[PD-04-013] GPU Programming for GIS Applications

Abstract

Graphics processing units (GPUs) are massively parallel computing environments with applications in graphics and general purpose programming. This entry describes GPU hardware, application domains, and both graphics and general purpose programming languages.

Keywords: GPGPU, GPU, graphics programming, parallel programming

Author & citation

Mower, J. (2018). GPU Programming for GIS Applications. The Geographic Information Science & Technology Body of Knowledge (4th Quarter 2018 Edition), John P. Wilson (Ed.). DOI:10.22224/gistbok/2018.4.5

Explanation

1. Definitions
2. GPUs and Programming Paradigms
3. Parallel Processing Essentials
4. Programming the Rendering Pipeline
5. General Purpose Computing on Graphics Processing Units (GPGPU)
6. GPGPU Performance Characteristics
7. Cloud Computing Resources
8. Future Developments

1. Definitions

Graphics Processing Unit (GPU): a computer hardware module composed of processors, registers, and dedicated random access memory. A GPU may be integrated within a desktop, mobile, or cloud computing platform.

Rendering Pipeline: a highly-optimized parallel processing environment intended for rendering 3D vertex data to a pixel. Ideally, all vertices (and all pixels) are processed independently from one another on dedicated processors. In practice, GPU physical processors host groups of virtual processors.

Framebuffer: a collection of video random access memory (VRAM) locations containing numeric color values mapped to display regions.

General-purpose computing on graphics processing units (GPGPU): GPU programming that is not primarily directed toward graphic output.

Client-Server Processing: The processing model used for most GPU computing where a CPU (central processing unit) is the client that launches a program on a GPU acting as a

server. In some GPGPU environments this relationship may be described as host-device processing.

Texture: a 1- or 2-dimensional data structure whose elements specify color or generic numeric values at locations.

Shader Programming Language: a primarily graphics-oriented programming language (such as GLSL or HLSL) for code development on GPUs.

Shader Program (or Shader): a program written in a shader programming language for execution on a GPU and interaction with the rendering pipeline

Shader Stage: one of several programmable processing stages in the rendering pipeline. Each shader stage understands a unique programming language that varies slightly from others in the same language group.

Compute Shader Stage: a stage in a shader program that performs GPGPU-style computing without interacting with the rendering pipeline

Shader Instance: a single execution thread associated with a unique virtual processor of a given shader stage.

Uniform variable: a read-only variable with respect to a shader stage with a value supplied by the client at the start of the encompassing shader program's execution.

Shader Storage Buffer Object (SSBO): a client side data buffer bound to a server buffer, frequently used for data transfer between the client and server.

2. GPUs and Programming Paradigms

This article discusses modern graphics processing unit (GPU) programming for spatial data handling on desktop and mobile computing platforms. Modern GPUs employ massively-parallel computing architectures that, for many application domains, improve overall data throughput over sequential computing alternatives (see Graphics Processing Units (GPUs)). Evolving from early GPUs containing as few as 16 processors (Robert et al. 1986), modern high-end desktop systems now contain over 3,800 processors (or cores) (see NVIDIA TITAN Xp specifications as an example). GPU cloud computing services host a variety of programming languages and development environments that provide remote access to high-performance GPUs for building graphics and **general-purpose computing on graphics processing units (GPGPU)** applications. These include machine learning and image processing tasks as well as 3D rendering applications (Esri currently offers the ArcGIS Desktop Virtualization environment as a cloud GPU solution for mapping graphics applications). The physical location of the GPU is largely irrelevant to the developer—the same programming languages apply to desktop and cloud systems.

Both graphics and GPGPU programs are launched from a "client" process (for graphics applications) or a "host" process (for GPGPU applications). Figure 1 illustrates the data flow between client and server for a typical graphics application.

