Coherence
Coherence Modes

• Exclusive
• Atomic
• Simultaneous
• …Relaxed…
About Simultaneous Coherence

If tasks $t_1$ and $t_2$ access $r$ with simultaneous coherence, they are guaranteed to be using the same physical instance of $r$

Implies they cannot make a copy of $r$
New Operations

- A task $t$ with simultaneous coherence on $r$ can
- Acquire $r$
  - Remove the copy restriction on $r$
- Release $r$
  - Restore copy restriction on $r$
  - Invalidates any copies made by $t$
  - Flushes any updates to the "master" copy
Phase Barrier

• A phase barrier has a number of *arrivers* and a number of *waiters*

• Arriving at a barrier increases the arrival count but does not block the arriving task

• Waiters proceed past the barrier once the expected number of arrivers have passed the barrier
Use Case

• Long running producer/consumer pattern

• Task 1 produces data that task 2 consumes
  - Share an instance with simultaneous coherence

• Task 1 arrives to indicate it has produced data
  - Task 2 then proceeds to read the data

• Task 2 arrives at a different barrier to indicate it has consumed the data
  - Task 1 then proceeds to produce more data
Upside/Downside

• Used in a task-based SPMD-style of programming
  - Still using tasks and regions, but long-running tasks can communicate with each other using explicit copies of regions

• Exposed to the pitfalls of concurrent programming
  - And in a more asynchronous model
Metaprogramming
What is Metaprogramming?

• Programs that generate programs

• Example: C++ template metaprogramming

• But a very old idea
  - Lisp in the 1950's
  - Explored extensively since the 1980's
Why Metaprogramming?

• Reason #1: Performance

• Consider a function $F(X,Y)$
  - $X$ changes with every call
  - $Y$ is one of a small set of possible values
  - Or fixed for long periods of time

• Generate versions $F_Y(X)$ for each value of $Y$
  - And optimize each $F_Y(.)$ separately
Why Metaprogramming?

• Reason #2: Software maintenance

• Maintaining versions $F_{y}(X)$ for each value of $Y$ by hand is painful

• Much easier to maintain a program that auto-generates the needed versions
Why Metaprogramming?

• Reason #3: Autotuning
  - Based on performance measurements, generate a new version of \( F(X) \)
  - Here, machine characteristics are a “hidden”, constant parameter

• May need to generate many versions \( F(X) \)
  - Which versions and how many are data dependent
  - The space of possible versions could be very large or even infinite
Templates using Metaprogramming

- Templates are an instance of metaprogramming
- Each template argument produces a distinct set of methods, customized to a particular type
- But templates are a crippled programming environment
How Does this Work?

• Lua and Terra (and Regent) share a lexical environment
  - Lua variables can be referred to in Terra & Regent

• Terra types are Lua values
  - E.g., $\text{Array(float)}$
Escape

- Lua can also be used to compute Terra code
  - Expressions or statements

- The *escape* operator \([e]\) inserts the value of the Lua expression \(e\) into a Terra context
  - \(e\) is Lua code
  - That evaluates to a Terra expression
Example

function create_expr(num, v)
    local value
    for i = 1,num do
        if value then
            value = `value + v
        else
            value = `v
        end
    end
    return value
end

terra scale(a: float): float
    return [create_expr(ITERATE,a)]
end
Circuit
Circuit

• Electrical simulation

• A graph
  - Wires are edges
  - Nodes are places where wires meet
Circuit

- Iterative simulation with three phases:
  - calculate_new_currents
  - distribute_charge
  - update_voltages
Look At

• Partitioning
• Tasks
• Mapping
• Optimizations
• Performance
• Legion version
Partitioning
Partitioning Outline

• Partition the graph into pieces

• Each piece consists of
  - Private nodes
    • Nodes with no edges cross into other pieces
  - Shared nodes
    • Nodes with at least one edge crossing to another piece
  - Ghost nodes
    • The neighbors of the shared nodes that are in other pieces
Circuit Dependent Partitioning

var pn_equal = partition(equal, rn, colors)
var pw_outgoing = preimage(rw, pn_equal, rw.in_ptr)
var pw_incoming = preimage(rw, pn_equal, rw.out_ptr)
var pw_crossing_out = pw_outgoing - pw_incoming
var pw_crossing_in = pw_incoming - pw_outgoing
var pn_shared_in = image(rn, pw_crossing_in, rw.out_ptr)
var pn_shared_out = image(rn, pw_crossing_out, rw.in_ptr)
var pn_private = (pn_equal - pn_shared_in) - pn_shared_out
var pn_shared = pn_equal - pn_private
var pn_ghost = image(rn, pw_crossing_out, rw.out_ptr)
Tasks
Mapping
Mapping

• Mapping is the process of assigning resources to Regent/Legion programs

• Conceptually
  - Assign a processor to each task
    • The task will execute in its entirety on that processor
  - Assign a memory to each region argument

• And many other things!
Understanding Mappers

• **Mapping is an API**
  - A set of callbacks

• **Each is called at a particular point in a task’s lifetime**
  - To write mappers, need to know this sequence of stages
The Legion Mapping API

• Mapping is currently done at the Legion level
  - C++

• A mapper implements the mapping API
  - A set of callbacks
High-Level Overview

• An instance of the Legion runtime runs on every node

• When a task is launched the local runtime
  - Makes mapper calls to pick a processor for the task
  - Makes mapper calls to pick memories for the region arguments
  - ... and other mapper calls as well ...
New Concepts

