APMA 2822B: Simulations of Compressible Fluid Dynamics on Graphics Processing Units

Justin Dong
Anna Lischke

May 2019

1 Introduction

The simulation of compressible fluid dynamics concerns fluids with large pressure gradients. This typically occurs in high-speed flows, for instance the study of high-speed aircraft and jet engines, atmospheric reentry of spacecraft, and flow-induced noise. The dynamics of such flows often include sharp gradients and discontinuities in fluid parameters (e.g. density and pressure). Thus, high-resolution numerical schemes are essential in capturing the correct fine-scale behavior of the fluid. However, even in two spatial dimensions, the computational power required to resolve this fine-scale behavior is considerable. As such, parallelization plays a central role in ensuring that simulations of compressible fluid flows may be carried out in a finite amount of time.

For concreteness, we consider the compressible Euler equations in two dimensions and apply a modal discontinuous Galerkin (DG) finite element method in order to compute a numerical solution. We implement our scheme in both OpenMP as well as CUDA. In particular, we consider a domain decomposition using two CUDA devices and explore CUDA's peer-to-peer transfer features in order to communicate between devices. In the literature, nodal discontinuous Galerkin methods have been implemented on NVIDIA GPUs in [4] and [3], with the latter dealing with systems of nonlinear conservation laws. In these papers, the authors achieved approximately 60-80x speedup over serial implementations. However, they work with nodal variants in which the degrees of freedom correspond directly to solution values at the chosen nodes. To the best of our knowledge, we have not found any examples of explicit benchmarking of modal discontinuous Galerkin schemes on GPUs for systems of conservation laws.

An outline of the report is as follows. First, we introduce the modal discontinuous Galerkin method for a two-dimensional conservation law, including time integrators and slope limiters. Next, we briefly describe the Rayleigh-Taylor instability, which is the example we have chosen to benchmark our code on. In sections 4 and 5, we discuss the implementation in OpenMP and CUDA. In particular, we cover the domain decomposition utilizing two GPU devices. Finally, we present our results and discussion.

2 Background

We are interested in numerically solving a two-dimensional conservation law given by

$$\begin{aligned} &u_t + F(u)_x + G(u)_y = 0 \quad \text{in } \Omega \\ &u(x, y, 0) = u_0(x, y) \end{aligned}$$

with suitable boundary conditions. Here, $F : \mathbb{R} \to \mathbb{R}$ and $G : \mathbb{R} \to \mathbb{R}$ are the so-called flux functions. The semi-discrete scheme scheme local to cell K is given by [1, 2]

$$\int_K \frac{du_h}{dt} v_h \, dx + \int_K \nabla \cdot (F(u), G(u)) \, dx = 0 \quad \forall v_h \in V_h.$$

Here, V_h is the space of all functions which are piecewise polynomial of degree p on cell K. Integrating the second term by parts, we obtain

$$\int_K \frac{du_h}{dt} v_h \, dK - \int_K (F(u_h), G(u_h)) \cdot \nabla v_h \, dK + \int_{\partial K} v_h (F(u_h), G(u_h)) \cdot \mathbf{n} \, dS = 0$$

for all $v_h \in V_h$. In the case of a regular quadrilateral mesh, the surface integral simplifies further as the normal vectors correspond to $(-1, 0)$, $(0, -1)$, $(1, 0)$, and $(0, 1)$ on the left, bottom, right, and top boundaries of element K, respectively.

We use an explicit three-stage Runge-Kutta scheme to integrate in time, and a uniform quadrilateral mesh is used to discretize a two-dimensional domain. After each time step, a moment limiter is applied to maintain stability of the numerical scheme [5].

We apply this scheme to solve the Euler equations

$$\begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix}_t + \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u(E + p) \end{bmatrix}_x + \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ v(E + p) \end{bmatrix}_y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{in } \Omega \tag{1}$$

with suitable initial and boundary conditions. The function ρ represents density, p is pressure, u is velocity in the x-direction, v is velocity in the y-direction, and E is the internal energy. The subscripts t, x, and y indicate a partial derivative is taken in terms of that variable.

This approach is highly parallelizable due to the explicit time-stepping scheme. The space domain may be decomposed, and information need only be shared when fluxes and limiters are computed at the boundaries of each subdomain. We emphasize that one of the most attractive features of the DG scheme is its compact stencil – independent of the degree of polynomial basis functions, each cell only requires information from its immediate neighboring cells to update its solution. Within each subdomain, CUDA may be used to leverage GPU resources for assembling the solution on each individual grid cell, while the domain decomposition is accomplished using cudaMemcpy from the initial condition array on the host into separate device arrays corresponding to each subdomain. Each subdomain is assigned to its own GPU, with one additional overlapping row of its neighbor subdomain.

