Model-based Performance Optimization for GPU DG-FEM

James Stevens

University of Illinois at Urbana-Champaign

February 28, 2017

Thanks

Nick Curtis
Dominic Kempf
Andreas Klöckner
Matt Wala
Tim Warburton
Lucas Wilcox

Outline

Outline

Outline

Factors influencing time to PDE solution

Problem:

- Desired accuracy
- Data: frequency, source terms, BCs, ...

Factors influencing time to PDE solution

Problem:

- Desired accuracy
- Data: frequency, source terms, BCs, ...

Numerical method:

- Order of accuracy
- Number of degrees of freedom

Factors influencing time to PDE solution

Problem:

- Desired accuracy
- Data: frequency, source terms, BCs, ...

Numerical method:

- Order of accuracy
- Number of degrees of freedom

Algorithm:

- Asymptotic flop count, mem access count
- How efficiently hardware is used

Factors influencing time to PDE solution

Problem:

- Desired accuracy
- Data: frequency, source terms, BCs, ...

Numerical method:

- Order of accuracy
- Number of degrees of freedom

Algorithm:

- Asymptotic flop count, mem access count
- How efficiently hardware is used

Machine:

- Cost of memory accesses
- Pattern of memory accesses
- Constants in the asymptotic estimates

Factors influencing time to PDE solution

Problem:

- Desired accuracy
- Data: frequency, source terms, BCs, ...

Numerical method:

- Order of accuracy
- Number of degrees of freedom

Algorithm:

- Asymptotic flop count, mem access count
- How efficiently hardware is used

Machine:

- Cost of memory accesses
- Pattern of memory accesses
- Constants in the asymptotic estimates

Observation: Orders of magnitude can be *saved or squandered* in each of these places

Factors influencing time to PDE solution

Problem:

- Desired accuracy
- Data: frequency, source terms, BCs, ...

Numerical method:

- Order of accuracy
- Number of degrees of freedom

Algorithm:

- Asymptotic flop count, mem access count
- How efficiently hardware is used

Machine:

- Cost of memory accesses
- Pattern of memory accesses
- Constants in the asymptotic estimates

Observation: Orders of magnitude can be *saved or squandered* in each of these places

Factors influencing time to PDE solution

Problem:

- Desired accuracy
- Data: frequency, source terms, BCs, ...

Numerical method:

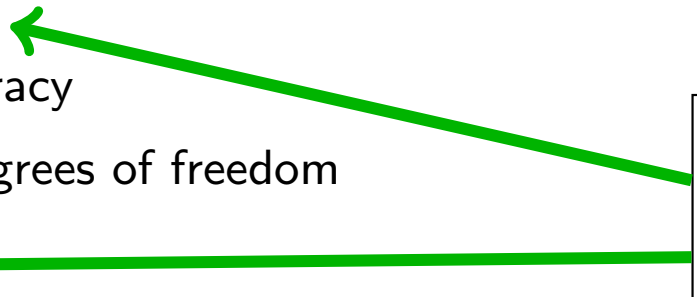
- Order of accuracy
- Number of degrees of freedom

Algorithm:

- Asymptotic flop count, mem access count
- How efficiently hardware is used

Machine:

- Cost of memory accesses
- Pattern of memory accesses
- Constants in the asymptotic estimates



Observation: Orders of magnitude can be *saved or squandered* in each of these places

Factors influencing time to PDE solution

Problem:

- Desired accuracy
- Data: frequency, source terms, BCs, ...

Numerical method:

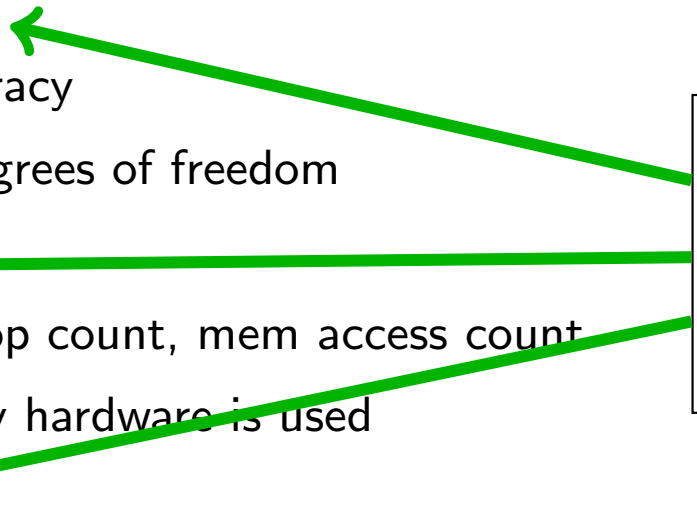
- Order of accuracy
- Number of degrees of freedom

Algorithm:

- Asymptotic flop count, mem access count
- How efficiently hardware is used

Machine:

- Cost of memory accesses
- Pattern of memory accesses
- Constants in the asymptotic estimates



Observation: Orders of magnitude can be *saved or squandered* in each of these places

Time to solution: Another aspect

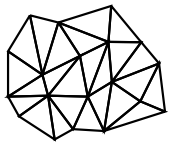
Time to numerical PDE solution is determined by...

...how quickly I can write a solver.

Outline

Discontinuous Galerkin Method

Let $\Omega := \bigcup_i \mathsf{D}_k \subset \mathbb{R}^d$.



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:

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

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

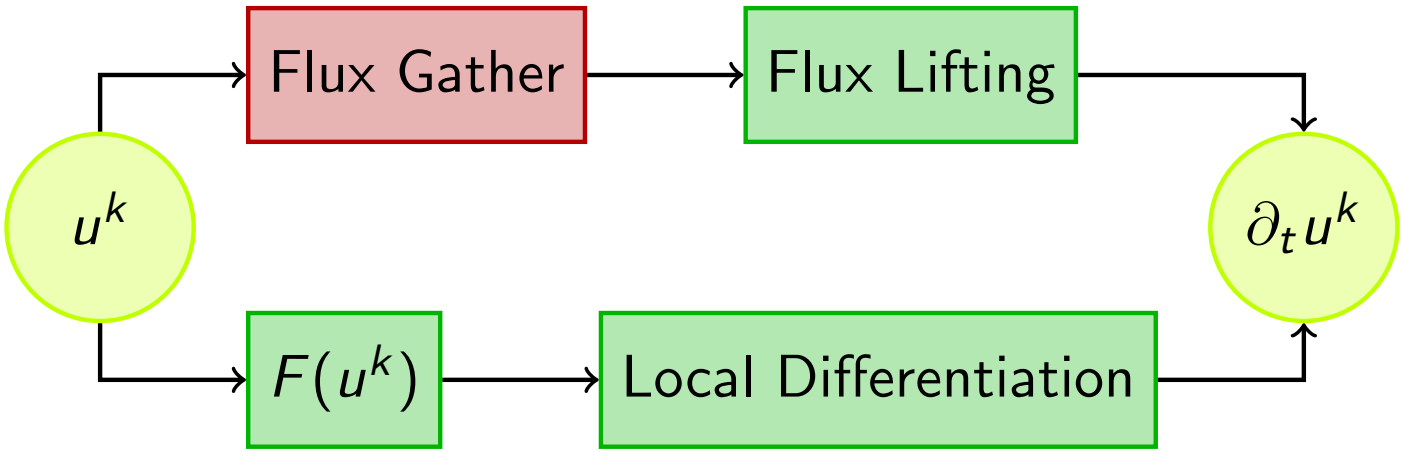
$$\partial_t u^k = -\sum_{\nu} D^{\partial_\nu, k}[F(u^k)] + L^k[\hat{n} \cdot F(u^k) - (\hat{n} \cdot F)^*]|_{A \subset \partial \mathrm{D}_k}.$$

