Stencil Solvers for PDEs on GPUs: An Example From Cosmology

Zachary J. Weiner, *University of Illinois at Urbana-Champaign, Urbana, IL, 61801, USA*

The increasingly diverse ecosystem of high-performance architectures and programming models presents a mounting challenge for programmers seeking to accelerate scientific computing applications. Code generation offers a promising solution, transforming a simple and general representation of computations into lower level, hardware-specialized and -optimized code. We describe the philosophy set forth by the Python package Pystella, a high-performance framework built upon such tools to solve partial differential equations on structured grids. A hierarchy of abstractions provides increasingly expressive interfaces for specifying physical systems and algorithms. We present an example application from cosmology, using finite-difference methods to solve coupled hyperbolic partial differential equations on (multiple) GPUs. Such an approach may enable a broad range of domain scientists to make efficient use of increasingly heterogeneous computational resources while mitigating the drastic effort and expertise nominally required to do so.

The rapidly evolving and diversifying landscape of high performance computing poses a tension between developer productivity and machine efficiency. This tradeoff can be burdensome for domain scientists, who are often accustomed to single-source programs that function over (many) generations of computing hardware, but who also increasingly require the capabilities of cutting-edge machines to meet their science goals.

Achieving near-peak efficiency on GPUs, whose massive parallelism targets high-throughput workloads, requires (at some level) a programming model that maps transparently to the underlying hardware. Even if one is willing to exclude CPUs as a target architecture, programming at this low level is inherently fragmented based on the details of the device at hand. Among NVIDIAs offerings alone, four generations of "datacenter-class" GPUs still see active use in HPC facilities. The adoption of AMD's and Intel's accelerators by several of the U.S. Department of Energy's next generation of supercomputers portends an ecosystem further stratified by vendor-specific programming paradigms.

The challenge to produce a correct implementation of a desired mathematical computation is exacerbated by also seeking optimal performance, even on a single architecture. An ideal programming system would realize a "separation of concerns" that compartmentalizes the expression of the intended computation and the hardware-specific parallelization strategy while guaranteeing that the latter does not affect the correctness of the former. This is the approach of Loopy,[1] a Python package upon which the present work is based. Namely, Loopy parses a minimal representation of a set of computations, or "kernel," and provides a number of methods to programmatically transform that kernel into a parallelized, optimized form.

While such a framework of code transformation and generation dramatically eases the development process for GPU programs, a user must still have some degree of expertise to know *which* transformations yield performant code—of course, an interpreted runtime such as Python provides a pleasant environment in which to experiment with and benchmark different strategies. Specifying a problem domain, however, often permits a number of assumptions about the relevant computational patterns that

Digital Object Identifier 10.1109/MCSE.2021.3083547
Date of publication 28 May 2021; date of current version 23 July 2021.

restrict the scope of appropriate transformations, enabling one to provide "templates" for generating classes of kernels.

The present work describes Pystella, a type of domain-specific language (DSL) or framework for the time evolution of three-dimensional partial differential equations (PDEs). Pystella currently supports explicit time integration and finite-difference methods with periodic boundary conditions, targeting sets of nonlinearly coupled hyperbolic PDEs from early-Universe cosmology as a principal application. Its goal could be summarized as to make achieving *peak* (and portable) performance in finite-difference simulations as straightforward and friendly as general Python programming (e.g., with `numpy` arrays). We discuss how Pystella's design and implementation aim to realize this goal. We emphasize, by way of example, how it can serve as a backend for high-level simulation drivers in Python. More generally, we argue that such a hierarchy of abstractions may be key to democratizing programming for high-performance computing for a broad set of computational scientists.

TARGET USE CASE

To establish the scope of computations as relevant for optimization, we first describe the class of problems Pystella was first created to solve. For concreteness, the general structure of the (continuum) PDEs we use as a working example is

$$\frac{\partial^2 u_i}{\partial t^2} + \gamma(t)\frac{\partial u_i}{\partial t} - \nabla^2 u_i + X(u_i) = 0. \tag{1}$$

Here, $u_i = u_i(t, x, y, z)$ is a set of n_u functions of time t and space (x, y, z), and ∇^2 is the Laplacian operator, $\partial^2/\partial x^2 + \partial^2/\partial y^2 + \partial^2/\partial z^2$. In general, u_i may comprise a set of scalar fields and vector fields, and X is a nonlinear function of these fields (and their partial derivatives), which varies based on the physical model of interest. The coefficient $\gamma(t)$ is in fact a function of the spatially averaged energy within u_i—that is, a function derived from quantities summed across space. The target applications vary widely in both the form of X and the sets of fields u_i involved, including one or multiple of scalar fields, vector fields, and rank-2 tensor fields. A primary design goal is to be flexible to this wide range of problem structures. We emphasize that Pystella does not impose that problems take the form of (1), but rather allows users to specify arbitrary systems involving local operators and globally derived quantities.