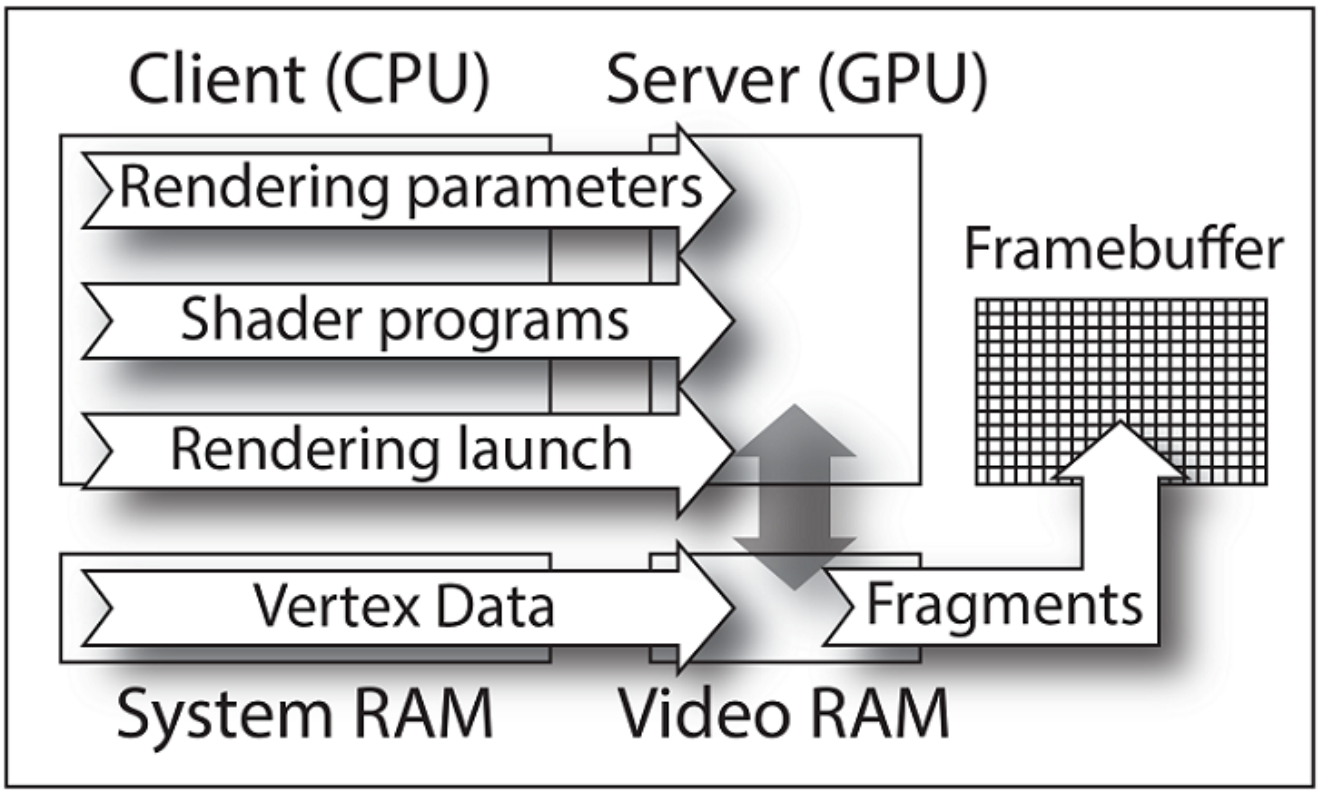


Figure 1. The data flow between a client process, a server process, and a framebuffer in a graphics program (from Mower 2016).

The client (or host) process, most often executed on a local computing resource with a conventional CPU, compiles and links a set of GPU source code files into a program, binds (copies) the compiled program to the GPU (the server or device), and initiates its execution. The programming languages understood by a given GPU depend on the development tools and definitions contained in its device driver. The specific manner of server/device program compilation, linking, and execution is language-dependent. Table 1 categorizes and describes some of the more popular client-server (host-device) programming languages.

Table 1. GPU Programming Environment Characteristics

Environment	Maintained By	Application Domain	Additional Supported Device / Server Languages
OpenGL	Khronos Group	Graphics	GLSL
DirectX	Microsoft	Graphics	HLSL
CUDA	NVIDIA	GPGPU	Not Applicable
OpenCL	Khronos Group	GPGPU	Not Applicable

Most GPU server programs operate on data supplied by the client in one or more ways. Graphics client-side programs typically copy a set of vertices (possibly interleaved with other data such as surface normals, texture coordinates, and so on) to the server-side program (usually referred to as a shader program) as an input stream for the 'rendering pipeline,' a highly-optimized series of processing stages that transform 3D vertex coordinates to color-valued pixels within a video **framebuffer**. Other data channels, such

as textures (mapped 1D or 2D image attribute data), shader storage buffer objects (similar to arrays of C struct types) and uniform variables (read-only scalar data copied from the client), can supply the server with an almost unlimited variety of attribute data. While some data channels are ‘read only’ with respect to the server, others allow modifications that are visible to the client on completion of a rendering pass. The price for visibility is the sometimes substantial cost in the time required to copy data from the server back to the client. Programs in the "general-purpose computing on graphics processing units" or GPGPU category apply similar mechanisms to the transfer of data between the client and server (or host and device) but rarely interact with the rendering pipeline.

3. Parallel Processing Essentials

The efficiency of a parallel computing solution is largely a function of the proportion of processors that are kept busy working on a problem at any given time. Some types of problems, known as ‘embarrassingly parallel,’ require little synchronization through communication with other processors. For example, if every pixel in a framebuffer were represented by a unique processor, any one of them could query its current color value and swap it by consulting a lookup table. No communication with other pixels would be required. At the other extreme, ‘inherently sequential’ problems such as drainage accumulation modeling over a DEM require continuous flow updates through extensive interprocessor communication. When multiple processors need to reference a common memory location (as would two uphill vertices in a drainage network that drain to a third common downhill vertex), the programmer must somehow ensure that each uphill vertex sees up-to-date data in the common downhill vertex before adding its flow to its downhill neighbor (Figure 2).

