

Overview

- outlook – languages
 - OpenCL: common language for GPUs and multicore
 - partitioned global address space languages
 - ◆ the HPCS initiative
 - ◆ Chapel: design principles, sum array example
 - ◆ performance of X10 and Chapel on the T2 and a 4-core Intel
- outlook – processors:
 - multicore/CMT: the UltraSparc T4
 - multicore/SIMD: the Fujitsu SPARC VIIIfx
 - intermission
 - multicore/GPU: the Intel Sandy Bridge
- review of common concepts
 - parallel programming
 - architectural considerations
 - programming paradigms

OpenCL Kernel Launch

- must explicitly create device handle, compute context and work-queue, load and compile the kernel, and finally enqueue it for execution

```
clGetDeviceIDs(..., CL_DEVICE_TYPE_GPU, 1, &device, ...);
context = clCreateContext(0, 1, &device, ...);
queue = clCreateCommandQueue(context, device, ...);

program = clCreateProgramWithSource(context, "reverseArray.cl",
clBuildProgram(program, 1, &device, ...);

reverseArr_k = clCreateKernel(program, "reverseArray", ...);
clSetKernelArg(reverseArray_k, 0, sizeof(cl_mem) &a_d);
clSetKernelArg(reverseArray_k, 0, sizeof(int) &N);
cnDimension = 1; cnBlockSize = blockSize;
clEnqueueNDRangeKernel(queue, reverseArray_k, 1, 0,
&cnDimension, &cnBlockSize, 0, 0, 0);
```
- note: CUDA host code is compiled into `.cubin` intermediate files which follow a similar sequence
- for usage on normal core (`CL_DEVICE_TYPE_CPU`), a `WorkItem` corresponds to an item in a work queue that a number of (kernel-level) threads get work from
 - compiler may aggregate these to reduce overheads

The Open Compute Language for Devices and Regular Cores

- open standard – not proprietary like CUDA; based on C (no C++)
- design philosophy: treat GPUs and CPUs as peers, data- and task- parallel compute model
- similar execution model to CUDA:
 - NDRange (CUDA grid): operates on `__global` data, units within cannot synch.
 - WorkGroup (CUDA block): units within can use `__local` data (CUDA `__shared__`), to synch.
 - WorkItem (CUDA thread): indpt. unit of execution, also has `__private` data
- example kernel:

```
__kernel void reverseArray(__global int *a_d, int N) {
    int idx = getGlobalId(0);
    int v = a[N-idx-1]; a[N-idx-1] = a[idx]; a[idx] = v;
}
```
- recall that in CUDA, we could launch as `reverseArray<<<1,blockSize>>>(a_d, N)`, but in OpenCL...

The High Productivity Computer Systems Initiative

- DARPA project (2003–) to provide “a new generation of economically viable high productivity computing systems ... that double in productivity (or value) every 18 months” HPCS:Kepner&Coster
 - envisions large-scale systems with aggressively multicore components
- P is for productivity (performance, programmability, portability & robustness
 - measure/predict the ease or difficulty of developing HPC applications
- introduce a variety of experimental languages:
 - X10 (IBM)
 - Chapel (Cray)
 - Fortress (Sun - no longer supported by DARPA)(until late 2008) implementation was in a poor state (very slow to compile & run, only support for shared memory targets; inadequate documentation)

Chapel: Design Principles

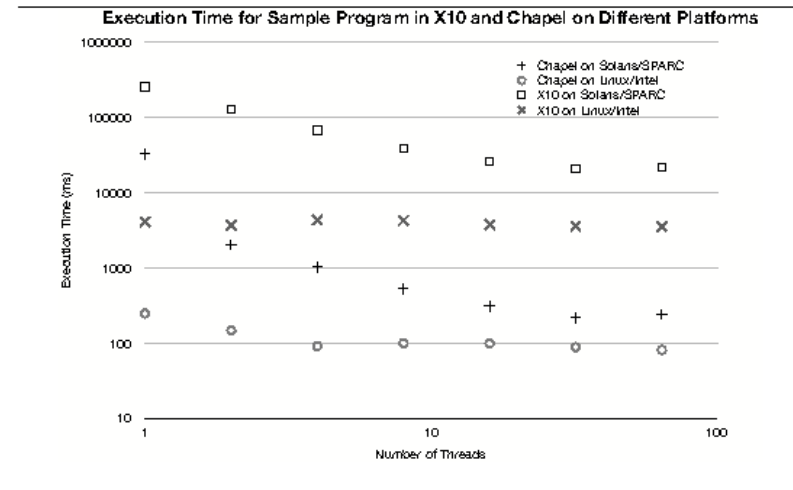
- object-oriented (Java-like syntax, but influenced by ZPL & HPF)
- supports exploratory programming
 - implicit (statically-inferable) types, run-time settable parameters (`config`), implicit `main` and module wrappings
- high-level abstractions for fine-grained parallel programming
 - `cforall`, `atomic` code sections; `sync` variables
 and for coarse grained parallel programming
 - tasks (`cobegin` block)
 - `locales` (UMA places to run tasks)
 - (index) `domains` used to specify arrays, iteration ranges
 - distributions (mappings of domains to locales)
 - claim to drastically reduce code size over MPI programs
- more info on home page; current implementation is in C++
- X10 is very similar; implementation improved over recent years