We find that utilizing CUDA kernels on a GPU with a single subdomain enhances performance a great deal over that of the CPU, with a speedup of 10x over the 24 CPU-core implementation. With two subdomains, that speedup increases to 20x over the CPU single domain implementation.

In the following sections, we will describe the Rayleigh-Taylor instability problem, outline the implementations of the CPU, GPU, and multi-GPU approaches, present performance metrics and profiles of the codes, and plot numerical results.

3 Rayleigh-Taylor instability

When a fluid is suspended above a second fluid with a higher density, e.g., water in oil, and acted upon by gravity, the resulting instability between the fluid interface is known as the Rayleigh-Taylor instability, which may be modeled with the compressible Euler equations (1). Due to velocity shearing, instabilities develop along the interface between the two fluids.

We simulate the Rayleigh-Taylor instability using a third-order discontinuous Galerkin scheme with three-stage explicit Runge-Kutta time stepping. The choice of an explicit time-stepping scheme avoids the necessity of inverting a matrix during each time step. Locally, we need only compute a mass matrix. However, by choosing our polynomial basis to be the tensor-product Legendre polynomials on quadrilaterals, the mass matrix reduces to a scaled identity matrix so that our scheme is truly matrix-free. The domain $\Omega = [0, 0.25] \times [0, 1]$, and we use the initial condition

$$\begin{cases} \rho = 1, & y \geq 0.5 \\ \rho = 2.0 & y < 0.5, \end{cases} \qquad \begin{cases} p = y + 1.5 & y \geq 0.5 \\ p = 2y + 1 & y < 0.5, \end{cases} \qquad u = 0, \quad v = -0.025\sqrt{\gamma p/\rho}\cos(8\pi x),$$

where we choose the gas constant to be $\gamma = 1.4$. We use a quadrilateral mesh to discretize Ω, and we solve up to time $T = 1.95$.

On the bottom and top boundaries, we apply Dirichlet boundary conditions, and on the left and right boundaries, we apply Neumann boundary conditions.

4 Serial CPU implementation with openMP

We first implement the discontinuous Galerkin Runge-Kutta scheme on the CPU with openMP directives to provide benchmarks for comparing the GPU implementations. We use the `omp parallel for` directive for each Runge-Kutta stage update for all four variables, as in Listing 1.

Listing 1: OpenMP directive example: first Runge-Kutta stage

```
#pragma omp parallel for
for (int i = 0; i < nElems*nLoc; ++i)
{
    Q1[i] = 3.0/4.0*Q10[i] + (Q1[i] + dt*rhsQ1[i])/4.0;
    Q2[i] = 3.0/4.0*Q20[i] + (Q2[i] + dt*rhsQ2[i])/4.0;
    Q3[i] = 3.0/4.0*Q30[i] + (Q3[i] + dt*rhsQ3[i])/4.0;
```

```
    Q4[i] = 3.0/4.0*Q40[i] + (Q4[i] + dt*rhsQ4[i])/4.0;
}
```

Similarly, each OpenMP thread computes the updated solution per time step on a single element.

Listing 2: OpenMP directive example: compute RHS

```
#pragma omp parallel for
for (int k = 0; k < nElems; ++k)
{
    // compute rhsQ[k*nLoc+j] for j=0 to nLoc, where nLoc is the
    // degrees of freedom per element
}
```

5 GPU implementation with CUDA

For the first GPU implementation of the solver, we implement the CUDA kernel `computeRHS` to compute the right-hand-side contributions in parallel on each element. We also implement four `updateCUDA<k>` kernels, $k = 1, 2, 3, 4$, for the parallel computation of each stage of the RK scheme, and the fourth kernel updates the initial condition at the end of the loop. We also implement a the kernel `momentLimiter` to apply the limiter during each time step. The structure of the code is provided below.

1. Allocate solution arrays and parameters in both host and device memory.
2. Generate quad-mesh (CPU).
3. Project initial condition onto DG basis (CPU).
4. Enter time stepper (while $t < T$).
   (a) Copy initial data into solution array $\vec{Q}$ on the GPU.
   (b) Run `computeRHS` kernel.
   (c) Run `CUDAupdate1` kernel for the first stage of the RK update.
   (d) Apply the `momentLimiter` kernel to each variable.
   (e) Repeat (b) - (d) for second and third RK stages, replacing `CUDAupdate1` with `CUDAupdate2` and `CUDAupdate3`, respectively.
   (f) Every N time steps, print the solution to a `.vtk` file.
   (g) Time step $t+ = \Delta t$.
5. Free memory.

