and 6. If we run the code on a regular multicore processor, then there will be no deadlock. This is because first threads 0 and 1 will wait at the *while* loop. Then either thread 2 or thread 3 will set x equal to 1 in Line 5. This will release threads 0 and 1. However, in the case of a GPU with our lockstepped threading model, this code will have a deadlock. All the four threads will first arrive at the *while* loop in Line 3. For two threads (0 and 1) the *while* loop is on the correct path, and for the other two threads (2 and 3), it is on the wrong path. Threads 2 and 3 will not execute the loop; however, they will wait for threads 0 and 1 to finish executing the *while* loop. Unfortunately, this is where the problem lies. Threads 0 and 1 will never come out of the loop. They will be stuck forever because $x = 0$. Threads 2 and 3 will never reach Line 5 where they can set x to 1. This is an unacceptable situation. We can perform such synchronizing accesses between threads across different warps but not between threads in the same warp!

The solution is to maintain separate execution state for each thread. This includes a separate thread specific program counter, and a call stack. This however does break the notion of threads executing in lockstep, and has the potential for increasing the overheads significantly. Let us look at how the designers of NVIDIA Volta solved this problem.

They introduced the notion of *restricted lockstep* execution. This is shown in Figure 6.18. In the figure we define three blocks of instructions: W (while loop), X ($x = 1$), and Y ($y = 1$).

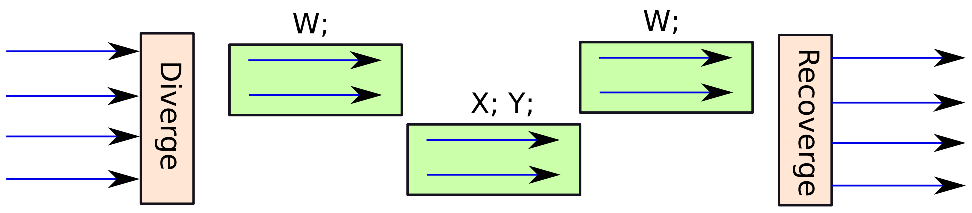

Figure 6.18: SIMT execution with per thread state

As we can see, the execution model is still SIMT. In any cycle, all the **active threads** in the warp still execute the same instruction. However, unlike our previous model, we do not proceed sequentially. We increase the degree of concurrency by executing the code blocks X and Y concurrently with the code block W. Let us follow the timeline. We first execute the code block W. We are not able to make progress because $x = 0$. Then we execute the code blocks X and Y. Subsequently, we execute the code block W once again. This time we are able to make progress because x has been set to 1. We thus leave the *if-else* block, and our divergent threads reconverge.

We can also force reconvergence between the threads by calling the CUDA function *__syncwarp*(). In general, the role of the GPU is to ensure as much of SIMT execution as possible. This means that it needs to group together as many active threads as it can per instruction, and also ensure that all threads make forward progress. The latter ensures that we do not have deadlocks as we showed in Listing 6.18.

The reader needs to convince herself that this method allows us to use lock and unlock functions in GPU threads similar to regular CPU threads.

6.4.5 The GPU Pipeline

Way Point 6

- *In our reference architecture, the GPU consists of a set of 6 GPCs, a large 6 MB L2 cache, 8 memory controllers, and 6 NVLink controllers.*
- *Each GPC consists of 14 SMs (streaming multiprocessors).*
- *Each SM consists of 4 processing blocks, an L1 instruction cache, and a 128 KB data cache.*
- *Each processing block (PB) contains 16 integer cores, 16 single precision FP cores, and 8 double precision FP cores. It additionally contains two tensor cores for matrix operations, 8 load/store units, and a dedicated special function unit.*
- *Each PB executes a warp of threads in parallel. The threads in the warp access the large 64 KB register file, and the L1 cache of the SM most of the time. If they record misses in these top level memories, then they access the L2 cache and finally the off-chip DRAM.*