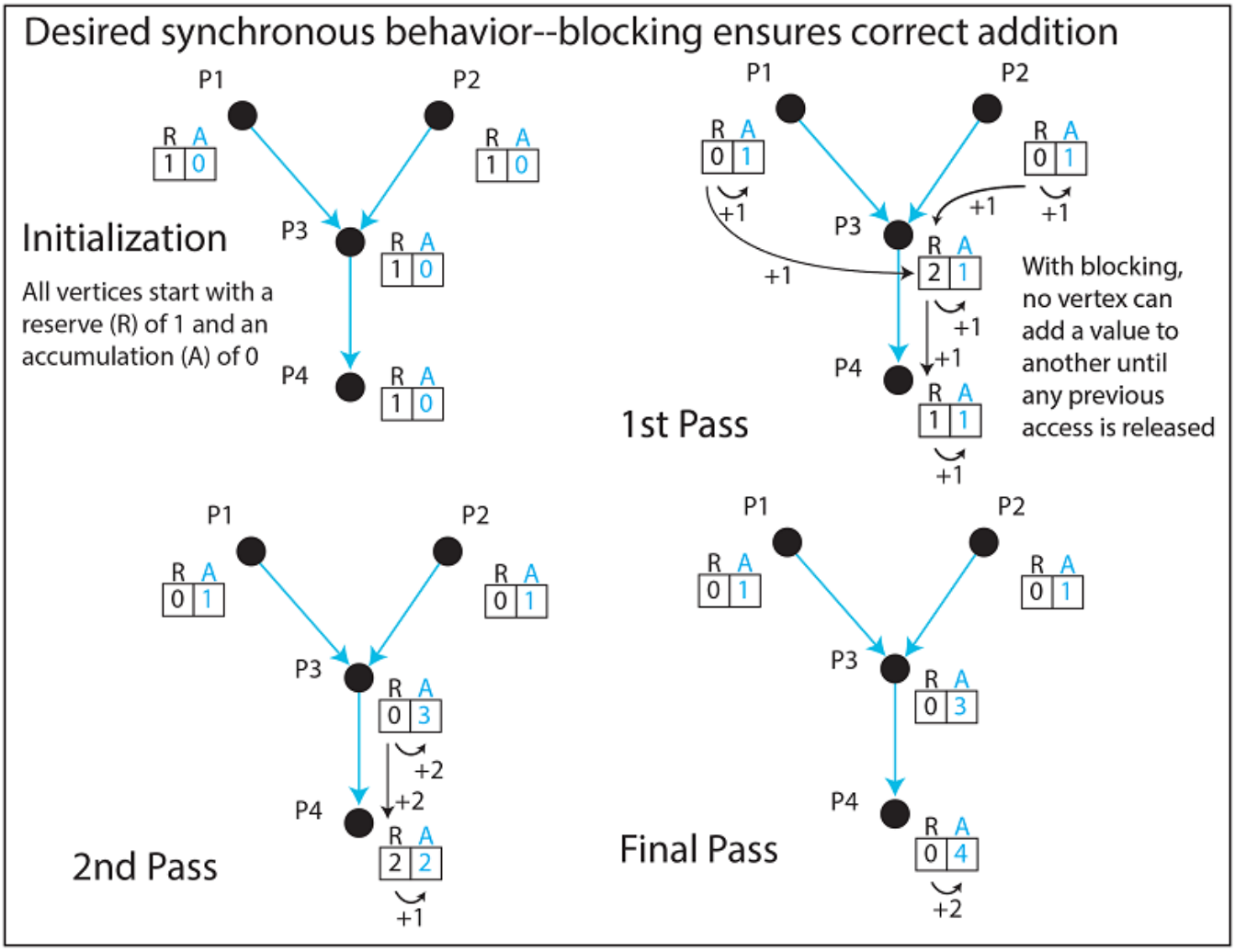


Figure 2. A portion of a drainage network. Blue lines indicate direction of flow. Processors P1 through P4 are initialized with a reserve value R of 1 and accumulation value A of 0. On each pass, a processor first adds its R value to its A value, then adds its R value to its downhill neighbor's R value, and then zeroes its own R value. Synchronous (atomic) addition ensures that every processor sees up-to-date values at other processors.

This can be achieved by allowing one uphill processor to "lock" the common downhill memory location until it has finished adding its contents through an "atomic" operation. This synchronization prevents the other uphill processor from adding its contents to a stale (invalid) value. Blocked processors remain idle until the blocking condition is cleared. Not doing so would likely produce undefined results (Figure 3). Synchronization usually implies a loss of efficiency as the number of simultaneously active processors is reduced.

