HPC for Idealists with Deadlines: Pragmatic Abstractions for High Performance

Andreas Kloeckner

University of Illinois

November 16, 2022

ILLINOIS

Acknowledgments

Kaushik Kulkarni
(UIUC)

Matt Wala
(UIUC $\rightarrow$ Apple)

James Stevens
(UIUC, grad. 2021)

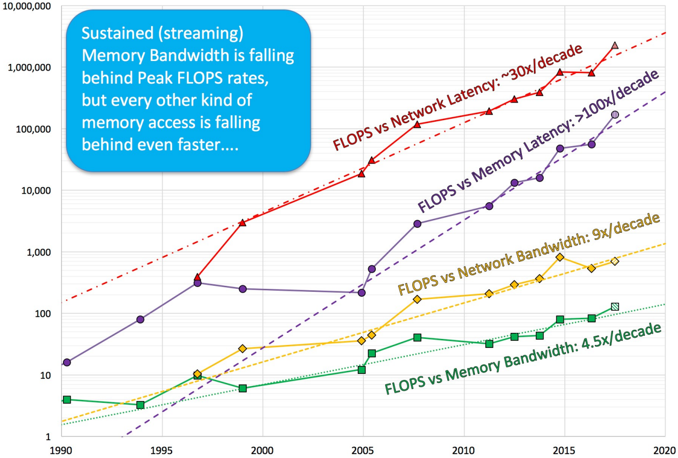
Funding:

- NSF (OAC-1931577, SHF-1911019)
- DOE (DE-NA0003963)

Outline

Outline

“Programming HPC Machines is Hard”



[McCalpin, Memory Bandwidth and System Balance in HPC Systems, SC16]

CPUs, GPUs: all subject to similar design pressures

HPC: What do you mean?

Not:

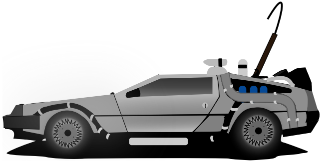
- 'Go-fast stripes' / Black-box / 4,000$\times$ faster

Instead:

- Build a quantitative understanding of what is possible (modeling)
- Iteratively approach that limit
  - Be an active participant
  - Expect some exposed wiring: understanding required
  - Use modeling as a guide

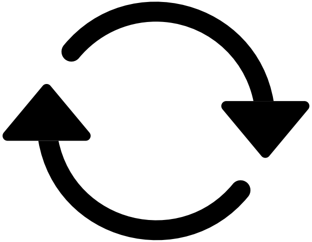
In this talk: Ideas and tools to...

- increase human effectiveness and efficiency
- help with separation of concerns
- help focus on the core issues



[OpenClipart / raulxav]

The Case for Code Transformation



[Bootstrap Icons]

Goals:

- Separation of concerns:
  additive rather than multiplicative effort
- Conciseness: code is the enemy
- Abstraction:
  not specifying details prematurely is a virtue

Approach:

- Program is a data structure
- Start with 'math'
- Gradually add detail
- Annotations at most descriptive, not prescriptive

As opposed to:

- Directives (a la OpenMP/OpenACC)
- Libraries

The Case for Just-in-Time Compilation

[Bootstrap Icons]

- What is ‘compile time’?
- At runtime is when you have the most information
  - Target device
  - Desired problem
- JIT gives ability to specialize for available knowledge
- Avoids false trade-off beetween generality and cost (“abstraction penalty”)
- Challenge: JIT cost must remain under control
  - At least: *Caching* easily avoids *repeated* expense

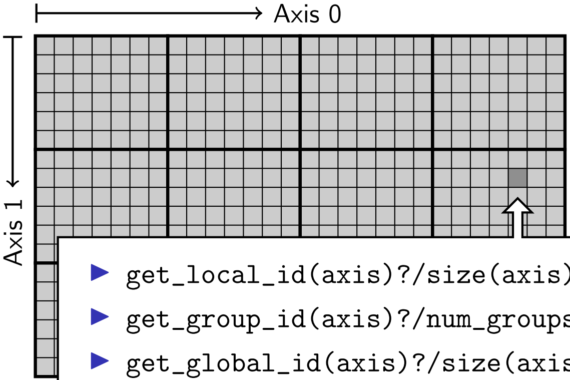
The Case for OpenCL

- Host-side programming interface (library)
- Device-side programming language (C)
- Device-side intermediate repr. (SPIR-V)

- Same compute abstraction as everyone else (focus on low-level)
- Device/vendor-neutral
  - On current and upcoming leadership-class machines
  - Will run even with no GPU in sight (e.g. Github CI)
- Just-In-Time compilation built-in
- Open-source implementations (Pocl, Intel GPU, AMD*, rusticl, clover)
- Mostly retain access to vendor-specific libraries/capabilties

[Khronos Group]

Wrangling the Grid

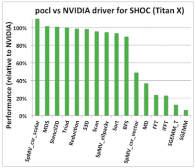


- `get_local_id(axis)?/size(axis)?`
- `get_group_id(axis)?/num_groups(axis)?`
- `get_global_id(axis)?/size(axis)?`

`axis=0,1,2,...`

Uncooperative vendor?

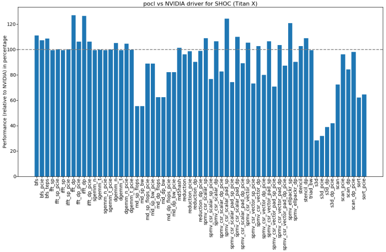
- OpenCL commoditizes compute
- Not universally popular with vendors
- Not an unchangeable fate



[http://portablecl.org/cuda-backend.html]

pocl-cuda:

- Based on `nvptx` LLVM target from Google
- Started by James Price (Bristol)
- Maintained by a team at Tampere Tech U
- We at Illinois helped a bit
- LLVM keeps improving
- Possible to talk to CUDA libraries
- Allows profiling



[http://portablecl.org/pocl-1.6.html]

The Case for Python

Frees up mental bandwidth. . .

for the *actually* difficult bits

How?

