Did that Code Transformation Break your Kernel? Formalizing Dependency Semantics in Loopy

James Stevens

November 14, 2019

Acknowledgements

- Andreas Klöckner
- Matt Wala
- Kaushik Kulkarni

Motivation for Program Abstraction and Code Generation

Parallel (GPU) code dev:	Hand-written	Compiler directives	Goal
Development time	↑	↓	↓
Code modification time	↑	↓	↓
P(optimal performance)	↑	↓	↑
P(coding errors)	↑	↓	↓
Modify source at runtime?	**No**	**No**	**Yes**

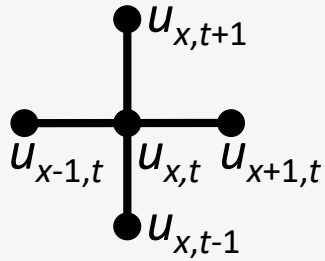
Loopy Example

- Programming system for array computations that targets GPUs, CPUs
- Separates mathematical intent from computational minutiae
- Realizes programs as Python objects initiated with (symbolic) mathematical intent, then transformed for performance

Example Loopy program: solve wave equation

$$u_{tt} = c \cdot u_{xx}$$

```
knl = lp.make_kernel(
  "[nx,nt] -> {[x, t]: 1<=x<nx-1 and 0<=t<nt}",
  """
  u[x, t+2] = (
   2*u[x, t+1] +
   dt**2/dx**2 * (u[x+1, t+1] - 2*u[x, t+1] + u[x-1, t+1])
   - u[x, t]) {id=0}
  """)
```



[Klöckner et al.(2016)Klöckner, Wilcox, and Warburton], [Klöckner(2015)], [Klöckner(2014)]

Loopy Example

Loopy-generated code:

```
__kernel void [...] wave_equation(float const dt, float const dx, int
    const nt, int const nx, __global float *__restrict__ u)
{
  for (int x = 1; x <= -2 + nx; ++x)
    for (int t = 0; t <= -1 + nt; ++t)
      u[(2 + nt) * x + 2 + t] =
        2.0f * u[(2 + nt) * x + 1 + t] +
        ((dt * dt) / (dx * dx)) * (u[(2 + nt) * (1 + x) + 1 + t] + -1.0
            f * 2.0f * u[(2 + nt) * x + 1 + t] + u[(2 + nt) * (-1 + x)
            + 1 + t]) +
        -1.0f * u[(2 + nt) * x + t];
}
```

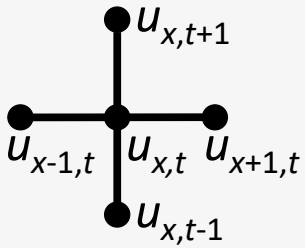
Loopy program transformations:

- Loop splitting, unrolling, vectorization, parallelization, prefetching, ...

Motivation for Formal, Checkable Dependency Semantics

LOOPY-generated wave equation code:

```
__kernel void [...] wave_equation([...])
{
  for (int x = 1; x <= -2 + nx; ++x)
    for (int t = 0; t <= -1 + nt; ++t)
      u[(2 + nt) * x + 2 + t] = [...];
}
```



Are dependencies satisfied?

Motivation for Formal, Checkable Dependency Semantics

To correct loop nesting order:

```
knl = lp.make_kernel([...])
knl = lp.prioritize_loops(knl, ("t", "x"))
```

```
__kernel void [...] wave_equation([...])
{
  for (int t = 0; t <= -1 + nt; ++t)
    for (int x = 1; x <= -2 + nx; ++x)
      u[(2 + nt) * x + 2 + t] = [...];
}
```

Are dependencies satisfied now?

