The Legion Programming Model

Alex Aiken
Stanford University & SLAC
Acknowledgments

The Legion project is joint work between Stanford, Los Alamos National Lab, NVIDIA, and SLAC.

Funding has come from many sources, but particularly the DOE and the leadership class facilities.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy’s Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation’s exascale computing imperative.
Tutorial Materials

The slides, example program, and performance profiles are at:

http://theory.stanford.edu/~aiken/ecp
OVERVIEW
Legion & Regent

**Legion is**
- a C++ runtime
- a programming model

**Regent is a programming language**
- For the Legion programming model
- Current implementation is embedded in Lua
- Has an optimizing compiler

This tutorial focuses on Regent
Why Use Legion?

- Easy access to GPUs
  - Simplifies programming complex hardware

- Easy control over data
  - Partitioning, placement and layout in memory

- Automated scheduling and latency hiding
  - Asynchronous tasking
  - Throughput oriented

- Performance portability
Regent Stack

Lua
Host language

Regent
Language and compiler

Legion
High-level runtime

Realm
Low-level runtime
Regent in Lua

- Embedded in Lua
  - Popular scripting language in the graphics community

- Excellent interoperation with C
  - And with other languages

- Python-ish syntax
  - For both Lua and Regent
PAGERANK
PageRank

Today’s example

Input: A directed graph.

Output: The *rank* of each node

- A measure of a node’s importance
- E.g., used for ranking web search results
  - Web pages are nodes
  - Hyperlinks are edges
\[ \text{rank}(n) = \frac{1 - \alpha}{N} + \alpha \sum_{p \in \text{pred}(n)} \frac{\text{rank}(p)}{|\text{succs}(p)|} \]
TASKS
The PageRank Task

**Task:**

```plaintext
task  pagerank: region(...), edges: region(...),
       pr_old: region(...), pr_new: region(...), alpha: float)
```

- **Tasks** are the unit of parallel execution.

- **Logical regions** are (typed) collections
  
  Logical:
  - no implied layout
  - no implied location
The PageRank Task

task pagerank(nodes: region(...), edges: region(...),
    pr_old: region(...), pr_new: region(...), alpha: float)
{
}

}
The PageRank Task

task pagerank(nodes: region(...), edges: region(...),
pr_old: region(...), pr_new: region(...), alpha: float)

where reads(nodes, edges, pr_old), writes(pr_new)

{ }

Privileges declare how a task will use its region arguments.
The PageRank Task

task pagerank(nodes: region(...), edges: region(...),
        pr_old: region(...), pr_new: region(...), alpha: float)
    where reads(nodes, edges, pr_old), writes(pr_new)
{
...
    for n in nodes do
        ... 
        score = 0
        for e in left, right do  -- indices of predecessor edges of n
            ...
            score = score + pr_old[edges[e].src]
        end
        ...
        score = (1 – alpha) / num_nodes + alpha * score
        pr_new[n] = score / out_degree
    end
}
REGIONS
Regions

- A region is a (typed) collection

Regions are the cross product of
- An *index space*
- A *field space*

<table>
<thead>
<tr>
<th>src</th>
<th>dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>y</td>
</tr>
</tbody>
</table>

The region’s index space

The edges region is constructed so that edges are grouped by dst node
Discussion

- Regions are *the* way to organize large data collections in Regent

- Regions can be
  - Structured (e.g., like arrays)
  - Unstructured (e.g., pointer data structures)

- Any number of fields

- Built-in support for 1D, 2D and 3D index spaces
Nodes & Edges

Nodes have two fields: \textit{out\_degree} and \textit{index}
A node’s index field points just beyond its last predecessor edge.
PARTITIONING
Partitioning

To enable parallelism on a region, *partition* it into smaller pieces
- And then run a task on each piece

Partitioning is built in to Legion/Regent
- A rich set of partitioning primitives

Use the primitives to build partitioning algorithms
Equal Partitions

One commonly used primitive is to split a region into a number of (nearly) equal size subregions.

\[
\text{num\_pieces} = \text{inspace(int1d, 4)}
\]
\[
r = \text{region(inspace(int1d, 12), int64)}
\]
\[
p = \text{partition(equal, } r, \text{ num\_pieces})
\]
The Legion runtime knows and uses the region tree to manage mapping and automate parallelism and data movement.
Partitions

- Partitions are first class objects
- An array of the subregions formed by a partition

\[ p = \text{partition}(\text{equal}, r, \text{num}\_\text{pieces}) \]
Discussion