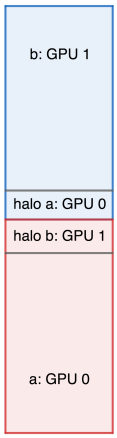

Figure 1: Decomposition of domain into two subdomains.

6 Multi-GPU implementation with domain decomposition

We decompose Ω into two subdomains as in Figure 1, with the goal of computing the solution on each subdomain on its own GPU with minimal memory transfers to the CPU. To do this, we begin by allocating memory for the full solution array only in the host memory. Once the mesh has been generated and the initial condition computed by the CPU as before, we allocate memory in the device for the two half-sized solution arrays `Qa` and `Qb` on device 0 and device 1, respectively. In the `computeRHS` and `momentLimiter` kernels, the top row of the bottom domain requires information from the bottom row of the top domain, and vice versa. To address this, we also allocate memory for the halo arrays, each the size of one row of elements `Nx`, `QaHalo` and `QbHalo`. The domain decomposition proceeds as in Algorithm 1.

Algorithm 1: Initialization and domain decomposition

```
cudaSetDevice(0); cudaDeviceEnablePeerAccess(1,0);
cudaSetDevice(1); cudaDeviceEnablePeerAccess(0,0);
// Project initial condition into host array
DGsolver::L2projection(Q0_h);
// Copy host initial condition into device subdomain arrays
cudaSetDevice(0);
cudaMemcpy(Qa_d, Q0_h,...,cudaMemcpyHostToDevice);
cudaMemcpy(QaHalo_d, Q0_h + nElems/2,...,cudaMemcpyHostToDevice);

cudaSetDevice(1);
cudaMemcpy(Qb_d, Q0_h + nElems/2,...,cudaMemcpyHostToDevice);
cudaMemcpy(QbHalo_d, Q0_h + nElems/2 - Nx,...,cudaMemcpyHostToDevice);
```

Throughout the time-stepping loop, we update the solution using the `computeRHS`, `CUDAupdate`,

and `momentLimiter` kernels as before, except on the subdomain arrays. The kernels are launched in parallel using one openMP thread for each domain's kernel. As transfer of data into the halo arrays is needed after the `CUDAupdate` and `momentLimiter` kernels complete, we use peer-to-peer memory exchange in order to bypass CPU memory altogether. This algorithm is listed in Algorithm 2.

Algorithm 2: Time stepper in multi-GPU implementation.

```
while t < T do
    for N from 1 to 3 do
        for j from a to b do
            computeRHSkernel<<< ... >>>(Qj_d, QjHalo_d,...);
            CUDAupdateN<<< ... >>>(Qj_d, QjHalo_d,...);

        cudaSetDevice(0); cudaDeviceSynchronize();
        cudaMemcpyPeer(QaHalo_d, 0, Qb_d, 1, Nx*size);
        cudaSetDevice(1); cudaDeviceSynchronize();
        cudaMemcpyPeer(QbHalo_d, 1, Qa_d + nElems/2 + Nx, 1, Nx*size);

        for j from a to b do
            momentLimiter<<< ... >>>(Qj_d, QjHalo_d, ...);

        cudaSetDevice(0); cudaDeviceSynchronize();
        cudaMemcpyPeer(QaHalo_d, 0, Qb_d, 1, Nx*size);
        cudaSetDevice(1); cudaDeviceSynchronize();
        cudaMemcpyPeer(QbHalo_d, 1, Qa_d + nElems/2 + Nx, 1, Nx*size);

    for j from a to b do
        CUDAupdate4<<< ... >>>(Qj_d, QjHalo_d,...);

    // Synchronize and print out results every S time steps (transfer from device to
     host)
    t+ = Δt;
```

In Figure 2, we show the profile results of the multi-GPU implementation, which demonstrates the concurrency of the kernels run on each GPU. In the MemCpy rows, we see that the host to device data transfer occurs only once at the beginning, followed by regular prints of the solution to `.vtk` files from each device to the host. The bulk of the memory transfers occur via peer-to-peer MemCpy, and the kernels are run concurrently. The most computationally expensive kernel is the computeRHS kernel in the first row for each device, followed by the momentLimiter. A possible next step to achieve better performance would be to further refine these kernels to make them more efficient.

