GPU-Accelerated Discontinuous Galerkin Methods for Vortex Transport Simulations

Guodong Chen cgderic@umich.edu

Abstract

In this project, we present the development of a GPU-accelerated discontinuous Galerkin (DG) finite element method (FEM) for two-dimensional vortex transport simulations. Compressible Euler equations are solved and integrated in time to accurately preserve the vortex during the transport. The DG code is implemented in C using NVIDIA's Compute Unified Device Architecture (CUDA). Numerical experiments on a set of computational meshes with different DG approximation orders are performed to demonstrate the performance of our implementation. Furthermore, several optimization techniques are investigated to further improve the code performance. The effectiveness of the GPU implementation encourage more development and optimization on the current implementation.

1 Introduction

Accurate transport of vortices is important in the analysis of many aeronautical systems, including fixed-wing aircraft and rotorcraft. High resolution in space and time, on unstructured meshes, is vital in preserving vorticity and minimizing the introduction of spurious numerical errors that dissipate the vortex. High-order numerical methods are essential for achieving the preferred accuracy and efficiency. Discontinuous Galerkin (DG) finite element methods (FEM) have received much attention over the past decades[1, 2], largely due to its high-order nature and robustness for dealing with convection-dominated problems, which govern the flow phenomena in most aeronautical systems. The vortex transport problem studied in this paper is on the limit of convection-dominated problems, where viscous effects are ignored, *i.e.,* pure convection without dissipation[1]. In this project, we will simulate an isentropic vortex in uniform flow, on a square domain with periodic boundaries. Figure 1 shows the setup for the vortex problem we are considering in this project. The left figure shows a sample unstructured mesh consists of non-overlap triangle elements, while the right one presents a sample pressure field contour over the computation domain. Different from the structured meshes with implicit connection information, unstructured meshes require explicit storage of the connections.

In DG spatial discretization, the solution is approximated by polynomials inside each element over the computational mesh, without continuity restrictions on the element interfaces.

[1]Although no nature dissipation is presented in the system, any numerical method may bring in the numerical dissipation to stabilize the system.

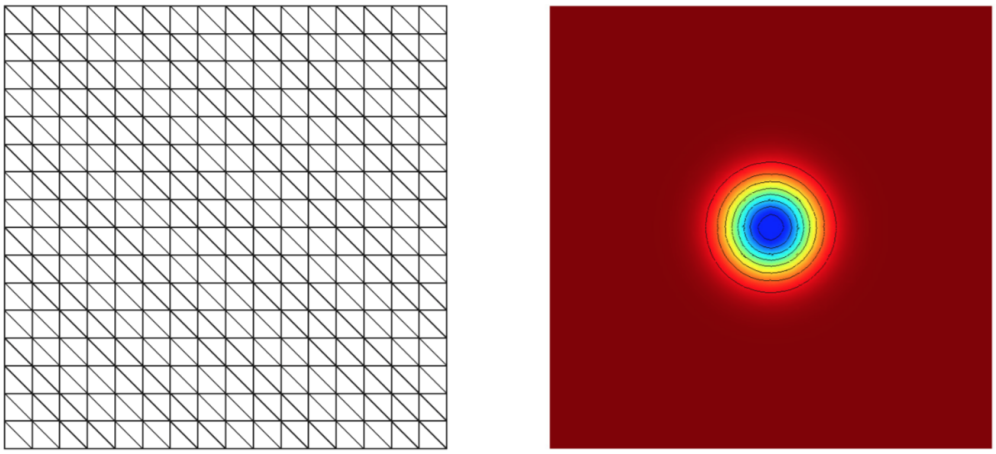


Figure 1: Sample computational mesh (left) and pressure field of a vortex (right).

In other words, the solution is only continuous inside an element, and can be discontinuous across the element interfaces. The coupling between adjacent elements only comes in with uniquely defined inter-element numerical fluxes, by choosing an appropriate Riemann solver[3]. Therefore, the computing and memory access pattern in DG is mostly local, hence is generally easy to parallelize. Moreover, the high-order solution representation in DG requires fewer data points to resolve the solution and hence less memory access compared to finite volume and finite difference methods, with the expense of higher arithmetic intensity per degree-of-freedom (data point). The relatively high computation intensity and low memory access make DG particularly suitable for parallelization on GPU. To investigate the feasibility of accelerating DG simulations using GPU and the potential performance gains, we implement the DG method using CUDA programming model and test it on the vortex transport problem mentioned above.

The remainder of this paper proceeds as follows. We describe the numerical approach including the governing equations and the DG discretization in Section 2. Detailed work partitioning and parallel algorithms are presented in Section 3. Numerical experiments are performed in Section 4 to investigate the correctness and measure the performance of our implementation. Section 5 Concludes the present work and discusses the potential future work.

