Discontinuous Galerkin, Python, and GPUs: the 'hedge' solver package

Andreas Klöckner

Courant Institute of Mathematical Sciences
New York University

May 22, 2011

Thanks

- Jan Hesthaven (Brown)
- Tim Warburton (Rice)
- Leslie Greengard (NYU)
- hedge, PyOpenCL, PyCUDA contributors
- Nvidia Corporation

Outline

1 Introduction

2 Hedge How-To

3 Under the Hood

4 Conclusions

Outline

1 Introduction
- The Method
- DG to Code

2 Hedge How-To

3 Under the Hood

4 Conclusions

Outline

1 Introduction
- The Method
- DG to Code

2 Hedge How-To

3 Under the Hood

4 Conclusions

Discontinuous Galerkin Method

Let $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$.

Discontinuous Galerkin Method

Let $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$.

Goal

Solve a *conservation law* on Ω: $u_t + \nabla \cdot F(u) = 0$

Discontinuous Galerkin Method

Let $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$.

Goal

Solve a *conservation law* on Ω: $u_t + \nabla \cdot F(u) = 0$

Example

Maxwell's Equations: EM field: $E(x,t)$, $H(x,t)$ on Ω governed by

$$\partial_t E - \frac{1}{\varepsilon}\nabla \times H = -\frac{j}{\varepsilon}, \qquad \partial_t H + \frac{1}{\mu}\nabla \times E = 0,$$

$$\nabla \cdot E = \frac{\rho}{\varepsilon}, \qquad \nabla \cdot H = 0.$$

Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

$$
\begin{aligned}
0 &= \int_{\mathsf{D}_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, \mathrm{d}x \\
&= \int_{\mathsf{D}_k} u_t \varphi - F(u) \cdot \nabla \varphi \, \mathrm{d}x + \int_{\partial \mathsf{D}_k} (\hat{n} \cdot F)^* \varphi \, \mathrm{d}S_x,
\end{aligned}
$$

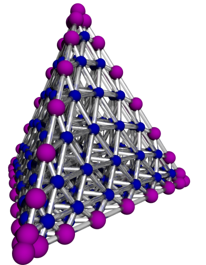
Subsitute in basis functions, introduce elementwise stiffness, mass, and surface mass matrices S, M, M_A:

$$
\partial_t u^k = -\sum_{\nu} D^{\partial_\nu,k}[F(u^k)] + L^k[\hat{n} \cdot F - (\hat{n} \cdot F)^*]|_{A \subset \partial \mathsf{D}_k}.
$$

For straight-sided simplicial elements:
Reduce D^{∂_ν} and L to reference matrices.

Nodal Field Representation

Computational representation of approximate fields:
Values at nodal points $\xi_{k\nu}$ on each tetrahedron D_k.
[Warp & Blend Lagrange Nodes: Warburton 06]



Node locations in 8th order unit tetrahedron

- (+) Smaller dependency footprint of surface data ($O(N^d) \to O(N^{d-1})$)
- (−) Modal operations more expensive (e.g. inner product, differentiation, filtering)

Outline

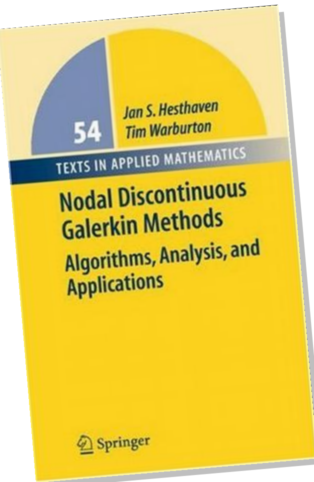
1 Introduction
- The Method
- DG to Code

2 Hedge How-To

3 Under the Hood

4 Conclusions

Step 1: Matlab DG



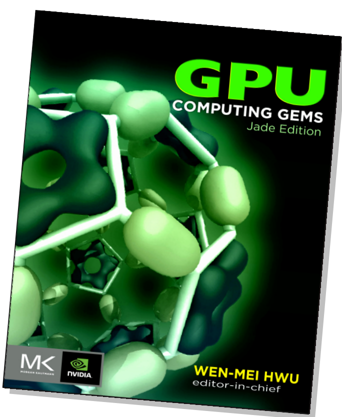
- $\sim$ 1000 lines of Matlab
- Documented by textbook
- Focus: Simplicity, exposition
- CPU, 1,2,(3)D

St

```
% evaluate fluxes
ndotdE = nx.*dEx+ny.*dEy;
fluxEx = -ny.*dHz + alpha*(ndotdE.*nx-dEx);
fluxEy = nx.*dHz + alpha*(ndotdE.*ny-dEy);
fluxEx(mapB) = -ny(mapB).*dHz(mapB) + ...
  (ndotdE(mapB).*nx(mapB)-dEx(mapB));
fluxEy(mapB) = nx(mapB).*dHz(mapB) + ...
  (ndotdE(mapB).*ny(mapB)-dEy(mapB));
fluxHz = nx.*dEy - ny.*dEx - alpha*dHz;

% local derivatives of fields
[dHzdx,dHzdy] = Grad2D(Hz);
[dExdx,dExdy] = Grad2D(Ex);
[dEydx,dEydy] = Grad2D(Ey);

% compute right hand sides of the PDE's
rhsEx = dHzdy       + LIFT*(Fscale.*fluxEx)/2.0 ;
rhsEy = -dHzdx      + LIFT*(Fscale.*fluxEy)/2.0;
rhsHz = -dEydx+dExdy + LIFT*(Fscale.*fluxHz)/2.0;
```

n

Step 2: Expository Python+GPU DG "Pydgeon"



- $\sim$ 1500 lines of Python, OpenCL C
- Very similar to Matlab code
- Documented by article in GCG Vol. 2
- Focus: Simplicity, exposition, perf.
- CPU/GPU, 2D
- Live visualization
- Maxwell only (so far) with PEC BC

St

```
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;

barrier(CLK_LOCAL_MEM_FENCE);
// ...
for (m=0;m < p_Nfaces*p_Nfp; ++m)
{
  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];
}
```

penCL C

e

n, perf.