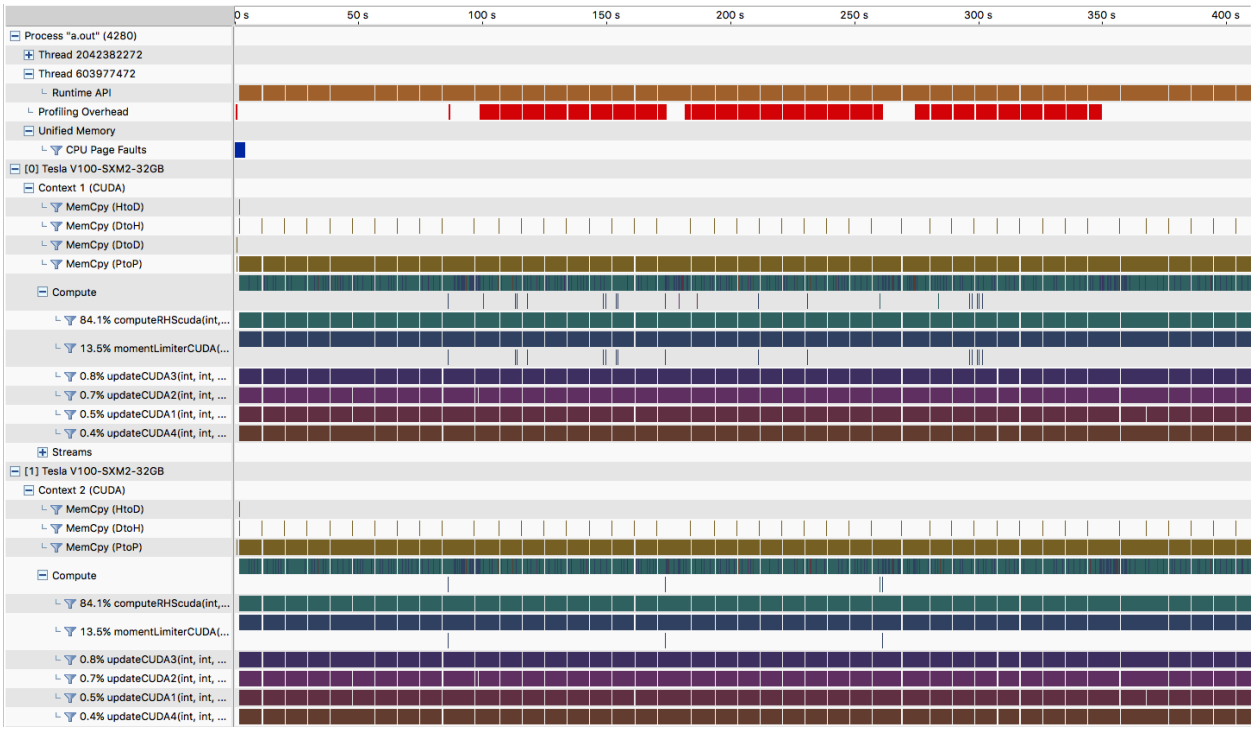

Figure 2: Profile of multi-gpu code, with mesh dimensions 256×1024, $T = 1.95$, 13000 iterations, run time of 408 s.

7 Performance Metrics

Table 1 shows the wall times for the OpenMP and CUDA variants. We see that CUDA provides an approximately 10x speedup over the OpenMP variant while the domain decomposition implementation using two devices provides a linear 2x speedup. This relation is shown in Figure 3. We note that while all runs were carried out on Tesla V100's, CCV's scheduling allows for other users to run jobs on devices which are currently in use by other jobs, our timing for the regular CUDA simulation on the finest mesh was adversely impacted. We did not have time to rerun this simulation.

Furthermore, we perform a strong scaling test by fixing the problem size at 256×1024 elements, each with 9 degrees of freedom for a total of 2 million degrees of freedom. We compute the speedup attained by increasing the number of CPU cores. Recall that Amdahl's

DOFs	OpenMP	CUDA	CUDA-DD
2.3×10^6	126.2m	12.6m	6.5m
9.4×10^6	1038.5m	101.9m	47.7m
3.7×10^7	–	1032.7m	367.6m

Table 1: Wall time in minutes for three different problem sizes using OpenMP with 24 CPU cores, CUDA with a single Tesla V100 device, and CUDA with domain decomposition (two Tesla V100's).