"Lifting" matrix L combines M^{-1} and M_A

Decomposition of a DG operator into Subtasks

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

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

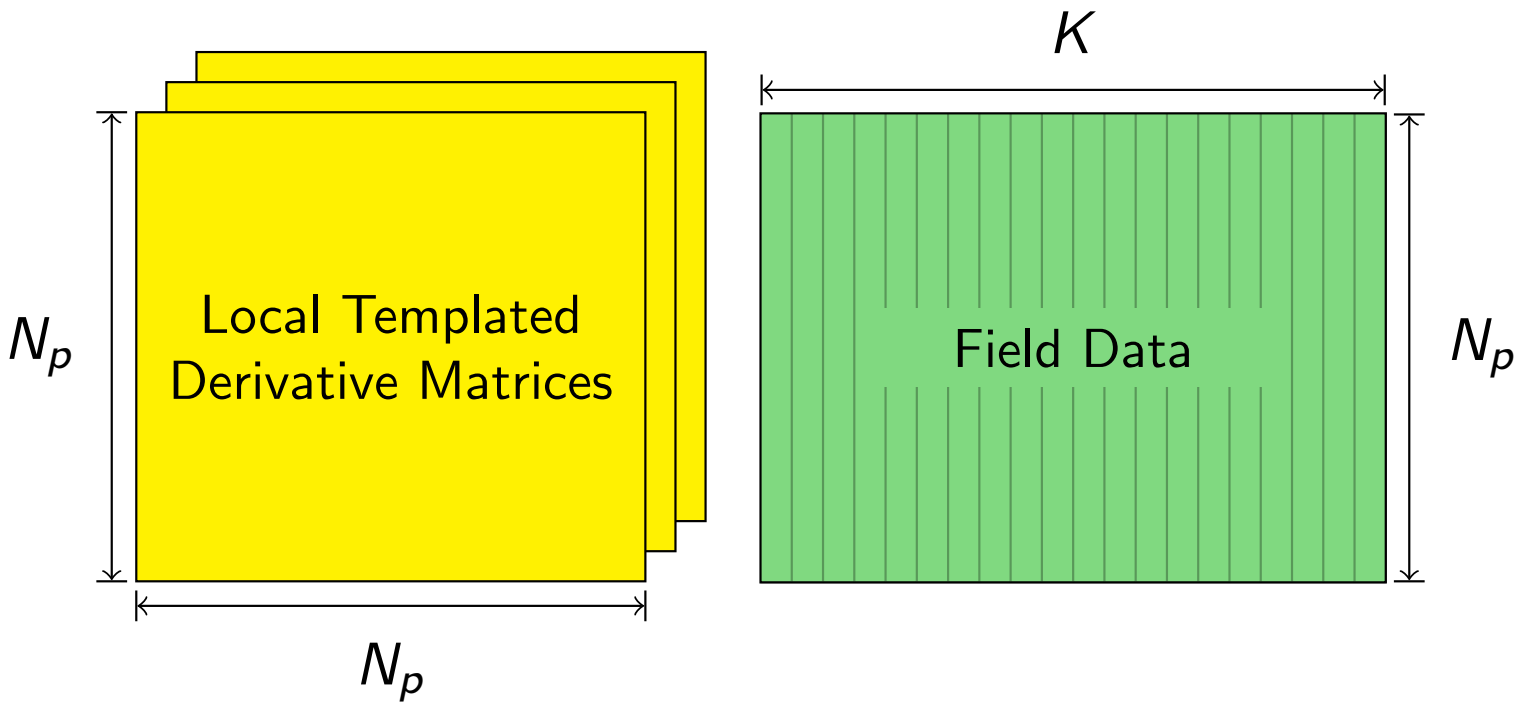


Green: Element-local parts of the DG operator.