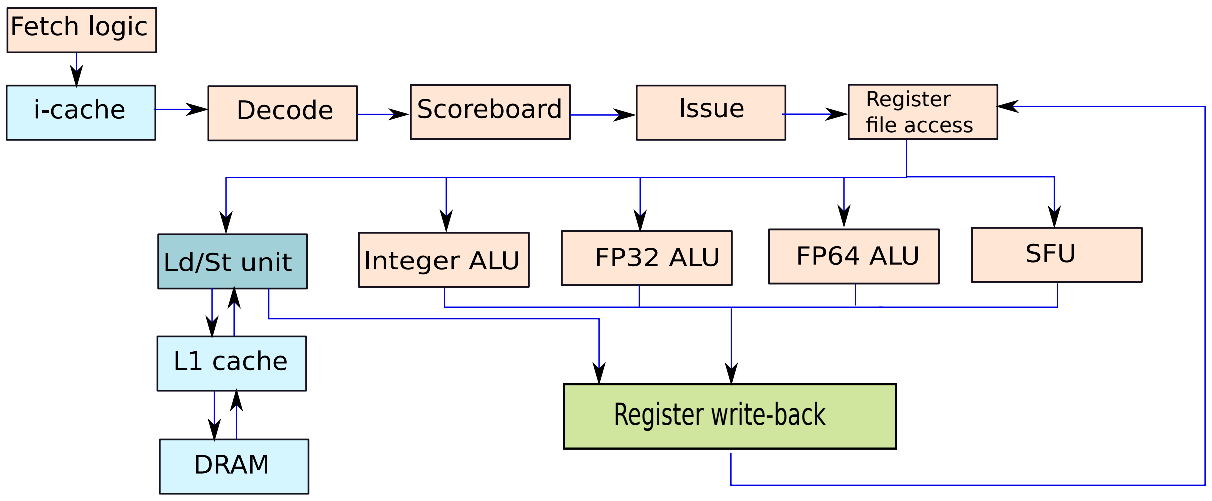

Figure 6.19: A GPGPU's pipeline

Figure 6.19 shows the pipeline of a GPGPU core. It is similar to a regular in-order pipeline as described in Section 2.1. Let us elaborate.

Once we have decided to schedule a warp, we read its instructions from the i-cache. We decode the instructions and while dispatching the instructions we check for dependences. We typically do not use expensive mechanisms like the rename table or reservation stations. They consume excessive amounts of power and are also not efficient in terms of area. We use the simple scoreboard based mechanism as described in Section 5.6.5. Recall that a scoreboard is a simple table that we use to track dependences between instructions. Once the instructions in a warp are ready to be issued, we send them to the register file. Unlike a CPU's register file, a register file in a GPU is a very large structure. It is almost as large as a cache – 64 KB in our reference architecture. We shall look at the design of a GPU's register file in Section 6.4.6.

To support lockstep execution of all the active threads, we need to read all their data at once. This requires a very high throughput register file. Once we have read all the data, we send it to the functional units. They compute the result, access memory (if required), and finally write the results back to the register file or the relevant memory structure.

