University of Glasgow | School of Computing Science

Concurrency 2: Programming Heterogenous Multicore Systems

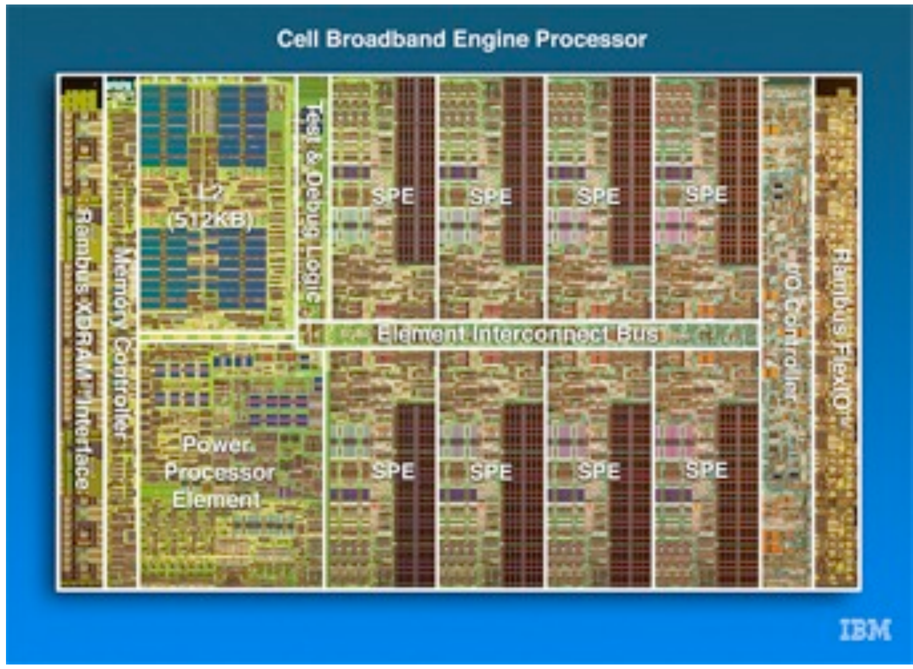
Advanced Operating Systems (M)
Lecture 17

Lecture Outline

- Heterogeneous instruction set systems
- Programming models
  - Multi-kernel peer model
  - Offload to slave processor
  - Abstraction via virtual machine
- Discussion
  - Hybrid models – Microsoft’s Accelerator framework

Heterogeneous Instruction Set Systems

- Increasingly common for a single system to have cores running different instruction sets
  - CPU + GPU
  - CPU + offload of TCP, crypto, or multimedia functions
  - Cell processor with PPE + multiple SPE
- Desirable when different instruction sets have radically different performance characteristics
  - GPU hardware does simple SIMD-style computations at high speed, but performs very poorly for code with large numbers of conditional branches
  - A typical CPU is better suited for complex conditional code, but performs poorly with SIMD operations



- CPU + GPU model is ubiquitous; others becoming more common

Programming Models

- How to program a heterogenous instruction set system?
  - If the cores have radically different characteristics, do they need different programming models and/or languages?
  - Should the cores be peers, or is a master/slave model appropriate?