PEC BC

Step 3: Production Python+GPU DG "hedge"

- $\sim$ 20,000 lines of Python
- Dedicated documentation, wiki, mailing list
- Focus: features, performance, ease of use
- Self-tuning CPU/GPU+MPI, nD
- Nonlinear problems: quadrature, shock capture
- General PDEs (nD Wave, nD CNS, nD Maxwell's, ...), general BCs (periodic, ...)
- SP/DP+Complex
- Multi-rate time integration

Step 3: Production Python+GPU DG "hedge"

```
flux = - join_fields(
    dot(v.avg, normal)
      - 0.5*(u.int-u.ext),

    u.avg * normal
      - 0.5*(normal
        * dot(normal, v.int-v.ext)))
```

```
op_template = InverseMassOperator()(
    join_fields (
        -dot(make_stiffness_t(d), v),
        -(make_stiffness_t(d)*u)
        )
  - (flux_op(w) + flux_op(
        BoundaryPair(
          w, dir_bc, TAG_ALL))))
```

ure

xwell's,

Outline

1 Introduction

2 Hedge How-To

3 Under the Hood

4 Conclusions

Anatomy of a Driver File

1. Generate mesh
   - MeshPy: Direct interfaces to Triangle, TetGen, Gmsh
2. Set up discretization
   - Given: exec. context, mesh, polynomial order, quad. order
3. Create optemplate
   - Given: parameters (e.g. dimension, boundary tags)
4. Compile optemplate
   - Given: optemplate, discretization. Yields: function
5. Set up time integration
   - Available: Many RK, IMEX, SSP, Adaptive, DUMKA
6. Main loop, including
   - Time stepper calls (calls compiled op. func)
   - Visualization

Operator Representation in Hedge

- Named Variables
  - `Variable`, `ScalarParameter`, `make_vector_field(name, dim)`
- Arithmetic, custom functions, conditionals
  - +, −, ∗, /, `IfPositive(cond, then, else)`
- Volume bilinear forms
  - `make_nabla()(x)`, `MassOperator()(x)`, `InverseMassOperator()(x)`, ...
- Surface bilinear forms ("fluxes") (next slide)
- Interpolation to quadrature points
  - `QuadratureGridUpsampler(tag)(x)`, ...
- Geometry information (normals, jacobians, ...)
- Common subexpression tags
  - `make_common_subexpression(x, name)`

Operator Representation in Hedge

- Named Variables
  - `Variable`, `ScalarParameter`, `make_vector_field(name, dim)`
- Arithmetic, custom functions, conditionals
  - +, −, ∗, /, `IfPositive(cond, then, else)`
- Volume bilinear forms
  - `make_nabla()(x)`, `MassOperator()(x)`, `InverseMassOperator()(x)`, . . .
- Surface bilinear forms ("fluxes") (next slide)
- Interpolation to quadrature points
  - `QuadratureGridUpsampler(tag)(x)`
- G
- C

Entire optemplate expressed in terms of scalar fields.

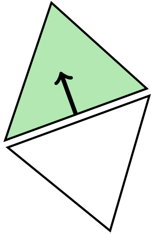
Vector of scalar expressions for PDE systems.

Flux Description

- *Different* (numbered!) scalar placeholders
  - `FluxVectorPlaceholder(dim)`
  - `.int`, `.ext`, `.avg`.
  - Refers to arguments from optemplate (outer level) by number
- Yields 'flux operator' which is usable in optemplate.
- A flux operator can be applied to
  - volume terms
  - `BoundaryPair(tag, vol_term, bdry_term)`

  in optemplate (outer level).

  Typically: Same flux for boundary and volume.



Specification Example: Wave equation

```
d = dimensions

w = FluxVectorPlaceholder(1+d)
u = w[0]
v = w[1:]
normal = make_normal(d)

flux = - join_fields (
    dot(v.avg, normal)
      - 0.5*(u.int -u.ext),

    u.avg * normal
      - 0.5*(normal
        * dot(normal, v.int -v.ext)))

w = make_vector_field("w", d+1)
u = w[0]
v = w[1:]
```

```
dir_u  = BoundarizeOperator(TAG_ALL)(u)
dir_v  = BoundarizeOperator(TAG_ALL)(v)
dir_bc = join_fields (-dir_u, dir_v)

# operator assembly
flux_op = get_flux_operator(flux)

op_template = InverseMassOperator()(
    join_fields (
        -dot(make_stiffness_t(d), v),
        -(make_stiffness_t(d)*u)
        )
    - (flux_op(w) + flux_op(
        BoundaryPair(
          w, dir_bc, TAG_ALL))))
```

