Discontinuous Galerkin Methods on Graphics Processing Units for Nonlinear Hyperbolic Conservation Laws

Martin Fuhry        Lilia Krivodonova

June 5, 2013

Abstract

We present an implementation of the discontinuous Galerkin (DG) method for hyperbolic conservation laws in two dimensions on graphics processing units (GPUs) using NVIDIA’s Compute Unified Device Architecture (CUDA). Both flexible and highly accurate, DG methods accommodate parallel architectures well, as their discontinuous nature produces entirely element-local approximations. High performance scientific computing suits GPUs well, as these powerful, massively parallel, cost-effective devices have recently included support for double-precision floating point numbers. Computed examples for Euler equations over unstructured triangle meshes demonstrate the effectiveness of our implementation. Benchmarking our method against a serial implementation reveals a speedup factor of over 50 times using double-precision with an NVIDIA GTX 580.

1 Introduction

Graphics processing units (GPUs) are becoming increasingly powerful, with some of the latest consumer offerings boasting over a teraflop of double precision floating point computing power. Their theoretical computing performance compared with their price exceeds that of nearly any other computer hardware. With the introduction of software architectures such as NVIDIA’s CUDA [13], GPUs are now regularly being used for general purpose computing. With support for double-precision floating point operations, they are now a serious tool for scientific computing. Even large-scale computational fluid dynamics (CFD) problems with several million element meshes can be easily handled in a reasonable amount of time, as video memory capacities continue to increase.

We describe an implementation of a modal discontinuous Galerkin (DG) method for solutions of nonlinear, two-dimensional, hyperbolic conservation laws on unstructured meshes for GPUs. High-order DG methods can be particularly suitable for parallelization on GPUs as the arithmetic intensity, that is, the ratio of computation time to memory access time, for high-order methods is significantly higher than, e.g., finite difference and finite volume methods due to the higher concentration of degrees-of-freedom per element. The amount of work per degree-of-freedom is also higher as it requires a computationally intensive high-order quadrature rule evaluation over an element and its boundary.
Compared to structured grid methods, implementations of unstructured grid CFD solvers have been sparse. Lower-order finite volume solvers handling nonlinear problems in [1, 5] have been successfully applied. The most relevant example to this work is Kloeckner et al [10, 9], who implemented a high-order nodal DG method for linear hyperbolic equations. This approach simplifies computation by taking advantage of linearity, i.e., the integrals may be precomputed and the resulting structure is dominated by a matrix-vector multiplication. Nonlinear solvers, on the other hand, must recompute nonlinear fluxes on each element and at every timestep. Thus, the main difference between linear and nonlinear DG solvers lies in the ability (or inability) to precompute the majority of the work. In addition, Riemann solvers and, especially, limiters require careful incorporation into DG implementations for nonlinear problems.

In comparison to a serial implementation, we obtain a speedup of more than 50 times using double-precision with an NVIDIA GTX 580, which favorably compares with results presented in previous work [10, 9, 7, 1, 5, 8, 3]. At present, a meaningful comparison between different implementations is difficult to make, as newer hardware rapidly replaces older hardware. In addition, implementations performing in single-precision will currently outperform those done in double-precision by a significant factor. At present, efficient implementations achieve a 30-50 times speedup over comparable serial versions.

Our implementation demonstrates that a straightforward, but careful, approach can achieve a significant gain even for complicated problems. We see a comparable performance to [10] by operating on a much simpler thread-per-element and thread-per-edge basis without needing to use shared memory between threads. As global memory transfers often represent the bottleneck for GPU computations, memory access patterns can completely determine the arithmetic intensity, and therefore, the overall performance of the implementation. We diverge from the usual memory storage patterns by storing solution coefficients, i.e. degrees of freedom associated with one element, appropriately allowing consecutive threads to access nearby data; in particular, we store these basis-function-wise as opposed to element-wise. We also extensively take advantage of constant memory, storing all possible precomputed evaluations for fast access.

The two major components of our implementation involve evaluating an integral over an element and evaluating an integral over its boundary. While the former computation is straightforward to implement, two different approaches may be taken towards implementing the latter. Element-wise integration over edges results in evaluating the same integral twice. Edge-wise integration significantly increases thread count and avoids performing this computation twice, but introduces race conditions for unpartitioned, unstructured meshes. Preventing these race conditions when parallelizing edge-wise proved particularly challenging, as atomic operators significantly degrade performance. Despite these difficulties, we found that the edge-wise approach provided roughly twice the performance as the element-wise approach.

In the final section of our paper, we present several representative examples of the Euler equations with and without limiting. We measure runtime performance, speedup over a serial implementation, and scalability for order of approximation $p$ ranging from one to five. To take full advantage of the processing power available in our test GPUs, we found that a mesh size of roughly 10,000 elements suffices. With $p = 1$, we found that problems of up to four million elements can be easily handled for Euler equations. Finally, we found that while limiters certainly inhibit performance on GPUs as they do
on CPUs, performance in our implementation does not degrade significantly.

2 The Discontinuous Galerkin Method

We are interested in the numerical approximation of the two-dimensional system of equations

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

for a vector $u = (u_1, u_2, \ldots, u_m)$, over a computational domain $\Omega \subset \mathbb{R}^2$, with a sufficiently smooth flux function $F = [F_1, F_2]$, where $F_1$ and $F_2$ are the fluxes in the $x$ and $y$ directions, respectively. We enforce the initial conditions

$$u(x, y, 0) = u_0(x, y),$$

and appropriate boundary conditions. We first partition $\Omega$ into a mesh of triangles

$$\Omega = \bigcup_{i=1}^{N} \Omega_i,$$

then multiply (1) by a test function $v \in H^1(\Omega_i)$, integrate over the element $\Omega_i$, and use the divergence theorem to obtain the weak formulation

$$\frac{d}{dt} \int_{\Omega_i} v u \, dx + \int_{\Omega_i} \nabla v \cdot F(u) \, dx - \int_{\partial \Omega_i} v F(u) \cdot n_i \, ds = 0,$$

where $n_i$ is the unit, outward-facing normal for element $\Omega_i$'s edges.