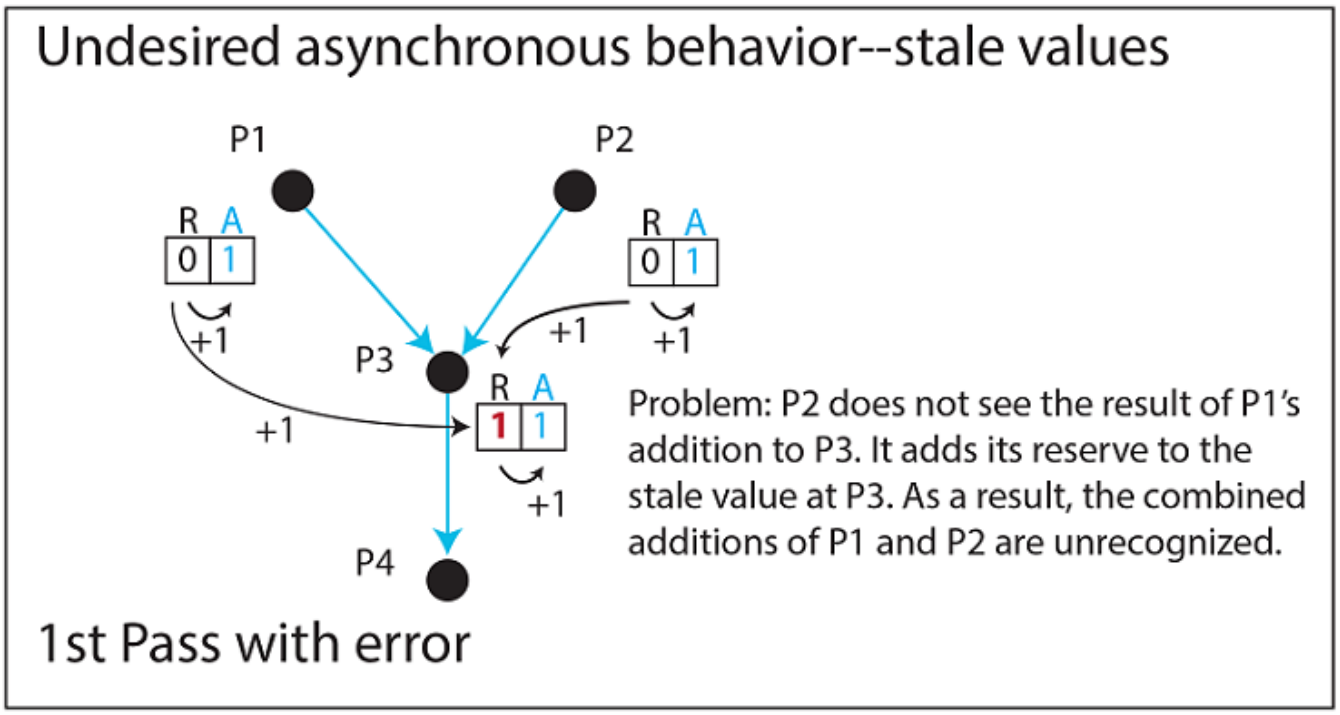


Figure 3. An error due to asynchronous addition. P1 adds its reserve value of +1 to the current 0 value at P3. P2 does not see the posted value at P3 resulting from P1's addition, instead observing the same 0 value that P1 saw (a stale value with respect to P2). P2 adds its reserve value of +1 to the stale value, and replaces R at P3 with 1 instead of the correct combined value of 2.

Most GPU environments provide a variation of single-instruction, multiple data (SIMD) computing in which a subset of processors (sometimes referred to as a workgroup) share a single, common processing thread executed in lockstep over each member processor's local data. Processors within a given workgroup run asynchronously with respect to those in other workgroups, implying a single-program, multiple data (SPMD) programming model.

The performance advantage of a parallel program over a functionally-equivalent sequential program is referred to as its "speedup factor." If an array of 100 cells requires 100 units of time to process sequentially, the same array might be expected to require 1 unit of time on a GPU if 100 available physical processors, each hosting an individual array cell, were able to run its task in 1 time unit, completely asynchronously with respect to all other processors. Although this potential 100x speedup is unlikely to be attained on real systems, it can provide an upper-end estimate of performance expectations (see section 6, GPGPU Performance Characteristics below for more realistic speedup comparisons).

A GPU can host a variety of graphic and GPGPU processing models. Graphics programming interfaces like OpenGL and DirectX are primarily intended for building rendering applications that require framebuffers to be filled at animation rates (at least 24 complete framebuffer renderings per second) to limit visible pauses in motion. Both interfaces have their own sets of server-side shader languages (GLSL for OpenGL; HLSL for DirectX) that optimize vertex processing by keeping inter-processor communication to a bare minimum. GPGPU languages like CUDA that do not generally interact with the rendering pipeline and

therefore are not subject to video refresh constraints tend to provide a broader range of utilities for asynchronous processor control flow and inter-processor synchronization. The default behaviors of GLSL and HLSL limit synchronization options to promote high frame rates during rendering pipeline operations.

It is possible to execute GPGPU programs written in CUDA and graphics programs written in either OpenGL or DirectX from the same client program (see Stam 2009, What Every CUDA Programmer Should Know About OpenGL). Alternatively, both OpenGL and DirectX provide a compute shading stage that allows GPGPU computing without interacting with the rendering pipeline and that provides synchronization utilities approaching the levels of control supplied by pure GPGPU languages such as CUDA and OpenCL (see section 5 below, and see Khronos Group (2018), Core GLSL, and Microsoft (2018), HLSL). If the GPGPU tasks do not require elaborate blocking techniques, then the use of a compute shader before or after a rendering pipeline pass may be a viable alternative to cross linking a graphics-dominant application with CUDA.

4. Programming the Rendering Pipeline

Shaders are usually written to transform a set of 3D vertices to their 2D framebuffer counterparts such that each execution pass of the shader program creates a single image (or a single frame in an animation sequence). Figure 4 illustrates the rendering pipeline from the point of view of an OpenGL program (DirectX uses a very similar model).

