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
  - Heterogeneous 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 heterogeneous 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 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

```csharp
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;
}
```

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