To obtain a simpler formulation, we map each $\Omega_i$ to a canonical triangle $\Omega_0$ with vertices at $(0, 0)$, $(1, 0)$, and $(0, 1)$ with the bijection

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 - r - s \\ r \\ s \end{pmatrix},$$

where $(x_k, y_k)$, $k = 1, 2, 3$, are the vertices of the given element, and $r = (r, s) \in \Omega_0$. We arrange the ordering of the vertices of $\Omega_i$ in a counter-clockwise direction to enforce a positive determinant of the Jacobian of the transformation (5), which we call $J_i$. With this mapping, the integral of the flux over $\Omega_i$ in the weak formulation (4) over element $\Omega_i$ becomes

$$\int_{\Omega_i} \nabla v \cdot F(u) \, dx = \int_{\Omega_0} (J_i^{-1} \nabla v) \cdot F(u) \det J_i \, dr,$$

which we refer to as the volume integral.

We separately map each edge of $\Omega_i$ to the canonical interval $I_0 = [-1, 1]$ by the bijective mapping given by

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \begin{pmatrix} \frac{1}{2}(1 - \xi) \\ \frac{1}{2}(1 + \xi) \end{pmatrix},$$

where $(x_k, y_k)$, $k = 1, 2, 3$, are the vertices of the given element, and $\xi \in [-1, 1]$. We arrange the ordering of the vertices of $\Omega_i$ in a counter-clockwise direction to enforce a positive determinant of the Jacobian of the transformation (5), which we call $J_i$. With this mapping, the integral of the flux over $\Omega_i$ in the weak formulation (4) over element $\Omega_i$ becomes

$$\int_{I_0} \nabla v \cdot F(u) \, dx = \int_{\Omega_0} (J_i^{-1} \nabla v) \cdot F(u) \det J_i \, d\xi,$$

which we refer to as the volume integral.
Figure 1: One basis function $\phi_5$ evaluated at $\partial\Omega_0$

(a) $\phi_5$ evaluated over $\Omega_0$

(b) $\phi_5$ evaluated over $\partial\Omega_0$

where $(x_k, y_k)$, $k = 1, 2$, are the endpoints of the given edge and $\xi \in I_0$. Using this mapping to $I_0$, the integral over $\partial\Omega_i$ in the weak formulation (4) becomes

$$\int_{\partial\Omega_i} v \mathbf{F}(\mathbf{u}) \cdot \mathbf{n}_i \, ds = \sum_{q=1}^{3} \int_{I_0} v_q \mathbf{F}(\mathbf{u}) \cdot \mathbf{n}_{i,q} l_{i,q} \, d\xi, \quad \text{(8)}$$

where $\mathbf{n}_{i,q}$ denotes the unit outward-facing normal vector for edge $q$, $v_q$ is the test function $v$ restricted to that edge, and $l_{i,q}$ denotes the determinant of the Jacobian of (7), given by

$$l_{i,q} = \frac{1}{2} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \quad \text{(9)}$$

We refer to (8) as the surface integral.

Let $\Phi = \{\phi_j\}_{j=1}^{N_p}$ be an orthonormal basis for the space of polynomials of degree at most $p$ on $\Omega_0$, $S^p(\Omega_0)$, consisting of

$$N_p = \frac{1}{2} (p + 1)(p + 2) \quad \text{(10)}$$

basis functions given in, e.g., [11]. Let $\mathbf{U}_i$ approximate $\mathbf{u}$ over $\Omega_i$ using a linear combination of basis functions $\phi_j$,

$$\mathbf{U}_i = \sum_{j=1}^{N_p} \mathbf{c}_{i,j}(t) \phi_j, \quad \text{(11)}$$

where $\mathbf{c}_{i,j}(t)$ is a vector of solution coefficients. Choosing each test function $v = \phi_j \in \Phi$ in (??) and using the orthonormality of the basis creates the system of ODEs

$$\frac{d}{dt} \mathbf{c}_{i,j}(t) = \frac{1}{\det J_i} \left( \int_{\Omega_0} \mathbf{F}(\mathbf{U}_i) \cdot (J_i^{-1} \nabla \phi_j) \, d\mathbf{r} - \sum_{q=1}^{3} \int_{I_0} \phi_{j,q} \mathbf{F}(\mathbf{U}_i) \cdot \mathbf{n}_{i,q} l_{i,q} \, d\xi \right), \quad \text{(12)}$$
where $\phi_{j,q}$ denotes the basis function $\phi_j$ evaluated on the edge $q$, as demonstrated in Figure 1. Note that, in general, $\phi_j$ assumes different values on each edge $q$.

The numerical solution on the edges of each element is twice defined, as we do not impose continuity between elements. To resolve these ambiguities, we replace the flux $F(U_i)$ on $\partial \Omega_i$ with a numerical flux function $F_n(U_i, U_k)$, using information from both the solution $U_i$ on $\Omega_i$ and the solution $U_k$ of the neighboring element $\Omega_k$ sharing that edge.