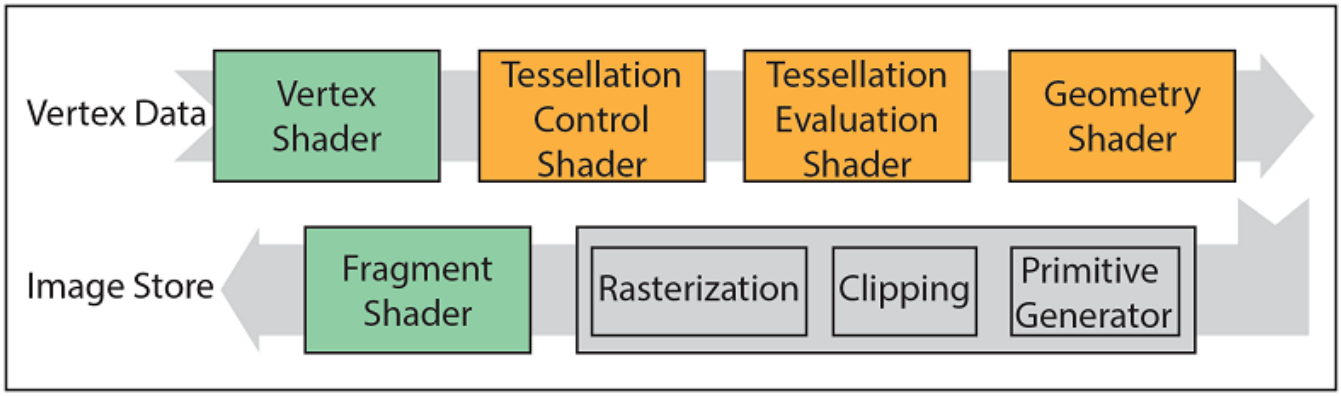


Figure 4. The OpenGL rendering pipeline. Green indicates mandatory user programmable stages, orange stages are optional, and gray stages are both mandatory and non-user-programmable (From Mower 2016).

OpenGL and DirectX client-side programs can be written in any standard programming language but C and C++ are generally preferred for the ease with which they link to the respective graphics library functions. GLSL and HLSL are somewhat different from one another in their keywords and syntax, but both inherit many of the conventions of the C programming language and perform similar graphics operations. GLSL and HLSL are actually groups of languages; different rendering stages require slightly different programming languages that may vary on the inclusion of stage-appropriate functions. For brevity, the remaining discussion pertains specifically to GLSL and OpenGL but is generally

applicable to HLSL and DirectX at a pseudocode level.

OpenGL, currently maintained and licensed by the Khronos Group, is available to end users without licensing requirements. Derived as a functional alternative to IrisGL (the proprietary rendering interface introduced by Silicon Graphics in the 1980s), OpenGL 1.0, introduced in 1992, invoked GPU rendering pipeline functionality through a fixed-function interface in which users were limited to modifying GPU rendering parameters through pre-defined OpenGL functions (Khronos Group 2018, History of OpenGL). The first programmable pipeline version of OpenGL became available in 2004 with the introduction of the GLSL family of shading languages. Early versions limited direct user access to the vertex and fragment stages of the programmable pipeline; by version 4.0 (introduced in 2010), all of the programmable stages in Figure 4 had been made available. Subsequent versions of OpenGL (and GLSL) have continued to increase access to functionality at each stage. Access is increased as manufacturers and language implementers "expose" additional features of the graphics hardware that were previously inaccessible to GPU programmers.

Client-side OpenGL programs are responsible for setting up the overall graphics environment, establishing a frame-by-frame rendering loop (for anything from a single frame to an animation sequence), and creating memory buffers for transferring data to and from the server. Library packages such as glut and glew standardize many of the video setup and display loop tasks across operating systems, making cross-OS development relatively straightforward.

Shader programs must include at least 2 mandatory stages: a vertex stage that typically transforms world or model coordinates to projected coordinates, and a fragment stage that determines the color values for fragments that in turn define mapped pixel values within a framebuffer. Referring back to Figure 4, the assembly of primitives (points, lines, triangles, and others) are performed by the non-programmable stages (in gray).

Each vertex shader instance transforms a single vertex in the client-side input stream independently from all other vertices. Each instance represents a unique execution thread hosted on a GPU processing core. A core may host more than one thread but threads do not communicate with one another, thereby eliminating any need for synchronization and any resulting decrease in performance.

The transformed vertices emitted from the vertex stage may be passed to optional tessellation and geometry stages depending upon the needs of the application. Both optional stages allow additional vertices to be added to the **rendering pipeline** beyond those supplied by the client. Figure 5 illustrates the use of tessellation shaders to create a Bezier patch. See Vlachos and others (2001) for more details.

