Solving Wave Equations on Unstructured Geometries

Andreas Klöckner* Timothy Warburton† Jan S. Hesthaven‡

23rd April, 2013

Waves are all around us–be it in the form of sound, electromagnetic radiation, water waves, or earthquakes. Their study is an important basic tool across engineering and science disciplines. Every wave solver serving the computational study of waves meets a trade-off of two figures of merit– its computational speed and its accuracy. Discontinuous Galerkin (DG) methods fall on the high-accuracy end of this spectrum. Fortuitously, their computational structure is so ideally suited to GPUs that they also achieve very high computational speeds. In other words, the use of DG methods on GPUs significantly lowers the cost of obtaining accurate solutions. This article aims to give the reader an easy on-ramp to the use of this technology, based on a sample implementation which demonstrates a highly accurate, GPU-capable, real-time visualizing finite element solver in about 1500 lines of code.

1 Introduction, Problem Statement, and Context

At the beginning of our journey into high-performance, highly accurate time-domain wave solvers, let us briefly illustrate by a few examples how common the task of simulating wave phenomena is across many disciplines of science and engineering, and how accuracy figures into each of these application areas. Consider the following examples:

*Courant Institute of Mathematical Sciences, New York University, New York, NY 10012

†Department of Computational and Applied Mathematics, Rice University, Houston, TX 77005

‡Division of Applied Mathematics, Brown University, Providence, RI 02912

- An engineer needs to understand the time-domain response of an oscillating structure such as an accelerator cavity. Real-life measurement of the desired properties is extremely costly, if it is possible at all. Accuracy is important because wrong results may lead to wrong conclusions.
- A seismic engineer has time-domain data from a sounding using geophones and needs to model an underground structure, characterized by different wave propagation speeds. Doing so requires a solver for the 'forward problem', i.e. a code that, given data about the location of the sources and wave propagation speeds throughout the underground domain, can model the propagation of the waves. Accuracy is important because these simulations often inform potentially very costly enterprises such as drilling or mining.
- An electrical engineer wants to model the stealth properties of a new airplane involving complicated nonlinear materials. Physical prototyping is expensive, and accurate predictions of scattering properties help minimize its necessity.

In the field of time-domain wave simulation, the main competitors of the discontinuous Galerkin method include finite-difference, finite-volume and continuous finite-element methods. In a nutshell, finite-difference solvers have trouble representing complicated geometric boundaries, finite-volume methods become very difficult (and very expensive) to implement at a high order of accuracy[1], and continuous finite-element methods typically assemble large, sparse matrices, whose application to a vector is necessarily memory-bound and thus unable to make use of the massive compute bandwidth available on a GPU.

In addition, while finite-difference methods have relatively benign implementation properties on GPUs Cohen, Micikevicius [2009], we will see that the computational structure of DG methods is even better suited to GPU implementation at high accuracy because they largely avoid the wide "halo" of outside values that must be fetched in order to apply a large (high-order) stencil to three-dimensional volume data.

This chapter complements an article [Klöckner et al., 2009a] which we have recently published that, in its spirit, is probably more like the other chapters

[1]The order of accuracy refers to the power with which the error decreases as the discretization is refined–for example, if the distance between neighboring mesh points is halved, a fourth-order scheme would decrease the error by a factor of sixteen.

in this volume in that it exposes all the technicalities and tricks that have enabled us to demonstrate high-speed DG on the GPU. To avoid redundancy between [Klöckner et al., 2009a] and this chapter, we have instead chosen to focus our treatment here on easing a prospective user's entry into using our technology. While [Klöckner et al., 2009a] is very technical and not entirely suited as an introduction to the subject, in this chapter we will be applying a number of simplifications to facilitate understanding and promote ease-of-use.

2 Core Method

Discontinuous Galerkin (DG) methods for the numerical solution of partial differential equations have enjoyed considerable success because they are both flexible and robust: They allow arbitrary unstructured geometries and easy control of accuracy without compromising simulation stability. Lately, another property of DG has been growing in importance: The majority of a DG operator is applied in an element-local way, with weak penalty-based element-to-element coupling.

The resulting locality in memory access is one of the factors that enables DG to run on off-the-shelf, massively parallel graphics processors (GPUs). In addition, DG's high-order nature lets it require fewer data points per represented wavelength and hence fewer memory accesses, in exchange for higher arithmetic intensity. Both of these factors work significantly in favor of a GPU implementation of DG.

Readers wishing a deeper introduction to the numerical method are referred to the introductory textbook Hesthaven and Warburton [2007].

3 Algorithms, Implementations, and Evaluations

3.1 Background Material

3.1.1 A Precise Mathematical Problem Statement

Discontinuous Galerkin methods are most often used to solve hyperbolic systems of conservation laws in the time domain. This rather general class

of partial differential equation (PDE) can be written in the form

$$\frac{\partial q}{\partial t} + \nabla_x \cdot F(q) = f. \tag{1}$$

DG methods generally solve the *initial boundary value problems* (IBVPs) of these equations on a bounded domain Ω. This means that in addition to the PDE (1), one needs to specify the finite geometry of interest, an initial value of the solution q at an initial time T_0 (which we will assume to be zero) as well as which (potentially time-dependent) conditions prevail at the boundary $\partial\Omega$ of the domain. In addition, source terms may be present. These are represented in (1) by f.