$$u^* = \hat{n} \cdot \{v\} - \frac{1}{2}(u^- - u^+),$$

$$v^* = \hat{n}\left(\{u\} - \frac{\hat{n}}{2} \cdot (v^- - v^+)\right)$$

$$\partial_t u + \nabla_x \cdot v = 0,$$

$$\partial_t v + \nabla_x u = 0$$

```
flux = - join_fields (
    dot(v.avg, normal)
      - 0.5*(u.int -u.ext),

    u.avg * normal
      - 0.5*(normal
        * dot(normal, v.int -v.ext)))

w = make_vector_field("w", d+1)
u = w[0]
v = w[1:]
```

```
_bc = join_fields (-dir_u, dir_v)

# operator assembly
flux_op = get_flux_operator(flux)

op_template = InverseMassOperator()(
   join_fields (
      -dot( make_stiffness_t (d), v),
      -( make_stiffness_t (d)*u)
      )
   - (flux_op(w) + flux_op(
      BoundaryPair(
         w, dir_bc , TAG_ALL))))
```

Sp… atic

$$u^* = \hat{n} \cdot \{v\} - \frac{1}{2}(u^- - u^+),$$

$$v^* = \hat{n}\left(\{u\} - \frac{\hat{n}}{2} \cdot (v^- - v^+)\right)$$

$$\partial_t u + \nabla_x \cdot v = 0,$$

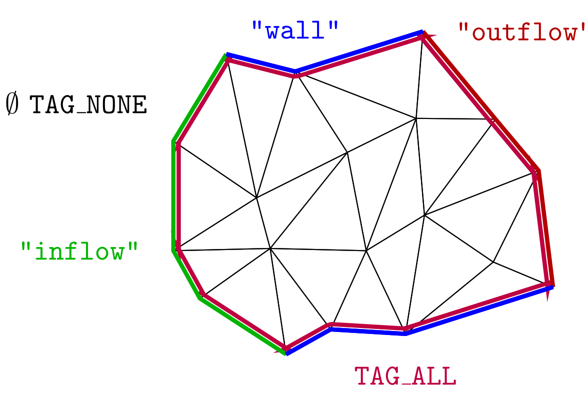
$$\partial_t v + \nabla_x u = 0$$

```
normal = make_normal(d)

flux = - join_fields (
    dot(v.avg, normal)
      - 0.5*(u.int -u.ext),

    u.avg * normal
      - 0.5*(normal
        * dot(normal, v.int -v.ext)))

w = make_vector_field("w", d+
u = w[0]
v = w[1:]
```

```
_u =                                         )
_v = BoundarizeOperator(TAG_ALL)(v)
_bc = join_fields (-dir_u, dir_v)

# operator assembly
flux_op = get_flux_operator (flux)

op_template = InverseMassOperator()(
    join_fields (
        -dot( make_stiffness_t (d), v),
        -( make_stiffness_t (d)*u)
        )
    - (flux_op(w) + flux_op(
```

Goal: Want to match or beat hand-written code for operators like this.

Boundary Tags



Boundary tags: Symbolic names for sets of boundary faces

Can be anything, usually strings.

`hedge.mesh.{TAG_NONE,TAG_ALL}` always defined

Outline

1 Introduction

2 Hedge How-To

3 Under the Hood
- RTCG Infrastructure
- Optemplate Pipeline

4 Conclusions

Outline

1 Introduction

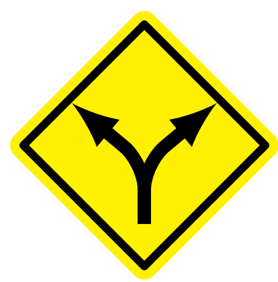
2 Hedge How-To

3 Under the Hood
- RTCG Infrastructure
- Optemplate Pipeline

4 Conclusions

How are High-Performance Codes constructed?

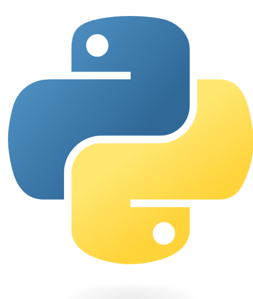
- "Traditional" Construction of High-Performance Codes:
  - C/C++/Fortran
  - Libraries
- "Alternative" Construction of High-Performance Codes:
  - Scripting for 'brains'
  - Generated code on GPUs for 'inner loops'
- Play to the strengths of each programming environment.



Scripting: Python

One example of a scripting language: Python

- Mature
- Large and active community
- Emphasizes readability
- Written in widely-portable C
- A 'multi-paradigm' language
- Rich ecosystem of sci-comp related software



What is OpenCL?

OpenCL (Open Computing Language) is an open, royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. [OpenCL 1.1 spec]

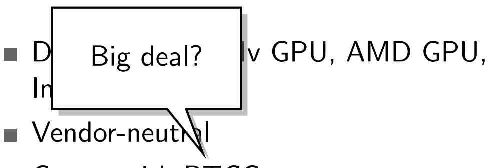
- Device-neutral (Nv GPU, AMD GPU, Intel/AMD CPU)
- Vendor-neutral
- Comes with RTCG

Defines:

- Host-side programming interface (library)
- Device-side programming language (!)

What is OpenCL?

