Flexible and high-performance stencil codes with GridTools4Py

Alberto Madonna *(madonna@cscs.ch)* **Lucas Benedicic** **Kean Mariotti** **Felipe A. Cruz**

CSCS, Swiss National Supercomputing Centre

Stencil codes

Stencil codes perform sweeps over 2D/3D arrays, computing new element values by accessing neighboring cells according to some fixed pattern, called *stencil* (Figure 1). Often associated with finite difference schemes over regular grids, stencils are more prominently employed in computational fluid dynamics, weather modeling, image processing, cellular automata approaches and PDE solving.

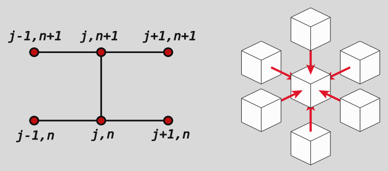

Figure 1: On the left, a 2D Crank-Nicholson stencil. On the right, a 3D 7-point stencil.

Challenges of traditional approaches

Stencil codes have been traditionally written in languages like Fortran or C/C++ (e.g. Figure 2, left panel) to achieve high performance. These approaches encounter significant challenges when working with complex scientific models:

- Advanced models require large numbers of elaborate stencils operating over many different data fields; correctly laying out the sweep ranges for an increasing amount of cascading stencils, while preventing access violations on arrays of varying shapes, quickly escalates the difficulty of the task.
- Low-level languages involve the use of repetitive boilerplate code, which affects readability and clarity.
- Fine tuning an implementation, for example by grouping stencils and data fields to improve locality, requires specific and extensive knowledge, that is often outside the scope of domain scientists.
- Code is not architecture independent. Different implementations are required to run the same operations at peak efficiency on different platforms.

C

```c
1  void horizontal_diffusion( double *data, double *weight,
2           double *diffusion, int m, int n){
3
4    /* Buffers for intermediate results */
5    double *laplacian, *flux_i, *flux_j;
6    laplacian = (double*)malloc(m*n*sizeof(double));
7    flux_i = (double*)malloc(m*n*sizeof(double));
8    flux_j = (double*)malloc(m*n*sizeof(double));
9
10   int i, j, idx;
11
12   /* Laplacian operator */
13   for (i=1; i<m-1; ++i){
14     for (j=1; j<n-1; ++j){
15       idx = j+i*n;
16       laplacian[idx] = -4.0 * data[idx] + (data[idx-n]
17                                          + data[idx+n] \
18                                          + data[idx-1] \
19                                          + data[idx+1]);
20     }
21   }
22   /* Horizontal flux */
23   for (i=2; i<m-2; ++i){
24     for (j=1; j<n-2; ++j){
25       idx = j+i*n;
26       flux_i[idx] = laplacian[idx+n] - laplacian[idx];
27     }
28   }
29   /* Vertical flux */
30   for (i=1; i<m-2; ++i){
31     for (j=2; j<-2; ++j){
32       idx = j+i*n;
33       flux_j[idx] = laplacian[idx+1] - laplacian[idx];
34     }
35   }
36   /* Diffusion */
37   for (i=2; i<m-2; ++i){
38     for (j=2; j<n-2; ++j){
39       idx = j+i*n;
40       diffusion[idx] = weight[idx] * (flux_i[idx-n]   \
41                                     - flux_i[idx]   \
42                                     + flux_j[idx-1] \
43                                     - flux_j[idx]);
44     }
45   }
46   free(laplacian); free(flux_i); free(flux_j);
47 }
```

