Basics of OpenCL Programming Model

MINSK 2015

Table of Contents

Abstract

This paper presents an overview of the OpenCL 1.1 (Open Computing Language) standard [Khronos 2012]. Author motivates the need for GPGPU computing and then discusses the various concepts and technological back ground necessary to understand the programming model. Also introduces participants to the fundamentals of the OpenCL programming language. Areas covered include the following:

- Introduction to GPU computing.
- Description of parallel computing with heterogeneous systems.
- Description of the OpenCL programming framework.

Introduction

In recent years performance scaling for general purpose CPUs has failed to increase as predicted by Gordon Moore in the early 1970s [Sutter 2005], and therefore raw throughput for sequential code has plateaued between subsequent processor generations. As a result of the issues of working with deep sub-micron lithography1, the primary motivation for moving to new processing nodes is less about increased performance or efficiency, but rather economic costs (for instance decreased cost per transistor and larger wafer sizes). While we have seen modest performance increases from the latest microprocessor architectures from Intel and AMD, these certainly haven‘t resulted in the doubling of performance that computer scientists relied upon from the 70‘s through to the early 2000‘s, in order to see performance increases in their code without changing the top level execution model.

Motivated by this lack of performance scaling, Graphics Processing Unit (GPU) manufacturers have opened up low level hardware traditionally used for graphics rendering only, in order to perform highly parallel computation tasks on what they call General Purpose GPU cores (GPGPU). While low level GPGPU execution cores lack branch prediction and out of order execution hardware that allow traditional superscalar CPU architectures to optimize sequential code, moving computation to the GPU trades off flexibility in execution models for raw performance. More-recently, CUDA2 and OpenCL3 are two frameworks that have seen significant traction and adoption by third-party software developers. This paper will focus on OpenCL (specifically version 1.1 of the specification) since it is an open cross-platform & cross-vendor standard. The paper is not a thorough investigation into the OpenCL standard (which is itself a massive body of

work), but is an overview of the programming methodologies one should be aware of when considering writing GPGPU code.

1 What is GPU Compute?

GPU compute is the use of Graphics Processing Units (GPUs) for general purpose computation instead of traditional graphics rendering. GPUs are high performance multicore processors that can be used to accelerate a wide variety of applications using parallel computing. The highly programmable nature of GPUs makes them useful as high-speed coprocessors that perform beyond their intended graphics capabilities.

1.1 A brief history of GPU compute

The birth of the GPU compute revolution occurred in November 2006, when AMD introduced its Close to Metal (CTM) low-level hardware programming interface that allowed developers to take advantage of the native instruction set and memory of modern GPUs for general-purpose computation. CTM provided developers with the low-level, deterministic, and repeatable access to hardware necessary to develop essential tools such as compilers, debuggers, math libraries, and application platforms.

With the introduction of CTM, a new class of applications was realized. For example, a GPU-accelerated client for Folding@Home was created that was capable of achieving a 20- to 30-fold speed increase over its predecessor (for more on this story, see http://folding.stanford.edu/English/FAQ-ATI#ntoc5.) AMD's continuing commitment to provide a fully-featured software development environment that exposes the power of AMD GPUs resulted in the introduction of the ATI Stream SDK v1 in December 2007. The ATI Stream SDK v1 added a new high-level language, called ATI Brook+. CTM evolved into ATI CAL (Compute Abstraction Layer), the supporting API layer for Brook+. The introduction of the ATI Stream SDK v1 meant that AMD was able to provide both high-level and low-level tools for general-purpose access to AMD GPU hardware.

A drawback to using the ATI Stream SDK v1 was that applications created with the SDK ran only on AMD GPU hardware. To achieve greater adoption for general-purpose computing, an open standard was needed. In June 2008, AMD, along with various industry players in GPU compute and other accelerator technologies, formed the OpenCL working group under The Khronos Group. Khronos, already known for leadership in other open specifications such as OpenGL®, was a logical choice to drive the OpenCL specification. Five months later, the group completed the technical specification for OpenCL 1.0 and

released it to the public. Immediately after that release, AMD announced its intent to adopt the OpenCL programming standard and integrate a compliant compiler and runtime into its free ATI Stream SDK v2. In December 2009, AMD released the ATI Stream SDK v2.0 with OpenCL 1.0 support.

