Inside X10

Olivier Tardieu, *Benjamin Herta*,
Dave Grove, Vijay Saraswat

IBM T.J. Watson Research Center

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002, by the Department of Energy, and by the Air Force Office of Scientific Research.

Parts of this tutorial are based upon material by Igor Peshansky and David Cunningham.

Goals of the Tutorial

- ~~Learning to code in X10~~
  - see http://x10-lang.org/documentation/tutorials/apgas-programming-in-x10-24.html

- Overview of X10 implementation

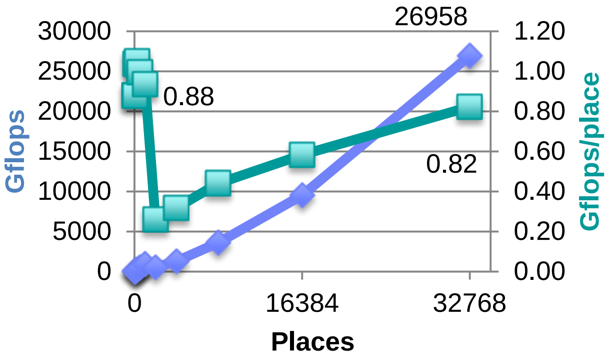
- Why did we do it?
- How did we do it?
- What did we learn?
- Where are we now?
- Where did we innovate?
- Where can we do better?
- Where do we go next?

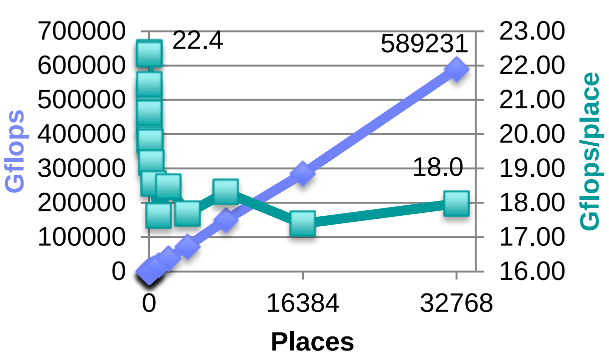
- Where do *you* go next?

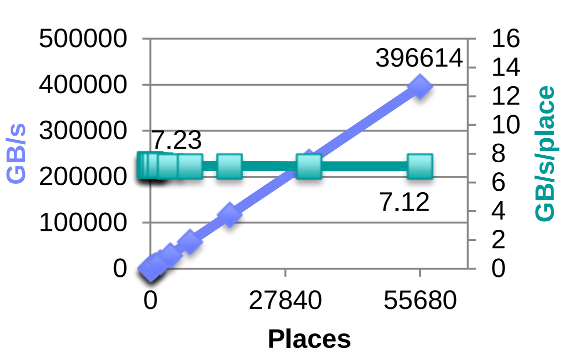
Background

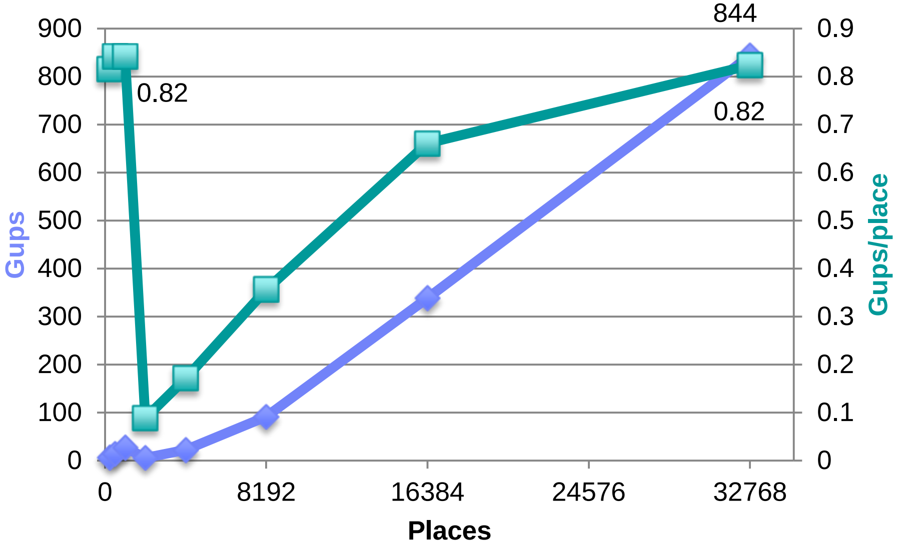
- X10 is
  - a programming language
  - an open-source tool chain
    - compiles X10 to C++ or Java
  - a growing community
- X10 has been developed since 2004
  - at IBM Research with support from DARPA, DoE, and AFOSR
- X10 tackles the challenge of programming at *scale*
  - first HPC, then clusters, now cloud
  - scale out: run across many distributed nodes
  - scale up: exploit multi-core and accelerators
  - elasticity and resilience
  - double goal: *productivity* and *performance*

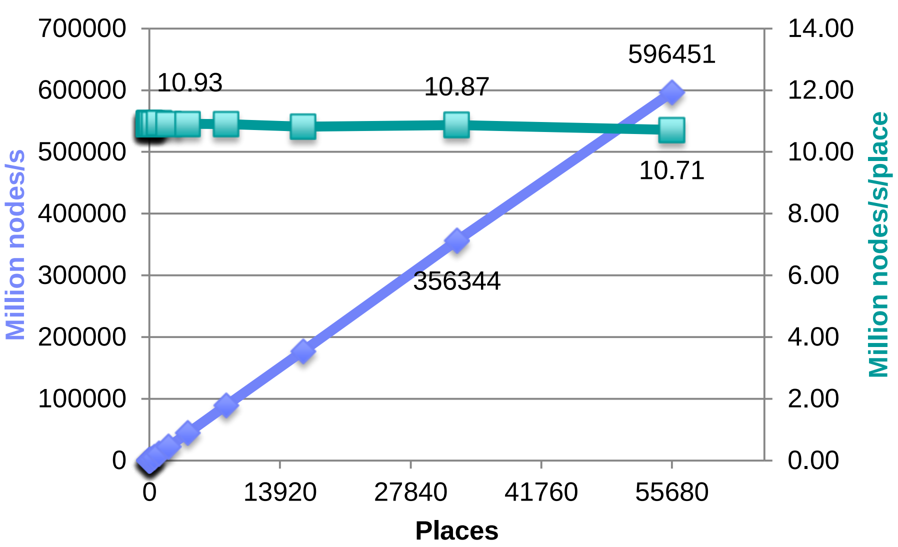
HPC Challenge 2012 – X10 at Petascale – Power 775


G-FFT


G-HPL


EP Stream (Triad)


G-RandomAccess


UTS

X10 Community and Applications

- X10 applications and frameworks
  - ANUChem, [Milthorpe IPDPS 2011], [Limpanuparb JCTC 2013]
  - ScaleGraph [Dayarathna et al X10 2012]
  - Invasive Computing [Bungartz et al X10 2013]
  - XAXIS [Suzumura et al X10 2012]; used in Megaffic (IBM Mega Traffic Simulator)
  - Global Matrix Library: distributed sparse and dense matrices
  - Global Load Balancing [Zhang et al PPAA 2014]

- X10 as a coordination language for scale-out
  - SatX10 [Bloom et al SAT'12 Tools]
  - Power system contingency analysis [Khaitan & McCalley X10 2013]

- X10 as a target language
  - MatLab [Kumar & Hendren X10 2013]
  - StreamX10 [Wei et al X10 2012]

Highlights

Pragmatic Design Choices

- Traditional sequential imperative object-oriented core (Java)
  - have few simple innovations where they can really make a difference
    - rich data types with a triple goal: performance, productivity, safety
- Explicit concurrency and distribution
  - program defines asynchronous tasks and synchronization
    - compiler and runtime map tasks to execution units and handle scheduling constraints
  - program specifies where to store data and where to run code
    - compiler and runtime verify locality requirement and handle data migration
- Explicit accelerator code and explicit use
  - accelerator code is written in X10 but restricted to a sublanguage
    - compiler and runtime verify requirements and handle layout and data migration
- Portability and interoperability
  - enable design of interoperable X10 libraries
  - balance the needs of C++ and Java-based tool chains
  - multi-mode execution