GridTools4Py

```python
1  # Definitions function
2  def horizontal_diffusion(data, weight):
3      i = gt.Index()
4      j = gt.Index()
5
6      laplacian = gt.Equation()
7      flux_i = gt.Equation()
8      flux_j = gt.Equation()
9      diffusion = gt.Equation()
10
11     # Laplacian operator
12     laplacian[i, j] = -4.0 * data[i, j] + (data[i-1, j]
13                                          + data[i+1, j]
14                                          + data[i, j-1]
15                                          + data[i, j+1])
16     # Horizontal flux
17     flux_i[i, j] = laplacian[i+1, j] - laplacian[i, j]
18     # Vertical flux
19     flux_j[i, j] = laplacian[i, j+1] - laplacian[i, j]
20     # Diffusion
21     diffusion[i, j] = weight[i, j] * (flux_i[i-1, j]
22                                     - flux_i[i, j]
23                                     + flux_j[i, j-1]
24                                     - flux_j[i, j])
25     return diffusion
26
27 # Create computation domain
28 my_domain = gt.domain.Rectangle((2, 2), (61, 61))
29
30 # Create stencil object
31 stencil = gt.Stencil(definitions_func=horizontal_diffusion,
32                      inputs={"data": array_a,
33                              "weight": array_b},
34                      outputs={"diffusion": array_out},
35                      domain=my_domain,
36                      mode=gt.mode.ALPHA)
```

Figure 2: Comparison of implementations for a Horizontal Diffusion stencil in plain C and in GridTools4Py. In the low-level language the developer must explicitly define stencil logic, sweep ranges (lines 13-14, 23-24, 30-31, 37-38) and direct array indexes (lines 15, 25, 32, 39). The high-level abstractions of GridTools4Py decouple and hide these elements, resulting in more concise, clear and readable code, benefiting productivity.

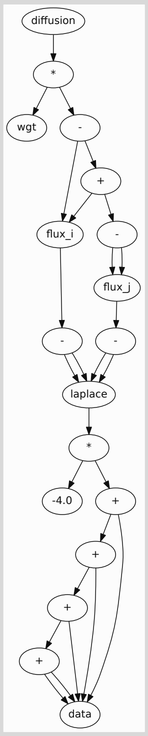

Figure 3: GT4Py abstract representation graph for the stencil of Figure 2.

Introducing GridTools4Py

GridTools4Py (GT4Py) is a Python package meant to facilitate development of effective stencil codes through a high-level, declarative syntax and a flexible execution model. A Stencil object can be created by combining a function (symbolically defining the operations that have to be performed) with the arrays and the domain region over which the stencil has to operate (Figure 2, right panel).

The power of Python's ecosystem

GT4Py works with standard NumPy arrays, meaning that all the tools available in Python's vast ecosystem are at the user's disposal. Operators can be prototyped quickly while working on interactive frontends like IPython and Jupyter notebooks. Results can be processed with any Python data analysis toolset, or visualized on the spot using popular packages like Matplotlib, Bokeh or PyQtGraph.

Automated stencil handling

Internally, GT4Py builds an abstract representation of the stencil, containing all the essential and distinctive traits of the requested mathematical operator (Figure 3). The abstract representation undergoes further analyses, providing for example the sweep ranges over intermediate data fields and array bounds checking, thus freeing the developer from directly managing these details.
The graph form of the abstract representation allows easier and more insightful analyses compared to parsing the source code.

Enabling performance and debug

When executing the stencil, the abstract representation is translated to specific code for different backends, according to user needs.
A pure Python backend can be used to immediately run the stencil for feedback or debugging by stepping into generated code.
A performance backend relies on the GridTools C++ library [1] to create a high performance module, at the cost of compilation time and debugging capabilities. GT4Py is built on a modular architecture, meaning that new backends can be supported by adding modules that generate code starting from the abstract stencil representation.

Conclusions

In this work, we presented GridTools4Py, a Python package to define stencil operators at a high level, leveraging Python's features and rich ecosystem to enable a fast and interactive development cycle. Users have the flexibility of executing stencil codes directly in Python for prototyping, or compile them into a high performance module, using the GridTools C++ library, all the while working from powerful frontends like Jupyter notebooks.

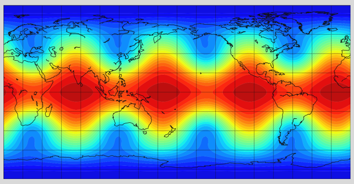

Figure 4: Fluid height for a Shallow Water Equations on the Sphere solver implemented with GT4Py. Image credit: S.Ubbiali

References

[1] GridTools - C++ libraries for applications on grids, available at *http://eth-cscs.github.io/gridtools/* (accessed March 2017).