Motivation for Program Abstraction and Code Generation

- Preview of transformed wave equation kernel used in Results section
  - Are dependencies satisfied?

```
__kernel void [...] wave_equation([...])
{
  for (int tt = 0; tt <= (2 + -1 * nt + 16 * ((-2 + nt) / 16) == 0 && 16 + -1 * nx >= 0 ? (-2 + nt) / 16
        : (14 + nt) / 16); ++tt)
    for (int tparity = (5 + nt + -16 * tt >= 0 && -3 + nt + nx + -16 * tt >= 0 ? 0 : ((-7 + -1 * nt + 16
          * tt >= 0 && 12 + nt + -16 * tt >= 0) || (2 + -1 * nt + -1 * nx + 16 * tt >= 0 && 6 + nt + -16
          * tt >= 0) ? 1 : 2 * tt + -1 * ((6 + nt) / 8))); tparity <= (tt == 0 ? 0 : 1); ++tparity)
      for (int tx = (-6 + -1 * nt + -8 * tparity + 16 * tt == 0 && -9 + nx + -8 * tparity >= 0 ? 0 : -1 *
            tparity); tx <= (-1 * nt + -8 * tparity + 16 * tt >= 0 && 6 + nt + 8 * tparity + -16 * tt >=
            0 && 18 + -1 * nt + 16 * ((-3 + nt) / 16) >= 0 && 13 + nt + nx + 16 * tparity + -32 * tt +
           16 * ((-3 + nt) / 16) >= 0 && -3 + nt + -16 * ((-3 + nt) / 16) >= 0 ? -1 + -1 * tt + (13 + nt
            + nx) / 16 : -1 + -1 * tparity + (14 + nx + 8 * tparity) / 16); ++tx)
        for (int itt = (tt == 0 && tparity == 0 ? 0 : (-1 + -1 * tparity + 2 * tt >= 0 && tparity + 2 *
              tx >= 0 && -9 + nx + -8 * tparity + -16 * tx >= 0 ? -7 : (1 + tx == 0 && -1 + tparity == 0
              && -1 + tt >= 0 && 13 + nt + -16 * tt >= 0 ? -6 : 2 + -1 * nx + 8 * tparity + 16 * tx)));
              itt <= ((1 + tx == 0 && -1 + tparity == 0 && -1 + -1 * nx >= 0 && -1 + tt >= 0 && 1 + nt +
              -16 * tt >= 0) || (1 + tx == 0 && -1 + tparity == 0 && -1 + nx >= 0 && -1 + tt >= 0 && 1 +
              nt + -16 * tt >= 0) ? 6 : (-1 * tparity + 2 * tt >= 0 && -8 + nt + 8 * tparity + -16 * tt
              >= 0 && tparity + 2 * tx >= 0 && -9 + nx + -8 * tparity + -16 * tx >= 0 ? 7 : ((8 + -1 * nx
               + 8 * tparity + 16 * tx >= 0 && -2 + -1 * nt + nx + -16 * tparity + 16 * tt + -16 * tx >=
              0) || (7 + -1 * nt + -8 * tparity + 16 * tt >= 0 && tparity + 2 * tx >= 0 && -9 + nx + -8 *
               tparity + -16 * tx >= 0) || (1 + tx == 0 && -1 + tparity == 0 && -2 + -1 * nt + 16 * tt >=
               0 && nt + -1 * nx + -16 * tt >= 0 && 13 + nt + -16 * tt >= 0) || (1 + tx == 0 && -1 +
              tparity == 0 && -1 + tt >= 0 && -2 + -1 * nt + 16 * tt >= 0 && -2 + -1 * nt + nx + 16 * tt
              >= 0 && 13 + nt + -16 * tt >= 0) || (1 + tx == 0 && -1 + tparity == 0 && -1 * tt >= 0 && -2
```

Motivation for Formal, Checkable Dependency Semantics

- Complicated kernels may undergo numerous transformations
- To ensure generated code does not violate dependencies, we need:
  - Formal dependency semantics
  - Dependencies that survive program transformations
  - Mechanism to determine whether dependencies have been violated

Formalize Program Schedule and Dependency Constraints

- **Program schedule**

  $P : \{(\ (n_s, i_0, ..., i_k) \rightarrow (l_0, ..., l_m)\ )\}$

  n_s: instruction number

  $i_0, ..., i_k$: values for all sequential loop indices ('inames')

  $(l_0, ..., l_m)$: point in a lexicographic ordering

  Relation P maps set of statement instances S to set of points in lexicographic time T_L

  $P : S \rightarrow T_L$

- **Statement instance:** Single integer point in polyhedral domain of loop index values for a LOOPY statement

Formalize Program Schedule and Dependency Constraints

Example program schedule

```
k = lp.make_kernel(
     "{[j]: 0<=j<3}",
     ["a[j] = b[j] {id=0}",
      "a[j] = a[j] + 1 {id=1}"]
     )
```

```
for (int j = 0; j <= 2; ++j)
{
  a[j] = b[j];
  a[j] = a[j] + 1;
}
```