- Not shiny, not exciting
- No/few distractions
  - Duck typing, automatic memory management
- Emphasizes readability
- Rich ecosystem of sci-comp related software
- Good for gluing: less reinventing
- Easy to deploy
- ‘Fast enough’ for logistics and code generation

[python.org]

PyOpenCL

PyOpenCL has

- Direct access to low-level OpenCL
  - Efficiency-minded: compiler cache, kernel enqueue
  - Made safe for use with Python
    (e.g. ‘nanny events’, deletion semantics)
- A bare-bones numpy-like array type
  - Parallel RNGs, indexing
  - Numpy-like, but limited broadcasting, most operations are 1D
- Foundational algorithm templates
  - Reduction, scan, sort (radix, bitonic), unique, filter, CSR build

https://github.com/inducer/pyopencl Also: PyCUDA

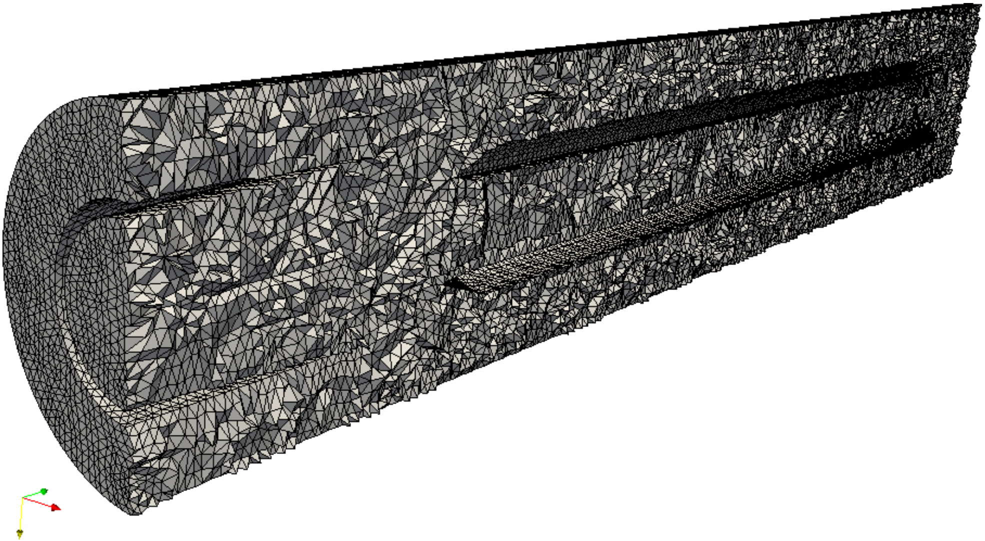
[Khronos Group, python.org]

Demo: PyOpenCL

https://github.com/inducer/pyopencl

Outline

Finite Elements: Meshes



[Matt Smith, UIUC]

Finite Element Action: Overview

Math: $\triangle u = f$ becomes

$$\int_\Omega \nabla u \cdot \nabla \psi dx = \int_\Omega f \psi dx$$

UFL (via Firedrake[1]):

```
a = inner(grad(u), grad(phi)) * dx
L = inner(f, phi) * dx
solve(a == L)
```

Computational kernel (for one DOF $\sim\in$ one element):

$$a_i = \sum_{j=1}^{N_q} w_j \partial\psi_i(x_j) \left(\sum_{k=1}^{N_{\text{DoF}}} u_k \partial\phi_k(x_j) \right)$$

Goal: Get this onto a GPU, generically

[1] David Ham et al., https://firedrakeproject.org

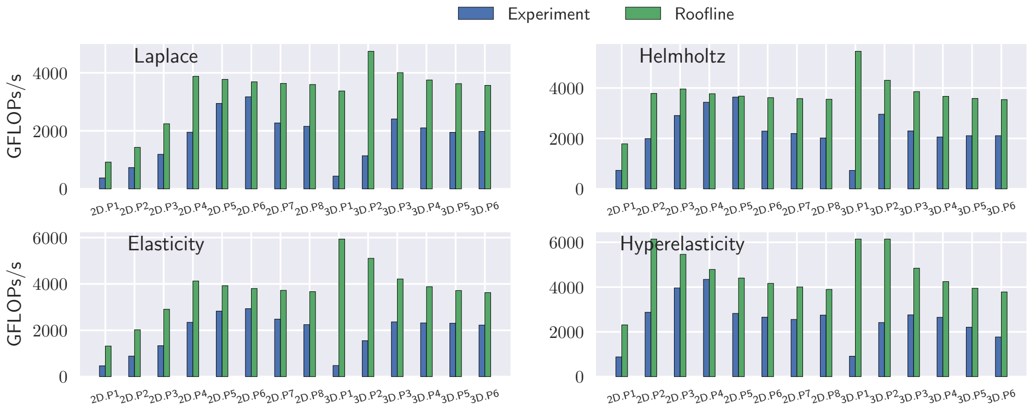
Finite Element Action: Workload Variation

- Dimension (2D, 3D)
- FE approximation spaces (CG, DG, BDM, RT, . . .)
  - also composed via product (often 'mixed') spaces
- Variational forms (e.g. Stokes):

```
a = (inner(grad(u), grad(v)) - p * div(v) + div(u) * q)*dx
L = inner(Constant((0, 0)), v) * dx
```

- Varying polynomial degrees

Results Preview



[Kulkarni, K, in prep.]