Pragmatic Implementation Choices

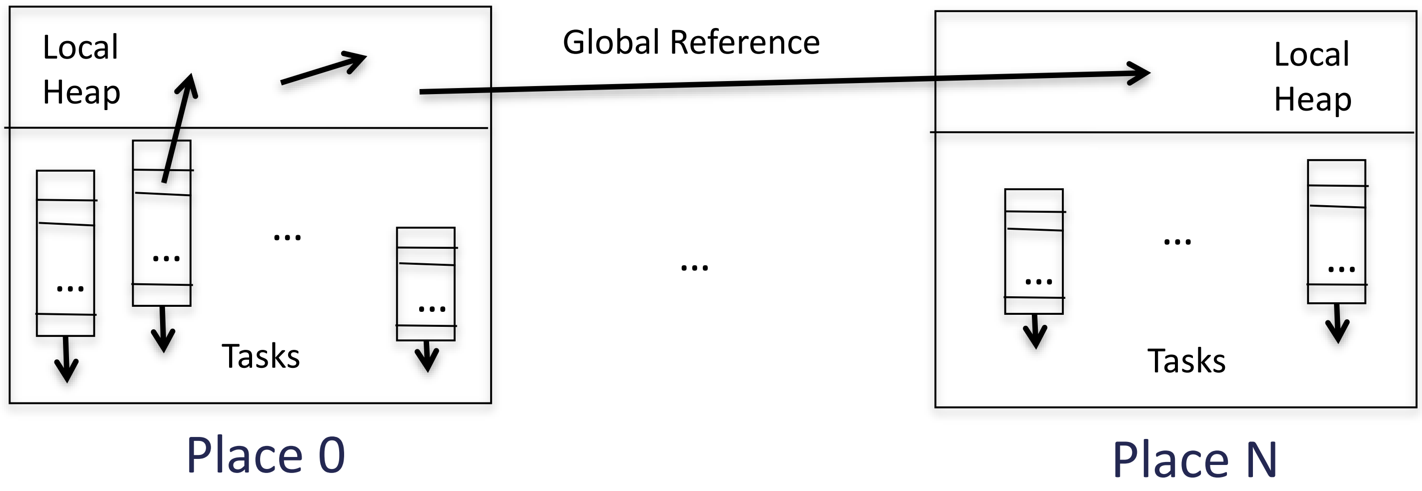
- Source-to-source compiler
  - generate Java, C++, and CUDA code
  - rely on backend to do most optimizations
- Stratification
  - lowering passes
  - implement runtime and class library mostly in X10
  - network abstraction layer
- Cooperative work-stealing schedulers
  - with and without compiler support
- Local garbage collection
  - explicit management of remote pointers
- Backend-specific representation of generic types
  - C++ templates
  - Java generics (erasure) + type descriptor objects

Outline

- Overview
  - X10 language
  - X10 tool chain
- Deep dives
  - scheduling
  - mapping X10 to C++ and Java
  - serialization
  - distribution
  - CUDA
- *PPAA 2014 talk: global load balancing*
- *PPoPP 2014 talks: scale out & resilience*

X10 Overview

Places and Tasks



Task parallelism

- **async** S
- **finish** S

Concurrency control

- **when**(c) S
- **atomic** S

Place-shifting operations

- **at**(p) S
- **at**(p) e

Distributed heap

- **GlobalRef**[T]
- **PlaceLocalHandle**[T]

Idioms

- Remote procedure call

```
v = at(p) evalThere(arg1, arg2);
```

- Active message

```
at(p) async runThere(arg1, arg2);
```

- Divide-and-conquer parallelism

```
def fib(n:Int):Int {
  if(n < 2) return n;
  val f1:Int;
  val f2:Int;
  finish {
    async f1 = fib(n-1);
    f2 = fib(n-2);
  }
  return f1 + f2;
}
```

- SPMD

```
finish for(p in Place.places()) {
  at(p) async runEverywhere();
}
```

- Atomic remote update

```
at(ref) async atomic ref() += v;
```

- Computation/communication overlap

```
val acc = new Accumulator();
while(cond) {
  finish {
    val v = acc.currentValue();
    at(ref) async ref() = v;
    acc.updateValue();
  }
}
```

Example: BlockDistRail.x10

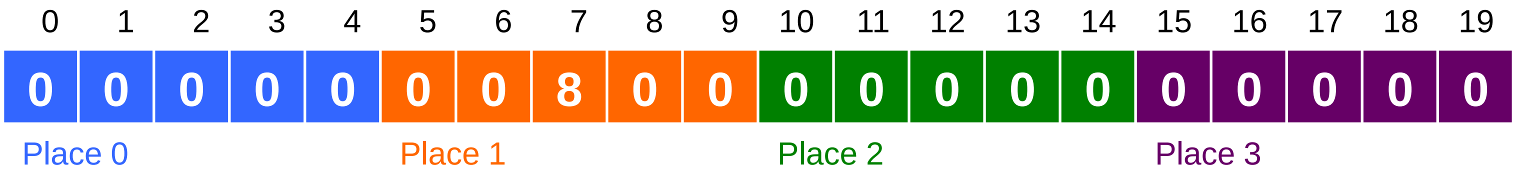
```
public class BlockDistRail[T] {
  protected val sz:Long; // block size
  protected val raw:PlaceLocalHandle[Rail[T]];

  public def this(sz:Long, places:Long){T haszero} {
    this.sz = sz;
    raw = PlaceLocalHandle.make[Rail[T]](PlaceGroup.make(places), ()=>new Rail[T](sz));
  }

  public operator this(i:Long) = (v:T) { at(Place(i/sz)) raw()(i%sz) = v; }

  public operator this(i:Long) = at(Place(i/sz)) raw()(i%sz);

  public static def main(Rail[String]) {
    val rail = new BlockDistRail[Long](5, 4);
    rail(7) = 8; Console.OUT.println(rail(7));
  }
}
```



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0
Place 0					Place 1					Place 2					Place 3				

X10 Tool Chain Overview

X10 Tool Chain

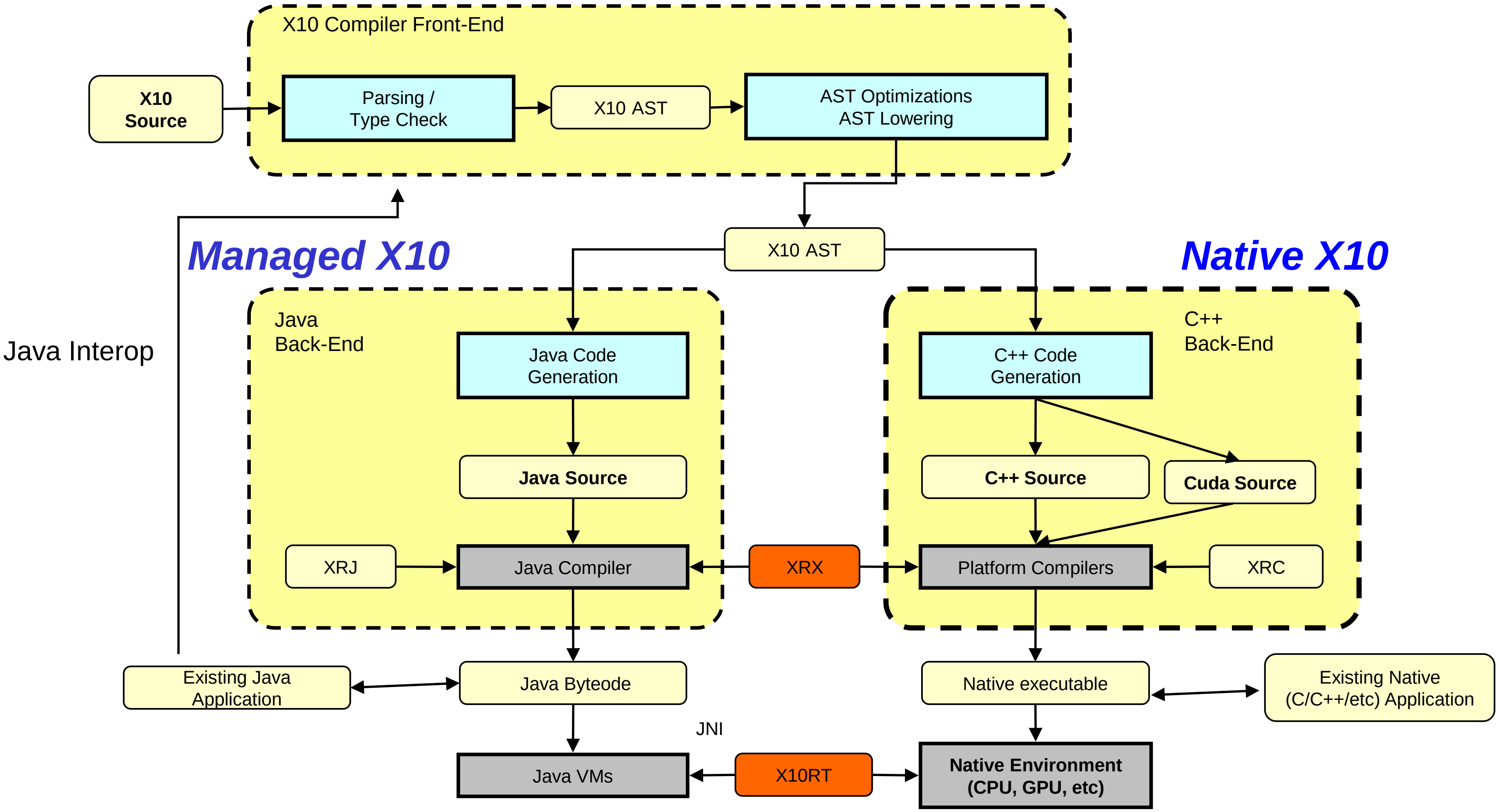
- Eclipse Public License

- “Native” X10 implementation
  - C++ based; CUDA support
  - distributed multi-process (one place per process + one place per GPU)
  - C/POSIX network abstraction layer (X10RT)
  - x86, x86_64, Power; Linux, AIX, OS X, Windows/Cygwin, BG/Q; TCP/IP, PAMI, MPI

- “Managed” X10 implementation
  - Java 6/7 based; no CUDA support
  - distributed multi-JVM (one place per JVM)
  - pure Java implementation over TCP/IP or using X10RT via JNI (Linux & OS X)

