Regent

Elliott Slaughter
Regent

- A language for the Legion programming model
- Implicit parallelism, sequential semantics
- Tasks + automatic discovery of dependences
- Automatic data movement

A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
Regent vs Legion API

A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)

Regent simplifies Legion prog. model
Regent achieves performance identical to hand-tuned Legion

runtime->unmap_region(ctx, physical_r);
TaskLauncher launcher_A(TASK_A, TaskArgument());
launcher_A.add_region_requirement(
  RegionRequirement(r, READ_WRITE, EXCLUSIVE, r));
launcher_A.add_field(0, FIELD_X);
launcher_A.add_field(0, FIELD_Y);
runtime->execute_task(ctx, launcher_A);
Domain domain = Domain::from_rect<1>(
  Rect<1>(Point<1>(0), Point<1>(2)));
IndexLauncher launcher_B(TASK_B, domain,
  TaskArgument(), ArgumentMap());
launcher_B.add_region_requirement(
  RegionRequirement(p, 0 /* projection */, 
    READ_WRITE, EXCLUSIVE, r));
launcher_B.add_field(0, FIELD_X);
runtime->execute_index_space(ctx, launcher_B);
TaskLauncher launcher_C(TASK_A, TaskArgument());
launcher_C.add_region_requirement(
  RegionRequirement(r, READ_ONLY, EXCLUSIVE, r));
launcher_C.add_field(0, FIELD_X);
launcher_C.add_field(0, FIELD_Y);
runtime->execute_task(ctx, launcher_C);
runtime->map_region(ctx, physical_r);
Pushing the Performance Envelope with Compilation

- Fine-Grained
- Coarse-Grained

- Static Analysis
- Dynamic Analysis

Scale: Small to Large
Task Granularity: Coarse-Grained to Fine-Grained

http://legion.stanford.edu
Data Model

task A(r : region(...)) where \textbf{writes}(r.\{x, y\}) do ... end

task B(r : region(...)) where \textbf{reads writes}(r.x) do ... end

task C(r : region(...)) where \textbf{reads}(r.\{x, y\}) do ... end

task main()

\hspace{1cm} \textbf{var} r = region(...)

\hspace{1cm} \textbf{var} p = partition(equal, r, ...)

\hspace{1cm} A(r)

\hspace{1cm} for i = 0, 3 do

\hspace{2cm} B(p[i])

\hspace{1cm} end

\hspace{1cm} C(r)

\hspace{1cm} end
Execution Model

\begin{itemize}
\item \texttt{var r = region(...)}
\item \texttt{var p = partition(disjoint, r, ...)}
\item \texttt{A(r)}
\item \texttt{for i = 0, 3 do}
  \texttt{B(p[i])}
\item \texttt{end}
\item \texttt{C(r)}
\end{itemize}
Regions

```plaintext
fspace point { x : int, y : int, z : int }
fspace node(list : region(node)) {
    idx : int2d,
    next : ptr(node(list), list),
}

task main()
    var bag = ispace(ptr, 28)
    var grid = ispace(int2d, {x = 4, y = 7})
    var points = region(grid, point)
    var list = region(bag, node(list))
    ...
```
Fills and Copies

task main()
  var grid, points, list = ...
  fill(points.{x, y, z}, 0)
  copy(points.{x, y}, list.idx.{x, y})
  ...

http://legion.stanford.edu
task init_pointers(grid : ispace(int2d),
    points : region(grid, point),
    list : region(node(list)))
where reads(points), reads writes(list.{idx, next}) do
    ...
end

task main()
    var grid, points, list = ...
    init_pointers(grid, points, list)
    ...

http://legion.stanford.edu
Control

task main()

    var grid, points, list = …
    if c1 then … elseif c2 then … else … end
    while c do … end
    for idx = 0, n do … end
    for idx in grid do … end
    for elt in list do … end

    …
task main()

var grid, points, list = …

var last = null(ptr(node(list), list))

for idx in grid do

    var elt = new(ptr(node(list), list))
    elt.next = last
    last = elt
    elt.point = idx
    points[idx].{x, y, z} += 1

end
Vectorization

task inc(grid : ispace(int2d), points : region(grid, point),
        list : region(node(list)))
where reads(list), reduces+(points) do
  __demand(__vectorize)
  for elt in list do
    points[elt.idx].{x, y, z} += 1
  end
end
CUDA

__demand(__cuda)

task inc(grid : ispace(int2d), points : region(grid, point),
          list : region(node(list)))

where reads(list), reduces+(points) do
  for elt in list do
    points[elt.idx].{x, y, z} += 1
  end
end
end
C Functions

local cstdio = terralib.includec(“stdio.h”)
local cmath = terralib.includec(“math.h”)

task main()
    cstdio.printf(“Hello, %f\n”, cmath.sin(1.0))
    ...

