Paper to GPU: Optimizing and executing discontinuous Galerkin operators in Python

Andreas Klöckner

Courant Institute of Mathematical Sciences
New York University

March 1, 2011
Thanks

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

1 Setting

2 Expressing DG Operators

3 Conclusions
Outline

1 Setting

2 Expressing DG Operators

3 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.
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]

- Device-neutral (Nv GPU, AMD GPU, Intel)
- 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. It comes with RTCG and defines:

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

It is device-neutral (Nv GPU, AMD GPU, Intel/AMD CPU) and vendor-neutral.
Flexibility in Computer Codes

User input requires flexibility.

When be flexible?

- **Compilation Time**
  - Anticipate input? (e.g. numpy)
  - Or put user in charge of compilation? (e.g. C++ templates)
Flexibility in Computer Codes

User input requires flexibility.

When be flexible?

- **Compilation Time**
  - Anticipate input? (e.g. numpy)
  - Or put user in charge of compilation? (e.g. C++ templates)

- **During Execution**
  Function pointers/Virtual Methods
  (e.g. numpy ufunc) → slow/difficult
Flexibility in Computer Codes

User input requires flexibility.

When be flexible?

- **Compilation Time**
  - Anticipate input? (e.g. `numpy`)
  - Or put user in charge of compilation? (e.g. C++ templates)

- **Do we need to decide?**
  No! ‘Compilation time’ is not that special. Generate specific code at runtime, but before execution.

- **During Execution**
  Function pointers/Virtual Methods (e.g. `numpy ufunc`) → slow/difficult
Outline

1 Setting

2 Expressing DG Operators
   - The Method
   - Representation

3 Conclusions
Outline

1 Setting

2 Expressing DG Operators
   - The Method
   - Representation

3 Conclusions
Discontinuous Galerkin Method

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

Goal
Solve a conservation law on $\Omega$:

$$u_t + \nabla \cdot F(u) = 0$$

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

$$\partial_t E - \frac{1}{\varepsilon} \nabla \times H = -j\varepsilon, \quad \partial_t H + \frac{1}{\mu} \nabla \times E = 0,$$
$$\nabla \cdot E = \rho \varepsilon, \quad \nabla \cdot H = 0.$$
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 \( \Omega \):
\[
    u_t + \nabla \cdot F(u) = 0
\]

**Example**
*Maxwell’s Equations:* EM field: \( E(x, t), H(x, t) \) on \( \Omega \) governed by
\[
\begin{align*}
    \partial_t E - \frac{1}{\varepsilon} \nabla \times H &= -\frac{j}{\varepsilon}, \\
    \nabla \cdot E &= \frac{\rho}{\varepsilon}, \\
    \partial_t H + \frac{1}{\mu} \nabla \times E &= 0, \\
    \nabla \cdot H &= 0.
\end{align*}
\]
Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

\[ 0 = \int_{D_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, dx \]
\[ = \int_{D_k} u_t \varphi - F(u) \cdot \nabla \varphi \, dx + \int_{\partial D_k} (\hat{n} \cdot F)^* \varphi \, dS_x, \]

Substitute in basis functions, introduce elementwise stiffness, mass, and surface mass matrices matrices \( S, M, M_A \):

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

For straight-sided simplicial elements:
Reduce \( D^{\partial \nu} \) and \( L \) to reference matrices.
Decomposition of a DG operator into Subtasks

DG’s execution decomposes into two (mostly) separate branches:

- Flux Gather
- Flux Lifting
- Local Differentiation
- $F(u^k)$
- $\partial_t u^k$