1.2 Heterogeneous computing

Heterogeneous computing involves the use of a various types of computational units. A computation unit can be a general-purpose processing unit (such as a CPU), a graphics processing unit (such as a GPU), or a special-purpose processing unit (such as digital signal processor, or DSP). In the past, most computer applications were able to scale with advances in CPU technologies. With modern computer applications requiring interactions with various systems (such as audio/video systems, networked applications, etc.) even the advances in CPU technology proved insufficient to cope with this need. To achieve greater performance gains, specialized hardware was required, making the system heterogeneous. The addition of various types of computation units in these heterogeneous systems allows application designers to select the most suitable one on which to perform tasks.

2 The OpenCL Standard

2.1 OpenCL architecture and hardware

OpenCL is a programming framework and standard set from Khronos, for heterogeneous parallel computing on cross-vendor and cross-platform hardware. It provides a top level abstraction for low level hardware routines as well as consistent memory and execution models for dealing with massively-parallel code execution. The advantage of this abstraction layer is the ability to scale code from simple embedded micro-controllers to general purpose CPUs from Intel and AMD, up to massively-parallel GPGPU hardware pipelines, all without reworking code. While the OpenCL standard allows OpenCL code to execute on CPU devices, this paper will focus specifically on using OpenCL with Nvidia and ATI graphics cards as this represents (in the authors opinion) the pinnacle of consumer-level high-performance computing in terms of raw FLOPS throughput, and has significant potential for accelerating “suitable” parallel algorithms.

Figure 1 shows an overview of the OpenCL architecture. One CPU-based “Host” controls multiple “Compute Devices” (for instance CPUs & GPUs are different compute

devices). Each of these coarse grained compute devices consists of multiple “Compute Units” (akin to execution units & arithmetic processing unit groups on multi-core CPUs - think “cores”) and within these are multiple “Processing Elements”. At the lowest level, these processing elements all execute OpenCL “Kernels” (more on this later).

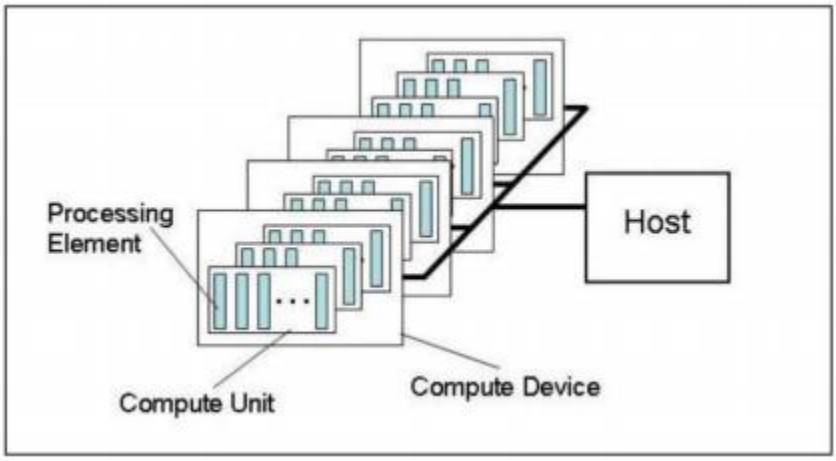

Figure 1: OpenCL Platform Model (from [Khronos 2011])

The specific definition of compute units is different depending on the hardware vendor. In AMD hardware, each compute unit contains numerous “stream cores” (or sometimes called SIMD Engines) which then contain individual processing elements. The stream cores are each executing VLIW 4 or 5 wide SIMD instructions. See figure 2 for an overview of ATI hardware. In NVIDIA hardware they call compute units “stream multiprocessors” (SM‘s) (and in some of their documentation they are refereed to as “CUDA cores”). In either case, the take away is that there is a fairly complex hardware hierarchy capable of executing at the lowest level SIMD VLIW instructions.