2 Numerical Approach

2.1 Governing Equations

In this ideal problem, the vortex gets transported without any viscous effects. The governing partial differential equations (PDEs) for the system are compressible Euler equations, which is derived from the inviscid limit of the compressible Navier-Stokes equations. In two spatial dimensions, the Euler equations can be written as

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \vec{\mathbf{F}}(\mathbf{u}) = \mathbf{0}, \tag{1}$$

where $\mathbf{u} \in \mathbb{R}^S$ is the conservative states with S components, defined by flow physical quantities, *e.g.,* density and velocities. $\vec{\mathbf{F}} \in [\mathbb{R}^S]^2$ is a two dimensional spatial flux vector with S components in each dimension (direction). S is often referred to as the state rank, $S = 4$ in this problem.

$$\text{state: } \mathbf{u} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{bmatrix}, \quad \text{flux: } \vec{\mathbf{F}} = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uH \end{bmatrix} \hat{x} + \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vH \end{bmatrix} \hat{y}. \tag{2}$$

In order to close the Euler equations, we need to include the perfect gas law. The variables in the whole system are defined as follows:

ρ	=	density	*Calorically-perfect gas*		
u, v	=	x, y components of velocity	e	=	$c_v T$
E	=	total energy per unit mass	h	=	$c_p T$
e	=	internal energy per unit mass	p	=	ρRT
H	=	total entropy per unit mass		=	$(\gamma - 1)\left[\rho E - \frac{1}{2}\rho\lvert\vec{v}\rvert^2\right]$
h	=	internal enthalpy per unit mass	H	=	$E + p/\rho$
p	=	pressure	M	=	$\lvert\vec{v}\rvert/c = \sqrt{u^2 + v^2}/c$
R	=	gas constant for air, $c_p - c_v$	c	=	$\sqrt{\gamma RT} = \sqrt{\gamma p/\rho}$
γ	=	ratio of specific heats, $c_p/c_v = 1.4$			
c_p	=	specific heat at constant pressure			
c_v	=	specific heat at costant volume			
c	=	speed of sound			
M	=	Mach number			

2.2 Initial and Boundary conditions

We use the analytical vortex solution of the Euler equations to get the initial condition for the states. The state at point $\vec{x} = (x, y)$ at time t is given as

$$\begin{aligned} \rho &= \rho_\infty f_1^{1/(\gamma-1)}, \\ u &= U_\infty - f_2(y - y_0 - V_\infty t), \\ v &= V_\infty + f_2(x - x_0 - U_\infty t), \\ p &= p_\infty f_1^{\gamma/(\gamma-1)}. \end{aligned} \tag{3}$$

The function f_0, f_1 and f_2 are defined as

$$\begin{aligned} f_0 &= 1 - \frac{|\vec{x} - \vec{x}_0 - \vec{V}_\infty|^2}{r_c^2}, \\ f_1 &= 1 - \epsilon^2(\gamma - 1)M_\infty^2 \frac{e^{f_0}}{8\pi^2}, \\ f_2 &= \epsilon \frac{|\vec{V}_\infty|}{2\pi r_c} e^{f_0/2}. \end{aligned} \tag{4}$$

The subscripts ∞ denotes a background "free-stream" state without vortex perturbation. In this problem, we have the units: $\vec{x}_0 = (0,0)$, $\vec{V}_\infty = (U_\infty, V_\infty) = (1,1)/\sqrt{2}$, $\rho_\infty = 1$, $M_\infty = 0.5$, $\gamma = 1.4$, $\epsilon = 0.3$, $r_c = 1.0$.

We initialize the state vector by taking $t = 0$ in Eqn. 3, and periodic boundary condition is used for this problem. By the choices of the initial and boundary conditions, we would expect that the vortex centered at the origin at $t = 0$ (Figure 1) will get transported by the free-stream velocity $\vec{V}_\infty$ diagonally to the upper-right corner of the computational domain and then come back from the lower-left corner (periodic). The DG method is intended to simulate this transport phenomenon.

2.3 DG Spatial Discretization

2.3.1 Solution Approximation

A discontinuous Galerkin (DG) finite element method is implemented to solve the given problem. In DG, the solution is approximated as a weighted sum of basis functions over the entire domain,

$$\begin{aligned} \mathbf{u}(\vec{x},t) &= [\mathbf{u}^s(\vec{x},t)], \quad s = 1,...,S;\ S = 4. \\ \mathbf{u}^i(\vec{x},t) &\approx \sum_{k=1}^{N_e}\sum_{j=1}^{N_p} \mathbf{U}^s_{k,j}\phi^p_{k,j}(\vec{x}). \end{aligned} \tag{5}$$

where N_e is the number of the elements, $\phi^p_{k,j}(\vec{x})$ is the j^{th} polynomial basis function, of order p, on the element k; N_p is the number of basis functions per element, and $\mathbf{U}^s_{k,j}$ are the time varying coefficients for $\phi^p_{k,j}(\vec{x})$ associated with s^{th} state component. Therefore, we have $N_e \times N_p$ unknown coefficients per state component, *i.e.*, the total degrees of freedom (DOF) are $N_e \times N_p \times S$. By the discontinuous definition, $\phi^p_{k,j}$ has only local support at element k, *i.e.*, $\phi^p_{k,j} = 0$ in all other elements. For easy exposition, the super script p in $\phi^p_{i,j}$ is omitted for the rest of the paper.