Green: Element-local parts of the DG operator.
Outline

1 Setting

2 Expressing DG Operators
   - The Method
   - Representation

3 Conclusions
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:]

# 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))))
\[ \begin{align*}
u^* &= \hat{n} \cdot \{v\} - \frac{1}{2}(u^- - u^+), \\
v^* &= \hat{n} \left( \left\{ u \right\} - \frac{\hat{n}}{2} \cdot (v^- - v^+) \right) \\
\end{align*} \]

\[ \partial_t u + \nabla_x \cdot v = 0, \quad \partial_t v + \nabla_x u = 0 \]

\[ \begin{align*}
\text{flux} &= - \text{join_fields} ( \\
& \quad \text{dot}(v.\text{avg}, \text{normal}) \\
& \quad - 0.5*(u.\text{int} - u.\text{ext}),
\end{align*} \]

\[ u.\text{avg} \ast \text{normal} \\
\quad - 0.5*(\text{normal} \ast \text{dot}(\text{normal}, v.\text{int} - v.\text{ext})) \]

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

```
normal = make_normal(d) 

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

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

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

$\partial_t v + \nabla_x u = 0$

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

This talk: Ideas on how to do so.
Hedge DG Solver

- High-Level Operator Description
  - Maxwell’s
  - Euler
  - Poisson
  - Compressible Navier-Stokes, …

- One Code runs…
  - …on CPU, CUDA
  - …on {CPU,CUDA}+MPI
  - …in 1D, 2D, 3D
  - …at any order

- Generates C++ or CUDA code at Run Time
- Open Source (GPL3)
- Written in Python, uses PyCUDA
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
Communication Insertion

Node 0

\[ \text{Int.Flux} \]

\[ \text{Vol } u \quad \text{Vol } v \]

Node 1

\[ \text{Int.Flux} \]

\[ \text{Vol } v \quad \text{Vol } u \]

Multiple neighbors: add more send/recv pairs

Scheduling: later
Communication Insertion

Node 0

\[
\begin{align*}
&\text{Int. Flux} \quad \text{Bdry. Flux} \\
&\text{Vol } u \quad \text{Vol } v \\
&\text{Fin. Recv.} \\
&\text{Post Send, Recv.} \\
&\text{Bdry } u \quad \text{Bdry } v
\end{align*}
\]

Node 1

\[
\begin{align*}
&\text{Bdry. Flux} \quad \text{Int. Flux} \\
&\text{Fin. Recv.} \\
&\text{Post Send, Recv.} \\
&\text{Bdry } v \quad \text{Bdry } u \\
&\text{Vol } v \quad \text{Vol } u
\end{align*}
\]
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

- Multiple neighbors: add more send/recv pairs
- Scheduling: later
What about Common Subexpressions?

- $a \times (4 - b)$
- $(4 - b) - f$

Common Subexpressions (CSEs) are important to reduce rewrite complexity. Using explicit CSE tagging rather than detection limits memory consumption. How to realize reuse? Later.

Andreas Klöckner

Paper to GPU: Optimizing and executing DG operators
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
Type Inference: Motivation

Data along each tree edge can have a variety of types:

- Field / Scalar
- Volume / Interior faces / Boundary
- Nodal / Quadrature grid

Useful to know which is which:

- Can identify user error
- Can specialize operators
  - Quadrature: User only specifies time of interpolation to q.grid
Type Inference: Algorithm

1. For each edge of the tree, assign a type
   - Initially ‘unspecified’

2. For each node in DFS order:
   1. Read type of input edges and output edge
   2. Compute new in/out types
   3. Unify inputs and output with new types

3. Repeat until types don’t change

Remarks:
- Better type inference algorithms exist
