MIRGE: Math $\rightarrow$ IR $\rightarrow$ Generation $\rightarrow$ Execution

Andreas Kloeckner

University of Illinois

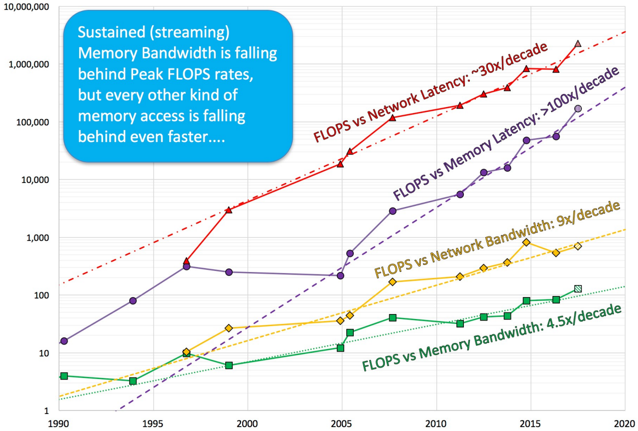
January 18, 2024

Outline

MIRGE

Code-Along

“Programming HPC Machines is Hard”



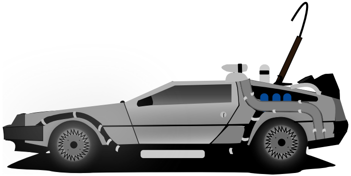
[McCalpin, Memory Bandwidth and System Balance in HPC Systems, SC16]

CPUs, GPUs: all subject to similar design pressures

HPC: What do you mean?

Goal:

- Build a quantitative understanding of what is possible
  - I.e. use modeling, supported by tools
- Iteratively approach that limit, with human involvement
  - I.e. not a black-box compiler
  - Expect some exposed wiring: understanding required
  - Use modeling as a guide

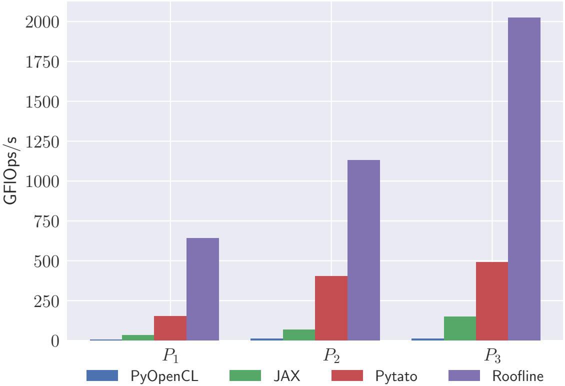


[OpenClipart / raulxav]

MIRGE: Ideas and tools to...

- increase human effectiveness and efficiency
- help with separation of concerns

A Glimpse of Some Results



(Simplicial DG for a Compressible Navier-Stokes Operator on Titan V)

MIRGE: Stages of a Computation

Stage 1: Capture an Array DFG *Array Context* → *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 *Array Context* and *Pytato*

- E.g. fold constants, apply math simplifications

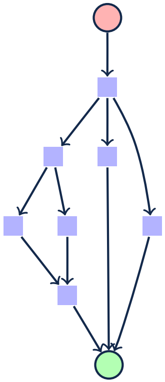
Stage 3: Rewrite to Scalar IR *Pytato* → *Loopy*

- Introduce time, memory, loops

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

- E.g. parallelize, loop/kernel fusion

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)

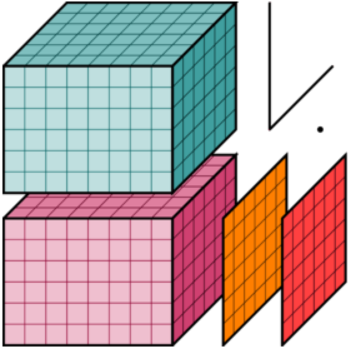
Numpy: Array programming

Numpy is an *array programming language*.

What can you do with that?

- `a + b - 3`
- `a[:, 4].reshape(10, 1) + b`
- Compute pairwise distances between point clouds
- `a[i]` where `i` is an array of indices
- `a>3`
- `np.where(a > 3, 0, 1)`
- `np.einsum("ij,j->i", a, b)`
- `np.sum(a, axis=0)`
- `np.concatenate((a, b), axis=0)`

If familiar: a little like 'fully vectorized Matlab'



[XRay Project]

User-Visible Restrictions (the "-ish" in numpy-ish)

[Bootstrap Icons]

- Data is computed lazily
  - "Looking at the data" costly: ask expliclitly (`freeze`)
  - Fine: `np.where(x > 15, 1, 0)`
  - Not fine: `if x[0] > 15:  print("BAD")`
- "In-place" modification is not allowed
  - Once created, an array is constant
- Looping over an array is very costly
  - Resulting DAG will be proportional to array size
- Does not encode memory layout (i.e. no stride trickery)
- For code with pre-recorded traces ("compiled"):
  - Python code is only run once
  - Needed for repeated tasks (e.g. time step)
  - *Cannot* look at data (run with placeholder arrays)

Numpy Switcheroo: Array Context

Replacing numpy:

- NOT: `import numpy as np` → `import mystuff as np`
- INSTEAD: `actx.np.zeros(...)`

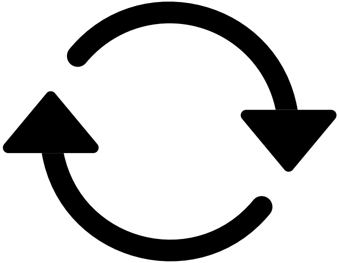
Why?