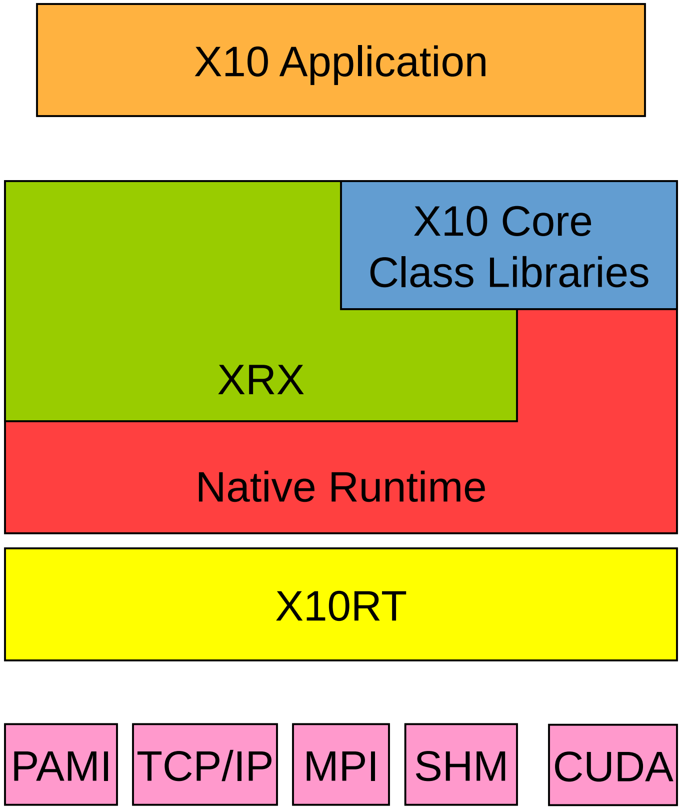
- X10DT (Eclipse-based IDE) available for Windows, Linux, OS X
  - supports many core development tasks including remote build & execute facilities

X10 Compilation and Execution



X10 Runtime

- X10RT (X10 runtime transport)
  - core API: active messages
  - extended API: collectives & RDMAs
    - emulation layer
  - two versions: C (+JNI bindings) or pure Java
- Native runtime
  - processes, threads, atomic ops
  - object model (layout, RTTI, serialization)
  - two versions: C++ and Java
- XRX (X10 runtime in X10)
  - async, finish, at, when, atomic
  - X10 code compiled to C++ or Java
- Core X10 libraries
  - x10.array, io, util, util.concurrent



Scheduling

Summary

- Goal: support programming model
  - programmer focuses on tasks not resources
  - compiler & runtime map many many tasks to few execution units (cores)
    - *in each X10 place*
- Solutions?
  - OS scheduler: pthreads
    - map one task to one pthread
    - drawback: too much overhead
  - library: qthreads…
    - map one task to one light-weight thread
    - drawback: not portable & no understanding of X10 task dependencies
- Our solution
  - cooperative work-stealing schedulers
    - biased towards fork-join task graphs but general purpose

Work-Stealing Schedulers

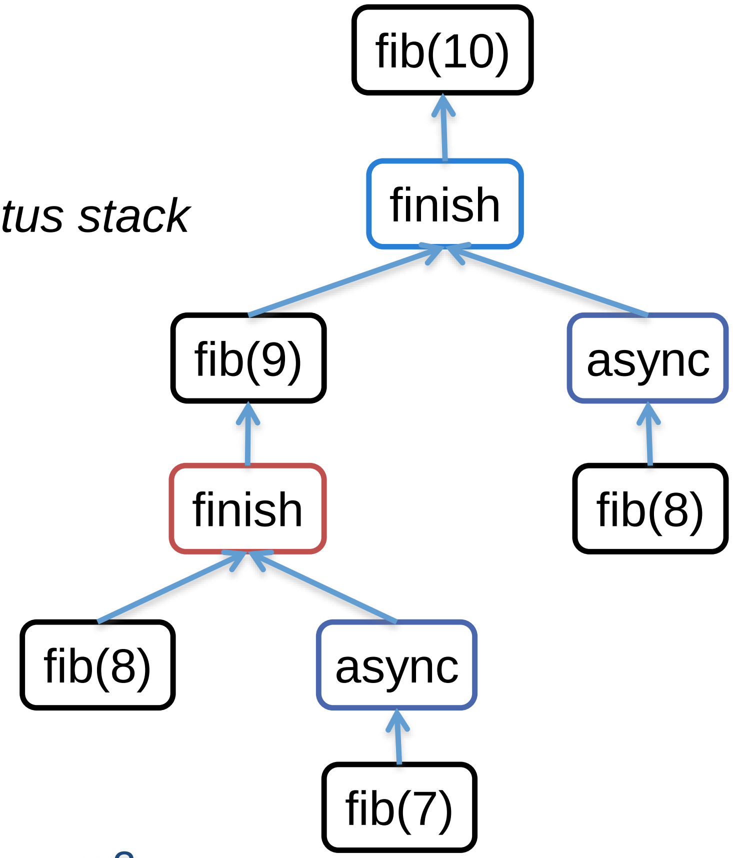
- Principles
  - dynamic load balancing
  - decentralized load balancing
  - reactive load balancing

- Basics
  - a pool of worker threads
  - per-worker deque (double-ended queue) of pending jobs
  - worker pushes and pops jobs from its deque
  - idle worker steals job from random worker from bottom of deque

- Benefits
  - low contention ➔ high utilization
  - low overhead ➔ even for fine-grained tasks

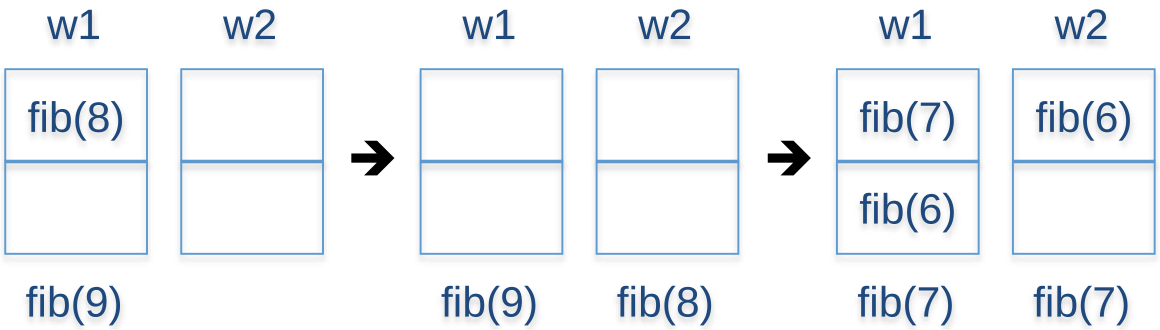
Example: Fibonacci

```
static def fib(n:Long) {
  if (n < 2) return n;
  val x:Long; val y:Long;
  finish {
    async x = fib(n-2);
    y = fib(n-1);
  }
  return x+y;
}
```



Cactus stack

- Computing fib(10)



Synchronization?

- Work stealing is great for non-blocking fork-join task graphs
  - task F pushes task T1,…, Tn, last T task pushes task J

- X10 has many kinds of synchronizations
  - sequencing: finish
  - critical section: atomic
  - condition: when
  - distribution: at
  - external event: I/O, sleep…

- How to implement waiting while maintaining utilization?
  - local finish: execute subtask if any ➔ infrequent loss of parallelism
  - atomic: wait ➔ brief loss of parallelism
  - special cases (guarded tasks, futures, etc): execute prereq ➔ no loss of parallelism
  - *other cases: dynamic context switch*

Dynamic Context Switches and Scheduling Policies

- Option 1: dynamic thread creation
  - benefit: OS & VM are very good at context switches
  - drawback: breaks the simple 1 core == 1 thread == 1 deque map
  - drawback: requires eager thread creation
    - if task A and B might need to be interleaved then A and B must run in separate threads

- Option 2: continuations
  - drawback: continuation machinery (performance cost, portability)
  - benefit: choice of work-first or help-first schedule

- Help-first policy (Java Fork/Join)
  - worker pushes spawned task to deque and continues executing parent

- Work-first policy (Cilk++)
  - worker pushes continuation of parent task to deque and executes spawned task

Our Take: Three Schedulers

- Variable-size thread pool (option 1)
  - pure runtime solution & portable
  - context switch implemented by increasing thread count
- Offline CPS transform (option 2)
  - X10 frontend compiler implements CPS transform
  - portable: CPS implemented as source-to-source rewriting
  - but performance is poor with Java tool chain
    - relies on C++-specific optimizations of continuations
- Lazy continuation extraction (option 2)
  - specific to JikesRVM
  - leverages stack-walking API & fast exception handling
  - requires offline light-weight code instrumentation

Scheduling via Offline CPS transform

- X10-to-X10 program transformation
  - implement cactus stack frames
    - synthesize a frame class for each basic block
      - up field, fields for locals and parameters
      - run instance method encapsulates basic block body
  - implement continuations
    - pc field in each frame
    - top-level switch(pc) statement in each run method
  - rewrite async into push and pop operations on the deque

- X10 work-stealing scheduler library
  - implement main worker loop
  - maintain cactus stack and deques
  - keep count of subtasks of finish frame
  - implement frame reallocation