An important caveat to keep in mind is that the marketing numbers for core count for NVIDIA and ATI aren‘t always a good representation of the capabilities of the hardware. For instance, on NVIDIA‘s website a Quadro 2000 graphics card has 192 “Cuda Cores”. However, we can query the lower-level hardware capabilities using the OpenCL API and what we find is that in reality there are actually 4 compute units, all consisting of 12 stream multiprocessors, and each stream multiprocessor is capable of 4-wide SIMD. $192 = 4 * 12 * 4$. In the author‘s opinion this makes the marketing material confusing, since you wouldn‘t normally think of a hardware unit capable only of executing floating point operations as a “core”. Similarly, the marketing documentation for a HD6970 (very high end GPU from ATI at time of writing) shows 1536 processing elements, while in reality the hardware has 24 compute units (SIMD engines), and 16 groups of 4-wide processing elements per compute unit. $1536 = 24 * 16 * 4$.

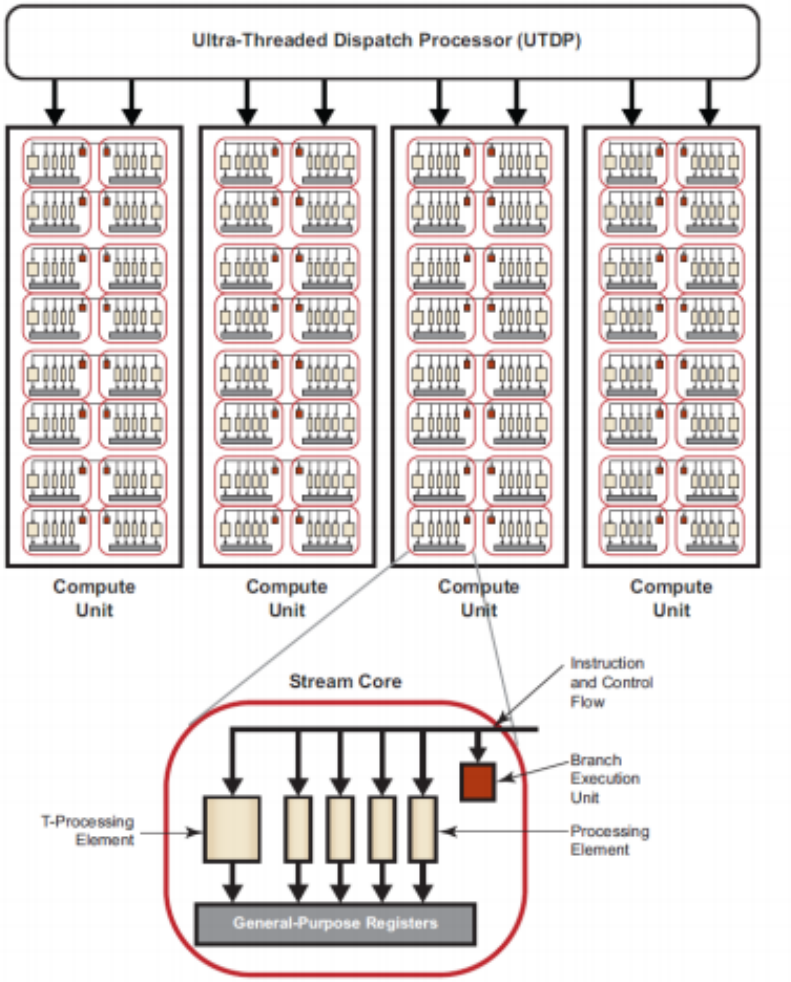

Figure 2: Simplified block diagram of ATI compute device (from [ATI 2011])

2.2 OpenCL Execution Models

At the top level the OpenCL host 4 uses the OpenCL API platform layer to query and select compute devices, submit work to these devices and manage the workload across compute contexts and work queues. In contrast, at the lower end of the execution hierarchy (and at the heart of all OpenCL code) are OpenCL "Kernels" running on the each processing element. These Kernels are written in OpenCL C 5 that execute in parallel over a predefined N-dimensional computation domain. In OpenCL vernacular, each independent element of execution in this domain is called a "work-item" (which NVIDIA refers to as "CUDA threads"). These work-items are grouped together into independent "work-groups" (which NVIDIA refers to as a "thread block"). See Figure 3 for a top level overview of this structure.