6.4.6 The Register File

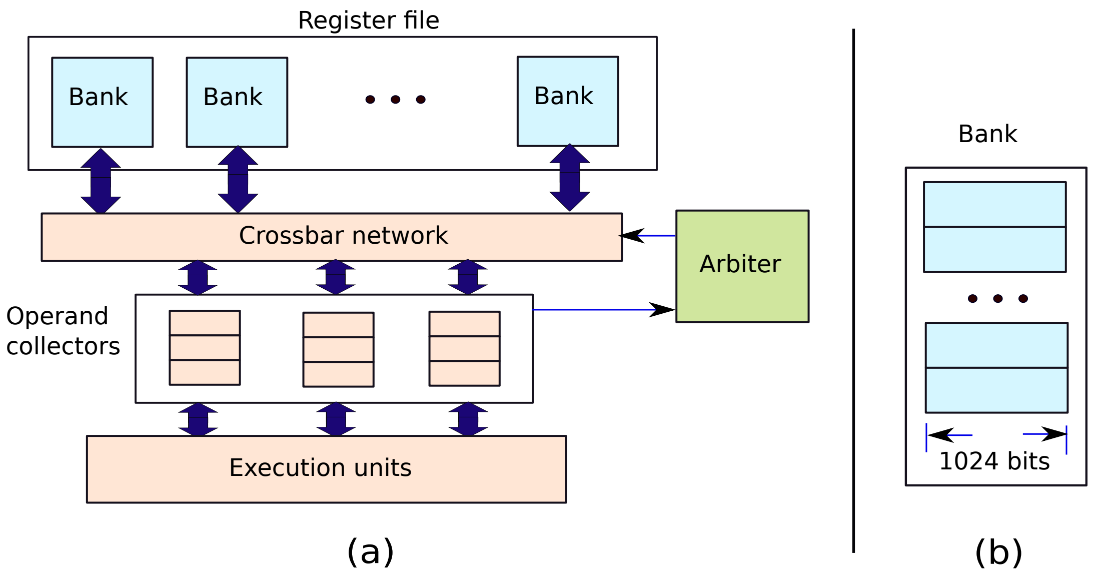

Figure 6.20: The register file in a GPU

The PTX ISA assumes an infinite number of registers. The advantage of this is that the PTX code can remain platform independent, and the code can be written in terms of virtual registers, which improves the effectiveness of a host of compiler optimizations. While generating the binary code we can assign real registers to the virtual registers. This can be done by the PTX assembler, *ptxas*, or in the process of JIT (just-in-time) compilation of the binary.

Let us now consider the design of the register file. In a GPU we need a very high-throughput register file given the bandwidth requirements. It is impractical to read and write data at the granularity of 4-byte words given that we have at least 32 threads running at the same time in a processing block. Consider a piece of code where all the threads in a warp use a 32-bit local variable. We need to create 32 copies of this variable. The total number of bytes that we need to allocate is $32 \times 32 = 1024$ bits. We thus set the block size in the register file to 1024 bits (or 128 bytes). This is shown in Figure 6.20(b) that shows a bank in a register file with a 1024-bit block size. A bank is defined as a subcache (see Section 7.3.2). We typically divide a large cache into multiple banks to increase the performance and reduce power consumption.

Let us now design a register file (see Figure 6.20(a)). Assume we have a 64 KB register file. We can divide it into 16 banks, where the size of each bank is 4 KB. If the size of a single entry is 128 bytes, we shall have 32 such entries in each register file bank. In a lot of modern GPUs that have many outstanding instructions and frequent warp switches, there are many memory instructions in flight. There is thus an elevated chance of *bank conflicts* – conflicting accesses to the same bank. This will further cause delays because we can process only one memory request at a time per bank. In addition, we cannot read and transfer 1024 bits at the same time; we need to read 1024 bits over several cycles. Moreover, we may have wider operands such as double precision values. In this case, we need to store the set of 32 values in multiple banks. All of these values have to be read, collected, and then sent to the execution units within the PB.

We thus create a set of buffers known as operand collectors, where each entry is associated with an instruction. It stores the values of all the source operands. We connect the banks and the operand collectors with a crossbar switch, which is an $N \times M$ switch. We have N ports for the banks in the register file, and M ports for the operand collectors. We can route data from any bank to any operand collector (all-to-all traffic). Once each entry in the operand collector receives all its source operands in entirety, the values are sent to the arrays of execution units within the PB.

6.4.7 L1 Caches

Similar to the register file, the memory bandwidth demands are very high for the caches particularly the L1 cache that sees all the accesses. If the L1 hit rate is high, then the traffic that reaches the L2 cache is significantly reduced. Furthermore, since the L2 cache is much larger, we can afford to create many more banks to sustain more parallelism. Hence, out of all the memory structures the L1 cache is the most critical.

In the case of memory accesses, we will have a set of accesses: one for each thread. Since we have 8 load/store units in our reference architecture, we can issue 8 or 16 memory accesses per cycle depending upon the parallelism in the load/store units. The second stage is an arbiter, which figures out the bank conflicts between the accesses. It splits the set of accesses into two subsets. The first subset does not have any bank conflicts between the accesses, and the second set of accesses have bank conflicts with the first subset, and might have more conflicts between them. The addresses in the first subset are then routed to the L1 cache. If the accesses are writes, then the process ends here. If the accesses are reads, we read the data and route them to the operand collectors associated with the register file. Subsequently, we send the requests from the second set that do not have bank conflicts, and so on.