Main Loop

```
var k:Frame;
while ((k = findContinuation()) != null) {
  try {
    while (k != null) {
      k.run(k.pc);
      k = k.up;
      if (k != null && k instanceof FinishFrame) {
          val f = k as FinishFrame;
          f.decreaseChildrenCount();
          if (f.hasOustandingChildren()) break;
    }
  } catch (Stolen) {}
}
```

Generated Code

```
static def fib(n:Int) {
  if (n<2) return;
  finish {
    async fib(n-2);
    fib(n-1);
  }
}
```

➔

```
static def _fib(up:Frame, n:Int) {
  new _fib(up, n).run(0);
}
class _fib extends RegularFrame {
  val n:Int;
  def this(up:Frame, n:Int) {super(up); this.n=n;}
  def run(pc:Int) {
    switch(pc) {
    case 0:
      this.pc = 1;
      if (this.n<2) return;
      new _fib_finish(this).run(0);
    case 1:
  }
}
…
```

Optimizations

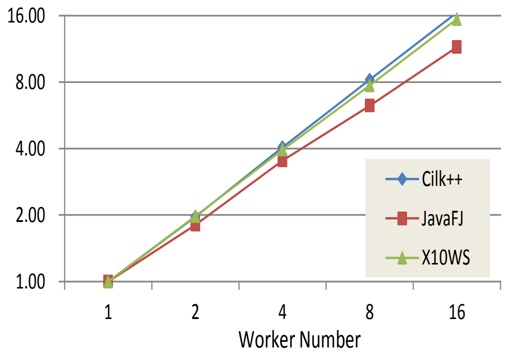
- Portable optimizations
  - only rewrite concurrent code
  - only count stolen tasks
  - devirtualization & inlining
    - recombine frames of the same method

- Advanced optimizations (C++)
  - lazy initialization of frame fields
    - avoid the cost of zeroing field values
  - speculative stack allocation
    - runtime allocate frame objects on the stack
    - thief copies stack frames to heap
    - victim waits for copy to complete before aborting
      - victim handles patches new frame if necessary (asynchronous initialization)

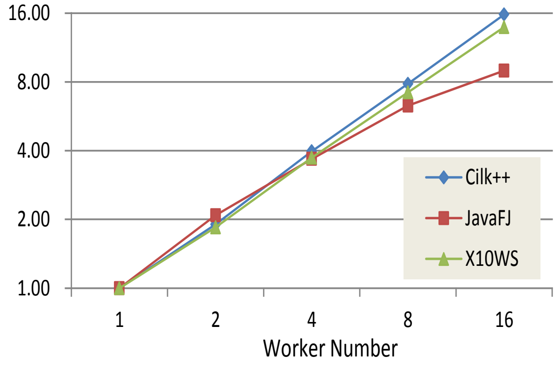
    ⇒ no heap allocation on the fast path

  ⇒ performance comparable to Cilk++

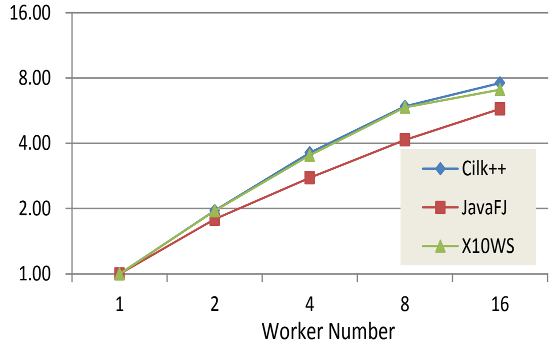
Micro Benchmarks



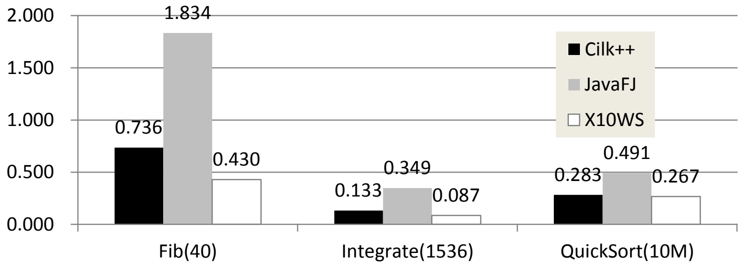
Fib speedup



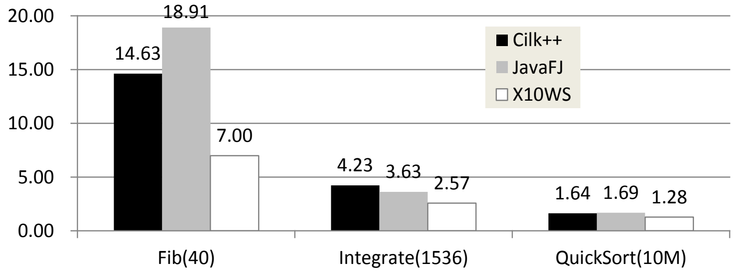
Integrate speedup



QuickSort speedup



execution time (s) with 16 threads

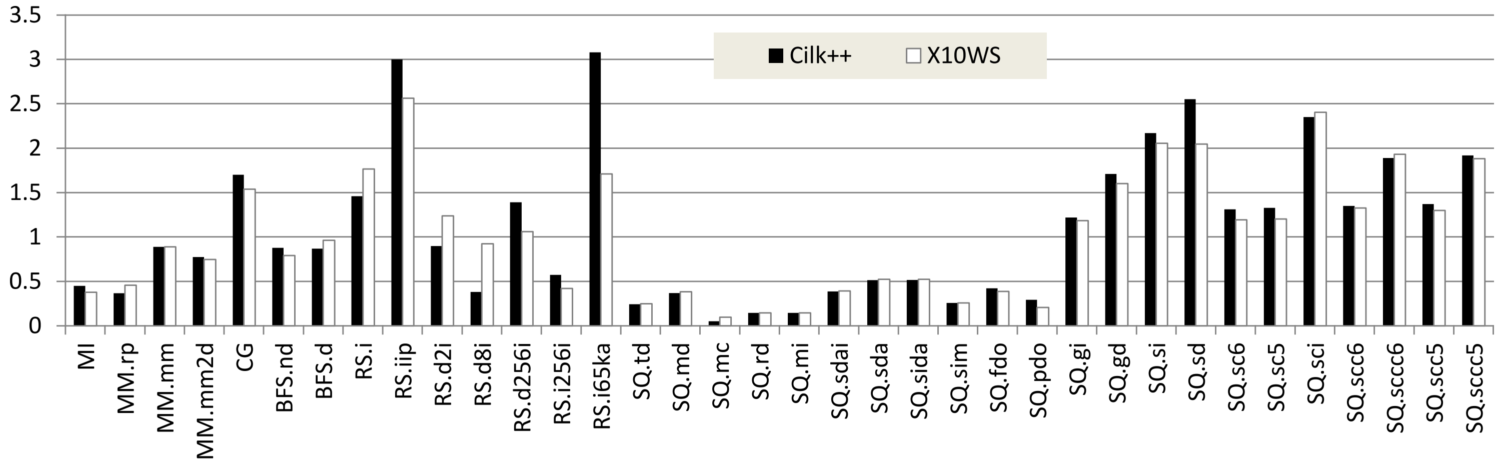


sequential overhead

X10 vs. Cilk++: PBBS Benchmarks

http://www.cs.cmu.edu/~guyb/pbbs/

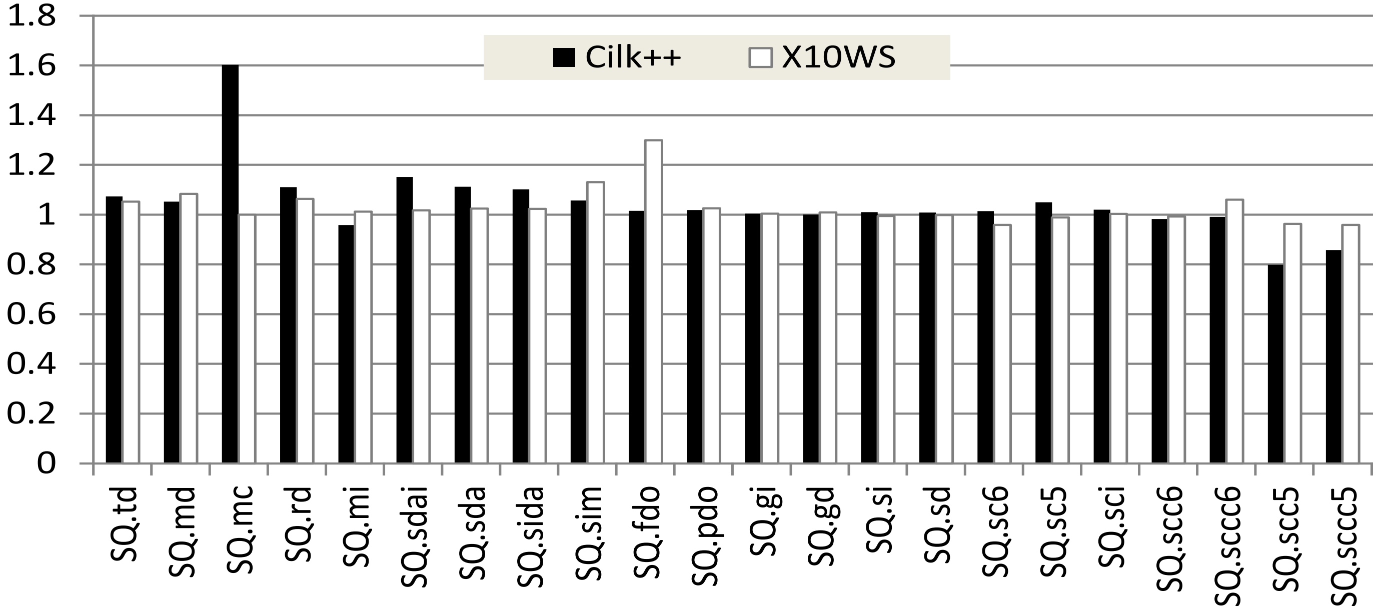
- Compared running time (s) with 16 cores



