A Programmatic Interface for Particle Plasma Simulation in Python

Min Ragan-Kelley[‡∗], John Verboncoeur[‡]

Abstract—Particle-in-Cell (PIC) simulations are a popular approach to plasma physics problems in a variety of applications. These simulations range from interactive to very large, and are well suited to parallel architectures, such as GPUs. PIC simulations frequently serve as input to other simulations, as a part of a larger system. Our project has two goals: facilitate exploitation of increasing availability of parallel compute resources in PIC simulation, and provide an intuitive and efficient programmatic interface to these simulations. We plan to build a modular backend with multiple levels of parallelism using tools such as PyCUDA/PyOpenCL and IPython. The modular design, following the goals of our Object-Oriented Particle-in-Cell (OOPIC) code this is to replace, enables comparison of multiple algorithms and approaches. On the frontend, we will use a runtime compilation model to generate an optimized simulation based on available resources and input specification. Maintaining NumPy arrays as the fundamental data structure of diagnostics will allow users great flexibility for data analysis, allowing the use of many existing powerful tools for Python, as well as the definition of arbitrary derivative diagnostics in flight. The general design and preliminary performance results with the PyCUDA backend will be presented. This project is early in development, and input is welcome.

Index Terms—simulation, CUDA, OpenCL, plasma, parallel

Introduction

Plasma physics simulation is a field with widely varied problem scales. Some very large 3D problems are long term runs on the largest supercomputers, and there are also many simple prototyping and demonstration simulations that can be run interactively on a single laptop. Particle-in-Cell (PIC) simulation is one common approach to these problems. Unlike full particle simulations where all particle-particle Coulomb interactions are computed, or fully continuous simulations where no particle interactions are considered, the PIC model has arrays of particles in continuous space and their interactions are mediated by fields defined on a grid. Thus, a basic PIC simulation consists of two base data structures and three major computation kernels. The data structures are one (or more) list(s) of particles and the grid problem, with the source term and fields defined at discrete locations in space. The first kernel (Weight) is weighing the particle contributions to the source term on the grid. The second kernel (Solve) updates the fields on the grid from the source term by solving Poisson's Equation or Maxwell's equations. The third kernel (Push) updates the position and velocity of the particles based on the field values on the grid, which involves interpolating field values from the grid locations to the particle positions.

∗ Corresponding author: minrk@berkeley.edu
‡ University of California, Berkeley

Our background in PIC is developing the Object Oriented Particle in Cell (OOPIC) project [OOPIC]. The motivation for OOPIC is developing an extensible interactive plasma simulation with live plotting of diagnostics. As with OO programming in general, the goal of OOPIC was to be able to develop new components (new Field Solvers, Boundary Conditions, etc.) with minimal change to existing code. OOPIC has been quite successful for small to moderate simulations, but has many shortcomings due to the date of the design (1995). The only way to interact with OOPIC is a mouse-based Tk interface. This makes interfacing OOPIC simulations with other simulation codes (a popular desire) very difficult. By having a Python frontend replace the existing Tk, we get a full interactive programming environment as our interface. With such an environment, and NumPy arrays as our native data structure, our users are instantly flexible to use the many data analysis and scripting tools available in Python and from the SciPy community [NumPy], [SciPy].

The performance component of the design is to use code generation to build the actual simulation program as late as possible. The later the program is built, the fewer assumptions made, which allows our code as well as compilers to maximize optimization. With respect to OOPIC, it also has the advantage of putting flexible control code in Python, and simpler performance code in C/CUDA, rather than building a single large C++ program with simultaneous goals of performance and flexibility.

Modular Design

1. Input files are Python Scripts.

With OOPIC, simulations are specified by an input file, using special syntax and our own interpreter. An input file remains, but it is now a Python script. OOPIC simulations are written in pure Python and interpreted in a private namespace, which allows the user to build arbitrary programming logic into the input file itself, which is very powerful.

2. Interfaces determine the simulation construction.

The mechanism for building a `Device` object from an input file follows an interface-based design, via `zope.interface` [Zope]. The constructor scans the namespace in which the input file was executed for object that provide our interfaces and performs the appropriate construction. This allows users to extend our functionality without altering our package, thus supporting new or proprietary components.