2.3.2 Weak Form

By substituting Eqn. 5 into the Euler equations, we have the strong from of the original PDE, which is not very useful as we can not solve the coefficient vector $\mathbf{U}^s_{k,j}$ with it. If we further multiply the strong form with the test functions (the same as our solution basis functions) and integrate by parts over each element, we have the weak form of the Euler equations. Consider test function $\phi_{k,i}$ which only has support over element k, Ω_k, the weak form can be written as

$$\int_{\Omega_k} \phi_{k,i}\frac{\partial \mathbf{u}}{\partial t}d\Omega - \int_{\Omega_k} \nabla\phi_{k,i}\cdot\vec{\mathbf{F}}d\Omega + \int_{\partial\Omega_k} \phi^+_{k,i}\hat{\mathbf{F}}(\mathbf{u}^+,\mathbf{u}^-,\vec{n})dl = \mathbf{0}. \tag{6}$$

where $\vec{n}$ is the normal vector that points out of Ω_k on its edges; $\hat{\mathbf{F}}$ is the numerical inter-element flux function, and the superscripts $^+/^-$ denote a quantity taken from the interior/exterior of element k.

Substituting Equation (5) into Equation (6) gives a system of ODEs,

$$\mathbf{M}\frac{d\mathbf{U}}{dt} + \mathbf{R}(\mathbf{U}) = \mathbf{0} \tag{7}$$

where $\mathbf{M}$ is the mass matrix and $\mathbf{R}(\mathbf{U})$ is the residual vector that depends on the state $\mathbf{U} = [\mathbf{U}^s], s = 1, ..., S$. The state $\mathbf{U}$ and residual $\mathbf{R}$ are unrolled vectors/arrays of size $N_e \times N_p \times S$, where S is the number of the state components. The mass matrix $\mathbf{M}$ is a $N_e \times N_p \times S$ by $N_e \times N_p \times S$ block-diagonal matrix with $N_p \times S$ by $N_p \times S$ block $\mathbf{M}_k$ for each element, due to the local support of the elemental integral. Since every state component use the same set of basis functions, $\mathbf{M}_k$ is again block diagonal with repeated $N_p \times N_p$ block $\mathbf{M}_{k,scalar}$ for each state. $\mathbf{M}_{k,scalar}$ only differs by a factor J_k (often referred to as Jacobian determinant) for different elements, accounting for different element geometries, *i.e.*, $\mathbf{M}_{k,scalar} = J_k\mathbf{M}_{scalar}$. As the Mass matrix does not change during the simulation, we can pre-compute $\mathbf{M}_{scalar}$ and J_k, and store them for the later simulation. This storage is much more efficient than storing the whole mass matrix $\mathbf{M}$, since only one small matrix $\mathbf{M}_{scalar} \in \mathbb{R}^{N_p \times N_p}$ and a vector of scaling factors $\mathbf{J} \in \mathbb{R}^{N_e}$ are stored.

2.4 Time Integration

In this project, we use a fourth-order Runge-Kutta (RK4) method to integrate Eqn. 7 in time. We first rewrite Eqn. 7 as

$$\frac{d\mathbf{U}}{dt} = -\mathbf{M}^{-1}\mathbf{R}(\mathbf{U}) = \mathbf{f}(\mathbf{U}), \tag{8}$$

then the RK4 time integration scheme adopted in this work can be written as

$$\begin{aligned}
\mathbf{f}_0 &= \mathbf{f}(\mathbf{U}^n, t^n), \\
\mathbf{f}_1 &= \mathbf{f}(\mathbf{U}^n + \frac{1}{2}\Delta t\mathbf{f}_0, t^n + \frac{\Delta t}{2}), \\
\mathbf{f}_2 &= \mathbf{f}(\mathbf{U}^n + \frac{1}{2}\Delta t\mathbf{f}_1, t^n + \frac{\Delta t}{2}), \\
\mathbf{f}_3 &= \mathbf{f}(\mathbf{U}^n + \Delta t\mathbf{f}_2, t^n + \Delta t), \\
\mathbf{U}^{n+1} &= \mathbf{U}^n + \frac{\Delta t}{6}(\mathbf{f}_0 + 2\mathbf{f}_1 + 2\mathbf{f}_2 + \mathbf{f}_3).
\end{aligned} \tag{9}$$

Sue to the diagonal pattern of $\mathbf{M}$, the inversion $\mathbf{M}^{-1}$ in Eqn. 8 can be done locally, and similarly the inverse matrix can also be stored as a small matrix and a set of scaling factors.