X10 vs. Cilk++: PBBS Benchmarks

http://www.cs.cmu.edu/~guyb/pbbs/

- Compared sequential overhead

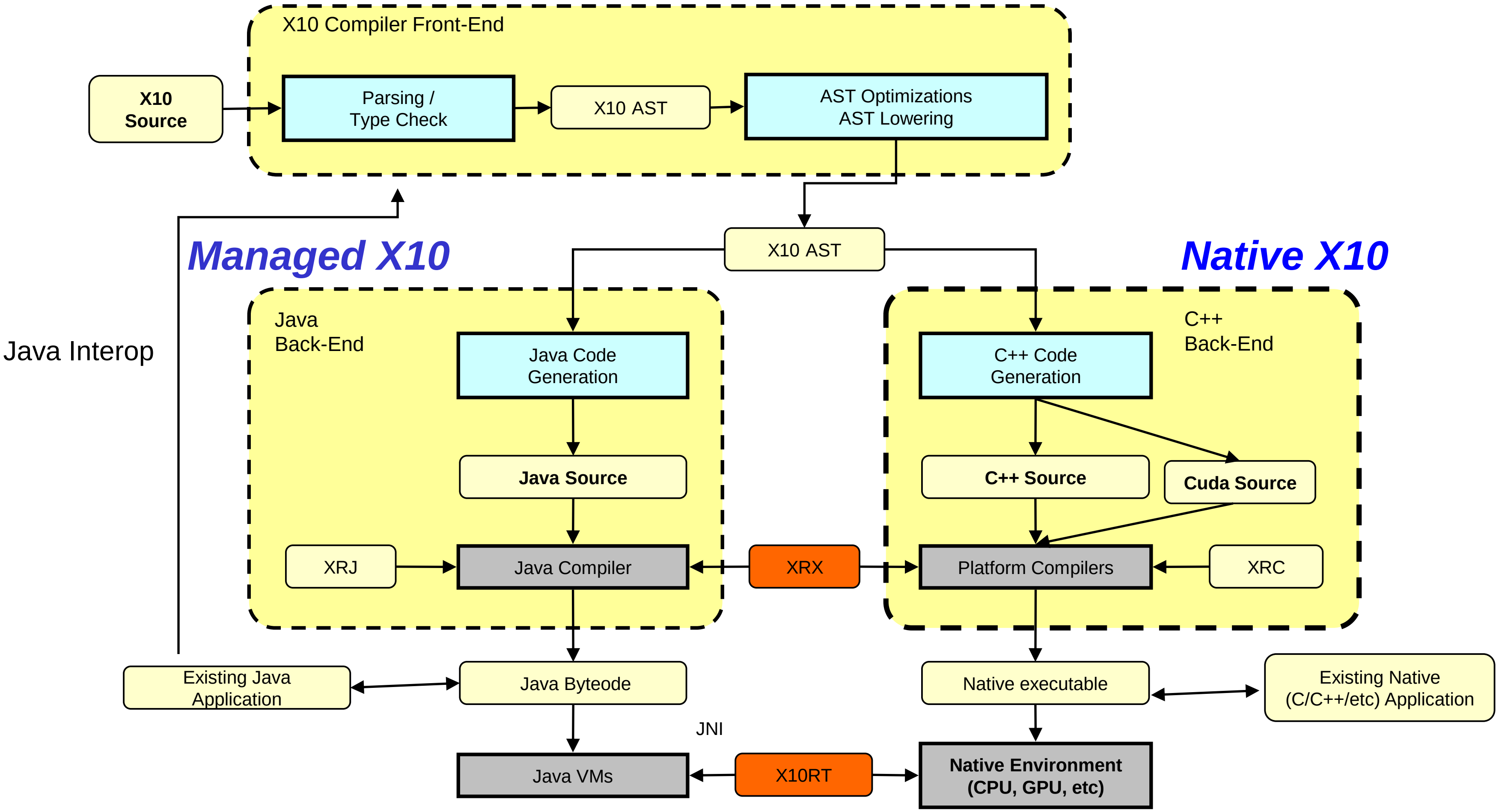


Scheduling Discussion

- What we do well
  - shared-memory work-stealing schedulers beyond fork-join task graphs
  - distributed work-stealing at scale (see PPAA 2014 talk)
- What we should do better
  - scheduling of atomic sections (for now: single lock)
  - scheduling of guarded tasks (for now: single queue)
  - busy waiting avoidance (for now: busy loop)
  - locality-aware scheduling (for now: globally help-first or work-first policy)
- Some open questions
  - user control
    - priorities, thread affinity, cancellation…
  - fairness
  - many-cores

Mapping X10 to C++ and Java

X10 Compilation and Execution (Reprise)



Implementation Strategy/Goals

- Desired features
  - Highly portable (reduce barriers to adoption/experimentation)
  - High performance possible; good out-of-the-box performance likely
  - Interoperability with existing HPC and commercial software stacks

- Pragmatic implications
  - Take source-to-source compilation approach
    - Portability (run anywhere with a JVM or C++ compiler)
    - Leverage host of traditional optimizations in JVM/C++ compilers
    - Significantly reduce implementation & maintenance cost
  - Where possible, reuse target language features
    - eg. X10 class → Java/C++ class
    - eg. X10 exceptions → Java/C++ exceptions
  - Still need optimizer in X10 compiler, but narrowly focused (next slide…)

X10 Optimizer

- Focus on high-level, X10/APGAS specific optimization
  - Communication optimizations
  - Workstealing compilation
  - Expansion of complex APGAS operations (ateach, clocks)
  - Lowering of core APGAS constructs (async, finish, at)
  - Exploitation of X10 type system (constraints, properties)
  - For loop comprehensions ➔ Simple (nested/counted) for loops
- Minimal classic optimization
  - Inlining: to increase scope for X10/APGAS optimization
  - Post-inlining cleanup (constant/copy prop, dead code/variables, etc)
- Guiding principle:
  Focus on high-impact optimizations Java/C++ compiler won’t do

Java vs. C++ as Implementation Substrate

- Java
  - Just-in-time compilation
  - Sophisticated optimizations and runtime services for OO language features
  - Straying too far from Java semantics can be quite painful
  - Implementing a language runtime in vanilla Java (no JVM extensions) is limiting
    - Restricted to Java type system & safety rules
    - No direct control on memory layout, JIT compilation decisions, etc.
    - Unable to exploit JVM implementation features (eg. JVM-supported workstealing)

- C++
  - Ahead-of-time compilation
  - Minimal optimization of OO language features
  - Implementing language runtime layer
    - Ability to write low-level/unsafe code (flexibility)
    - Much fewer built-in services to leverage
    - Much more direct control possible (with explicit effort)

Summary of X10 Object Model

- X10 Class
  - Strong similarity to other single inheritance, multiple interface languages (eg Java)
  - Constructors
  - Mutable instance fields
  - Instance methods with overriding & overloading
  - May be generic
- X10 Struct
  - Usage: User-defined “primitive type”
  - No inheritance, but may implement interfaces
  - Immutable instance fields
  - Constructors and instance methods (but no inheritance ➔ no overriding)
  - May be generic
- X10 Closure Literal
  - Implement function type (interface)
  - May only capture immutable variables from lexically enclosing environment

Implementing Classes and Structs

- Managed X10
  - Classes and Structs both mapped to Java classes
  - Indirection, pass by reference, alias on assignment
  - No significant performance difference between Class and Struct

- Native X10
  - Class
    - C++ class with vtable
    - Instances heap allocated, indirection, pass by reference, alias on assignment
    - C++ type: Foo*
  - Struct
    - C++ class with no vtable (no virtual methods)
    - Instances inlined into containing structure/array/stack, pass by value, copy on assignment
    - C++ type: Foo

Implementing Interfaces

- Managed X10:
  - Direct implementation by mapping to Java interfaces
  - Leverage JVM optimizations for Java interfaces
- Native X10:
  - Implemented outside of C++ object model
    - Not using multiple virtual inheritance (thunks, extra vtables, constructor overhead) Motivation: single-inheritance subset of C++ is simpler/more portable
    - Using searched ITables (tables of function pointers) for interface method dispatch
  - Slower than Managed X10 and hard for C++ compiler to effectively optimize
- Managed & Native X10:
  - 'implements' does not cost performance
  - Interface calls expected to be slower than non-interface calls

Implementing Generics in Native X10

- X10 generics map very naturally to C++ templates
  - Both instantiation based
  - Get efficient code for X10 generics over primitives/structs
  - Get almost all of desired semantics “for free”
  - Same potential for code bloat as C++ templates