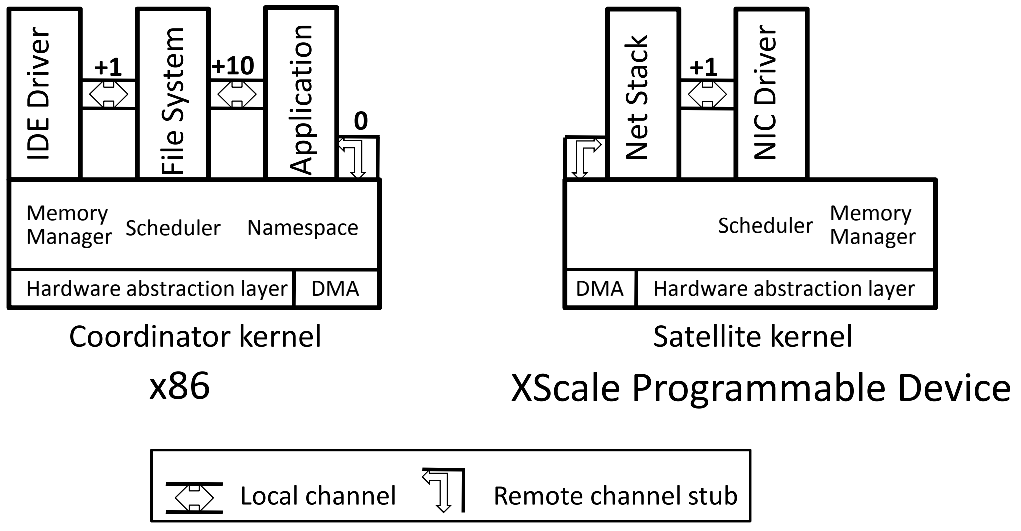
- Three main alternatives have been explored
  - Multi-kernel model – heterogenous cores
  - Offload to slave processor
  - Abstraction via virtual machines

Multi-kernel: Heterogenous Cores

- If cores are full-featured, a multi-kernel model may be appropriate
    - The multi-kernel model is a distributed system, with message passing – the underlying instruction set is unimportant, but the kernel needs to be recompiled for each architecture
    - Applications are either limited to a subset of the cores, require compilation as fat binaries, or use JIT compilation
        - May not be possible to effectively balance load across the system, due to limitations where certain processes can execute
        - Performance may suffer if related processes can't be co-located due to resource constraints
    - Heavy-weight approach, but offers considerable flexibility
    - Not widely implemented – systems with multiple full-featured cores generally use a homogenous instruction set

Multi-kernel: Example – Helios

- A research prototype multi-kernel system designed to exploit heterogenous cores
- Multi-kernel extension to Singularity
  - Runs on x86 NUMA systems, and on x86 systems with offload to an ARM processor on a RAID card
- Based on a *satellite kernel* abstraction, allowing weak cores to delegate some work to more full-featured cores
  - All kernels export the same services and message-passing APIs, but some services are implemented by forwarding messages to other cores
  - Applications distributed as JIT compiled byte code; express *affinity* to other processes in metadata to allow dynamic load balancing across cores
  - Good performance on benchmarks, but these only considered a limited set of processes on the ARM core, with clear communication patterns and affinity – unclear how this will work in general with highly asymmetric cores



Helios multi-kernel architecture

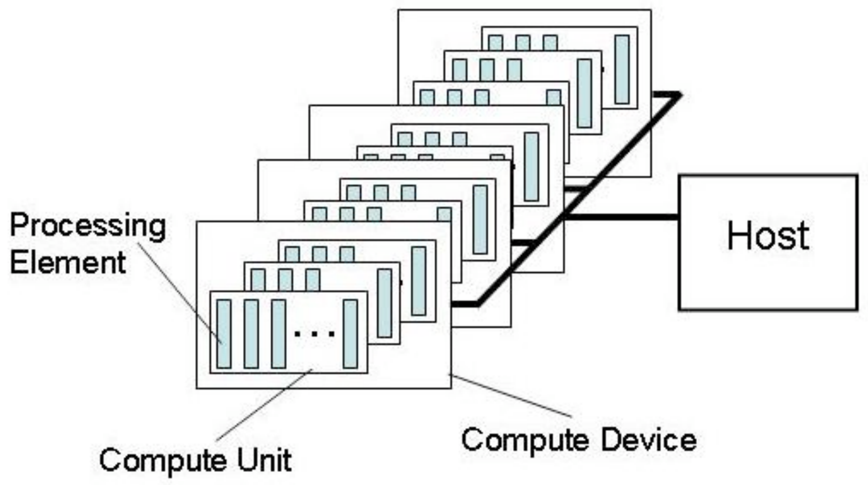
E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In Proceedings of the Symposium on Operating Systems Principles (SOSP), Big Sky, MT, USA, October 2009. ACM. DOI 10.1145/1629575.1629597