3 Implementation

The main computational effort in this simulation is the evaluation of the residual vector $\mathbf{R}$ in Eqn. 7, and the RK4 time integration which is relatively easy to implement. We now discuss the steps to parallelize the evaluation of the residual vector, $\mathbf{R}$. The residual vector $\mathbf{R}(\mathbf{U})$ is the last two terms in Eqn. 6,

$$\mathbf{R}(\mathbf{U})_{k,i} = -\int_{\Omega_k} \nabla\phi_{k,i} \cdot \vec{\mathbf{F}}(\mathbf{u})d\Omega + \int_{\partial\Omega_k} \phi^+_{k,i}\hat{\mathbf{F}}(\mathbf{u}^+, \mathbf{u}^-, \vec{n})dl. \tag{10}$$

This computation involves a volume integral and face integral, which are the two main kernels in the CUDA implementation. We call them `calculateVolumeRes` kernel and `calculateFaceRes` kernel in the rest of the report. By convention, face and edge are used interchangeably in the work to denote the element boundary.

3.1 Volume Integration Kernel

For the volume integral term

$$\mathbf{R}(\mathbf{U})^v_{k,i} = -\int_{\Omega_k} \nabla\phi_{k,i} \cdot \vec{\mathbf{F}}(\mathbf{u}) d\Omega, \tag{11}$$

the parallisim is fairly straightforward. We use one thread per element to evaluate the element local volume integral. As we can see in Eqn. 11, $\vec{\mathbf{F}}(\mathbf{u})$ only depends on the local state, $\mathbf{U}_k$. Thus, each thread only needs to read in the local state vector and the local element geometry information (Jacobian matrix and its determinant) to perform the integration. For both volume and face integral evaluations, we use numerical integration with standard Gauss-Legendre points. More details about numerical integration is given in Section 3.4.1.

3.2 Face Integration Kernel

The face integral term for each element can be written as

$$\mathbf{R}(\mathbf{U})^f_{k,i} = \int_{\partial\Omega_k} \phi^+_{k,i} \hat{\mathbf{F}}(\mathbf{u}^+, \mathbf{u}^-, \vec{n}) dl = \sum_{e=1}^{3} \int_{e\in\partial\Omega_k} \phi^+_{k,i} \hat{\mathbf{F}}(\mathbf{u}^+, \mathbf{u}^-, \vec{n}) dl, \tag{12}$$

where e indexes the edges of the element. The numerical flux term $\hat{\mathbf{F}}(\mathbf{u}^+, \mathbf{u}^-, \vec{n})$ at each edge depends on the state vector on both side, brings in the coupling between adjacent elements.

The face integral turns out requiring more considerations for parallelization. First, we have two choices to partition the work: element-based or edge-based fashion. Suppose we have N_e elements ($3N_e/2$ faces in average): in the former approach, each element requires 3 memory access to the state vectors from neighboring elements and 1 access to its own state vector, $4N_e$ memory accesses in total; while in the latter one, each edge requires 2 memory access to the states on both sides, thus $3N_e/2 \times 2 = 3N_e$ total memory accesses. On the other hand, the element-based approach requires 3 integral evaluations per element, $3N_e$ in total; while the edge-based approach only needs 1 integration which can be used by both sides, thus only $3N_e/2$ edge integral evaluations in total. Given the fact that we are using unstructured meshes, the data storage of adjacent elements are not contiguous in general, the memory access can be expensive, thus the latter approach is preferred for less memory accesses and less computations as well.

Given the comparison above, the edge-based approach is used in this work. Each interior face is assigned a thread to do the face integral, then the face residual contribution is added to the elements on both sides. However, the edge-based approach encounters a race condition: different threads may want to add the residual contribution to the same element at the same time. The race condition prevents us from simply adding the resulting face integral together

with the volume integrals for left element Ω_l and the right element Ω_r as we compute them. As a work-around, we store the residual contributions to left element and right element separately as $\mathbf{R}^{f,l}$ and $\mathbf{R}^{f,r}$, and add them later to the residual vector when we do the time integration. $\mathbf{R}^{f,l}$ and $\mathbf{R}^{f,r}$ has the same dimension, and are both indexed by global face indices, *i.e.*, we have $\mathbf{R}^{f,l} = [\mathbf{R}_i^{f,l}]$ and $\mathbf{R}^{f,r} = [\mathbf{R}_i^{f,r}]$, $i = 0, 1, ..., N_{Face} - 1$.

3.3 RK4 Kernel

Before the RK4 time integration, we need to add the volume integral and face integral contributions to the residual vector. We again assign one thread per element. Every thread loops over the edges of current element, determines if the element is considered as the left or the right element of that edge, then adds the residual contribution from $\mathbf{R}^{f,l}$ or $\mathbf{R}^{f,r}$ accordingly. This kernel is referred to as `addRes`.