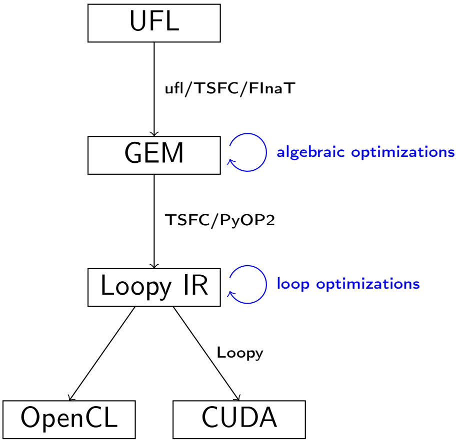
Approach Overview

```
from firedrake import *
set_offloading_backend(cuda)

#...(define mesh, function spaces)

a = dot(grad(u), grad(v))*dx
L = f*v*dx
sp = {"mat_type": "mat_free"}

with offloading():
    solve(a == L, s,
          solver_parameters=sp)
```



- using an approximate cost model
- to prune an autotuning search space

Outline

Kernel IR: Design Aspects

Single shared medium, must:

- Express computational intent with little information loss
- Enable program transform tools
- Be human-readable to enable performance work

Needs:

- Metadata capture for transformation targeting
- Precise dependency tracking
- Precise hardware mapping
  (meets CL/CUDA machine model, specified, no heuristics!)

Community IR innovation:

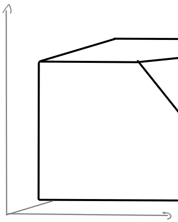
- *C. Lattner, J. Pienaar* "MLIR Primer: A Compiler Infrastructure for the End of Moore's Law." (2019).
- *R. Baghdadi et al.* "Tiramisu: A polyhedral compiler for expressing fast and portable code." Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization. IEEE Press. (2019)
- *T. Ben-Nun et al.* "Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs.", SC '19. (2019)

What and why: polyhedral?

Loop nest

```
do i = 1,n
    do j = 1,n
        do k = 1,n−i−k
            A(i,j,k) = ...
            B(i,j,k) = ...
        end do
    end do
end do
```

Polyhedron



`{[i,j,k]:0 <= i,j < n and... }`

S. Verdoolaege "isl: An integer set library for the polyhedral model." International Congress on Mathematical Software. Springer, Berlin, Heidelberg, 2010

https://github.com/indcuer/islpy

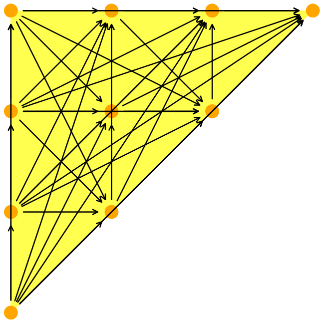
Not just sets: also dependencies

Loop domain: $\{(i,j) : 0 \le i,j \le 4 \wedge i \le j\} \subset \mathbb{Z}^2$

Parametric loop domain: $n \mapsto \{(i,j) : 0 \le i,j \le n \wedge i \le j\} \subset \mathbb{Z}^3$

Dependencies: $\{((i,j),(i',j')) : \dots\} \subset \mathbb{Z}^4$

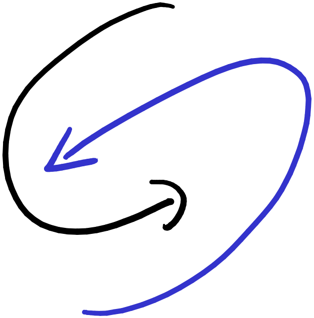
$+$ parameter: $n \mapsto \{((i,j),(i',j')) : \dots\} \subset \mathbb{Z}^5$



- Way to represent
  - sets of integer tuples
  - graphs on sets of integer tuples

  and operate on them:
  Π, $\cap$, $\cup$, $\circ$, $\subset^?$, $\setminus$, min, lexmin
- parametrically
- need decidability: (quasi-)affine expr.
  - no: $i \cdot j$, $n \bmod p$
  - yes: $n \bmod 4$, $4i - 3j$

Code Transforms



- Unroll
- Stride changes (Row/column/something major)
- Prefetch
- Precompute
- Tile
- Reorder loops
- Fix constants
- Parallelize (Thread/Workgroup)
- Affine map loop domains
- Texture-based data access
- Loop collapse

Even More Code Transforms

- Kernel and Loop Fusion
- Scans and Reductions
- Global Barrier by Kernel Fission
- Explicit-SIMD Vectorization
- Reuse of Temporary Storage
- SoA $\rightarrow$ AoS
- Buffering, Storage substitution
- Save flops using Distributive Law
- Arbitrary nesting of Data Layouts
- Realization of ILP
- Array compression/reindexing [Seghir, et al. '06]

Automatic Operation Counting

Can obtain *parametric*, piecewise polynomial operation counts/bounds[2], directly from IR:

- Flops performed $\approx \sum_{\text{Statement } s} |\text{Domain}(s)| \cdot \text{flops}(s)$
- Mem. Ops performed $\leq \sum_{\text{Statement } s} |\text{Domain}(s)| \cdot \text{Mem. Ops}(s)$
- Mem. Ops performed $\geq \sum_{\text{Variable } v} |\text{Access Footprint}(v)|$

Can use these for computer-aided performance model fitting[3].

[2] Verdoolaege et al. 2007
[3] Stevens, K 2020

Demo: Loopy

https://github.com/inducer/loopy

Loopy in the context of the FEM action

$$a_i = \sum_{j=1}^{N_q} w_j \partial\psi_i(x_j) \left(\sum_{k=1}^{N_{\text{DoF}}} u_k \partial\phi_k(x_j) \right)$$

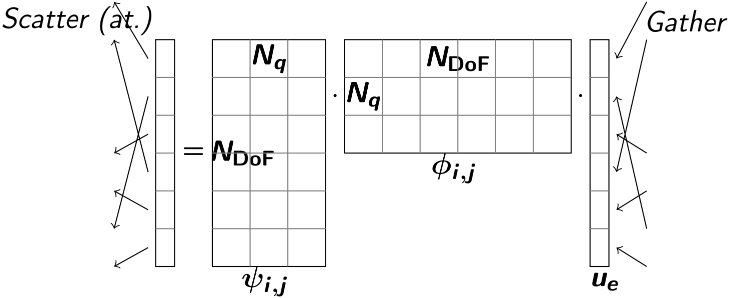
```
knl = lp.make_kernel(
    "{[e,i,j,k]: 0<=e<nelements and 0<=i,k<ndofs and 0<=j<nq}",
    """
    quad(e, j) := sum(k, u[k,e] * phi[k, j])
    a[e,i] = sum(j, w[j] * psi[i,j] * quad(e, j))
    """)
```