(n_s, j)	(l_0, l_1)
$(0,0)$	$\rightarrow (0,0)$
$(1,0)$	$\rightarrow (0,1)$
$(0,1)$	$\rightarrow (1,0)$
$(1,1)$	$\rightarrow (1,1)$
$(0,2)$	$\rightarrow (2,0)$
$(1,2)$	$\rightarrow (2,1)$

$$P : \{(n_s, j) \rightarrow (j, n_s) \ : \ 0 \leq n_s < 2 \text{ and } 0 \leq j < 3\}$$

Imperfectly-nested loops and instructions with differing iname sets add complexity

Formalize Program Schedule and Dependency Constraints

$$L : \{ (l_0, l_1) \to (l'_0, l'_1) \ : \ l_0 < l'_0 \text{ or } (l_0 = l'_0 \text{ and } l_1 < l'_1)\}$$

- **Lexicographic order**

  $L : \{t_L \to t'_L \ : \ t_L \prec t'_L\}$

  $\prec$: "lexicographically before"

$$\begin{array}{ll}
(0,0) \to (0,1) & (0,1) \to (2,0) \\
(0,0) \to (1,0) & (0,1) \to (2,1) \\
(0,0) \to (1,1) & (1,0) \to (1,1) \\
(0,0) \to (2,0) & (1,0) \to (2,0) \\
(0,0) \to (2,1) & (1,0) \to (2,1) \\
(0,1) \to (1,0) & (1,1) \to (2,0) \\
(0,1) \to (1,1) & (1,1) \to (2,1) \\
 & (2,0) \to (2,1)
\end{array}$$

Formalize Program Schedule and Dependency Constraints

- **(Dependency) Constraints**

  $C : \{s \rightarrow s' \ : \ s \text{ must happen before } s'\}$

  where $s, s' \in S$

  Relation C mapping statement instances to statement instances that must occur later

```
# make kernel:
k = lp.make_kernel(
  "{[j]: 0<=j<3}",
  ["a[j] = b[j] {id=0}",
   "a[j] = a[j] + 1 {id=1}"
  ])
```

Example: $C : \{(0, j) \rightarrow (1, j') \ : \ j = j'\}$

Formalize Program Schedule and Dependency Constraints

How can we use these three relations to evaluate program validity?

- **Program schedule:** $P : \{s \rightarrow t_L\} \qquad s \in S, t_L \in T_L$
- **Lexicographic order:** $L : \{t_L \rightarrow t'_L \ : \ t_L \prec t'_L\}$
- **Constraints:** $C : \{s \rightarrow s' \ : \ s \text{ must happen before } s'\}$

Formalize Program Schedule and Dependency Constraints

First, use P and L to create **statement instance order** mapping each statement instance to every statement instance that occurs after it

- **Program schedule:** $P : \{s \to t_L\} \qquad s \in S, t_L \in T_L$
- **Lexicographic order:** $L : \{t_L \to t'_L : \ t_L \prec t'_L\}$
- **Constraints:** $C : \{s \to s' : \ s \text{ must happen before } s'\}$
- **Statement instance order:** $O = P \circ L \circ P^{-1}$

  $O : \{s \to s' : \ s \text{ happens before } s'\}$

Formalize Program Schedule and Dependency Constraints

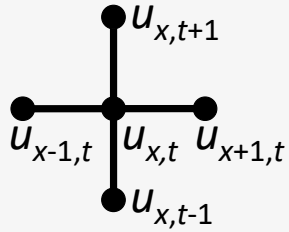
Then use C and O to determine if constraints have been met

- **Constraints:** $C : \{s \to s' : \ s \text{ must happen before } s'\}$
- **Statement instance order:** $O : \{s \to s' : \ s \text{ happens before } s'\}$
- **Constraints satisfied?** $C \overset{?}{\subseteq} O$

