A Programmatic Interface for Particle Plasma Simulation in Python

Preliminary Results with PyCUDA

Min Ragan-Kelley (UCB AS&T)
June 30, 2010
OUTLINE

• What is PIC
• Motivations for the Project
• Design Goals of Python PIC
• Interface Design
• Use Cases
• PyCUDA Test Problem and Results
What is PIC

\[ \nabla^2 \Phi_{[i,j]} = -\frac{\rho_{[i,j]}}{\epsilon} \]

\[ \ddot{x}(\vec{r}) = -\frac{q}{m} \nabla \Phi(\vec{r}) \]

• Particle-In-Cell (PIC) Simulations are popular in plasma physics

• E/M Fields are defined on a discrete grid

• Particles live in continuous space

• Particles do not directly interact, but rather are mediated by the grid

• PIC: Weight (p→g), Solve(g), Interpolate (g→p), Push(p)
THE PAST: XOOPIC

• Object Oriented Particle-In-Cell
• Rebuild the application if you add anything
• Good model, but not scalable
• Mouse-only interface to simulations
• Better tools exist now, that didn’t in 1995
**INTERFACE GOALS**

- Represent Device as Python object
- Provide superset of XGrafix controls as methods of that object
- Present existing diagnostics as numpy arrays
- **Allow arbitrary derivative diagnostics**
- Pure Python for input files (no custom syntax)
- Multiple Backends (PyCUDA, 2D, 3D, etc.)
- Cluster management for parallel simulations (IPython)
Performance Goals

• Exploit ubiquitous parallel hardware to study larger problems
• GPU parallelism
• Multicore/Multi-chip parallelism
• Cluster parallelism
• Short Term: build a model that will use any of these
• Long Term: use all of them
INTERFACE DESIGN
INTERFACES

• Each type of object provides an Interface

• For extended functionality, subclass an Interface

• All Diagnostics provide IDiagnostic

• All Boundary Diagnostics provide IDiagnostic AND IBoundaryDiagnostic

IDevice

IBackend
  IDEcomposedBackend

IDiagnostic
  IBoundaryDiagnostic
  IParticleDiagnostic
  IFieldDiagnostic
  ITimeDiagnostic

ISpatialRegion

IFieldSolve
  ISolvePoisson
  ISolveMaxwell

IParticleBin

ISpatialRegion
```python
import zope.interface as zi

class IDiagnostic(zi.Interface):
    data = Attribute("The Data for this diagnostic")
    interval = Attribute("The update interval")

    def save(fname=None):
        """save me to a file""

    ...

class BaseDiagnostic(object):
    zi.implements(IDiagnostic)

    @property
    def interval(self):
        return self._interval

    @interval.setter
    def interval(self, newinterval):
        self._interval = newinterval
        self.backend.change_diag_interval(self, newinterval)

    def save(self, fname):
        self.data.tofile(fname)
```
• Rather than writing a parser, input files are Python scripts, executed in a namespace.

• That namespace is scanned for **Objects** providing **Interfaces**, and a Simulation is built from them.
The Design
**Use Model**

- User builds Logical Model in Python
- Python Backend builds Native Simulation Objects (C/CUDA)
- Control and Derivative Diagnostics in Python
- Inner Loops and Native Diagnostics in C/CUDA
**Workflow**

- Solve Field
  - Compute Field Diagnostics
- Push Particles
  - Particle Diagnostics
- Weight Particles
  - defer to Python for Control

**Python**

- Collect Derivative Diagnostic dependencies
- Make adjustments
- defer to parent for Control
  - <resume worker>
- Compute Derivative Diagnostics

**Backend**

- start next step(s)
PROXY OBJECTS

• In parallel Sims: diagnostics, etc. will actually be Proxy Objects, relaying instructions to Spatial Regions

• Data will not be retrieved unless explicitly requested

• Due to data moving, diagnostics will have update intervals at each level
• The basic data structure of all Diagnostics will be NumPy Arrays.

