Pending

Alexandre Carlessi Sahand Kashani-Akhavan

June 7, 2013

Introduction

A graphics processing unit, also known as a GPU, is a specialized electronic circuit initially designed for fast memory manipulations needed to accelerate the creation of images in a frame buffer, which are then outputted to a display for viewing. Nowadays, GPUs are present in almost all electronics, including, but not limited to embedded systems, mobile phones, personal computers, workstations, and game consoles.

Modern GPUs have highly parallel structures, making them much more effective than general-purpose CPUs for algorithms which process large blocks of data in parallel. Thus, GPUs have become very efficient at manipulating imagery, which consists of applying an algorithm parallely to all output pixels, which explains their abundant use in computer graphics.

In this report, we look at ways to apply

Programmability

GPUs were initially designed to accelerate texture mapping, polygon rendering, and geometry. The first GPUs had a fixed-function rendering pipeline, and could not be used for anything other than common geometry transformations and pixel-shading functions that were pre-defined in hardware.

With the introduction of the NVIDA GeForce 3 in 2001, GPUs added programmable shading to their capabilities, which allowed developers to define their own straight-line shading programs to perform custom effects on the GPU. Bypassing the fixed-function rendering pipeline opened the door towards many future graphics novelties, such as cel shading, mip mapping, shadow volumes, oversampling, interpolation, bump mapping, and many others.

The 2 main shaders were the fragment shader (also known under the name of pixel shader), and the vertex shader (also known under then name of geometry shader). The vertex shader processed each geometric vertex of a scene and could manipulate their position and texture coordinates, but could not create any new vertices. The vertex shader's output was sent to the fragment shader for further processing. The fragment shader processed each pixel and computed its final color, as well as other per-pixel attributes by using supplied textures as inputs. Soon, shaders could execute code loops and lengthy floating point math instead of straight-line code, which pushed them to quickly become as flexible as CPUs, while being orders of magnitude faster for image processing operations. The shaders were written to apply transformations to a large set of elements at a time, such as to each pixel of the screen, or to every vertex of a 3D geometric model.

Prior to the introduction of the GeForce 8800 GPU in 2006 in 2006, GPUs had different processing units for each type of shader. The GeForce 8800 pipeline merged the separate programmable graphics stages to an array of unified processors, which allowed dynamic partitioning of the computing elements to the different shaders. This allowed the GPU to achieve better load balancing. The unified processor array of the GeForce 8800 GTX is shown on Figure 1.

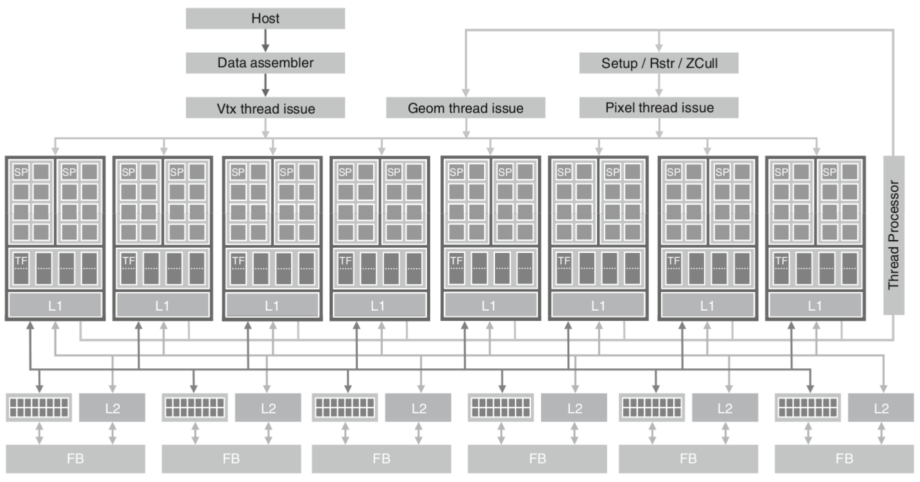

Figure 1: Unified programmable processor array of the GeForce 8800 GTX

The GPGPU Era

GPU hardware designs evolved towards more unified processors, and were getting more similar to high-performance parallel computers. Computations performed by programmable shaders mostly involved matrix and vector operations, for which GPUs were very well suited (processing blocks of data in parallel). The availability of high speed linear algebraic operations, as well as the unified procesors pushed scientists to start studying the use of GPUs for non- graphical calculations. This was achieved through the use of GPGPU techniques.

GPGPU, a shorthand for General Purpose computing on Graphics Processing Units, consists of using a GPU, which typically only handles computations related to computer graphics, to perform computations for applications which are traditionally handled by a CPU. Such a consideration is possible since GPUs support a functionally complete set of operations on arbitrary bits, thus can computer any value.