3 Implementation

GPU computing, while providing perhaps the most cost-effective performance to date, is not without its challenges. The sheer floating point power dispensed by GPUs is only realized when unrestricted by the GPUs limited memory bandwidth. As such, it is of paramount importance to carefully manage memory access. The three main CUDA memory containers we use in this implementation are the global, local, and constant memories. Global memory on the GPU is very large, but is very slow to access, introducing large latencies. We therefore manage global memory by minimizing these memory accesses, coalescing the necessary global memory transactions by instructing nearby groups of threads to access the same blocks of memory. Local memory is physically located in global memory, but is cached, allowing much faster access. Constant memory is cached and extremely fast, but very limited.

Programming GPUs using CUDA is unlike programming CPUs [13]. CUDA machine code is separated into kernels - parallel code to be executed on the GPU. Global data in dedicated video memory is manipulated by many threads running the same instruction set. Programming is done from a thread perspective, as each thread reads the same instruction set, using it to manipulate different data. Threads have a unique numeric identifier, a thread index, allowing, e.g., each thread to read from a different memory location. In order to take full advantage of the processing power available, enough warps, that is, collections of sixteen threads, must be created to fully saturate the device. Load balancing is done entirely by CUDA’s warp scheduler. When memory access requests introduce latency, e.g., global memory access requests, the warp scheduler will attempt to hide the latency by swapping out the idle warps and replacing them with ready warps. Problems with low arithmetic intensity spend most of their run time waiting during latencies in memory transactions. Problems with high arithmetic intensity, on the other hand, allow CUDA’s warp scheduler to hide memory latency behind computation by scheduling ready warps ahead of those waiting on memory transactions.

Boolean expressions pose problems for GPUs, as two threads in the same warp may evaluate a boolean condition to different values. When this happens, the warp splits into branches, where each branch of threads executes a different path of code split by the boolean. This branching, called warp divergence, harms performance as both instructions must be run for this warp.

Below, we give a brief overview of our parallelization approach for this implementation. The following subsections provide more detailed descriptions of our method. The right-
hand side of equation (12) combines a volume integral

$$\int_{\Omega_0} \mathbf{F}(\mathbf{U}_i) \cdot (J_i^{-1} \nabla \phi_j) \det J_i d\mathbf{r}$$

and a surface integral

$$\sum_{q=1}^{3} \int_{I_0} \phi_{j,q} \mathbf{F}(\mathbf{U}_i) \cdot \mathbf{n}_{i,q} l_{i,q} d\xi$$

consisting of three independent line integrals, for each $j = 1, \ldots, N_p$. Parallelization is therefore straightforward. As the volume integral contributions (13) for one element require only local information from that element, i.e., the element’s coefficients and inverse Jacobian, these contributions can be computed independently from each other. Similarly, each edge’s surface integral contributions (14) are computed independently as these computations require local information from two elements sharing that edge, i.e., both elements’ coefficients. An explicit integration scheme advances the solution coefficients in time.

We use quadrature rules in [6] of order $2p$ to approximate the volume integral and the Gauss-Legendre quadrature rules of order $2p + 1$ to approximate the surface integral. At each integration point $\mathbf{r}_k$ in the interior of $\Omega_0$ shown in Figure 2a, we precompute the values of $\phi_j(\mathbf{r}_k)$ and $\nabla \phi_j(\mathbf{r}_k)$. We also precompute the values of $\phi_j(\mathbf{r}_{q,k})$ at the integration points $\mathbf{r}_{q,k}$ on $\partial \Omega_0$, shown in Figure 2b, where $q = 1, 2, 3$ denotes which side of the canonical triangle the integration points reside on. By storing these precomputed evaluations in GPU constant memory, we drastically reduce computation time.

3.1 Mesh Connectivity

Figure 3 shows an example of a simple mesh consisting of two elements. Elements $\Omega_1$ and $\Omega_2$ point to their respective edges; edges $e_1, \ldots, e_5$ point back to their respective elements. Each edge stores two pointers: one each to its left and right elements, $\Omega_l$ and $\Omega_r$. We arbitrarily assign each of the two elements sharing an edge as either left or right. Edges lying on the boundary of the domain have only a left element and store a negative
integer describing the type of boundary conditions assigned to them in place of a pointer to a right element. The normal vector belonging to an edge points from $\Omega_l$ to $\Omega_r$, by our convention. For edges lying on the boundary of the domain, the normal vector therefore points outward.

3.2 Volume Integration Kernel

Parallelizing the volume integral contribution (13) computation is straightforward. The integration over $\Omega_i$ requires only local information, i.e., the coefficients $c_{i,j}, j = 1, \ldots, N_p,$ and the precomputed matrix $J^T_i = (\det J_i)J_i^{-1}$ for that element. We create one thread for each $\Omega_i$ tasked with computing (13) over that element. Figure 4 shows the volume integral computations in this kernel from a thread perspective. Thread $t_i$ reads the coefficients $c_{i,j}, j = 1, \ldots, N_p,$ and the matrix $J^T_i$. It evaluates $U_i$ and the flux $F(U_i)$ at the interior integration point $r_k$. The result is multiplied by each $\nabla \phi_j(r_k), j = 1, \ldots, N_p,$ and added to a right-hand side storage variable $RHS\_volume$.

At each integration point $r_k$, we compute

$$U_i(r_k) = \sum_{j=1}^{N_p} c_{i,j} \phi_j(r_k).$$

(15)
Figure 5: Two orderings and memory access patterns for the coefficients

(a) Storing coefficients element-wise results in an uncoalesced memory access pattern

(b) Storing coefficients basis-function-wise results in a coalesced memory access pattern

We precompute the matrix $\Psi$ with elements $(\Psi)_{j,k} = \phi_j(r_k)$ and store it in GPU constant memory. Therefore, to compute $U_i$ at any integration point $r_k$, we need only compute a vector-vector multiplication. We also precompute and store the constant matrix $J^r_i$ for each $\Omega_i$ in GPU global memory.