Outline

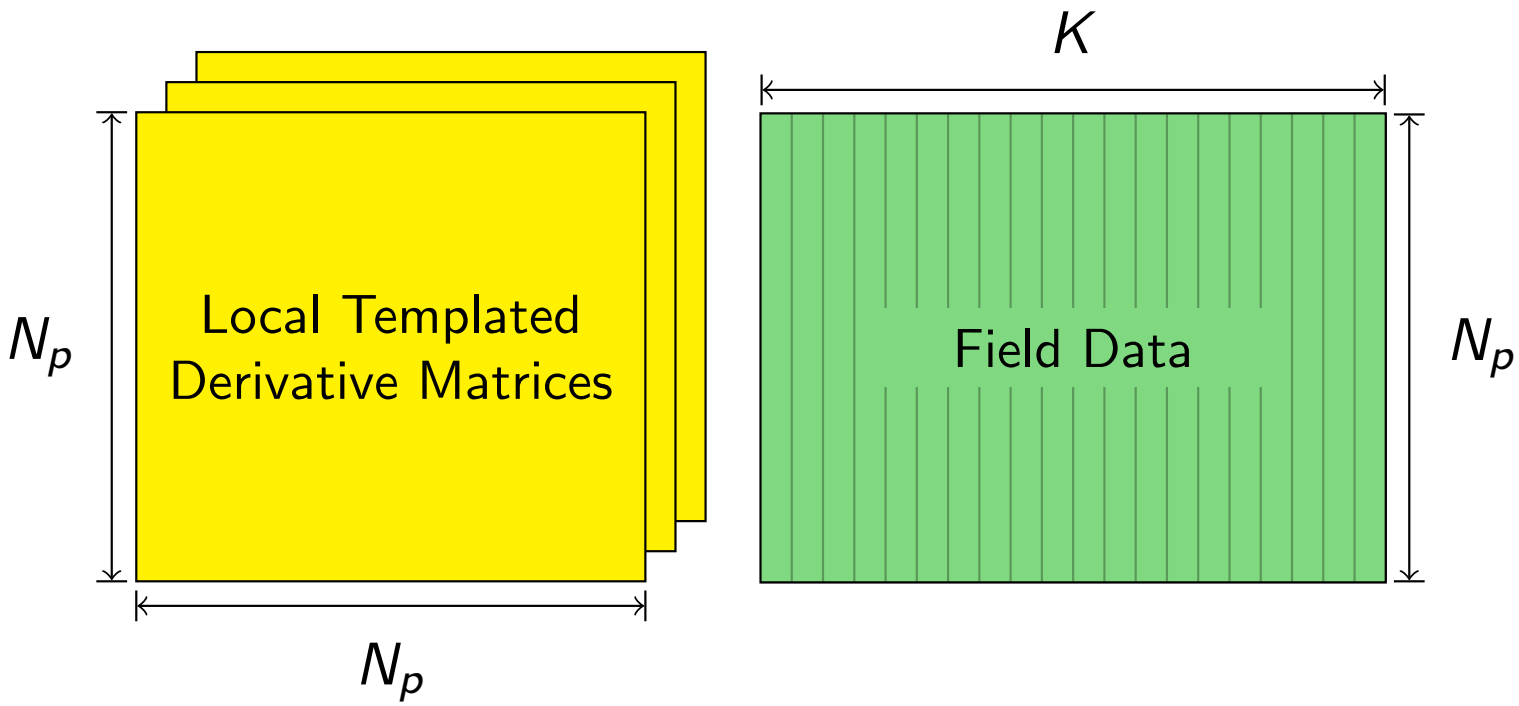
Outline

Element-Local Operations: Differentiation



$$\sum_{\nu} D^{\partial_\nu,k}[F(u^k)]$$

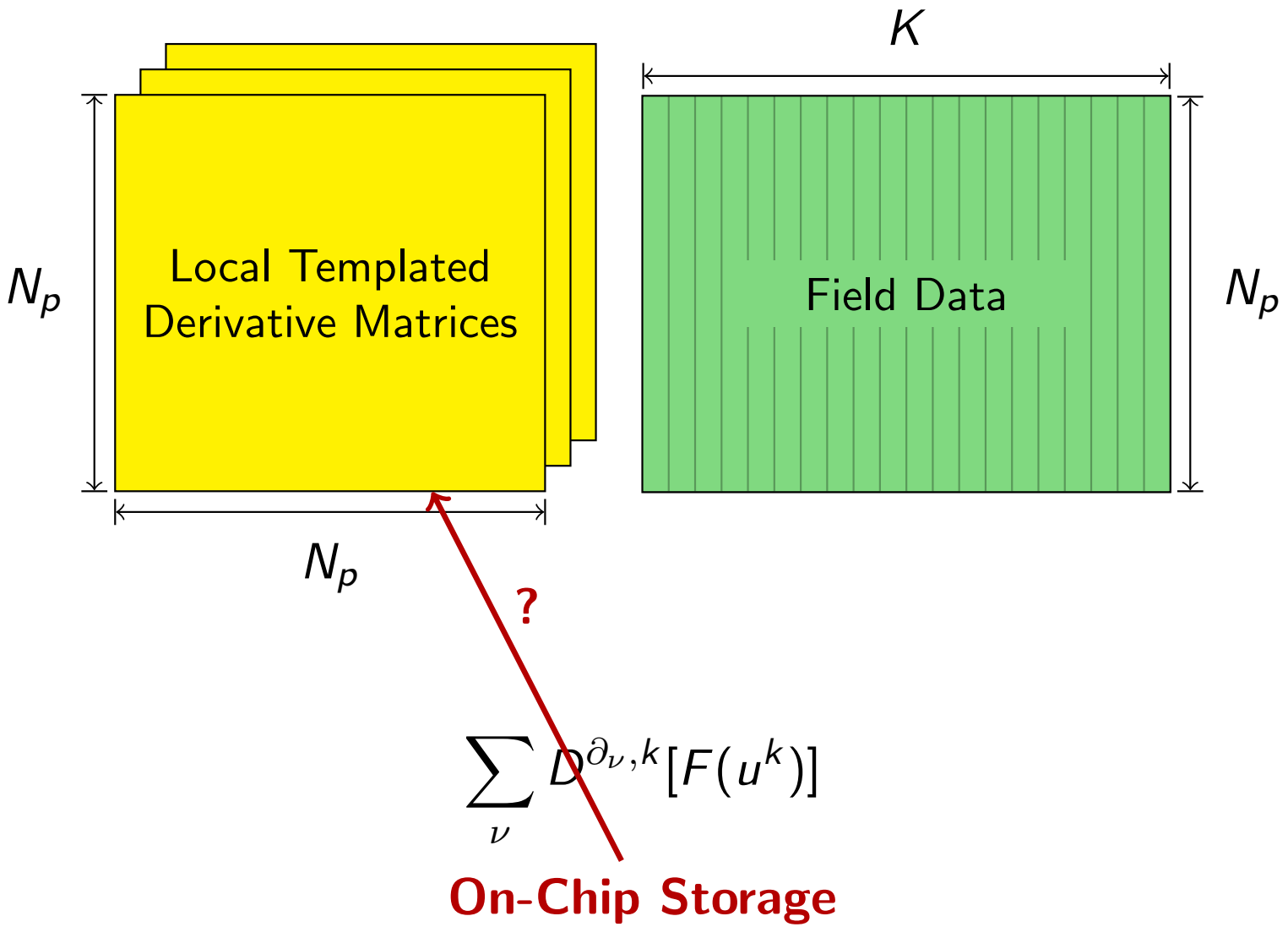
Element-Local Operations: Differentiation