OpenCL (Open Computing Language) is an open, royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. [OpenCL 1.1 spec]



- D lv GPU, AMD GPU, In
- Vendor-neut l
- Comes with RTCG

Defines:

- Host-side programming interface (library)
- Device-side programming language (!)

Machine-generated Code

Why machine-generate code?

- Automated Tuning
  (cf. ATLAS, FFTW)
- Data types
- Specialize code for given problem
- Constants faster than variables
  ($\rightarrow$ register pressure)
- Loop Unrolling

Why do Scripting for GPUs?

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

  $\rightarrow$ complement each other
- CPU: largely restricted to control tasks (~1000/sec)
  - Scripting fast enough
- Python + OpenCL = **PyOpenCL**
- Python + CUDA = **PyCUDA**

PyOpenCL, PyCUDA: Vital Information

- `http://mathema.tician.de/software/pyopencl` (or `/pycuda`)
- Complete documentation
- MIT License
- Arrays, Elementwise op., Reduction, Scan
- Compiler Cache, RAII, Error checking
- Require: numpy, Python 2.4+ (Win/OS X/Linux)
- Community: mailing list, wiki, add-on packages (FFT, scikits.cuda, . . .)

A taste of PyOpenCL

```
 1  import pyopencl as cl, numpy
 2
 3  a = numpy.random.rand(256**3).astype(numpy.float32)
 4
 5  ctx = cl.create_some_context()
 6  queue = cl.CommandQueue(ctx)
 7
 8  a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
 9  cl.enqueue_write_buffer(queue, a_dev, a)
10
11  prg = cl.Program(ctx, """
12      __kernel void twice(__global float *a)
13      { a[get_local_id(0)+ get_local_size(0)*get_group_id(0)] *= 2; }
14      """).build()
15
16  prg.twice(queue, a.shape, (256,), a_dev)
```

A taste of PyOpenCL

```
1  import pyopencl as cl, numpy
2
3  a = numpy.random.rand(256**3).astype(numpy.float32)
4
5  ctx = cl.create_some_context()
6  queue = cl.CommandQueue(ctx)
7
8  a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
9  cl.enqueue_write_buffer(queue, a_dev, a)
10
11 prg = cl.Program(ctx, """
12     __kernel void twice(__global float *a)
13     { a[get_local_id(0)+ get_local_size(0)*get_group_id(0)] *= 2; }
14     """).build()
15
16 prg.twice(queue, a.shape, (256,), a_dev)
```

Compute kernel

PyOpenCL: Code Generation using Templates

```
<%def name="chunk_for_with_tail(loop_var, start , chunk_size , end)">
    uint ${loop_var} = ${start};
    while (${loop_var} + ${chunk_size} < ${end})
    {
      ${ caller .body( is_tail =False)}
      ${loop_var} += ${chunk_size};
    }
    ${ caller .body( is_tail =True)}
</%def>

<%self: chunk_for_with_tail loop_var="isource_base" start ="0"
chunk_size="128" end="nsource" args="is_tail">
    % if is_tail :
        if ( isource_load < nsource)
    % endif
    % for i in range(dimensions):
        s_l${i}[ lid ] = s_g${i}[ isource_base + lid ];
    % endfor
</%self: chunk_for_with_tail >
```

Outline

1 Introduction

2 Hedge How-To

3 Under the Hood
- RTCG Infrastructure
- Optemplate Pipeline

4 Conclusions

Tree Representations

Advantages:

- Simple
- Expressions naturally map to trees
  - Easy to build for user
- Good for 'peephole' rewriting
  - Computer Algebra uses trees

Problems:

- Redundant Subexpressions
- Many temporaries
- Not good for 'global' rewriting

Tree Representations

Advantages:

- Simple
- Expressions naturally map to trees
  - Easy to build for user
- Good for 'peephole' rewriting
  - Computer Algebra uses trees

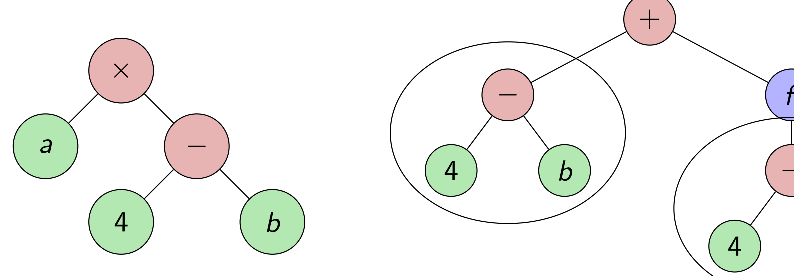
Problems

- Redun
- Many
- Not g

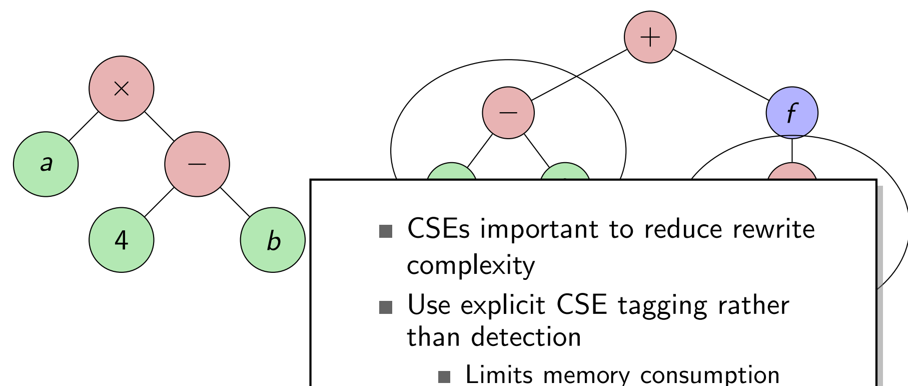
Why not 'direct execution'?

