Legion: Expressing Locality and Independence with Logical Regions

Authors: Bauer, M., Treichler, S., Slaughter, E., Aiken, A. @International Conference on Supercomputing (SC 2012)

Presented by Alexey Voronin
For CSE598APK-FA18 @UIUC
Outline

● What is Legion
● Design Goals
● Programming Model
  ○ Data
  ○ Tasks
● Example
● Mapping Interface
● Run Time System
● Conclusion
● Afterword
Legion & Regent

- Legion
  - A data-centric programming model
  - Asynchronous many task C++ runtime system
- Regent
  - Programming language for legion programming model
  - Current implementation is embedded in Lua
  - Has an optimizing compiler
System Architecture
System Architecture

Focus of the talk

Compiled Regent Applications
Regent Compiler
C++ Legion Applications
Default Mapper
Custom Mappers
Legion C++ Runtime API
Mapper Interface
Legion Runtime
Realm Runtime
GASNet
CUDA
Additional Modules
Legion/Regent Design Goals

- **Programmability**
  - Sequential Semantics
    - Readable code
    - Parallelism extracted automatically

- **Throughput oriented**
  - Latency of single task is irrelevant
  - Overall time is what matters
  - Good performance on heterogeneous architectures

- **Run time decision making**
  - Asynchronous execution
  - Runtime decision making because of software/hardware dynamics
Legion/Regent Design Goals

- **Programmability**
  - Sequential Semantics
    - Readable code
    - Parallelism extracted automatically
- **Throughput oriented**
  - Latency of single task is irrelevant
  - Overall time is what matters
  - Good performance on heterogeneous architectures
- **Run time decision making**
  - Asynchronous execution
  - Runtime decision making because of software/hardware dynamics

How?
Every core has
- queue of independent work
- queue of transfers to do
Minimize synchronization points
Separation of Concerns

- Legion Programs
- Legion Mappers

Compilation and Runtime Understanding of Data

- Extraction of parallelism
- Management of data transfers
- Task scheduling and Latency hiding
- Data-Dependent Behavior
High Level View of Legion

**Tasks** (execution model)
Describe parallel execution elements and algorithmic operations with sequential semantics, out-of-order execution

**Regions** (data model)
Describe decomposition of computational domain.
- Privileges (read-write, read-only, reduce)
- Coherence (exclusive, atomic)

**Mapper**
Describes how tasks and regions should be mapped to the target architecture

Mapper allows architecture-specific optimization without effecting the correctness of the task or domain descriptions

*(pic source in comments)*
Logical Regions

- Typed Collection
  - Structured (like arrays)
  - Unstructured (like pointer data structures)

Suppose we want an array of cell_ts:

```c
typedef struct cell_t {
  double temp, pres;
} cell_t;
```

In Legion it would look like this -

```
<table>
<thead>
<tr>
<th>Index Space</th>
<th>Field Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell 0</td>
<td>temp₀</td>
</tr>
<tr>
<td></td>
<td>pres₀</td>
</tr>
<tr>
<td>cell 1</td>
<td>temp₁</td>
</tr>
<tr>
<td></td>
<td>pres₁</td>
</tr>
<tr>
<td>cell 2</td>
<td>temp₂</td>
</tr>
<tr>
<td></td>
<td>pres₂</td>
</tr>
<tr>
<td>cell 3</td>
<td>temp₃</td>
</tr>
<tr>
<td></td>
<td>pres₃</td>
</tr>
</tbody>
</table>
```
Structured Logical Regions

- Regions are split into partitions
  - Enables parallelism and allows tasks to run on each piece
  - Same data can be partitioned multiple (overlapping) ways

<table>
<thead>
<tr>
<th>Index Space</th>
<th>Field Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell 0</td>
<td>temp_0</td>
</tr>
<tr>
<td>cell 1</td>
<td>temp_1</td>
</tr>
<tr>
<td>cell 2</td>
<td>temp_2</td>
</tr>
<tr>
<td>cell 3</td>
<td>temp_3</td>
</tr>
</tbody>
</table>

Subregion/Partition -> Task
Tasks

- Units of parallel execution
- Runs until block or terminate
- Task that takes region arguments must declare what type of privileges it has in the region
  - Reads, writes, reduce
- Legion allows the user to define multiple different variants of the same task
  - CPU, GP, CPU that supports AVX instructions, etc.
  - Each variant is up to the user to implement
  - Mapper chooses which variant to use
Electrical Circuit Simulation Example

The circuit consists of a collection of *wires* and *nodes* where wires meet. At each time step the simulation calculates currents, distributes charges, and updates voltages.

