High-level Programming Comparison

Xinyuan Hou
Lukas Joss
Marco Kronenberg

Supervisor: Stefano Ubbiali

7. August 2020

Abstract

In this project, three high-level programming techniques are used to solve the three dimensional heat equation on different-sized grids, with the aim to quantify their performance in terms of speed and evaluate their user-friendliness. Compared are implementations in `numpy`, `cupy`, `Devito`, and `GT4Py`. In terms of speed, the two domain specific languages `GT4Py`, `Devito` outperform `numpy` on CPU, while on GPU only `GT4Py` is able to outperform a `cupy` implementation. In terms of user-friendliness, we conclude that both `GT4Py` and `Devito` are generally very readable, allowing intermediate coder to write reasonably fast code. However, both have a somewhat limited documentation as of now.

1 Introduction

In our project for the course High Performance Computing of Weather and Climate, we solve a three dimensional heat equation

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \tag{1}$$

where α denotes the thermal diffusivity and $(x, y, z) \in \Omega := (0, L)^3, t > 0$.
The initial conditions are

$$u(x, y, z, 0) = \begin{cases} T_{hot}, & (x, y, z) \in \Omega_{in} := (\frac{L}{4}, \frac{3L}{4})^3, \\ T_{cool}, & (x, y, z) \in \Omega \setminus \Omega_{in} \end{cases} \tag{2}$$

with the boundary condition

$$u(x, y, z, t) = T_{cool}, \quad (x, y, z) \in \partial\Omega, t > 0 \tag{3}$$

We discretize the heat equation as

$$\begin{aligned} \frac{u^{n+1}_{i,j,k} - u^n_{i,j,k}}{\Delta t} = \alpha \Bigg[& \frac{u^n_{i+1,j,k} - 2u^n_{i,j,k} + u^n_{i-1,j,k}}{(\Delta x)^2} \\ & + \frac{u^n_{i,j+1,k} - 2u^n_{i,j,k} + u^n_{i,j-1,k}}{(\Delta y)^2} \\ & + \frac{u^n_{i,j,k+1} - 2u^n_{i,j,k} + u^n_{i,j,k-1}}{(\Delta z)^2} \Bigg] \end{aligned} \tag{4}$$

where we used the Euler step in time and central difference in space. Solving for $u_{i,j,k}^{n+1}$ yields

$$\begin{aligned} u_{i,j,k}^{n+1} = u_{i,j,k}^{n} &+ \frac{\alpha \Delta t}{(\Delta x)^2}(u_{i+1,j,k}^{n} - 2u_{i,j,k}^{n} + u_{i-1,j,k}^{n}) \\ &+ \frac{\alpha \Delta t}{(\Delta y)^2}(u_{i,j+1,k}^{n} - 2u_{i,j,k}^{n} + u_{i,j-1,k}^{n}) \\ &+ \frac{\alpha \Delta t}{(\Delta z)^2}(u_{i,j,k+1}^{n} - 2u_{i,j,k}^{n} + u_{i,j,k-1}^{n}). \end{aligned} \tag{5}$$

In order to satisfy the stability condition, the time step has to adhere to the CFL criterion:

$$\Delta t \leq \frac{\alpha}{2}\left(\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{1}{(\Delta z)^2}\right)^{-1} \tag{6}$$

Using `numpy`, `cupy`, `GT4Py` and `Devito`, we conduct the usability and performance analysis on CPU and GPU.

2 Usability

2.1 `numpy` and `cupy`

The operator written in `numpy` takes an in-field and an out-field to compute the step update under the pre-defined spatial and temporal discretization.

```
out[1:-1, 1:-1, 1:-1] = (in[1:-1, 1:-1, 1:-1] + alpha * dt * (
(in[2:  , 1:-1, 1:-1] - 2* in[1:-1, 1:-1, 1:-1] + in[0:-2, 1:-1, 1:-1]) / dx**2  +
(in[1:-1, 2:  , 1:-1] - 2* in[1:-1, 1:-1, 1:-1] + in[1:-1, 0:-2, 1:-1]) / dy**2  +
(in[1:-1, 1:-1, 2:  ] - 2* in[1:-1, 1:-1, 1:-1] + in[1:-1, 1:-1, 0:-2]) / dz**2))
```

The boundaries in every dimension are set to the cool temperature at each step. The implementation in `cupy` is similar to that in `numpy` by adding:

```
try:
    import cupy as cp
except ImportError:
    cp = np
```