Sum Array in Chapel

```

use Time;
config const size: int = 5*1000*1000;
class ArraySum {
  var array: [0 .. size-1] int = 1;
  var currentSum: sync int = 0;
  def sum(numThreads: int) {
    var blockSize: int = size / numThreads;
    coforall i in [0 .. numThreads - 1] {
      var partSum: int = 0;
      forall j in [i*blockSize .. (i+1)*blockSize-1]
        do partSum += array(j);
      currentSum += partSum;
    } } }
var theArray: ArraySum = new ArraySum();
var sumTimer: Timer = new Timer();
for i in [0 .. 6] {
  theArray.currentSum.writeXF(0);
  sumTimer.start();
  theArray.sum(2**i);
  writeln(2**i, " threads ", sumTimer.elapsed(milliseconds), " ms ");
  sumTimer.clear();
}
    
```

Performance of X10 and Chapel on the T2 and Intel (dated)



credits: Tom Gilligan, Summer Scholar, 2008 (Sum Array program & evaluation)

The UltraSPARC T4 8-way CMP / 8-way CMT Processor

- released Sep 2011; note that the T3 (2010) was a 16-way CMP!
- optimized for single-thread performance (5× of 1.65GHz T3): (heresy!)
 - 3.0 GHz, dual instruction issue (no hardware thread groups), out-of-order execution
 - 2 integer execution units/core; fused f.p. multiply-add
 - cores are paired, enabling sharing of resources (e.g. 1st 14 stages of the execution pipeline)
 - ◆ this permits a 4 × 9 crossbar; 16KB L1\$ (D\$), 128KB L2\$ (not shared), 4MB L3\$ (diagram, courtesy Oracle)
- enhanced cryptographic accelerators; now accessible by user-level instructions!
- motivation: maintain application niche of T2/T3 while performing on more diverse (e.g. f.p. intensive) workloads
 - or were there scalability issues with 16 cores?
- power-hungry: single socket system 240W (2× that of T2)
- can be packaged in a 'supercluster': ≤ 16 T4s connected by fast internal network (Infiniband) in a single cabinet

The Octocore/SIMD SPARC64 VIIIfx

- overall layout (note: no CMT) ref: Maruyama et al, Micro 2009
- for petascale computing (power efficient - 58W, 128 GFLOPs d.p. @ 2GHz)
- very large register sets:
 - 8 × 32 general-purpose register windows *plus* an extra 32 'global' registers
 - 256 f.p. registers (+ 48 'renaming'); a 'prefix' instrn. specifies the upper 3 bits
- 2 load-store units, 4 f.p. units with fused multiply-add (8 FLOPs / cycle)
- highly superscalar: 2 load/store and 2 f.p. per cycle (include 2-way SIMD)
- the 5MB L2\$ is 9-way set-associative; each set can be configured as hardware- (i.e. normal) or software-controlled
 - latter is effectively fast local memory (c.f. GPU)
 - useful for streaming data (little temporal locality) or for hand-optimized algs.
- fast hardware synchronization between cores
- review: why so fast and yet power efficient?

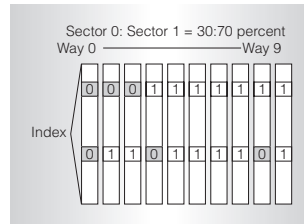


Figure 5. Sector cache. At each cache index, sector 0 contains 3 cache ways and sector 1 contains 7 cache ways.