The spatial dependence of the equations is discretized onto a structured grid of N^3 points with (in this case) periodic boundary conditions. The system may then be solved with the standard method of lines, using finite-difference stencils to approximate spatial derivatives and ordinary differential equation (ODE) solvers, such as Runge–Kutta methods, for time evolution. Thus, in terms of the type of kernels to be parallelized, the solver algorithm requires elementwise operations (i.e., functions that operate on values at only a single gridpoint), stencil computations (involving multiple neighboring gridpoints), and reductions (combining values across the entire grid into a single result). Reductions are required for outputting diagnostic and summary statistics as well as computing $\gamma(t)$ at each step of the solver. Other infrequent simulation outputs rely on fast Fourier transforms (FFTs), which are outsourced to other packages, and histograms.

FINITE-DIFFERENCE SIMULATIONS ON GPUS

We now briefly recap the model of parallelism relevant for GPU programming (and establish terminology). The prominent GPU computing paradigms, OpenCL and CUDA, both represent parallelism at a coarse and a fine-grained level. The coarse level, referred to as work groups (OpenCL) or blocks (CUDA), maps to individual compute cores, each of which share on-chip memory resources and have a number of threads or vector lanes for execution. These threads (or work items in OpenCL parlance) are the fine-grained level of parallelism; adjacent work items typically map to consecutive iterations of an inner loop. In contrast to standard CPU programming, core-local memory caches are managed by the user. Observe that in the terms of this abstraction, CPUs and GPUs only differ in the hardware counts (e.g., number of cores, vector lane width, and cache sizes).

Because it provides programming and machine models that are essentially isomorphic to CUDA's without being restricted to a single vendor (nor even just to GPUs), we target OpenCL as an underlying compute language. OpenCL is an open standard, and implementations can and do exist for the most prevalent HPC platforms. NVIDIAs support is robust, and though it long lagged at version 1.2, as of its R465 series driver NVIDIA is OpenCL 3.0 conformant. Beyond vendor implementations, which AMD's ROCm and Intel both provide, the Portable Computing Language (PoCL)[2] supports a variety of common CPUs and even provides a CUDA backend for NVIDIA devices, independent of NVIDIA's OpenCL support. Furthermore, Pystella's code-generation framework would drastically simplify the process of porting between platforms, as discussed in the following.

General Performance Considerations

We now heuristically describe the primary constraints on performance and the scaling of problem size for our use case. As a general principle, minimizing memory transfers between the CPU and GPU (which are nearly two orders of magnitude slower than a modern GPU's internal bandwidth) requires retaining all arrays on the GPU itself. Because the intensity of computations is relatively low (when compared to the high ratio of the throughputs of floating point operations and memory transfer), the performance of most kernels is bound by the bandwidth between the GPU's DRAM and its processors. As a result, to good approximation, the execution time of a single timestep is simply proportional to the number of reads from and writes to N^3 arrays. An important goal is to minimize this number, for instance by combining computations into as few kernels as possible (i.e., "kernel fusion").

Because the equations are second order in time, two arrays are required for each degree of freedom, as for each i both u_i and $\partial u_i/\partial t$ are integrated in time (excepting some specialized algorithms). We favor the low-storage variants of Runge–Kutta methods (LSRK), which achieve high (e.g., fourth) convergence order with only one auxiliary array for temporary storage. Next, the number of spatial derivatives required per degree of freedom i (often both the Laplacian and the gradient) requires separately computing and storing each of these in global GPU memory. Thus, the problems of interest entail, as an upper bound, 8 arrays per degree of freedom (n_u), and so the size of available GPU DRAM is the primary constraint on accessible grid sizes (rather than runtime). For example, with $N^3 = 256^3$ points, each of the (on the order of 10–20) degrees of freedom requires $1\ \mathrm{GB}$ of RAM, nearly saturating the $16-32\ \mathrm{GB}$ available on modern GPUs.

To enable scaling to larger problem sizes N^3, Pystella is also parallelized via the message passing interface (MPI) using `mpi4py`.[3] The computational grid is divided in up to two of the three dimensions (a "pencil" decomposition), with each subdomain residing on a different MPI rank (and so GPU). (A fully three-dimensional domain decomposition is not yet implemented merely because distributed FFT algorithms require at least one dimension of the grid to be fully contiguous.) The MPI communication patterns are relatively simple, requiring the sharing of boundary "halo" points between neighboring pencils as required for stencil computations. The number of such halo layers (on each face of the grid) depends on the access footprint of the employed finite-difference stencils. While we do not discuss MPI in detail, note that special care is taken to ensure that switching between single- and multirank operation is seamless, without sacrificing performance in either case.