  - Worst case: $O(\text{depth})$ DFS to converge
- Tricky to get CSE caching right
Simple Optimizations

Simple optimizations:

- **Linearity:**
  \[ \partial_x(A) + \cdots + \partial_x(B) \rightarrow \partial_x(A + B) + \cdots \]

- **Associativity:**
  \[ M^{-1}(L(x)) \rightarrow (M^{-1}L)(x) \]

- **Associativity+Linearity:**
  \[ M^{-1}(\alpha L(x)) \rightarrow \alpha(M^{-1}L)(x) \]

User should not be burdened with these.

- Enables use of abstractions in operator building
Rewriting Stages

Tree-based processing:

1. Check for errors, fold constants (empty surfaces, etc.)
2. BC-to-flux rewrite
3. Insert communication
4. Infer ‘types’, specialize
5. Transcribe to reference element
6. Specialize to hardware
7. Join element-local operators
8. Join derivatives
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.
Dealing with Asynchrony

For distributed memory runs, get asynchronous operations:

- MPI transfers
- GPU-CPU transfers

So far: Only waiting semantics captured.

**Idea:** Each instruction can either finish its job and return a result, or return a “promise” of a future result.

A promise can:

- Check for its completion
- Wait for its completion
### Scheduling

#### Algorithm

1. Find ‘dead’ variables
2. Check promises for completion
3. Find highest-priority instruction w/ satisfied dependencies
   - If available: Run, perform assignments
   - If none available: Wait for a future
4. Repeat until no instructions left

#### Remarks:

- Quadratic algorithm
- Fix 1: Cache based on VM state
- Fix 2: Attempt to generate static schedule

---

*Andreas Klöckner* | *Paper to GPU: Optimizing and executing DG operators*
Scheduling

Algorithm

1. Find ‘dead’ variables
2. Check promises for completion
3. Find highest-priority instruction w/ satisfied dependencies
   - If available: Run, perform assignments
   - If none available: Wait for a future
4. Repeat until no instructions left

Remarks:
- Quadratic algorithm
- Fix 1: Cache based on VM state
- Fix 2: Attempt to generate static schedule
- Possible issue: Number of temporaries a.k.a. “register pressure”
DAG for Wave Example

initial

\[
\begin{align*}
_p0: _\text{expr0} & \leftarrow \text{MInvST0}(w[1]) \\
_p0: _\text{expr1} & \leftarrow \text{MInvST1}(w[2]) \\
_p0: \{ \text{ /* Lift(0) */ } \\
& \text{_expr2} \leftarrow \text{Lift}((-1)*(0.5*(\text{Int}[1] + \text{Ext}[1])*\text{Normal}(0) + 0.5*(\text{Int}[2] + \text{Ext}[2])*\text{Normal}(1) + (-1)*0.5*(\text{Int}[0] + (-1)*\text{Ext}[0])))>(\text{array}(w[0], w[1], w[2]), \text{array}(), \text{TAG_ALL}) \\
& \text{_expr3} \leftarrow \text{Lift}((-1)*(0.5*(\text{Int}[0] + \text{Ext}[0])*\text{Normal}(0) + (-1)*0.5*\text{Normal}(0)*(\text{Normal}(0)*(\text{Int}[1] + (-1)*\text{Ext}[1]) + \text{Normal}(1)*(\text{Int}[2] + (-1)*\text{Ext}[2])))>(\text{array}(w[0], w[1], w[2]), \text{array}(), \text{TAG_ALL}) \\
& \text{_expr4} \leftarrow \text{Lift}((-1)*(0.5*(\text{Int}[0] + \text{Ext}[0])*\text{Normal}(1) + (-1)*0.5*\text{Normal}(1)*(\text{Normal}(0)*(\text{Int}[1] + (-1)*\text{Ext}[1]) + \text{Normal}(1)*(\text{Int}[2] + (-1)*\text{Ext}[2])))>(\text{array}(w[0], w[1], w[2]), \text{array}(), \text{TAG_ALL}) \\
\} \\
\end{align*}
\]

\[
\begin{align*}
_p0: \{ \text{ /* B[TAG_ALL]Lift(0) */ } \\
& \text{_expr5} \leftarrow \text{B[TAG_ALL]Lift}((-1)*(0.5*(\text{Int}[1] + (-1)*\text{Int}[1])*\text{Normal}(0) + (-1)*0.5*\text{Normal}(0)*(\text{Normal}(0)*(\text{Int}[1] + (-1)*\text{Int}[1]) + \text{Normal}(1)*(\text{Int}[2] + (-1)*\text{Int}[2])))>(\text{BPair}(\text{array}(w[0], w[1], w[2]), \text{array}(), \text{TAG_ALL}) \\