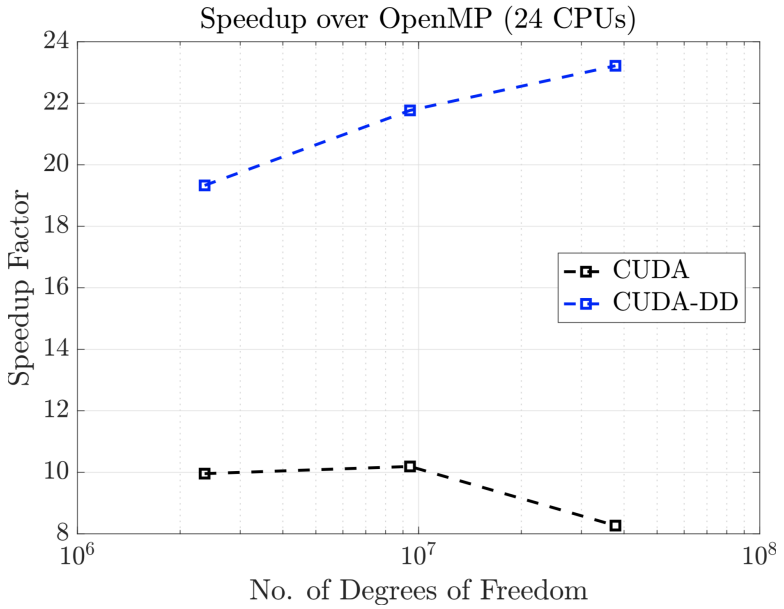

Figure 3: Speedup of the CUDA implementations over OpenMP.

Law provides a theoretical speedup attained by a code that is $p\%$ parallelizable. Figure 4 shows our attained speedup along with the theoretical predictions for a variety of parallelizable codes. We see that in the asymptotic limit, our code appears to closely resemble the behavior of a code that is 90% parallelizable. As the DG method is known to be highly parallelizable, this result suggests that perhaps we have not parallelized our code in the most efficient way.

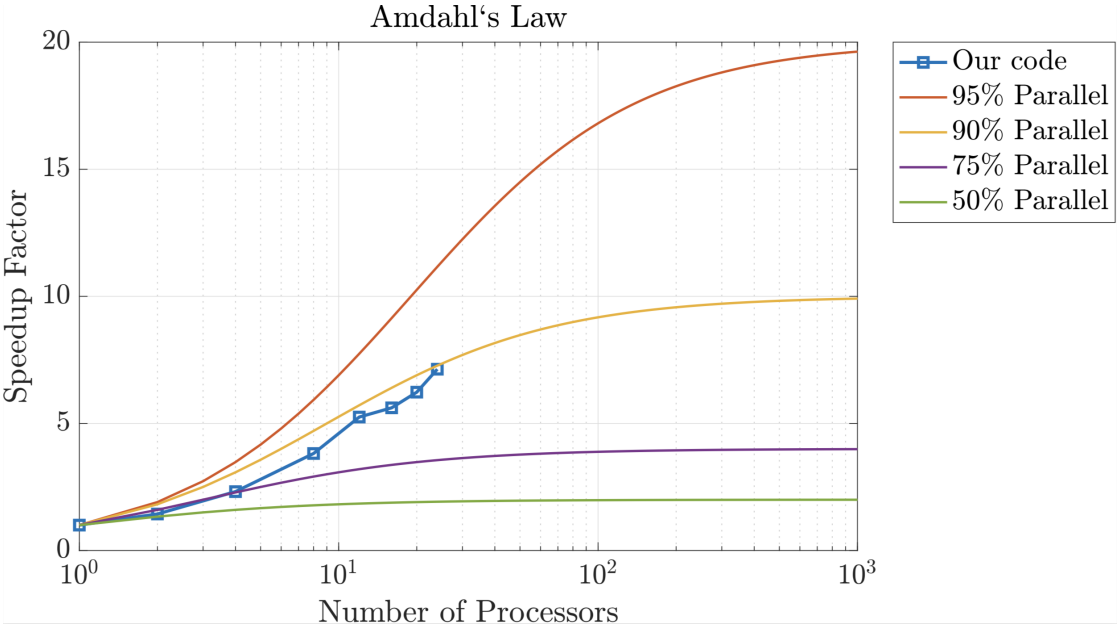

Figure 4: Amdahl's Law.