Pystella's multi-GPU capability thus partially alleviates the constraint on problem size from the rather large number of required auxiliary arrays. The overhead of halo sharing is not significant at the $\mathcal{O}(10\mathrm{s})$ of GPUs we have had the opportunity to benchmark on (and is poised to become even less so, as the next generations of network interfaces will double and quadruple bandwidth over PCIe Gen 3). While one would ideally compute derivatives on the fly via on-chip, scratchpad-type memory, the large number of degrees of freedom (n_u) involved in our principal applications prohibits doing so efficiently given the size of these resources on current GPUs. Globally caching spatial derivatives, which are computed by a dedicated kernel, has the added benefit of simplifying the parallelization of the "physics-focused" kernels and of being flexible to an arbitrary number of degrees of freedom. Note that Pystella does not impose this choice absolutely—generalizing the time stepper code generation to permit either choice would be straightforward—but much of the workflow we describe depends upon it.

CODE GENERATION AND TRANSFORMATION

At its lowest level, Pystella serves to generate and tune computational kernels using Loopy. We first briefly describe Loopy's data model and workflow; its design and features have been detailed elsewhere.[1,4] Put simply, a computational kernel is represented by a loop domain and a set of statements. The loop domain, as its name suggests, specifies the variables used in for loops and their iteration bounds. Statements are represented as scalar-valued assignments between multidimensional arrays, with the right-hand side of the assignment given as an expression tree, composed of symbolic nodes representing variables, sums, products, and so on. For example, a sum node represents the sum of its branches, some of which may also be composite objects (e.g., a sum of products, or a sum of a quotient and a number).

For our application, the loop domain is nearly always a set of three indices spanning the grid (namely, the portion of the full N^3 grid resident on a given GPU). This fundamental representation of the domain changes under the various kernel transformations that Loopy provides. A prominent example is "splitting" a loop index into two, i.e., replacing the equivalent of

```
for i in range(N):
    f[i] = ...
```

with

```
for i_outer in range(N // n + 1):
    for i_inner in range(n):
        if i_inner + n * i_outer < N:
            f[i_inner + n * i_outer] = ...
```

To realize parallelization, one designates `i_inner` as the work item index and `i_outer` the work group, achieved with Loopy by "tagging" the loop variables.

Transformations also exist to, e.g., modify data access, such as buffering accesses to global arrays in core-local, scratchpad-type memory banks shared within a work group. Such "prefetching" operations are key for efficient stencil computations. Loopy implements a broad range of code transformations more generally.[1,4]

Loopy can interoperate with PyOpenCL[5] for runtime execution, and Pystella leverages this integration. Behind the scenes, Loopy generates OpenCL code when a kernel is first invoked, at which point it is compiled "just in time." Bear in mind that Loopy is not bound to OpenCL, nor PyOpenCL, nor even Python as a runtime environment. Loopy can generate code in a variety of languages (including CUDA, C, and others) that can be integrated into other codebases, and other integrated runtimes can be implemented. As such, were it ever necessary to do so, it would take substantially less effort to migrate away from (or abstract) Pystella's dependence on PyOpenCL than to migrate an equivalent, static codebase from OpenCL to another similar language. Deriving from the flexibility of Loopy's abstraction makes Pystella, to the extent that OpenCL-like programming models remain relevant, future-proof.

Building Blocks: The Field Object

The fundamental symbolic type in Pystella represents the (scalar, vector, etc.) fields u_i. First, the halo layers that accommodate MPI distribution make indexing the interior of the domain (the computations for which each MPI rank is responsible) cumbersome. Furthermore, indexing every access to an array (or field) is repetitive and error-prone. Pystella's symbolic representation of fields keeps track of both indexing and the offset relative to halo layers. For example,

```
>>> import pystella as ps
>>> f = ps.Field("f", offset="h")
>>> print(ps.index_fields(f))
f[i + h, j + h, k + h]
```

The call to `index_fields` expands the indexing to the (symbolic) array `f`, including offset information. At the moment, `h` is a symbolic variable itself; its value can be fixed (i.e., substituted with an integer) before the kernel is compiled to OpenCL.