• Partitioning and region creation are dynamic
  - Can be done at any time
  - Regions and partitions are first class values

Regions trees can be any depth
  - Subregions can be partitioned, too

Regions can be partitioned in multiple ways
  - A program can define multiple views of its data

Defining regions/partitions does not materialize them
  - Gives names to subsets of the data
  - Actual computations access physical instances of regions
Partitioning Strategy for PageRank

• Use *edge* partitioning
  • Approximately equal number of edges per subregion
  • Better than node partitioning if nodes can have very different out degrees

• But
  • Keep all predecessor edges of a given node in the same subregion

• So
  • Calculate the range of edges for each subregion
  • Partition the edges by range
  • Partition the nodes compatibly with the edge partition
Nodes & Edges

a node’s index field points just beyond its last predecessor edge
First Step: Calculate Edge Ranges

task init_partition( edge_range : region(ispace(int1d),rect1d),
    edges : region(ispace(int1d), EdgeStruct),
    avg_num_edges : E_ID,
    num_parts: int)
where writes(node_range, edge_range), reads(nodes)
do
...
  for p = 0, num_parts do
    right_bound = min(avg_num_edges * (p + 1), total_num_edges)
    var my_dst: V_ID = edges[right_bound].dst
    -- extend the right bound to the last edge of the current node
    while (right_bound < total_num_edges) do
      var next_dst : V_ID = edges[right_bound+1].dst
      if (my_dst<next_dst) then break end
      right_bound = right_bound + 1
    end
    edge_range[p] = {left_bound, right_bound}
  end
DEPENDENT PARTITIONING
Partitioning, Revisited

Why do we want to partition data?
- For parallelism
- We will launch many tasks over many subregions

A problem
- We often need to partition multiple data structures in a consistent way
- E.g., given that we have partitioned the nodes a particular way, that will dictate the desired partitioning of the edges
Dependent Partitioning

- Distinguish two kinds of partitions

  - Independent partitions
    - Computed from the parent region, using, e.g.,
      - partition(equals, ...)

  - Dependent partitions
    - Computed using another partition
Dependent Partitioning Operations

- **Image**
  - Use the image of a partition to define a new partition
  - E.g., the image of a field
  - E.g., or a range of values

- **Preimage**
  - Use the pre-image of a field in a partition …

- **Set operations**
  - Form new partitions using the intersection, union, and set difference of other partitions
  - Not illustrated today
Computes elements reachable via a field lookup

Can be applied to index space or another partition

Computation is distributed based on location of data
Preimage

- Inverse of image
  - Computes elements that reach a given subspace
  - Preserves disjointness

- Multiple dependent partitioning operations can be combined
  - Capture complex task access patterns
Dependent Partitioning in PageRank

- The use of dependent partitioning in PageRank is simple.

- Define a partition of the edges
  - Using the computed edge ranges

- Then define a partition of the nodes using the destination node of each edge.
Picture (Reminder)

NODES

EDGES
Picture

= dst field of edge

NODES

EDGES

EDGE_RANGE

0:1  2:4  5:7  8:9
Picture

= dst field of edge
The diagram illustrates the concept of edge ranges in a graph. The nodes are represented at the top, with arrows indicating the direction of edges. The edge ranges are shown at the bottom, with the notation 

\[ \text{EDGE\_RANGE} = \{0:1, 2:4, 5:7, 8:9\} \]

Each edge is associated with a destination (dst) field, as indicated by the upward arrows pointing to the respective node destination fields.
PAGERANK MAIN
PARALLELISM
Program Semantics

- A Legion program is a sequence of task launches

- The runtime analyzes the tasks for *interference*
  - Tasks with conflicting accesses to the same data
  - Non-interfering tasks can execute in parallel
  - Interfering tasks are serialized

- Guarantees sequential semantics
  - Program result is as if it had executed sequentially
  - Very useful for debugging at scale
Task Graphs

- When Legion discovers interfering tasks, an edge is added to the task graph recording the dependency.

Three wavefronts:
- The runtime building the task graph
- The application executing the graph
- The runtime collecting resources from finished tasks
Parallel Loop from PageRank

for p in part do
  pagerank(part_nodes[p], part_edges[p],
            pr_score0, part_score1[p], ... )
end

The different calls to pagerank don’t interfere.
Why? Only part_score1[] is written and it is a disjoint partition.

Note the use of different views on to the data. We use both the entire pr_score0[] region and subregions of part_score1[].
MAPPING
Mapping Interface

- Application selects:
  - Where tasks run
  - Where regions are placed

- Mapping computed dynamically

- Decouple correctness from performance
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!