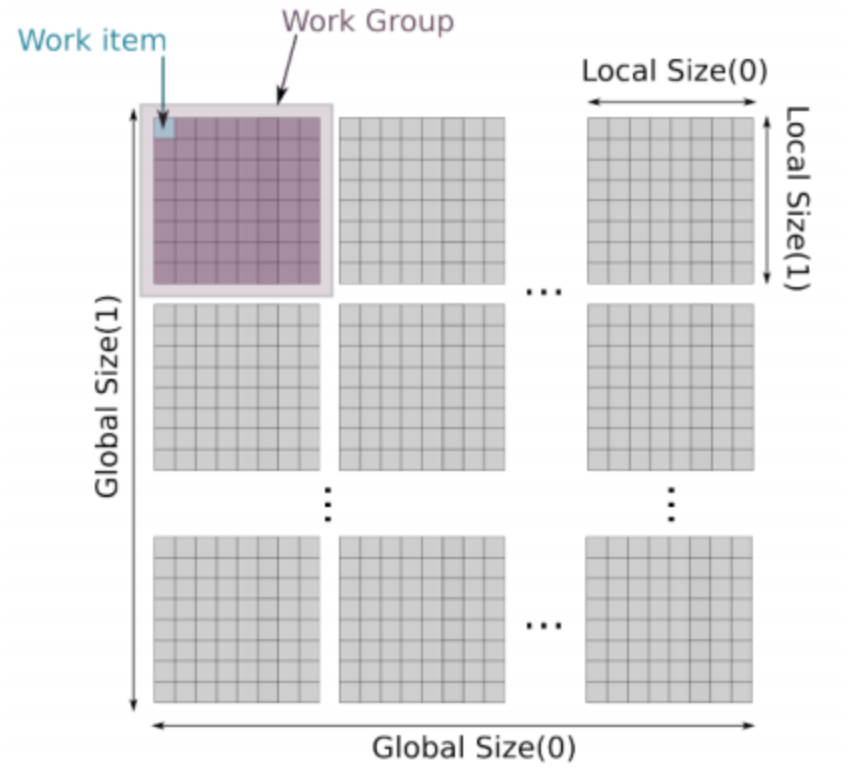

Figure 3: 2D Data-Parallel execution in OpenCL (from [Boydstun 2011])

According to the documentation, the execution model is "finegrained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism" [NVIDIA 2012]. Data-parallel programming is where the domain of execution for each thread is defined by some region over a data structure or memory object (typically a range of indices into an N-by-N array as depicted by Figure 3), where execution over these sub-regions are deemed independent. The alternative model is task-parallel programming, whereby concurrency is exploited across domains of task level parallelism. OpenCL API exploits both of these, however since access to global memory is slow one must be careful in writing Kernel code that reflects the memory access performances of certain memory locations in the hierarchy (more on memory hierarchy later). In this way work-groups can be separated by task-parallel programming (since threads within a work-group can share local memory), but are more likely sub-domains in some larger data structure as this benefits hardware memory access (since getting data from DRAM to global GPU memory is slow, as is getting data from global GPU memory to local work-group memory).

Since hundreds of threads are executed concurrently which results in a linear scaling in instruction IO bandwidth, NVIDIA uses a SIMT (Single-Instruction, Multiple-Thread) architecture. One instruction call in this architecture executes identical code in

parallel by different threads and each thread executes the code with different data. Such a scheme reduces IO bandwidth and allows for more compact thread execution logic. ATI's architecture follows a very similar model (although the nomenclature is different).

With the framework described above, we can now outline the basic pipeline for a GPGPU OpenCL application.

1. Firstly, a CPU host defines an N-dimensional computation do main over some region of DRAM memory. Every index of this N-dimensional computation domain will be a work-item and each work-item executes the same Kernel.
2. The host then defines a grouping of these work-items into work-groups. Each work-item in the work-groups will execute concurrently within a compute unit (NVIDIA streaming multiprocessor or ATI SIMD engines) and will share some local memory (more later). These work-groups are placed onto a work-queue.
3. The hardware will then load DRAM memory into the global GPU RAM and execute each work-group on the work-queue.
4. On NVIDIA hardware the multiprocessor will execute 32 threads at once (which they call a "warp group"), if the work group contains more threads than this they will be serialized, which has obvious implications on the consistency of local memory.

Each processing element executes purely sequential code. There is no branch prediction and no speculative execution, so that all instructions in a thread are executed in order. Furthermore, some conditional branch code will actually require execution of both branch paths, which are then data-multiplexed to produce a final result. I will refer the reader to the Khronos OpenCL, ATI and NVIDIA documentations for further details since the details are often complicated. For instance, a "warp" in NVIDIA hardware executes only one common instruction at a time on all threads in the work-group (since access to individual threads is through global SIMT instructions), so full efficiency is only realized when all 32 threads in the warp agree on their execution path.

