University of Glasgow | School of Computing Science

Concurrency

Advanced Operating Systems (M)
Tutorial 7

Tutorial Outline

- Review of lectures
- Key learning outcomes
- Discussion

Review of Lectures

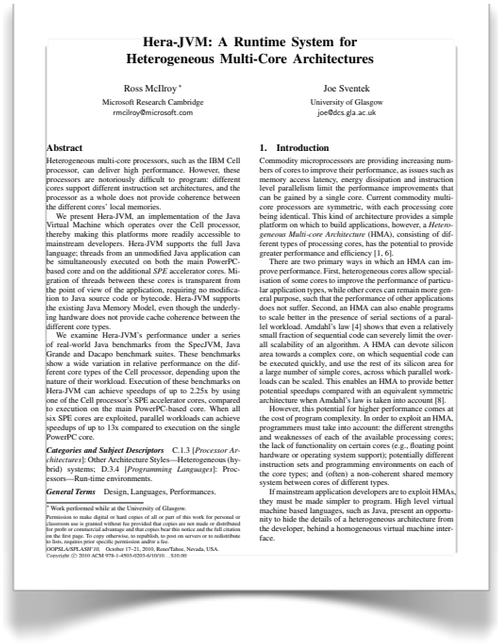
- Hardware trends towards multicore
  - Moore's law, power consumption, cores, and interconnects
  - NUMA optimisations: locality-aware memory allocation; replication of kernel data structures; locality-aware scheduling
  - Difficulty of maintaining cache coherence; benefits of message passing
- Multi-kernel model
  - Distributed system model; explicit communication; replicated state
  - Example: Barrelfish
- Heterogenous instruction set systems
  - Programming models: slave core, multi-kernel model, VM with transparent offload, hybrid VM with device-specific features
  - Examples: Helios, OpenCL, Hera-JVM, Accelerator

Key Learning Outcomes

- Understanding of the hardware trends towards multicore systems, heterogeneity
- Benefits and limitations of the multi-kernel model for structuring operating systems
- Understanding the trade-off between the different approaches to handling heterogeneity
- Familiarity with concepts underlying the example systems: Helios, OpenCL, Hera-JVM, Accelerator

Discussion: Hera-JVM

- Compilation to the SPE core – relatively standard
- Software caching of heap objects
  - DMA access to main memory only, used to cache entire objects
  - Java memory model maintained through cache invalidation (§5.3) – expensive
  - Software-managed cache
- Software caching of methods
  - DMA used to copy methods to SPE when they're invoked
  - Each object may be JIT compiled twice, once for PPE, once for SPE
  - Problems with returning from a method, if caller has been evicted (§5.2)
- How to trigger migration to SPE core?
  - Explicit core annotations on a method
  - Annotating methods with the sort of operations performed, with the runtime deciding which core is appropriate – transparent but difficult
- Performance
- Programming model – is this a good approach?



Hera-JVM: A Runtime System for Heterogeneous Multi-Core Architectures

Ross McIlroy *
Microsoft Research Cambridge
rmcilroy@microsoft.com

Joe Sventek
University of Glasgow
joe@dcs.gla.ac.uk

Abstract

Heterogeneous multi-core processors, such as the IBM Cell processor, can deliver high performance. However, these processors are notoriously difficult to program: different cores support different instruction set architectures, and the processor as a whole does not provide coherence between the different cores' local memories.

We present Hera-JVM, an implementation of the Java Virtual Machine which operates over the Cell processor, thereby making this platforms more readily accessible to mainstream developers. Hera-JVM supports the full Java language; threads from an unmodified Java application can be simultaneously executed on both the main PowerPC-based core and on the additional *SPE* accelerator cores. Migration of threads between these cores is transparent from the point of view of the application, requiring no modification to Java source code or bytecode. Hera-JVM supports the existing Java Memory Model, even though the underlying hardware does not provide cache coherence between the different core types.