At the time, programmers could only interface with GPUs through graphics APIs such as the OpenGL or DirectX. However, APIs had been designed to only match features required in graphics. To access the computational resources, programmers had to map their problem into graphics operations so the computations could be launched through OpenGL or DirectX API calls. With this consideration in mind, programmers had to arrange their data in such a way to “trick” the GPU in performing the calculations defined in the programmers’ shaders as if they were graphics calculations, whereas in reality, they were scientific computations.

GPGPU concepts

There are 4 main GPGPU concepts.

1. Data arrays are equivalent to GPU textures.

The native data layout for CPUs is the one-dimensional array.

Higher dimensional arrays are available for programmer convenience, but are actually implemented as one-dimensional arrays, and compilers use linear transformations to adapt the indices accordingly.

On the other hand, GPUs use two-dimensional arrays as their native data layout, and are in fact textures.

To make a data array available to the GPU, the CPU would need to create the data, then map it to a GPU texture which a shader could later read in order to process.

In order to correctly use the memory available to a GPU, one needs to find a mapping between the CPU array indices, and the GPU texture coordinates.

Once the mapping is done, the CPU would then transfer the data towards the GPU texture.

The second GPGPU concept is that computation code (kernel) is equivalent to a shader.

There is a fundamental difference between the computing model of a CPU and a GPU, and this impacts the way one needs to think algorithmically.

GPUs follow the data-parallel programming paradigm, whereas CPUs follow the sequential programming paradigm.

As such, CPUs code is usually implemented as loop-oriented programs, since they have to iterate over all elements and apply a function to each one.

In contrast, GPUs have highly parallel structures which can apply the same algorithm to large blocks of data parallely, assuming that there is no dependence between the operations.

To show the contrast in the programming model, let's compare how one would compute the addition of 2 N-element vectors and store the result in the first vector.

Assume we have the following 2 vectors already pre-filled with their respective data:

int a[N]; int b[N];

The CPU code to perform this vector addition would look like this:

for (int i = 0; i < N; i += 1) a[i] = a[i] + b[i];

Note that the CPU will have to loop over all indices of the 2 arrays, and add each element one by one in a sequential way.

It is important to note that each of the N computations are completely independent, as for a given output index, there are distinct input memory locations, and there are no data dependencies between elements in the result vector.

For example, once we have computed a[0] = a[0] + b[0], its result will be of no help when it comes to computing a[1] = a[1] + b[1].

As such, assuming we have a computation unit with N parallel structures, we would be able to compute the vector addition without the need of any loop by assigning one vector element addition to each computation element.

This is easily done by adapting the index of the vector elements that are provided to each computation unit.

This is the core idea behind GPGPU computing: separating the identical, but independent calculations from one another, and assigning them to execution units which can then execute them at the same time.

Algorithms are extracted into computational kernels which are no longer vector expressions, but scalar templates of the algorithm that form a single output value from a set of input values.

These algorithms are implemented in shaders which will then calculate the independent computations parallely.

For the vector addition used above, the 2 vectors will have to be written into textures by the CPU, then the shader will read the appropriate elements from the texture to perform its independent computation.

The third GPGPU concept is that computations are equivalent to "drawing".

Indeed, the final output of a GPU is an "image", therefore all computations have to, in some way or another, write their "results" to the frame buffer for it to be available to the programmer.

Therefore, the programmer must tell the graphics API (either OpenGL or DirectX) to draw a rectangle over the whole screen (since our screens are rectangles), so that the fragment shader can apply its code to each pixel independently and output an answer.

If the API were not instructed to draw something that fills the whole screen, then the fragment shader's code would not be applied to all our data in the textures we created, but only to a subset of it.

By rendering the simple rectangle, we can be sure that the kernel is executed for each data item in the original texture.

The fourth GPGPU concept is feedback.

On CPUs, data is read from memory locations, and results are written to memory locations.

On GPUs, we just saw that the final output is written to the frame buffer.

However, a huge number of algorithms are not straight-line code, and require the GPU's result to be used as input for another subsequent computation.

To achieve this on a GPU, we need to execute another rendering pass

This is achieved by writing the current result to another texture, binding this texture as well as another input and output textures, and potentially also binding another shader for the algorithm to continue.

This is known as the ping-pong technique since one has to keep juggling between textures until the algorithm is done and the result is outputted to the frame buffer.

To recap all that is needed for GPGPU on graphics APIs, one needs to create data on the CPU, map it to GPU textures, write shaders to perform computations based on the data in the textures, and finally write the result back to the frame buffer.

As one can see, early GPGPU programming can quickly become quite tedious, even for simple algorithms, as one must understand the complete graphics rendering pipeline in order to trick the GPU in thinking it's performing graphics calculations, whereas the programmers are actually manipulating their data on GPU textures, and writing their kernels in shaders.

Indeed, graphics APIs were not intended for scientific computations, and are thus not easily programmable.

In order to fully benefit from the parallel processing power of GPUs without having to know anything about the graphics rendering pipeline, more computational oriented languages were created.