Part 6: Loopy

Andreas Klöckner

Computer Science · University of Illinois at Urbana-Champaign

Outline

Outline

Automating GPU Programming

High-performance programming can be a time-consuming trial-and-error process.

Obvious idea: Let the computer do it. How?

- One way: "Smart" compiler, "dumb" developer
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile
- Another way: "Smart" developer, "dumb" compiler
  - Error-prone
  - Expensive in developer time
  - User can use manual/automatic tuning

Automating GPU Programming

High-performance programming can be a time-consuming trial-and-error process.

Obvious idea: Let the computer do it. How?

- So compromise! Following: an idea of a compromise.
  - Heuristics are fragile
- Another way: "Smart" developer, "dumb" compiler
  - Error-prone
  - Expensive in developer time
  - User can use manual/automatic tuning

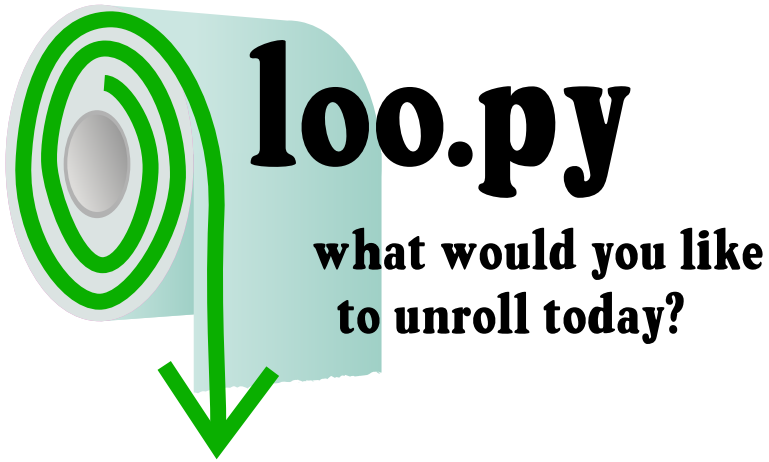
Setting the Stage

Idea: Create IR + library of transformations

- Start with math-y statement of the operation
- "Push a few buttons" to optimize for the target device
- Strongly separate these two parts

Philosophy:

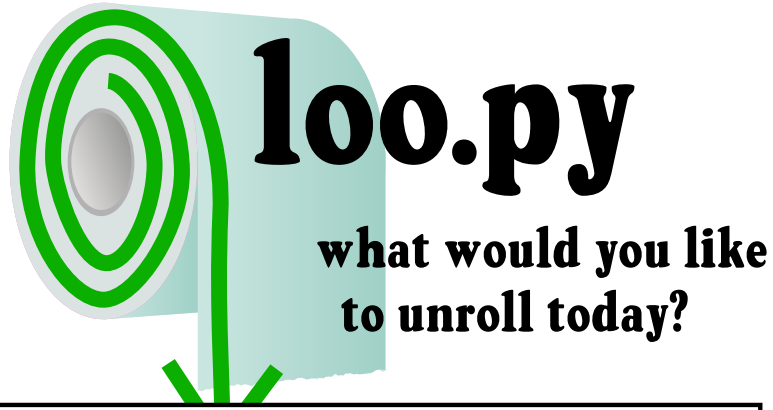
- Avoid "intelligence"
- User can assume partial responsibility for correctness
- Embedding in Python provides generation/transform flexibility



Setting the Stage

Idea: Create IR + library of transformations

- Start with math-y statement of the operation
- "Push a few buttons" to optimize for the target device
- Strongly separate these two parts



Philosophy:

- Avoid "intelli
- User can assu
  responsibility
- Embedding i
  generation/tr

Loopy is infrastructure.

Auto-tuners and domain-specific libraries are "above" loopy conceptually.

DEMO TIME

Capturing Variants

```
knl = ...

def variant_cpu(knl):
    knl = lp.split_dimension(knl, "i", 16*4096, outer_tag="g.0", slabs=(0, 1))
    knl = lp.split_dimension(knl, "i_inner", 16,
            inner_tag="unr")
    return knl

def variant_gpu(knl):
    knl = lp.split_dimension(knl, "i", 4*256, outer_tag="g.0", slabs=(0, 1))
    knl = lp.split_dimension(knl, "i_inner", block_size,
            outer_tag="unr", inner_tag="l.0")
    return knl

for variant in [variant_cpu, variant_gpu]:
    kernel_gen = lp.generate_loop_schedules(variant(knl))
    # ...
```

Capturing Variants

```
knl = ...

def variant_cpu(knl):
    knl = lp.split_dimension(knl, "i", 16*4096, outer_tag="g.0", slabs=(0, 1))
    knl = lp.split_dimension(knl, "i_inner", 16,
            inner_tag="unr")
    return knl

def variant_gpu(knl):
    knl = lp.split_dimension(knl, "i", 4*256, outer_tag="g.0", slabs=(0, 1))
    knl = lp.split_dimension(knl, "i_inner", block_size,
            outer_tag="unr", inner_tag="l.0")
    return knl

for variant in [variant_cpu, varian
    kernel_gen = lp.generate_loop_s
    # ...
```

Easy to *non-redundantly* capture multiple variants of the same kernel.

Ordering

- Completely *un*ordered by default
- Program only well-formed
  if domain traversal order does not matter
- Depdencies
  can dictate execution order
  within largest set of shared loops

Loo.py vs reality

- Two modes of operation:
  - Standalone
  - In-process
- Flat data structure:
  - Easy to manipulate
  - Kernel fusion
- Register-your-own:
  - Functions
  - Symbols
  - Reductions
- Literal code ‘escape hatch’
- Predicated execution
- Tree-of-domains for data-dependent control flow

Bonus Features

Free extras:

- A-priori bounds checking
- Generate a sequential version of the code
- Automatic Benchmarking
- Free tuning advice
  - Local memory layout
  - Suboptimal use of hw parallelism
  - Based on knowledge about target hardware
- Automatic Testing
  - . . . against sequential version
  - . . . which is easier to verify

DEMO TIME