The solution coefficients can be organized as a vector and stored as a continuous block in memory. To allow for coalesced memory access, the storage arrangement must defer from the one usually employed in serial implementations. Instead of storing the coefficients for one element side-by-side, we store the coefficients for one order, i.e. corresponding to one basis function, side-by-side. This difference in ordering is illustrated in Figure 5b for a scalar equation. A similar approach can be taken for systems of equations. Similarly, Jacobians are stored as four vectors of length $N$ with each vector representing a different component of the matrix.

Each thread in a block must read and store $N_p \times m$ doubles to load coefficients $c_{i,j}$ into memory. Exclusively using register memory to store these coefficients quickly overloads the GPU streaming multiprocessors’ registers, thereby reducing thread occupancy. We therefore use local rather than register memory to store the coefficients. This optimization increases thread occupancy and reduces our computation time by a factor of nearly thirteen.

### 3.3 Surface Integration Kernel

Parallelizing the surface integral contribution (14) computation proves more challenging. There are two approaches to evaluating the surface integral contribution in (12): element-wise and edge-wise. In the element-wise approach, one thread is created for each element to evaluate (14). Alternatively, in the edge-wise approach, one thread evaluates the surface integral over a single edge, adding the resulting contributions to the elements sharing that edge. We found that the latter approach is about twice as fast. We thus
create one thread \( t_i \) for each edge \( e_i \), shared by elements \( \Omega_l \) and \( \Omega_r \) to compute
\[
\int_{I_0} \phi_{j,q} F_n(U_l, U_r) n_i l_i \, ds.
\] (16)

The numerical flux function \( F_n(U_l, U_r) \) is identical in (16) for coefficients \( c_{l,j} \) and \( c_{r,j} \). This function represents the most expensive computation in the surface integral, and we therefore structure our implementation to compute it only once for each edge. Additionally, the Jacobian (7) remains the same for both \( c_{l,j} \) and \( c_{r,j} \), and the normal vector may be reused, as it points from the left to the right element.

While the numerical flux, the Jacobian, and the normal are the same, the final surface integral contributions along an edge are not necessarily the same for \( c_{l,j} \) and \( c_{r,j} \). As our implementation supports any unstructured triangular mesh, elements \( \Omega_l \) and \( \Omega_r \) may map the same edge to different sides of the canonical triangle. For example, \( \Omega_l \) may map edge \( e_i \) to the side defined by \((0, 0), (0, 1)\), which we call \( s_1 \), while \( \Omega_r \) maps that same edge to the side defined by \((1, 0), (0, 1)\), which we call \( s_2 \). In this case, the basis function evaluated on \( s_1 \) is not equal to the same basis function evaluated on \( s_2 \); refer to Figure 1.

We store two identifiers, \( L \) and \( R \), for each edge \( e_i \) to indicate which side of the canonical triangle \( \Omega_0 \) that edge \( e_i \) is mapped to by elements \( \Omega_l \) and \( \Omega_r \), respectively. These identifiers, which we call side mappings, determine which side of the canonical triangle to evaluate the basis function on in (14). We therefore compute two separate integrals in the same thread for edge \( e_i \), for each \( j = 1, \ldots, N_p \). First, we compute
\[
\int_{I_0} \phi_{j,L} F_n(U_l, U_r) n_i l_i \, ds,
\] (17)
for the surface integral contribution to \( c_{l,j} \), and then we compute
\[
\int_{I_0} \phi_{j,R} F_n(U_l, U_r)(-n_i) l_i \, ds,
\] (18)
for the surface integral contribution to \( c_{r,j} \), where \( n_i \) represents the normal vector pointing from \( \Omega_l \) to \( \Omega_r \) and \( l_i \) indicates the Jacobian given in (7). As discussed above, (17) is not necessarily equal to (18), and two separate surface integrals must be computed for each side.

Race conditions prevent us from simply adding the resulting surface integral contributions together with the volume integrals for \( \Omega_l \) and \( \Omega_r \) as we compute them. For example, two threads assigned to two edges belonging to the same \( \Omega_i \) may compute their surface integral contributions at the same time. When they both attempt to simultaneously add that contribution to \( c_{i,j} \), that memory becomes corrupted. We attempted to use the \texttt{atomicAdd} operator, in order to bypass race conditions by serialising conflicting addition operations in this kernel. Atomic operators are known, however, to significantly degrade performance; in our implementation, runtime while using atomic operators increased by a factor of nine. In order to avoid using atomic operators, we chose instead to store each term separately in GPU global memory variables \texttt{RHS\_surface\_left} and \texttt{RHS\_surface\_right} and combine them later.

The parallel computation of (16) is schematically demonstrated from a thread perspective in Figure 6. Each thread reads the coefficients for its left and right elements...
Figure 6: Thread structure for the surface integral kernel

\[
\begin{align*}
&\text{\textbf{Figure 6: Thread structure for the surface integral kernel}} \\
&\begin{array}{c}
\begin{pmatrix} c_{l,1} \\ \vdots \\ c_{l,N_p} \end{pmatrix} \\
U_l \\
F_n(U_l, U_r) \\
\int_{I_0} \phi_{j,L} F_n(U_l, U_r) n_l l_1 d\xi \\
\text{RHS surface left}[0]
\end{array}
\end{align*}
\]

\[
\begin{align*}
&\begin{array}{c}
\begin{pmatrix} c_{r,1} \\ \vdots \\ c_{r,N_p} \end{pmatrix} \\
U_r \\
F_n(U_l, U_r) \\
\int_{I_0} \phi_{j,R} F_n(U_l, U_r)(-n_1) l_1 d\xi \\
\text{RHS surface right}[0]
\end{array}
\end{align*}
\]