We examine Hera-JVM's performance under a series of real-world Java benchmarks from the SpecJVM, Java Grande and Dacapo benchmark suites. These benchmarks show a wide variation in relative performance on the different core types of the Cell processor, depending upon the nature of their workload. Execution of these benchmarks on Hera-JVM can achieve speedups of up to 2.25x by using one of the Cell processor's SPE accelerator cores, compared to execution on the main PowerPC-based core. When all six SPE cores are exploited, parallel workloads can achieve speedups of up to 13x compared to execution on the single PowerPC core.

Categories and Subject Descriptors C.1.3 [*Processor Architectures*]: Other Architecture Styles—Heterogeneous (hybrid) systems; D.3.4 [*Programming Languages*]: Processors—Run-time environments.

General Terms Design, Languages, Performances.

* Work performed while at the University of Glasgow.

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.
JSOPSLA/SPLASH'10, October 17-21, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM 978-1-4503-0203-6/10/10. . . $10.00

1. Introduction

Commodity microprocessors are providing increasing numbers of cores to improve their performance, as issues such as memory access latency, energy dissipation and instruction level parallelism limit the performance improvements that can be gained by a single core. Current commodity multi-core processors are symmetric, with each processing core being identical. This kind of architecture provides a simple platform on which to build applications, however, a *Heterogeneous Multi-core Architecture* (HMA), consisting of different types of processing cores, has the potential to provide greater performance and efficiency [1, 6].

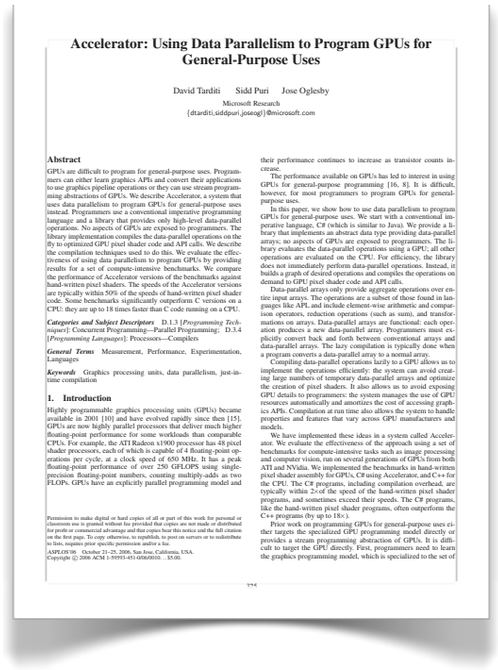
There are two primary ways in which an HMA can improve performance. First, heterogeneous cores allow specialisation of some cores to improve the performance of particular application types, while other cores can remain more general purpose, such that the performance of other applications does not suffer. Second, an HMA can also enable programs to scale better in the presence of serial sections of a parallel workload. Amdahl's law [4] shows that even a relatively small fraction of sequential code can severely limit the overall scalability of an algorithm. A HMA can devote silicon area towards a complex core, on which sequential code can be executed quickly, and use the rest of its silicon area for a large number of simple cores, across which parallel workloads can be scaled. This enables an HMA to provide better potential speedups compared with an equivalent symmetric architecture when Amdahl's law is taken into account [8].

However, this potential for higher performance comes at the cost of program complexity. In order to exploit an HMA, programmers must take into account: the different strengths and weaknesses of each of the available processing cores; the lack of functionality on certain cores (e.g., floating point hardware or operating system support); potentially different instruction sets and programming environments on each of the core types; and (often) a non-coherent shared memory system between cores of different types.

If mainstream application developers are to exploit HMAs, they must be made simpler to program. High level virtual machine based languages, such as Java, present an opportunity to hide the details of a heterogeneous architecture from the developer, behind a homogeneous virtual machine interface.

Discussion: Accelerator

- GPGPU offload using data parallel arrays in C#
- Architecture of GPGPU device
- Programming model
  - Data parallel arrays
  - Lazy evaluation, JIT compilation and execution when result assigned to a standard array



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

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

- Comparison with OpenCL
  - Programming model is simpler but more restrictive
  - Is it sufficient?

- Trade-offs compared to Hera-JVM?