- Features that are hard to implement via C++ templates:
  - Non-final generic instance methods (not implemented; C++ doesn’t support)

    def m[T] (x:T, ….) { … }
  - Instance methods of generic classes with guards (extra level of templates needed)

    def m(x:T, y:T, z:T) { T <:Arithmetic[T] } : T { return x + y - z; }

Implementing Generics in Managed X10

- Issues in mapping X10 generics to Java generics
  - Java generics based on erasure
  - Java does not allow generics over primitives
  - For efficiency, want to use unboxed types (int, not Integer) as much as possible

- Key ideas
  - Type lifting: compile X10 generic to Java generics + type descriptor objects
    - RTT objects as parameters to constructors, stored in instance fields
    - RTT objects provide erased type information when needed
  - Extensive name-mangling to implement overloading semantics
  - Bridge methods to avoid boxing when possible

- Challenges
  - Overheads of manipulating RTT objects
  - Overheads of self-dispatch methods for implementing generic interfaces
  - Balancing inlining (to increase applicability of bridge methods) vs. code size
  - Usability of generated code for source-level Java/X10 interoperability

Implementing Rails

- Rail[T] is a generic class in X10
  - The underlying core data structure for all collections, arrays, etc.
  - Nominally generic, but must be implemented efficiently (no boxing of elements)

- Native X10
  - Straightforward implementation via C++ templates
  - Single C++ object with vtable, length, and variable-size data section “off the end”

- Managed X10
  - Must avoid boxing and use Java int[], float[], etc for Rail[Int], Rail[Float], etc.
  - Approach
    - Rail is Java class with backing raw array stored in instance field of type Object
    - All access to elements require downcast: if T is int, then a(i) ➔ ((int[])a.raw)[i]
    - When T is known statically, X10 optimizer injects downcast during compilation & applies code motion to lift out of loops. In practice generate good code for Rail operations in loops.

Memory Management

- Intra-place memory management
  - Managed X10: JVM’s provide high performance memory management
  - Native X10: use BDW conservative garbage collector
  - Different performance characteristics complicate X10 performance model

- Inter-place memory management
  - GlobalRef and PlaceLocalHandle create cross-place pointers
  - Initial implementation: an escaped global pointer makes object uncollectable
    - Usable via combination of avoidance and manual deallocation
    - Significant restrictions on expressivity and/or safety
  - Distributed GC for Managed X10
    - Implement portable distributed GC using Java weak references
  - Future research opportunities
    - Distributed GC for Native X10
    - Workload evaluation and more sophisticated algorithms (chicken & egg problem)
    - Distributed cycle collection

Serialization

X10' s Distributed Object Model

- Objects live in a single Place
- Objects are only accessible by tasks running in the Place where they live
- Cross-place references
  - **GlobalRef[T]** reference to an object that can be transmitted to other Places
  - **PlaceLocalHandle[T]** "handle" for a distributed data structure with state (objects) at many places. Optimized representation for a set of **GlobalRef[T]** (one per place).
- Implementing **at**
  - Compiler analyzes the body of **at** and identifies roots to copy (exposed variables)
  - Entire object graph reachable from roots is serialized and sent to destination Place
  - A new (unrelated) copy of the object graph is created at the destination Place
- Controlling object graph serialization
  - Instance fields of class may be declared transient (won' t be copied)
  - **GlobalRef[T]/PlaceLocalHandle[T]** serializes id, not the referenced object
  - Implementing CustomSerialization interface allows user-defined behavior

- Major evolutions in object model: X10 1.5, 1.7, 2.0, 2.1 (stable since 2011)

Design Considerations

- X10 semantics: serialization creates isomorphic object graph
  - Detect sharing/aliasing and faithfully recreate it in copy
- Support communication between Places in the same program
  - Assume same version of program running in all Places
  - Assume same JVM (if Managed X10) in all Places
  - Assume same executable (if Native X10) in all Places
- Java/X10 interoperability
  - Object graph statically unknown mix of X10 and Java objects
    - Preserve X10 copying semantics
    - Serialize Java objects that don’t implement java.io.Serializable
    - Want higher performance than vanilla Java serialization
  - Dynamic class loading
    - Serialization ids assigned dynamically (per message)
    - Support for OSGi and other complex usage of Java classloaders in deserialization

Serialization Implementation Highlights

- Compiler support
  - Compiler analyzes body of at to identify roots
  - For every X10 class, generate specialized serialize/deserialize methods
    - Directly handle the instance fields declared by that class
    - Chained together to handle inheritance
  - Generics
    - Managed X10, also serialize RTT objects (hidden instance fields)
    - Native X10, serialize/deserialize code is also templatized
- Runtime support
  - Dynamic detection of sharing (per object graph hash tables)
  - Managed X10
    - Per-class thunks created lazily to amortize meta-programming overheads
    - Use reflection to serialize vanilla Java classes

Distribution

X10RT

- X10RT abstracts transport layers to enable X10 on a range of systems
  - standalone (shared mem), sockets (TCP/IP), PAMI, DCMF (via PGAS), MPI, CUDA
  - X10RT backend chosen at application compile time
  - Each X10RT backend is tied to a launcher
    - custom launcher for sockets and standalone
    - mpirun for MPI, poe or loadleveler for PAMI, etc.

- API for active messages
  - message ids registered with callbacks ahead of time
  - send message API that takes a message id, char* buffer
    - buffer is encoded/decoded at a higher level
  - x10rt_probe() called frequently from arbitrary threads
  - received message causes callback to be executed inside x10rt_probe() call
- API for direct copying of arrays (no serialization, using RDMAs if available)
- API for collectives + collective emulation layer (barrier, bcast, alltoall…)

XRX: At Implementation

- at (p) async
  - source side: synthetize active message
    - async id + serialized heap + control state (finish, clocks)
    - compiler identifies captured variables (roots)
    - runtime serializes heap reachable from roots
  - destination side: decode active message
    - polling (when idle + on runtime entry)
    - new Activity object pushed to worker’s deque

- at (p)
  - implemented as “async at” + return message
  - parent activity blocks waiting for return message
    - normal or abnormal termination (propagate exceptions and stack traces)

- ateach (broadcast)
  - elementary software routing

XRX: Finish Implementation

- Distributed termination detection is hard
  - arbitrary message reordering

- Base algorithm
  - one row of n counters per place with n places
  - increment on spawn, decrement on termination, message on decrement
  - finish triggered when sum of each column is zero

- Optimized algorithms
  - local aggregation and message batching (up to local quiescence)
  - pattern-based specialization
    - local finish, SPMD finish, ping pong, single async
  - software routing
  - uncounted asyncs
  - pure runtime optimizations + static analysis + pragmas ➔ scalable finish

CUDA

Why Program GPUs with X10?

Why program the GPU at all?

Many times faster for certain classes of applications

Hardware is cheap and widely available (as opposed to e.g. FPGA)

Required for performance on top supercomputers, e.g. Titan

Why write X10 (instead of CUDA/OpenCL)?

Use the same programming model (APGAS) on GPU and CPU

Easier to write parallel/distributed GPU-aware programs

Better type safety

Higher level abstractions => fewer lines of code

X10/GPU Programming Experience

Don't hide the GPU – Give the programmer access to the GPU to enable efficient kernels.

We try to make it as easy as possible.

Correctness

Can debug X10 kernel code on CPU first, using standard techniques

Static errors avoid certain classes of faults.

Goal: Eliminate all GPU segfaults with negligible overhead.

Currently detect all dereferences of non-local data (place errors)

TODO: Static array bounds checking

TODO: Static null-pointer checking

Performance: Need understanding of the CUDA performance model

Must know advantages+limitations of [registers, SHM, global memory]

Avoid warp divergence

Avoid irregular/misaligned memory access

Use CUDA profiling tool to debug kernel performance (very easy and usable)

Can inspect and disassemble generated cubin file

Easier to tune blocks/threads using auto-configuration

Why target CUDA and not OpenCL?

In early 2009, OpenCL support was limited

CUDA is based on C++, OpenCL based on C

C++ features help us implement X10 features (e.g. generics).