- Prevents many optimizations
- Prevents information discovery
- Requested op. is a data structure
  - Can be built programmatically

What about Common Subexpressions?

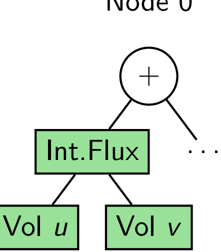


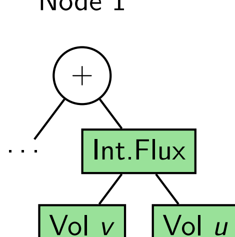
What about Common Subexpressions?



- CSEs important to reduce rewrite complexity
- Use explicit CSE tagging rather than detection
  - Limits memory consumption
- How to realize reuse? later

Communication Insertion





Communication Insertion

Node 0

- $+$
  - Int.Flux
    - Vol u
    - Vol v
  - Bdry.Flux
    - Fin. Recv.
      - Post Send, Recv.
        - Bdry u
        - Bdry v

Node 1

- $+$
  - Bdry.Flux
    - Fin. Recv.
      - Post Send, Recv.
        - Bdry v
        - Bdry u
  - Int.Flux
    - Vol v
    - Vol u

Communication Insertion

Node 0

Node 1

+

Int.Flux

Bdry.Flux

Vol u

Vol v

Fin. Recv.

Post Send, Recv.

Bdry u

Bdry v

+

Bdry.Flux

Int.Flux

Fin. Recv.

Vol v

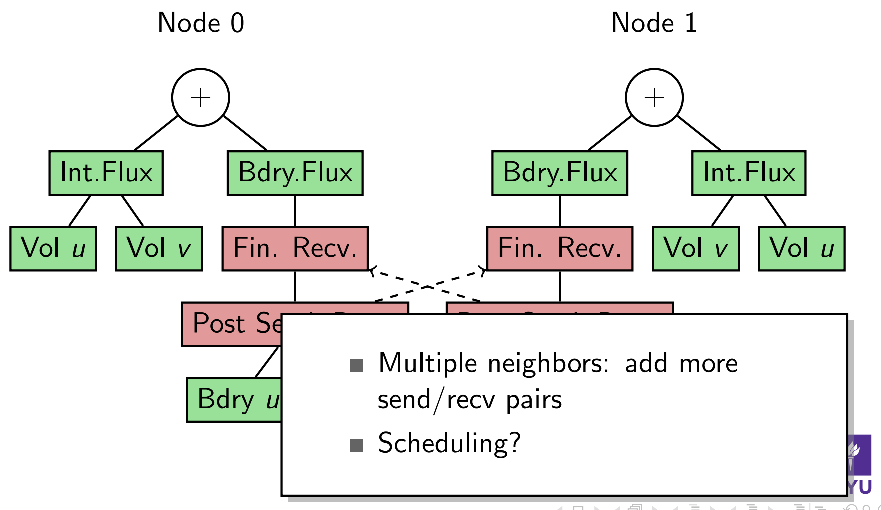
Vol u

Post Send, Recv.

Bdry v

Bdry u

Communication Insertion



- Multiple neighbors: add more send/recv pairs
- Scheduling?

Simple Optimizations

Simple optimizations:

- Linearity:
  $\partial_x(A) + \cdots + \partial_x(B) \to \partial_x(A + B) + \cdots$
- Associativity:
  $M^{-1}(L(x)) \to (M^{-1}L)(x)$
- Associativity+Linearity:
  $M^{-1}(\alpha L(x)) \to \alpha(M^{-1}L)(x)$

User should not be burdened with these.

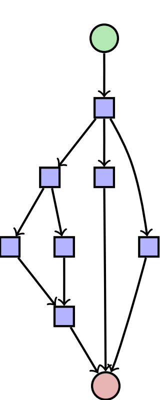
- Enables use of abstractions in operator building

Towards Execution

Already seen: Tree representation has disadvantages for execution.

Idea

Rewrite as a set of single static assignment instructions carrying dependency information.



Graph-based processing steps:

1. Build from tree
   - Assign variable names for node results
   - Realize CSEs reuse
2. Kernel fusion
3. Code generation
4. Scheduling

"Fusion"

Common for vector abstractions (e.g. `numpy`):

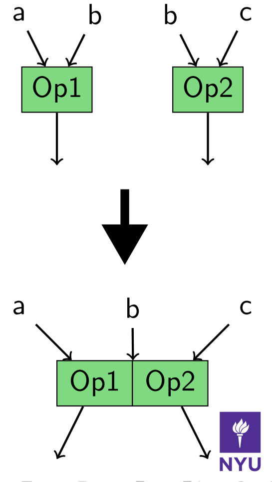
- Make temporary for result
- Load 2 (vector) operands, store 1, repeat

Issues:

- Redundant store/fetch traffic
  - No data reuse
- Little latency hiding (GPUs: in-order)
- Temporary churn

Idea

Joining instructions (vector ops, fluxes, derivatives) solves all these.