Let us now look at some special cases. Assume we have a miss in the cache. We then use a structure like an MSHR (miss status handling register) to record the miss (see Section 7.4.2 for the definition of an MSHR). Similar to a traditional MSHR, we can merge requests if they are for different words in the same block. Once the block arrives at the cache, we lock the corresponding line to ensure that it is not evicted, and *replay* the load/store accesses for different words within that block from the MSHR. Similarly, we replay the instructions that access words within the block and could not be sent to the cache because of bank conflicts. Once the accesses are done, we unlock the line.

6.5 Summary and Further Reading

6.5.1 Summary

Summary 5

1. *General purpose processors are limited by their power consumption and IPC. In practice, the IPC almost never exceeds 4.*
2. *Linear algebra operations that form the core of much of today's scientific computing are often embarrassingly parallel and have a high ILP. This parallelism cannot be exploited by general purpose processors. Hence, we need specialized hardware.*
3. *Graphics processors (GPUs) were initially used for rendering graphics. Their core was a shader program whose job was to translate users' directives into graphical objects.*
4. *The input to shaders was a list of triangles, and the output was a set of triangles or pixels after applying various graphical effects.*
5. *The four parts of a basic rendering pipeline are the Vertex processor, Rasterizer, Fragment processor, and the Pixel engine.*
   (a) *The Vertex processor accepts a list of triangles as input, and then performs geometric transformations on them.*
   (b) *The Rasterizer converts all the triangles to a set of pixels. It can also optionally perform visibility calculations to determine which set of pixels (fragments) are visible in the final scene.*
   (c) *The Fragment processor has three major functions: interpolation of colors, texture mapping, and fog computation.*
      i. *It computes the color of each pixel using interpolation based techniques. There are two common approaches to do this: Goraud shading and Phong shading.*
      ii. *It adds texture information to each fragment.*
      iii. *Based on the distance of an object from the view point, it shades objects differently (fog computation). This provides a perception of distance to the human eye.*
   (d) *Then we have the pixel engine that computes the depth and visibility of each object, and applies transparency effects. If an object is transparent or translucent, then objects behind it are also visible.*
6. *The Vertex processor in modern GPUs has been replaced by a Polymorph engine. It performs the following roles.*
   (a) *Vertex fetching: It computes the coordinates of the objects in the scene and their orientation, and can add visual or geometric effects to the objects in the scene.*
   (b) *Tessellation: Break down all the objects into a set of triangles and creates a degree of fine detail on the surface.*
   (c) *Viewport Transformation: We typically render large scenes in a virtual coordinate system. However, all the objects need not be entirely visible on the screen (the viewport). In this stage we compute the parts of the objects that are visible.*

   (d) *Attribute setup: Annotate each triangle with the depth information for visibility calculations.*
   (e) *Stream output: Write the information computed by the Polymorph engine to memory.*