How to group data into regions?
How are regions partitioned into subregions?
Electrical Circuit Simulation Example

How to group data into regions?
How are regions partitioned into subregions?

Regions:
- Nodes
  - Private
  - Shared
  - Ghost
- Wires
This region tree data structure plays an important role in scheduling tasks for out-of-order execution.
Specifies that the regions are accessed with read-write privileges and exclusive coherence (i.e., no other task can access these two regions concurrently).
Specifies that the regions are accessed with read-write privileges and exclusive coherence (i.e., no other task can access these two regions concurrently).
Specifies that the regions are accessed with read-write privileges and exclusive coherence (i.e., no other task can access these two regions concurrently).

spawn indicates a task call.

→ parallelism

interpass dependencies are determined automatically based on the region access declarations.
// ROE = Read-Only-Exclusive

void calc_new_currents(CircuitPiece piece):
    RWE(piece.rw_pvt), ROE(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
        foreach(w : piece.rw_pvt)
            w→current = (w→in_node→voltage - w→out_node→voltage) / w→resistance;
    }

    // Rda = Reduce-Atomic

void distribute_charge(CircuitPiece piece, float dt):
    ROE(piece.rw_pvt), RdA(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
        foreach(w : piece.rw_pvt) {
            w→in_node→new_charge += -dt * w→current;
            w→out_node→new_charge += dt * w→current;
        }
    }

void update_voltages(CircuitPiece piece): RWE(piece.rn_pvt, piece.rn_shr) {
    foreach(n : piece.rn_pvt, piece.rn_shr) {
        n→voltage += n→new_charge / n→capacitance;
        n→new_charge = 0;
    }
}
Circuit Example (cont.)

```c
// ROE = Read-Only-Exclusive
void calc_new_currents(CircuitPiece piece):
    RWE(piece.rw_pvt), ROE(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
        foreach(w : piece.rw_pvt)
            w→current = (w→in_node→voltage − w→out_node→voltage) / w→resistance;
    }

    // RdA = Reduce-Atomic
void distribute_charge(CircuitPiece piece, float dt):
    ROE(piece.rw_pvt), RdA(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
        foreach(w : piece.rw_pvt) {
            w→in_node→new_charge += −dt * w→current;
            w→out_node→new_charge += dt * w→current;
        }
    }

void update_voltages(CircuitPiece piece):
    RWE(piece.rn_pvt, piece.rn_shr) {
        foreach(n : piece.rn_pvt, piece.rn_shr) {
            n→voltage += n→new_charge / n→capacitance;
            n→new_charge = 0;
        }
    }
```

Legion uses tasks’ region arguments to compute which tasks can run in parallel
Mapping Interface

- Isolates mapping decisions from application code
- Gives programmers control over where tasks run and where region instances are placed
- Allows for dynamic decision making based on input data
- Currently done at Legion level

Properties:

- Program correctness is unaffected by mapper decisions
- Isolates machine-specific decisions to the mapper, resulting in portability of Legion programs
Mapping Interface

Consists of 10 methods that Legion runtime system call for mapping decisions.

A mapper implementing these methods has access to following properties
- List of processors and their type (e.g. CPU, GPU)
- List of memories visible to each processor
- Related latencies and bandwidths

Three most important interface calls:
- select_initial_processor
- permit_task_steal
- map_task_region
Mapping Interface

Three most important interface calls:

- **select_initial_processor**
  - SOOP will ask for a processor for a task \( t \)
  - Mapper can keep task \( t \) local or send it to any other processor

- **permit_task_steal**
  - SOOP asks which tasks can be stolen
  - If no stealing allowed, return empty set

- **map_task_region**
  - For each logical region \( r \) used by a task, a SOOP asks for a prioritized list of memories where a physical instance of \( r \) should be placed
  - Mapper returns a priority list of memories in which the SOOP should attempt to either reuse or create a physical instance of \( r \)
Default Mapping Interface

Legion provides a default mapping interface for a quick start:

- **select_initial_processor**
  - Mapper checks the type of processors for which task $t$ has implementations
  - If the fastest implementation is for the local processor, the mapper keeps $t$ local

- **permit_task_steal**
  - Mapper inspects the logical regions for the task being stolen and marks that other tasks using the same logical regions should be stolen as well

