Simulating Shock Waves Using a Domain-Specific Language (DSL) in Python

HPC4WC REPORT

Ruoyi Cui
Shuchang Liu
Shihao Zeng

Supervisor
Stefano Ubbiali

September 15, 2021

Contents

1 Introduction

The higher-order problems may lead to over- and undershoot around a jump discontinuity, which is known as the Gibbs phenomenon. Harten et al. (1987) first introduced Essentially non-oscillatory schemes (ENO) to deal with the discontinuities and numerical oscillations. Later, Liu et al. (1994) further developed the weighted ENO (WENO) to fine-tune the stencil selection process that enhances stability and accuracy. Compared to earlier attempts to the conservation law, e.g., total variation diminishing (TVD) method can only yields first-order accuracy near smooth extrema, while ENO/WENO may achieve arbitrarily and uniformly high order accuracy. The fifth-order finite difference WENO scheme (WENO5) has been successfully used in many applications.

As discussed before, lower-order methods have difficulties solving the problems with both strong shocks and smooth structures. Thus, it's attractive to use the aforementioned WENO method to simulate shock waves. In this project, we consider the 2D shock waves with the inviscid Burgers' equation:

$$u_t + (\frac{u^2}{2})_x + (\frac{u^2}{2})_y = 0 \tag{1}$$

that subject to the periodic boundary condition and the initial state given by:

$$u(x, y, 0) = u_0(x, y) \ = sin(2\pi x) \tag{2}$$

Figure 1 below shows the numerical solution of the equations (1,2) using WENO5 method, where the shock wave is very well captured.

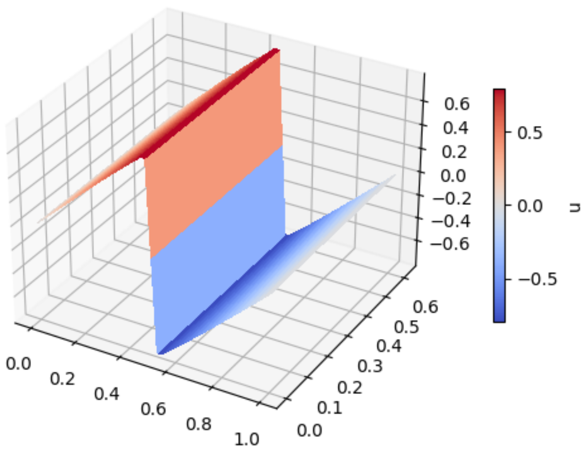

Figure 1: WENO5 solution for Burger's equation.

GridTools For Python (GT4Py) is an open-source domain-specific Python library designed for high-performance stencil computations. It uses the GridTools Framework, thus the code can be written in a stencil-like pattern and optimized for different architectures. Motivated by this, we use GT4Py to implement stencils in the WENO5 scheme to simulate shock waves and evaluate the performance using different backends. The performance assessment was done on Piz Daint at the Swiss National Supercomputing Center (CSCS).

2 Implementation

2.1 Interpolation and reconstruction

The main idea of ENO/WENO scheme is the reconstruction procedure. As illustrated in Figure 2, the mesh size is assumed to be constant $\Delta x = x_{i+1} - x_i$, and the half grid points $x_{i+\frac{1}{2}} = \frac{1}{2}(x_i + x_{i+1})$ are considered as computational cell interfaces. Now consider the cell average:

$$\overline{u} = \frac{1}{\Delta x}\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} u(x)dx \tag{3}$$

the goal is to find an approximation of $u(x)$ at half grid points $x_{i+\frac{1}{2}}$. Once the cell average values are known, the values at all half grid points can be obtained from:

$$U(x_{i+\frac{1}{2}}) = \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} u(\xi)d\xi = \Delta x \sum \overline{u}_i \tag{4}$$