2.2 `GT4Py`

At the core of the `GT4Py` (Parades et al., 2020) implementation lies the definition for the Laplace operator $\nabla^2\phi$.

```
def laplace_gt4py_3D(phi, dx, dy, dz):
    lap = ((-2 * phi[0,0,0] + phi[1, 0, 0] + phi[-1, 0, 0])/dx**2 +
           (-2 * phi[0,0,0] + phi[0, 1, 0] + phi[ 0,-1, 0])/dy**2 +
           (-2 * phi[0,0,0] + phi[0, 0, 1] + phi[ 0, 0,-1])/dz**2 )
    return lap
```

It uses the discretization presented in Equation 4. To define the stencil, an origin as well as a domain has to be provided. The stencil is compiled using the `gtscript.stencil` function based on the provided definition. In it, the computation is specified, i.e. whether computation should be parallelized or performed in sequence and the interval is defined.

```
    with computation(PARALLEL):
        with interval(1,-1):
            u_next = u_now + dt*alpha*laplace_gt4py_3D(u_now, dx, dy, dz)
```

Here lies a potential pitfall for newcomers to GT4Py. The `interval` command handles the third/vertical dimension only. Thus, additional attention has to be paid to the definition of `origin` and `domain` for the stencils. Unfortunately, the documentation for GT4Py is not yet finished and only three small examples of applications and a quick start overview are provided, which can be seen as a bit lacking.

However, the code is overall easy to read, once one becomes accustomed to the relative indices in the stencil definition.

2.3 Devito

For Devito (Louboutin et al., 2019) one first has to define the computational grid whose size has to match the input field. The extent defines the total physical size of the domain.

```
grid = Grid(shape = (nx, ny, nz), subdomains = (mid, ),
extent = (2., 2., 2.))
```

Since the stencil which is defined further below is not applied on the whole domain, one has to define a subdomain on which one will eventually apply the stencil.

```
class Middle(SubDomain):
    name = 'middle'
    def define(self, dimensions):
        d = dimensions
        return {d: ('middle', 1, 1) for d in dimensions}
mid = Middle()
```

Now we can define the function, specifying the time and space order, and assign the field:

```
u = TimeFunction(name='u', grid=grid, time_order=1,
                 space_order=2, dtype=np.float64)
u.data[:, :, :, :] = in_field
```

Then we can define the differential equation and create the stencil by solving the equation. We do not have to solve the equation ourselves, Devito takes care of it.

```
eq = Eq(u.dt, alpha * ((u.dx2) + (u.dy2) + (u.dz2)))
stencil = solve(eq, u.forward)
eq_stencil = Eq(u.forward, stencil, subdomain = grid.subdomains['middle'])
```

Next, we define the operator which iteratively applies the stencil on the subdomain and applies the boundary condition on the border; platform defines if Devito compiles the code for GPU or CPU.

```
op = Operator([eq_stencil]+bc, platform=platform)
op(time=nt, dt=dt)
```

Afterwards Devito creates a C file containing the stencil with all provided information and then compiles the C file for computing the result. In the outer loops of the stencil the code performs blocking in x- and y-direction when iterating over the array to make use of the cache and then uses `omp` in z-direction for parallelization as seen in the code snippet below which contains the stencil for CPU. Since this code is written by Devito, it is harder to read, for the variables are rather cryptic, but we can still see the stencil shape as seen in `numpy` and GT4Py.

```
#pragma omp simd aligned(u:32)
for (int i1z = i1z_ltkn + z_m; i1z <= -i1z_rtkn + z_M; i1z += 1){
  double r3=-2.0*u[t0][i1x+2][i1y+2][i1z+2];
  u[t1][i1x+2][i1y+2][i1z+2]=dt*((r3+1.0*(u[t0][i1x+2][i1y+2][i1z+1]+
  u[t0][i1x+2][i1y+2][i1z+3]))/((h_z*h_z))+(r3+1.0*(u[t0][i1x+2][i1y+1][i1z+2]+
  u[t0][i1x+2][i1y+3][i1z+2]))/((h_y*h_y))+(r3+1.0*(u[t0][i1x+1][i1y+2][i1z+2]+
  u[t0][i1x+3][i1y+2][i1z+2]))/((h_x*h_x))+u[t0][i1x+2][i1y+2][i1z+2]/dt);}
```

3 Performance

3.1 The test case and validation

The timings were performed on Piz Daint. The timing was performed with 300 time steps of size $dt = \frac{\alpha}{10000}\left(\frac{1}{dx^2} + \frac{1}{dy^2} + \frac{1}{dz^2}\right)^{-1}$ to guarantee a non-violation of the CFL Condition (6). The physical extent was defined by 2 in each dimension and we used the thermal diffusivity of air $\alpha = 19\ mm^2 s^{-1}$ at 300 K (EngineeringToolBox, 2018). The initial condition was defined as in the introduction and $T_{hot} = 400$ K and boundary condition $T_{cold} = 300$ K.