Transformations (illustrative):

```
knl = lp.split_iname(knl, "e", 128)
knl = lp.tag_inames(knl, {"e_outer": "g.0"})
```

Outline

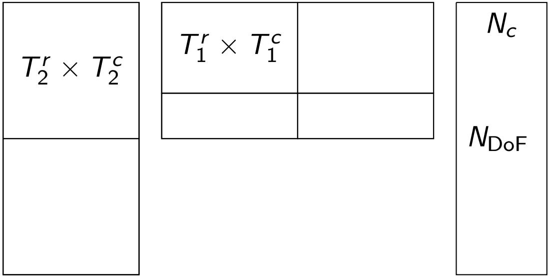
Workload



- N_q: #Quadrature pts.
- N_{DoF}: #local DoFs
- Geometric factors, quadrature weights not shown

Transform Approach

- Tile the accesses to the matrices as $T_1^r \times T_1^c$, $T_2^r \times T_2^c$
- Group computation of N_c cells to be operated on by a workgroup
- Inner products within each tile divided among N_t SIMT work items
- Block size $= N_c N_t$ SIMT work items



- N_{DoF}: Number of local DoFs.
- N_c: Cells in a block
- $T_{1,2}^{r,c}$: Tile sizes

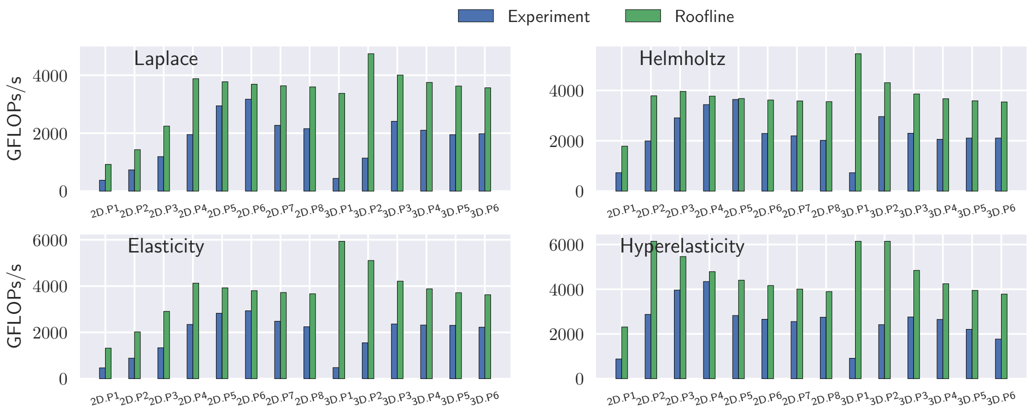
Cost Model and Roofline

$$t_{\text{heur}} = \frac{\text{Total Global Memory accesses}}{\beta_{\text{glob}}^{\text{model}}} + \frac{\text{Total Local Memory accesses}}{\beta_{\text{local}}^{\text{model}}}$$

- $\text{AI}_{\text{global}}$: Arith. intensity wrt global memory access count
- AI_{local}: Arith. intensity wrt local memory access count

$$\mathcal{F}_{\text{roofline}} = \min\left(\text{AI}_{\text{global}}\beta_{\text{global}}^{\text{peak}},\ \text{AI}_{\text{local}}\beta_{\text{local}}^{\text{peak}},\ \mathcal{F}_{\text{peak}}\right)$$

Performance evaluation (Titan V)



[Kulkarni, K, in prep.]

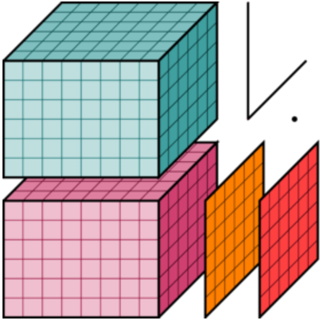
Statistical Performance Achievability Study

"test cases" are "winners" across settings (2D/3D, PDEs, poly orders)

[Kulkarni, K, in prep.]

Outline

Improving the Scientist Interface



[XArray]

Loopy intermediate representation:

- *Somewhat* user-friendly, some idiosyncrasies
- *Still* specifies some detail prematurely

What might a better scientist interface look like?

(Not a new) Idea: numpy-like multi-dimensional arrays

- E.g. JAX, Theano, Tensorflow, ...

Specialize:

- *Undetermined* data layout
- *Immutable* once created
- Allows building an array-valued DFG

  $\rightarrow$ represent entire workload as one giant expression

Pytato: Demo

https://github.com/inducer/pytato

Stages of a Computation

Stage 1: Capture an Array DFG *Pytato*

- Goal: Build an Array-Valued Data Flow Graph (DFG)
  - By tracing execution of a *Numpy*-ish array program
- Use Lazy Evaluation to do so:
  - Feed in (symbolic) placeholder data
  - Return an opaque value that 'remembers' what was done

Stage 2: Transform the DAG *Pytato*

- E.g. fold constants, apply math simplifications

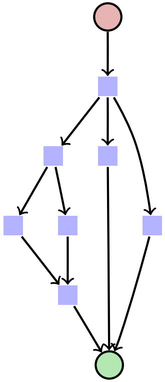
Stage 3: Rewrite to Scalar IR *Pytato* → *Loopy*

- Introduce time, memory, loops

Stage 4: Scalar IR Transformations *Array Context* and *Loopy*

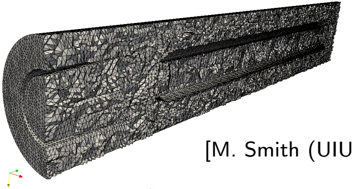
- E.g. parallelize, optimize for the memory hierarchy

Stage 5: Emit Target Code *Loopy* → *OpenCL*

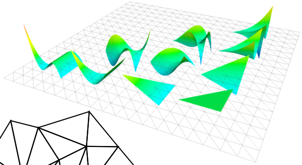


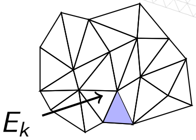
B = f(A)	C = g(B)
E = f(C)	F = h(C)
G = s(E,F)	P = p(B)
Q = q(B)	R = r(G,P,Q)

What Workload?



