The Legion Mapping Interface

Mike Bauer
Philosophy

- Decouple specification from mapping
  - Performance portability
- Expose all mapping (perf) decisions to Legion user
  - Guessing is bad!
  - Don’t want to fight Legion for performance
  - Propagate mapping control up through layers of abstraction
- Dynamic Mapping
  - React to machine changes
  - React to app. changes
Machine Model

- Machine is a graph of processors and memories
- Nodes contain attributes:
  - Processors: kind, speed
  - Memories: kind, capacity, “hardness”
- Edges describe relationships:
  - Processor-Memory affinity
  - Memory-Memory affinity
  - Bandwidth, latency
- Machine object interface
Mapper Model

- Create a separate mapper for each processor
  - Mappers can specialize themselves
  - Avoid bottlenecks on mapping queries
- Support arbitrary number of mappers per application
  - Compose applications and libraries with different mappers
- Initialized at start-up of Legion runtime
  - `set_registration_callback` function
  - Add mappers with the `add_mapper` function
  - Default mapper is always given `MapperID 0`
    - Can be replaced with `replace_default_mapper` function

```
+-----+  D 1 2  D 1 2  D 1 2 +-----+  D 1 2  D 1 2  D 1 2 +-----+  D 1 2  D 1 2  D 1 2 +-----+
|    |    CPU |    CPU |    GPU |    |    CPU |    CPU |    GPU |    |
| Node|      |      |      | Node|      |      |      |      |
```
Mapper API

- Mapping API is a pure virtual interface
  - Easy to extend existing mappers
- Methods invoked by the runtime as queries
  - At most one invocation per mapper at a time
  - No need for locks
- Mappers can be stateful
  - Memoize information
  - State is distributed

```cpp
class Mapper {
  public:
    virtual void select_task_options(Task*) = 0;
  public:
    virtual void pre_map_task(Task*) = 0;
    virtual void map_task(Task*) = 0;
    virtual void post_map_task(Task*) = 0;
  ...
};
```
Default Mapper

- Legion comes with a default mapper
- Implement custom mappers that inherit from default mapper
  - Only need to customize specific mapper calls
  - Leverage open/closed principle of software engineering
- Lends itself to a natural performance tuning loop
  - Repeat for each application+architecture
  - Easy to autotune

Start

Profile Application Execution

Determine Performance Bottlenecks

Continue Customizing Mapper

Profile Application Execution

Determine Performance Bottlenecks

Continue Customizing Mapper
The Lifetime of a Task

- Mapper calls for tasks follow the task pipeline

- Not all calls handled by the same mapper object
  - Tasks can map both locally and remotely
  - Guaranteed to be handled by mappers of the same ID

```
Always Done Locally

Select Task Options
Pre-Map Task

Can Be Done Locally or Remotely

Map Task
Select Task Variant
Post-Map Task
```

Select
Task
Options
Pre-Map
Task
Map Task
Select
Task
Variant
Post-Map
Task

Select
Task
Variant

Can Be Done Locally or Remotely
Mapping Locally vs. Remotely

Three important processors associated with Task

- Origin Processor (\texttt{orig\_proc}): where task was launched
- Current Processor (\texttt{current\_proc}): owner of the task
- Target Processor (\texttt{target\_proc}): current mapping target

Tasks can be mapped locally or remotely

- Locally: \texttt{current\_proc} == \texttt{orig\_proc}
- Remotely: \texttt{current\_proc} == \texttt{target\_proc} (\texttt{!}== \texttt{orig\_proc})

![Diagram of mapping process]

Task Meta-Data >> Launch Meta-Data

Remote Mapping -> More Parallelism
Select Task Options

virtual void select_task_options(Task *task)
  
  Currently decorate fields of Task object
  
  Planned: structure describing options

Assign the following fields:
  
  target_proc – pick the first owner processor
  
  inline_task – inline using parent task’s physical regions
  
  spawn_task – make eligible for stealing
  
  map_locally – map the task locally or remotely
  
  profile_task – capture profiling information
Task Mapping (Part 1)