There is a default mapper with reasonable heuristics

- Just another mapper, but a generic one
Mapping Interface

- At the Legion level, 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

- Regent has a mapping DSL
  - Concise, easy to use
  - Compiles to the Legion mapping API
  - Currently supports only static mappings
High-Level Overview of Mapping

- An instance of the Legion runtime runs on each node

When a task is launched on the local runtime:
  - The mapper picks a processor for the task
  - The mapper picks memories for the region arguments
  - ... and other things 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
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

A machine model is written once for each machine
Components of a 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

- \((proc,mem)\) and \((mem,mem)\) pairs
- Also include bandwidth and latency information
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
Variants 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
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
- …
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
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
PAGERANK MAPPER
PROFILING
Legion Prof

- A tool for showing performance timeline
  - Each processor is a timeline
  - Each operation is a time interval
  - Different kinds of operations have different colors

- White space = idle time
  - Want to understand why there is white space
Example Profiles from PageRank

1 node, 8 cpus
- pagerank/run_pr.sh --program baseline --cpus 8 --nodes 1 --gpus 0

1 node, 1 GPU
- pagerank/run_pr.sh --program baseline --cpus 4 --nodes 1 --gpus 1

1 node, 2 GPUs
- pagerank/run_pr.sh --program baseline --cpus 4 --nodes 1 --gpus 2

1 node, 4 GPUs
- pagerank/run_pr.sh --program baseline --cpus 4 --nodes 1 --gpus 4
Performance Results

PageRank Performance with Different Mapping Configurations

- **1 GPU/node (GPU Memory)**
- **2 GPU/node (GPU Memory)**
- **4 GPU/node (GPU Memory)**
- **1 GPU/node (Zero-Copy Memory)**
- **2 GPU/node (Zero-Copy Memory)**
- **4 GPU/node (Zero-Copy Memory)**

**Run time per iteration (seconds)**
OTHER APPLICATIONS
S3D: Combustion Simulation

Simulates chemical reactions
- DME (30 species)
- Heptane (52 species)
- PRF (116 species)

Two parts
- Physics
  - Nearest neighbor communication
  - Data parallel
- Chemistry
  - Local
  - Complex task parallelism
- Large working sets/task

Recent 3D DNS of auto-ignition with 30-species DME chemistry (Bansal et al. 2011)
Mapping for Heptane $48^3$

Dynamic Analysis for (rhfs+2)  Clean-up/meta tasks

4 AMD Interlagos Integer cores for Legion Runtime

8 AMD Interlagos FP cores for application

NVIDIA Kepler K20
Mapping for Heptane $96^3$

- Handle larger problem sizes per node
  - Higher computation-to-communication ratios
  - More power efficient

- Different mapping
  - Limited by size of GPU framebuffer

- Legion analysis is independent of problem size
  - Larger tasks $\rightarrow$ fewer runtime cores
Weak Scaling: PRF on Titan
Fast Graph Analytics

Conventional wisdom:
- Graph processing has trouble taking advantage of distributed memory

High performance graph processing systems are dominated by shared-memory CPU-based systems

Observation
- GPUs provide higher memory bandwidth than CPUs
- Can avoid communication by careful placement of data in the memory hierarchy
Fast Graph Processing [VLDB’18]

Competitive with state-of-the-art single-GPU graph processing engines.

Orders of magnitude speedup compared to state-of-the-art distributed/shared memory CPU systems.
In CNNs, data is commonly organized as 4D tensors.

\[ \text{tensor} = [\text{image}, \text{height}, \text{width}, \text{channel}] \]

Existing tools parallelize the *image* dimension.

**Motivation**
- Explore other parallelizable dimensions
- Allow each layer to be parallelized differently
RESULTS

Figure 1: Training throughput on 16 GPUs.

Figure 2: Data transfers in each step on 16 GPUs with a minibatch size of 512.
Perspectives
Separating Concerns

- Current practice entangles functionality, scheduling, and mapping
  - Consider a code written in MPI + OpenMP + CUDA

- Alternative
  - Specify functionality and dependencies first
  - Then focus on mapping and scheduling for a machine
  - A lot of the benefits of Legion flow from this design
Programmer Productivity

In the end, it’s all about productivity

How much work is needed to achieve a desired level of performance?

Legion philosophy

- More expressive data model
- Requires more initial work from the programmer
- But makes later stages easier & more flexible
  - Easy to try different partitioning strategies
  - Easy to explore alternative mappings
Legion

Legion website: http://legion.stanford.edu