[M. Smith (UIUC)]





$$\partial_t q + \nabla \cdot F(q) = 0$$

Test with ϕ, integrate by parts:

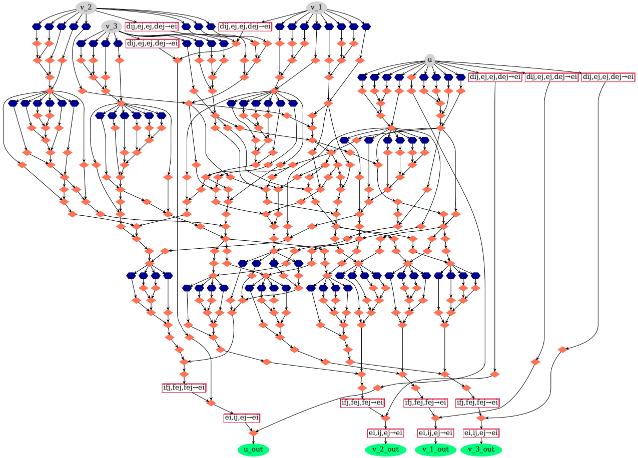
$$0 = \int_{E_k} q_t^k \phi dx - \int_{E_k} F \cdot \nabla \phi dx$$
$$+ \int_{\partial E_k} (F \cdot \hat{n})^* \phi dx$$

In matrix form:

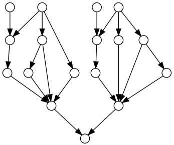
$$0 = \mathcal{M}^k \partial_t u^k - \sum_{\nu} \mathcal{S}^{k,\partial_\nu} [F(u^k)]$$
$$+ \sum_{A \subset \partial E_k} \mathcal{M}^{k,A} (\hat{n} \cdot F)^*$$

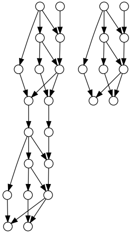
Multi-species, reacting, heat transfer, materials, ...

A View of the DFG



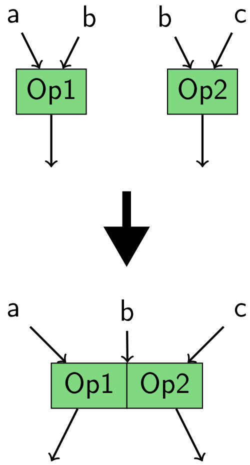
DAG Capture





- How to capture result reuse?
  - Imperative codes must store intermediate results, breeds global state
    (often a significant challenge in science codes)
  - We can recompute with impunity: simpler app code
  - Approach: Recognize and collapse repeated DAG segments via hashing
  - Future work: Allow asserting no recomputation via metadata
- Issue: Repeated sub-DAGs (distinct 'inputs') increase (transform, code gen, execution) cost
  - Salient example: *Interior fluxes* repeated for each neighbor rank

Fusion



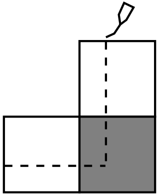
- Need a more global view, including data flow between kernels to enable fusion
- Benefits:
  - Eliminate memory traffic
  - Reduce control overhead

Realized in two stages:

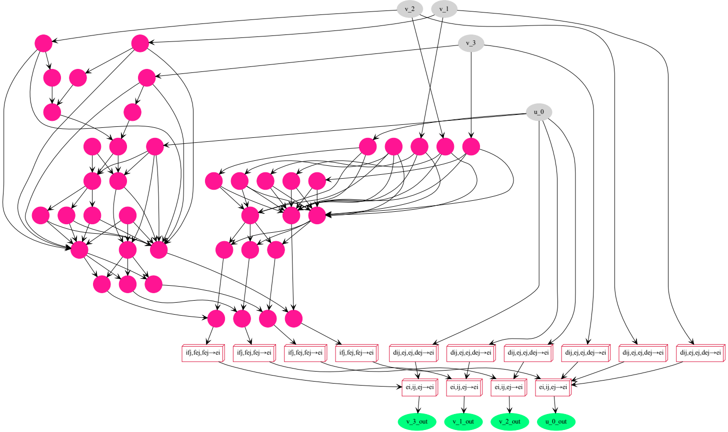
- Starting point: entirely abstract view of the computation. Essentially: a giant formula.
  - Which array values should be stored?
  - Approach: Materialize if ≥ 2 predecessors, successors
- Which temporary arrays can be eliminated?
  - Approach: Graph-based array contraction building on [Kennedy, et al. '93]

Transformation and Metadata

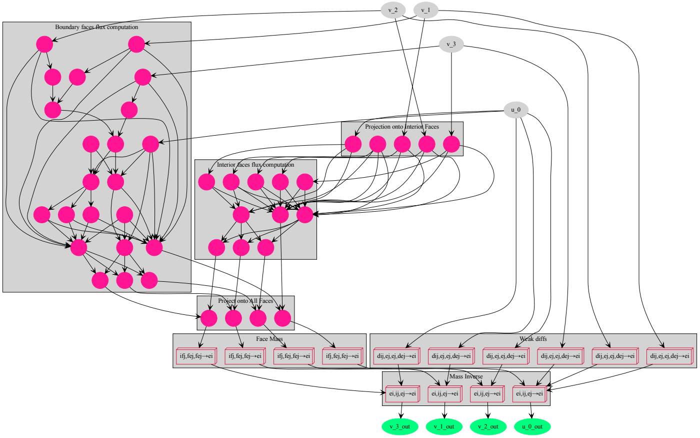
- Transform strategy is application-specific, relies on metadata—from where?
  - Approach: sparse annotations applied by infrastructure (meshmode, grudge) are propagated and suffice to fully 'type' array axes
- Loops with data reuse need storage management, tiling for near-roofline performance
  - Perform optimization at the level of a 'fused Einstein summation'
  - Einstein summation: $\sum_k a_{ik} b_{kj} \to$ `ik,kj->ij`
  - 'Fused:' Consider groups without result dependencies as one unit, typically sharing input data as one unit
  - Build database of transform templates, match via normal form, use stored optimizations