We already have all the ingredients required for a mini-DSL for expressing stencil computations. Using the `shift_fields` method, we can produce the second-order accurate finite-difference approximation to $\partial f / \partial x$ (working in one dimension for conciseness):

```
>>> f = ps.Field("f", offset="h",
                 indices=("i",))
>>> d = (ps.shift_fields(f, [1])
         - ps.shift_fields(f, [-1])) / 2
>>> print(ps.index_fields(d))
(f[i + h + 1] + (-1)*f[i + h + -1]) / 2
```

Note that `shift_fields` traverses the entire expression tree passed as its first argument and shifts the indexing of any field it encounters.

```
>>> f = ps.Field("f", indices=("i",))
>>> g = ps.Field("g", indices=("i",))
>>> d = ps.shift_fields(f * g - f, [4])
>>> print(ps.index_fields(d))
f[i + 4]*g[i + 4] + (-1)*f[i + 4]
```

One can (as implemented in Pystella) express arbitrary stencils by iterating over, e.g., a dictionary mapping relative locations to coefficients:

```
f = ps.Field("f", offset="h",)
coefs = {1: 672/840, 2: -168/840,
         3: 32/840, 4: -3/840}
d8 = 0
for s, c in coefs.items():
    d8 += c * ps.shift_fields(f, [s, 0, 0])
    d8 -= c * ps.shift_fields(f, [-s, 0, 0])
```

which builds an eighth-order accurate approximation to $\partial f / \partial x$. The expression of a stencil is reduced to its most minimal, fundamental representation: One is specified solely by an enumeration of its coefficients and then generated by straightforward Python code.

`Fields` can have array shapes with dimensions beyond just the three spatial ones. To create a 3-vector `Field`,

```
>>> f = ps.Field("f", shape=(3,))
>>> print(ps.index_fields(f[1]))
f[1, i, j, k]
```

Recall that we store the time and space derivatives of f in arrays. The `DynamicField` object extends `Field` by tracking the relationship between a field and these derivatives:

```
>>> f = ps.DynamicField("f")
>>> # 0 through 3 denote t, x, y, z
>>> print(f.d(0), ps.index_fields(f.d(3)))
dfdt dfdx[2, i, j, k]
```

Even better, the provided symbolic differentiation routine understands this relationship:

```
>>> print(ps.diff(3 * f**2, "t"))
3*2*f*dfdt
```

The coupling function $X(u_i)$ often itself includes derivatives of functions of u_i, which can be conveniently computed with diff.

Parallel Primitives

Having shown how one expresses fundamental computations with Fields, we now demonstrate how to generate GPU kernels that perform them. Pystella implements a few fundamental "parallel primitives" to enable the rapid development of GPU kernels (and serve as the base drivers for all of Pystella's functionality). Note that Fields merely provide an extension to valid symbolic input; their use is not required in Pystella kernels. Indeed, the kernel creation routines internally convert all Field instances into expressions that Loopy can natively process.

For the most part, we do not emphasize performance metrics; while kernels are in general configured to maximize performance on recent NVIDIA GPUs, a user may tune or even override optimization strategies if the need arises. Rather, our purpose is to demonstrate a path to high-performance code that is accessible for Python programmers and requires minimal consideration of the underlying hardware and performance details.

Elementwise Operations

The fundamental kernel type is the elementwise map. A user creates an instance of the ElementWiseMap class, specifying the statements to be executed. One then invokes the kernel by calling this object. To create a kernel that doubles every element of an array in place, one executes

```
f = ps.Field("f")
stmnts = [(f, 2*f)]
doubler = ps.ElementWiseMap(stmnts)
```

To run on a 256^3 array of random 64-b floats, one simply calls ewmap:

```
import pyopencl as cl
import pyopencl.clrandom as clr
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
ary = clr.rand(queue, (256,)*3, "float64")
doubler(queue, f=ary)
```

The first lines, aside from required imports, create an OpenCL context, a queue for the submission of kernels, and a random array on the device. Thus, fewer than a dozen lines of Python create, compile, and run an optimized elementwise kernel.

Additional statements may be added by extending the stmnts list, and statements writing to temporary (local or thread-private) variables may be specified separately. Such kernels are parallelized with the loop-splitting strategy described earlier: The loops over spatial dimensions are split into outer and inner loops, mapping to work groups and work items, respectively. As one might expect of such a simple parallel kernel, this strategy easily saturates memory bandwidth on recent NVIDIA GPUs (e.g., the Tesla P100 and V100), as well as on an Intel Xeon E5-2683 v4 using PoCL.

While ElementWiseMap merely produces the prototypical GPU kernel using a prototypical parallelization strategy, its highly frequent use makes such a template extremely convenient, saving a substantial amount of verbose, error-prone boilerplate code.

Stencil-Type Computations

A fundamental extension of elementwise kernels adapts the parallelization strategy to accommodate stencil-type operations, relevant to, for example, convolution algorithms as well as finite differencing. Because stencils are nonlocal, involving values at multiple neighboring gridpoints, a primary optimization target is to improve data reuse. A first solution is to tile the computational domain, as done for elementwise kernels, and cache input values on each tile in on-chip scratchpad-type memory accessible to all work items in a given work group. In this manner, neighboring grid elements (whose input footprints overlap) read from this cache rather than redundantly retrieving values from global memory. While Loopy already implements transformations for such data "prefetching," we now describe a more sophisticated strategy that extends Loopy's existing prefetching transformation (and whose merging into Loopy is in progress).