- **map_task_region**
  - Select the final (set of) processor(s) that the task will be executed on
  - Select the variant of the task to execute
  - Select the physical instances to hold the data for each logical region
  - Optionally select the task priority
Software Out of Order Processor (SOOP)

- Dynamically schedules a stream of tasks
- Constrained by region dependences
- Pipelined, distributed, and extracts nested parallelism from subtasks
- Must hide extremely long latencies
  - Deferred execution model
    - Tasks run until they block or terminate
    - Blocking does not prevent independent work from being done
    - Blocking does prevent the task from continuing and launching more tasks
More on (Non-)Blocking Execution

- Futures
  - Objects which represent a pending return value from a task
  - Two ways to use them,
    - Blocking
      - Task pauses until the subtask that is completing the future returns (bad for performance)
    - Non-blocking
      - Can pass it as an argument to a function, which won’t execute until the result becomes available
        - Allows the Legion runtime to discover as much task-level parallelism as possible.
Circuit Simulation

- **Machines**
  - Linux based
  - Pthreads for managing CPU threads
  - CUDA for GPUs
  - GASNet for inter-node communication

- **Legion Set-up**
  - Runtime handles all of the resource allocation, scheduling, and data movement across the cluster of GPUs.
    - Mapper queries the list of GPUs in the machine and identifies each GPU’s framebuffer memory and zero-copy memory
    - Circuit partitions are assigned a home GPU in round-robin fashion
Tasks and data for the circuit simulation on a cluster of GPUs
Circuit simulation speed relative to single-GPU implementation

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Sapling</th>
<th>Viz</th>
<th>Keeneland</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>4</td>
<td>10</td>
<td>32 (120)</td>
</tr>
<tr>
<td>CPUs/Node</td>
<td>2x Xeon 5680</td>
<td>2x Xeon 5680</td>
<td>2x Xeon 5660</td>
</tr>
<tr>
<td>HyperThreading</td>
<td>on</td>
<td>off</td>
<td>off</td>
</tr>
<tr>
<td>GPUs/Node</td>
<td>2x Tesla C2070</td>
<td>5x Quadro Q5000</td>
<td>3x Tesla M2090</td>
</tr>
<tr>
<td>DRAM/Node</td>
<td>48 GB</td>
<td>24 GB</td>
<td>24 GB</td>
</tr>
<tr>
<td>Infiniband</td>
<td>2x QDR</td>
<td>QDR</td>
<td>2x QDR</td>
</tr>
</tbody>
</table>
Circuit simulation speed relative to single-GPU implementation

Cluster | Sapling | Viz | Keeneland
--- | --- | --- | ---
Nodes | 4 | 10 | 32 (120)
CPUs/Node | 2x Xeon 5680 | 2x Xeon 5680 | 2x Xeon 5660
HyperThreading | on | off | off
GPUs/Node | 2x Tesla C2070 | 5x Quadro Q5000 | 3x Tesla M2090
DRAM/Node | 48 GB | 24 GB | 24 GB
Infiniband | 2x QDR | QDR | 2x QDR

(b) Overhead of circuit simulation on Keeneland with 3 GPUs/node.
Applications Using Legion

- **Snap**
  - Neutral Particle Transport mini-app
  - Results
    - Different mappers allows the application to specialize itself for different target architectures
    - Verbose codebase due to many tasks variants
    - Legion specific calls amount to only in 2% of overall code
- **Legion Version of High Performance Conjugate Gradients (HPCG) Benchmark**
  - No performance results published
- **Lux**
  - A distributed multi-GPU system for fast graph processing.
  - Much better multi-GPU performance than competitors, mostly due to smarter load balancing
Conclusion

Legion Goals

- High Performance
- Performance Portability
- Programmability (using sequential semantics)

Central role of Legion

- Schedule tasks in a way that preserves “locality” and “independence”
- Determine when to run the tasks

What legion doesn’t do

- Automatically generate tasks
- Automatically map tasks/data to hardware
Afterword

- Don’t spend too much on the paper
- Go to [https://legion.stanford.edu](https://legion.stanford.edu)
  - Tutorials
    - Rudimentary
  - Bootcamp
    - 20+ hours of ‘getting started’ videos
- [Regent](#)
- [ATPESC training video](#)
- Copper Mountain Conference on Multigrid Methods [Presentation](#) (2015)