3. Python Objects generate C/CUDA kernels or code snippets.

Once the `Device` has been fully specified, the user invokes a compile method, which prompts the device to walk through its various components to build the actual simulation code. In the crudest cases, this amounts to simply inserting variables and code blocks in code templates and compiling them.

4. The simulation is run interactively

The primary interface is to be an IPython session [IPython]. Simple methods, such as `run()` will advance the simulation in time, `save()` dumps the simulation state to a file. But a method exposing the full power of a Python interface is `runUntil()`. `runUntil()` takes a function and argument list, and executes the function at a given interval. The simulation continues to run until the function returns true. Many PIC simulations are run until a steady state is achieved before any actual analysis is done. If the user can quantify the destination state in terms of the diagnostics, then runUntil can be used to evolve the system to a precisely defined point that may occur at an unknown time.

5. Diagnostics can be fetched, plotted, and declared on the fly

All diagnostics are, at their most basic level, exposed to the user as NumPy arrays. This allows the use of all the powerful data analysis and plotting tools available to Python users. Since the simulation is dynamic, diagnostics that are not being viewed are not computed. This saves on the communication and possible computation cost of moving/analyzing data off of the GPU. The Device has an attribute Device.diagnostics, which is a dict of all available diagnostics, and a second list, Device.activeDiagnostics, which is only the diagnostics currently being computed. Users can define new diagnostics, either through the use of provided methods, such as `cross(A,B)`, which will efficiently add a diagnostic that computes the cross product two diagnostics, or even the fully arbitrary method of passing a function to the constructor of `DerivativeDiagnostic`, which will be evaluated each time the diagnostic is to be updated. This can take a method, and any other diagnostics to be passed to the function as arguments. This way, perfectly arbitrary diagnostics can be registered, even if it does allow users to write very slow functions to be evaluated in the simulation loop.

Interfaces

Interface based design, as learned developing the parallel computing kernel of IPython, provides a good model for developing pluggable components. Each major component presents an interface. For instance, a whole simulation presents the interface IDevice. New field solvers present ISolver, all diagnostics present a simple set of methods in IDiagnostic, and more specific diagnostic groups provide extended sets, such as ITimeHistory and IFieldDiagnostic. Some common interface elements are provided below.

IDiagnostic

IDiagnostic provides the basic interface common to all Diagnostics:

- `save()`: save the data to a file, either ascii or `numpy.tofile()`
- `data`: a NumPy array, containing the data
- `interval`: an integer, the interval at which the Diagnostic's data is to be updated

IDevice

IDevice is the full simulation interface:

- `save(fname)`: dumps the full simulation state to a file
- `restore(fname)`: reciprocal of save()
- `run(steps=None)`: run either continuously, or a specified number of steps
- `step()`: equivalent to run(1)
- `runUntil(interval, f, args)`: run in batches of interval steps until f(*args) returns True.
- `diagnostics`: a list of diagnostics available
- `activeDiagnostics`: a list of diagnostics currently being evaluated
- `addDiagnostic(d)`: registers a new diagnostic to be computed, such as derivative diagnostics

Diagnostics

Diagnostics will have two classes. First class diagnostics are fast, native diagnostics, computed as a part of the compute kernel in C/CUDA. The second class of diagnostics, Derivative Diagnostics, are more flexible, but potential performance sinks because users can define arbitrary new diagnostics interactively, which can be based on any Python function.

PyCUDA tests

We built a simple test problem with PyCUDA [PyCUDA]. It is a short-range n-body particle simulation where particles interact with each other within a cutoff radius. The density is controlled, such that each particle has several (~10) interactions. The simulation was run on two NVIDIA GPUs (C1060 and GTX 260-216) with various numbers of threads per block (tpb) [C1060], [GTX260]. This was mainly a test of simple data structures, and we found promising performance approaching 40% of the theoretical peak performance on the GPUs in single precision [Figure 1].

The sawtooth pattern in Figure 1 is clarified by plotting a normalized runtime of the same data [Figure 2]. The runtime plot reveals that adding particles does not increase the runtime until a threshold is passed, because many particles are computed in parallel. The threshold is that number of particles. Since there is one particle per thread, the steps are located at intervals of the number of threads-per-block (tpb) times the number of blocks that can be run at a time (30 for C1060, and 27 for GTX-260).

Challenges

There are a few points where we anticipate challenges in this project.