The efficiency of the tiling approach becomes limited in higher dimensions (e.g., three) for stencils with a large radius or footprint. The required amount of scratchpad memory grows with dimension and stencil radius, leading to smaller "inner" tiles in which the stencil is actually computed. A solution is to tile the domain in all but one dimension (rather than all dimensions), using the remaining dimension as a "streaming" axis,[6] as depicted in Figure 1. In three dimensions, this strategy performs computations on a single two-dimensional tile (in, say, the $y-z$ plane) at a time, storing only however many layers of data (along the x

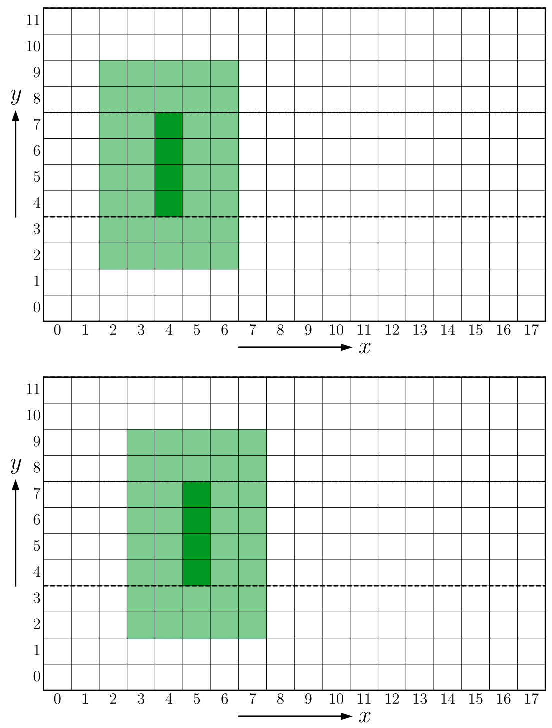

FIGURE 1. Pictorial, two-dimensional representation of the access footprint for two sequential iterations ($x = 4$ and 5) of the streaming prefetch strategy. The dark green cells make up the inner tile in which the stencil computation is performed, while light green encompasses the entire footprint required to compute the stencil. The "radius" of the footprint here is two. The columns with $3 \leq x \leq 6$ are common to the footprints for both iterations; these values need not be redundantly read from global memory as the kernel sequentially "streams" along the x direction.

direction) are required for the stencil on the current tile. Cached input data are cycled between sequential iterations over the streaming axis, thus reducing redundant global memory reads between loop iterations along this axis.

As an example of the streaming strategy's reduction of redundant global data accesses, consider computing a stencil in three dimensions with a radius of two gridpoints. With each work group assigned an 8^3 tile, the pure-tiling strategy results in $(8 + 2 \cdot 2)^3$ global reads per work group, leading to a ratio of $12^3/8^3 \approx 3.4$ total global reads to inner computational elements. The streaming strategy reduces this by a factor of $12/8$ to 2.25. The benefits are twofold, as streaming further permits larger inner work groups than tiling, increasing throughput at a fixed scratchpad memory requirement.

We report benchmark results comparing the tiling and streaming strategies in Table 1, which demonstrate that streaming prefetches offer a speedup that grows rapidly with stencil radius h. Furthermore, discounting the redundant reads due to overlapping stencil footprints across work groups, the streaming prefetch strategy achieves $\gtrsim 80\%$ of peak bandwidth for $h = 1$ and 2 and still $\sim 70\%$ for $h = 3$.

As an example, consider the eight-point, eighth-order accurate finite-difference operator defined above, `d8`. Creating a kernel employing streaming prefetching is as simple as

```
stmnts = [(Field("deriv",), d8)]
compute_d8 = ps.StreamingStencil(stmnts, prefetch_args=
["f"], halo_shape=4)
```

The pure tiling approach is accessible via the `Stencil` class.

Reductions and Histograms

For completeness, we mention the other two parallel primitives that Pystella supports. Reductions are specified by a dictionary of keys and (lists of) expressions

```
f = ps.Field("f")
reducers = {
  "mean": [f],
  "mean_sq": [f**2],
  "max": [(f, "max")]}
reducer = ps.Reduction(decomp, reducers)
```

Here, `decomp` is an instance of the `DomainDecomposition` class that implements all required MPI communication, here used to aggregate results across all MPI ranks. Calling `reducer` returns a dictionary with the same keys as `reducers` whose values are the results in the form of `numpy` arrays. An arbitrary number of reductions may be specified, achieving fusion for reduction kernels. `Histogrammer` provides a similar interface to compute histograms.

