General Purpose GPU Programming (1)

Advanced Operating Systems
Lecture 14
Lecture Outline

- Heterogenous multi-core systems and general purpose GPU programming
- Programming models
  - Heterogenous multi-kernels
  - Main core with heterogenous offload
Heterogeneous Instruction Set Systems

- Increasingly common for a single system to have cores running heterogeneous instruction sets
  - CPU + general purpose 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 in parallel at high speed, but performs poorly for code with large numbers of conditional branches
  - A typical CPU is better suited for complex conditional code, but performs poorly with SIMD-style operations
  - Combination of several CPU cores and a GPU is now a ubiquitous system design
  - Other types of heterogeneity are becoming more common as designers try to make better use of the extra transistors which are coming available due to Moore’s law
Programming Models

• What programming model to use for systems with heterogeneous instruction sets?
  • Radically different cores → radically different programming model?
  • How are cores organised – peers or master/slave architecture?

• Three models have been explored:
  • Heterogeneous multi-kernel
  • Main core with heterogeneous offload
  • Heterogeneous virtual machines (→ next lecture)
Heterogeneous Multi-kernel Systems

- If cores full-featured, multi-kernel may be suitable
  - Multi-kernel is a message passing distributed system – if messages have a standardised format, processor architecture is unimportant
  - Each core may run a different instruction set, since they don’t share data
  - Kernel must be separately compiled for each architecture
  - Applications limited to subset of 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

- Not widely implemented
  - Systems with multiple full-featured cores generally have homogenous instruction sets
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

- Some cores types have limited functionality
  - All kernels export the same services; all interactions between tasks, and with kernel, use message-passing
  - Some cores implement certain services by forwarding messages to other cores

- Applications distributed as byte code
  - JIT compilation, as is usual in Singularity
  - Express affinity to other tasks in metadata to allow dynamic load balancing across cores

- Interesting proof-of-concept, but only limited functionality

---

Main Core With Heterogeneous Offload

• Typical modern hardware architecture:
  • Several full-featured main processor cores, with a common instruction set, run the main operating system
  • Specialised, limited functionality, processors support the main cores, with functions being offloaded from the main cores as appropriate
    • These processors typically have radically different instruction set and/or programming model to the main processor cores
    • Example: a multicore CPU with offload of graphics to a separate GPU

• Implications for operating system design:
  • Programming model for offload processors differs from the main cores
    • Typically not sufficient to run a full operating system
    • May be too limited to support a standard programming language
  • Offload processors don’t run independently – they’re resources invoked by the master CPU (e.g., for graphics, network stack, or crypto offload)
Graphics Offload Hardware

- A ubiquitous offload model is for graphics: general purpose GPU programming
- Offload processors are GPU devices – a large grid of compute cores designed for SIMD-style array processing
  - Thousands of cores available on modern GPUs
  - Very high sequential memory bandwidth – designed for array processing, with limited support for pointers and arbitrary memory access
  - Weak support for conditional branches – the model is that each core runs the same code on different data → SIMD processing
  - No access to I/O devices other than the screen; interactions with the rest of the system via block DMA transfers
- Parallel SIMD programming model makes such cores unsuitable for general-purpose programming languages
  - Cannot effectively run general purpose software
Programming Model

- Use multiple threads rather than loops
  - Operate on entire arrays in parallel, rather than iterating over elements
  - Threads may run in lock-step across thousands of cores – a branch may disrupt execution across threads if it increases execution time

- Designed for batch array processing – “general purpose” implies flexibility in how each element is processed

- Task scheduler on offload processor assigns work
  - Implicit parallelism – programmers write *kernel function*, system parallelises
Example: OpenCL (1)

- An open, vendor neutral, standard for programming general purpose GPU offload devices
  - http://www.khronos.org/opencl/
- General purpose GPU code is written in OpenCL C
  - An extended subset of ISO C99
  - 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 (explicitly copy inputs and outputs to/from device)
  - Very restricted standard library
  - Defines the concept of a kernel function that can be JIT compiled and executed on a device
- OpenCL framework provides JIT compilation and device management
- OpenCL runtime manages execution of code as a large number of threads, running kernel functions on different parts of the data
Example: OpenCL (2)

- Main component of OpenCL C programs: kernel functions executed on device
  - The `global` qualifier on pointers specifies memory region on which they operate
  - The `get_global_id(...)` API function identifies work item currently being processed by this kernel

- Groups of kernel functions are queued to operate on offload device
  - Kernel functions JIT-compiled and code cached when queued for execution
  - Sizes of vectors and arrays on which they operate specified when enqueueing work
  - Execution of kernels is parallel and asynchronous to main processors

- Complex low-level API provided for querying device capabilities, offloading work onto the device

```c
void trad_mul(int n,
    const float *a,
    const float *b,
    float *c)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] * b[i];
}

kernel void dp_mul(global const float *a,
    global const float *b,
    global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over "n" work-items
```
Example: OpenCL (3)

```c
kernel void square(global float* input, global float* output) {
    int i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```

<table>
<thead>
<tr>
<th>Input</th>
<th>6 1 1 0 9 2 4 1 1 9 7 6 1 2 2 1 9 8 4 1 9 2 0 0 7 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>36 1 1 0 81 4 16 1 1 81 49 36 1 4 4 1 81 64 16 1 81 4 0 0 49 64</td>
</tr>
</tbody>
</table>

Source: Khronos Group, OpenCL Overview

- Extended subset of C is familiar for programmers
- A cleaner model might be a purely functional language, with built-in array and vector types
  - Explicitly operate on arrays, with compiler implicitly deriving kernel functions
  - Rather than explicitly operating on kernel functions, with array dimensions defined to runtime library implicitly via OpenCL API calls
Integration With Main Operating System

• Host operating system manages offload hardware
  • Responsible for loading code onto the offload device
  • Responsible for scheduling execution of code on the offload device
  • Offload 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
Discussion and Further Reading


• Complexity versus performance trade-off in OpenCL – how much does this limit usefulness?

• How might SIMD-style processing be more cleanly incorporated into modern languages?