After obtaining the total residual vector, we can get the right hand side (RHS) $\mathbf{f}$ of Eqn. 8. We dedicate another kernel doing this mass matrix inversion, $\mathbf{f} = \mathbf{M}^{-1}\mathbf{R}$. As mentioned in Section 2.3.2, the inverse of the global matrix $\mathbf{M}^{-1}$ can be pre-computed. Only one matrix $\mathbf{M}^{-1}_{scalar}$ and one scaling factor $1/J_k$ for each element is needed to store the whole inverse matrix. It's a natural idea to partition the work element-wise, and each thread is responsible for doing a local mass matrix inversion (actually only matrix multiplication given the inverse matrix pre-computed and stored). This kernel is called `Res2RHS`.

With the RHS obtained by `Res2RHS`, we can update the state vector using RK4 time integration. Since each RK4 stage requires a global synchronization, thus each of them launches an individual kernel, although the intermediate stages share the same kernel function `inter_rk4` due to the similarity while the final stage has its own kernel function `final_rk4`.

3.4 Memory Management/Optimization

It's generally easy to parallelize algorithms like DG on GPU, in which most operations are local. However, good performance is not always easy to achieve, unless the memory is appropriately managed and optimized. Several memory optimizations are done during the development to improve the performance.

3.4.1 Pre-Computed Data and Constant Memory

There are a lot of data reuse in the DG code, *e.g.*, the mesh information, mass matrices and the numerical integration data. We take the numerical integration data as an example of pre-computed data usage.

Both the volume and face integrals are evaluated using numerical integration. Standard Gauss-Legendre points are used as quadrature rules in this work. The integral of a function f, can be approximated with a weighted sum of the function evaluated at the quadrature points. Thus the volume integral in Eqn. 11 inside every element requires the basis function gradients evaluated at 2D quadrature points, while the the face integral in Eqn. 12 requires the basis functions evaluated at 1D quadrature points. Although the shapes of the elements are different, these evaluations can be done in a reference space and then mapped to each

element during the integral. The gradients of the N_p basis functions at N_{q2} 2D quadrature points are stored in a matrix $\nabla\mathbf{\Phi} \in \mathbb{R}^{N_{q2}\times N_p}$, with corresponding integration weights $\mathbf{w}_{q2}$. Similarly, for face integral we have N_p basis functions evaluated at N_{q1} quadrature points, $\mathbf{\Phi} \in \mathbb{R}^{N_{q1}\times N_p}$, with weights $\mathbf{w}_{q1}$.

Numerical integration data is an example of the data that can be pre-computed and repeatedly used for each thread. Data like mesh connection information, global mass matrix can also be pre-computed and stored for later use in the simulation. Some of these data like the quadrature matrix will be accessed by many threads at the same time, at the same memory address. This is the best usage for constant memory, which has dedicated on-chip caches and is optimized for this kind of uniform memory access pattern. On the other hand, data like mesh connections are not suitable for the constant memory as different thread may read the connection data for different elements, thus at different memory addresses. This kind of memory access in constant memory is as slow as global memory access.

3.4.2 Thread Local Data

During both volume and face integrations mentioned above, each thread needs to load the element local state vector from the global memory, together with the quadrature data from the constant memory, in order to evaluate the physical states $\mathbf{u}$ or states gradients $\nabla\mathbf{u}$ inside the element or on the element edges. After getting the physical states, each thread also needs to calculate and store the physical flux $\mathbf{F}$ or the numerical flux $\hat{\mathbf{F}}$. The size of these data depend on the approximation order and the number of quadrature points. At the very beginning of my implementation, I call each thread to allocate memory in the GPU heap to store these data, *i.e.*, calling `malloc()` and `free()` on the device by each thread. However, the code performs poorly as all of these data then go to GPU global memory. This "heap" type global memory seems to have extremely high memory latency even compared to "stack" type global memory, although no explicit differences are found in the CUDA programming Guide [4].

A better approach to deal with these "dynamic" memory in my experience on GPU is to use "stack" static memory declaration, *i.e.*, telling the compiler to dedicate a fixed amount memory (maximum amount) to store these data in the compiling stage. By doing this, the device global memory bandwidth achieved is more than 50 times higher than the "heap" global memory. My guess is that by using "heap" global memory, each thread memory allocation may end up at random global memory address (as all concurrent threads allocate memory at the same time), and this cannot be optimized as it is run-time allocation. Another guess is that all the threads are concurrently allocating memory in the global memory, the device may have to somehow serialize the allocation to avoid race conditions in memory allocations, resulting a huge bandwidth loss. However by using "stack" type global memory, it can be optimized in the compilation, such that the thread memory access pattern can be optimized to achieve high memory bandwidth. More investigation is needed to determine the reason for the differences between these two approaches.

3.4.3 Max Registers per Thread and Occupancy

As mentioned in Section 3.4.2, the intermediate data storage in the `calculateVolumeRes` and `calculateFaceRes` may need a lot of memory, which will quickly fill out the registers on the chip. When the the available registers are used up for active running thread blocks, each streaming multiprocessor (SM) is limited to simultaneously run only few thread blocks, resulting low SM occupancy. This low occupancy can be increased by limiting the maximum registers per thread, by either defining the kernel lunch bounds in the source code or using the `--maxrregcount` flag in the compiling stage. The latter approach is examined in the current implementation. By setting `--maxrregcount` to a low value, higher SM occupancy is achieved. However, the performance is not increased since each thread block runs slower as less registers are assigned, although more concurrent blocks are running simultaneously.