PYSTELLA IN ACTION

We demonstrate how the aforementioned parallel primitives are used to produce finite-difference simulations. We discuss Pystella's finite-differencing and time-stepping kernels and how its symbolic framework enables kernel fusion. Finally, we present an example of how one might organize the symbolic specification of physical models into a high-level, mini-DSL.

Finite Differencing

The `FiniteDifferencer` provides a convenient interface to compute the finite-difference Laplacian and/or (components of the) gradient of an array. As one frequently requires multiple spatial derivatives of the

TABLE 1. Comparing stencil optimization strategies for 3-D stencil kernels with various stencil radii h, reporting throughput in giga-gridpoints per second.

	Laplacian only			gradient+Laplacian		
h	**Tile** $[10^9/\mathrm{s}]$	**Stream** $[10^9/\mathrm{s}]$	**Speedup**	**Tile** $[10^9/\mathrm{s}]$	**Stream** $[10^9/\mathrm{s}]$	**Speedup**
1	20.7	26.0	1.25	11.3	10.6	0.946
2	7.30	18.4	2.53	6.87	9.91	1.44
3	3.64	10.3	2.81	1.83	8.77	4.78
4	0.531	5.31	10.0	0.294	4.81	16.4

For each individual kernel, results are reported for the work group size that maximizes throughput and are measured as the average of 200 runs (after two warm-up runs). Discounting redundant global reads, the Laplacian kernel reads and writes one (approximately) 512^3 array of 64-b floats, whereas the gradient+Laplacian kernel reads one and writes four. Throughputs may thus be converted to bandwidth (GB/s) by multiplying by 16 and 40 for the Laplacian and gradient+Laplacian kernels, respectively. Benchmarks were ran on a NVIDIA Tesla P100 with a (measured) peak bandwidth of $\approx 500\,\mathrm{GB/s}$. Fusing the gradient and Laplacian computations leads to much-improved GPU utilization, especially for $h = 3$ and 4.

same array, fusing combinations of individual computations into a single kernel is crucial. Calling `FiniteDifferencer` dispatches to the correct kernel based on which derivative arrays (of the Laplacian and gradient components) are passed. While one may supply custom stencils (e.g., upwind schemes), the default computes the highest order centered difference permitted by the specified number of halo layers. For further convenience, `FiniteDifferencer` ensures that halos are synchronized before calling kernels. An initialization flag specifies whether to implement the streaming optimization.

Another advantage to computing derivatives in a dedicated kernel and storing them globally is that one may swap out routines to compute derivatives via spectral collocation (i.e., computing the Fourier-space derivative via forward and backward FFTs). The `SpectralCollocator` class implements an identical interface to `FiniteDifferencer` to achieve this. For further generality, Pystella wraps both single-device, OpenCL-based FFTs (via `clfft`) and distributed, CPU-side FFTs (via `mpi4py-fft`[7]) in a common interface, so that the methods operate transparently in either single-GPU or MPI-distributed contexts.

Time Stepping

Recall that the method of lines treats the spatial and temporal discretizations separately, yielding a set of (coupled) equations that may be solved with conventional ODE solvers. Typically, ODE-solver libraries require a function `f(t, y)` that returns the right-hand side of the system, expressed in the form

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{f}(t, \mathbf{y}). \tag{2}$$

While this interface is transparent and general to a wide range of ODE solvers, it usually prevents kernel fusion. Symbolically representing the right-hand side computation, however, immediately allows the right-hand side computation and the ODE step to occur in the same, generated kernel. To create a kernel that uses a five-stage, fourth-order, LSRK method[8] to evolve the wave equation, $\partial^2 f/\partial t^2 = \nabla^2 f$, one writes

```
f = ps.DynamicField("f", offset="h")
rhs_dict = {f: f.dot, f.dot: f.lap}
stepper = ps.LowStorageRK54(rhs_dict, dt=dt, halo_shape=
halo_shape)
```

for predetermined values of `halo_shape` and the timestep `dt`.

Each stage of a LSRK method first updates the auxiliary array using `f(t, y)` and then increments `y` using the auxiliary array. These two computations can too be fused, reducing redundant reads from global memory, provided one assumes that the computation of `f(t, y)` at each gridpoint is independent of the value of `y` at all other gridpoints. Concretely, this condition is realized by globally caching all nonlocal quantities computed in a separate, prior kernel. At each gridpoint, then, `stepper` computes and stores `f(t, y)` (for each degree of freedom) in registers before performing the LSRK step.

After initializing a PyOpenCL queue and arrays `f`, `dfdt`, and `lap_f`, a single timestep takes the form

```
for stage in range(stepper.num_stages):
    derivs(queue, fx=f, lap=lap_f)
    stepper(stage, queue=queue, f=f,
            dfdt=dfdt, lap_f=lap_f)
```

where `derivs` stores the Laplacian of `f` into `lap_f` (i.e., a `FiniteDifferencer` or `SpectralCollocator`).