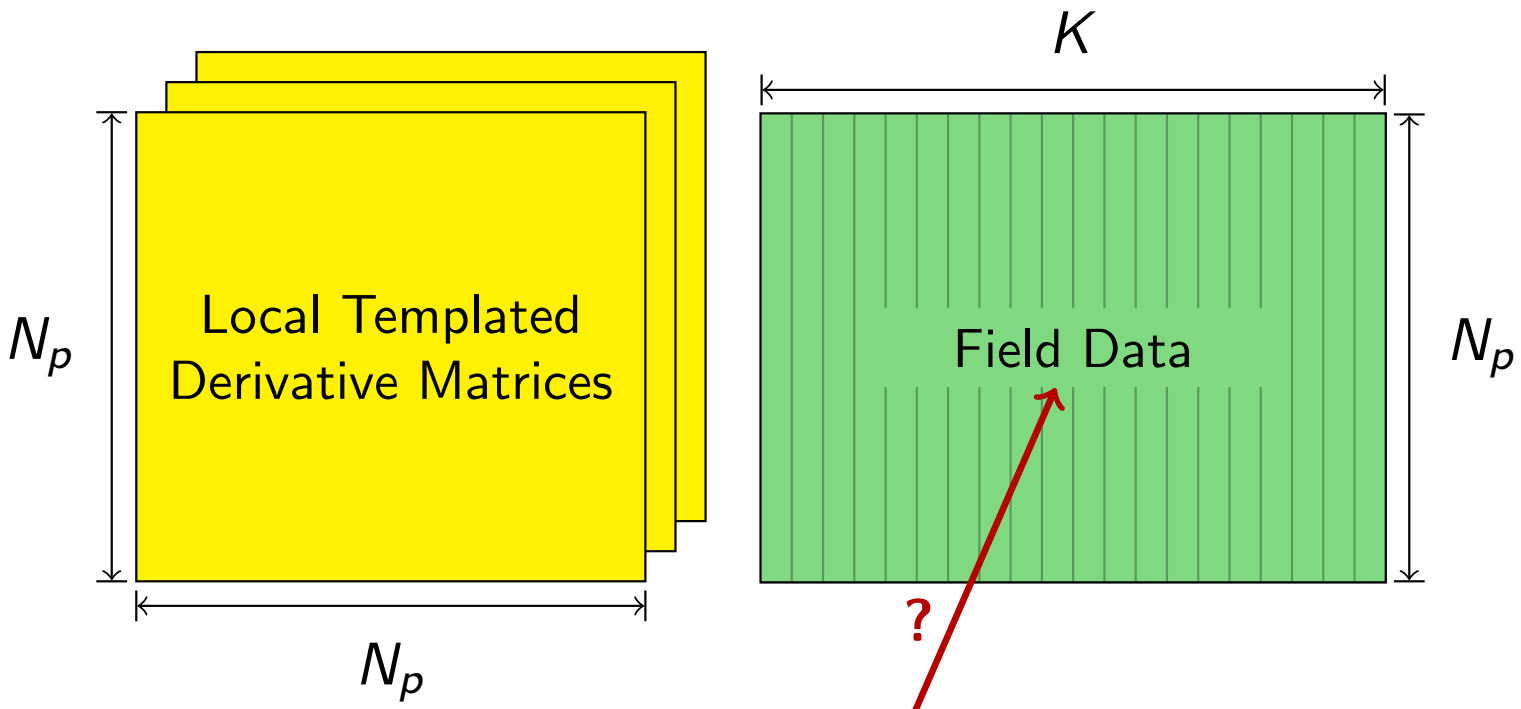
$$\sum_{\nu} D^{\partial_\nu,k}[F(u^k)]$$

On-Chip Storage

Element-Local Operations: Differentiation



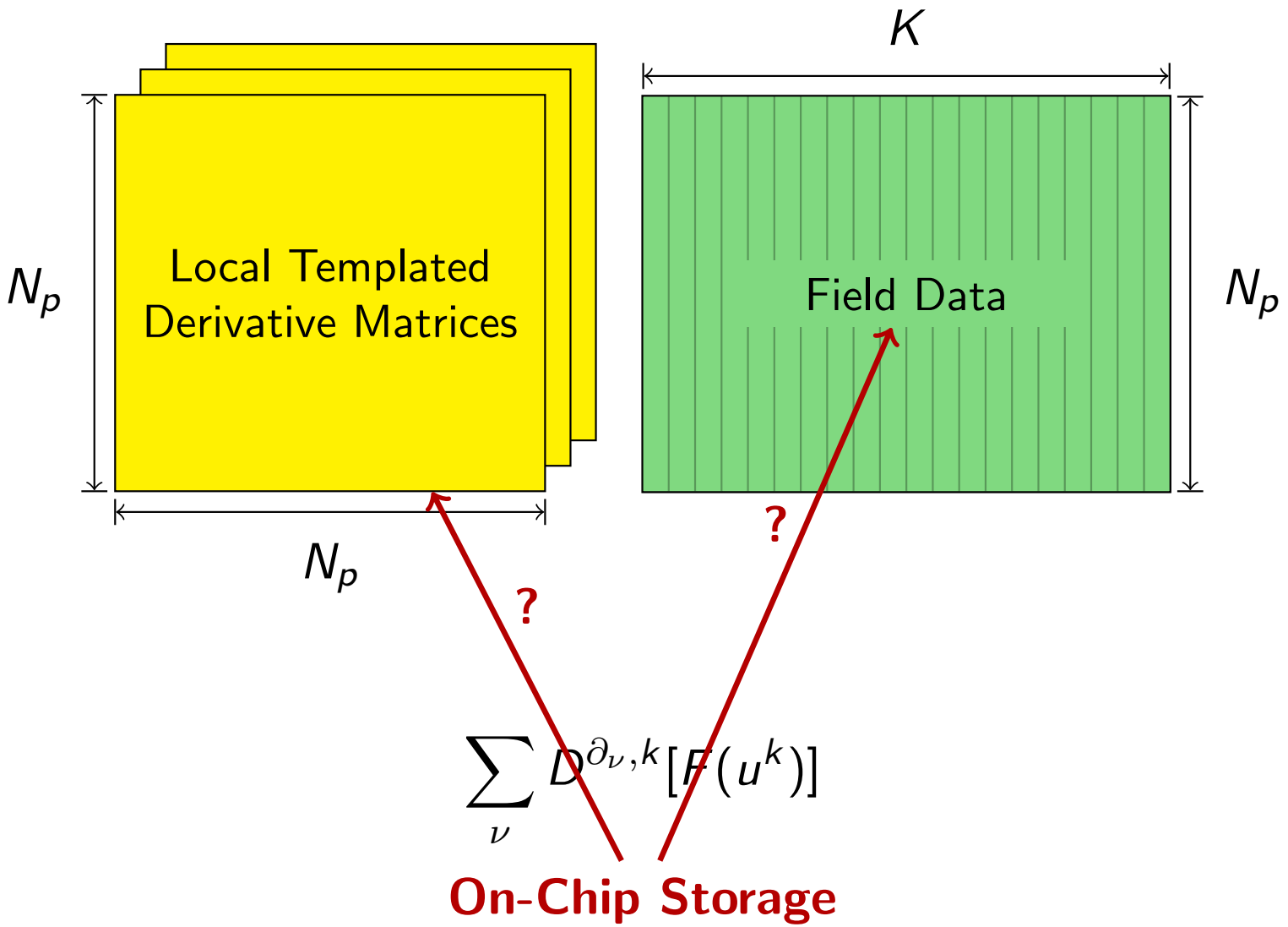
Element-Local Operations: Differentiation



$$\sum_{\nu} D^{\partial_\nu,k}[F(u^k)]$$

On-Chip Storage

Element-Local Operations: Differentiation



Best use for on-chip memory?