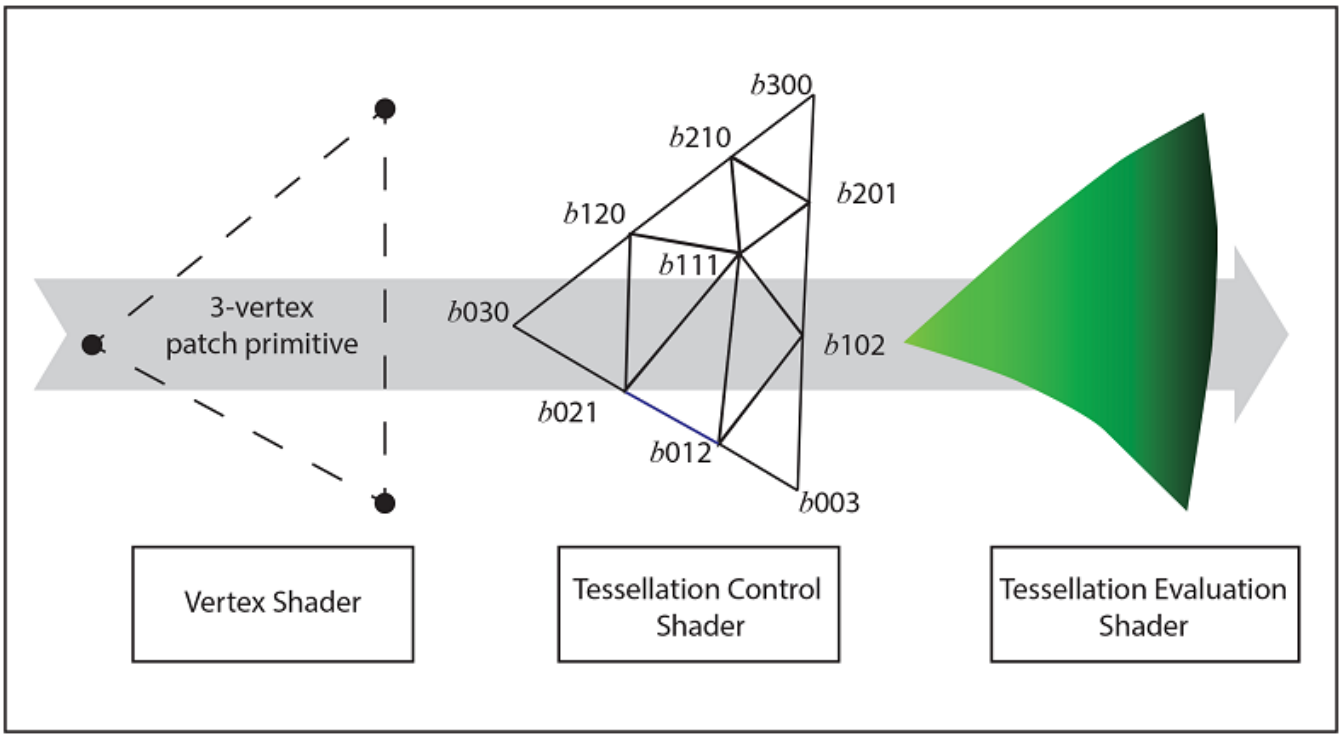


Figure 5. Making a Bezier patch. The tessellation control shader (TCS) groups 3 vertices from the previous vertex stage to form an abstract patch. In this example, the labeled control points of the Bezier surface (defined in the TCS) determine the evaluation of intermediate points (at any desired resolution) calculated within the tessellation evaluation shader (TES) stage.

The mandatory, non-programmable primitive generator takes its input (as a collection of vertices representing a point, line, or shape) from the preceding vertex, tessellation, or geometry stage and constructs the selected type of graphic object (primitive). Primitives are passed to the clipping stage which rejects any vertices that fall outside the viewing volume, defined by the user's viewing parameters and represented as a truncated pyramid (Figure 6). The following rasterizing stage converts the remaining vertices to screen coordinates and produces a rectangular tessellation of the primitive into fragments. Each resulting fragment has a unique location in the frame buffer specified by a UV coordinate pair in the range 0 to 1. Following rasterization, each fragment is represented by its own fragment shader instance which is primarily responsible for setting the color value of its associated pixel in the framebuffer. OpenGL and DirectX environments allow the creation of multiple framebuffers, one of which is for immediate display while the others serve as hosts for storing images as textures.