Take the first stencil S_0 as an example. Construct a polynomial P(x) of order $m = 3$ interpolates the function $U(x)$ at $m+1$ interfaces $(x_{i-\frac{1}{2}}, i = 0, ..., m)$. Let $p_0(x) = P_0'(x)$:

$$(\overline{p}_0)_j = \frac{1}{\Delta x}\int_{x_{j-\frac{1}{2}}}^{x_{j+\frac{1}{2}}} p(x)dx = \overline{u}_j, \quad j = i, i+1, i+2 \tag{5}$$

that yields $m = 3$ order accuracy approximation $p_0(x_{i+\frac{1}{2}}) - u(x_{i+\frac{1}{2}}) = O(\Delta x^3)$ if $u(x)$ is smooth in the stencil S_0. Finally, the reconstruction can be expressed as:

$$u^0_{i+\frac{1}{2}} = c_0\overline{u}_i + c_1\overline{u}_{i+1} + c_2\overline{u}_{i+2}, \quad \sum_{s=0}^{m-1} c_s = 1 \tag{6}$$

where the explicit coefficients can be solved for the stencil S_0 with $c_0 = \frac{1}{3}, c_1 = \frac{5}{6}, c_2 = -\frac{1}{6}$. For ENO approximation, one of 3 approximations $(u^0_{i+\frac{1}{2}} u^1_{i+\frac{1}{2}}$ and $u^2_{i+\frac{1}{2}})$ based on 3 stencils $(S_0, S_1$ and $S_2)$ will be selected based on the local smoothness of the reconstruction polynomials, measured by devided differences (Shu (1998)).

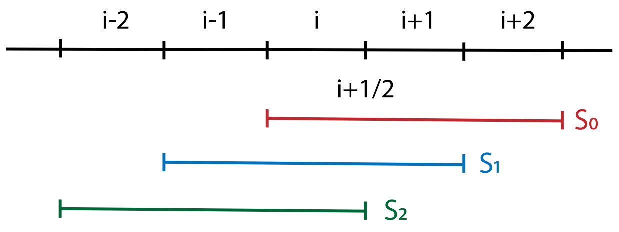

Figure 2: Illustration of 5th order WENO5 stencil with the associated three 3rd order ENO stencils S_0, S_1 and S_2.

2.2 WENO approximation

WENO approximation is based on ENO, instead of using only one of the stencils, it uses a convex combination of all the associated ENO stencils $(S_0, S_1$ and $S_2)$ and associated weights to each stencil based on its local smoothness indicator. With this, a more smooth numerical flux than ENO can be obtained with higher accuracy:

$$u_{i+\frac{1}{2}} = \gamma_0 u^0_{i+\frac{1}{2}} + \gamma_1 u^1_{i+\frac{1}{2}} + \gamma_2 u^2_{i+\frac{1}{2}}, \quad \sum_{s=0}^{m-1} \gamma_s = 1 \tag{7}$$

when $m = 3$, $\gamma_0 = \frac{1}{10}, \gamma_1 = \frac{3}{5}, \gamma_2 = -\frac{3}{10}$ yields $O(\Delta x^{2m-1})$ approximation. For Burger's equation, where $u(x)$ is not always smooth, the non-linear weights are adopted accordingly to avoid the discontinuities in the stencil. The detailed implementations will be discussed in the following sections. With this, a more smooth numerical flux than ENO approximation can be obtained with higher accuracy.

