L20: GPU Architecture and Models

scribe(s): *Abdul Khalifa*

20.1 Overview

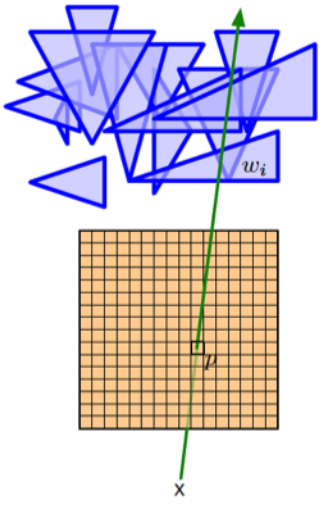
GPUs (Graphics Processing Units) are large parallel structure of processing cores capable of rendering graphics efficiently on displays. The original intent when designing GPUs was to use them exclusively for graphics rendering. People realized later that GPUs can in fact be used to process highly parallel data structures and became popular for solving problems in other scientific domains which require intensive computational power.

This notes will give a short history of graphics and then move to discuss GPU's pipeline. It then concludes with a section about GPU hierarchy and show an example of performance measure.

20.2 Graphics History

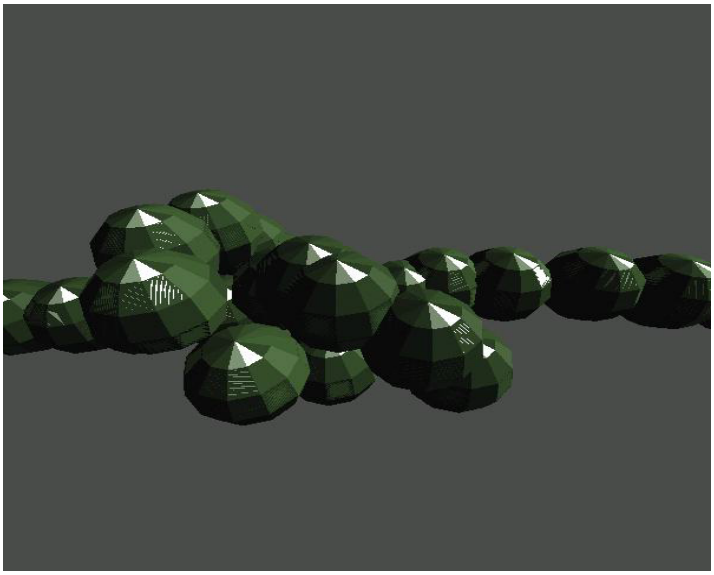
How a pixel is drawn on the screen? The process by which a computer transforms shapes stored in memory into actual objects drawing in the screen is known as *rendering*. And most well known and used technique during rendering is called *rasterization*. The basic concept can be described as shown in figure 20.1 where each pixel is drawn on the screen if there exists a trace of light (the arrow) that intersect one of the triangle shapes stored in memory. This simplistic view also shows that if a triangle is also further back than other surrounding triangles then this should be reflected on the display and if a triangle is on top of another triangle then the latter will be partly obscured by the former on the display. It is this approach whats makes the foundation for 3D graphic support on computer screens. Figure 20.2 shows few examples of images rendered with this approach. Eventually, GPUs are the hardware responsible for transforming the representation in memory to actual objects in the screen. And since this is needs to be done for every pixel, GPUs were developed to handle that in parallel.

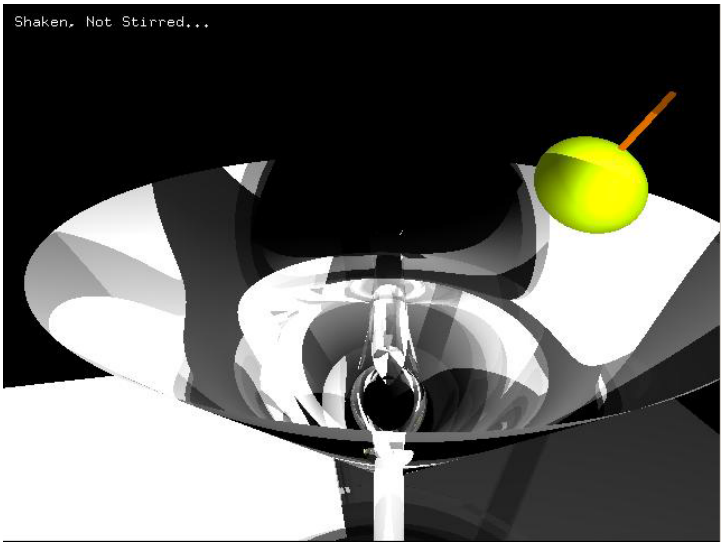
Figure 20.1: How pixels are drawn on display.