DAG for Wave Example

```
initial
  --w--> p0: { _expr8 <- MInvST0(w[0])
               _expr9 <- MInvST1(w[0])
         }
  --w--> p0: { /* B[TAG_ALL]Lift(0) */
           _expr5 <- <B[TAG_ALL]Lift((-1)*(0.5*(Int[0] +
              (-1)*Int[0])*Normal(0) +
              (-1)*0.5*Normal(0)*(Normal(0)*(Int[1] +
              (-1)*Int[1]) + Normal(1)*(Int[2] +
              (-1)*Int[2]))))>(BPair(array(w[0], w[1],
              w[2]), array(), TAG_ALL))
           _expr6 <- <B[TAG_ALL]Lift((-1)*(0.5*(Int[1] +
              Int[1])*Normal(0) + 0.5*(Int[2] +
              Int[2])*Normal(1) + (-1)*0.5*(Int[0] +
              (-1)*(-1)*Int[0]))))>(BPair(array(w[0], w[1],
              w[2]), array(), TAG_ALL))
           _expr7 <- <B[TAG_ALL]Lift((-1)*(0.5*(Int[0] +
              (-1)*Int[0])*Normal(1) +
              (-1)*0.5*Normal(1)*(Normal(0)*(Int[1] +
              (-1)*Int[1]) + Normal(1)*(Int[2] +
              (-1)*Int[2]))))>(BPair(array(w[0], w[1],
              w[2]), array(), TAG_ALL))
         }
  --w--> p0: { /* Lift(0) */
           _expr2 <- <Lift((-1)*(0.5*(Int[1] +
              Ext[1])*Normal(0) + 0.5*(Int[2] +
              Ext[2])*Normal(1) + (-1)*0.5*(Int[0] +
              (-1)*Ext[0]))))>(array(w[0], w[1], w[2]))
           _expr3 <- <Lift((-1)*(0.5*(Int[0] +
              Ext[0])*Normal(0) +
              (-1)*0.5*Normal(0)*(Normal(0)*(Int[1] +
              (-1)*Ext[1]) + Normal(1)*(Int[2] +
              (-1)*Ext[2]))))>(array(w[0], w[1], w[2]))
           _expr4 <- <Lift((-1)*(0.5*(Int[0] +
              Ext[0])*Normal(1) +
              (-1)*0.5*Normal(1)*(Normal(0)*(Int[1] +
              (-1)*Ext[1]) + Normal(1)*(Int[2] +
              (-1)*Ext[2]))))>(array(w[0], w[1], w[2]))
         }
  --w--> p0: _expr0 <- MInvST0(w[1])
  --w--> p0: _expr1 <- MInvST1(w[2])

_expr8, _expr3, _expr5 --> p0: _expr11 <- /* compiled */ (-1)*_expr8 +
                              (-1)*(_expr3 + _expr5)
_expr9, _expr4, _expr7 --> p0: _expr12 <- /* compiled */ (-1)*_expr9 +
                              (-1)*(_expr4 + _expr7)
_expr0, _expr1, _expr2, _expr6 --> p0: _expr10 <- /* compiled */ (-1)*(_expr0 + _expr1) +
                              (-1)*(_expr2 + _expr6)

_expr11, _expr12, _expr10 --> result
```

DAG for Compressible Navier-Stokes

Metaprogramming DG: Flux Terms

$$0 = \int_{\mathrm{D}_k} u_t \varphi + [\nabla \cdot F(u)]\varphi \, \mathrm{d}x - \underbrace{\int_{\partial \mathrm{D}_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*]\varphi \, \mathrm{d}S_x}_{\text{Flux term}}$$

Metaprogramming DG: Flux Terms

$$0 = \int_{\mathrm{D}_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, \mathrm{d}x - \underbrace{\int_{\partial \mathrm{D}_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*] \varphi \, \mathrm{d}S_x}_{\text{Flux term}}$$

Flux terms:

- vary by problem
- expression specified by user
- evaluated pointwise

Metaprogramming DG: Flux Terms Example

Example: Fluxes for Maxwell's Equations

$$\hat{n} \cdot (F - F^*)_E := \frac{1}{2}[\hat{n} \times ([\![H]\!] - \alpha \hat{n} \times [\![E]\!])]$$

Metaprogramming DG: Flux Terms Example

Example: Fluxes for Maxwell's Equations

$$\hat{n} \cdot (F - F^*)_E := \frac{1}{2} [\hat{n} \times ([\![H]\!] - \alpha \hat{n} \times [\![E]\!])]$$

User writes: Vectorial statement in math. notation

```
flux = 1/2*cross(normal, h.int-h.ext
       -alpha*cross(normal, e.int-e.ext))
```

Metaprogramming DG: Flux Terms Example

Example: Fluxes for Maxwell's Equations

$$\hat{n} \cdot (F - F^*)_E := \frac{1}{2} [\hat{n} \times ([\![H]\!] - \alpha \hat{n} \times [\![E]\!])]$$

We generate: Scalar evaluator in C (6×)

```
a_flux += (
    ((( val_a_field5 - val_b_field5 )* fpair->normal[2]
      - ( val_a_field4 - val_b_field4 )* fpair->normal[0])
     + val_a_field0 - val_b_field0 )* fpair->normal[0]
    - ((( val_a_field4 - val_b_field4 ) * fpair->normal[1]
        - ( val_a_field1 - val_b_field1 )* fpair->normal[2])
      + val_a_field3 - val_b_field3 ) * fpair->normal[1]
    )*value_type (0.5);
```