Multicore/Manycore Outlook

- diversity in approaches; post-RISC ideas will still be tried
 - "two strong oxen or 1024 chickens" (Seymour Cray, late 80's) debate to continue
- energy issue will generally increase in prominence
 - however, engineering effort can overcome inherently inefficient design
- increasing level of integration expected (power & time savings vs. fewer cores)
- overcoming the memory wall continues to be a major factor in design
 - increasing portion of design effort and chip area devoted to data movement
- predict the coherency wall will begin to bite at 32 cores
 - long-term future for inter-socket coherency?
- two important scenarios: everyday and high-end
- will programming languages deliver the *silver bullet*? (or cover devices & cores?)
 - PGAS languages can deal with distributed memory, both within and across (network-connected) chips
 - may need hierarchical notions of locality (places)

The Intel Sandy Bridge Processor: GPU within Main Chip

- refs: RealWorldTech and Hardware Secrets articles
- 3.4 GHz; 144 (f.p.) & 160 (int) register files; AVX (SIMD) operations (3 units) - 8 d.p. FLOPs/cycle
- high memory bandwidth; support large number of 'in-flight' memory operations
- microarchitecture:
 - a ring bus (like Cell BE): request, data, ack., and snoop rings
 - any CPU can request any LL\$ bank
 - CPUs send graphics programs to GPU
- graphics processor
 - not a general-purpose GPU; ≤ 1.5 GHz; ≤ 12 UEs (≡ SPs); can use LL\$ (L3\$)
- CPU/Graphics power budgeting - not possible with external GPU
 - Turbo Boost: can over-clock CPU for limited periods (till temperature rises)
- by having the device integrated on chip, can avoid host ↔ device transfers, share resources (main memory and LL\$), and optimize power usage

Review of Common Concepts: Parallel Programming

- top 10 reasons why parallel programming is hard
 - Amdahl's Law (and for the heterogeneous case!), communication, load balancing, synchronization; portability; data races, deadlocks, incorrect data access, sheer size of computation, non-determinism. . .)
 - need to isolate the simplest case of failure for debugging
 - need systematic approaches to design and self-testing (Concurrent UML, Parallel Application Design Layers, Predicate Breakdown Structure); use of common design patterns
 - refactoring for parallel execution can be hard!
- importance of profiling (with hardware performance counters) to understand where and why performance bottlenecks
 - requires use hardware performance counters
 - constructive and destructive sharing applies to both T2 and GPUs

Review of Common Concepts: Overall Architectural Considerations

- concurrency can hide latencies (e.g. L1/L2\$ misses) and improve load balance (hardware threading, GPU thread blocks)
also permits (threads executing on) multiple cores – for speedup
 - SIMD execution: limited to data parallelism, but simple model enables highly efficient (and parallel) design
- efficiency and power considerations:
 - $P \propto CV^2f \propto Cf^3$, $R = pf$ (ideally)
 - use low f ; only use what you need (power throttling - SCC power and frequency islands)
 - post-RISC designs have reached their limits; tradeoff between complexity and large core counts
 - heterogeneous multicore to efficiently execute 'simple' services

Review of Common Concepts: Memory Systems and the OS

- memory system design dominant across all forms of multicore
 - need scalable, high-bandwidth networks (e.g. T2 crossbar, SCC mesh, T2/GPU/SCC memory banks)
 - efficient use of network important (use 'burst mode', avoid collisions)
 - ◆ packet and circuit-based switching; wormhole routing
 - ◆ energy issue in data transfer becomes increasingly dominant
 - ◆ *cache coherency considered harmful*: many $O(p^2)$ effects
 - ◆ especially atomic operations (time and power)
 - ★ locking: scalable algorithms avoid > 1 thread contending for each lock
 - ★ tree barrier is scalable because it avoids atomics
- operating system issues become critical for multicore (more resources to manage, need to offer scalable services)
- virtualization and multicore: can have separate OS on each vCPU
 - two-level memory mapping ensures isolation between OS, even though sharing caches & TLBs
 - T2 Crypto Units requires hypervisor; approach was abandoned in the T4

Review of Common Concepts: Programming Paradigms

- OpenMP: permits incremental parallelization
 - data sharing between threads is important (carefully distinguish global vs thread-level data),
tuning via `nowait`, `schedule` clauses
 - `critical` / `atomic` constructs for shared data objects
 - notion of fork-join threads develops into thread pools and work queues
- CUDA kernels similar to OpenMP parallel regions:
 - use thread id to determine what data to operate on (global indexing), using e.g. the `block` distribution scheme
 - synchronization only *within* a thread block – but much faster!
 - however, need to transfer data to/from device (overhead)
 - reducing memory accesses (matrix multiply) is the key to good performance
 - reductions require extra care in all paradigms

Review of Common Concepts: Programming Paradigms (II)

- message passing:
 - distribution schemes are basically fixed
 - need to find local address corresponding to a global element, using the process id
 - messages can also be used for synchronization
- transactional memory as alternative to lock-based shared object protection
 - STM requires an atomic increment, plus a lock acquisition for every variable written to
 - HTM based on extended coherency protocol: T-bit in cache for transactional variables, with atomic write-back
 - can be generalized to speculative execution: also 2 copies of all speculative data (register checkpoints in Rock, speculative write buffers in Hydra) which can be atomically committed