\[
\begin{align*}
&\text{\textbf{Figure 6: Thread structure for the surface integral kernel}} \\
&\begin{array}{c}
\begin{pmatrix} c_{l,1} \\ \vdots \\ c_{l,N_p} \end{pmatrix} \\
U_l \\
F_n(U_l, U_r) \\
\int_{I_0} \phi_{j,L} F_n(U_l, U_r) n_{l_1} l_{N_e} d\xi \\
\text{RHS surface left}[N_e-1]
\end{array}
\end{align*}
\]

\[
\begin{align*}
&\begin{array}{c}
\begin{pmatrix} c_{r,1} \\ \vdots \\ c_{r,N_p} \end{pmatrix} \\
U_r \\
F_n(U_l, U_r) \\
\int_{I_0} \phi_{j,R} F_n(U_l, U_r)(-n_{N_e}) l_{N_e} d\xi \\
\text{RHS surface right}[N_e-1]
\end{array}
\end{align*}
\]

c_{l,j} and c_{r,j} for j = 1, \ldots, N_p, as information from both \(\Omega_l\) and \(\Omega_r\) is required in order to compute the numerical flux function \(F_n(U_l, U_r)\). The unstructured nature of our mesh prohibits us from sorting the left and right coefficients in memory to allow coalesced reads in both the volume integral kernel and the surface integral kernel. We may, however, sort the edge index list to enable coalesced reads for either \(c_{l,j}\) or \(c_{r,j}\) by appropriately reordering one or the other, but not both. As the computations of (17) and (18) also require the normal vector \(n_i\) and constant \(l_i\), we precompute and store these values in GPU global memory, sorted edge-wise for coalesced access by the threads in this kernel.

One final detail we emphasise involves the ordering of our boundary integration points. We map integration points from \(I_0\) to their respective sides of the canonical triangle in

Figure 7: The integration points for \(\Omega_l\) and \(\Omega_r\) must be traversed in opposite directions.

\[
\begin{array}{c}
\Omega_0 \\
r_{1,1}
\end{array}
\begin{array}{c}
\Omega_l \\
r_{1,1}
\end{array}
\begin{array}{c}
\Omega_r \\
r_{2,3}
\end{array}
\]

\[
\begin{array}{c}
\Omega_0 \\
r_{1,1}
\end{array}
\begin{array}{c}
\Omega_l \\
r_{1,1}
\end{array}
\begin{array}{c}
\Omega_r \\
r_{2,3}
\end{array}
\]

\[
\begin{array}{c}
\Omega_0 \\
r_{1,1}
\end{array}
\begin{array}{c}
\Omega_l \\
r_{1,1}
\end{array}
\begin{array}{c}
\Omega_r \\
r_{2,3}
\end{array}
\]

\[
\begin{array}{c}
\Omega_0 \\
r_{1,1}
\end{array}
\begin{array}{c}
\Omega_l \\
r_{1,1}
\end{array}
\begin{array}{c}
\Omega_r \\
r_{2,3}
\end{array}
\]

\[
\begin{array}{c}
\Omega_0 \\
r_{1,1}
\end{array}
\begin{array}{c}
\Omega_l \\
r_{1,1}
\end{array}
\begin{array}{c}
\Omega_r \\
r_{2,3}
\end{array}
\]
a counter-clockwise direction, as illustrated in Figure 2b. The integration points must be traversed in opposite directions for $\Omega_l$ and $\Omega_r$ in order to reside in the same physical space, as shown in Figure 7. Thus, the flux evaluation at the $k$'th integration point is

$$F_n(U_l(r_{L,k}), U_r(r_{R,2p+1-k})). \quad (19)$$

We precompute three matrices $(\Psi_q)_{j,k} = \phi_j(r_{q,k})$ for each side $q = 1, 2, 3$, of the canonical triangle and store them row-by-row in a single, flattened array in GPU constant memory. By using each edge's two side mapping indices as an offset, we are able to lookup the correct integration points to use while avoiding boolean evaluations, which would create warp divergence.

If edge $e_i$ lies on a domain boundary, a ghost state $U_g$ is created and assigned to $U_r$. We index our edges so that all boundary edges appear sorted and first in our edge list. This avoids warp divergence, as the boundary edges with boundaries of the same type will be grouped in the same warp.

Each thread must read $2 \times N_p \times m$ coefficients to compute $U_l$ and $U_r$. Like in the volume integral kernel, storing these coefficients in register memory would inhibit maximum thread capacity. As such, we store both $c_{l,j}$ and $c_{r,j}$ in local memory.

### 3.4 Right-Hand Side Evaluator Kernel

The right-hand side evaluator kernel combines data from the three temporary storage variables $RHS_{surface_left}$, $RHS_{surface_right}$, and $RHS_{volume}$ from each $\Omega_i$ to compute the right-hand side of equation (12). Each thread $t_i$ in the right-hand side evaluator kernel combines the contributions from the surface and volume integrals for coefficients $c_{i,j}$, $j = 1, \ldots, N_p$. For each edge of element $\Omega_i$, the thread must determine if that edge considers element $\Omega_i$ a left or right element. If the edge considers $\Omega_i$ a left element, it reads from $RHS_{surface_left}$; on the other hand, if it considers $\Omega_i$ a right element, it reads from $RHS_{surface_right}$. In either case, the thread reads from each $j = 1, \ldots, N_p$, appropriate memory location of the three temporary storage variables, combining them to form the right-hand side of (12).