3.5 Concurrent Kernel Launches

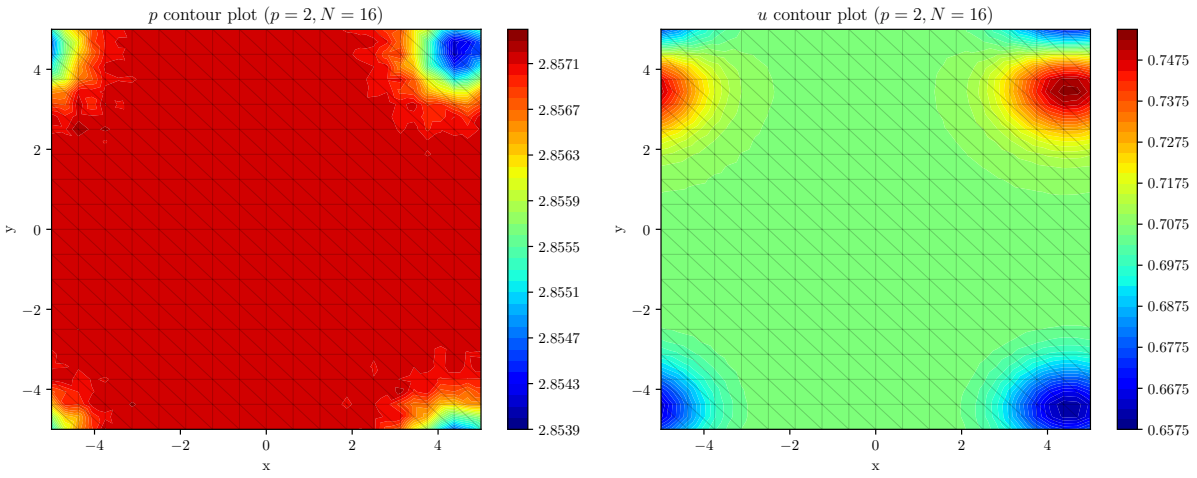
As we can see that the volume integration kernel and the face integration kernel are independent, we can issue the kernel launches into concurrent streams. On the other hand, all the other kernels have dependency and hence can only by launched serially. The performance gains by concurrent kernel launching can be considerable only when the resources are sufficient, *e.g.*, enough registers. Therefore, the benefits of concurrent launches are limited in our problem where the registers limit the performance very quickly.

4 Numerical Experiments and Performance Analysis

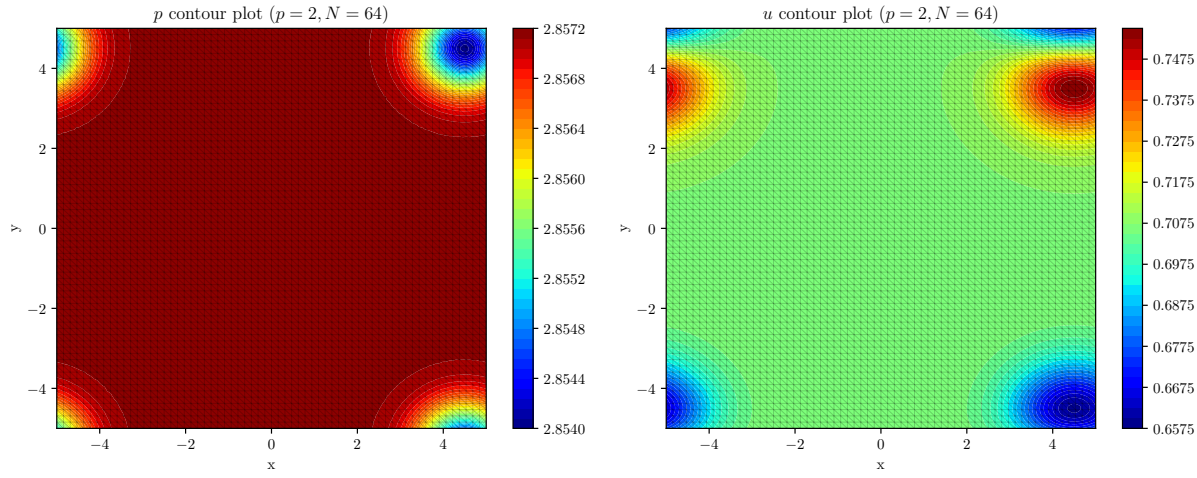
4.1 Code Verification

The CUDA code developed in this project is first verified by running a set of tests on different computational meshes with different orders. The results are compared to an equivalent CPU version of the DG code. The CUDA code is able to produce the same results as the CPU code, indicating correctness of the CUDA implementation. The GPU used in this work is the Nvidia GeForce GTX 1080, and the CPU used is the AMD Ryzen 7 1800X.