Slave Processor

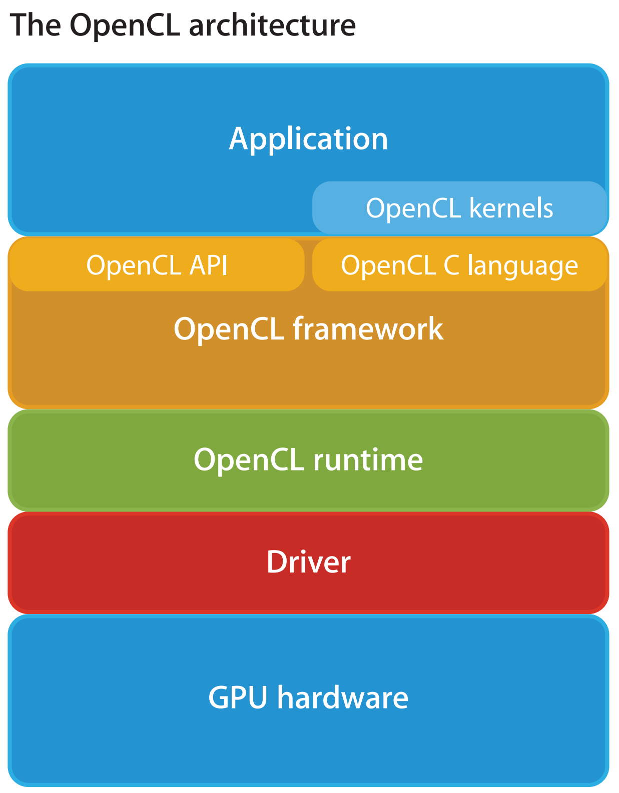
- The system has a master CPU, plus one or more slave processors running a different instruction set
  - The programming model for the slave processors is different to the master
  - Slave processors often too limited to run a full kernel and general-purpose programming language; code for the slave processors written in a special language, and compiled separately (e.g., OpenCL or CUDA)
- Slave processors don’t run independently – they’re resources invoked by the master CPU
  - Graphics processing
  - TCP or crypto offload

Slave Processor: Example – OpenCL

- Slave processors are GPU devices – large grid of compute cores designed for SIMD-style array processing
  - 512 cores available on modern GPUs
  - Weak support for conditional branches – the model is that each core runs the same code on different data
- Slave processor code written in OpenCL C
  - Extended subset of C: http://www.khronos.org/opencl/
  - Adds built-in vector, 2D, and 3D image types
  - Adds pointer qualifiers to reference host and GPU memory; use of pointers restricted since memory is not shared between host and device
    - Must explicitly copy inputs and outputs to/from slave device
  - Very restricted standard library
  - Defines the concept of a *kernel function* that can be JIT compiled and executed on a device, and runtime support code to allow device management
  - Runtime creates massive numbers of threads, each running the kernel on different parts of the data



[Source: The OpenCL specification, v1.0]



[Source: Apple, OpenCL technology brief]