Classes of partial differential equations more general than (1), such as parabolic and elliptic equations, can be solved using DG methods. In this chapter, we will focus on hyperbolic equations, and for the sake of exposition, on one particularly important example of these equations, the second-order wave equation in two dimensions. To emphasize the equation's grounding in reality, we will cast this equation as (the transverse-magnetic version of) the linear, isotropic, constant-coefficient Maxwell's equations in two dimensions and show the method's development by its example. The equation itself is given by

$$0 = \mu\frac{\partial H_x}{\partial t} + \frac{\partial E_z}{\partial y}, \tag{2a}$$

$$0 = \mu\frac{\partial H_y}{\partial t} - \frac{\partial E_z}{\partial y}, \tag{2b}$$

$$0 = \epsilon\frac{\partial E_z}{\partial t} - \frac{\partial H_y}{\partial x} + \frac{\partial H_x}{\partial y}. \tag{2c}$$

One easily verifies that this equation can be rewritten into the more well-known second order form of the wave equation,

$$\frac{\partial^2 E_z}{\partial t^2} = c^2 \triangle E_z$$

with $c^{-2} = \epsilon\mu$. For simplicity, we may assume $c = \epsilon = \mu = 1$. Together with an initial condition as well as perfectly electrically conducting (PEC) boundary condition

$$E_z(x,t) = 0 \quad \text{on } \partial\Omega.$$

Observe that no value is prescribed for the magnetic fields H_x, H_y, which we leave to obey *natural boundary conditions*. In terms of the second-order wave equation, PEC corresponds to a Dirichlet boundary.

3.1.2 Construction of the Method

To begin the discretization of (2), we assume that the domain Ω is polyhedral, so that it may be represented as a union $\Omega = \biguplus_{k=1}^{K} \mathsf{D}_k \subset \mathbb{R}^2$ consisting of disjoint, straight-sided, face-conforming triangles D_k.

We demonstrate the construction of the method by the example of equation (2c). We begin by multiplying (2c) with a test function ϕ and integrating over the element D_k:

$$\begin{aligned} 0 &= \int_{\mathsf{D}_k} \frac{\partial E_z}{\partial t}\phi \,\mathrm{d}V - \int_{\mathsf{D}_k} \frac{\partial H_y}{\partial x}\phi \,\mathrm{d}V + \int_{\mathsf{D}_k} \frac{\partial H_x}{\partial y}\phi \,\mathrm{d}V \\ &= \int_{\mathsf{D}_k} \frac{\partial E_z}{\partial t}\phi \,\mathrm{d}V + \int_{\mathsf{D}_k} \nabla_{(x,y)} \cdot \underbrace{(-H_y, H_x)^T}_{F=} \phi \,\mathrm{d}V. \end{aligned}$$

Observe that the vector-valued F indicated here assumes the role of the flux F in (1). Integration by parts yields

$$0 = \int_{\mathsf{D}_k} \frac{\partial E_z}{\partial t}\phi \,\mathrm{d}V - \int_{\mathsf{D}_k} (-H_y, H_x)^T \cdot \nabla_{(x,y)}\phi \,\mathrm{d}V + \int_{\partial \mathsf{D}_k} \hat{n} \cdot (-H_y, H_x)^T \phi \,\mathrm{d}S, \tag{3}$$

where $\hat{n}$ is the unit normal to $\partial\Omega$. Now a key feature of the method enters. Because no continuity is enforced on H_x and H_y between D_k and its neighbors, the value of H_x and H_y on the boundary is not uniquely determined. For now, we will record this fact by a superscript asterisk, denote these chosen values the *numerical flux*, and leave a determination of what value should be used for later.

To revert the so-called *weak form* (3) to a shape more closely resembling the original equation (2c), we integrate by parts again, obtaining the so-called *strong form*

$$\begin{aligned} 0 = \int_{\mathsf{D}_k} \frac{\partial E_z}{\partial t}\phi \,\mathrm{d}V + \int_{\mathsf{D}_k} \nabla_{(x,y)} \cdot (-H_y, H_x)^T \phi \,\mathrm{d}V \\ - \int_{\partial \mathsf{D}_k} \hat{n} \cdot (-(H_y - H_y^*), H_x - H_x^*)^T \phi \,\mathrm{d}S \end{aligned} \tag{4}$$

where we carefully observe that the boundary term obtained in the last step has stayed in place.

To determine the values of H^*, we note that in many cases a simple average across neighboring faces, i.e. $H^* := (H^+ + H^-)/2$ leads to a stable and

accurate numerical method, where H^- denotes the values on the local face. This is termed a *central flux*. We choose a more dissipative (but less noisy) *upwind flux* Mohammadian et al. [1991], given by

$$\hat{n} \cdot (F - F^*) = \begin{bmatrix} \hat{n}_y[E_z] + \alpha \hat{n}_x(\hat{n}_x[H_x] + \hat{n}_y[H_y] - [H_x]) \\ -\hat{n}_x[E_z] + \alpha \hat{n}_y(\hat{n}_x[H_x] + \hat{n}_y[H_y] - [H_y]) \\ \hat{n}_y[H_x] - \hat{n}_x[H_y] - \alpha[E_z] \end{bmatrix}. \tag{5}$$

The value to be used for $\hat{n} \cdot (-(H_y - H_y^*), H_x - H_x^*)^T$ in (4) can be read from the third entry of the right hand side of (5), and the first two entries apply to equations (2a) and (2b). We have used the common notation $[q] = q^- - q^+$ for the inter-element jumps. α is a parameter, commonly chosen as 1. Obviously, $\alpha = 0$ recovers a central flux.

We expand E, H, and ϕ into a basis of N_p Lagrange interpolation polynomials l_i spanning the space P^N of polynomials of total degree N, where the Lagrange interpolation points are purposefully chosen for numerical stability Warburton [2006]. Substituting the expansions into (4) combined with (5) yields a numerical scheme that is discrete in space, but not yet in time.

