Wrapup and Open Issues

Kevin Skadron
University of Virginia Dept. of Computer Science
LAVA Lab
Outline

Objective of this segment: explore interesting open questions, research challenges

- CUDA restrictions (we have covered most of these)
- CUDA ecosystem
- CUDA as a generic platform for manycore research
- Manycore architecture/programming in general
CUDA Restrictions – Thread Launch

- Threads may only be created by launching a kernel
- Thread blocks run to completion—no provision to yield
- Thread blocks can run in arbitrary order
- No *writable* on-chip persistent state between thread blocks

Together, these restrictions enable
- Lightweight thread creation
- Scalable programming model (not specific to number of cores)
- Simple synchronization

© Kevin Skadron, 2008
CUDA Restrictions – Concurrency

- Task parallelism is restricted
- Global communication is restricted
- No writable on-chip persistent state between thread blocks
- Recursion not allowed
- Data-driven thread creation, and irregular fork-join parallelism are therefore difficult

Together, these restrictions allow focus on
- perf/mm²
- scalability
CUDA Restrictions – GPU-centric

- No OS services within a kernel
  - e.g., no malloc

- CUDA is a manycore but *single-chip* solution
  - Multi-chip, e.g. multi-GPU, requires explicit management in host program, e.g. separately managing multiple CUDA devices

- Compute and 3D-rendering can’t run concurrently (just like dependent kernels)
CUDA Ecosystem Issues

- These are really general parallel-programming ecosystem issues
  - #1 issue: need parallel libraries and skeletons
    - Ideally the API is platform independent
  - #2 issue: need higher-level languages
    - Simplify programming
    - Provide information about desired outcomes, data structures
    - May need to be domain specific
    - Allow underlying implementation to manage parallelism, data
    - Examples: D3D, MATLAB, R, etc.
  - #3 issue: debugging for correctness and performance
    - CUDA debugger and profiler in beta, but…
    - Even if you have the mechanics, it is an information visualization challenge
    - GPUs do not have precise exceptions
Outline

- CUDA restrictions
- CUDA ecosystem
- CUDA as a generic platform for manycore research
- Manycore architecture/programming in general
CUDA for *Architecture* Research

CUDA is a promising vehicle for exploring many aspects of parallelism at an interesting scale on real hardware.

CUDA is also a great vehicle for developing good parallel benchmarks.

What about for architecture research?

- We can’t change the HW
- CUDA is good for exploring bottlenecks
  - Programming model: what is hard to express, how could architecture help?
  - Performance bottlenecks: where are the inefficiencies in real programs?
- But how to evaluate benefits of fixing these bottlenecks?
  - Open question
  - Measure cost at bottlenecks, estimate benefit of a solution?
    - Will our research community accept this?
Programming Model

General manycore challenges...

Balance flexibility and abstraction vs. efficiency and scalability
- MIMD vs SIMD, SIMT vs. vector
- Barriers vs. fine-grained synchronization
- More flexible task parallelism
- High thread count requirement of GPUs

Balance ease of first program against efficiency
- Software-managed local store vs. cache
- Coherence

Genericity vs. ability to exploit fixed-function hardware
- Ex: OpenGL exposes more GPU hardware that can be used to good effect for certain algorithms—but harder to program

Balance ability to drill down against portability, scalability
- Control thread placement

Challenges due to high off-chip offload costs
- Fine-grained global RW sharing
- Seamless scalability across multiple chips ("manychip"?), distributed systems
Scaling

- How do we use all the transistors?
  - ILP
  - More L1 storage?
  - More L2?
  - More PEs per “core”?
  - More cores?
  - More true MIMD?
  - More specialized hardware?

- How do we scale when we run into the power wall?
Thank you

Questions?

Additional slides
CUDA Restrictions – Task Parallelism (extended)

- Task parallelism is restricted and can be awkward
  - Hard to get efficient fine-grained task parallelism
  - Between threads: allowed (SIMT), but expensive due to divergence and block-wide barrier synchronization
  - Between warps: allowed, but awkward due to block-wide barrier synchronization
  - Between thread blocks: much more efficient, code still a bit awkward (SPMD style)
  - Communication among thread blocks inadvisable
    - Communication among blocks generally requires new kernel call, i.e. global barrier
    - Coordination (i.e. shared queue pointer) ok
    - No writable on-chip persistent state between thread blocks

- These restrictions stem from focus on perf/mm²
  - support scalable number of cores
CUDA Restrictions – HW Exposure

CUDA is still low-level, like C – good and bad
- Requires/allows manual optimization
- Manage concurrency, communication, data transfers
- Manage scratchpad

Performance is more sensitive to HW characteristics than C on a uniprocessor (but this is probably true of any manycore system)
- Thread count must be high to get efficiency on GPU
- Sensitivity to number of registers allocated
  - Fixed total register file size, variable registers/thread, e.g. G80: # registers/thread limits # threads/SM
  - Fixed register file/thread, e.g., Niagara: small register allocation wastes register file space
  - SW multithreading: context switch cost f(# registers/thread)

Memory coalescing/locality
- Many more interacting hardware sensitivities (# regs/thread, # threads/block, shared memory usage, etc.)—performance tuning is tricky

…These issues will arise in most manycore architectures

Need libraries and higher-level APIs that have been optimized to account for all these factors
More re: SIMD Organizations

SIMT: independent threads, SIMD hardware
- First used in Solomon and Illiac IV
- Each thread has its own PC and stack, allowed to follow unique execution path, call/return independently
  - Implementation is optimized for lockstep execution of these threads (32-wide warp in Tesla)
  - Divergent threads require masking, handled in HW
- Each thread may access unrelated locations in memory
  - Implementation is optimized for spatial locality—adjacent words will be coalesced into one memory transaction
  - Uncoalesced loads require separate memory transactions, handled in HW
- Datapath for each thread is scalar
More re: SIMD Organizations (2)

**Vector**
- First implemented in CDC, commercial big time in MMX/SSE/Altivec/etc.
- Multiple lanes handled with a single PC and stack
  - Divergence handled with predication
  - Promotes code with good lockstep properties
- Datapath operates on vectors
  - Computing on data that is non-adjacent from memory requires vector gather if available, else scalar load and packing
  - Promotes code with good spatial locality
- Chief difference: more burden on software

*What is the right answer?*
Other Manycore PLs

- Data-parallel languages (HPF, Co-Array Fortran, various data-parallel C dialects)
- DARPA HPCS languages (X10, Chapel, Fortress)
- OpenMP
- Titanium
- Java/pthreads/etc.
- MPI
- Streaming

Key CUDA differences/notables
- Virtualizes PEs
- Supports offload model
- Scales to large numbers of threads, various chip sizes
- Allows shared memory between (subsets of) threads