DFG after ‘Vertical Fusion’



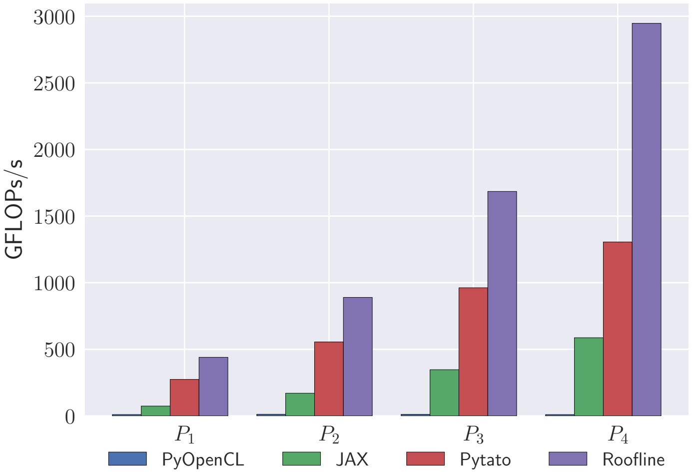
DFG after ‘Horizontal Fusion’



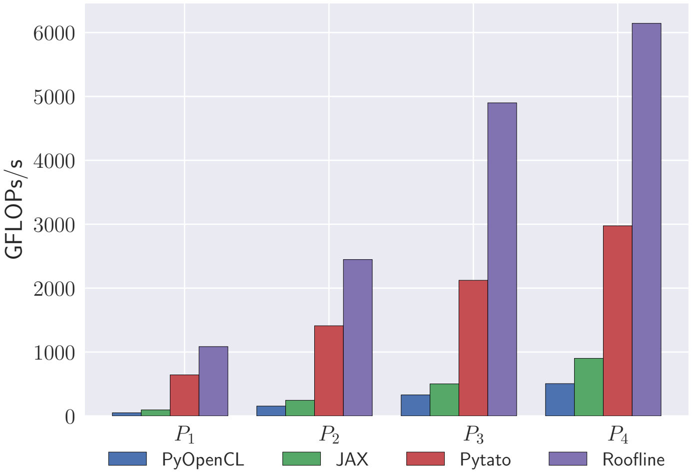
Experimental Setup

- Nvidia Titan V
  - Peak Double prec. FlOps: 6144 GFlOps/s
  - Peak bandwidth: 652.8 GB/s
- $p \in \{1, 2, 3, 4\}$, 3D tetrahedra
- Elements in mesh: 200K (for high orders), 700K (for lower orders)
- OpenCL Implementation: PoCL-CUDA (v1.8)
  - Performance roughly equivalent to Nvidia CL
- Roofline $= \min\left(\text{Device's Peak FlOps/s}, \frac{\text{Kernel FlOps}\cdot\text{Device's Peak Bandwidth}}{\text{Memory Footprint}}\right)$
  - Uses the *Fused-Einsum* kernel granularity to model Global Memory Footprint

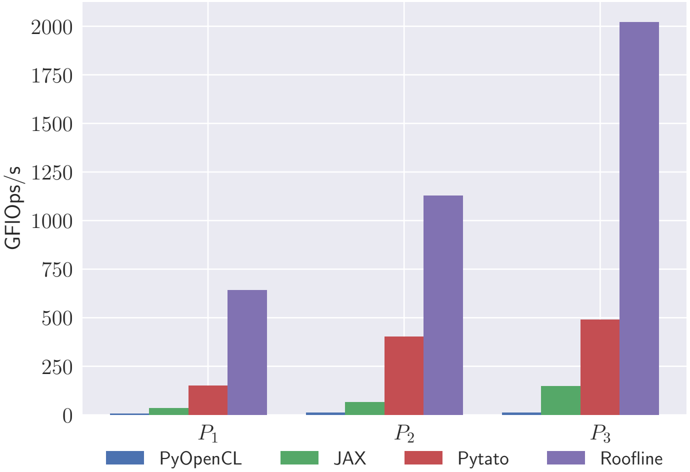
Results: Wave Equation



Results: Maxwell Equations



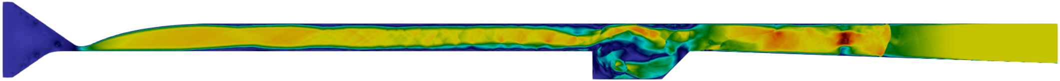
Results: Compressible Navier-Stokes



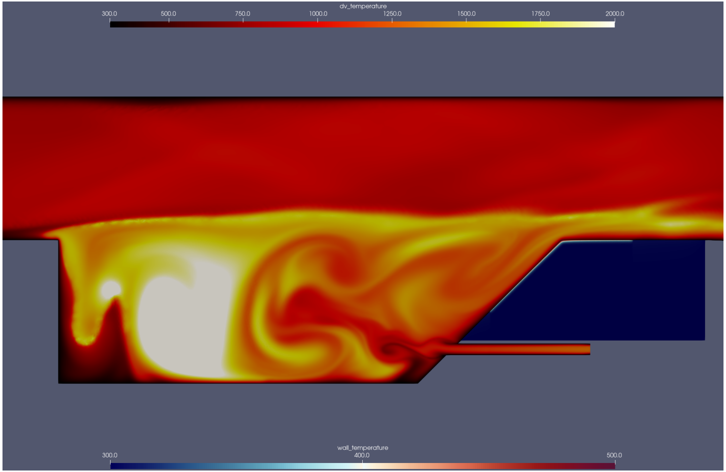
Scramjet Application

Model of a supersonic combustion ramjet (scramjet):