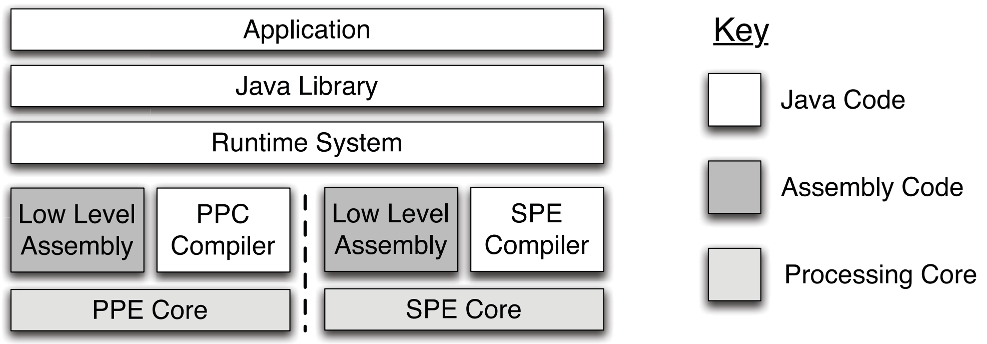
Slave Processor: Example – OpenCL

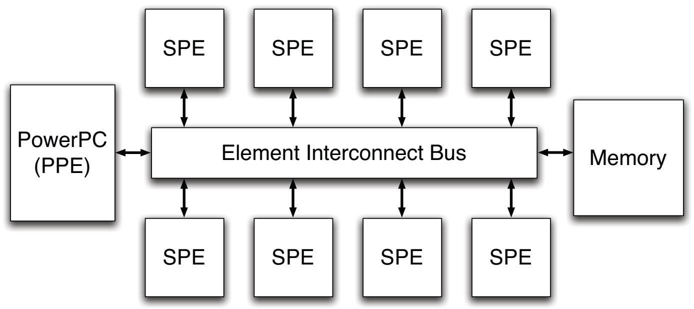
- OpenCL runtime on host system manages offload of OpenCL code to slave devices
  - JIT compilation; host driver code controls exactly what OpenCL functions execute, and when
  - OpenCL devices do not run an OS – they're dumb devices, managed by a device driver
- Low-level API and programming model
  - High conceptual burden to use
  - Cannot run general purpose code; programming and communications model is too restricted
  - Does not easily integrate with host applications – too much boilerplate

VM Abstraction

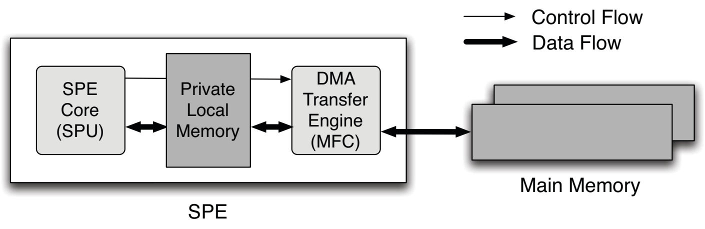
- Use of separate languages and compilation stages for slave processors is complex and error prone
  - OpenCL and CUDA have complex and poorly-defined semantics
  - High cognitive overhead on programmers; difficult to develop and debug
- Alternative: write in a high-level language targeting a virtual machine; let the VM handle the offload
  - E.g., a Java virtual machine that can JIT compile for different cores
  - Pushes complexity onto the VM – simple for application programmer
  - But, high-level languages often not a good fit for slave processors
    - e.g., the JVM has no natural means to express SIMD-style array processing operations, and encourages conditional execution, imperative code, and mutable state – the opposite of what is needed for good GPU code
    - But, a language optimised for GPU processing would perform poorly on a general-purpose CPU, with a small number of cores optimised for imperative code

VM Abstraction: Example – Hera-JVM





(a) The architecture of the Cell processor.



(b) An SPE core's memory subsystem.

R. McIlroy and J. Sventek, Hera-JVM: A Runtime System for Heterogeneous Multi-Core Architectures, Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Reno, Nevada, October 2010.
DOI: 10.1145/1869459.1869478

- A JVM for the Cell processor, than can offload methods from PPE to SPE cores
  - JIT compilation; methods compiled for appropriate core based on runtime code placement algorithm
  - Data caching: SPE memory is not cache coherent; data cached on SPE when method starts; cache flushed at synchronisation points, following Java memory model
  - Methods copied to SPE memory in their entirety; migration onto the SPE causes an entire method, and any methods it calls, to run on the SPE
  - Garbage collector understands both architectures, and the caches on the SPEs
  - Hard to decide which methods to migrate to SPE:
    - Explicit annotations (@RunOnSPECore, @RunOnPPECore) work, but place high overhead on programmer
    - Behaviour hints (@ArithmeticCode, @ObjectAccessCode, @LargeWorkingSet) allow the JVM runtime to automatically migrate methods to the SPEs, but are suboptimal
    - Optimal solution is an open problem