Earlier GPUs in the 1990s were offered with a *fixed functional* pipeline that was implemented directly via mainly two APIs: OpenGL and DirectX. This meant that GPUs were programmable with a low level

Figure 20.2: Two computer graphics created by drawing a pixel through tracing triangular shapes.





set of operations that interact directly with the hardware similar to assembly languages. At that time almost most computer graphics look similar because they use either one of these two available APIs. OpenGL in particular was not a programming language but rather a vendor specification of how the hardware interface should be implemented. On the other hand, DirectX offered by Microsoft was essentially a library that allowed development of graphics in computer games. The pipeline then evolved into a more advanced level and shading capabilities were added into the APIs. The two main vendors who were manufacturing GPUs were nVidia and ATI. In the next section, we shed more light on GPU's pipeline.

20.3 GPU Pipeline

Figure 20.3 shows how early GPUs pipeline looked like. The input to the pipeline was the description of graphics in memory in terms of vertices and triangular shapes. The input will be transformed using a vertex shader which process every vertex individually and creates rasterized pixels in 2D screen. The fragment shader will then take that and produce the final output.

Figure 20.3: Early GPU pipeline.

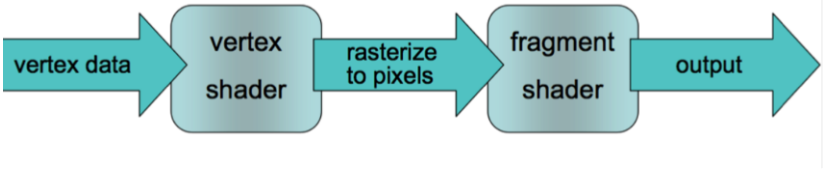


Figure 20.4: Improved GPU pipeline.

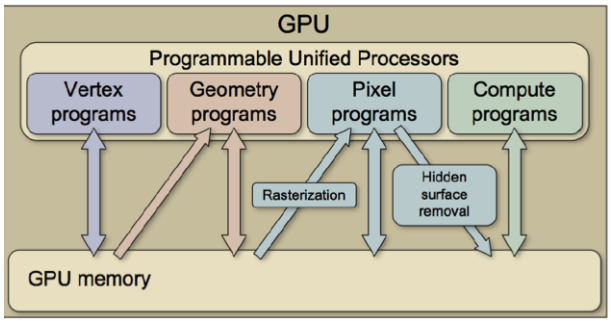


Figure 20.4 shows how the GPU pipeline looks now. Unlike early GPUs which used different hardware for different shaders, the new pipeline uses a unified hardware that combines all shaders under one core and shared memory. In addition, nVidia introduced CUDA (Compute Unified Device Architecture) which added the ability to write general-purpose C code with some restrictions. This meant that a programmer has access to thousands of cores which can be instructed to carry out similar operations in parallel.

20.4 GPU Hierarchy & Programming

In this section we will zoom out to look at the hierarchal structure of GPUs and give a simple example of performance measure. Top of the line GPUs now are claimed to support few teraflops and 100+ GB/sec of memory bandwidth internally. They are also easier to program through support of C++ (as in nVidia's Fermi architecture) and MATLAB integration. This is essentially why GPUs became very popular because it was possible now to carry out expensive computational task without the need for expensive hardware

cluster setup. Therefore, many modern applications such as Phtotoshop and Mathematica took advantage of GPUs. However, in order to fully take advantage of GPUs, the program must be **highly** parallel and contain fine-grain **simple** operations which can be distributed easily across the e.g. thousands of cores.

Figure 20.5 shows the hierarchy of a GPU. The bottom of diagram is the computer's system memory (RAM and disk) and then on top of that is GPU's internal memory. Then there are two levels caches: L2 and L1, and each L1 cache is connected to a processor known as *stream processor* (SM). The L1 cache has a small memory range of about 16 to 64 kBs. Similarly, L2 cache in the GPU is significantly smaller than typical L2 cache in computer's memory, with the former in the order of few hundred kBs and the latter few MBs. The trend is also the same at the memory level, with GPU's memory up to 12 GBs and computer's main memory ranging between 4 GBs up to 128 GBs. Furthermore, the GPU's memory is not coherent meaning there is no support for concurrent read and concurrent write (CRCW).