Sectors

To demonstrate how one might use Pystella in a real-world scenario, we present the general framework for model specification used in our primary applications in cosmology (see, e.g., Adshead *et al.*[9]). Recall that the target use case involves a set of hyperbolic PDEs with a nonlinear coupling function $X(u_i)$ and a damping coefficient $\gamma(t)$ derived from quantities (energies) averaged over the grid. For a concrete example, consider

$$X(f) = \frac{\partial V(f)}{\partial f} + (\nabla f) \cdot \nabla W(f) \tag{3}$$

where the functions V and W fully specify the physical model (in the continuum). A `Sector` would take in as input these functions, e.g.,

```
def V(f):
    return f**2 / 2 + f**4 / 4

from pymbolic.functions import exp
from pymbolic import var

def W(f):
    return exp(var("beta") * f)
```

Here, we have used the symbolic `exp` function from Pymbolic, the library that forms the basis of Loopy's symbolic framework, and have created a symbol for a model parameter `beta`.

The coupling function and energy components are encapsulated in various `Sector` classes. Internally, the `Sector` takes `V` and `W` as input and generates the `rhs_dict` along the lines of

```
f = ps.DynamicField("f", offset="h")
gamma = var("gamma")
X = ps.diff(V(f), f)
X += sum(ps.diff(f, x) * ps.diff(W(f), x)
         for x in ("x", "y", "z",))
rhs_dict = {
    f: f.dot,
    f.dot: - gamma * f.dot + f.lap + X
}
```

Additionally, the `Sector` creates a dictionary of values to reduce at each step, i.e., the energy components required to compute $\gamma(t)$. These might take the form

```
reducers = {
    "kinetic": f.dot**2 / 2,
    "gradient": (W(f)/2,
        * sum(ps.diff(f, x)**2
            for x in ("x", "y", "z",))),
    "potential": V(f)
}
```

Aside from compartmentalizing the expression of a physical model from the baggage of performance details, organizing model specification into `Sector`s (or something in the same spirit) enables composability. For example, one could introduce a `Sector` for vector fields. These have different equations of motion and energy contributions and also couple to sectors of scalar fields in a manner that would also be wholly specified by simple-to-define functions like V and W.

Another extension relevant in cosmology is the emission of gravitational waves by the scalar and vector field dynamics. Gravitational waves, a consequence of (linearized approximations to) Einstein's general relativity, are a symmetric, 3×3 tensor field sourced by particular components of the "stress-energy" tensor, which itself encodes the energy, pressure, momentum flux, and shear stress of matter. A gravitational wave sector, supplied a list of `sectors` each implementing a `stress_tensor` method, could operate as

```
h = ps.DynamicField("hij", offset="h",
                    shape=(3, 3))
rhs_dict = {}
for i in range(3):
    for j in range(3):
        rhs_dict[h[i, j]] = h.dot[i, j]
        T = sum(sector.stress_tensor(i, j)
                for sector in sectors)
        dh = h.lap[i, j] + T
        rhs_dict[h.dot[i, j]] = dh
```

(In practice, because `h` is a symmetric tensor one would only store the six nonredundant components.) The right-hand side dictionaries for all of these sectors are combined to generate a single time stepping kernel.

CONCLUSIONS

We have presented a hierarchy of abstractions that mitigate the standard tradeoff between expressive, user-friendly software and performance on cutting-edge technologies. As a foundation, OpenCL's abstract machine model promises to effectively map on to all prominent architectures (both GPUs and CPUs) for the foreseeable future, and Loopy provides a programming system that drastically eases the process of producing performant programs on them. While building higher level frameworks may inevitably require narrowing the computational scope, we have provided an example that facilitates generating high-performance code for an important class of problems.

In comparison to more traditional, static codebases (such as the CUDA-based software that Pystella replaced), one might be hesitant to take on the relatively large number of dependencies to support, e.g., Loopy. Community-driven package and environment management tools have been largely successful in enabling automated and reproducible installations: One can simply execute `pip install pystella`. Numerous more complex dependencies (for example, MPI and PoCL) can also be easily installed from the community-led software packager and distributor, conda-forge, even on cloud-based continuous integration platforms.

Aside from accelerating development and evading the need for machine-specific implementations, code generation more generally curtails the duplication of hand-written and boilerplate code, leading to substantially smaller codebases for one to maintain. While it is difficult to quantify increases in developer productivity, Pystella eliminates some of the most unpleasant and error-prone parts of high-performance programming. More broadly, its tiers of abstractions enables high-level GPU development by programmers without expertise in GPUs while also providing convenient lower level entry points for optimization and experimentation.