Outline

1 Introduction

2 Hedge How-To

3 Under the Hood

4 Conclusions

Maxwell DG on Nvidia GTX280

GPU

GFlops/s

0 50 100 150 200 250 300

2 4 6 8

Polynomial Order N

Memory Bandwidth on a GTX 280

Global Memory Bandwidth [GB/s]

Polynomial Order N

Gather
Lift
Diff
Assy.
Peak

Multiple GPUs via MPI: 16 GPUs vs. 64 CPUs

Flop Rates: 16 GPUs vs 64 CPU cores

GPU
CPU

GFlops/s
0
1000
2000
3000
4000

0
2
4
6
8
10

Polynomial Order N

GPU-DG in Double Precision

GPU-DG: Double vs. Single Precision

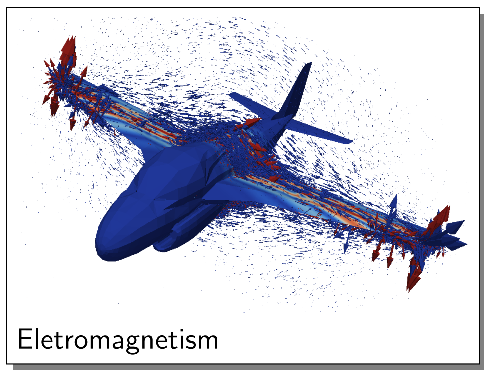
Single
Double
Ratio

GFlops/s: 0 50 100 150 200 250 300 350 400

Ratio: 0.0 0.2 0.4 0.6 0.8 1.0

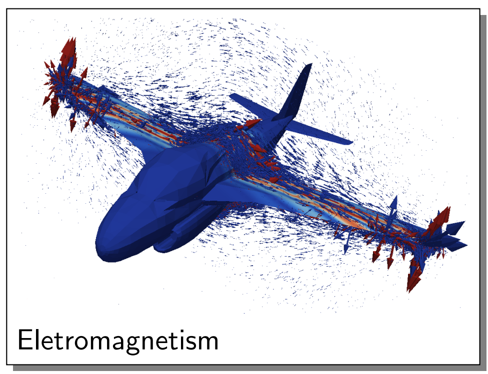
Polynomial Order N: 0 2 4 6 8 10

GPU DG Showcase

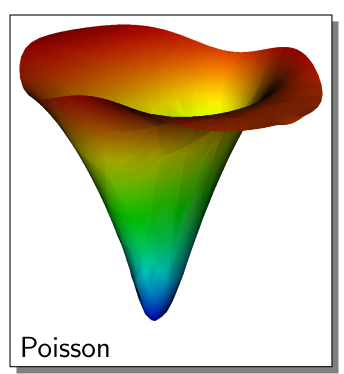


Eletromagnetism

GPU DG Showcase

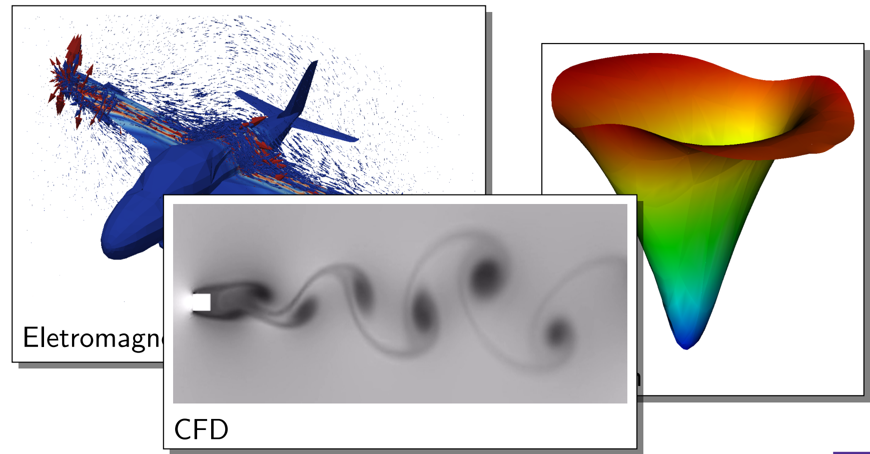


Eletromagnetism

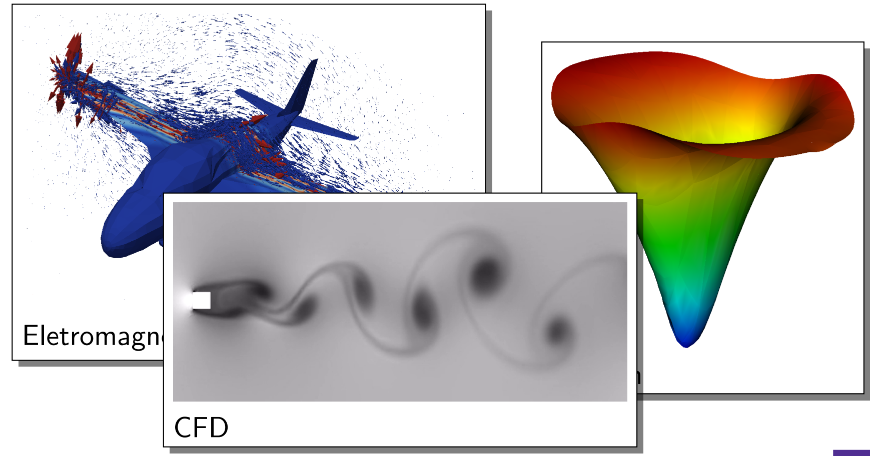


Poisson

GPU DG Showcase



GPU DG Showcase



Conclusions

- hedge: High-performance DG code with lots of features
  - MIT license
  - Comprehensive tests
  - Comes with “zoo of tools”
    - Logging
    - Visualization
- Scripting+OpenCL/GPU: Greater than the sum of its parts
  - Efficient, safe, easy, RTCG
- Code generation enables efficient flexibility
  - Without code gen, nothing in this talk would result in an efficient scheme
  - Question: How to process user input to obtain that efficient scheme?

Questions?

?

Thank you for your attention!

`http://www.cims.nyu.edu/~kloeckner/`

▸ image credits

Image Credits

- Python logo: python.org
- Machine: flickr.com/13521837@N00 (cc)
- C870 GPU: Nvidia Corp.
- Floppy disk: flickr.com/ethanhein (cc)
- Tree: sxc.hu/bertvthul
- Brick House: sxc.hu/Avolore

Outline

5 Automatic GPU Programming

6 Example 2: Boundary Integral Equations

7 DG Fluxes

Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

- Obvious idea: Let the computer do it.
- One way: Smart compilers

Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

- Obvious idea: Let the computer do it.
- One way: Smart compilers
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile

Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

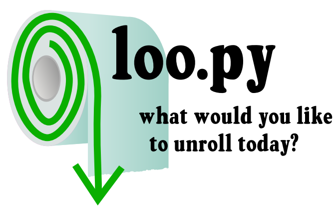
- Obvious idea: Let the computer do it.
- One way: Smart compilers
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile
- Another way: Dumb enumeration
  - Enumerate loop slicings
  - Enumerate prefetch options
  - Choose by running resulting code on actual hardware

