Hardware and Compute Abstraction Layers For Accelerated Computing Using Graphics Hardware and Conventional CPUs

Justin Hensley
justin.hensley@amd.com
Graphics Product Group, Advanced Micro Devices, Inc.

Introduction

Graphics processors (GPUs) make an attractive platform for numerically-intensive calculation because of their inexpensive high-performance SIMD arrays originally designed to shade pixel fragments. These arrays consist of up to several dozen processors connected to a high-bandwidth (greater than 100 GB/sec on current hardware), latency-hiding memory system [1]. This combination can outperform CPUs, sometimes by more than an order of magnitude, on numerically-intense applications. Examples of algorithms that have been implemented with success on GPUs range from collision detection and response to medical image segmentation to fluid simulation.

Unfortunately, such implementations rely on either OpenGL [2] or Direct3D [3] to access the hardware. OpenGL and Direct3D contain many elements irrelevant to GPU computation, and failure to ensure that such elements are set properly can lead to confusion program errors. More significant is that these APIs, by design, hide architectural details that are of importance to GPU programmers. As part of this hiding, graphics drivers make policy decisions, such as where data resides in memory and when they are copied, that may dramatically reduce the performance of non-graphics application, thereby undermining the motivation to implement a algorithm on a GPU in the first place.

Recently, Nvidia has released CUDA (Compute Unified Device Architecture) [4]. While CUDA relieves developers of so-called GPGPU (general purpose computation using graphics hardware) applications of dealing with extraneous, graphics-centric settings, CUDA still hides the underly hardware and does not allow the application developer to make low-level optimizations which can prevent an application from achieving peak performance from the hardware. Also, CUDA is focused solely on graphics hardware.

The following two sections briefly describe two of our APIs for accelerated computing: HAL - Hardware Abstraction Layer, and CAL - Compute Abstraction Layer. Both of which were developed with the lessons learned during the development CTM [5]. Figure 1 shows the relationship of HAL and CAL to the application software stack. Finally, some preliminary results are presented from current research to accelerate AES and DES key searches using graphics hardware.

HAL - Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) exposes the graphics hardware directly to the programmer. While it allows for applications to squeeze as much performance out of the hardware as possible, portions of HAL are device specific, and applications that directly use HAL are not necessary forward compatible: GPUs in the past have had the luxury of being exposed to programmers via a high-level API, as such each new generation of GPUs can be dramatically different from previous generation. For example, AMD's X1K family (R5XX) of graphics processors has a completely different instruction set architecture from AMD's HD2K family (R6XX).

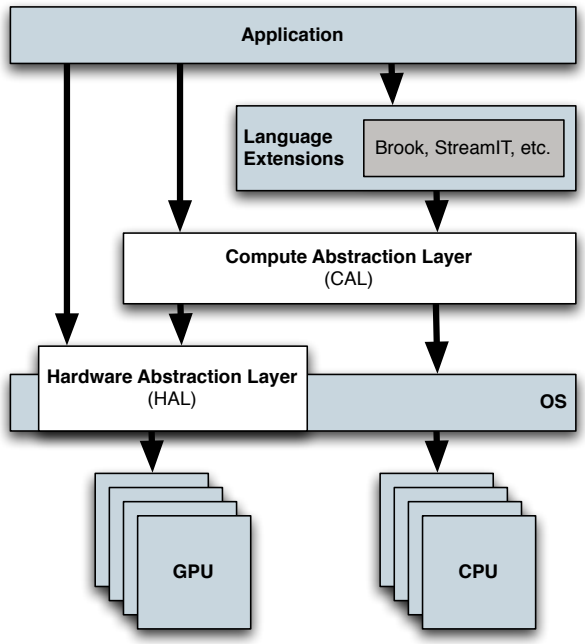

Figure 1: The compute abstraction layer (CAL) hardware abstraction layer (HAL) show in relation to the application programming stack.

Abstractly, a HAL exposes the the five major components of a GPU: a command processor, a memory controller, *render back-ends* (sometimes referred to as ROPs), texture (memory) address logic, and a data parallel processor array.

The command processor accepts commands from applications packed into a *command buffer*. These commands are device specific and include program code for the processor array, cache control and invalidation, input and output locations, et cetera. Additionally, the command processor is responsible for scheduling the processor array.

The render back-ends are dedicated hardware typically designed to handle tasks such as z-buffering, bending, and stencil tests. An example z-buffer mode would only write incoming fragments to memory if they are smaller than the value already stored at specific location.