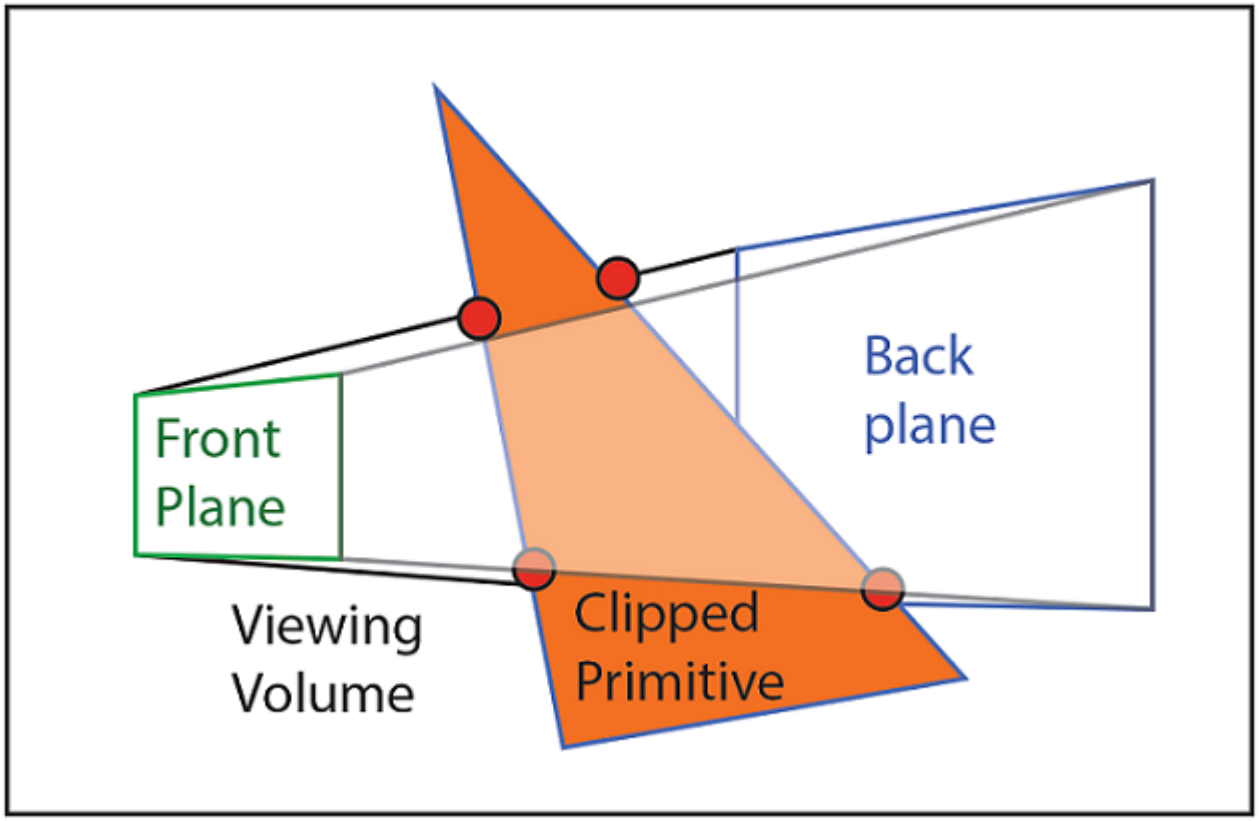


Figure 6. The clipping stage. In OpenGL, a triangle primitive with vertices falling outside the truncated viewing pyramid is clipped to its boundaries by interpolating new vertices (in red) at the intersection of the primitive and the bounding planes. Vertices outside the viewing pyramid are rejected.

A single client-side application may choose to execute more than one rendering pass to build up complex image products. Each preliminary pass uses a unique shader program that ultimately writes its output to a texture (or some other buffer) that is accessible to subsequent passes. The fragment shader of the final pass typically writes its output to the on-screen framebuffer for display.

OpenGL and DirectX both provide optional **compute shaders** that have no direct access to the rendering pipeline. Compute shaders are best used for performing non-graphic transformations of data where little inter-processor data sharing is required. If an application is not required to create any graphic output, then another language with finer control flow (such as CUDA) is often preferable.

Many excellent tutorials on GPU graphics programming are available. The author has made extensive use of the OGLdev Modern OpenGL Tutorials. Such tutorials provide essential information on coding, compilation, linking, and execution for a variety of client programming languages and operating systems.

5. General Purpose Computing on Graphics Processing Units (GPGPU)

GIS and remote sensing applications such as image stretching, coordinate transformations, image warping, and many others can benefit from the data throughput available through GPGPU programming. GPGPU programs, like shader programs, typically associate an individual data element with a unique virtual processor. See Christophe, Michel, and Inglada (2011) for a history of parallel solutions for remote sensing applications ranging from multi-core CPU implementations through early GPU solutions.

Table 2 lists a selection of the most common proprietary and open source GPGPU programming languages. Languages such as CUDA, developed and maintained as a proprietary product of NVIDIA, and OpenCL, an open-source language overseen by the Khronos Group, share many GPU programming methods and data structures. Due to the limitation of CUDA applications to NVIDIA hardware, OpenCL is more commonly applied for cross-platform development. However, OpenCL programs may require more manual optimization to achieve an equivalent level of performance to a functionally equivalent CUDA program. CUDA and OpenCL library bindings exist for many standard programming languages including C, C++, and Python. DirectCompute shares many characteristics of OpenCL but is proprietary and limited to execution within Microsoft Windows DirectX media environments.

Table 2. Common GPGPU Programming Languages

Language	Open Source	Developer	Characteristics and Use
CUDA	No	NVIDIA	Direct GPU programming and backend GPU processing through CPU applications (limited to NVIDIA hardware)
OpenCL	Yes	Khronos Group	Cross-platform GPGPU standard
DirectCompute	No	Microsoft	GPGPU programming for Microsoft Windows DirectX environments
OpenACC	Yes	Cray, CAPS Enterprise, NVIDIA, PGI	Standard for mixed CPU/GPU parallel programming

CUDA, OpenCL, and DirectCompute programs begin execution on a host CPU and generally make data available to the GPU (the device) by using functions that create a memory space visible to both the host and the GPU. CUDA programs must additionally specify the number of threads that will run in parallel to solve the given problem and add a qualifying tag before a function definition indicating whether it should be allowed to run in parallel on the GPU. Simple CUDA applications benefit from their strong resemblance to comparable sequential applications with the addition of a relatively small number of keywords and functions. For example, the functions in Table 3 have familiar C++ counterparts:

Table 3. CUDA and C++ Cognate Functions

CUDA Library Function	Operation	Comparable C++ Function
cudaMemcpy()	Copy data between host and device	memcpy()
cudaFree ()	Free device memory	free()
cudaBindTexture()	Bind a memory area to a texture	glBindTexture() // OpenGl library // function

CUDA also provides a number of vector data types that would be familiar to those who have used MATLAB or a modern graphics library. Some examples are float2, float3, and float4 for 2D, 3D, and 4D vector types with attributes x, y for float2, x, y, z, for float3 and x, y, z, w for float4. Matrix algebra operations compatible with these vector types (MatAdd(), MatMul(), etc) are similarly provided.