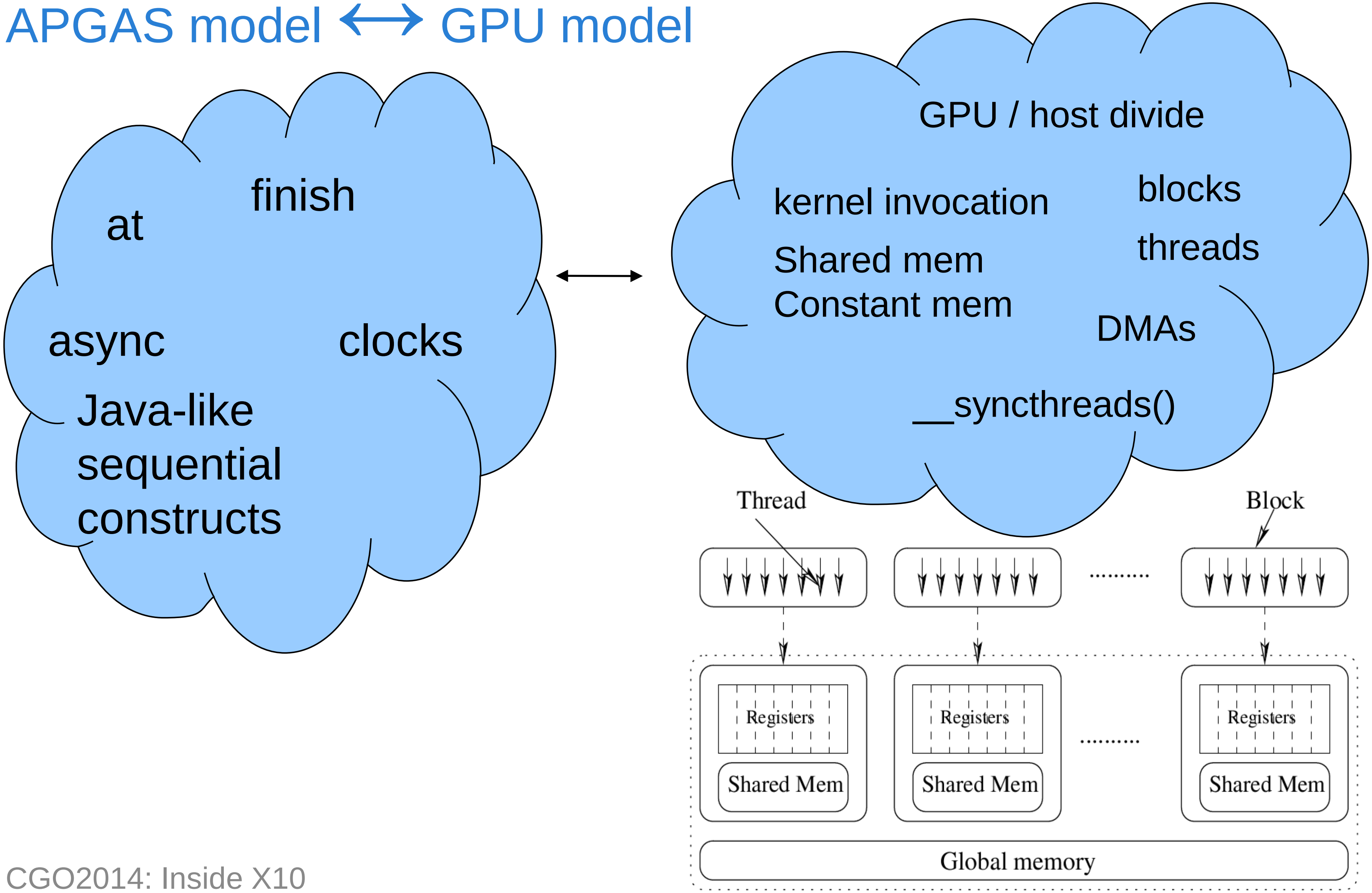
- By targeting CUDA we can re-use parts of existing C++ backend

Strategic relationship between IBM and Nvidia

However there are advantages of OpenCL too...

We could support it in future

APGAS model ⟷ GPU model



at

finish

async

clocks

Java-like sequential constructs

GPU / host divide

kernel invocation

blocks

threads

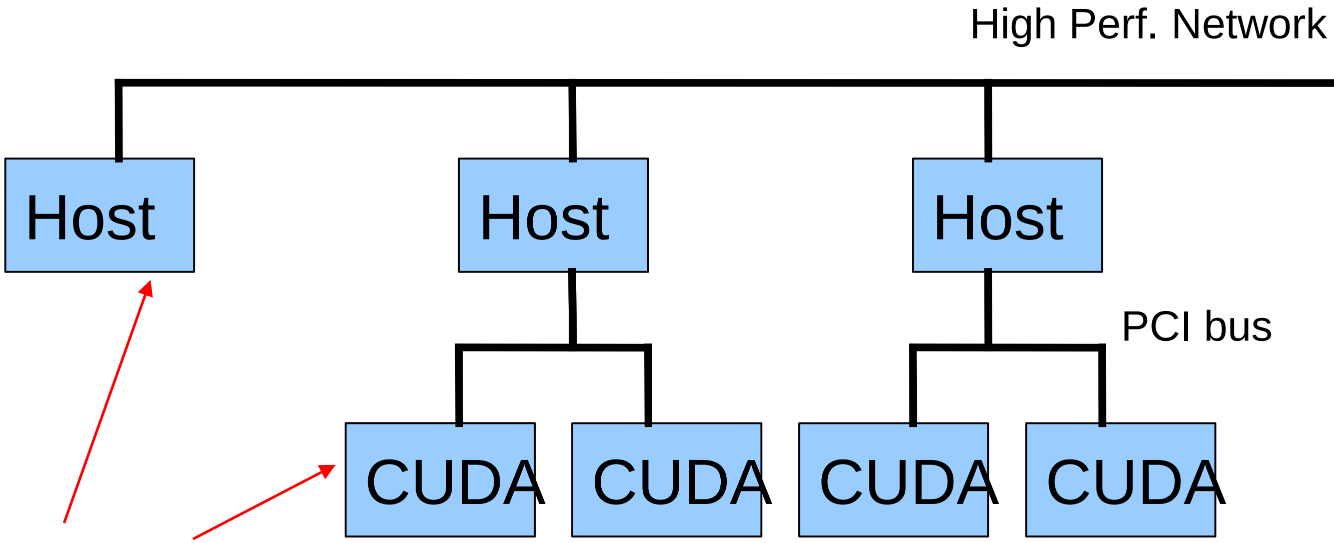
Shared mem

Constant mem

DMAs

__syncthreads()

GPU / Host divide



A 'place' in
APGAS model

Re-use existing language concepts wherever possible

Independent GPU memory space ⇒ new place

Place count and topology unknown until run-time

Same X10 code works on different run-time configurations

Could use same approach for OpenCL, FPGA, Xeon Phi …

X10/CUDA code (mass sqrt example)

```
for (host in Place.places()) at (host) {
    val init = new Rail[Float](1000, (i:Int)=>i as Float);
    val recv = new Rail[Float](1000);
    for (accel in here.children().values()) {
        val remote = CUDAUtilities.makeGlobalRail[Float](accel, 1000, (Int)=>0.0f);
        val num_blocks = 8, num_threads = 64;
        finish async at (accel) @CUDA {
            finish for ([block] in 0..(num_blocks-1)) async {
                clocked finish for ([thread] in 0..(num_threads-1)) clocked async {
                    val tid = block*num_threads + thread;
                    val tids = num_blocks*num_threads;
                    for (var i:Int=tid ; i<1000 ; i+=tids) {
                        remote(i) = Math.sqrtf(init(i));
        }   }   }   }
        // Console.OUT.println(remote(42));
        finish Rail.asyncCopy(remote, 0, recv, 0, recv.size);
        Console.OUT.println(recv(42));
    }
}
```