Legion Interop

terralib.linklibrary(“my.so”)
local my = terralib.includec(“my.h”)

task main()
  my.legion_task(__runtime(), __context())
...

Metaprogramming

function make_inc(t, v)
    local task inc(r : region(t)) where reads writes(r) do
        for x in r do x += v end
    end
    return inc
end
local inc1 = make_inc(int, 1)

task main()
    var r = …
    inc1(r)
    …
Optimization: Index Launches (Before)

```plaintext
var r = region(…)
var p = partition(disjoint, r, …)
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
```
Optimization: Index Launches (After)

```plaintext
var r = region(…)
var p = partition(disjoint, r, …)
A(r)
for i = 0, 3: B(p[i])
C(r)
```
Optimization: Leaf Tasks (Before)

var $r = \text{region}(\ldots)$

var $p = \text{partition}(\text{disjoint}, r, \ldots)$

$A(r)$

for $i = 0, 3$ do

$B(p[i])$

end

$C(r)$

how many subtasks?

don’t know until here

app thread

runtime thread

app thread

app thread

app thread

app thread

app thread

http://legion.stanford.edu
Optimization: Leaf Tasks (After)

\[ \text{var } r = \text{region}(\ldots) \]
\[ \text{var } p = \text{partition(} \text{disjoint, } r, \ldots) \]
\[ A(r) \]
\[ \text{for } i = 0, 3 \text{ do} \]
\[ \quad B(p[i]) \]
\[ \text{end} \]
\[ C(r) \]
Optimization: Mapping (Before)

```plaintext
var r = region(…)
var p = partition(disjoint, r, …)
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
```

data race!
Optimization: Mapping (Runtime)

unmap(r)
A(r)
map(r) -- blocks
for i = 0, 3 do
  unmap(r)
B(p[i])
map(r) -- blocks
end
Optimization: Mapping (Compiler)

unmap(r)
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
map(r) -- blocks
Other Optimizations

- Futures
- Pointer Check Elision
- Dynamic Branch Elision
- Vectorization
- CUDA Kernel Generation
Work In Progress: Static Dependences

var r = region(…)
var p = partition(disjoint, r, …)
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
Work In Progress: Static Dependences

```
var r = region(...)
var p = partition(disjoint, r, ...)
A(r)
for i = 0, 3 do
    B(p[i])
end
C(r)
```
Work In Progress: SPMD

phase barriers

time

app thread

runtime thread

app thread

node 0

node 1

node 2

node 3

http://legion.stanford.edu
Pushing the Performance Envelope with Compilation

Task Granularity

Dynamic Analysis

Static Analysis

Coarse-Grained

Fine-Grained

Small

Large

Scale
Questions?
Lines of Code

<table>
<thead>
<tr>
<th></th>
<th>Regent</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circuit</td>
<td>825</td>
<td>1,701</td>
</tr>
<tr>
<td>PENNANT</td>
<td>1,789</td>
<td>2,416</td>
</tr>
<tr>
<td>MiniAero</td>
<td>2,836</td>
<td>3,993</td>
</tr>
</tbody>
</table>

Non-comment, non-blank lines of code
Circuit: Absolute Performance

This graph shows the performance of two systems, Regent and Legion, in GFLOPS (Giga Floating Point Operations Per Second) as a function of the total number of CPUs. The performance increases linearly with the number of CPUs for both systems.
PENNANT: Absolute Performance

![Graph showing zones per second (in millions) vs. total CPUs for Regent and OpenMP.](http://legion.stanford.edu)
MiniAero: Absolute Performance

**Single Node**

![Graph showing performance comparison between Regent and MPI+Kokkos for different numbers of CPUs.](image)

- **Cells per second (in millions)**
- **Total CPUs**

**Multiple Nodes**

![Graph showing performance comparison between Regent and MPI+Kokkos for different numbers of nodes.](image)

- **Cells per second (in millions)**
- **Total Nodes (8 CPUs per Node)**
Impact of Optimizations

Circuit

GFLOPS

Cells per second (in millions)

individual optimizations disabled

best single-threaded performance

http://legion.stanford.edu

December 8, 2015
Impact of Optimizations

Zones per second (in millions)

http://legion.stanford.edu