Basic Problem

On-chip storage is scarce. . .

. . . and will be for the foreseeable future.

Possible uses:

- Matrix/Matrices
- Part of a matrix
- Field Data
- Both

How to decide? When does it matter?

DG on GPUs: Overcoming Challenges

- Challenging implementation decisions with performance consequences
- Choices are hardware-specific
- Lots of time spent writing/tuning code

DG on GPUs: Overcoming Challenges

- Challenging implementation decisions with performance consequences
- Choices are hardware-specific
- Lots of time spent writing/tuning code

Need: Software framework that converts PDE description to code, then tunes code to hardware automatically.

- Decrease reliance on knowledge of hardware internals
- Focus on solving mathematical problem rather than tuning code

Outline

Outline

Framework

Grudge: A purpose-built description language for DG operators

- Takes description of PDE and "compiles" it into OpenCL code
- More operator-focused than Fenics
- Focus on performance rather than generality

Grudge Specification Example: Wave Equation

```
result = (
    - join_fields (
        -self.c*np.dot(nabla, v),
        -self.c*(nabla*u)
        )
    +
    sym.InverseMassOperator()(
        sym.FaceMassOperator()(
            flux(sym.int_tpair(w))
            + flux(sym.bv_tpair(
                self.dirichlet_tag, w, dir_bc))
            + flux(sym.bv_tpair(
                self.neumann_tag, w, neu_bc))
            + flux(sym.bv_tpair(
                self.radiation_tag, w, rad_bc))
            )))
```

```
def flux(self, w):
    u = w[0]
    v = w[1:]
    normal = sym.normal(w.dd,
                        self.ambient_dim)

    flux_weak = join_fields(
            np.dot(v.avg, normal),
            u.avg * normal)
    ...
    if self.flux_type == "upwind":
        flux_weak -= \
            self.sign * join_fields(
                0.5*(u.int-u.ext),
                0.5*(normal * np.dot(
                    normal, v.int-v.ext)
                    ))
```

Grudge Specification Exampl

```
result = (
    - join_fields (
        -self.c*np.dot(nabla, v),
        -self.c*(nabla*u)
        )
    +
    sym.InverseMassOperator()(
        sym.FaceMassOperator()(
            flux(sym.int_tpair(w))
            + flux(sym.bv_tpair(
                self.dirichlet_tag, w, dir_bc))
            + flux(sym.bv_tpair(
                self.neumann_tag, w, neu_bc))
            + flux(sym.bv_tpair(
                                    , w, rad_bc))
```

$$\partial_t^2 u = c^2 \Delta u$$

$$\searrow$$

$$\partial_t u + c \nabla \cdot v = 0,$$

$$\partial_t v + c \nabla u = 0$$

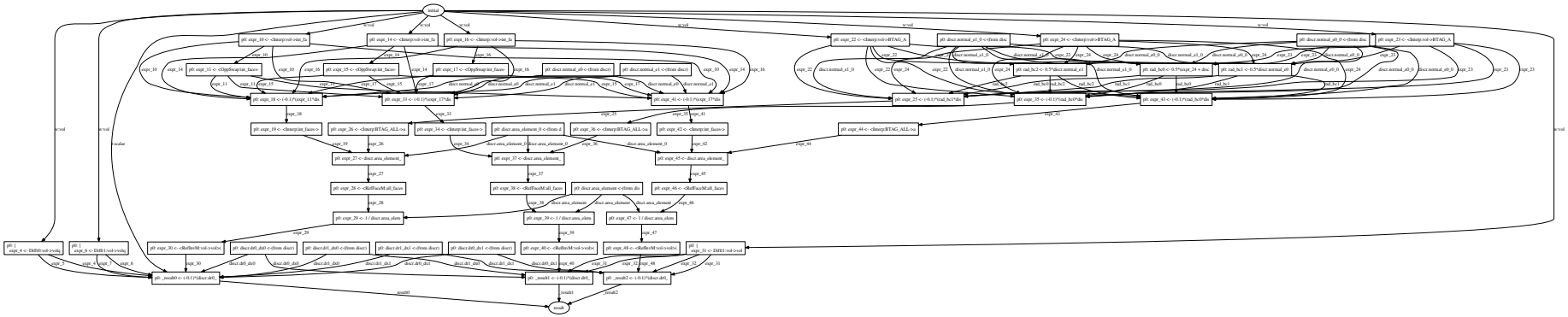
$$u^* = \hat{n} \cdot \{v\} - \frac{1}{2}(u^- - u^+),$$

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

```
def flux(self, w):
    u = w[0]
    v = w[1:]
    normal = sym.normal(w.dd,
                        self.ambient_dim)

    flux_weak = join_fields(
            np.dot(v.avg, normal),
            u.avg * normal)
    ...
    if self.flux_type == "upwind":
        flux_weak -= \
            self.sign* join_fields(
                0.5*(u.int-u.ext),
                0.5*(normal * np.dot(
                    normal, v.int-v.ext)
                    ))
```

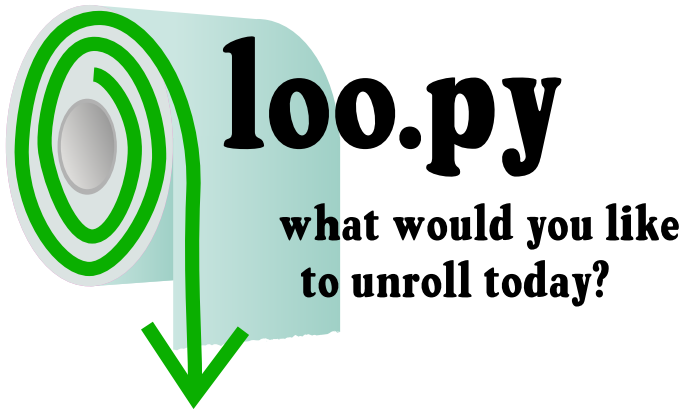
DAG for Wave Example



Outline

Loo.py

- Programming system embedded in Python
- Defines a **data model** for array-style computations and a library of **transformations** that operate on this model



Transformations: Loop tiling, instruction-level parallelism, vectorization, unrolling, prefetching, AoS↔SoA, and more!

Loo.py

Specify **mathematical intent**:

```
kn= make_kernel(
      "{[i,k,j]: 0<=i<n and 0<=k<m and 0<=j<l}", # loop domain
      "c[i, j] = sum(k, a[i, k]*b[k, j])" # instructions
      , name="matmul", assumptions="n,m,l >= 1")
```

Specify **transformations**:

```
# parallelize i and j loops
kn= split_iname(kn, "i", 16, outer_tag="g.0", inner_tag="l.1")
kn= split_iname(kn, "j", 16, outer_tag="g.1", inner_tag="l.0")
```

Outline

Modeling Execution Time

Model execution time as linear combination of kernel properties

$$T_{\text{wall}}(\mathbf{n}) \approx \sum_{i=1}^{N_{\text{properties}}} \alpha_i p_i(\mathbf{n}),$$

where $\mathbf{n}$ is parameter set governing problem size and α_i is weight (run time cost) for i^{th} property

Modeling Execution Time

$$T_{\text{wall}}(\mathbf{n}) \approx \sum_{i=1}^{N_{\text{properties}}} \alpha_i p_i(\mathbf{n}),$$

Which properties p_i contribute ∼linearly to execution time?

