Regent and Pygion

Elliott Slaughter / SLAC National Accelerator Laboratory
Legion Retreat, December 7, 2022
Legion Programming Interfaces

C++ API

- The venerable Legion C++ API, used directly from C++ applications
- Template-based metaprogramming
- Statically type checked, but limited (or no) checking of Legion features
- Code is verbose *
- Code has to be written "just right" to execute efficiently in Legion *
- Write GPU code manually in CUDA, HIP, Kokkos, etc.
- Immediate access to bleeding edge Legion features

Regent

- Language written to target the Legion programming model
- Powerful metaprogramming via Lua
- Statically type checked (includes full checking of Legion features)
- Code is compact
- Automatically optimizes Legion API calls to improve execution efficiency without user intervention
- Automatically generate GPU code for tasks

Pygion

- Programming interface for Legion in Python
- No metaprogramming (but dynamic)
- Dynamically type checked (includes full checking of Legion features)
- Code is compact
- API optimization partially automated, requires some knowledge of “good” code patterns, but ergonomic to write
- Call Python libraries for GPU (CuPy, PyTorch, etc.)

* BYOA: Bring Your Own Abstraction
Legion is generally intended to be used with user-provided abstractions
IndexSpace colors = 
    runtime->create_index_space(ctx, Rect<1>(0, 1));
float a = 1.23;
IndexLauncher launch(
    TID_SAXPY, colors,
    TaskArgument((void *)&a, sizeof(a)),
    ArgumentMap());
launch.add_region_requirement(RegionRequirement(
    P, 0, READ_WRITE, EXCLUSIVE, S));
launch.add_region_requirement(RegionRequirement(
    P, 0, READ_ONLY, EXCLUSIVE, S));
launch.add_field(0, FID_Y);
launch.add_field(1, FID_X);
runtime->execute_index_space(ctx, launch);
Code Sample: A GPU Task

### C++ API (and CUDA)

```cpp
__global__
void gpu_saxpy(const float a,
    Rect<1> rect,
    FieldAccessor<READ_ONLY, float, 1> acc_x,
    FieldAccessor<READ_WRITE, float, 1> acc_y)
{
    int p = bounds.lo + (blockIdx.x * blockDim.x) + threadIdx.x;
    if (p <= bounds.hi)
        acc_y[p] += a * acc_x[p];
}

__host__
void saxpy(const Task *task,
    const std::vector<PhysicalRegion> &regions,
    Context ctx, Runtime *runtime) {
    FieldAccessor<READ_WRITE,float,1> acc_y(regions[0], FID_Y);
    FieldAccessor<READ_WRITE,float,1> acc_x(regions[1], FID_X);
    float a = *(const float*)(task->args);
    Rect<1> rect =
        runtime->get_index_space_domain(
            ctx, task->regions[0], region.get_index_space());
    size_t num_elements = rect.volume();
    size_t cta_threads = 256;
    size_t total_ctas = (num_elements + (cta_threads - 1))/cta_threads;
    gpu_saxpy<<<total_ctas, cta_threads>>>(a, rect, acc_x, acc_y);
}
```

### Regent

```regent
@task(privileges=[RW('y') + R('x')])
task saxpy(S : region(fields), a : float)
where reads writes(S.y), reads(S.x)
do
    for i in S do
        S[i].y += a * S[i].x
    end
end
```

### Pygion

```pygion
@task(privileges=[RW('y') + R('x')])
def saxpy(S, a):
    x = cupy.asarray(S.x)
    y = cupy.asarray(S.y)
    y += a * x
    S.y[:] = cupy.asnumpy(y)
```
What’s New in Regent

Since June 2021

• Predication
• Launcher provenance
• **AMD GPU support**
  • Updated LLVM support (*and performance-tested against old benchmarks*)
• Custom serializers
• Dynamic interference tests in index launches
• Index launches on multi-level partitions
• “Local” tasks (do not launch a Legion task)
• Much more extensive Pygion interop (now able to pass regions bidirectionally to/from Pygion)

Coming Up Next

• Intel GPU support

Further Out (?)

• More flexible assignment of regions/partitions
• Gather/scatter copies
• Compact sparse instances
• Talk to me if you need something!
AMD GPU Code Generation

- Reminder: Regent sits on top of LLVM for code generation
- The good news: LLVM supports AMD GPU
- The bad news: many un(der)documented parts of the AMD stack
  - Object format, calling convention, intrinsics, APIs...

```plaintext
__demand(__cuda)

task saxpy(S : region(fields), a : float)

where reads writes(S.y), reads(S.x)

do
  for i in S do
    S[i].y += a * S[i].x
  end
end
```
Regent Case Study: Combustion

S3D: DNS for Turbulent Combustion

• Full-scale code from Sandia National Labs, developed over many years
  • Original code: Fortran MPI
  • Ported initially to C++ Legion
  • Migrated to Regent to make the code easy and approachable for domain scientists

• Porting experience on Crusher:
  • Regent kernels worked out of the box
  • A small amount of custom CUDA code had to be ported
  • Debugged some network issues in Slingshot 11: isolated to libfabric with HPE

• Also tested on Frontier up to 512 nodes

S3D weak scaling on Crusher:

Crusher specs:
• OLCF Frontier testbed system
• 4x AMD MI-250X GPUs / node
• HPE Slingshot 11 network
• 196 nodes
What’s New in Pygion

Since June 2021

• Layout constraints
• Bidirectional Regent interop
• Substantial improvements in general Python infrastructure
  • E.g., CLI frontend to match Python

Coming Next

• Subprocesses
• Python/CUDA integration
Regent / Pygion Interop

Pygion as an Interface for Fast Scripting

- Pygion and Regent speak the same calling convention
  - Regent => Pygion: call Python libraries in a simulation
  - Pygion => Regent: call efficient Regent implementations of tasks from Python
- Pygion has the ability to call arbitrary Python code (NumPy, CuPy, scikit, etc.)
  - Convenient for rapid development
- nHTR uses PyTorch (via Pygion) for online training of ML models during the ensemble run on free CPU cores

Case Study: online PyTorch training in nHTR

[Laurent and Maeda, APS 2022]
Pygion Case Study: Single Particle Imaging

ExaFEL: ECP AD Project for X-ray Lasers

- Data analysis to reconstruct particle structure from X-ray diffraction images
- “Kitchen sink” Python code: NumPy, CuPy, Numba, custom CUDA kernels, third-party CUDA libraries
  - These libraries provide access to single-GPU implementations
- Uses Pygion for parallel, distributed, multi-GPU execution
- Speedup over MPI is due to Pygion’s ability to optimize data layout for GPU
## Python Interfaces: Pygion, Legate, FlexFlow

### Pygion
- General-purpose: call any library written in Python
- Python libraries **do not** automatically parallelize: they run on a single node / single core
- User works at the level of tasks (like Legion/Regent), just in a Python syntax

### Legate (cuNumeric)
- Interface to write parallel/distributed Python libraries that seamlessly replace the original, sequential versions
- Pitch: “change one line and your code runs distributed and on GPUs”
- Each library must be rewritten to support Legate (and the larger the library, the longer this takes)

### FlexFlow
- Domain-specific library for machine learning with support for PyTorch and Keras interfaces
- Special optimizations specific to DNNs
- Outperforms TensorFlow on some use cases
Questions?