3.1.3 Implementation Aspects

To actually implement this scheme, we express (4) in matrix form. To do so, first note that in our setting, each element $\mathsf{D}_k \subset \Omega$ can be obtained by an affine map $\Psi(r, s) = A_k(r, s)^T + b_k$ from a reference element I. Now define the mass matrix

$$\mathcal{M}_{ij}^k := \int_{\mathsf{D}_k} l_i l_j \, \mathrm{d}V = |A_k| \mathcal{M} := |A_k| \int_{\mathsf{I}} l_i l_j \, \mathrm{d}V.$$

$|A_k|$ is the determinant of the matrix A_k. Also let $\mathcal{D}^{\partial\nu}$ be the matrix that realizes polynomial differentiation along the reference element's νth axis in Lagrange coefficients. Polynomial differentiation along *global* coordinates is realized as a linear combination of these local differentiation matrices, according to, e.g.,

$$\mathcal{D}^{k,\partial x} = (A_k^{-1})_{11} \mathcal{D}^{\partial 1} + (A_k^{-1})_{12} \mathcal{D}^{\partial 2}. \tag{6}$$

This allows us to express an implementation of the volume part of (4):

$$\begin{aligned} 0 = |A_k| \mathcal{M} \frac{\partial (E_z)_N}{\partial t} + |A_k| \mathcal{M} (\mathcal{D}^{k,\partial x}(-H_y)_N + \mathcal{D}^{k,\partial y}(H_x)_N) \\ - \int_{\partial \mathsf{D}_k} \hat{n} \cdot (-(H_y - H_y^*), H_x - H_x^*)^T \phi \, \mathrm{d}S. \end{aligned} \tag{7}$$

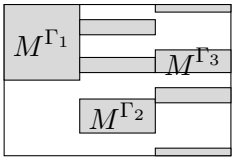
For numerical stability at increasing N, the matrices $\mathcal{M}^k$ and $\mathcal{D}^{\partial\nu}$ are computed by ways of orthogonal polynomials on the triangle [Dubiner, 1991, Koornwinder, 1975].

To implement the surface terms, define the surface mass matrix for a single face Γ of the reference triangle I:

$$M_{ij}^{\Gamma} := \int_{\Gamma \subset \partial \mathsf{I}} l_i l_j \, \mathrm{d}S.$$

Suppose we compute values of $\hat{n} \cdot (F - F^*) = \hat{n} \cdot (-(H_y - H_y^*), H_x - H_x^*)^T$ along all faces and concatenate these into one vector. Then the sum over all facial integrals may be computed through a carefully assembled matrix:

$$\int_{\partial \mathsf{D}_k} \hat{n} \cdot (F - F^*) \phi \, \mathrm{d}S =$$



$$\left(J_1 \hat{n} \cdot (F - F^*)|_{\Gamma_1} \middle| \cdots \middle| J_3 \hat{n} \cdot (F - F^*)|_{\Gamma_3} \right). \quad (8)$$

We denote this matrix $\mathcal{M}^{\partial \mathsf{I}}$ and the vector to which we are applying it $\mathbf{f}^k$. The factors J_n are the determinants of the affine maps parametrizing the faces of D_k with respect to the faces of I.

Returning to (7), we left-multiply by $|A_k|^{-1} \mathcal{M}^{-1}$ to obtain

$$0 = \frac{\partial (E_z)_N}{\partial t} + (\mathcal{D}^{k,\partial x}(-H_y)_N + \mathcal{D}^{k,\partial x}(H_x)_N) - |A_k|^{-1} \mathcal{M}^{-1} \mathcal{M}^{\partial \mathsf{I}} \mathbf{f}^k \quad (9)$$

Despite all the machinery involved, (9) is strikingly simple, consisting of three data-local element-wise matrix-vector multiplications (two differentiations, one combined face mass matrix) and a surface flux exchange term. A view of the flow of data is provided by Figure 1.

Even better, the time derivative $\frac{\partial (E_z)_N}{\partial t}$ occurs on its own, making it possible to use simple, explicit Runge-Kutta methods for integration in time.

3.2 A Minimal Implementation

After this very quick (but mostly self-contained) introduction to discontinuous Galerkin methods, we will now discuss how (9) and its analogous extension to (2a) and (2b) may be brought onto the GPU to form a solver for the 2D TM variant of Maxwell's equations.

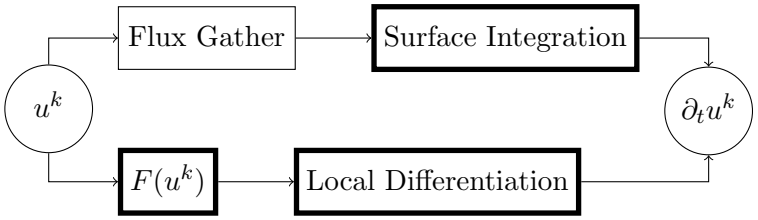


Figure 1: Decomposition of a DG operator into subtasks. Element-local operations are highlighted with a bold outline.

3.2.1 Introduction

To make the discussion both more tangible and easier to follow, we have created a simple implementation of the ideas presented here. This implementation may be downloaded from the URL `http://tiker.net/gcg-dg-code-download`. As improvements are made, the code at this address may change from time to time. The source code may also be browsed on-line at `http://tiker.net/gcg-dg-code-browse`.

We will begin by briefly discussing the construction of this package. The solver is written in Python. We feel that this allows for clearer code that, in both notation and structure, closely resembles the MATLAB codes of Hesthaven and Warburton [2007], and yet allows a simple and concise GPU implementation to be added using, in this case, PyOpenCL (PyCUDA, whose use is demonstrated in another chapter of this volume, would have been another obviously possible implementation choice). In addition, the solver is designed for clarity, not peak performance. What we mean here is that the compute kernels we show are rather simple and lack a few performance optimizations. The solver's performance is not related to its implementation language. High-performance GPU codes can easily be constructed using PyOpenCL (and PyCUDA), which is demonstrated below and in a number of other chapters of this volume. Finally, we would like to remark that the kernels as shown below are optimized for Nvidia GPUs and run well on chips ranging from the G80 to the GF100.

In discussing the solver, we focus on the performance-relevant kernels running on the GPU. There are other, significant parts of the solver that deal with preparation and administrative issues such as mesh connectivity and polynomial approximation. These parts are obviously also important to the success of the method, but they are beyond the scope of this chapter. The interested reader may find them explained more fully in the introductory

book Hesthaven and Warburton [2007]. Once we have discussed the functioning of the basic solver, we will describe any additional steps that may be taken to improve performance. Lastly, we will discuss a set of features that may be added to this rather bare implementation to make it more useful. In the next chapter, we close by showing performance numbers first for this solver, and then for our production solver, which is a more complete implementation of the ideas to follow.

3.2.2 Computing the Volume Contribution

First in our examination of implementation features is what we call the "volume kernel", which achieves element-local differentiation as described in (6) and (7). The key parts of the kernel's OpenCL C source are given in Listing 1.