- Data motion
- Arithmetic
- Synchronization
- Launch overhead

Modeling Execution Time

$$T_{\text{wall}}(\mathbf{n}) \approx \sum_{i=1}^{N_{\text{properties}}} \alpha_i p_i(\mathbf{n}),$$

Which properties p_i contribute ∼linearly to execution time?

- Data motion
- Arithmetic
- Synchronization
- Launch overhead

How do we automatically gather necessary kernel statistics?

- Examine instructions and domains inside loo.py kernel

Modeling Execution Time

$$T_{\text{wall}}(\mathbf{n}) \approx \sum_{i=1}^{N_{\text{properties}}} \alpha_i p_i(\mathbf{n}),$$

Which properties p_i contribute ∼linearly to execution time?

- Data motion
- Arithmetic
- Synchronization
- Launch overhead

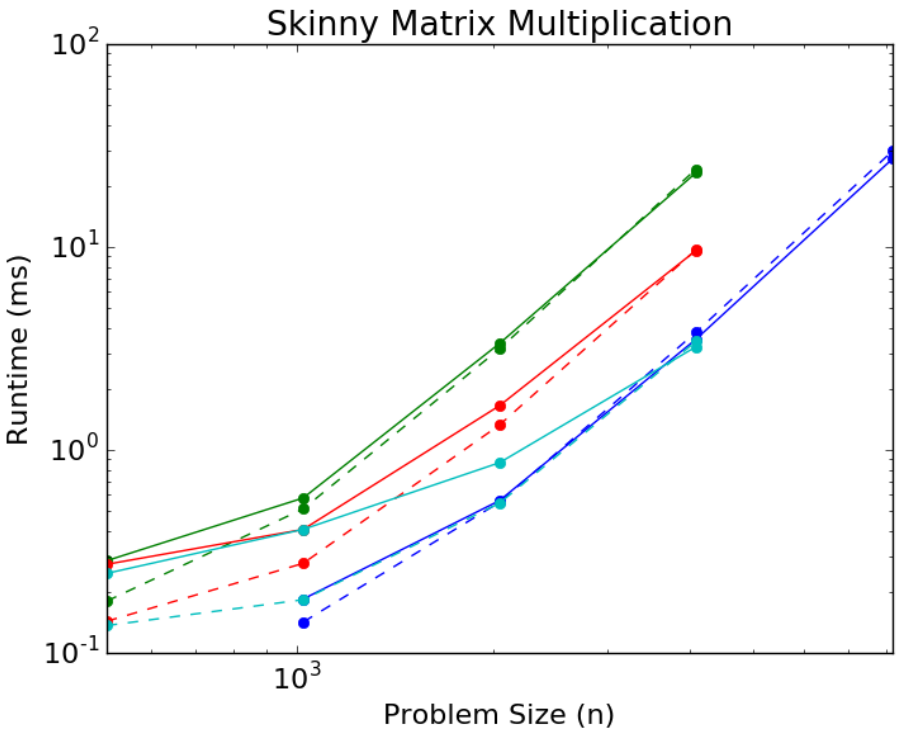
How do we automatically gather necessary kernel statistics?

- Examine instructions and domains inside loo.py kernel

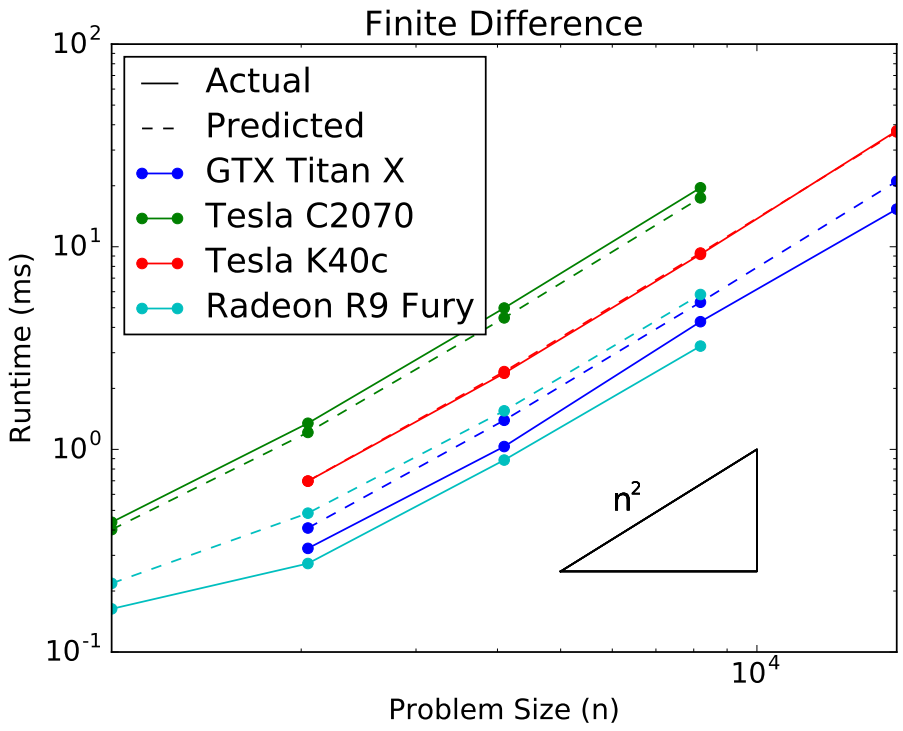
How do we determine hardware-specific property weights α_i?