- Tasks always have an “owner” processor

- Owner can be changed until a task is mapped
  - Once a task is mapped it will run on owner processor

- Mapping a task consists of three decisions
  - Fixing the owner processor
  - Selecting memories for physical instances of each region
  - Determining layout constraints for physical instances
Task Mapping (Part 2)

virtual bool map_task(Task *task)
  
  Choose memory ordering for each region requirement
  Return ‘true’ to be notified of mapping result

Task has a vector of application-specified regions
  
  Represented by region requirements
  Called regions

Legion provides list of current memories with data
  
  Called current_instances
  Boolean indicates if contains valid data for all fields

Mapper ranks target memories in target_ranking
  
  Runtime tries to find or create instance in each memory
  Will issue necessary copies and synchronization
  Choose layout by selecting blocking_factor
Task Mapping (Part 3)

- Legion automatically computes copies based on mapping decisions
  - Sometimes there might be multiple valid sources
  - Never guess! (Legion knows what to do if only one source)

- `virtual void rank_copy_sources(...)`
  - Set of possible source memories
  - Memory containing physical instance
  - Populate a vector ranking source memories by preference

![Diagram showing memory types and connections]

- DRAM
- GASNet
- Framebuffer
Task Mapping (Future)

- New mapping API in progress
  - Switch from memory centric to physical instance centric
  - Be field aware
  - Support more data layout formats

- map_task will not change much
  - Legion will provide information about physical instances
    - Layout, field sizes, which fields are valid
  - Mappers provide ranking of physical instances

- Physical instances specified as a set of constraints
  - Order of index space dimensions + field interleaving
  - Constraints on specific pointers and offsets
Selecting Task Variants

- Task variant selected based on mapping decisions

- Legion examines all constraints and picks variant
  - Processor kind, physical instance memories and layouts

- If there are multiple valid variants then query mapper
  - `virtual void select_task_variant(Task *task)`

- Might require many variants. Is there a better way?
  - Yes! Task generator functions (using meta-programming)
Dealing with Failed Mappings

- Mappings can fail for many reasons
  - Resource utilization
  - Memories not visible from target processor
  - No registered task variant based on constraints

```c
virtual void notify_mapping_failed(...)
```
  - Region requirements annotated by mapping_failed field

- Failed tasks automatically ready to map again
  - Mappers can try mapping them again later
  - Watch out for repeated failures (looks like livelock)
  - Future work: establish conditions for re-mapping
Pre-Map Task (Part 1)

```cpp
virtual bool pre_map_task(Task *task)
```
- Can early-map region requirements to physical instances
- Performed on origin processor before task can be moved
- Return ‘true’ to be notified of pre-mapping result

Handle some special cases
- Read-Write coherence on index space task region
Pre-Map Task (Part 2)

- Runtime performs "close operations" as part of pre-mapping task

- Handle translation between different views

- Two options:
  - Concrete Instance
  - Composite Instance

- Composite Instances
  - Memoize intersection tests to amortize cost

virtual bool rank_copy_targets(...) return true for composite instance
Post-Map Task (In Progress)

- Create optional checkpoints of logical regions
  - Generate physical instances in hardened memories
  - Copies automatically issued by Legion runtime
  - Control which logical regions and fields are saved
Mapping Other Operations

- Legion maps many operations other than tasks
  - Inline mappings
  - Explicit region-to-region copies

- Similar mapping calls, all on origin processor
  - `virtual void map_copy(Copy *copy)`
  - `virtual void map_inline(Inline *inline_op)`

- Map region requirements the same as tasks
Managing Deferred Execution

- Legion is an out-of-order task processor
  - How far do we run ahead (into the future)?
  - Machine and application dependent -> mapper decision

Analysis from iteration i+2
Tasks from iteration i

Utility Core

AMD Interlagos cores

NVIDIA Kepler K20

http://legion.stanford.edu
Managing Deferred Execution (2)

- Two components of managing run-ahead
  - How many sub-tasks outstanding per task?
  - When should tasks begin the mapping process?