Wave equation dependency checking

$$C : \{$$
$$(0, x, t) \to (0, x', t') :$$
$$t' = 1 + t \quad \wedge \quad -1 + x' \leq x \leq 1 + x' \quad \wedge \quad D_{x,x',t,t'},$$
$$(0, x, t) \to (0, x', t') :$$
$$t' = 2 + t \quad \wedge \quad x' = x \quad \wedge \quad D_{x,x',t,t'}$$
$$\}$$

$$u_{tt} = c \cdot u_{xx}$$



note: primes are not derivatives

Wave equation dependency checking

```
__kernel [...]
{
 for (int x = 1; x <= ...
  for (int t = 0; t <= ...
   u[...] = ...
}
```

$$P : \{(0,t,x) \to (0,x,0,t,0) : \ D_{x,t} \ \}$$

$$O : \{$$

$$(0,t,x) \to (0,t',x') : x' > x \wedge D_{x,t,x',t'}$$

$$(0,t,x) \to (0,t',x') : x' = x \wedge t' > t \wedge D_{x,t,x',t'}$$

$$\}$$

```
__kernel [...]
{
 for (int t = 0; t <= ...
  for (int x = 1; x <= ...
   u[...] = ...
}
```

$$P^* : \{(0,t,x) \to (0,\mathbf{t},0,\mathbf{x},0) : \ D_{x,t} \ \}$$

$$O^* : \{$$

$$(0,t,x) \to (0,t',x') : \mathbf{t}' > \mathbf{t} \wedge D_{x,t,x',t'}$$

$$(0,t,x) \to (0,t',x') : \mathbf{t}' = \mathbf{t} \wedge \mathbf{x}' > \mathbf{x} \wedge D_{x,t,x',t'}$$

$$\}$$

Wave equation dependency checking

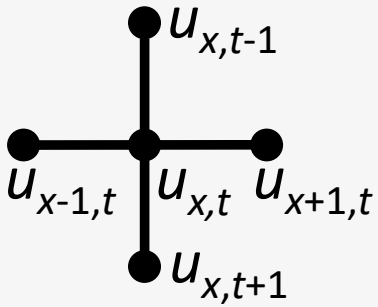
$$C: \{ (0,x,t) \to (0,x',t') : \quad t' = 1 + t \quad \wedge \quad -1 + x' \leq x \leq 1 + x' \quad \wedge \quad D_{x,x',t,t'},$$
$$(0,x,t) \to (0,x',t') : \quad t' = 2 + t \quad \wedge \quad x' = x \quad \wedge \quad D_{x,x',t,t'} \}$$

$$O: \{ (0,t,x) \to (0,t',x') : \quad x' > x \quad \wedge \quad D_{x,t,x',t'},$$
$$(0,t,x) \to (0,t',x') : \quad x' = x \quad \wedge \quad t' > t \quad \wedge \quad D_{x,t,x',t'} \}$$

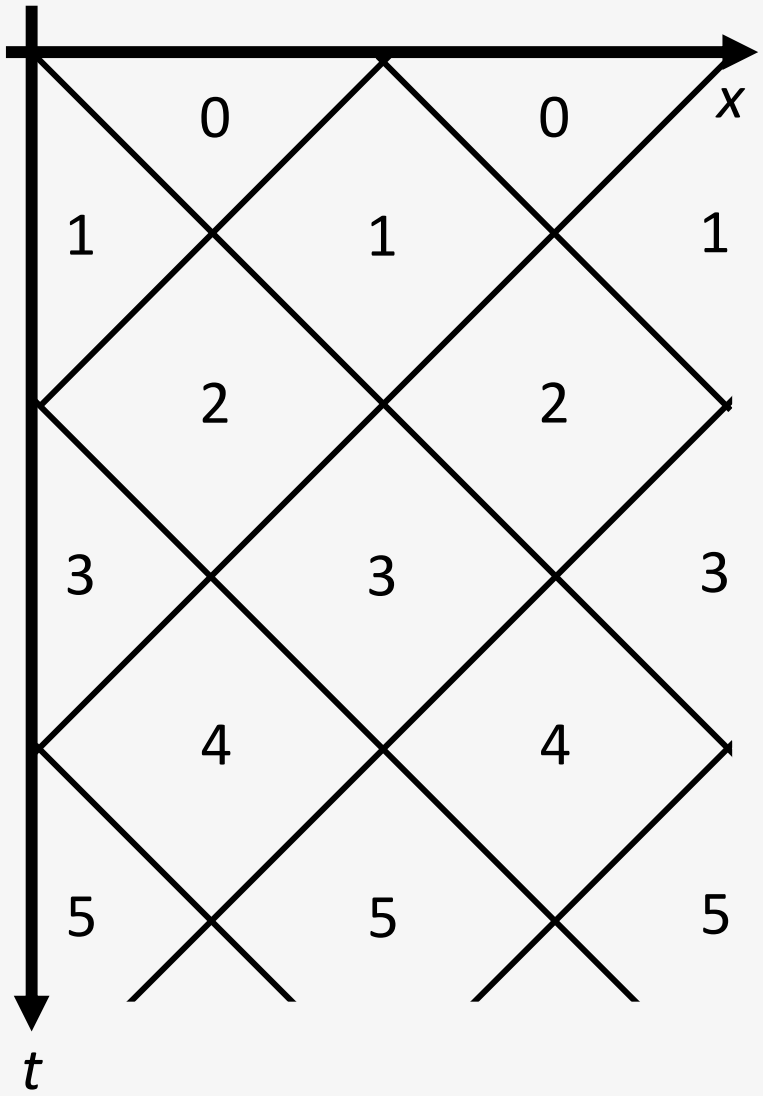
$$O^*: \{ (0,t,x) \to (0,t',x') : \quad \mathbf{t}' > \mathbf{t} \quad \wedge \quad D_{x,t,x',t'},$$
$$(0,t,x) \to (0,t',x') : \quad \mathbf{t}' = \mathbf{t} \quad \wedge \quad \mathbf{x}' > \mathbf{x} \quad \wedge \quad D_{x,t,x',t'} \}$$