The code example for WENO approximation is shown as follows:

```
@gtscript.stencil(backend=backend)
def BurgersWENO2D(du:Field[float], xe:Field[float], ue: Field[float], um:Field[float],
up:Field[float], dx: float, m:int, p:int, eps:float, maxvel:float):
    with computation(PARALLEL), interval(0,1):
      if m == 1:
        um = ue
        up = ue

      else:
        # Compute smoothness indicators for u-/u+ based on different stencils
        upl0 =  1/3 * ue[ 0, 0, 0] + 5/6 * ue[ 1, 0, 0] -  1/6 * ue[2, 0, 0]
        upl1 = -1/6 * ue[-1, 0, 0] + 5/6 * ue[ 0, 0, 0] +  1/3 * ue[1, 0, 0]
        upl2 =  1/3 * ue[-2, 0, 0] - 7/6 * ue[-1, 0, 0] + 11/6 * ue[0, 0, 0]

        uml0 = 11/6 * ue[ 0, 0, 0] - 7/6 * ue[ 1, 0, 0] + 1/3 * ue[2, 0, 0]
        uml1 =  1/3 * ue[-1, 0, 0] + 5/6 * ue[ 0, 0, 0] - 1/6 * ue[1, 0, 0]
        uml2 = -1/6 * ue[-2, 0, 0] + 5/6 * ue[-1, 0, 0] + 1/3 * ue[0, 0, 0]

        beta0 = 10/3 * ue[0, 0, 0] * ue[0, 0, 0] - 31/3 * ue[0, 0, 0] * ue[1, 0, 0] \
              + 25/3 * ue[1, 0, 0] * ue[1, 0, 0] + 11/3 * ue[0, 0, 0] * ue[2, 0, 0] \
              - 19/3 * ue[1, 0, 0] * ue[2, 0, 0] + 4/3  * ue[2, 0, 0] * ue[2, 0, 0]

        beta1 = 4/3 * ue[-1, 0, 0] * ue[-1, 0, 0] - 13/3 * ue[-1, 0, 0] * ue[0, 0, 0] \
              + 13/3 * ue[0, 0, 0] * ue[ 0, 0, 0] +  5/3 * ue[-1, 0, 0] * ue[1, 0, 0] \
              - 13/3 * ue[0, 0, 0] * ue[ 1, 0, 0] +  4/3 * ue[ 1, 0, 0] * ue[1, 0, 0]

        beta2 = 4/3 * ue[-2, 0, 0] * ue[-2, 0, 0] - 19/3 * ue[-2, 0, 0] * ue[-1, 0, 0] \
             + 25/3 * ue[-1, 0, 0] * ue[-1, 0, 0] + 11/3 * ue[-2, 0, 0] * ue[ 0, 0, 0] \
             - 31/3 * ue[-1, 0, 0] * ue[ 0, 0, 0] + 10/3  * ue[0, 0, 0] * ue[ 0, 0, 0]

        # Compute alpha weights - classic WENO
        alpham0 = (1/10) / (beta0 + eps) ** (2 * p)
        alpham1 =  (3/5) / (beta1 + eps) ** (2 * p)
        alpham2 = (3/10) / (beta2 + eps) ** (2 * p)

        alphap0 = (3/10) / (beta0 + eps) ** (2 * p)
        alphap1 =  (3/5) / (beta1 + eps) ** (2 * p)
        alphap2 = (1/10) / (beta2 + eps) ** (2 * p)

        # Compute nonlinear weights and cell interface values
        um = (alpham0 * uml0 + alpham1 * uml1 +  alpham2 * uml2) / \
             (alpham0 + alpham1 + alpham2)

        up = (alphap0 * upl0 +  alphap1 * upl1 +  alphap2 * upl2) / \
             (alphap0 + alphap1 + alphap2)

  with computation(FORWARD), interval(...):
    # Evaluate Lax Friedrich numerical flux
    du =  - (BurgersLF(up[1, 0, 0], um[2, 0, 0], 0, maxvel)-
             BurgersLF(up[0, 0, 0], um[1, 0, 0], 0, maxvel)) / dx
```

2.3 Smoothness indicator and non-linear weight

In order to get the non-linear weights, the smoothness indicators are needed, which measure the smoothness of the function in the stencil. The smaller the β_j, the smoother the $u(x)$ in the stencil Sj. It can be written explicitly when $m = 3$:

$$\begin{aligned}\beta_0 &= \frac{13}{12}(\overline{u}_j - 2\overline{u}_{j+1} + \overline{u}_{j+2})^2 + \frac{1}{4}(3\overline{u}_j - 4\overline{u}_{j+1} + \overline{u}_{j+2})^2 \\ \beta_1 &= \frac{13}{12}(\overline{u}_{j-1} - 2\overline{u}_j + \overline{u}_{j+1})^2 + \frac{1}{4}(\overline{u}_{j-1} - \overline{u}_{j+1})^2 \\ \beta_2 &= \frac{13}{12}(\overline{u}_{j-2} - 2\overline{u}_{j-1} + \overline{u}_j)^2 + \frac{1}{4}(\overline{u}_{j-2} - 4\overline{u}_{j-1} + 3\overline{u}_j)^2\end{aligned} \tag{8}$$

with the smoothness indicators $\beta_{s=0,1,2}$, the non-linear weights can be calculated as follows:

$$w_s = \frac{\alpha_s}{\alpha_0 + \alpha_1 + \alpha_2}, \quad \alpha_s = \frac{\gamma_s}{(\epsilon + \beta_s)^{2p}}, \quad s = 0, 1, 2 \tag{9}$$

where $\epsilon = 10^{-6}$ to avoid the denominator being zero, and $p = 1$ to alter the impact of the smoothness indicator (Hesthaven (2018)). The final approximation at cell interfaces as a convex combination of 3 ENO stencils gives:

$$\hat{f}_{i+\frac{1}{2}} = \sum_{s=0}^{2} w_s \hat{f}^s_{i+\frac{1}{2}}, \quad \sum_{s=0}^{2} w_s = 1 \tag{10}$$