GPU Code

Discover GPUs

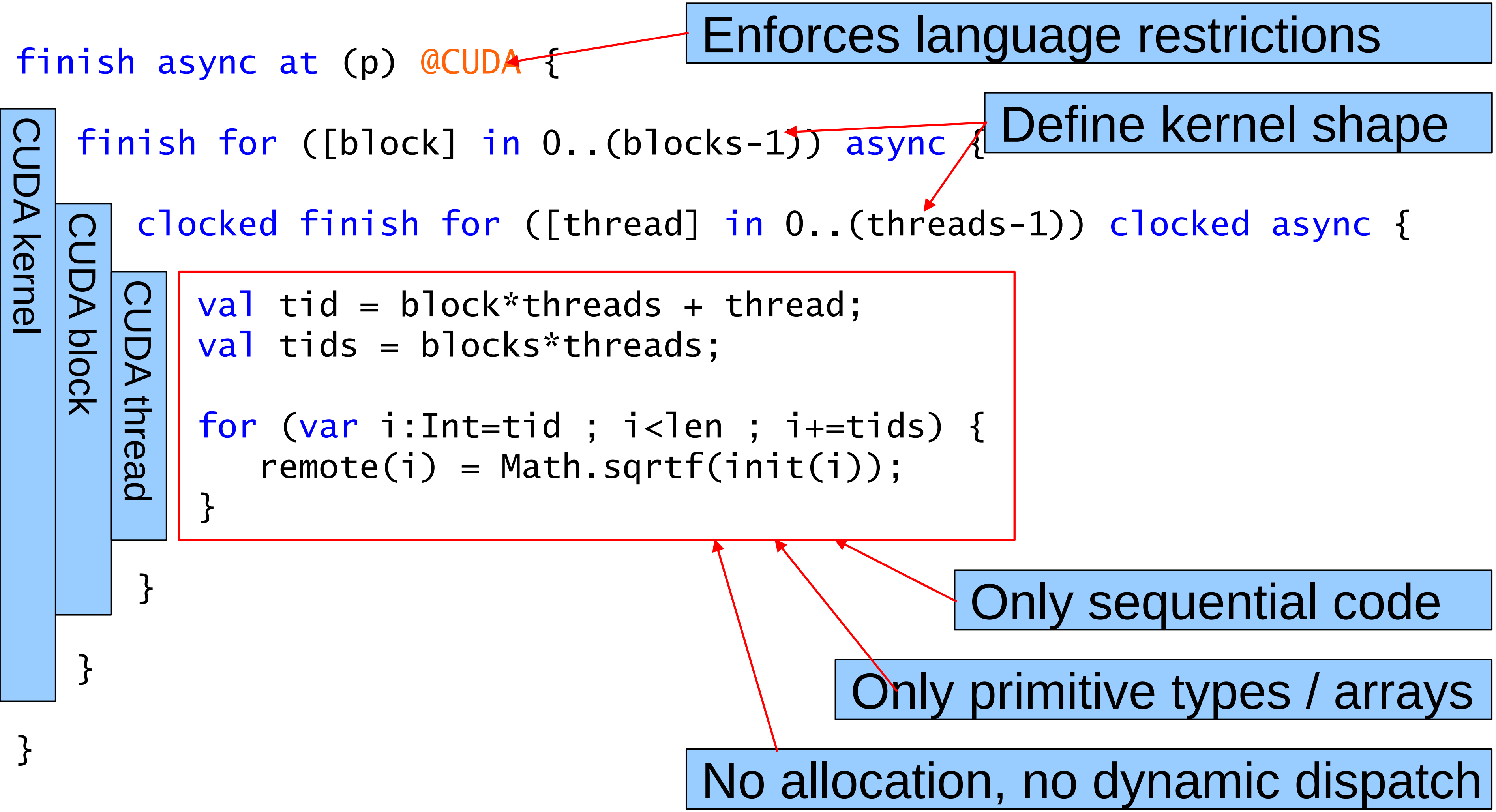
Alloc on GPU

Implicit capture and transfer to GPU

Static type error

Copy result to host

CUDA threads as APGAS activities



```
finish async at (p) @CUDA {
   finish for ([block] in 0..(blocks-1)) async {
      clocked finish for ([thread] in 0..(threads-1)) clocked async {
         val tid = block*threads + thread;
         val tids = blocks*threads;

         for (var i:Int=tid ; i<len ; i+=tids) {
            remote(i) = Math.sqrtf(init(i));
         }
      }
   }
}
```

Enforces language restrictions

Define kernel shape

Only sequential code

Only primitive types / arrays

No allocation, no dynamic dispatch

Dynamic Shared Memory & Barrier

```
finish async at (p) @CUDA {

    finish for (block in 0..(blocks-1)) async {

        val shm = new Rail[Int](threads, (Int)=>0);

        clocked finish for (thread in 0..(threads-1)) clocked async
{

            shm(thread) = f(thread);
            Clock.advanceAll();
            val tmp = shm((thread+1) % threads);

}   }   }
```

Compiled to CUDA 'shared' memory

Initialised on GPU (once per block)

CUDA barrier represented with clock op

General philosophy:

Use existing X10 constructs to express CUDA semantics

APGAS model is sufficient

Constant memory

Compiled to CUDA 'constant' memory

```
val cmem = new CUDAConstantRail[Int](threads, (Int)=>0);
finish async at (p) @CUDA {
    finish for ([block] in 0..(blocks-1)) async {
        clocked finish for ([thread] in 0..(threads-1)) clocked async {
            val tmp = cmem(42);
}   }   }
```

Automatically uploaded to GPU (just before kernel invocation)

General philosophy:

- **Quite similar to 'shared' memory**
  - **Again, APGAS model is sufficient**

Blocks/Threads auto-config

Maximise 'occupancy' at run-time according to a simple heuristic

```
finish async at (p) @CUDA {

   val blocks = CUDAUtilities.autoBlocks(),
       threads = CUDAUtilities.autoThreads();

   finish for ([block] in 0..(blocks-1)) async {

      clocked finish for ([thread] in 0..(threads-1)) clocked async {

         // kernel cannot assume any particular
         // values for 'blocks' and 'threads'

      }

   }

}
```

If called on the CPU, autoBlocks() == 1, autoThreads() == 8

KMeansCUDA kernel

```
finish async at (gpu) @CUDA {
    val blocks = CUDAUtilities.autoBlocks(),
        threads = CUDAUtilities.autoThreads();
    finish for ([block] in 0..(blocks-1)) async {
        val clustercache = new Rail[Float](num_clusters*dims, clusters_copy);
        clocked finish for ([thread] in 0..(threads-1)) clocked async {
            val tid = block * threads + thread;
            val tids = blocks * threads;
            @Unroll(20) for (var p:Int=tid ; p<num_local_points ; p+=tids) {
                var closest:Int = -1;
                var closest_dist:Float = Float.MAX_VALUE;
                for ([k] in 0..(num_clusters-1)) {
                    var dist : Float = 0;
                    for ([d] in 0..(dims-1)) {
                        val tmp = gpu_points(p+d*num_local_points_stride)
                                  - clustercache(k*dims+d);
                        dist += tmp * tmp;
                    }
                    if (dist < closest_dist) {
                        closest_dist = dist;
                        closest = k;
                    }
                }
                gpu_nearest(p) = closest;
            }
        }
    }
}
```

Cache clusters in SHM

Drag inputs from host each iteration

Strip-mine outer loop

Pad for alignment (stride includes padding)

Allocated earlier (via CUDAUtilities)

Above Kernel is only part of the story...
- **Use kernel once per iteration of Lloyd's Alg.**
- **Copy gpu_nearest to CPU when done**
- **Rest of algorithm runs faster on the CPU!**

K-Means graphical demo

Map of USA (can scroll / zoom)

300 Million points (2 dimensional space)

Data auto-generated from zip-code population data

- Town pop. normally distributed around town centers

Use K-Means to find 50 Centroids for USA population

- Possible use case: Delivery distribution warehouses

Visualization uses GL

- Runs on the GPU
- Completely independent from CUDA
- GPUs can be used for graphics too! :)

Can compare K-Means performance on CPU and GPU

K-Means End-to-End Performance

Colours show scaling up to 4 Tesla GPUs

- Y-axis gives Gflops normalized WRT native CUDA implementation (single GPU, host)

2M/4M points, K=100/400, 4 dimensions

- Higher K ⇒ more GPU work ⇒ better scaling

Analysis

Performance sensitive to many factors:

- CPU code: fragile g++ optimisations
- X10/CUDA DMA bandwidth (GPUs share PCI bus)
- Infiniband for host<->host communication (also share PCI bus)
- nvcc register allocation fragile

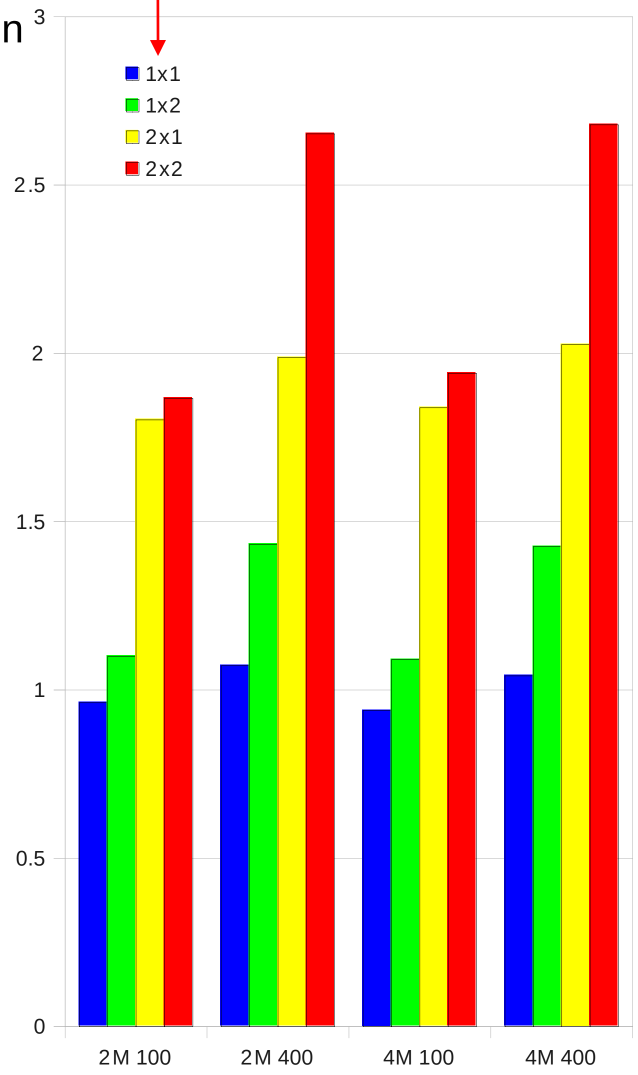
Outstanding DMA issue (~40% bandwidth loss)

- CUDA requires X10 CPU objects allocated with cudaMalloc
- Hard to integrate with existing libraries / GC
- Currently staging DMAs through pre-allocated buffer...
- complaints by users on CUDA forums

Options for further improving performance:

- Use multicore for CPU parts
- Hide DMAs

Hosts x GPUs (per host)



Future Work

Support texture memory...

- Explicit allocation: val t = CUDATextureRail.make(...)
- at (gpu) { … t(x, y) … }

Fermi & Kepler architecture presents obvious opportunities for supporting X10

- Indirect branches ⇒ Object orientation and Exceptions
- Drop support for older hardware?

OpenCL

- No conceptual hurdles
- Different code-gen and runtime work
- X10 programming experience should be very similar

Memory Allocation on GPU

- CUDA now has a 'malloc' call we might use

GPU GC cycle between kernel invocations

- A research project in itself

atomic operations, 64bit values, auto CUDADirectParams, non-Rails