7. *Most GPUs have their own ISAs. NVIDIA GPUs can be programmed using the CUDA framework, which uses an extension of C++. The code is separately compiled for the CPU and the GPU, respectively.*
8. *CUDA programs are typically compiled to the PTX ISA, which is a virtual ISA. At runtime the compiler that is a part of the GPU driver converts PTX to native GPU code (SASS code).*
9. *A CUDA function that is invoked by the host CPU and runs on the GPU is known as a kernel. A GPU spawns multiple instances of the kernel, and assigns each instance to a thread. In the general case, the threads are organized as a 3D matrix – this is known as a block. The blocks can further be arranged in a 3D form – this is known as a grid. Every thread knows its coordinates in its block, and its block's coordinates in the grid. This information is used to split the input data among the threads.*
10. *A CUDA program has access to four types of memory: device (on the GPU), host (on the CPU), shared (resides in per-block shared memory), and managed (can be used from both the CPU and GPU).*
11. *The design of a GPGPU is as follows:*
    (a) *A GPGPU has a multitude of GPCs (graphics processing clusters), an L2 cache, and high speed links to memory.*
    (b) *A GPC further contains multiple TPCs (texture processing clusters), and each TPC contains multiple SMs (streaming multiprocessors).*
    (c) *Each SM consists of multiple processing blocks, where each processing block contains an i-cache, warp scheduler, dispatch unit, register file, a set of integer and floating point ALUs, load-store units, special function units, and a few tensor cores.*
    (d) *Typically, 32 GPU threads are grouped together as a warp. They execute together in lockstep.*
    (e) *The tensor cores contain units for performing matrix multiplication. This is useful in linear algebra and machine learning algorithms.*
12. *Most GPUs implement predicated execution where each thread processes instructions in both the correct and wrong paths of a branch. The instructions on the wrong path are dynamically converted to nops. This is known as predicated execution.*
13. *Efficient warp scheduling is very important. Scheduling multiple warps at a time makes better use of resources.*
14. *GPUs have very large and sophisticated register files. Their size is typically between 32 and 128 KB.*
15. *The L1 cache is divided into multiple banks. We divide the set of accesses by a warp into two sets: accesses that do not have bank conflicts between them, and accesses that have conflicts. The latter set of accesses are sent to the cache after the first set of accesses. They are said to be* replayed.
16. *The L2 cache in a GPU is connected to external DRAM using a high bandwidth interconnect.*

6.5.2 Further Reading

Readers should start with the CUDA programming language. The most authentic CUDA programming resources are available on NVIDIA's website [NVIDIA, 2018]. Readers can additionally consult the book by Rob Farber [Farber, 2011] on developing GPU applications. In addition to CUDA, interested readers can also learn OpenCL if they wish to program AMD GPUs. The book by Kaeli et al. [Kaeli et al., 2015] is a good starting point.

For the architecture of GPUs, the starting point is NVIDIA's documentation for their architectures: Tesla [Lindholm et al., 2008], Fermi [Wittenbrink et al., 2011], Kepler [Corporation, 2014b], Maxwell [GTX, 2014], Pascal [Corporation, 2014a], and Volta [NVIDIA Inc., 2017]. To conduct research in this area users need a software based GPU simulator. Some popular simulators are GPUTejas [Malhotra et al., 2014], and GPGPU-Sim [Bakhoda et al., 2009]. GPUWattch [Leng et al., 2013] is a power model that can be plugged into the GPGPU-Sim simulator. Programming GPUs is increasingly getting easier. The Theano [Bergstra et al., 2010] project proposes a Python based framework to run code on the GPU. For studying GPUs further we can write specialized micro-benchmarks to understand their performance characteristics using the approach proposed by Wong et al. [Wong et al., 2010], or we can use Hong et al.'s analytical models [Hong and Kim, 2009].

Exercises

Ex. 1 — What is a better idea: have more threads per block and less blocks per grid or have less threads per block and more blocks per grid? Assume that the total number of threads is the same.

Ex. 2 — Write CUDA code to implement the following algorithms:

1.Matrix multiplication.

2.Matrix multiplication using the Tensor Processing Units of the NVIDIA Turing GPU.

3.Solution of the Fourier heat equation.

4.Sorting 10 million numbers.

5.Find edges in an image.

Ex. 3 — How does a GPU core handle WAW and WAR hazards?

Ex. 4 — What are the pros and cons of having a large register file in the context of a CPU and a GPU?

Ex. 5 — Theoretically analyze the computation time of a matrix multiplication operation on a GPU and a CPU.

Ex. 6 — What is the *_syncthreads()* instruction in CUDA? When is it useful?

Ex. 7 — Can we replace the register file of a GPU with a block of shared memory? Explain you answer.

Ex. 8 — How do we detect and handle bank conflicts while accessing the first level cache in a GPU?

* **Ex. 9** — Describe the architectural mechanism for SIMT execution with per-thread state.

Design Problems

Ex. 10 — Understand the working of the GPU simulator *GPUTejas*$^{\text{TM}}$.

Ex. 11 — Implement predicated execution in *GPUTejas*.