Loo.py Example

Empirical GPU loop optimization:

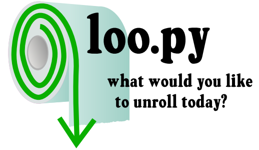
```
a, b, c, i, j, k = [var(s) for s in "abcijk"]
n = 500
k = make_loop_kernel([
    LoopDimension("i", n),
    LoopDimension("j", n),
    LoopDimension("k", n),
    ], [
    (c[i+n*j], a[i+n*k]*b[k+n*j])
    ])

gen_kwargs = {
        "min_threads": 128,
        "min_blocks": 32,
        }
```

→ Ideal case: Finds 160 GF/s kernel without human intervention.



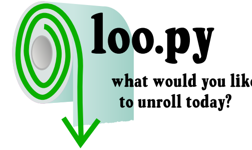
Loo.py Status

- Limited scope:
  - Require input/output separation
  - Kernels must be expressible using "loopy" model (i.e. indices decompose into "output" and "reduction")
  - Enough for DG, LA, FD, ...



Loo.py Status

- Limited scope:
  - Require input/output separation
  - Kernels must be expressible using "loopy" model
    (i.e. indices decompose into "output" and "reduction")
  - Enough for DG, LA, FD, ...
- Kernel compilation limits trial rate
- Non-Goal: Peak performance
- Good results currently for dense linear algebra and (some) DG subkernels



Outline

5 Automatic GPU Programming

6 Example 2: Boundary Integral Equations

7 DG Fluxes

Integral Equations

Given a kernel, e.g. the *Helmholtz* kernel

$$g_k(x) := \frac{1}{4\pi} \frac{e^{ik|x|}}{|x|},$$

define *layer potential operators*

$$S_k\sigma(x) := \int_\Gamma g_k(x-y)\sigma(y)\,\mathrm{d}y$$

$$D_{n,k}\sigma(x) := \int_\Gamma (n \cdot \nabla_y g_k(x-y))\sigma(y)\,\mathrm{d}y$$

and their target derivatives $\nabla_x S_k\sigma$, $\nabla_x D_{n,k}\sigma$.

User interface example

Magnetic field integral equation: $(x \in \Gamma)$

$$-\frac{1}{2}\mathbf{J}_\Gamma(y) + \hat{n} \times \nabla_x \int_\Gamma g(x-y) \times \mathbf{J}_\Gamma(y)\,\mathrm{d}y = \underbrace{-\hat{n} \times \mathbf{H}_{inc}(x)}_{\text{RHS data}}$$

Code:

```
curl_SJ = make_obj_array([
    sum(
        levi_civita((l, m, n)) * IntGdTarget(k, J[n], m)
        for m in range(3) for n in range(3))
    for l in range(3)])

mfie = -(1/2)*J + np.cross(make_normal(3), curl_SJ)
```

BIE Observations

- Very similar machinery works for FMM/BIE code
- Build in-memory representation
- Layer potentials can be evaluated on-/off-surface
  - Sometimes both within same expression
  - Infer target-bound operators

(tree-level))
- E.g.: $S_k(u)$, $\nabla_x S_k(u)$ use same expansion, can be evaluated together
  - Find, join (insn-level)

Outline

5 Automatic GPU Programming

6 Example 2: Boundary Integral Equations

7 DG Fluxes