One key objective of this subroutine is the multiplication of the local differentiation matrices $\mathcal{D}^{\partial\nu}$ by a large number of right hand sides, each representing degrees of freedom ("DOFs") on an element. This is a suitable point to realize that matrix-vector multiplication by a large number of vectors is equivalent to matrix-matrix multiplication by a very fat, moderately short matrix that encompasses all elemental vectors glued together. Perhaps the most immediate approach to such a problem might be to use Nvidia's CUBLAS. Unfortunately, while CUBLAS successfully covers a great many use cases, the matrix sizes in question here resulted in uninspiring performance in our experiments [Klöckner et al., 2009a]. We are thus left considering the choices for a from-scratch implementation.

In the design of computational kernels for GPUs, perhaps *the* key defining factor is the work decomposition into thread blocks (in CUDA terminology) or work groups (in OpenCL terminology). In our demonstration solver, we choose a very simple alternative, a one-to-one mapping between elements and work groups, and a one-to-one mapping between output degrees of freedom and threads (CUDA) or work items (OpenCL). This choice is simple and expedient, but it can be improved upon in a number of cases, as we will discuss in Section 3.3.3.

The next key decision is the memory layout of the data to be worked on. Again, we make a simple choice and describe possible improvements later. As was discussed in Section 3.1.2, the data we are working on consists of N_p coefficients of Lagrange interpolation polynomials for each of the K elements.

```
const int n = get_local_id (0);
const int k = get_group_id (0);

int m = n+k*BSIZE;
int id = n;

l_Hx[id] = g_Hx[m];
l_Hy[id] = g_Hy[m];
l_Ez[id] = g_Ez[m];

barrier (CLK_LOCAL_MEM_FENCE);

float dHxdr=0,dHxds=0;
float dHydr=0,dHyds=0;
float dEzdr=0,dEzds=0;

float Q;
for(m=0; m<p_Np; ++m)
{
  float4 D = read_imagef(i_DrDs, samp, (int2)(n, m));

  Q = l_Hx[m]; dHxdr += D.x*Q; dHxds += D.y*Q;
  Q = l_Hy[m]; dHydr += D.x*Q; dHyds += D.y*Q;
  Q = l_Ez[m]; dEzdr += D.x*Q; dEzds += D.y*Q;
}

const float drdx = g_vgeo[0+4*k];
const float drdy = g_vgeo[1+4*k];
const float dsdx = g_vgeo[2+4*k];
const float dsdy = g_vgeo[3+4*k];

m = n+BSIZE*k;
g_rhsHx[m] = -(drdy*dEzdr+dsdy*dEzds);
g_rhsHy[m] = (drdx*dEzdr+dsdx*dEzds);
g_rhsEz[m] = (drdx*dHydr+dsdx*dHyds
  - drdy*dHxdr-dsdy*dHxds);
```

Listing 1: OpenCL kernel implementing element-local volume contribution in the discontinuous Galerkin method, consisting of element-local polynomial differentiation.

Observe that, by their being coefficients of interpolation polynomials, they each represent the exact value of the represented solution at a point in space belonging to a certain element.

To ensure that each work group can fetch element data in the least number of memory transactions, we choose to pad each element up to $\lceil N_p \rceil_{16}$ floating point values, where the notation $\lceil x \rceil_y$ represents x rounded up to the nearest multiple of y.

With data layout and work decomposition clarified,we can now examine the implementation itself, as shown in Listing 1. After getting the element number `k` and the number of the elemental degree of freedom `n` from group and local IDs, respectively, first the elemental degrees of freedom for all three fields (H_x, H_y, E_z) are fetched into local memory for subsequent multiplication by the differentiation matrix.

As indicated above, the work being performed is effectively matrix-matrix multiplication, and therefore existing best practices suggest that also fetching the matrix into local memory might be a good idea. At least on pre-Fermi chips, this does not turn out to be true. We will take a closer look at the trade-offs involved in Section 3.3.2. For now, we simply state that the matrix is streamed into core through texture memory, and its fetch cost amortized by reusing it for not just one, but all three fields (H_x, H_y, and E_z).

Once the derivatives along each element's axes are computed by matrix multiplication, they are converted to global x and y derivatives according to (6), using separate per-element geometric factors. Finally, the results are stored, where our memory layout and work decomposition permit a fully coalesced write.

3.2.3 Computing the Surface Contribution

The second (and slightly more complicated) part of our sample implementation of the DG method is what we call the "surface kernel", which, as part of the same subroutine, achieves both the extraction of the flux expression of (5) and the surface integration of (8). The key parts of the OpenCL C source code of the kernel is shown in Listing 2 and continued in Listing 3.

We use the same one-work-group-per-element work partition as in the previous section, and obviously the memory layout of the element data is likewise

```
const int n = get_local_id (0);
const int k = get_group_id (0);

 __local float l_fluxHx [p_Nafp];
 __local float l_fluxHy [p_Nafp];
 __local float l_fluxEz [p_Nafp];

int m;

/* grab surface nodes and store flux in shared memory */
if (n < p_Nafp)
{
  /* coalesced reads (maybe) */
  m = 6*(k*p_Nafp)+n;

  const int   idM = g_surfinfo [m]; m += p_Nafp;
  int         idP = g_surfinfo [m]; m += p_Nafp;
  const float Fsc = g_surfinfo [m]; m += p_Nafp;
  const float Bsc = g_surfinfo [m]; m += p_Nafp;
  const float nx  = g_surfinfo [m]; m += p_Nafp;
  const float ny  = g_surfinfo [m];

  float dHx=0, dHy=0, dEz=0;
  dHx = 0.5f*Fsc*(    g_Hx[idP] - g_Hx[idM]);
  dHy = 0.5f*Fsc*(    g_Hy[idP] - g_Hy[idM]);
  dEz = 0.5f*Fsc*(Bsc*g_Ez[idP] - g_Ez[idM]);

  const float ndotdH = nx*dHx + ny*dHy;

  l_fluxHx [n] = -ny*dEz + dHx - ndotdH*nx;
  l_fluxHy [n] =  nx*dEz + dHy - ndotdH*ny;
  l_fluxEz [n] =  nx*dHy - ny*dHx + dEz;
}

/* make sure all element data points are cached */
 barrier (CLK_LOCAL_MEM_FENCE);
```

(Continued in Listing 3.)