The address logic takes care of converting 2D and 3D locations in to memory addresses. Additionally, the texture address units typically contain filtering hardware to perform bi-linear and tri-linear interpolation. Typically GPUs have multiple addressing modes besides a simple linear addressing; *tiled* modes are included as well to increase texture cache efficiency with graphics applications.

The data parallel processing array (fragment processors in graphics hardware parlance) is the computational heart of modern graphics processors. It can access some number of inputs (textures), and some number of outputs (render targets and z-buffer). Beyond this, the details of the processing capabilities of the processor array are machine specific. HAL specifies an application binary interface (ABI) that exposes the native instruction set of the processor arrays.

Exposing the native instruction set of the processor array brings several benefits. First, once a program is compiled (or directly hand-coded in assembly) it is immune to compiler changes resulting from driver updates that might affect performance (or even correctness). Further, access to machine language simplifies performance tuning when a compiler fails to produce code of the desired efficiency.

Finally, since HAL gives the programmer direct access to the hardware, it facilitates the development of compute libraries and novel compilers. Additionally, it is the only API that we are aware of that allows applications to have low-level access to graphics hardware. The has great potential benefits for high-performance embedded systems.

CAL - Compute Abstraction Layer

At a higher level, the Compute Abstraction Layer (CAL) is designed to provide a forward-compatible, interface to the high-performance, floating-point, parallel processor arrays found in graphics hardware *and* CPUs. The computational model of CAL is processor independent and allows the user to easily switch from directing a computation from GPU to CPU or vice versa. CAL was designed with the following goals in mind:

1. Ease of programming
2. Optimized multi-GPUs **and** multi-core CPU support
3. An easy environment to develop compute kernels
4. Multiple OS platforms such as Windows XP, Windows Vista, Linux (32-bit and 64-bit on all)

A CAL system is comprised of at least one CPU device and zero or more GPU devices. Both CPU and GPU devices can communicate with its own memory subsystems. A single memory subsystem is attached to the all CPU devices and is also known as *system memory*. The memory subsystem attached to each GPU device is known as that GPU's *local memory*. CPU devices can read and write to the system memory (current bandwidths are approximately 8 to 10 GB/s). Each GPU device can read and write to its own local memory (current GPU bandwidths are greater than 100 GB/s). PCI Express, allows CPU devices to read and write to the GPU device memory and the GPU device to read and write the system memory. Additionally, CAL has support for a given GPU in a multi-GPU system to read and write to another GPU's device memory, also known as peer-to-peer transfer.

An important feature of CAL, is that the application developer is given the flexibility to perform unique optimizations; unlike API's such as OpenGL, DirectX, and CUDA, CAL exposes the GPU's ability to read and write from *system memory*. Since the application developer is not forced to copy data from system memory to local GPU memory, he or she is free to make optimizations where reading or writing to system memory can yield performance benefits. For example, it is possible to double performance when applying several image processing filters (e.g. bayer-pattern to RGB conversion, image warping, etc) on captured video frames that will be further processed on the CPU by reading from and writing to system memory. Alternatively, the developer is also given access to a DMA engine that can be used to asynchronously transfer data across different memory systems when available.

CAL compute kernels can be developed using a variety methods and languages, and includes an extensible compiler interface for new compilers to be added by third parties. Currently, CAL provides support for using HLSL and GLSL, both of which are high level *shading* languages. Additionally, an internally developed architecture-agnostic assembly language is supported. Finally, the Brook [6] streaming extension to the C-language are natively supported.

DES and AES key search

The presented results are for current and ongoing research. One example application of using AMD's accelerated computing API's is performing DES and AES key searches. On a single HD 2900 XT, we able to test keys at a rate of approximately 40 Gbps. Note, this rate is approximately 70 times that of a single CPU. To perform a brute force search on a single GPU an average of 2^{55} keys would have to be tested, which would take approximately 1.79 years. Using a parallel cluster of 1000 GPUs, the same brute force search could be performed in approximately 16 hours. With AES, our current implementation tests keys at a rate of 21 Gbps. One obvious extension is to use a directed attack instead of simply performing a brute force search.

References

[1] ATI Research, Inc, *The Radeon X1x00 Programming Guide,* www.ati.com, 2005.

[2] M. Segal and K. Akeley, *The OpenGL Graphics System: A Specification, Version 2.0.* www.opengl.org, 2004.

[3] Microsoft, Inc, *Direct3D Reference,* msdn.microsoft.org, 2004.

[4] Nvidia, Inc, CUDA, www.nvidia.com, 2007.

[5] ATI Research, Inc, *CTM Programming Guide, research.ati.com*, 2007.

[6] I. Buck, et al. "Brook for GPUs". In *Proceedings of SIGGRAPH '04* (2004), pp. 777-786