Only two set of results on different computational meshes are shown in Figure 2 for conciseness. The computational domain is first divided to equal squares and then each square is divided to two equal triangle elements. Thus the mesh size is characterized by the number of squares on each side, N. The total number of triangular mesh elements can be related to N as $N_e = 2N^2$. For later performance analysis, we use N instead of N_e as the mesh metric. In the actual tests, very large N is tested, however only small N is shown in Figure 2 due to the difficulties of visualizing the mesh. The computational domain is in $[-5, 5]^2$, and the simulation starts at $t = 0$, ends at $t = 10$, when the vortex starts leaving the upper-right corner and entering the lower-left corner. We can see that that our implementation is able to accurately simulate the vortex transport, and as the mesh gets finer the accuracy is higher.



(a) Pressure p and x-velocity u contours, DG approximation order $p = 2$, Mesh $N = 16$



(b) Pressure p and x-velocity u contours, DG approximation order $p = 2$, Mesh $N = 64$

Figure 2: Code verification on different meshes with approximation order $p = 2$.

4.2 Kernel Lunch Configuration Analysis

The kernel launch parameters are studied to determine the optimal kernel launch configuration. In this test, we studied several mesh sizes N and approximation orders p with only 10 RK4 iterations. The results are summarized in Table 1. We can see that for most of the tests, the performance stalls at 1 warp per SM, indicating very low occupancy. This is mainly due to the high memory requirement per thread, using up all available registers very quickly. The occupancy can be increased by limiting the registers per thread as mentioned in Section 3.4.3, though no performance improvements have been found due to slower thread blocks as described in Section 3.4.3.

	thread per block	32	64	128	256	512
time (ms)	$N = 160, p = 0$	41.978	52.031	50.972	51.697	52.069
	$N = 160, p = 1$	211.883	227.306	235.896	228.627	221.035
	$N = 160, p = 2$	1037.709	1123.094	1135.604	1130.568	1140.236
	$N = 320, p = 0$	109.953	159.936	162.701	165.591	168.216
	$N = 320, p = 1$	695.748	889.766	905.512	915.440	853.13
	$N = 320, p = 2$	3641.40	4071.97	4083.567	4084.301	3246.27

Table 1: Different kernel configuration on various mesh sizes and approximation orders. Here $p = 0$ means order 0 polynomial, which is constant inside an element.

4.3 Speedup and Scaling Analysis

The performance of the GPU accelerated DG code is first studied by comparing with the equivalent CPU version. The CPU code is a serial code run on AMD 1800x, and the GPU code is run on GeForce GTX 1080 with 32 threads per block. Both codes are run with only 10 RK4 iterations, and the timing results are reported in Table 2 and Figure 3.

		Mesh size N	32	64	128	256	512
time(ms)	$p = 0$	CPU	59.718	238.092	1011.105	4127.669	16266.888
		GPU	10.013	11.588	33.315	78.892	260.675
		speedup	5.964	20.56	30.349	52.321	62.403
		Mesh size N	32	64	128	256	512
time(ms)	$p = 1$	CPU	262.168	1069.318	4357.10	17851.866	71707.630
		GPU	26.742	30.622	125.91	464.836	1676.459
		speed up	9.804	34.920	34.605	38.403	42.784
		Mesh size N	32	64	128	256	512
time(ms)	$p = 2$	CPU	787.065	3279.467	13300.709	53283.997	212828.953
		GPU	30.374	91.682	701.729	2446.404	7851.896
		speedup	25.920	35.770	18.973	21.784	27.087

Table 2: GPU relative speedup compare to CPU implementation on different mesh sizes.