- Run set of carefully chosen measurement kernels to calibrate

Model Accuracy (general)

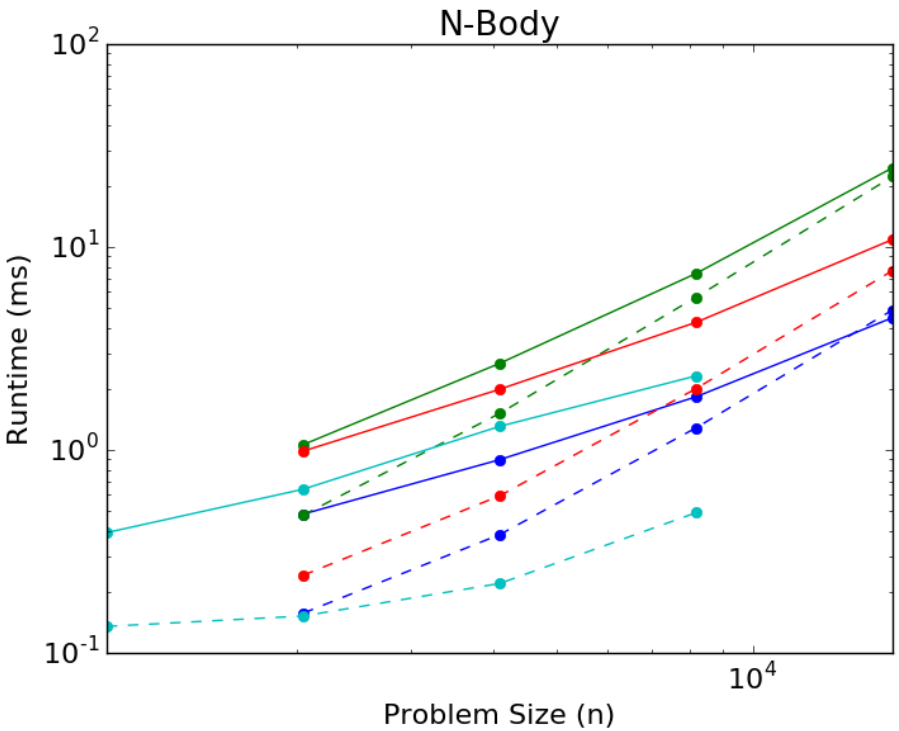


Geo-mean Error	
Titan X	0.08
C2070	0.10
K40c	0.13
R9 Fury	0.28
Overall	**0.13**

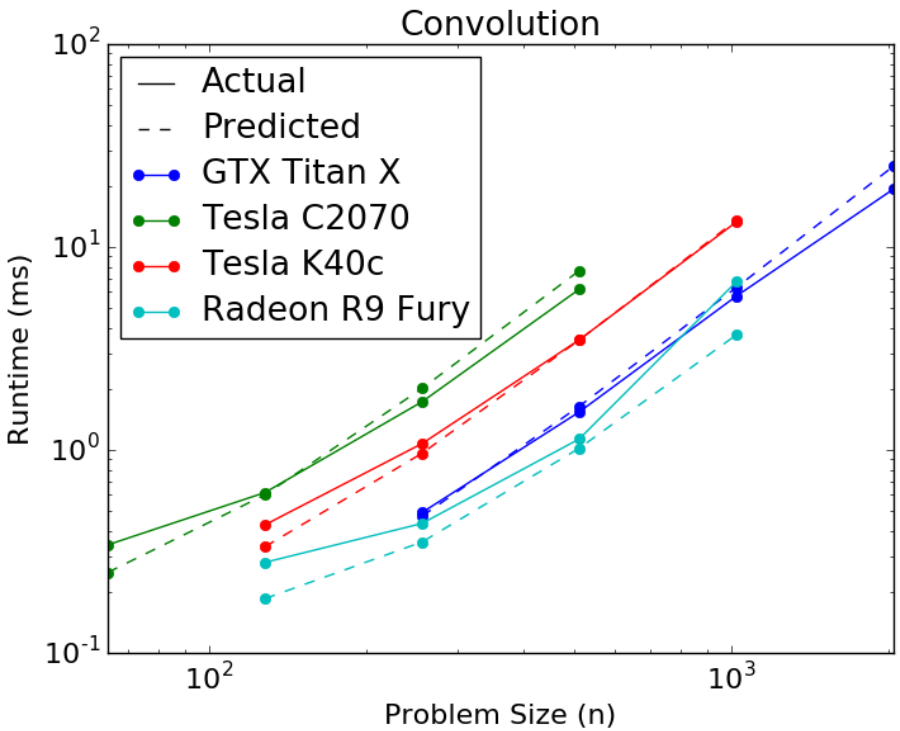


Geo-mean Error	
Titan X	0.30
C2070	0.10
K40c	0.01
R9 Fury	0.63
Overall	**0.11**

Model Accuracy (general)

