University of Glasgow | School of Computing Science

General Purpose GPU Programming

Advanced Operating Systems
Tutorial 7

Tutorial Outline

- Review of lectured material
- Key points
- Discussion
    - OpenCL
    - Future directions

Review of Lectured Material

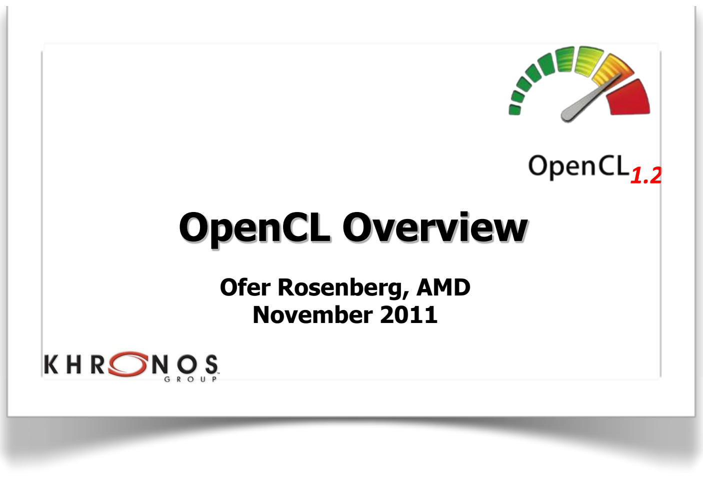
- Heterogeneous instruction set systems
- Heterogeneous multi-kernel systems – Helios
- Main core with heterogenous offload
  - Graphics offload hardware – GPGPU
  - Programming model
  - OpenCL
  - Integration with operating systems
- Heterogenous virtual machines – Hera JVM
- Hybrid models – Accelerator
  - Lazy encoding of SIMD-style operations and JIT compilation into type system

Key Points

- Increasing heterogeneity of hardware
- Programming models are complex
    - Too limited to run a full operating system
    - Too different to effectively run standard programming languages
- OpenCL-style offload model performs well, but is complex to program
- Attempts to hide complexity in VM have had mixed success

Discussion

- What is complexity versus performance trade-off in OpenCL – how does this limit usefulness?
- How can SIMD-style processing be more cleanly incorporated into modern languages?
- Is the embedded DSL approach of Accelerator a set in the right direction, or is the complexity of the VM excessive?
- How to use heterogenous processing resources?



OpenCL 1.2

OpenCL Overview

Ofer Rosenberg, AMD
November 2011

KHRONOS GROUP

Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses

David Tarditi Sidd Puri Jose Oglesby
Microsoft Research
{dtarditi,siddpuri,joseogl}@microsoft.com

Abstract

GPUs are difficult to program for general-purpose uses. Programmers can either learn graphics APIs and convert their applications to use graphics pipeline operations or they can use stream programming abstractions of GPUs. We describe Accelerator, a system that uses data parallelism to program GPUs for general-purpose uses instead. Programmers use a conventional imperative programming language and a library that provides only high-level data-parallel operations. No aspects of GPUs are exposed to programmers. The library implementation compiles the data-parallel operations on the fly to optimized GPU pixel shader code and API calls. We describe the compilation techniques used to do this. We evaluate the effectiveness of using data parallelism to program GPUs by providing results for a set of compute-intensive benchmarks. We compare the performance of Accelerator versions of the benchmarks against hand-written pixel shaders. The speeds of the Accelerator versions are typically within 50% of the speeds of hand-written pixel shader code. Some benchmarks significantly outperform C versions on a CPU: they are up to 18 times faster than C code running on a CPU.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Measurement, Performance, Experimentation, Languages

Keywords Graphics processing units, data parallelism, just-in-time compilation

1. Introduction

Highly programmable graphics processing units (GPUs) became available in 2001 [10] and have evolved rapidly since then [15]. GPUs are now highly parallel processors that deliver much higher floating-point performance for some workloads than comparable CPUs. For example, the ATI Radeon x1900 processor has 48 pixel shader processors, each of which is capable of 4 floating-point operations per cycle, at a clock speed of 650 MHz. It has a peak floating-point performance of over 250 GFLOPS using single-precision floating-point numbers, counting multiply-adds as two FLOPs. GPUs have an explicitly parallel programming model and their performance continues to increase as transistor counts increase.

The performance available on GPUs has led to interest in using GPUs for general-purpose programming [16, 8]. It is difficult, however, for most programmers to program GPUs for general-purpose uses.

In this paper, we show how to use data parallelism to program GPUs for general-purpose uses. We start with a conventional imperative language, C# (which is similar to Java). We provide a library that implements an abstract data type providing data-parallel arrays; no aspects of GPUs are exposed to programmers. The library evaluates the data-parallel operations using a GPU; all other operations are evaluated on the CPU. For efficiency, the library does not immediately perform data-parallel operations. Instead, it builds a graph of desired operations and compiles the operations on demand to GPU pixel shader code and API calls.

Data-parallel arrays only provide aggregate operations over entire input arrays. The operations are a subset of those found in languages like APL and include element-wise arithmetic and comparison operators, reduction operations (such as sum), and transformations on arrays. Data-parallel arrays are functional: each operation produces a new data-parallel array. Programmers must explicitly convert back and forth between conventional arrays and data-parallel arrays. The lazy compilation is typically done when a program converts a data-parallel array to a normal array.

Compiling data-parallel operations lazily to a GPU allows us to implement the operations efficiently: the system can avoid creating large numbers of temporary data-parallel arrays and optimize the creation of pixel shaders. It also allows us to avoid exposing GPU details to programmers: the system manages the use of GPU resources automatically and amortizes the cost of accessing graphics APIs. Compilation at run time also allows the system to handle properties and features that vary across GPU manufacturers and models.

We have implemented these ideas in a system called Accelerator. We evaluate the effectiveness of the approach using a set of benchmarks for compute-intensive tasks such as image processing and computer vision, run on several generations of GPUs from both ATI and NVidia. We implemented the benchmarks in hand-written pixel shader assembly for GPUs, C# using Accelerator, and C++ for the CPU. The C# programs, including compilation overhead, are typically within 2×of the speed of the hand-written pixel shader programs, and sometimes exceed their speeds. The C# programs, like the hand-written pixel shader programs, often outperform the C++ programs (by up to 18×).

Prior work on programming GPUs for general-purpose uses either targets the specialized GPU programming model directly or provides a stream programming abstraction of GPUs. It is difficult to target the GPU directly. First, programmers need to learn the graphics programming model, which is specialized to the set of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'06 October 21–25, 2006, San Jose, California, USA.
Copyright © 2006 ACM 1-59593-451-0/06/0010...$5.00.

325