There are some important limitations on work-groups to always keep in mind. Firstly, the global work size must be a multiple of the work-group size, or another way of saying that is that the work groups must fit evenly into the entire data structure. Secondly, the work-group size (which of a 2D array would be the *size*2) must be less than or equal to the CL KERNEL WORK GROUP SIZE flag. This is a hardware flag stating the limitation on the maximum con current threads within a work-group. OpenCL will return an error code if either of these conditions are violated 6.

2.3 OpenCL Memory Model

The OpenCL memory hierarchy (shown in Figure 4) is structured in order to "loosely" resemble the physical memory configurations in ATI and NVIDIA hardware. The mapping is not 1 to 1 since NVIDIA and ATI define their memory hierarchies differently. However the basic structure of top global memory vs local memory per work-group is consistent across both platforms. Furthermore, the lowest level execution unit has a small private memory space for program registers.

These work-groups can communicate through shared memory and synchronization primitives, however their memory access is independent of other work-groups (as depicted in Figure 5). This is essentially a data-parallel execution model, where the domain of independent execution units is closely tied and defined by the underlining memory access patterns. For these groups, OpenCL implements a relaxed consistency, shared memory model. There are exceptions, and some compute devices (notably CPUs) can execute task-parallel compute Kernels, however the bulk of OpenCL applications on GPGPU hardware will execute strictly data-parallel workers.

An important issue to keep in mind when programming OpenCL Kernels is that memory access on the DRAM global and local memory blocks is not protected in any way. This means that seg-faults are not reported when work-items dereference memory outside their own global storage. As a result, GPU memory set aside for the OS can be clobbered unintentionally, which can result in behaviors ranging from benign screen flickering up to frustrating blue screens of death and OS level crashes.

Another important issue is that mode-switches may result in GPU memory allocated to OpenCL to be cannibalized by the operating system. Typically the OS allocates some portion of the GPU memory to the "primary-surface", which is a frame buffer store for the rendering of the OS. If the resolution is changed during OpenCL execution, and the size of this primary-surface needs to grow, it will use OpenCL memory space to do so. Luckily these events are caught at the driver level and will cause any call to the OpenCL runtime to fail and return an invalid context error.

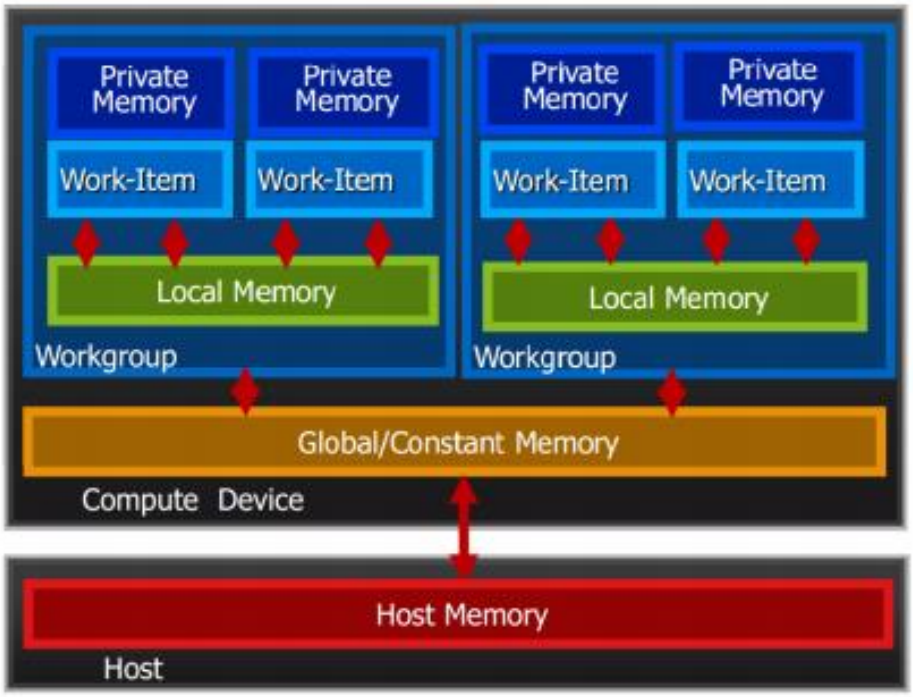

Figure 4: OpenCL Memory Model (from [Khronos 2011])