Discussion

- Offload to slave processor model is common
    - Hard for programmer, but gives good performance
    - Main kernel treats the GPU as a resource, that can be claimed by a process, and managed as any other resource
- Abstraction via virtual machine conceptually clean
    - In principle, allows transparent offload of work from main processor to subordinate processors such as GPUs
    - Difficult in practice: applications written without account for the different processor types and capabilities, and don't aid the runtime; insufficient information for the runtime to effectively offload work – likely inefficient

Hybrid Virtual Machine/Slave Processor

- Hybrid model: wrap a device-specific programming model in the virtual machine, alongside a general purpose language
  - E.g., for GPU offload, add a data parallel array datatype, then JIT compile operations on those arrays to execute on the GPU cores
- Explicit model of device-specific operations, and control over when they execute
- Virtual machine hides low-level details

Example: Accelerator

- Extension to C# to provide data-parallel arrays with GPU offload
    - Support operations such as conversion to/from standard arrays, element-wise arithmetic, reductions, transformations, and matrix algebra
    - Data parallel arrays are lazy, and don't compute their value until converted back to a standard array
    - Lazy evaluation helps efficiency: runtime JIT compiles all operations on a single data parallel array at once, and passes to the GPGPU for execution as a single block
- Similar model to OpenCL, except the complexity of managing the GPU is pushed onto the VM
    - Programming model is very similar, and there is similar control over when code executes on the GPU

```
static float[,] Blur(float[,] array, float[] kernel) {
  float[,] result;
  DFPA parallelArray = new DFPA(array);

  FPA resultX = new FPA(0f, parallelArray.Shape);
  for (int i = 0; i < kernel.Length; i++) {
    int[] shiftDir = new int[] { 0, i};
    resultX += PA.Shift(parallelArray, shiftDir) * kernel[i];
  }

  FPA resultY = new FPA(0f, parallelArray.Shape);
  for (int i = 0; i < kernel.Length; i++) {
    int[] shiftDir = new int[] { i, 0 };
    resultY += PA.Shift(resultX, shiftDir) * kernel[i];
  }
  PA.ToArray(resultY, out result);
  parallelArray.Dispose();
  return result;
}
```

D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose use. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose, CA, USA, October 2006. ACM.

Discussion and Further Reading

- D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose use. Proc. International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA, October 2006. DOI: 10.1145/1168857.1168898

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.

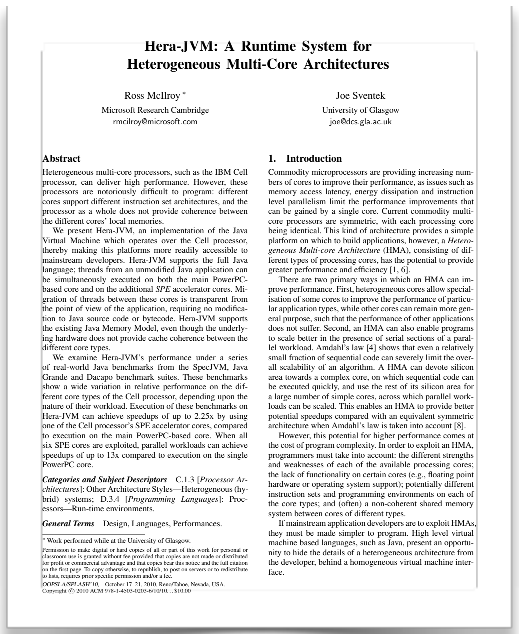
We have implemented these ideas in a system called Accelerator. We evaluate the effectiveness of the approach using a set of benchmarks for compute-intensive tasks such as image processing and computer vision, run on several generations of GPUs from both ATI and NVidia. We implemented the benchmarks 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

- R. McIlroy and J. Sventek, Hera-JVM: A Runtime System for Heterogeneous Multi-Core Architectures, Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Reno, Nevada, October 2010. DOI: 10.1145/1869459.1869478



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.
OOPSLA/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.

- Both to be discussed in tutorial tomorrow (conceptual level – no need to consider performance results)