Listing 2: Part 1 of the OpenCL kernel implementing inter-element surface contribution in the discontinuous Galerkin method, consisting of the calculation of the surface flux of (5).

(Continued from Figure 2.)

```
if (n < p_Np)
{
  float rhsHx = 0, rhsHy = 0, rhsEz = 0;
  int col = 0;

  /* can manually unroll to 3 because there are 3 faces */
  for (m=0;m < p_Nfaces*p_Nfp;)
  {
    float4 L = read_imagef(i_LIFT, samp, (int2)(col, n));
    ++col;

    rhsHx += L.x*l_fluxHx[m];
    rhsHy += L.x*l_fluxHy[m];
    rhsEz += L.x*l_fluxEz[m];
    ++m;

    rhsHx += L.y*l_fluxHx[m];
    rhsHy += L.y*l_fluxHy[m];
    rhsEz += L.y*l_fluxEz[m];
    ++m;

    rhsHx += L.z*l_fluxHx[m];
    rhsHy += L.z*l_fluxHy[m];
    rhsEz += L.z*l_fluxEz[m];
    ++m;
  }

  m = n+k*BSIZE;

  g_rhsHx[m] += rhsHx;
  g_rhsHy[m] += rhsHy;
  g_rhsEz[m] += rhsEz;
}
```

Listing 3: Part 2 of the OpenCL kernel implementing inter-element surface contribution in the discontinuous Galerkin method, consisting of the lifting of the surface flux contribution, as described in (8).

unchanged. It is however important to note that the kernel in question here operates on two different data formats during its lifetime. First, the output of the flux gather results in a vector of facial degrees of freedom as displayed in (8). The number of entries in this vector is $3N_{fp}$, where three is the number of faces in a triangle, and $N_{fp} = N + 1$ is the number of degrees of freedom required to discretize each face. In general $3N_{fp} \neq N_p$, where we recall that N_p is the number of volume degrees of freedom. At each of the two stages of the algorithm, we employ a design that uses one work item per degree of freedom. The number of work items required per work group is therefore $\max(N_p, N_{fp})$, and at the start of each stage of the algorithm, we need to verify whether the local thread number is less than the number of outputs required in that stage, to avoid computing (and perhaps storing) spurious extra outputs. This is necessary in both stages because either of N_p or $3N_{fp}$ may be larger.

After fixing the DOF and element indices n and k, the kernel begins by allocating $3N_{fp}$ degrees of freedom of local storage (N_{fp} per face) for each of the three fields (H_x, H_y, and E_z). This local memory serves as a temporary storage for the facial vector of (8). Next, index and geometry information is read from a surface descriptor data structure called surfinfo. For each facial degree of freedom, and hence for each work item, this data structure contains the index of the volume degree of freedom the work item processes, as well as the index of its facial neighbor (idM and idP, respectively). In addition, surfinfo contains geometry information, namely the surface unit normal of the face being integrated over (nx and ny), the surface Jacobian divided by the element's volume Jacobian (Fsc) and a boundary indicator (Bsc) used for the implementation of boundary conditions which takes values of ± 1. Based on this information, the kernel computes jump terms $[H_x]$, $[H_y]$, $[E_z]$ which are then scaled with the geometry scaling Fsc and stored as dHx, dHy, and dEz. The computation of the flux expression (5) and its temporary storage in local memory concludes the first part of the kernel, displayed in Listing 2.

The second part of the kernel, of Listing 3, is much like local polynomial differentiation as discussed in Section 3.2.2 in that it represents an element-local matrix multiplication. We have applied the same design decisions as above for simplicity, mainly based on the facial flux data already being resident in local memory. Again, data for the matrix is streamed in using the texture units, and the streamed matrix is reused for each of the three fields. One trick we were able to apply here is the three-fold unrolling of the loop. This is valid because we know that the combined face mass

matrix $\mathcal{M}^{\partial \mathsf{l}}$ covers three faces and hence must have a column count that is divisible by three. Naturally, the same applies to the lifting matrix $\mathcal{L} := \mathcal{M}^{-1}\mathcal{M}^{\partial \mathsf{l}}$. The result of this matrix-vector product is then added to global destination arrays in which the volume contribution to the right-hand side $\partial(H_x, H_y, E_z)^T/\partial t$ is already stored, completing the computation the entire right-hand side of (9).

This concludes our description of the basic kernel implementing the computation of the ODE right-hand side for nodal discontinuous Galerkin methods. What is missing to complete the implementation of the method is a simple Runge-Kutta time integrator, which we have implemented using PyOpenCL's built-in array operations. We now proceed to discuss a number of ways in which performance of these basic kernels can be improved.

3.3 Improving Performance

3.3.1 Avoiding Padding Waste: Data Aggregation

In the above codes, each element is represented by $\lceil N_p \rceil_{16}$ floating point values for alignment and fetch efficiency reasons. Especially in two dimensions, or in three dimensions for elements of relatively small polynomial order N, this extra padding can be rather inefficient–not just in terms of GPU memory use, but especially also in computational resources. All of our kernels adopt a one-work-item-per-output design, and hence wasted memory has a one-to-one correspondence to wasted computational power. This is all the more true once one realizes that Nvidia hardware schedules computations in units of 32-wide *warps*, such that a rounding to 16 has a chance of 50 per cent of leaving the trailing half-warp of the computation unused. An obvious remedy for this problem is the aggregation of multiple elements into a single unit. This aggregation represents a trade-off against the work partition flexibility of all kernels operating on the data, and should therefore be chosen as small as possible, while still minimizing waste.