Figure 20.5: GPU Hierarchy

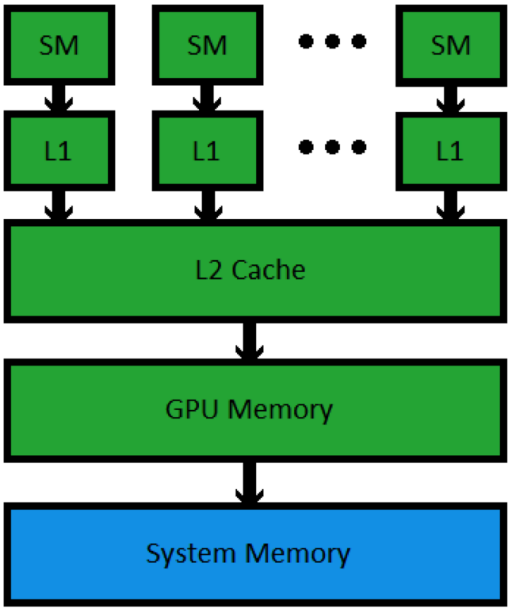


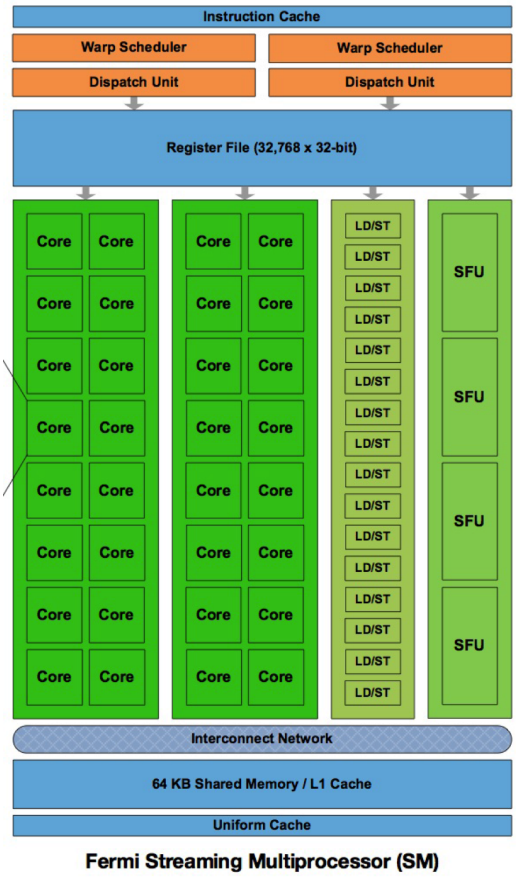
Figure 20.6 is showing an example structure of each of stream processors inside of GPU. As can be seen from the bottom of digram there is an L1 cache. There is also register memory (top of diagram) that is shared between the 32 cores, with each core has access to fixed set of registers.

There has been some excitement in the research community about GPU's 100-200x speed-up gains which often overlooked some aspects of the GPUs and which were driven mainly by unfair comparison to the general-purpose CPUs. Some of the key issues to consider when evaluating the gain in speed-up in the GPU in comparison to CPU are, for instance, whether similar optimized code was used on both GPU and CPU, and whether single precision or double precision numbers were used. The double precision is known to cause slow performance in GPU. Further, it is often the case that the transfer time between computer's memory to GPU's memory is neglected. As it turn out in some cases this can be nontrivial and it is a hardware issue that can not be handled by software. Taking this transfer time into account when measuring performance and making comparisons is essential.

To illustrate this final point of memory transfer time, we will now show a simple experiment of GPU performance measure using a Matlab program. Consider the following lines of Matlab code,

```
cpu_x = rand(1,100000000)*10*pi;
gpu_x = gpuArray(cpu_x);
```

Figure 20.6: GPU's Stream Processor.




```
gpu_y = sin(gpu_x);
cpu_y = gather(gpu_y);
```

The first line creates a large array data structure with hundreds of millions of decimal numbers. The second line loads this large array into GPU's memory. The third executes the sin function on each individual number of the array inside the GPU. And last line transfer the results back into computer's main memory. The plot in the top of figure 20.7 shows the runtime performance comparison as a result of executing this program with increasing input size. The middle line is the baseline performance of computer's CPU while the bottom line is GPU's runtime excluding memory transfer and the line at the top is GPU's runtime performance when taking memory transfer into account. Obviously, this shows that GPU's runtime performance is better by about a factor of 3 without considering transfer time and it is worse than CPU's runtime performance when transfer time is included. However, now consider changing the the third line in our Matlab program and instead use a more expensive function such as `big-trig-function` instead of `sin`. The plot at the bottom of figure 20.7 shows the resulting runtime performance. The top line is CPU performance while the two lines below are runtime performance including transfer time of two different brands of GPUs. This time the GPUs outperform the CPU by about a factor of 3 to 6 even when the transfer time is included. Therefore, one has to be careful when measuring GPUs speedup gains for certain tasks and not to overlook important aspects that may affect a fair comparison.

Figure 20.7: GPU runtime performance measure.