

For Committee: Slides containing key recent work are starred ★

Summary of New Accomplishments Since Feb. Meeting

- Extension of statement ordering semantics to precisely verify schedules w/GPU parallelism in the presence of local and global barriers
 - Based on novel 'lexicographic barrier segment count' procedure
 - Enables reasoning about dependency violations across work-items
- Handling of dependencies during set of complex transformations
 - Enables pre-transform dependency expression w/post-transform check
- Applying dependency checking to *transformed* wave example
- Demonstrate ingestion of a complex, application-level benchmark (NPB LU), and applicability of semantics therein
- Program transformation UI: Expand scope of transformations, esp. via direct code manipulation; improve scalability of action counts with respect to search space size
- Demonstrate UI comprehensiveness, scalability in complex application

(non-committee members may ignore)

Program Transformation and Code Generation for Developing, Modeling, and Optimizing GPU Programs

Dissertation Defense

James Stevens
PhD Candidate

July 2, 2021

Acknowledgements

Advisor:

- Andreas Klöckner
LOOPY, LOOPY-UI, PERFLEX, UIPiCK

Thesis Committee:

- Andreas Klöckner
- William Gropp
- Edgar Solomonik
- John Owens

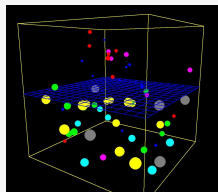
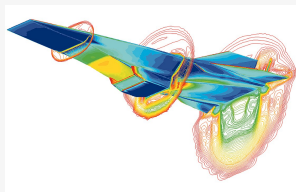
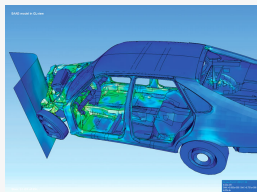
LOOPY:

- Matt Wala
- Kaushik Kulkarni

LOOPY-UI:

- Summer Xia
- Eunsun Lee
- Juefei Chen
- Feng Hou
- Bogdan Enache

Challenge



High-performance computation crucial for numerous scientific and engineering problems

Challenge

Performance requires tailoring algorithm to architecture



Can we obtain benefits of *both* fully manual and fully automated approaches to program transformation and optimization using *partially automated, human-guided* strategies?

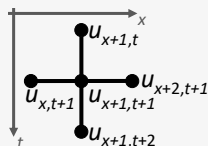
Contributions: Mechanisms to Address Challenge

- 1 Programming system providing transformation-based code generation for GPUs and CPUs
- 2 Mechanism for balancing accuracy and scope in cross-machine black-box GPU performance modeling
- 3 Visual user interface for code transformation, analysis, and optimization

Formal Dependency Verification and Loop Nesting Semantics in LOOPY

Motivating LOOPY Example

- Programming system for array computations providing GPU/CPU code generation
- Separates mathematical intent from efficiency decisions



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

Example LOOPY¹ program: solve wave equation

```
kn1 = lp.make_kernel(
    "[nx,nt] -> {[x, t]: 0<=x<nx and 0<=t<nt}", # Domain
    "u[t+2,x+1] = 2*u[t+1,x+1] + dt**2/dx**2" # Statement
    " * (u[t+1,x+2] - 2*u[t+1,x+1] + u[t+1,x]) - u[t,x+1] {id=stmt}")
```

¹ [Klo14], [Klo15], [KWW16]

Motivating LOOPY Example

Transformations:

```

knl = lp.add_dtypes(knl, {"u,dx,dt": np.float32})
knl = lp.split_iname(knl, "x", 14)
knl = lp.assume(knl, "nx % 14 = 0 and nt >= 1 and nx >= 1")
knl = lp.tag_inames(knl, "x_outer:g.0, x_inner:l.0")

```

LOOPY-generated code:

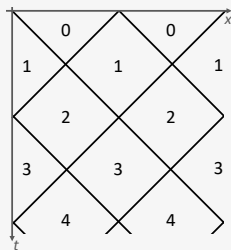
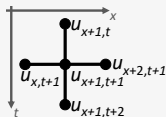
```

... loopy_kernel(float const dt, ..., __global float *__restrict__ u)
{
  for (int t = 0; t <= -1 + nt; ++t)
    u[(nx + 2) * (t + 2) + 1 + 14 * gid(0) + lid(0)] =
      2.0f * u[(nx + 2) * (t + 1) + 1 + 14 * gid(0) + lid(0)] + ...;
}

```

Motivating LOOPY Example

- Are dependencies satisfied?
- Apply diamond tiling² to expose further parallelism?
 - Correctness requires precise ordering of statement instances
- Prior LOOPY:
 - Dependency not capturable, no correctness verification
 - No guarantee of desired loop nest structure



² [BPB12], [BOH⁺15], [BBP17]