See NVIDIA 2018, GPU Accelerated Computing with C and C++ for explicit instructions for building and running sample CUDA applications, and The Khronos Group 2015, An Introduction to OpenCL C++ for similar information and sample code for OpenCL.

Many programmers will choose to use CUDA or another GPGPU language indirectly through an application such as MATLAB. Under this approach, the developer can continue to use a familiar programming interface without having to learn the details of GPU program execution. Many machine learning projects written for MATLAB use this approach.

OpenACC, introduced as a standard in 2012 and developed jointly by Cray, CAPS Enterprise, NVIDIA, and Portland Group (PGI), provides higher-level parallel processing solutions than do CUDA, OpenCL, DirectCompute, and other languages supporting manual coding of GPGPU programs. Promoted as a parallel programming environment requiring limited 'hands on' GPU coding, OpenACC programs are written in C, C++, or FORTRAN and contain embedded annotations tagging sections of code to be run on available multi-core CPU and GPU resources where present. As of August, 2018, the open-source gcc compiler supports OpenACC but other common development environments, notably Microsoft Visual Studio, do not.

6. GPGPU Performance Characteristics

The speedup factor of a parallel computing solution over a comparable sequential solution depends on many factors, the most important of which are listed here:

- The degree of optimization for the compared sequential and parallel solutions
- The number and performance characteristics of the tested CPU cores
- The number and performance characteristics of the tested physical GPU cores
- The amount of available RAM and VRAM
- The data transfer speeds between GPU and CPU resources

Similarly, the relative speeds at which 2 functionally comparable parallel implementations run when written in 2 different languages depend largely upon the efficiency of the coded parallel programming solutions and the resulting compiled code. Gregg and Hazelwood (2011) advocate for controlled benchmarking in comparing GPU performance factors. Showing that in some cases data transfer elapsed times between the GPU and CPU can exceed 50x the GPU execution time, the authors provide a comparison framework accounting for all CPU and GPU program resources. Without such controls, speed comparisons between parallel implementations or between parallel solutions and sequential solutions are at best anecdotal.

Nonetheless, good examples of sufficiently controlled measurements of speedup values for GPGPU solutions can be found in the literature. For example, Moustafa and others (2016) report GPU speedup values of 12x to 70x for an image processing task that varied by the

number of threads allocated to each image. Clearly, well-written GPGPU programs have the capacity for attaining large speedup values over comparable sequential solutions.

7. Cloud Computing Resources

Cloud computing services are rapidly becoming preferred platforms for GPGPU applications. NVIDIA Nimbix, Amazon AWS, IBM Cloud, and Microsoft Azure all provide GPGPU environments for CUDA as well as more limited support for OpenCL, OpenGL, and DirectX. Vendors typically charge hourly rates for compute time. Of these services, NVIDIA Nimbix provides service interfaces that most resemble client-server programming tasks in a desktop environment with an attached GPU. Many other services provided by these vendors make implicit use of GPU computing in the background.

8. Future Developments

As high-performance GPUs in desktop, mobile, and cloud computing environments become more pervasive, it is reasonable to expect that an increasing number of graphics applications in cartography and GIS, and compute-intensive problems in remote sensing will see increases in the number of solutions that invoke GPU platform implementations. This will likely increase the range and complexity of available augmented reality and animated cartographic products and to facilitate projects in GIS and remotes sensing that operate over large spatial data collections.

References

Christophe, E., Michel J., Inglada, J. (2011). Remote Sensing Processing: From Multicore to GPU. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 4(3):643-652.

Gregg, C., and Hazelwood, K. (2011). Where is the data? Why you cannot debate CPU vs. GPU performance without the answer, (IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software, Austin, TX, 2011, pp. 134-144.

Moustafa, M., Ebeid, H., Helmy, A., Nazmy, T. M., and Tolba, M. F. (2016). Rapid Real-time Generation of Super-resolution Hyperspectral Images through Compressive Sensing and GPU. International Journal of Remote Sensing, 37(18): 4201-4224.

Mower, J. E. (2016) Real-Time Drainage Basin Modeling Using Concurrent Compute Shaders. Proceedings, AutoCarto 2016, Albuquerque, September 2016 pp. 113-127.

Robert, F., Hopgood, A., Hubbold, R. J., Duce, D. A., Eds. (1986). Advances in Computer Graphics II. Berlin: Springer Verlag.

Vlachos, A., Peters, J., Boyd, C., and Mitchell, J. L. (2001). Curved PN triangles. AI3D '01: Proceedings of the 2001 Symposium on Interactive 3D Graphics, pp. 159–166.