In [Klöckner et al., 2009a], we pursue a compromise strategy, where we choose a granularity that combines enough elements so that less than a given percentage (e.g. 10) of waste occurs. All further occurring granularities are then required to operate integer multiples of this smallest possible granularity. To differentiate this granularity from the generally larger work group size (or "block size" in CUDA terminology), we have introduced the term *microblock* to denote it.

3.3.2 Which Memory for What?

On-chip memory in a GPU setting is always somewhat scarce, and we foresee that this will remain so for the foreseeable future. As already discussed in Section 3.2, it is far from clear which portions of the GPU's on-chip memory should be used for what data. In discussing this question, we focus on the element-local matrix-vector polynomial differentiation as this asymptotically (and practically) dominates runtime as the polynomial degree N increases.

In [Klöckner et al., 2009a], we discuss two possible strategies for element-local matrix multiplication, the first of which proceeds by loading the matrix into local memory, and the second of which loads field data into local memory, as we have done above.

To allow the flexibility of being able to choose which strategy to use for each each of the two element-local matrix products (differentiation and lift), the surface kernel of Section 3.2.3 may have to be split into its two constituent parts, necessitating an extra store-load cycle. We find that this disadvantage is entirely compensated by the advantage of being able to use a more immediately suitable work partition for each part.

The enumeration of the two strategies begs an immediate question–why is the strategy of loading *both* quantities into local memory not considered? The reason for this lies rooted in a number of important practicalities. While generic matrix multiplication routines are free to optimize for the case of large, square matrices, the matrices we are faced with are small. A generic blocking strategy would therefore leave us with many inefficient corner cases which would come to dominate our run time. In addition, in the case of three-dimensional geometry, the three differentiation matrices of size $N_p \times N_p$ exhaust the local memory on Nvidia hardware even for moderate N.

We find that, at low-to-moderate N and in general in two dimensions, we can derive a gain of about 20 per cent by making the matrix-in-local strategy available in addition to the field-in-local strategy shown above. Nonetheless, the latter strategy has fewer size restrictions than the former, is thus more generally applicable, and it successfully uses register and texture memory to avoid many redundant matrix fetches. This justifies our choice of the strategy in our demonstration code. In these codes, we amortized matrix fetch costs by operating on three fields at once. Note that even in a scalar (i.e. single-field) case, such amortization is possible, simply by operating on multiple elements within the same work item.

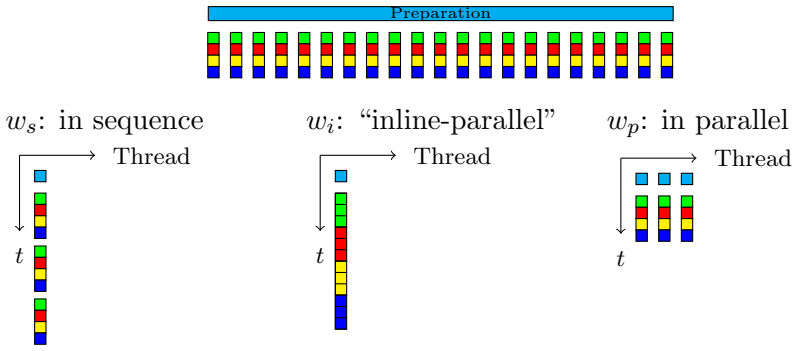


Figure 2: Possibilities for partitioning a large number independent work units, potentially requiring some preparation, into work groups. Each work unit consists of multiple stages, symbolized by different colors. Preparation is shown in a cyan color. An example of this would be multiple independent matrix-vector multiplications, each consisting of individual multiply-add cycles.

3.3.3 Rethinking the Work Partition

Because of the inherent advantages of using one work item per output value, the question of work partitioning into work groups and work items on GPU hardware is never far removed from that of memory use and data layout, as discussed above. The work partition chosen in the demonstration code was purposefully simple–one work item per degree of freedom, one work group per element. Moving beyond that, while taking into account the lessons of Section 3.3.1, one naturally arrives at a partition of one microblock per work group. But even this can be further generalized, as one may work on more than one microblock in each work group. We assume here that the work on each work item is independent, but may depend on some preparation, such as fetching matrix data into on-chip memory. This opens up a number of possible avenues: One may...

- process microblocks in parallel, adding more work items to each work group, to achieve better usage of individual compute units.
- process microblocks in sequence, leaving the number of work items unchanged, but doing more work in each work item, thus amortizing preparation work.
- process multiple work items along with each other, reusing auxiliary data (such as matrices) that is already present in machine registers. We term this usage "*in-line parallel*".

- use any combination of the above.

Figure 2 illustrates these possibilities.

If a strategy is chosen that exploits the parallel processing of multiple microblocks in one work group (the first option) above, subtle questions of thread ordering arise that may influence the number of local memory bank conflicts. In [Klöckner et al., 2009a], we discuss one such question in more detail.

Note that all combinations of parallel, sequential, and in-line parallel do the same amount of work, and should, in theory, require similar time to complete. In practice, this is not the case. This begs the question of how to decide between the numerous different possibilities. It is of course possible to explore manually which combination yields the best performance, but this is tedious, error-prone, and needs to be repeated for nearly every change to the hardware on which the code is run. This clearly undesirable, but a potential solution is described in the next section.

3.3.4 Using Run-Time Code Generation

In [Klöckner et al., 2009b], we discuss the numerous benefits of being able to generate computational code immediately before it is used, i.e. being able to perform C-level *run-time code generation*. We are delighted that OpenCL has this capability built into its specification. We do note however that the feature can be retrofitted onto CUDA through the use of the *PyCUDA* Python package. In our DG demonstration code, we already make simple use of this facility, by using string substitution on the source code of our kernels to make certain problem size parameters known to the compiler at compilation time. This helps decrease register pressure and allows the compiler to use a number of optimizations such as static loop unrolling for loops whose trip counts are now known.

But this is far from the only benefit. Another immediate advantage is the ability to perform automated tuning to answer questions such as the one raised at the end of the last section, where individual kernels can be generated to cover any number of code variants to be tried. Once this has been accomplished, implementing automated tuning can be as simple as looping over all variants and comparing timing data for each. For larger search spaces, a more sophisticated strategy might be desirable. This entire topic is discussed in much greater detail in [Klöckner et al., 2009b].