- There are a number of concepts at the mapping level that don’t exist in Regent
  
  - Machine models
  - Variants
  - Physical Instances

- More on this later …
Machine Model

- To pick concrete processors & memories, the runtime must know:
  - How many processors/memories there are
    - And of what kinds
  - And where the processors/memories are
    - At least relative to each other
Machine Model

• Processors
  - LOC
  - TOC
  - PROC_SET
  - UTILITY
  - IO

• Memories
  - GLOBAL
  - SYSTEM
  - RDMA
  - FRAME_BUFFER
  - ZERO_COPY
  - DISK
  - HDF5
Affinities

- **Processor -> Memory**
  - Which memories are attached to a processor

- **Memory -> Memory**
  - Which memories have channels between them

- **Memory -> Processor**
  - All processors attached to a memory

- Affinities are provided as a list of \((\text{proc}, \text{mem})\) and \((\text{mem}, \text{mem})\) pairs
Task Variants

• A task can have multiple variants
  - Different implementations of the same task
  - Multiple variants can be registered with the runtime
  - Variants can have associated constraints

• Examples
  - A variant for LOC
  - Another variant for TOC
  - Variants for different data layouts
Physical Instances

- A *region* is a logical name for data

- A *physical instance* is a copy of that data
  - For some set of fields

- There can be 0, 1 or many physical instances of a specific field of a region at any time
Physical Instances

• Can be *valid* or *invalid*
  - Is the data current or not?

• Live in a specific memory

• Have a specific layout
  - *Column major, row major, blocked, struct-of-arrays, array-of-structs, ...*

• Are allocated explicitly by the mapper

• Are deallocated by the runtime
  - *Garbage collected*
A Word About Physical Instances

- Many physical instances of a region can exist simultaneously
  - Including different versions of the same data

- A task writing version 0 to disk
- A task reading version 5
- A task writing version 6
  - The current version!
- A task scheduled to read version 6
- A task scheduled to write version 7
- A (meta)task scheduled to deallocate version 6
- ...

Legion Bootcamp 2017
A Mapper

- The circuit custom mapper, circuit.cc
Create Mappers

• *Called once on start-up*
  - *On each node*
Mapper Calls: Picking a Processor

• There are three stages, in order:

• Select task options
  - Like it says, choose among some options

• Slice task
  - Break up index launches into chunks and distribute
  - Fixes the node of the task

• Map task
  - Bind the task to a processor
Controlling Processor Choice in Regent

- Place immediately before a task declaration
  - __demand(__cuda)

- Causes both CPU and GPU task variants to be produced

- And the default mapper always prefers to pick a GPU variant if possible
Layout Constraints

• Tasks can have layout constraints on physical instances
  - “This task requires data in row major order”

• Constraints are just that
  - Don’t specify an exact layout
  - Multiple instances may satisfy the constraints
Selecting Physical Instances

• The default mapper first checks if there is an existing valid instance for a region requirement
  - That satisfies the layout constraints
  - And has affinity to the processor

• If so, return it
• If not, create a new instance
  - In system memory (for a CPU mapped task)
  - In frame buffer memory (for a GPU mapped task)
An Exception

• *Reduction instances are always created new*
  - Never reused

• **Note**
  - The framebuffer is not the best place for a reduction instance on the GPU
  - If you map tasks with reduction privileges to the GPU, you may need some custom mapper code.
Reduction Instances

• A reduction instance is a special instance used for reductions

• Pattern

```python
for i in R do
    i.field += val1
    i.field += val2
```

```python
fill(R', 0)
for i in R.indices do
    R'[i] += val1
    R'[i] += val2

... later ...

R += R'
```
Virtual Mappings

• It is also possible for a mapper to map a region to no instance
  - If the task does not use the region itself
  - E.g., only passes it to subtasks

• This is a virtual mapping
Summary

• **Mapping**
  - Selects processors for tasks
  - Selects memories for physical instances
    • Satisfying region requirements of tasks

• **Many options**
  - Default mapper does reasonable things
  - But any sufficiently complex program will need some customization
Regent Optimizations
Index Launches

• A normal task call launches a single task

• An index task call launches a set of tasks
  – One for each point in a supplied index space

• Index launches are more efficient than launching many tasks individually
  – Regent automatically transforms loops of single task launches into index task launches
Example

for x in prt.colors do
    task(prt[x])

becomes

index_launch(task,prt,prt.colors)

(if there are no dependencies)
Control Replication

repeat
  for r in part do
    A(r)
  end
  for r in part do
    B(r)
  end
end

for r in part do
  repeat
    A(r)
    B(r)
  end
end
Control Replication

repeat
  for r in part do
    A(r)
  end
  for r in part
    B(r)
  end
end

for r in part do
  repeat
    A(r)
    ... data movement
    ... B(r)
  end
end
Control Replication

• Control replication is crucial to scalability
  - At least, if one wants to write natural code

• Without it
  - Width of index task launches increases with machine size
  - Depth is small: a single task

• With it depth can increase to the running time of the program
Performance
Regent Circuit Implementation

• Look at three mappings

• All tasks on CPUs, regions in system memory
• All tasks on GPUs, regions in frame buffer
• All tasks on GPUs
  - Shared and ghost regions in zero copy memory
  - Private regions in frame buffer memory
Circuit in Legion
More on Differences Legion vs. Regent

- Runtime object
  - Task registration
- Mappers
  - Mapper creation/registration
- Task context
- Region requirements
- Physical Instances
  - Inline mappings, unmap calls
  - Layout constraints
- Futures
- Accessors
Default Mapper