- 'Real' numpy used alongside, e.g. by supporting libraries
- Avoids `np.mystuff(...)`: The `numpy` namespace belongs to numpy.
  - Natural place for additional API: E.g. `actx.freeze()`
- Avoids global state for device selection (e.g. Jax)
- Can be subclassed by user to supply transform strategies

[Bootstrap Icons]

(`actx` is a user-controlled instance of a user-controlled subclass of `ArrayContext`.)

The Case for Code Transformation



[Bootstrap Icons]

- Program is a data structure
- Start with 'math' ($\approx$ `numpy`)
- Gradually add detail
- Annotations descriptive, not prescriptive

As opposed to:

- Directives (a la OpenMP/OpenACC)
- Libraries

Properties:

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

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

Loopy: A Glimpse

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

```
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:

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

`github.com/inducer/loopy`

In the Code-Along

Roadmap for the code-along:

- Let's code a mini *pytato*
  - Expression trees/graphs as program representation
  - Lowering to *loopy*
- Let's build a finite difference solver with the MIRGE stack
- Getting your feet wet with *Loopy*

Outline

MIRGE

Code-Along

Getting on the Jupyterhub

- Primary (NCSA)
  `https://ceesd.class.ncsa.illinois.edu/jupyter/`
  User / Password from paper snippets
- Fallback (Homebrew)
  `https://andreask.cs.illinois.edu/nuwest`
  User name: Pick your favorite! / Password: (announced if needed)

Building a Mini Pytato

Notebook: Mini Pytato

Lessons from Mini Pytato

- Graphs are an appropriate data structure for expressions
- A shape axis becomes a loop
- Processing graphs is necessarily recursive
- Naive handling of common subexpressions leads to exponential complexity

Array Comprehensions / `IndexLambda`

Observation: To define an array, just need

- shape
- a (scalar) expression for array entry `array[i,j]`.

Examples:

- A 10×5 array defined by $(i, j) \mapsto 3i + 5j$
- A 10×10 array defined by $(i, j) \mapsto \delta_{i,j}$
- A 10×10 array defined by $(i, j) \mapsto a[i, j] + b[i]$

Idea: Use that

- as a large part of the intermediate representation
- as a pathway toward code generation
  (many operations “lower” to scalar expressions)

Pytato vs Mini Pytato

- Computations with multiple results (`DictOfNamedArrays`)
- Constants (`DataWrapper`)
- Many more operators, functions
- Arbitrary shapes (including symbolic)
- Broadcasting
- Slicing, Indexing

- Reductions (e.g. sums over axes)
- `einsum`, matrix products
- Metadata ("tags") on arrays, axes
- Visualization
- Distributed compute
- "Call loopy" as an expression node

Let's code finite differences

Notebook: Finite Difference Code-Along

What is an array context?

- `actx.np`
- `actx.freeze` / `actx.thaw`
- `actx.np.zeros`
- `actx.from_numpy` / `actx.to_numpy`
- `actx.tag` / `actx.tag_axis`
- `actx.compile(f)`

What happens in `PytatoPyOpenCLACtx.compile(f)`?

Returns a function that

- once called, looks at arguments passed (which maybe array containers)
- replaces actx arrays with placeholders
- Calls `f` with those placeholders
- Take the resulting `pytato` DAG, feed to Loopy
- Lastly, call the generated loopy code with the passed arguments
  - Return results as actual data (`pyoepncl` arrays)
- If called again with arguments of matching type/shape:
  - do not call `f`
  - go straight to calling generated code

What happens in `PytatoPyOpenCLACtx.freeze`?

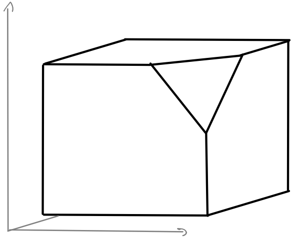
- Simple: build code to evaluate computation graph
  - Return result as actual data
- No placeholders, only `DataWrapper` (=constant) instances
  - `thaw`: package data in a `DataWrapper`
- Try to avoid redundant code generation
  - But: expensive! Always at least need to compare (and therefore, traverse!) graphs
- Potential gotchas
  - Freeze same graph again: redundant codegen, computation
  - Freeze superset graph: redundant codegen, computation

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

`github.com/inducer/islpy`

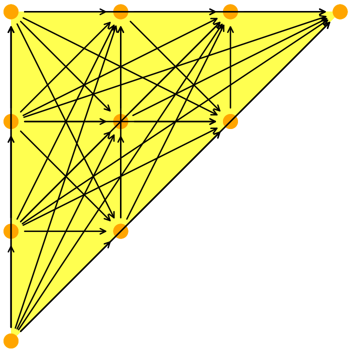
Not just sets: also dependencies

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: $n \bmod 4$, $4i - 3j$

A Taste of Loopy

Demo: A Taste of Loopy

What is an array container?

- A thing that can contain actx arrays and other array containers
- Allows "serialization" and "deserialization", i.e. generic traversals
- Allows nested data structures
- E.g.:
  - structure-like (`ConservedVars`, `TracePair`)
  - array-like (`DOFArray`, object array)
- Defined in `arraycontext`
- Works with many `ArrayContext` operations

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

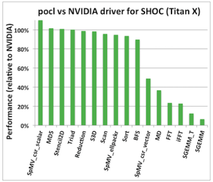
[Khronos Group]

Uncooperative vendor?

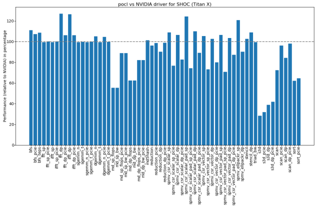
- OpenCL commoditizes compute
- Not universally popular with vendors
- Not an unchangeable fate

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/cuda-backend.html]



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

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]

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]