3.3.5 Further Tuning Opportunities

In the demonstration code, some inefficiency lies buried in the way the surface fluxes are evaluated. Because the data required by the surface evaluation grows as $O(N^{d-1})$, whereas the volume data's size grows as $O(N^d)$, this is mainly felt at low-to-moderate N, which are particularly relevant for practical purposes.

First, the index data loaded into idM and idP has significant redundancy and can easily be compressed by breaking it down into element offsets added to one particular entry from a list of subindex lists. This list of subindex lists is comparatively small and has better odds of being able to reside in on-chip memory.

Second, data for faces lying opposite to each other is fetched twice–once for each side–in our current implementation. Through a blocking strategy (which, unfortunately, introduces significant complexity) these redundant fetches can be avoided. The strategy is discussed in detail in [Klöckner et al., 2009a].

The last opportunity for tuning we will discuss in this setting is the use of multiple GPUs through MPI or Pthreads. Since only facial data for flux computation needs to be exchanged between GPUs, such a code is cheap in communications bandwidth and relatively easy to implement. We will now turn our discussion here from opportunities for speed increase to ways of making the technology more useful and more broadly applicable.

3.4 Adding Generality

GPU-DG as demonstrated in the demonstration code in this chapter can be extended in a number of ways to address a larger number of application problems.

Three Dimensions Perhaps the most gentle, but also the most immediately necessary generalization is the use of three-dimensional discretizations. The main complexity here lies in adapting the set-up code that generates matrices and computes mesh connectivity. The GPU kernels only require mild modification, although a number of complexity trade-offs change, requiring different tuning decisions. It is further helpful to generate general, n-dimensional code from a single source through run-time code generation, to reduce the amount of

code that needs to be maintained and debugged.

Double Precision We have found that for most engineering problems, single precision calculations are more than sufficient. We do however acknowledge that some problems *do* require double precision, and our methods can be easily adapted to accommodate it.

General Boundary Conditions Our demonstration code only provided facilities for a single (Dirichlet) boundary condition. In nearly all practical problems, multiple boundary conditions (BCs) are needed, ranging from Neumann to absorbing BCs to even more complicated conditions arising in fluid dynamics.

General Linear Systems of Conservation Laws In addition to more general BCs, one obviously often wants to solve more general PDEs than the 2D wave/Maxwell equation discussed here–this might include Maxwell's equations in 3D or the equations of aeroacoustics. Again, making these adaptations to the demonstration code is relatively straightforward and mainly entails implementing a different local differentiation operator along with a new flux expression.

Nonlinear Systems of Conservation Laws Once general *linear* systems are treated by GPU-DG, it is, conceptually, not a very big step to also treat *non-linear* systems of conservation laws, such as Euler's equations of gas dynamics or the even more complicated Navier-Stokes equations, potentially along with various turbulence models. The good news is that the methods presented so far again generalize seamlessly and work well for simple problems.

However, due to the subtle subject matter, a number of refinements of the method may be required to successfully treat real application problems.

The first issue revolves around the evaluation of non-linear terms on a nodal grid and the aliasing error thus introduced into the method. One possibility of addressing this is filtering, which is easily implemented, but impacts accuracy. Another is over-integration using quadrature and cubature rules to more accurately approximate the integrals involved in the method. A more detailed discussion of these subjects is beyond the scope of this chapter and can be found in [Hesthaven and Warburton, 2007].

Shocks, i.e. the spontaneous emergence of very steep gradients in

the solution, are another complication that only arises in nonlinear problems. Some initial ideas on using GPU-DG in conjunction with shock-laden flow computations are available in [Klöckner et al., 2011].

Curved Geometries While finite element methods already offer much greater flexibility in the approximation of geometry than solvers on structured grids, the demonstration solver shown here is restricted to geometries that consist of straight surfaces. A cost-efficient way to extend this solver to curved geometries is shown in [Warburton, 2008].

Local Time Integration Lastly, as the solvers described in this chapter all employ explicit marching in time, time step restrictions may become an issue if the mesh involves very small elements. [Klöckner, 2010, Chapter 8] describes a number of time stepping schemes that can help overcome this problem.

As we have seen, a simple solver employing discontinuous Galerkin methods on the GPU can be written without much effort. On the other hand, far more effort can be spent on performance tuning and adaptation to more general problems. A free solver that implements nearly all of the improvements described here is available at `http://mathema.tician.de/software/hedge` under the GNU Public License.

4 Final Evaluation

In the present chapter, we have shown that, even with limited effort, large performance gains are realizable for discontinuous Galerkin methods using explicit time integration. Figure 3 shows a live snapshot of the simulation of a wave propagation problem on a moderately complex domain as shown during run time by the solver if the `mayavi2`[2] visualization package is installed.

Figure 4(a) shows the performance of the demonstration solver developed here and compares it to the performance of hedge, our (freely available) production solver. Note that these graphs should not be compared directly, as one shows the result of a 2D simulation, while the other portrays the performance of a three-dimensional computation. With some care, a few observations can be made however.