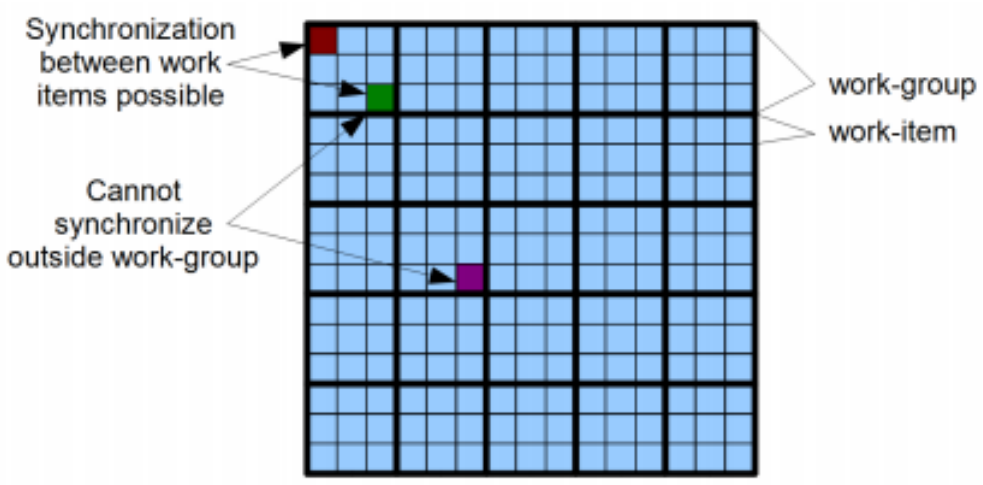

Figure 5: OpenCL Work-group / Work-unit structure

Memory fences are possible within threads in a work-group as well as synchronization barriers for threads at the work-item level (between individual threads in a processing element) as well as at the work-group level (for coarse synchronization between work groups). On the host side, blocking API functions can perform waits for certain events to complete, such as all events in the queue to finish, specific events to finish, etc. Using this coarse event control the host can decide to run work in parallel

across different de vices or sequentially, depending on how markers are placed in the work-queue (as depicted in Figure 6).

Finally, you should also be careful when statically allocating local data (per work-group). You should check the return conditions from the host API for flags indicating that you‘re allocating too much per work-group, however you should also be aware that sometimes the Kernel will compile anyway and will result in a program crash