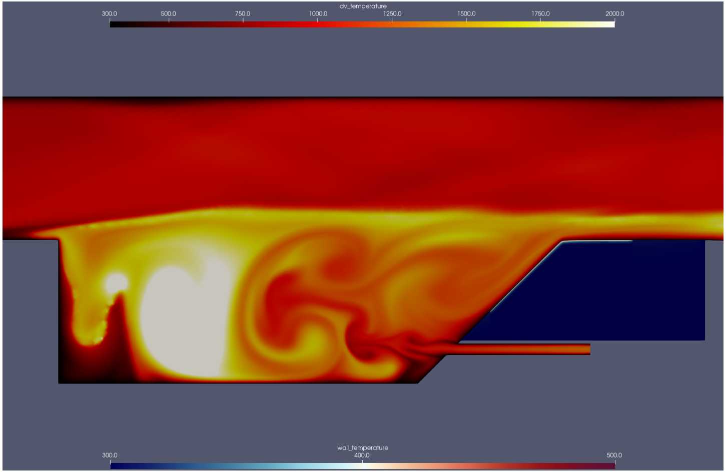
- supersonic with combustion
- fuel injector, flame-holding cavity
- isolator, nozzle
- inlet: not yet



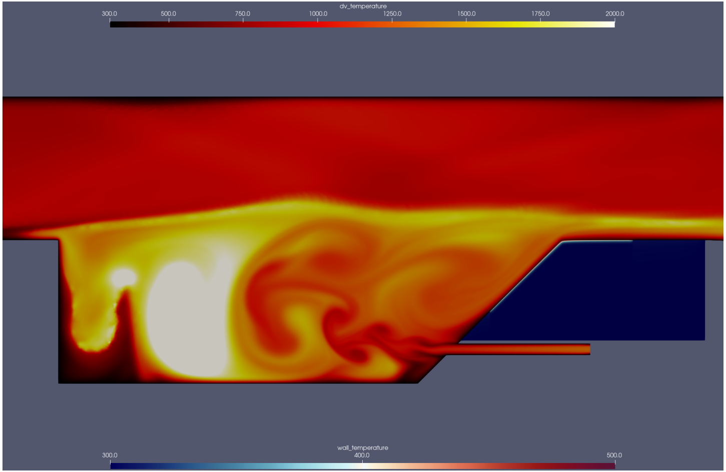
Combustion Movie



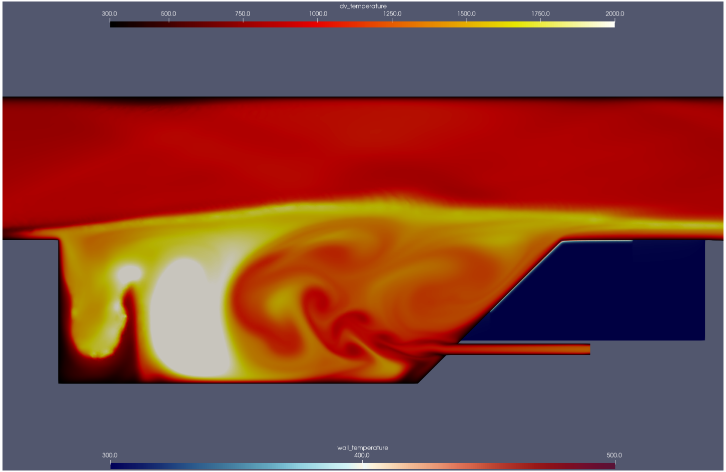
Combustion Movie



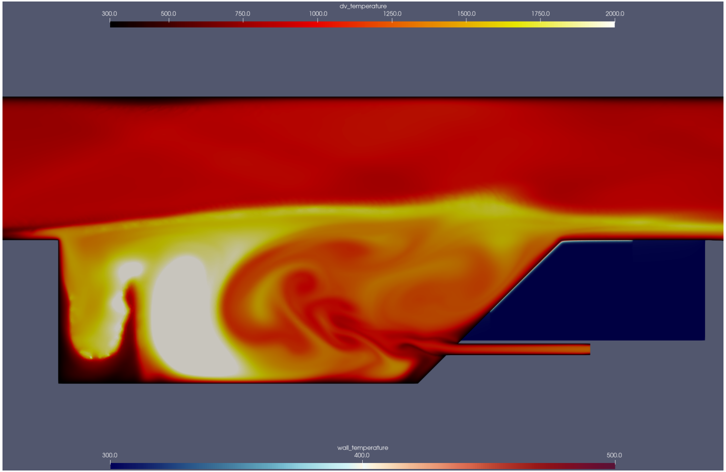
Combustion Movie



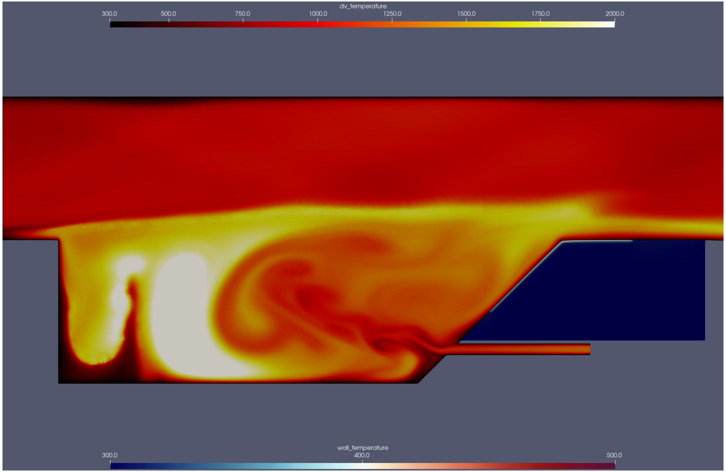
Combustion Movie



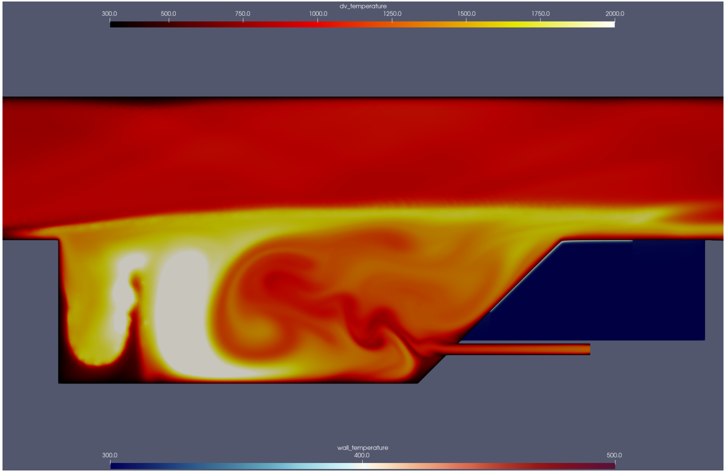
Combustion Movie



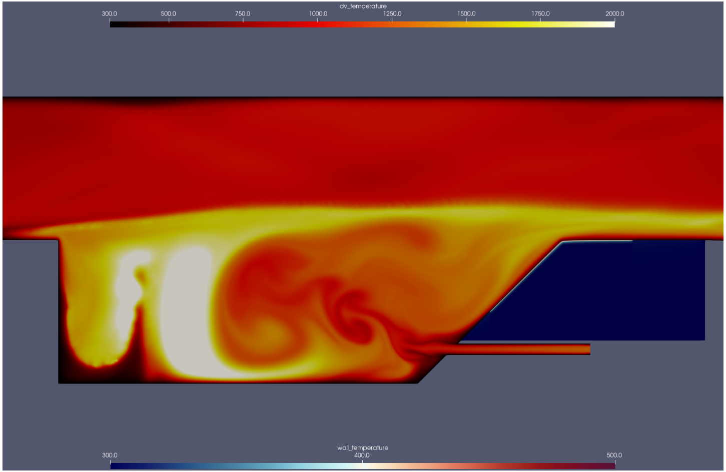
Combustion Movie



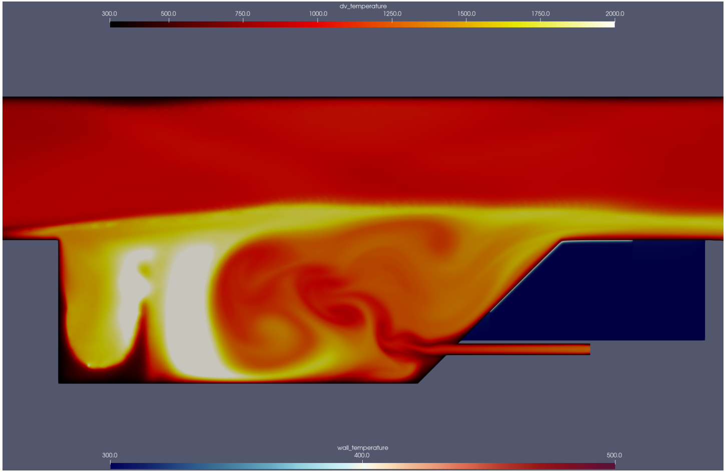
Combustion Movie



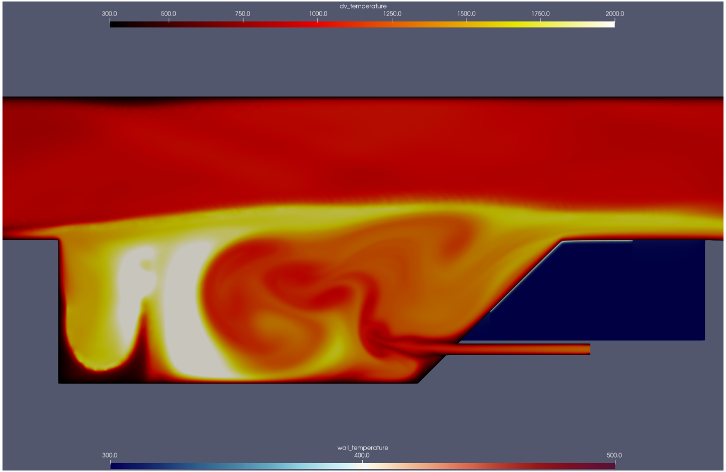
Combustion Movie



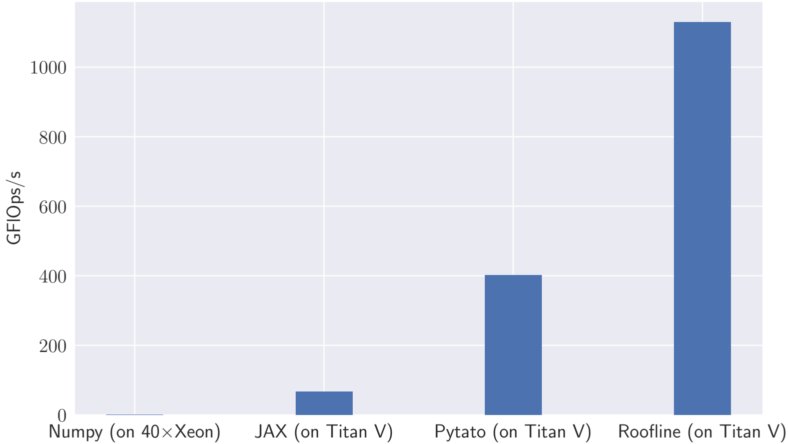
Combustion Movie



Combustion Movie



Performance on a Proxy for the Application (3D $p = 2$)



Outline

Conclusions

GPU Finite Elements

- Simple, ~ generic analytical model achieves effective pruning
- At least 50% roofline for 70% of test cases
  - i.e. attains considerable generality
- Tuning strategy relatively low-cost, no user involvement needed
- Transforms permit separation of concerns between
  - domain-specific compiler and
  - performance work

DG Array DFG

- A stark reminder of the value of domain knowledge
- Array DFG capture:
  quite mature, very general
- DG transformation parts:
  still quite WIP
- Additional part: distributed memory (via send/recv nodes in rank-local DAG)

- `https://github.com/inducer/{pyopencl,loopy,pytato}`