& \text{_expr6} \leftarrow \text{B[TAG_ALL]Lift}((-1)*(0.5*(\text{Int}[0] + (-1)*\text{Int}[0])*\text{Normal}(0) + (-1)*0.5*\text{Normal}(0)*(\text{Normal}(0)*(\text{Int}[1] + (-1)*\text{Int}[1]) + \text{Normal}(1)*(\text{Int}[2] + (-1)*\text{Int}[2])))>(\text{BPair}(\text{array}(w[0], w[1], w[2]), \text{array}(), \text{TAG_ALL}) \\
& \text{_expr7} \leftarrow \text{B[TAG_ALL]Lift}((-1)*(0.5*(\text{Int}[1] + (-1)*\text{Int}[1])*\text{Normal}(1) + (-1)*0.5*\text{Normal}(1)*(\text{Normal}(0)*(\text{Int}[1] + (-1)*\text{Int}[1]) + \text{Normal}(1)*(\text{Int}[2] + (-1)*\text{Int}[2])))>(\text{BPair}(\text{array}(w[0], w[1], w[2]), \text{array}(), \text{TAG_ALL}) \\
\} \\
\end{align*}
\]

\[
\begin{align*}
_p0: \{ \text{ /* compiled */ } -1*_{\text{expr9}} + (-1)*(_{\text{expr4}} + _{\text{expr7}}) \\
_p0: \{ \text{ /* compiled */ } -1*_{\text{expr8}} + (-1)*(_{\text{expr3}} + _{\text{expr5}}) \\
_p0: \{ \text{ /* compiled */ } -1*_{\text{expr11}} + (-1)*(_{\text{expr12}} + _{\text{expr10}}) \\
\} \\
\end{align*}
\]
DAG for Compressible Navier-Stokes
Outline

1. Setting
2. Expressing DG Operators
3. Conclusions
GPU DG Showcase

Eletromagnetism
GPU DG Showcase

Eletromagnetism

Poisson
GPU DG Showcase

Eletromagnetism

CFD
Conclusions

- 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?

- Expression processing:
  Shares ideas with compiler writing

- Expression-with-operators:
  Common pattern in PDE codes
  - Very similar machinery works in BIE code

- Machinery:
  - 3k LOP for symbolic math
  - 5k LOP DG-specific
Questions?

Thank you for your attention!

http://www.cims.nyu.edu/~kloeckner/
Image Credits

- Brain in box: sxc.hu/svilen001
- Tree: sxc.hu/bertvthul
- Wrench: ?
- Brick House: sxc.hu/Avolare
- Steps: sxc.hu/ibeeby
- Clock: sxc.hu/cema
Outline

4 Automatic GPU Programming

5 Example 2: Boundary Integral Equations

6 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
Empirical GPU loop optimization:

```python
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

4 Automatic GPU Programming

5 Example 2: Boundary Integral Equations

6 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*

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

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

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

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

\[-\frac{1}{2} J_\Gamma(y) + \hat{n} \times \nabla_x \int_{\Gamma} g(x-y) \times J_\Gamma(y) \, dy = -\hat{n} \times H_{inc}(x)\]

RHS data

Code:

```python
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

4 Automatic GPU Programming

5 Example 2: Boundary Integral Equations

6 DG Fluxes
Metaprogamming DG: Flux Terms

\[ 0 = \int_{D_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, dx - \int_{\partial D_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*] \varphi \, dS_x \]

Flux term
Metaprogramming DG: Flux Terms

\[ 0 = \int_{D_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, dx - \int_{\partial D_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*] \varphi \, dS_x \]

Flux terms:

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

```python
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\times)

```c
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);
```