To ascertain the equivalency of our different implementations, a quick test was conducted, using a single case with a grid extension of $2^5 = 32$ points per dimension and slightly changed condition on thermal diffusivity, and hot as well as cold initial condition.

3.2 CPU

The timing on CPU follows an exponential growth using `numpy`. As the size of the grid increases, non-surprisingly the computational cost grows with it. `GT4Py` behaves very similarly, once a certain grid size is reached. Below it, we observe a substantial overhead, only once grid-sizes of more than $2^5 = 32$ grid points per dimension are reached, `GT4Py` starts to outperform `numpy`. At around the same point, a seemingly exponential growth of timing starts to kick in for `GT4Py`. In the case of `Devito`, we observe a very similar behavior, with an even larger overhead which only vanishes for grid sizes above $2^7 = 128$, likely because the C code only gets compiled once the operator is called. Above this threshold, `Devito` outperforms its competition.

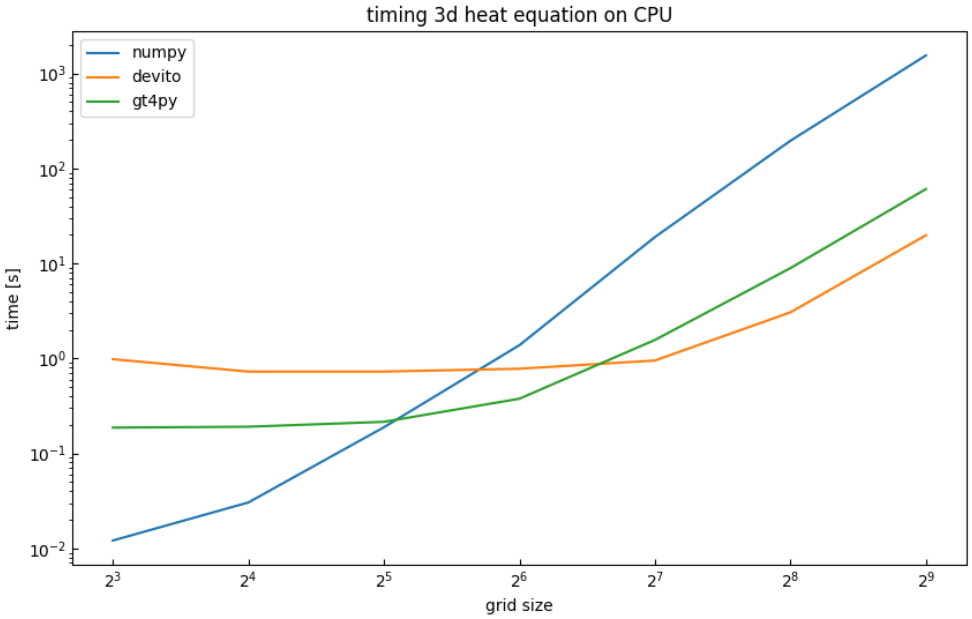

Figure 1: Runtime on CPU, 300 iterations

3.3 GPU

On GPU, integrations were performed faster using `cupy` (as compared to `numpy`) and using `GT4Py`. Especially the speed up for larger grid sizes in `cupy` is remarkable, with an order of two magnitudes difference for the largest grid size. In `GT4Py`, we observe a smaller difference for the largest grid size, with only one order of magnitude difference. As before, it outperforms `cupy`, once a threshold is met. A special case is `Devito`, especially at the lowest end of our grid sizes. Here, we observe a longer runtime than for an increased grid size, we again suspect that this might have something to do with compiling when applying the stencil. Even for larger grid sizes we do not observe a substantial speed-up for `Devito`.