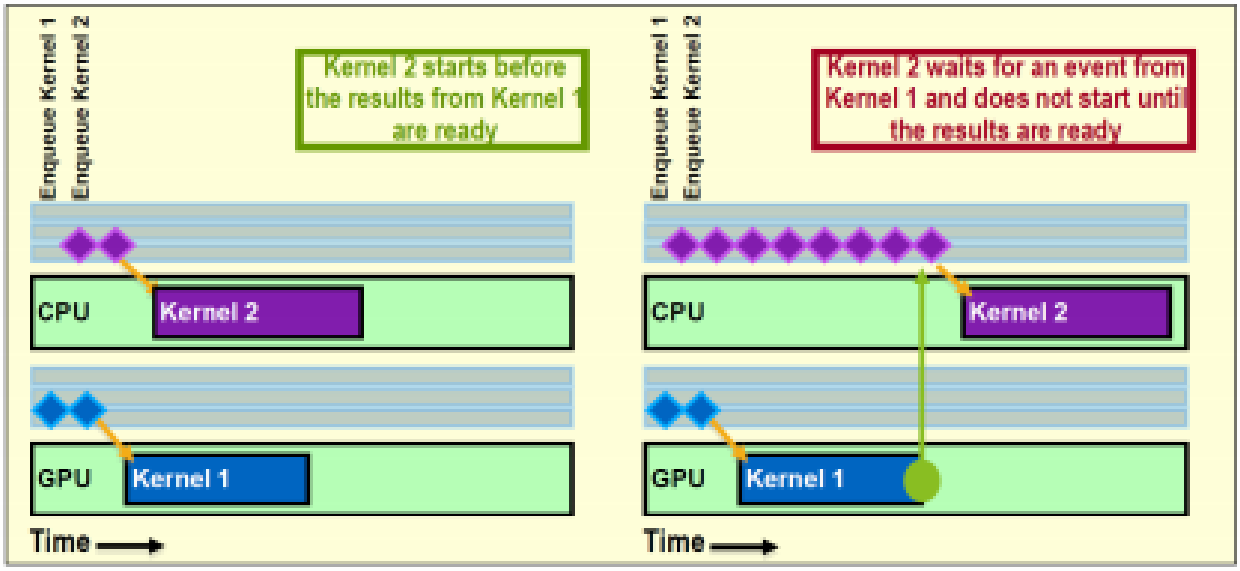

Figure 6: Concurrency control with OpenCL event-queueing

Conclusion

A primary benefit of OpenCL is substantial acceleration in parallel processing.

OpenCL takes all computational resources, such as multi-core CPUs and GPUs, as peer computational units and correspondingly allocates different levels of memory, taking advantage of the resources available in the system. OpenCL also complements the existing OpenGL® visualization API by sharing data structures and memory locations without any copy or conversion overhead.

A second benefit of OpenCL is cross-vendor software portability. This low-level layer draws an explicit line between hardware and the upper software layer. All the hardware implementation specifics, such as drivers and runtime, are invisible to the upper-level software programmers through the use of high-level abstractions, allowing the developer to take advantage of the best hardware without having to reshuffle the upper software infrastructure. The change from proprietary programming to open

standard also contributes to the acceleration of general computation in a cross-vendor fashion.

References

1. ATI, 2011. Programming guide: AMD accelerated parallel processing. http://developer.amd.com/sdks/amdappsdk/assets/amd_accelerated_parallel_processing_opencl_programming_guide.pdf.

2. BOYDSTUN, K., 2011. Introduction opencl (caltech lecture). http://www.tapir.caltech.edu/~kboyds/OpenCL/opencl.pdf.

3. COPPERSMITH, D., AND WINOGRAD, S. 1987. Matrix multiplication via arithmetic progressions. In Proceedings of the nineteenth annual ACM symposium on Theory of computing, ACM, New York, NY, USA, STOC '87, 1–6.

4. HUSS-LEDERMAN, S., JACOBSON, E. M., TSAO, A., TURNBULL, T., AND JOHNSON, J. R. 1996. Implementation of strassen's algorithm for matrix multiplication. In Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM), IEEE Computer Society, Washington, DC, USA, Supercomputing '96.

5. KHRONOS, 2011. Opencl overview. http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf/.

6. KHRONOS, 2012. Opencl 1.2 reference pages. http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/.

7. NVIDIA, 2012. Opencl programming guide for the cuda architecture, version 4.2. http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf.

8. SUTTER, H. 2005. The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal.

Glossary

Plateaued – Плато

Subsequent – Последовательный

Execution model – Модель исполнения

Lithography – Литография

Low level hardware – Низкоуровневое аппаратное обеспечение

Open Computing Language – Открытый Язык Вычислений

Graphics Processing Unit – Графический процессор

Parallel computation – Параллельные вычисления

Superscalar – Суперскалярный

Cross-platform – Кросс-платформенный

Deterministic – Детерминированный

Intend – Предназначать

Heterogeneous computing – Гетерогенные вычисления

Digital signal processor – Процессор цифровых сигналов

Specification – Спецификация

Memory model – Модель памяти

Consistent memory – Консистентная память

FLOPS (Floating-point Operations Per Second) – Операций с плавающей точкой в секунду

VLIW (Very Long Instruction Word) – Очень длинная инструкция

Vernacular – Общеупотребительный

Thread – Поток

Finegrained parallelism – Мелкозернистый параллелизм

Coarse-grained parallelism – Крупнозернистый параллелизм

SIMT (Single-Instruction, Multiple-Thread) – ОКМП (Одиночный поток команд, множественные потоки выполнения)

IO bandwidth – Пропускная способность ввода-вывода

Branch prediction – Предсказание ветвления

Speculative execution – Спекулятивное исполнение

Work-item – Рабочий элемент

Shared memory – Разделяемая память

Local memory – Локальная память

Shared memory – Разделяемая память