As each thread must determine if $\Omega_i$ is considered a left or a right element by each of its edges, three boolean evaluations must be computed in this kernel. This introduces unavoidable warp divergence.

### 3.5 Limiting Kernel

We implement the Barth-Jespersen limiter [2] for linear $p = 1$ approximations. We aim to limit the maximum slope in the gradient of the scalar equation

$$U_i(r) = \bar{U}_i + \alpha_i(\nabla U_i) \cdot (r - r_0), \quad (20)$$

by selecting a limiting coefficient $\alpha_i$. In (20), $\bar{U}_i$ is the average value of $U_i$ over $\Omega_i$ and $r_0$ is the coordinate of the centroid of $\Omega_0$. Limiting systems of equations involves finding a separate $\alpha_i$ for each variable in the system.

Suppose that element $\Omega_i$ is surrounded by elements $\Omega_a$, $\Omega_b$, and $\Omega_c$, as shown in Figure 8. We choose $\alpha_i$ so that $U_i$ introduces no new local extrema at the integration points on
Figure 8: To limit the solution over $\Omega_i$, we evaluate the centroid values of surrounding elements $\Omega_a$, $\Omega_b$, and $\Omega_c$

the boundaries relative to these three surrounding elements. We first evaluate $U_i$, $U_a$, $U_b$, and $U_c$ at their centroids. We then define the maximum centroid value

$$U_{i}^{\text{max}} = \max \{U_i(r_0), U_a(r_0), U_b(r_0), U_c(r_0)\}$$  \hfill (21)

and minimum centroid value

$$U_{i}^{\text{min}} = \min \{U_i(r_0), U_a(r_0), U_b(r_0), U_c(r_0)\}.$$  \hfill (22)

Our implementation of this limiter operates element-wise. Each thread $t_i$ computes $\alpha_i$ to limit the slope of the approximation over a single element $\Omega_i$. Thread $t_i$ first computes $U_{i}^{\text{max}}$ and $U_{i}^{\text{min}}$ as in (21) and (22). Then, at each integration point $r_{q,k}$ on the boundary of $\Omega_i$, thread $t_i$ computes $U_i(r_{q,k})$ in order to compute $\alpha_{i,k}$. The smallest of the $\alpha_{i,k}$ values becomes the limiting constant $\alpha_i$. Finally, the coefficients $c_{i,2}$ and $c_{i,3}$ are multiplied by this $\alpha_i$. This is repeated for each variable in the system.

Each evaluation of $\alpha_i$ requires a significant number of boolean evaluations. As such, unavoidable warp divergence certainly inhibits performance. This will be shown numerically in Section 5.1.1.

3.6 Memory Management

Here we summarize the considerations put in storing data. We store the variables that grow with mesh size in the global memory and the data independent of mesh size in the constant memory.

In GPU constant memory, we store precomputed basis functions evaluated at the interior and edge integration points and those basis functions’ gradients evaluated at only the interior integration points. In addition, we store all of the integration points and weights on $\Omega_0$ and $\partial\Omega_0$. GPU constant memory is easily able to store these for practical values of degree of approximation $p$. For example, with $p = 5$, we store 2,268 doubles for the precomputed data, which occupies only twenty-nine percent of the available constant memory space on NVIDIA Fermi architectures.

In global memory, we store the following: solution coefficients, precomputed matrices $J^T_i$, normals, and the determinants of edge mappings’ Jacobians (7). All precomputed
data is appropriately sorted element-wise or edge-wise to allow coalesced reads as discussed in Sections 3.2 and 3.3.

The total memory required for computation depends on four factors. First, the size of the mesh $N$ determines the number of elements and edges, as we store the element vertices in GPU global memory. Second, the degree of the polynomial approximation $N_p$ determines the number of coefficients required to approximate the solution. Third, the size of the system, $m$, requires a vector of solution coefficients for each variable in that system. For each element, we require $m \times N_p$ coefficients to represent the approximated solution over that element. Finally, the ODE solver typically needs extra storage variables for intermediate steps or stages, which must be stored in global memory.

4 Computed Examples

<table>
<thead>
<tr>
<th>Table 1: GPU Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA GTX 580</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>CUDA Cores</td>
</tr>
</tbody>
</table>

We now present computed examples from this implementation of the DG method. Each example demonstrates solutions of Euler equations in two dimensions. Our simulations ran on two different graphics cards on separate workstations, detailed in Table 1. All tests were run on Ubuntu Linux using CUDA 4.0.

This implementation makes use of double precision floating point numbers whenever necessary. In NVIDIA Fermi GPU architectures used in these tests, double precision computing is performed at one quarter the speed of single precision computing. See [13] for more information.

Mesh generation and postprocessing was done using GMSH and a custom Python script. All solutions displayed in GMSH were plotted using linear interpolation with no smoothing applied. The discontinuous nature of the numerical solution allows sharp jumps at isolines whenever solution values differ greatly between elements.

The Euler equations describe the flow of an inviscid, isotropic, compressible fluid. In two dimensions, they are given by

$$\partial_t \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix} + \partial_x \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u(E + p) \end{pmatrix} + \partial_y \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ v(E + p) \end{pmatrix} = 0,$$

where $\rho$ is the density, $u$ and $v$ are the velocity components, and $E$ is the energy. The variable $p$ in equation (23) is the pressure given by an equation of state, which we choose to be

$$p = (\gamma - 1) \left( E - \frac{\rho ||v||^2}{2} \right),$$

for an adiabatic constant $\gamma$ and velocity vector $v = (u, v)$. For air, we take $\gamma = 1.4$. 