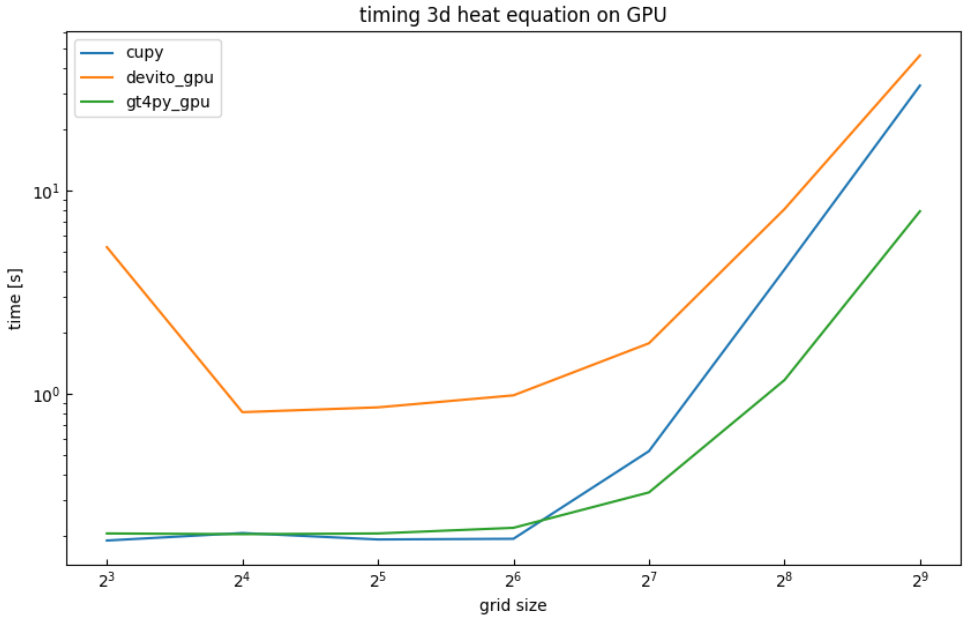

Figure 2: Runtime on GPU, 300 iterations

4 Conclusion

As we have seen, `GT4Py` and `Devito` outperform `numpy` on CPU, once a certain grid size is used. In our case, the threshold lies in around $2^6 = 64$ grid points in all three dimensions. On GPU, we would not recommend the use of `Devito` in terms of performance, since `cupy` outperforms it. For smaller grids, i.e. below $2^7 = 128$ grid points per dimension - `GT4Py` boasts around the same runtime as `cupy`, and outperforms it at larger sizes.

In the context of weather and climate models, this threshold is easily surpassed, with for example the COSMO-1 model employed by MeteoSwiss using a vertical grid of 1075×691 points and 80 vertical layers (MeteoSwiss, 2020). Of course, this particular model does not solve the heat equation presented in this report, but at the heart of our problem lies the Laplace operator and with its second order derivatives, thus highlighting potential skill in performance even for such a large use case - though the benchmark in question would be potentially even lower.

Concerning usability, both domain specific languages used reach their goal of creating fast and easily readable code. Both are not without fault though.

`GT4Py` is easy to adapt to, coming from a `numpy` implementation of a given stencil. Loops are replaced with element-wise operations using relative indices in an easy-to-understand manner. Somewhat more cumbersome is the definition for the stencils and special care has to be taken while handling `origin` and `backend` arguments, but those challenges can be easily overcome with enough practice.

One more serious drawback of `GT4Py` is the rather limited documentation. It consists basically of three different use cases and a short quick start guide summarizing the use cases. This is sufficient in most cases, but more in-depth challenges could be hard to solve with such limited resources. Fortunately, a more in-depth documentation is in the works.

`Devito` uses a completely different approach concerning implementation of a given problem from `GT4Py` or `numpy`. A symbolic approach is employed, removing the need to discretize a given problem, which makes it really easy to use since a lot is hidden from the user so that one does not have to think about it. For `devito`, the documentation is good for simple examples, which the tutorial covers thoroughly, for more advanced things (e.g. compile flags) the documentation, as with `GT4Py`, gets sparse. Again, only a few use cases are provided. This is especially problematic given the wide variety of information available for more mainstream techniques based on `numpy`. They however come with the price of much longer run times, especially critical if one has to solve a large-scale problem.

References

EngineeringToolBox (2018). *Air - Thermal Diffusivity*. URL: `https://www.engineeringtoolbox.com/air-thermal-diffusivity-d_2011.html`. (accessed: 07.08.2020).

Louboutin, M., Lange, M., Luporini, F., Kukreja, N., Witte, P. A., Herrmann, F. J., Velesko, P., and Gorman, G. J. (2019). "Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration". In: *Geoscientific Model Development* 12.3, pp. 1165–1187. DOI: `10.5194/gmd-12-1165-2019`. URL: `https://www.geosci-model-dev.net/12/1165/2019/`.

MeteoSwiss (2020). *COSMO forecasting system*. URL: `https://www.meteoswiss.admin.ch/home/measurement-and-forecasting-systems/warning-and-forecasting-systems/cosmo-forecasting-system.html`. (accessed: 07.08.2020).

Parades, E., Groner, L., Dahm, J., Davis, E., Ehrengruber, T., and Vogt, H. (2020). *Gridtools/Gt4py*. URL: `https://github.com/GridTools/gt4py`. (accessed: 07.08.2020).