$\mathbf{C} \not\subseteq \mathbf{O}$ $\qquad$ $\mathbf{C} \subseteq \mathbf{O}^*$

Wave equation dependency checking



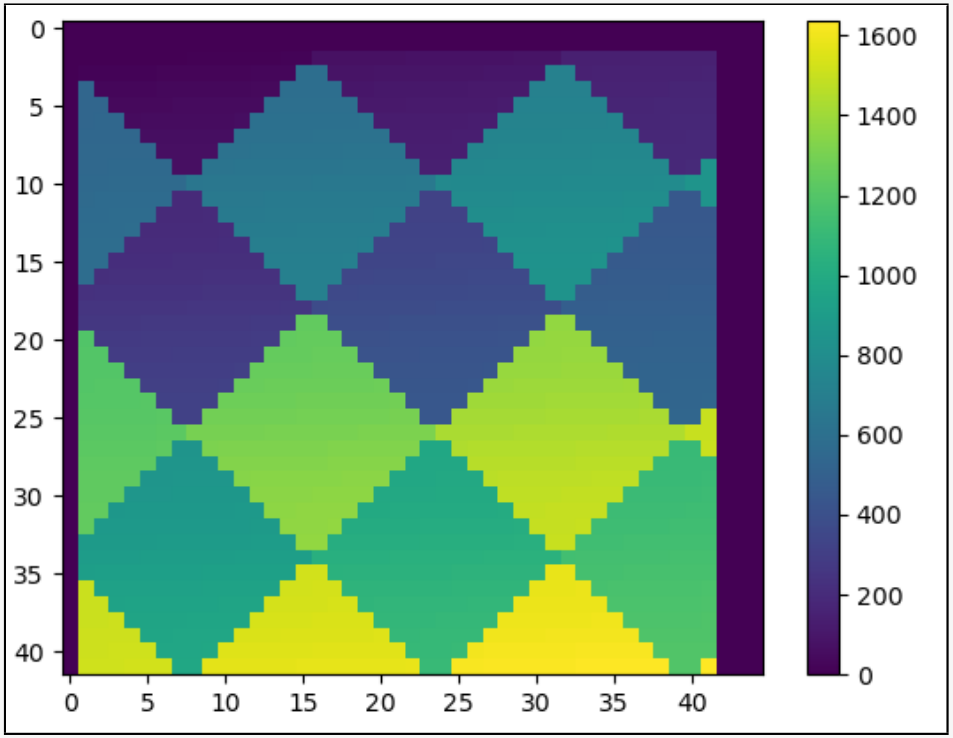
Can we create opportunities for concurrency while satisfying dependencies?