13
Table 2: $L^2$ error in density and convergence rate $r$ for levels of $h$- and $p$-refinement for the supersonic vortex test problem

<table>
<thead>
<tr>
<th>Mesh</th>
<th>$p = 1$</th>
<th>$p = 2$</th>
<th>$p = 3$</th>
<th>$p = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{Error}$</td>
<td>$r$</td>
<td>$\text{Error}$</td>
<td>$r$</td>
</tr>
<tr>
<td>$A$</td>
<td>4.934E−3</td>
<td>-</td>
<td>3.708E−4</td>
<td>-</td>
</tr>
<tr>
<td>$B$</td>
<td>1.226E−3</td>
<td>2.009</td>
<td>6.003E−5</td>
<td>2.627</td>
</tr>
<tr>
<td>$D$</td>
<td>8.695E−5</td>
<td>1.910</td>
<td>1.043E−6</td>
<td>2.953</td>
</tr>
</tbody>
</table>

4.1 Supersonic Vortex

The supersonic vortex test problem models supersonic fluid flow through a curved quarter-cylinder tube. This problem has a known smooth analytical solution, which we prescribe as the initial conditions. We use curved reflecting boundary conditions, detailed in [12], along the curved edges and inflow and outflow boundary conditions at the inlet and outlet. We run the simulation until numerical convergence occurs, defined by

$$\max \{|e_{ij}^{n+1} - e_{ij}^n|\} \leq 10^{-14}. \quad (25)$$

To verify the convergence of this implementation for nonlinear problems, we use a convergence analysis. We call our meshes $A$ through $D$ with mesh $A$ containing 180 elements. Meshes $B$ through $D$ were created by successive refinement of the previous mesh by splitting each triangle into four triangles, quadrupling the total number of elements with each refinement.

The $L^2$ error between the numerical steady state and analytical steady state are compared for each combination of $h$- and $p$-refinement in Table 2. These convergence rates match theoretical convergence rates, verifying the accuracy of our implementation for nonlinear problems. Our results look both quantitatively and qualitatively similar to those found elsewhere; e.g., [12].

4.2 Double Mach Reflection

The double Mach reflection test problem models a planar shock wave over a reflecting angled wedge. This is equivalent to modeling an angled shock wave moving over a straight reflecting wall. The reflections of the initial shock wave create additional shock waves and contact discontinuities during this simulation.

We begin by sending a down- and right-moving incident Mach 10 shock wave with propagation angle $\theta = 60^\circ$ from the $x$-axis; i.e., we assume the wedge has a half-angle of $30^\circ$. The computational domain $\Omega$ is defined by $x = [0, 4], y = [0, 1]$. The lower boundary in our domain models a reflecting wedge beginning at $x_0 = \frac{1}{5}$.

We assume that the unperturbed flow’s density and pressure are equal to 1.4 and 1 respectively. The after shock values are calculated to satisfy the Rankine-Hugoniot condition.

The left boundary condition sets an inflow with $U_s$ as the parameter values. The boundary condition along the top of the domain keeps up with the speed of the incident shock wave to simulate the effects of an infinitely long wave. Along the top boundary, at
Figure 9: Computational domain $\Omega$ for the double Mach reflection test problem. The striped triangle represents the reflecting wedge. The shaded region on the left is the shock region $U_s$ while the region on the right is the pre-shock condition $U_q$.

Figure 10: Density for the double Mach reflection problem using $p = 1$

(a) Density for mesh $C$

(b) Density isolines for mesh $C$
integration points to the left of the shock wave, the exact values from $U_s$ are prescribed, while points to right of the shock wave use the values from $U_q$. The lower boundary condition prescribes the values of the shock $U_s$ at $x \leq x_0$ and uses reflecting boundary conditions beyond to simulate a perfectly reflecting wedge.

Our test set runs over three unstructured, triangular meshes of varying mesh sizes, reported in Table 3. We compute the solution until $t = 0.2$ when the shock has moved nearly across the entire domain. Our solution is computed using $p = 1$ linear polynomials with the slopes limited using the Barth-Jespersen limiter. Mesh refinement is done by setting a smaller maximum edge length and creating a new mesh with GMSH.

The density and density isolines at $t = 0.2$ for the most refined mesh, $C$, are plotted in Figure 10. Our jet stream travels and expands as in similar simulations in [4]. The exact boundary condition at the top edge of our domain introduces small numerical artifacts which can be seen at the front and top of the shock.

Table 3 also reports total runtime for both GPUs used and memory costs for these simulations using the classical second-order Runge-Kutta time integration scheme. The simulation time, even for the very large meshes, is not prohibitive. Meshes of $C$’s size are typically too large to be run in serial implementations, usually requiring supercomputing time. In contrast, the GTX 580 completed this simulation in just over an hour.

5 Benchmarks

We now present our benchmarks of this implementation. First, we demonstrate the speedup when compared with a single-core, serial implementation. Then, we examine the performance degradation introduced by our limiter. Finally, we show how our implementation scales with mesh sizes. In all tests following, we use the classical fourth-order Runge-Kutta time integration scheme.

5.1 Serial Comparison on a CPU

We compare the performance of this implementation with a CPU implementation computing in serial. To create the serial implementation, we rewrote the GPU implementation in C, making the following necessary adjustments. We replace every parallel computation run by a thread with a for loop. We also replace all GPU memory allocation with CPU memory allocations. In addition, we change the memory access pattern in the CPU implementation so that the coefficients are located nearby each other from an element perspective as opposed to a thread perspective. Finally, we remove all unnecessary memory copies as we no longer transfer data between the GPU and CPU. The surface integral contributions are still computed separately from the volume integral contributions and
Table 4: Mesh sizes for the supersonic vortex test problem used for benchmarking

<table>
<thead>
<tr>
<th>Mesh</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements</td>
<td>180</td>
<td>720</td>
<td>2,880</td>
<td>11,520</td>
<td>46,080</td>
<td>184,320</td>
</tr>
<tr>
<td>Edges</td>
<td>293</td>
<td>1126</td>
<td>4,412</td>
<td>17,464</td>
<td>69,488</td>
<td>277,216</td>
</tr>
</tbody>
</table>

later recombined, maintaining the same basic structure as the GPU implementation.