8 Visualization

The visualization of our results was carried out in Paraview. Figure 5 shows the density distribution at time $T = 2.6$ on a square domain consisting of 1024×1024 elements. We see that our numerical scheme can properly capture the sharp gradients along the fluid interface. While not shown here, our tests demonstrate that the use of quadratic basis functions provides a clear advantage over linear basis functions and constant basis functions (essentially a reduction to the first-order finite volume scheme). In particular, quadratic basis functions on even coarse meshes (e.g. 512×512) are able to capture the instabilities at the interface, while linear and constant basis functions on very fine meshes (e.g. 2048×2048) have difficulty in this task.

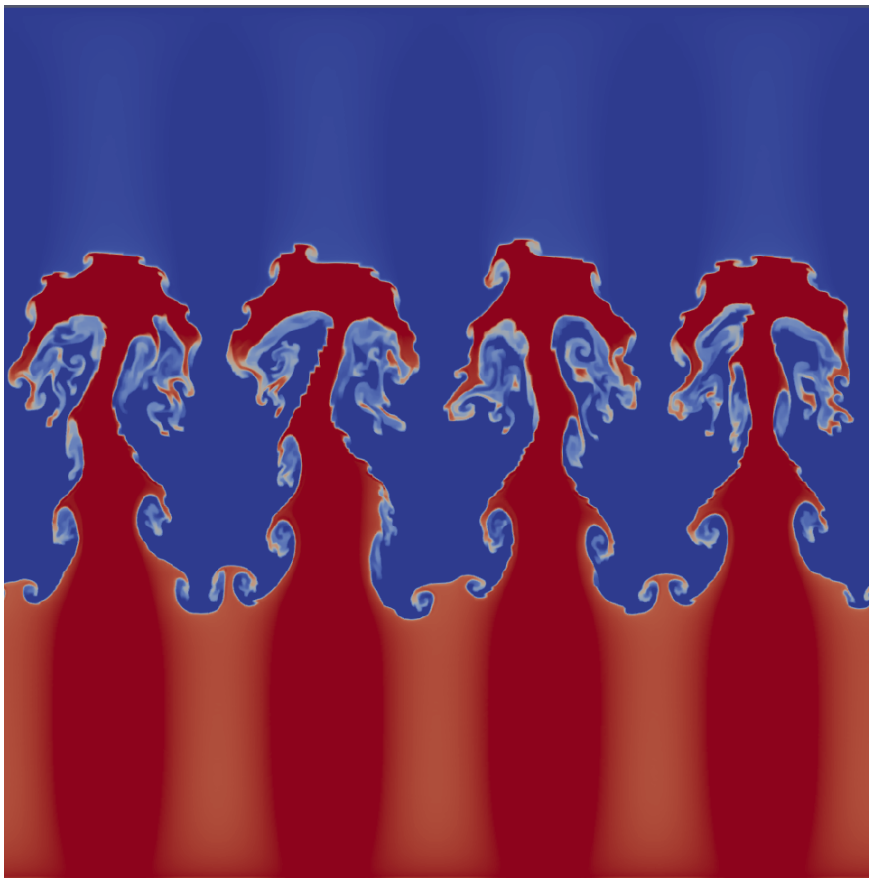

Figure 5: Rayleigh-Taylor instability obtained from our code on the domain $(0, 1)^2$ at $T = 2.6$.

9 Conclusions & Future Work

Our results demonstrate that the DG method is highly parallelizable and scales well using two NVIDIA GPUs. We would like to extend our domain decomposition code to utilize four GPUs, the standard configuration for a single GPU node on CCV. Furthermore, we would

like to integrate our code with MPI, utilizing CUDA's peer-to-peer memory exchanging within nodes and MPI's features to communicate between nodes.[5]

References

[1] B. Cockburn, S. Hou, and C.-W. Shu, *The runge-kutta local projection discontinuous galerkin finite element method for conservation laws. iv. the multidimensional case*, Mathematics of Computation, 54 (1990), pp. 545–581.

[2] B. Cockburn and C.-W. Shu, *The runge–kutta discontinuous galerkin method for conservation laws v: multidimensional systems*, Journal of Computational Physics, 141 (1998), pp. 199–224.

[3] M. Fuhry, A. Giuliani, and L. Krivodonova, *Discontinuous galerkin methods on graphics processing units for nonlinear hyperbolic conservation laws*, International Journal for Numerical Methods in Fluids, 76 (2014), pp. 982–1003.

[4] A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven, *Nodal discontinuous galerkin methods on graphics processors*, Journal of Computational Physics, 228 (2009), pp. 7863–7882.

[5] L. Krivodonova, *Limiters for high-order discontinuous galerkin methods*, Journal of Computational Physics, 226 (2007), pp. 879–896.