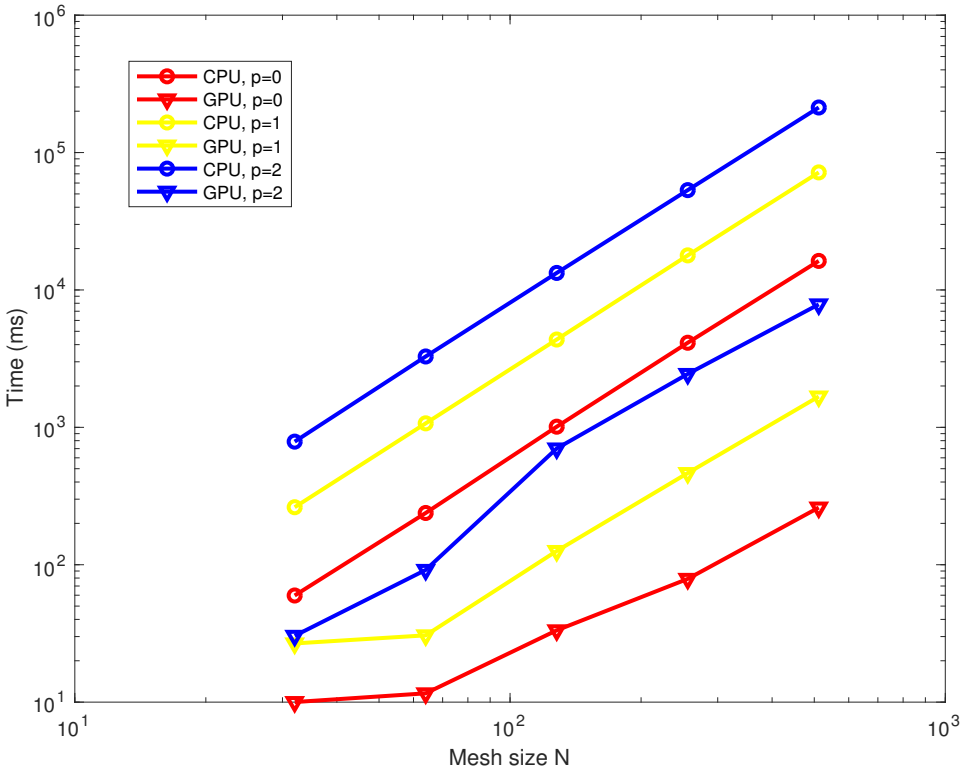


Figure 3: GPU scaling

First we can see in the table that the GPU implementation achieves considerable speedups compared to the serial CPU code, demonstrating the effectiveness of our implementation. Ideally, if the resources (compute power and memory band-width) are sufficient, GPU can handle problems with different sizes in similar amount of time. In practice, this can not be achieved due to the compute and memory latency and hence the GPU timing also scales with the problem size. As we can see in the Table 2 and Figure 3, the CPU timing scales linearly with respect to the problem size (linear in log with respect to N, *i.e.*, linear with respect to the total work $\Theta(N^2)$), GPU scales sub-linearly for lower order p, while linearly for higher orders.

4.4 Performance Metrics

A detailed profiling is done for a fixed mesh with $N = 128$ but various approximation orders, the memory band-width and floating point operations per second are measured and summarized in Table 3. As we can see in the table that the face integral kernel has higher Flops and in general lower band-width. The high Flops in face integral kernel is achieved mainly in the numerical flux $\hat{\mathbf{F}}$ calculations, which repeatedly uses the data loaded. On the other hand, the face integral kernel loads the data from adjacent elements, which are in general not contiguous in memory, resulting a low memory band-width. On the contrary, the volume integral kernel only reads the data belongs to the same element, which has higher

band-width. However, as the data operations are not intense in this kernel, the Flops are lower compared to the face integral kernel. Moreover, as the order p increases, the volume integral memory burden becomes worse as the degrees-of-freedom inside an element grow as p^2. This explains the Flops loss as $p = 2$ for the volume integral kernel. Finally, in current work, we store the global state vector in a way that all degrees-of-freedom per element are stored contiguous. This is in general how we store the state data in DG. However, this is not optimal for GPU as each thread accesses different element state vectors, prevents coalescing memory access in each thread warp. The profiling results suggest more optimization on memory to achieve higher memory band-width and higher Flops.

	Kernel	`calculateVolumeRes`	`calculateFaceRes`
$N = 128, p = 0$	Band-Width (GB/s)	129.1	77.4
	Flops (GFlop/s)	25.9	72.7
$N = 128, p = 1$	Band-Width (GB/s)	143.8	87.5
	Flops (GFlop/s)	27.9	68.6
$N = 128, p = 2$	Band-Width (GB/s)	100.1	107.7
	Flops (GFlop/s)	12.08	51.4

Table 3: Peak memory bandwidth and flops for the two main kernels

5 Conclusion and Future Work

In this work, we implemented the discontinuous Galerkin (DG) finite element method to simulate the vortex transport problem. The code is tested and verified on different testing meshes, showing the correctness of the current implementation. The comparison with a CPU implementation shows the effectiveness of the GPU acceleration in DG simulations. More detailed performance profiling shows that the memory band-width and double precision flops are relative low compare to the device peak performance. As the main kernels involve considerable memory transactions, the memory optimization is essential for achieving good performance. Furthermore, our current code partitions the work to each element or each face, largely due to the relatively easy implementation. However, these work partition may not be the optimal. Better parallel algorithm for DG is also a potential direction for the future work.

References

[1] Bernardo Cockburn and Chi-Wang Shu. Runge–kutta discontinuous galerkin methods for convection-dominated problems. *Journal of Scientific Computing*, 16(3):173–261, 2001.

[2] F. Bassi and S. Rebay. GMRES discontinuous Galerkin solution of the compressible Navier-Stokes equations. In Bernardo Cockburn, George Karniadakis, and Chi-Wang Shu, editors, *Discontinuous Galerkin Methods: Theory, Computation and Applications*, pages 197–208. Springer, Berlin, 2000.

[3] P.L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43:357–372, 1981.

[4] Design Guide. Cuda c programming guide. *NVIDIA, July*, 2013.