We now compare the computation run time between the CPU and GPU implementations. Both the GPU and CPU will compute the same number of timestep iterations for the supersonic vortex test problem described in Section 4.1 for each combination of mesh size and orders of approximation. Mesh sizes used for this benchmark are reported in Table 4 with each successive mesh obtained from nested refinement of the previous mesh.

The CPU implementation runs on a single core on an Ubuntu 12.04 machine with an Intel Q6600 CPU running at 2.4GHz with 4GB of RAM. The GPU implementation runs on each of the two NVIDIA GPUs described in Table 1. The number of threads per block may differ for each test; only the best results are reported. A sufficient number of timesteps was computed for each of the tests to allow the simulation to run for a nontrivial amount of time on each device. Measured computation time does not include any precomputations, mesh loading time, or the computation of the initial projection from the initial conditions.

Measured speedup factors over meshes D through F are reported in Figure 11. The GTX 460 performed quite well, considering its relative cost to the GTX 580. Unfortunately, the $p = 5$ test on mesh F requires more memory to run than the GTX 460 has available. The speedup factors tend to decrease for higher $p$, although this is not strictly uniform; e.g., $p = 3$ over mesh C outperforms $p = 2$ on the same mesh. The overall result, however, is clear. As $p$ increases, thread work increases without any respective change in parallelism, decreasing our overall speedup. This implementation performs, at best, an impressive 52.5 times faster on the GTX 580 and 31.8 times faster on the GTX 480 than on a single core CPU implementation.

5.1.1 Serial Comparison with Limiting

We also benchmark our implementation’s performance with the Barth-Jespersen limiter described in Section 3.5. We aim to measure how much warp divergence inhibits performance. Using the same supersonic vortex test problem with $p = 1$, we first compare execution time on both GPUs without limiting with the same problem run with the limiter over 10,000 timesteps. Our results, shown in Figure 12, demonstrate that limiting, significantly increases execution time, as expected. We then measure the new speedup factor comparing GPU execution times with CPU execution times with the limiter in Figure 12. The measured speedups with limiting compare surprisingly well with those without limiting in Table 5. We achieve, at most, a 42.6 times speedup on the GTX 580 and a 21.2 times speedup on the GTX 460.
Figure 11: GPU speedups

(a) NVIDIA GTX 460

(b) NVIDIA GTX 580
Figure 12: GPU execution time with $p = 1$ and 10,000 timesteps with and without limiting

(a) NVIDIA GTX 460

(b) NVIDIA GTX 580
Figure 13: GPU execution times

(a) GPU execution time for $p = 1$ and 10,000 timesteps

(b) GPU execution time for $p = 4$ and 1,000 timesteps
Table 5: GPU speedups with limiting and $p = 1$

<table>
<thead>
<tr>
<th>Mesh</th>
<th>$D$</th>
<th>$E$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 460</td>
<td>15.9</td>
<td>21.2</td>
<td>21.2</td>
</tr>
<tr>
<td>GTX 580</td>
<td>30.5</td>
<td>42.3</td>
<td>42.6</td>
</tr>
</tbody>
</table>

### 5.2 Scaling

Our final benchmark demonstrates the scalability of our implementation by measuring performance at device saturation. Device saturation requires a sufficiently large number of threads to be running simultaneously. As the total number of simultaneously running threads depends on the size of the mesh, we reach device saturation by computing over larger meshes.

We first fix $p = 1$ and compute the supersonic vortex test problem described in Section 4.1 over 10,000 timesteps on meshes $A$ through $F$ from Table 4. Next, we fix $p = 4$ and repeat the test using 1,000 timesteps over the same meshes. The computation run times for these two tests are displayed in Figure 13. We see roughly an order of magnitude difference in execution times between $p = 1$ and $p = 4$, even though computational work and degrees of freedom increases by roughly thirty times. This demonstrates that even while the relative speedup compared with a CPU tends to decrease with higher $p$, our implementation does indeed scale quite well when considering the increased number of computations required. We also note that until we reach mesh $C$, execution time does not increase linearly as not enough threads are created to saturate the device.

### 6 Conclusion

Our implementation performs over 50 times faster on the GTX 580 than a CPU implementation and over 30 times faster on the GTX 460 for two-dimensional nonlinear problems. This implementation easily computes the double Mach reflection test problem with nearly one million elements on the GTX 580 in about an hour. The three gigabytes of video memory on the GTX 580 can compute linear approximations on meshes of around four million triangles. New GPU hardware containing even more video memory can allow us to tackle even larger problems.

In future developments, we aim to reduce our memory usage and increase parallelism. Currently, we require three extra temporary storage variables to compute the right-hand side in (12). By eliminating the race conditions through edge list partitioning, we could add each contribution to the right-hand side as we compute them. This would involve developing an edge coloring algorithm, which is not straightforward for complicated meshes. We also would like to increase thread count with higher orders of approximation $p$ to achieve further gains in speedup over serial implementations. Additionally, adaptive mesh refinement would fit into our implementation in a straightforward way. Our implementation is also fast enough to include real-time visualization. Finally, we intend to include support for three-dimensional problems.
7 Acknowledgment

This research was supported in part by Natural Sciences and Engineering Research Council (NSERC) of Canada grant 341373-07.

Bibliography