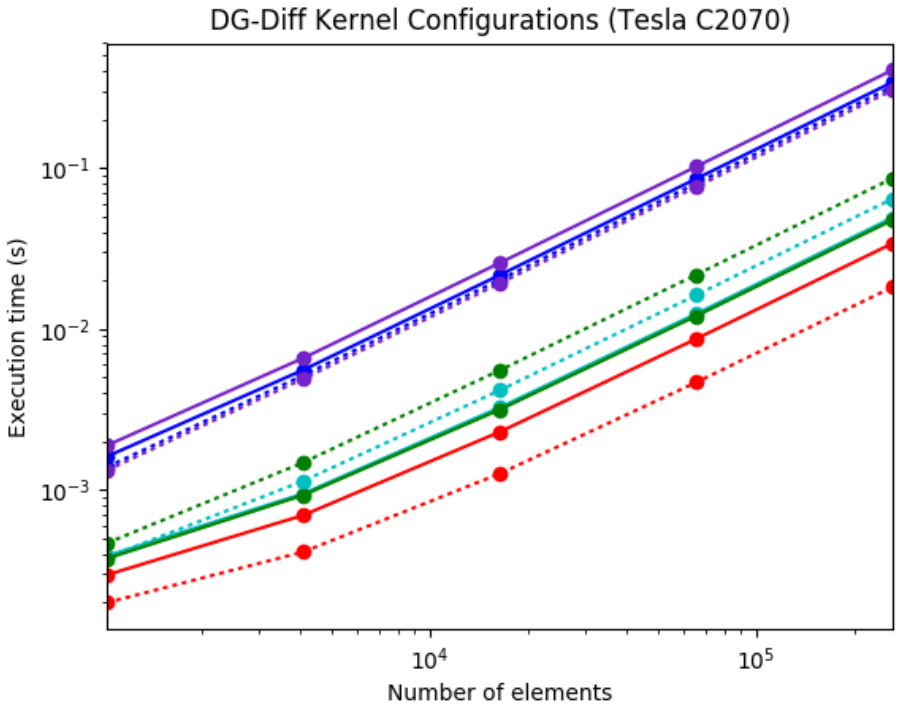


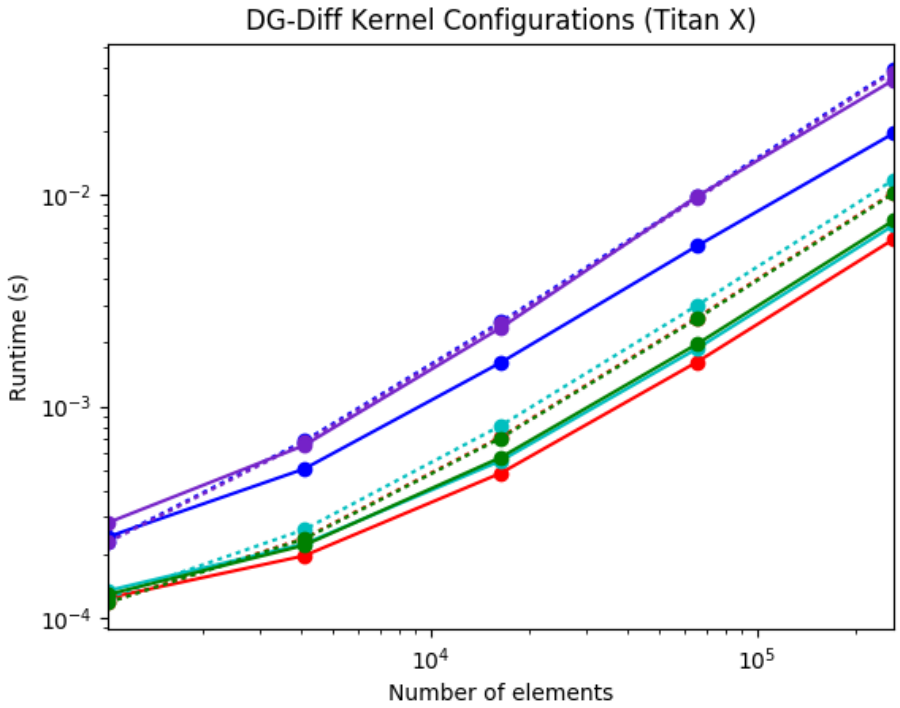
Geo-mean	Error
Titan X	0.32
C2070	0.27
K40c	0.54
R9 Fury	0.76
Overall	**0.43**



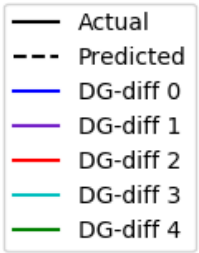
Geo-mean	Error
Titan X	0.10
C2070	0.13
K40c	0.03
R9 Fury	0.23
Overall	**0.10**

Model Accuracy (DG Differentiation Kernel)



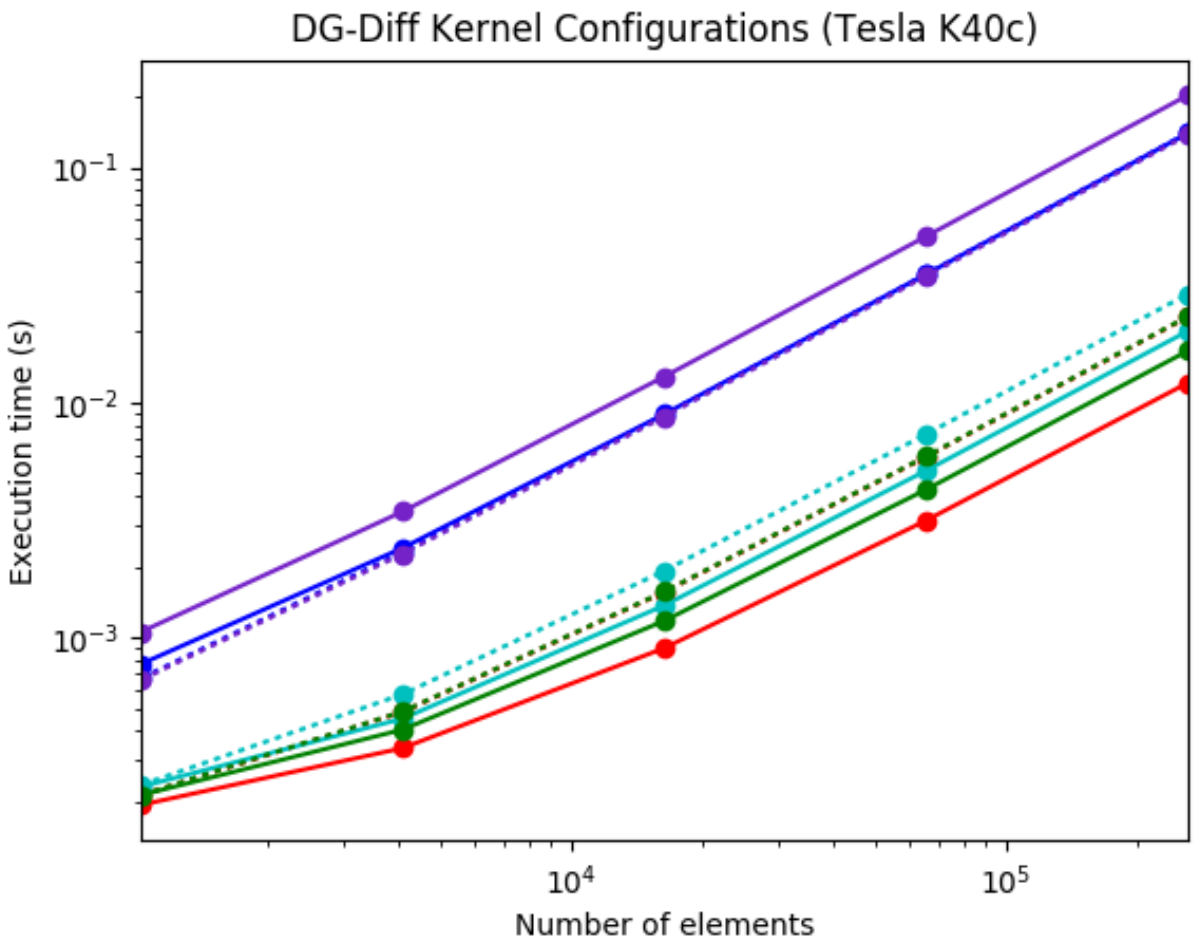


Geo-mean Error	
Overall	**0.24**



Geo-mean Error	
Overall	**0.20**

Model Accuracy (DG Differentiation Kernel)



Geo-mean Error	
Overall	**0.12**

Outline

Conclusions

- Hardware-specific implementation decisions affect execution time of DG code; tuning code by hand is tedious and slow
- **Grudge** takes symbolic description of PDE, produces DAG of tasks, and then "compiles" DAG into OpenCL code
- Grudge is built on **Loo.py**, which allows kernel reconfiguration at execution time and automated gathering of kernel statistics
- Our **cross-hardware performance model** accounts for Loo.py kernel execution times with accuracy sufficient to determine which of two kernels is faster in many (not all) cases
- Our model can be evaluated quickly enough to allow runtime performance predictions, which we intend to use for runtime performance tuning

Image Credits

- Question Mark: sxc.hu/svilen001
- ?/! Marks: sxc.hu/svilen001