[2] `http://code.enthought.com/projects/mayavi/`

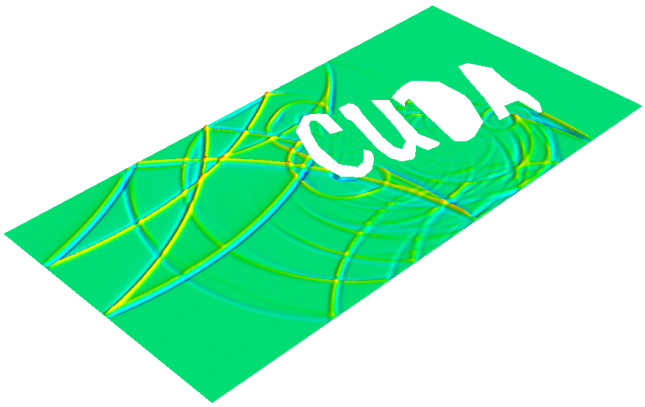
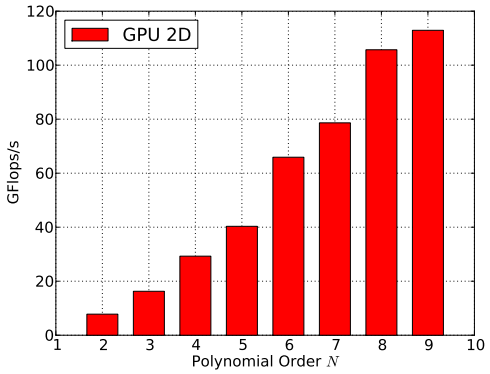
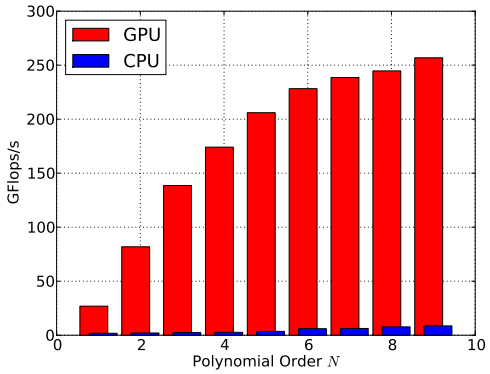


Figure 3: Screen shot of the demonstration solver showing a live snapshot of the simulation of a wave propagation problem on a moderately complex domain.



(a) GFlops/s plotted vs. polynomial degree for the 2D demonstration solver described in this chapter.



(b) GFlops/s plotted vs. polynomial degree for the 3D DG solver hedge.

Figure 4: Performance figures for the demonstration solver (2D) and the full-featured (3D) solver 'hedge', both executing solvers for the Maxwell problem on an Nvidia GTX 295 in single precision.

It is possible to observe that, as the element-local matrices grow with the third power of the degree in three dimensions (as opposed to the second power in 2D), they constitute a larger part of the computation and contribute many more floating point operations, leading to higher performance. For the same reason, performance increases as the polynomial degree N increases.

Many of the tuning ideas described in Section 3.3 (such as microblocking and matrix-in-local) are designed to help performance in the case of lower N, and hence, smaller matrices. Without comparing absolute numbers, we observe that the initial performance increase at low N is much faster in the production solver than in the demonstration solver. We attribute this to the implementation of these extra strategies, that lead to markedly better performance at moderate N. In [Klöckner et al., 2009a], we also briefly study the individual and combined effects of a few of these performance optimizations.

As a final observation, we would like to remark that the performance obtained here is very close to to the performance obtained in a C-based OpenCL solver written for the same problem–the use of Python as an implementation language does not hamper the speed of the solver at all. In particular, this facilitates a very logical splitting of computational software into performance-critical, low-level parts written in OpenCL C, and performance-uncritical set-up and administrative parts written in Python.

5 Future Directions

We have shown that, using our strategies, high-order DG methods can reach double-digit percentages of published theoretical peak performance values for the hardware under consideration. This computational speed translates directly into an increase of the size of the problem that can be reasonably treated using these methods. A single compute device can now do work that previously required a roomful of computing hardware–even using the simplistic implementation demonstrated here.

It is our stated goal to further broaden the usefulness of the method through continued investigation of the treatment of nonlinear problems, improved time integration characteristics, and coupling to other discretizations to optimally exploit the characteristics of each. GPUs present a rare opportunity,

and it is fortuitous that a method like DG, which is known for highly accurate solutions, can benefit so tremendously from this computational advance.

Acknowledgments

We would like to thank Xueyu Zhu, who performed the initial Python port of the Matlab codes from which the demonstration solver is derived.

TW acknowledges the support of AFOSR under grant number FA9550-05-1-0473 and of the National Science Foundation under grant number DMS 0810187. JSH was partially supported by AFOSR, NSF, and DOE. AK's research was partially funded by AFOSR under contract number FA9550-07-1-0422, through the AFOSR/NSSEFF Program Award FA9550-10-1-0180 and also under contract DEFG0288ER25053 by the Department of Energy. The opinions expressed are the views of the authors. They do not necessarily reflect the official position of the funding agencies.

References

Jonathan Cohen. OpenCurrent. URL `http://code.google.com/p/opencurrent/`.

Moshe Dubiner. Spectral methods on triangles and other domains. *Journal of Scientific Computing*, 6:345–390, December 1991. doi: 10.1007/BF01060030.

J. S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Springer, first edition, November 2007. ISBN 0387720650.

A. Klöckner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *J. Comp. Phys.*, 228: 7863–7882, 2009a. doi: 10.1016/j.jcp.2009.06.041.

Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda: Gpu run-time code generation for high-performance computing. Technical Report 2009-40, Scientific Computing Group, Brown University, Providence, RI, USA, November 2009b. URL `http://www.dam.brown.edu/scicomp/reports/2009-40/`. submitted.

A. Klöckner, T. Warburton, and J. S. Hesthaven. Viscous Shock Capturing in a Time-Explicit Discontinuous Galerkin Method. *Mathematical Modelling of Natural Phenomena*, 2011. to appear.

Andreas Klöckner. *High-Performance High-Order Simulation of Wave and Plasma Phenomena*. PhD thesis, Brown University, 2010.

T. Koornwinder. Two-variable analogues of the classical orthogonal polynomials. *Theory and Applications of Special Functions*, pages 435–495, 1975.

P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

Alireza H. Mohammadian, Vijaya Shankar, and William F. Hall. Computation of electromagnetic scattering and radiation using a time-domain finite-volume discretization procedure. *Computer Physics Communications*, 68(1-3):175 – 196, 1991. doi: 10.1016/0010-4655(91)90199-U.

T. Warburton. An explicit construction of interpolation nodes on the simplex. *J. Eng. Math.*, 56:247–262, 2006.

Timothy Warburton. Accelerating the Discontinuous Galerkin Time-Domain Method. In *Proceedings of the Workshop "Non-standard Finite Element Methods"*, number 36/2008 in Oberwolfach Reports. Mathematisches Forschungsinstitut Oberwolfach, 2008.