First, and most basic, is simply mapping PIC to the GPU. Ultimately we intend to have backends for multi-machine simulations leveraging both multicore CPUs and highly parallel GPUs, likely with a combination of OpenCL and MPI. However, the first backend is for 1 to few NVidia GPUs with CUDA/PyCUDA. This is a useful starting point because the level of parallelism for modestly sized problems is maximized on this architecture. We should encounter many of the data structure and API issues involved. PIC is primarily composed of two problems: grid-based field solve, and many particle operations. Both of these models

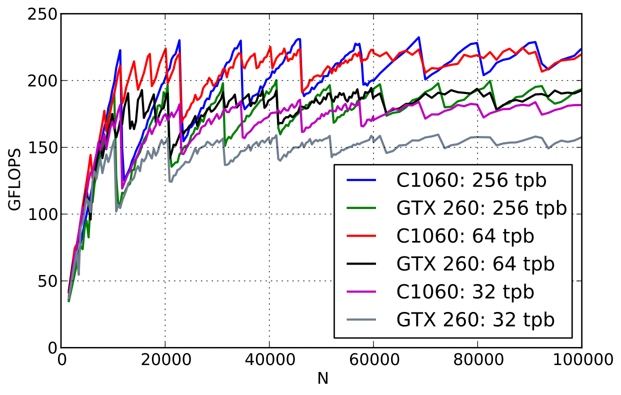

Fig. 1: *FP performance vs number of particles in the simulation (N). 230 GFLOPS is 37% of the 622 GFLOPS theoretical peak of a C1060, when not using dual-issue MAD+MUL. 'tpb' indicates threads-per-block - the number of threads allowed in each threadblock.*

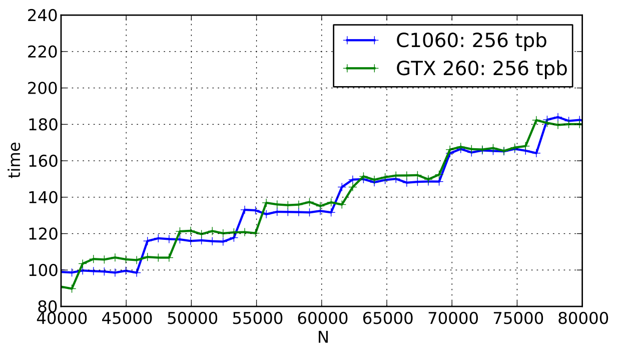

Fig. 2: *Normalized runtime increases at discrete steps of tbp* # of blocks: 256*30=7680 for C1060, and 256*27=6912 for GTX-260.*

are popular to investigate on GPUs, but there is still much to be learned about the coupling of the two.

Diagnostics also pose a challenge because it is important that computing and displaying diagnostics not contribute significantly to execution time. Some target simulations run at interactive speeds, and an important issue to track when writing Python code in general, and particularly multi-device code, is data copying.

Code generation is another challenge we face. Our intention is to build a system where the user specifies as little of the backend as possible. They enter the physics, and likely the spatial and time resolution, and our Python code generates C+CUDA code that will run efficiently. This is not easily done, but once complete will be quite valuable.

Future Plans

Ultimately we intend to have a GUI, likely built with Chaco/ETS, to replicate and extend functionality in OOPIC, as well as extending backends to fully general hardware [ETS]. But for now, there is plenty of work to do exploring simpler GPU simulations and code generation strategies behind the interactive Python interface.

The code will be licensed under the GNU Public License (GPL) once it is deemed ready for public use [GPL].

References

[OOPIC] J.P. Verboncoeur, A.B. Langdon and N.T. Gladd, *An Object-Oriented Electromagnetic PIC Code*, Comp. Phys. Comm., 87, May11, 1995, pp. 199-211.
[NumPy] http://numpy.scipy.org
[SciPy] http://www.scipy.org
[Zope] http://www.zope.org/Products/ZopeInterface
[IPython] http://ipython.scipy.org
[PyCUDA] http://mathema.tician.de/software/PyCUDA
[GTX260] http://www.nvidia.com/object/product_geforce_gtx_260_us.html
[C1060] http://www.nvidia.com/object/product_tesla_c1060_us.html
[ETS] http://code.enthought.com/projects
[GPL] http://www.gnu.org/licenses/gpl.html