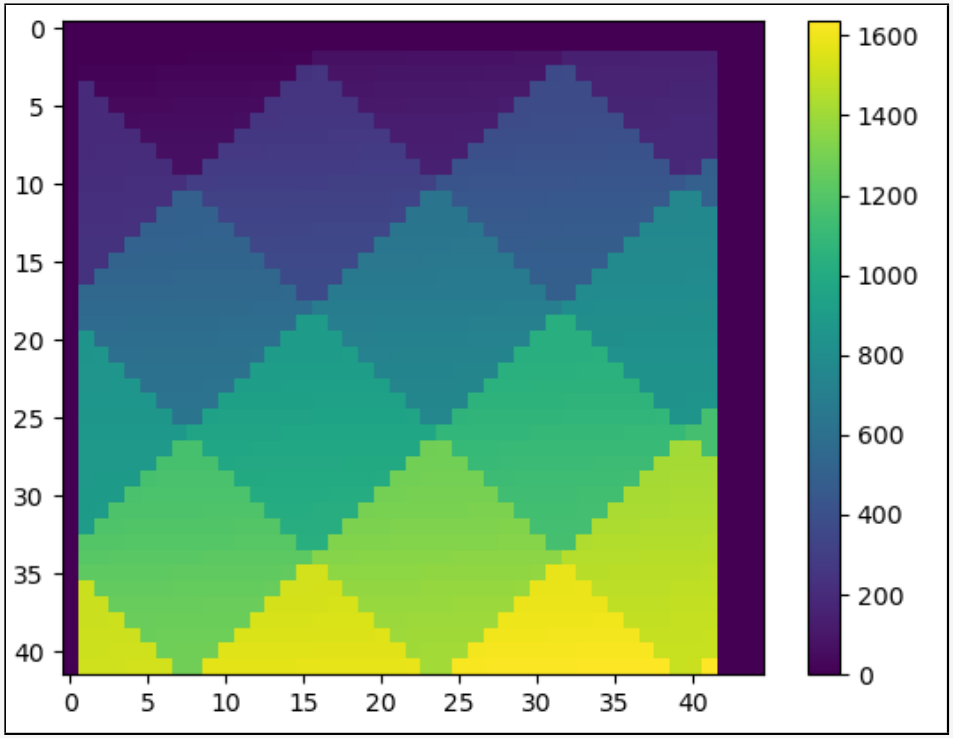
Transformed wave equation kernel dependency checking

```
wave_knl = lp.make_kernel(
  "[nx,nt] -> {[x, t]: 1<=x<nx-1 and 0<=t<nt}",
  """
  u[x, t+2] = (
      dt**2/dx**2 * (u[x+1, t+1] - 2*u[x, t+1] + u[x-1, t+1])
      + 2*u[x, t+1] - u[x, t]) {id=0}
  """)

m = isl.BasicMap(
  "[nx,nt] -> {[x, t] -> [tx, tt, tparity, itt, itx]: "
  "16*(tx - tt) + itx - itt = x - t and "
  "16*(tx + tt + tparity) + itt + itx = x + t and "
  "0<=tparity<2 and 0 <= itx - itt < 16 and 0 <= itt+itx < 16}")

wave_knl = lp.map_domain(wave_knl, m)
wave_knl = lp.prioritize_loops(wave_knl, "tt,tparity,tx,itt,itx")

constraint_map = constraint_map.apply_range(m).apply_domain(m)
```

Domain execution order visualization



$\mathbf{C} \not\subseteq \mathbf{O}$



$\mathbf{C} \subseteq \mathbf{O}$

Conclusions

- Accomplished so far:
  - Defined formal semantics for dependencies and statement instance order that can be used to determine whether dependencies are violated
  - Partially-automated mechanism to determine whether dependencies have been violated
- Still to do:
  - Convenient user input specifying arbitrary dependencies
  - Dependencies survive all program transformations
  - Fully-automated mechanism to determine whether dependencies have been violated

Bibliography I


Andreas Klöckner, Lucas C. Wilcox, and T. Warburton.
Array program transformation with loo.py by example: High-order finite elements. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pages 9–16, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4384-8. DOI: 10.1145/2935323.2935325.


Andreas Klöckner.
Loo.Py: Transformation-based Code Generation for GPUs and CPUs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 82:82–82:87, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2937-8. DOI: 10.1145/2627373.2627387.

Bibliography II

Andreas Klöckner.
Loo.Py: From Fortran to Performance via Transformation and Substitution Rules. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2015, pages 1–6, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3584-3. DOI: 10.1145/2774959.2774969.