Consider three α weights on each side of the cell interface, temporary variables are introduced to store these intermediate results, as shown in line (11-25). In practice, arrays in GT4Py are capable of using more than three dimensions, which indicates more readable codes without usage of temporary variables. Further attempts to transform the 3 temporary variables needed for WENO5 (e.g., $\alpha_0[:,:,:]$, $\alpha_1[:,:,:]$, $\alpha_2[:,:,:]$) from 3D to 4D (e.g., $\alpha[:,:,:][:]$). Also, exact values of non-stencil variables (e.g., the smoothness indicator $\beta_{s=0,1,2}$) are resolved, and the performance of higher-dimensional implementation is then evaluated in Section 3.

2.4 Lax-Friedrich flux

Until now, for each u_i, we already have the approximation at cell interfaces $u^-_{i\pm\frac{1}{2}}$ and $u^+_{i\pm\frac{1}{2}}$. Finally, we apply the Lax-Friedrich monotone flux to evaluate the numerical flux with first-order accuracy. The integral form of the conservation law reads:

$$\frac{du_i(t)}{dt} = -\frac{1}{\Delta x_i}[\hat{f}(u^-_{i+\frac{1}{2}}, u^+_{i+\frac{1}{2}}) - \hat{f}(u^-_{i-\frac{1}{2}}, u^+_{i-\frac{1}{2}})] \tag{11}$$

with the Lax-Friedrich flux:

$$\hat{f}(u^-_{i+\frac{1}{2}}, u^+_{i+\frac{1}{2}}) = \frac{1}{2}[f(u^-_{i+\frac{1}{2}}) + f(u^+_{i+\frac{1}{2}}) - max_u|f(u)|(u^-_{i+\frac{1}{2}} - u^+_{i+\frac{1}{2}})] \tag{12}$$

this function is included as **BurgersLF** function as shown below and directly called in the stencil (line 47-50). Note that the approximation $f^+(u_{i+\frac{1}{2}})$ takes one more point to the left in the stencil, while $f^-(u_{i+\frac{1}{2}})$ takes one more point to the right. The size of the halo region to update u is m.

```
@gtscript.function
def BurgersLF(u, v, lam, maxvel):
    """Evaluate global Lax Friedrich numerical flux for Burgers equation"""
    return (u ** 2 + v ** 2) / 2 - maxvel / 2 * (v - u)
```

2.5 Strong stability perserving Runge-Kutta (SSP-RK) scheme

Since we apply a high order WENO5 for spatial approximation to solve Burger's equation, the higher-order time integration method is preferred. Otherwise, the oscillation will emerge as a result of temporal integration. Therefore, we coupled the 5th order WENO method (WENO5, $m = 3$) with the 3rd order 3-stage Strong Stability Preserving Runge Kutta (SSP-RK (3,3)) method as the time integration method to update the solution.

To sum up, the main skeleton of the code we implemented is given in the pseudo-code below:

Algorithm 1 Structure of the code

```
while $time < finaltime$ do
    $maxvel \leftarrow max(2 * abs(u))$
    $k \leftarrow CFL * dx/maxvel$
    if $time + k \geq Finaltime$ then
        $k = finaltime - time$
    end if
    extend data and boundary conditions
    $L(u^n) \leftarrow S_0^n, S_1^n, S_2^n$
    $u^{(1)} \leftarrow u^n + kL(u^n)$                                      ▷ 1st stage SSP-RK3 (3,3)
    $L(u^{(1)}) \leftarrow S_0^{(1)}, S_1^{(1)}, S_2^{(1)}$
    $u^{(2)} \leftarrow \frac{3}{4}u^n + \frac{1}{4}u^{(1)} + \frac{k}{4}L(u^{(1)})$          ▷ 2nd stage SSP-RK3 (3,3)
    $L(u^{(2)}) \leftarrow S_0^{(2)}, S_1^{(2)}, S_2^{(2)}$
    $u^n \leftarrow \frac{1}{3}u^n + \frac{2}{3}u^{(2)} + \frac{2k}{3}L(u^{(2)})$          ▷ 3rd stage SSP-RK3 (3,3)
    $time \leftarrow time + k$
end while
```