• This gives us (and users) free access to:
  • slicing
  • native BLAS, FFT, etc.
  • rich analysis tools with SciPy etc.
  • all major Python plotting tools - Matplotlib, Gnuplot, Chaco, etc.
Use Cases
from OOPIC import *
sim = Simulation("/path/to/inputfile.inp")
defaultDiags = sim.diagnostics
sim.run() # run simulation until interrupt
sim.run(10) # run 10 steps
sim.step() # step forward once
J = sim.diagnostics["Current"]
B = sim.diagnostics["Magnetic Field"]
JcrossB = cross(J[:,6], B[:,6]) # adds JxB Diagnostic
JcrossB.interval = 10 # update every 10 timesteps
pylab.plot(sim.X, JcrossB)

def f(N):
    return N > 1e9
n = sim.diagnostics["Number"]
sim.runUntil(f, (n,)) # runs until there are 1e9 particles
Multiple Simulations

• As long as you can define your analysis, you can do it.

• Second simulation depends on results of first (sim2 could be DSMC or anybody else's code)

```python
from mystuff import *

sim1 = Simulation("sim1.py")
def stop_condition(sim):
    metric = check_some_stats(sim.diagnostics)
    if metric > threshold:
        return True
sim1.run_until(stop_condition, (sim1,))

analysis = perform_analysis(sim1)
del sim1
sim2 =Simulation("sim2.py")
update_with_analysis(sim2, analysis)
sim2.run()
```
• here we have a set of parameters that we want to find
• run a simulation, and get new input parameters from analysis
• start again with new parameters
• repeat until new parameters are close enough to the most recent input.

```python
sim = Simulation("sim0.py")
prev = None
next = init_parameters()
while prev is not None and \n    norm(next-prev) > tol:
    sim.reset()
    configure(sim,next)
    sim.run_until(stop_condition)
    prev = next
    next = analyze(sim)
```
• The UI and the Analysis systems are *the same*.

• Interfaced Based Backend model facilitates extensions of Physics

• Python provides intuitive, extensive Sim building tools

• Ground up Parallel design will scale better than XOOPIC

• PyCUDA is the first Backend
The Test Case
TEST CASE: SHEET BEAM

- Goal: Simulate a number of particles with PyCUDA (2D Cartesian)
- Direct Coulomb repulsion (short-range approximation)
- Initial Drift in +Y
- Specular bounce off X walls
- Periodic in Y (continuous infinite beam)
RESULTS
Performance

![Graph showing performance comparison between different models and configurations. The x-axis represents the value of N, and the y-axis represents GFLOPS. The graph includes lines for C1060: 256 tpb, GTX 260: 256 tpb, C1060: 64 tpb, GTX 260: 64 tpb, C1060: 32 tpb, and GTX 260: 32 tpb. Each line represents a different configuration's performance over varying values of N.]}
Zero time spent on new particles until a new set of blocks is needed
N/tpb = 7

28

Sunday, July 4, 2010
Adjusting for the difference in core clock, performance of the two chips is very similar.
• Performance is periodic in (threads/block * GPU cores)

• The GPU is most fully loaded when the work can be broken into full-sized work groups

• when the work doesn’t fit, some of the time is spent with the GPU not fully utilized
RESULTS

• Did achieve good (30% of 1 TFlop peak for C1060) performance on simple test case

• Did not get as far in exploring the problem as originally intended (comparing multiple schemes)
• Develop test code into full PIC Backend kernel

• Plug the PyCUDA simulation into the Python Diagnostics and steering part of the code

• Eventually: Port PyCUDA backend to PyOpenCL in order to use MultiCore CPU as well
SOURCES

• CS267
• ParLab Bootcamp videos and pdfs
  http://parlab.eecs.berkeley.edu/bootcampagenda
  Specifically Demmel, Catanzaro, Mattson

• NVIDIA GPU Gems
  http://developer.nvidia.com/object/gpu_gems_home.html

• NVIDIA CUDA/OpenCL Docs
  • http://www.nvidia.com/object/cuda_develop.html

• HPG/SIGGRAPH 2009
THANKS

- Fernando Perez & Brian Granger (IPython)
- Andreas Klöckner (PyCUDA/PyOpenCL)
- John Verboncoeur (Advisor)
- You (for coming)