A number of future directions could further leverage Python as a high-level scripting language for run-time code generation. Automated performance tuning, varying not only work group sizes but also optimization strategies themselves, will be valuable as GPUs from multiple vendors become more commonplace. Programmatically tracking dependencies between kernels (including MPI communication) would allow for increased, automated concurrency while ensuring correctness. Simple first steps in this direction could analyze input to time stepping kernels to determine which spatial derivatives must be computed for each degree of freedom, and one could also implement MPI communication asynchronously with computational kernels. Finally, a crucial feature to make Pystella versatile to a broader class of problems is an expressive and efficient interface to implement nonperiodic boundary conditions.

Pystella provides an expressive, well-defined symbolic system that achieves a separation of concerns between performance optimization and the specification of physics and algorithms. Using standard Python programming practices, one can design transparent and straightforward interfaces for specific physics cases that can offload a substantial amount of symbolic work to the computer, all without compromising the performance of the resulting program. Workflows based on code generation and abstractions, like Pystella and Loopy, enable users to leverage both the advantages of Python's high-level programming environment and the potential of the target hardware. They can empower a broad community of computational scientists to maximize investments in high-performance computing to advance science.

ACKNOWLEDGMENTS

The author would like to thank Andreas Klöckner for highly generous support, encouragement, and advice throughout the development of Pystella. This work was supported by the United States Department of Energy Computational Science Graduate Fellowship, provided under Grant DE-FG02-97ER25308. This work made use of hardware provided by NASA Astrophysics Theory under Grant NNX17AG48G, as well as the Extreme Science and Engineering Discovery Environment (XSEDE),[10] which is supported by National Science Foundation under Grant ACI-1548562.

REFERENCES

1. A. Klöckner, "Loo.Py: Transformation-based code generation for GPUs and CPUs," in *Proc. ACM SIGPLAN Int. Workshop Lib., Lang., Compilers Array Program.*, 2014, pp. 82–87, doi: 10.1145/2627373.2627387.
2. P. Jääskeläinen, C. Sánchez de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "PoCL: A performance-portable OpenCL implementation," *CoRR*, vol. abs/1611.07083, 2016, doi: 10.1007/s10766-014-0320-y.
3. L. Dalcín, R. Paz, M. Storti and J. D'Elía, "for Python: Performance improvements and MPI-2 extensions," *J. Parallel Distrib. Comput.*, vol. 68, no. 5, pp. 655–662, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731507001712, doi: 10.1016/j.jpdc.2007.09.005.
4. A. Klöckner, "Loo.Py: From Fortran to performance via transformation and substitution rules," in *Proc. 2nd ACM SIGPLAN Int. Workshop Lib., Lang., Compilers Array Program.*, 2015, pp. 1–6, doi: 10.1145/2774959.2774969.
5. A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation," *Parallel Comput*, vol. 38, no. 3, pp. 157–174, 2012, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819111001281, doi: 10.1016/j.parco.2011.09.001.
6. P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proc. 2nd Workshop Gen. Purpose Process. Graph. Process. Units*, 2009, pp. 79–84, pp. 1–6, doi: 10.1145/1513895.1513905.
7. L. Dalcin, M. Mortensen, and D. E. Keyes, "Fast parallel multidimensional FFT using advanced MPI," *J. Parallel Distrib. Comput.*, vol. 128, pp. 137–150, 2019, doi: 10.1016/j.jpdc.2019.02.006.

8. M. H. Carpenter and C. A. Kennedy, Fourth-Order 2N-Storage Runge-Kutta Schemes, 1994.
9. P. Adshead, J. T. Giblin, M. Pieroni, and Z. J. Weiner, "Constraining axion inflation with gravitational waves across 29 decades in frequency," *Phys. Rev. Lett.*, vol. 124, no. 17, 2020, doi: 10.1103/PhysRevLett.124.171301.
10. J. Towns *et al.*, "XSEDE: Accelerating scientific discovery," *Comput. Sci. Eng.*, vol. 16, no. 5, pp. 62–74, Sep./Oct. 2014, doi: 10.1109/MCSE.2014.80.

ZACHARY J. WEINER is defending his Ph.D. degree in June 2021 at the University of Illinois at Urbana-Champaign, Champaign, IL, USA. His research interests include cosmology, the physics of the early Universe, gravitational waves, and high performance computing. He received the B.A. degree in physics and mathematics from Kenyon College, Gambier, OH, USA, in 2016, and the M.S. degree in physics from the University of Illinois at Urbana-Champaign in 2019. Part of his graduate studies was supported by the Department of Energy Computational Science Graduate Fellowship. Contact him at zweiner2@illinois.edu.