3 Performance

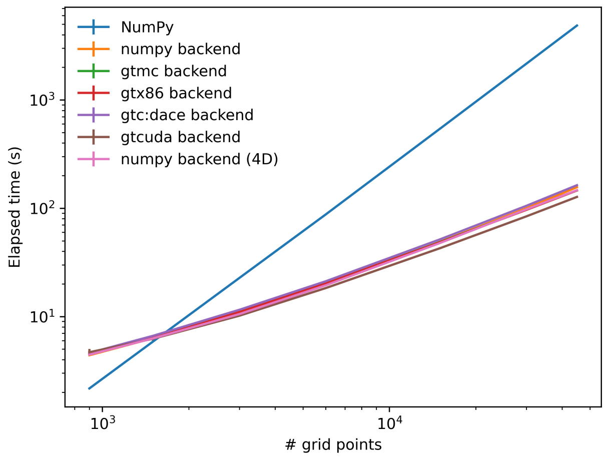

Figure 3: Performance of WENO5 for Burgers'equation using different GT4Py backends.

The performance was evaluated on Piz Daint (Intel® Xeon® E5-2670 CPU and Nvidia® Tesla® P100 GPU). We compared the performance among the original NumPy version, GT4Py with different backends (including *numpy*, *x86*, *cuda*, and *gtc:dace*) and GT4Py *numpy* without using temporary variables. From Figure 3, the x-axis is the number of grids of the computational domain, and the y-axis is the corresponding computation time of either different program versions or GT4Py backends.

The computational time of the original NumPy version shows a linear increase concerning the number of grids. Yet with the GT4Py stencil implementation, the computational time increased only when the number of grids reaches a certain degree, and the speed of increase is significantly lower than the NumPy version, meaning that GT4Py is capable to handle big data and large number/dimensional computations. It is worth mentioning that the application of GPU (*gtcuda* backend) has large potential in reducing the computational time, especially when the computational domain (number of grids) is large. In that sense, since GT4Py has the advantage of easily switching between GPU and CPU with self-managed memory transfer that the programmer doesn't need to pay special attention to, coding with GT4Py might enable further exploiting of the capability of GPU in the field of weather and climate modelling.

4 Discussion and conclusions

In this project, we implemented the high order WENO5 approximation for spatial approximation and SPP-RK (3,3) as time integration using GT4Py to solve the Burger's equation. From the velocity results, it successfully simulates the 2D shock wave.

During implementations, we dealt with the following technical issues. Since GT4Py is still under development, further work can be done to make it support more features.

- Horizontal loops are deduced automatically with increasing order by dependency and offset analysis. As shown in Figure 2, the index for spatial dimension is ascending while the position of the stencil is negatively offset. To deal with this, the cell interfaces are applied explicitly with temporary variables.
- The shape of the matrix should be consistent for stencil implementations. The smoothness indicator can be calculated as: $\beta = \overline{u}^T \widetilde{Q} \overline{u}$, where $\widetilde{Q}$ is a $m \times m$ matrix operator calculated using a Gaussian quadrature with a different shape other than $\overline{u}$. To deal with this, we write $\beta_{s=0,1,2}$ explicitly and replaced the index that is not associated to the grid points with numbers, since it's constant over time. However, it may be problematic when it changes in time.

Finally, we compared the performance of numpy and GT4Py with different backends. The results have shown that the elapsed time of using GT4Py is significantly lower than original numpy implementation. And the computational time increased only when the number of grid points reach a certain degree. When indicates the great potential of GT4Py in stencil computations.

References

Harten, A., Engquist, B., Osher, S., and Chakravarthy, S. R. (1987). Uniformly high order accurate essentially non-oscillatory schemes, III. *Journal of Computational Physics*, 71(2):231–303.

Hesthaven, J. S. (2018). *Numerical Methods for Conservation Laws.* Society for Industrial and Applied Mathematics.

Liu, X.-D., Osher, S., and Chan, T. (1994). Weighted essentially non-oscillatory schemes. *Journal of Computational Physics*, 115(1):200–212.

Shu, C.-W. (1998). Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws. In *Lecture Notes in Mathematics*. Springer Berlin Heidelberg.