University of Glasgow | School of Computing Science

# General Purpose GPU Programming (2)

Advanced Operating Systems
Lecture 15

# Lecture Outline

- Programming models (cont'd)
  - Heterogenous virtual machines
- Discussion
- Hybrid and alternative architectures

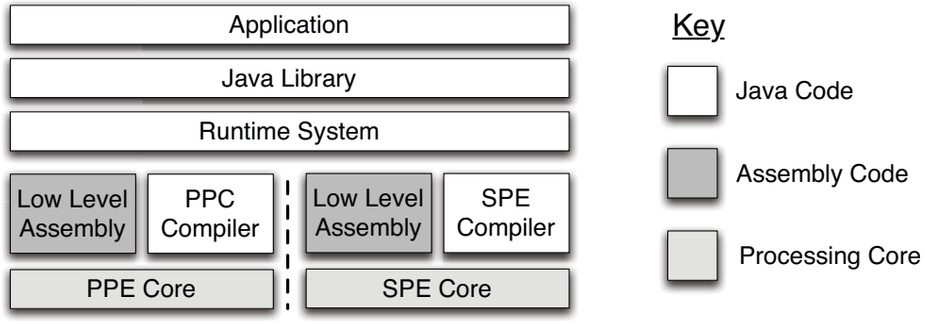
# Heterogeneous Virtual Machines

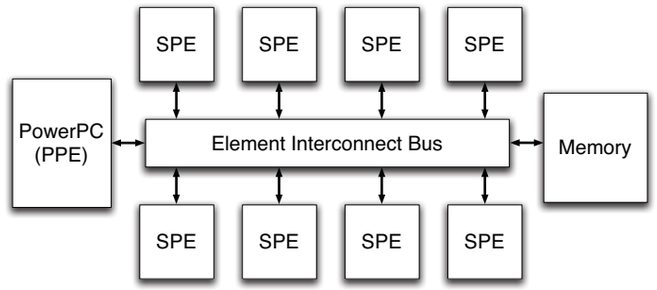
- Multi-kernel and offload models problematic:
  - Heterogeneous multi-kernel model is conceptually simple, but not a good fit for modern hardware
  - Heterogenous offload processors are widely used:
    - But – have high cognitive overhead on programmers, due to SIMD programming model
    - Have a complex and high-overhead offload process, exposing too many low-level details
    - Are difficult to reason about and debug
- Can a heterogeneous virtual machine (VM) model hide some complexity?
  - Rather than expose details of the heterogeneous processor and offload process, hide offload complexity in a virtual machine?
  - Can a JIT compiler translate regular code to fit programming model of the heterogenous offload processor?

# Heterogeneous VM Programming Model

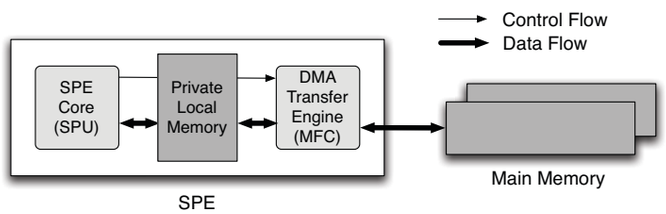
- Write in high-level language targeting VM, ignoring the distinction between processor cores
  - High-level code desirable – specify *what* needs to be done, leaving *how* to the VM and/or run time libraries
  - The VM can implement operations differently depending on the processor architecture targeted
- Let the VM handle the offload
  - The VM can query and setup the heterogenous processor code, exposing only a high-level API (if any) to the programmer
  - The VM can JIT compile code for different processor architectures
  - Pushes complexity onto the VM – simple for application programmer
  - Requires close integration of JIT and VM with operating system kernel

# 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 OOPSLA Conference, October 2010. DOI:10.1145/1869459.1869478

- A JVM for the Cell processor, 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
  - Poor performance, since cannot make effective use of vector instructions on SPE cores

# Limitations of Heterogenous VM Model

- Hera JVM shows high-level languages often not a good fit for heterogenous offload processors
  - Example: JVM cannot express SIMD-style array processing operations, encourages conditional execution, imperative code, and mutable state – opposite of what is needed for good GPU code
  - But, GPU-optimised language would perform poorly on general-purpose CPUs, with small number of cores optimised for imperative code

- Automatically extracting parallelism hasn't been an effective approach
  - Difficult for a single processor architecture
  - Offload to heterogenous cores only complicates problem, due to need to manage offload overhead

# 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
  - Effective, but overly complex programming model
- 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
  - Straight forward programming model, but not effective

# Hybrid Architectures

- Can we wrap a device-specific programming model in the virtual machine, alongside a general purpose language?
  - Add types that represent SIMD-style operations, so giving the VM hints when to offload, and also easing programming model
  - Explicit model of device-specific operations, and control over when they execute
- Virtual machine hides low-level details
- High-level model – coding SIMD-style operations in type system – eases programming

# 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", Proc. ACM ASPLOS, October 2006, DOI:10.1145/1168857.1168898

# Discussion

- Embedding lazy SIMD operations in types eases programming burden
  - Restricted set of operations can be performed in parallel, on appropriate array types – rough match to hardware features
  - Only exploits functional SIMD operations – no flexibility for conditional processing, even if hardware allows
  - Lazy operation can be confusing to programmers – when does the offload and computation occur? – but less complex than OpenCL-style model
- Considerable complexity pushed into VM
  - Good performance needs effective operation of lazy JIT compilation in VM
  - Opaque, and difficult to tune

# Future Directions

- Heterogeneous offload model (e.g., OpenCL) is the only effective solution to date
  - Heterogenous VM offers poor performance – too big a mismatch between VM language and GPGPU hardware
  - Hybrid model has potential, but opaque to tuning, and limited functionality

- Future directions:
  - Higher-level APIs for offload management?
  - DSLs for programming SIMD-style hardware – a minimal pure functional language, with data parallel arrays as main datatype, but link compatible with C++, to replace OpenCL?

# Further Reading

- D. Tarditi, S. Puri, and J. Oglesby, “Accelerator: using data parallelism to program GPUs for general-purpose use”, Proc. ACM ASPLOS, 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