- Control max outstanding sub-tasks with window size
  - virtual void configure_context(Task *task)
  - Set max_window_size (default 1024)
  - Can be unbounded (any negative value)
  - Trade-off parallelism discovery with memory usage

- Control max outstanding sub-tasks with frames
  - Call issue_frame at the start of each iteration
  - Set max_outstanding_frames
Managing Deferred Execution (3)

Legion maintains ready queue for each mapper
- Contains tasks that are ready to map

Mappers decide when to start mapping tasks
- virtual void select_tasks_to_schedule(list<Task*>)
- Open question: when to stop invocation?
  - Right now: when “enough” tasks outstanding (-hl:sched)

Can perform one of three operations for each task
- Start mapping (set schedule field of Task* to true)
- Change current_proc to new processor to send remotely
- Do nothing: important for loading balancing
Inter-Node Load Balancing

Legion supports both push and pull load balancing

- Push tasks to other nodes
- Pull work from other nodes (e.g. stealing)

Two forms of push

- Change current_proc in select_tasks_to_schedule
- virtual void slice_domain(…)
  - Decompose index space into subsets and distribute
  - Recursively slice subsets, specify target processor
- Index space tasks: slice into subsets of points
  - Look at is_index_space to determine if slice or single task
  - index_domain gives bounds of slice
Task Stealing

Legion also supports pull load balancing via stealing
- Stealing is totally under control of mappers
- Mappers can only steal tasks from mappers of the same kind

Stealing in Legion is two-phase
- Send requests: virtual void target_task_steal(…)
  - Choose targets for stealing (no guessing by Legion)
- Approve requests: virtual void permit_task_steal(…)
- Tasks can only be stolen from ready queues
  - Cannot steal already mapped tasks

Node 0

Steal Request
Stolen Tasks!

Node 1

Steal Request
Steal Failure!

Advertise

Invoke Permit
Invoke Permit
New Task!

http://legion.stanford.edu

December 4, 2014
Intra-Node Load Balancing

- Stealing is inefficient within a node
  - Support mapping tasks onto multiple processors in a node

- Can assign additional_procs in map_task
  - Must be of the same kind as target_proc, on same node
  - Must be able to access all physical instances
  - Legion automatically checks these properties
    - Will ignore bad processors and issue warning

- Create internal queues for running these tasks
  - Processors pull highest priority task from all their queues
Program Introspection

- Mappers can (immutably) introspect data structures
  - Region tree forest: index space trees, logical region trees
  - Task variant collections
  - Semantic tags describing tasks and regions
  - Dynamic dependence graph (computed by runtime)

- Mappers can profile task execution
  - Set profile_task to true in select_task_options
  - virtual void notify_profiling_info(Task *task)
  - Currently profile basic properties (e.g. execution time)
  - What else do we need?
Other Mapping Features

- **Tunable variables**
  - Abstract variables that depend on machine (e.g. # of partitions)
  - `virtual int get_tunable_variable(...)`

- **Virtual mappings**
  - Some tasks only need privileges, don’t need a physical inst.
  - Virtually map region requirement by setting `virtual_map`
  - Child task mapping flows back into parent context

- **Controlling speculation**
  - Mapper controls speculation on predicated tasks
  - `virtual void speculate_on_predicate(...)`
  - Don’t speculate for now, available soon 😊
Avoiding Resource Deadlocks

- Sibling tasks with a dependence cannot map until all children have mapped
  - Enforced automatically by the runtime

- Necessary to avoid resource deadlock

- Is there a better way?
Open Mapping Questions

- Resource constrained mapping
  - Right now we map one task at a time
  - Map multiple tasks together to optimize resource usage
  - Trade-off parallelism with resource usage

- Task fusion + mutation of dynamic dependence graph
  - Fuse operations to support better data re-use
    - More on this in meta-programming talk later today
  - Other manipulations on dynamic dependence graph?

- Task replication
  - Why move data when you can compute it multiple times?
  - Replicate tasks to reduce overall data movement costs