Motivating LOOPY Example

- Manually determine whether dependencies violated?
 - OPENCL generated after diamond-tiling transformation:

```

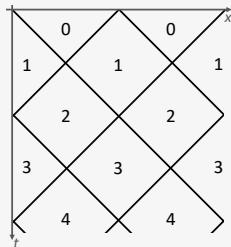
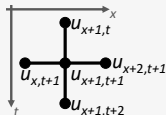
__kernel void [...] wave_stencil([...])
{
  for (int tt = 0; tt <= (14 + nt) / 32; ++tt)
    for (int tparity = 0; tparity <= (1 + -1 * nt + 32 * tt) >= 0 || (17 + -1 * nx + -1 * nt + 32 * tt) >= 0 && -2 + nt + -32 * tt >= 0) ? 0 : 1); ++tparity)
      for (int tx = 0; tx <= (-1 * nt + 16 * tparity + 32 * tt) >= 0 && 14 + nt + -16 * tparity + -32 * tt >= 0 && 14 + nx + nt + -32 * tparity + -32 * tt >= 0 ? -1 * tparity + -1 * tt + (14 + nx + nt) / 32 : -1 * tparity + (15 + nx + 16 * tparity) / 32); ++tx)
        for (int itt = (tt == 0 && tparity == 0 ? 0 : (-1 + tparity + 2 * tt) >= 0 && nx + -16 * tparity + -32 * tx >= 0 ? -15 : -15 + -1 * nx + 16 * tparity + 32 * tx)); itt <= (-16 + nt + -16 * tparity + -32 * tt) >= 0 && nx + -16 * tparity + -32 * tx >= 0 ? 15 : (15 + -1 * nt + 16 * tparity + 32 * tt) >= 0 && 16 + nx + -1 * nt + 32 * tt + -32 * tx >= 0 && 14 + nx + nt + -32 * tparity + -32 * tt + -32 * tx >= 0 ? -1 + nt + -16 * tparity + -32 * tt : 15 + nx + -16 * tparity + -32 * tx)); ++itt)
          for (int itx = (tx == 0 && tparity == 0 && itt + 32 * tt >= 0 ? 16 : ((-1 + -1 * itt) >= 0 && itt + 16 * tparity + 32 * tt >= 0 && -16 + itt + 16 * tparity + 32 * tx >= 0) || (tx == 0 && -1 + tparity == 0 && -1 + -1 * itt >= 0) ? -1 * itt : itt)); itx <= (itt >= 0 && -16 + nx + itt + -16 * tparity + -32 * tx >= 0 ? 31 + -1 * itt : (-1 + -1 * itt) >= 0 && itt + 16 * tparity + 32 * tt >= 0 && -16 + nx + -1 * itt + -16 * tparity + -32 * tx >= 0 ? 31 + itt : 15 + nx + -16 * tparity + -32 * tx)); ++itx)
            u[(2 + nx) * (2 + 32 * tt + 16 * tparity + itt) + -15 + 16 * tparity + 32 * tx + itx] = 2.0f * u[(2 + nx) * (1 + 32 * tt + 16 * tparity + itt) + -15 + 16 * tparity + 32 * tx + itx] + ((dt * dt) / (dx * dx)) * (u[(2 + nx) * (1 + 32 * tt + 16 * tparity + itt) + -14 + 16 * tparity + 32 * tx + itx] + -1.0f * 2.0f * u[(2 + nx) * (1 + 32 * tt + 16 * tparity + itt) + -15 + 16 * tparity + 32 * tx + itx] + u[(2 + nx) * (1 + 32 * tt + 16 * tparity + itt) + -16 + 16 * tparity + 32 * tx + itx]) + -1.0f * u[(2 + nx) * (32 * tt + 16 * tparity + itt) + -15 + 16 * tparity + 32 * tx + itx];

```

Motivating LOOPY Example

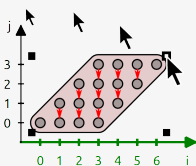
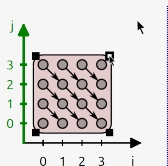
Need dependencies that:

- Are well-defined
- Operate at statement-instance level
- Survive transformations
- Can be automatically checked for violations
- Are user-accessible



Related Work

```
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    z[i+j] += x[i] * y[j];
```



```
for (i = 0; i < 2*N-1; ++i)
  for (j = max(0, i-N+1);
       j < min(i+1, N); ++j)
    z[i] += x[i-j] * y[j];
```

3

Polyhedral Model of Computation

- Statements executed over points in polyhedral iteration domain
 - Bounded by affine inequalities
- Enable mathematical reasoning about execution order, dependencies, correctness, transformations
- Facilitates automated transformation application

³ [ZHB18]

Related Work (Selected)

Largely automated

- PLUTO⁴, PoCC⁵, POLLY⁶, PPCG⁷, RSTREAM⁸, PIPS⁹
- Minimize affine objective func. while obeying deps
 - (-) Expensive
- (-) Less user exposure to heuristics, transform decisions

More user-guided

- ALPHA/ALPHAZ¹⁰: expression-based
 - Space, time arise via transformations
 - (-) Very abstract; hard to write, refactor
- CHILL¹¹: Composition of affine transforms
 - Automatically generated data deps
 - (-) *Only* affine transforms; C-based
 - (-) Scripting language is stateful w/ inflexible addressing
- DACE¹²: Data-flow graph IR, transforms
 - (-) More restrictive, graph-based prog. IR

Details in Appendix 6 of presentation and Section 3.1.1 of [dissertation]

⁴ [BHR08] ⁵ [PBB⁺09] ⁶ [GGL12] ⁷ [VJC⁺13] ⁸ [MVW⁺11], [MLV⁺09] ⁹ [IJT91]
¹⁰ [Mau89], [YGK⁺13], [YBG⁺12] ¹¹ [CCH08], [ZVBH16] ¹² [BNdFLZ⁺19]

Formal, Verifiable Dependency Semantics

statement: assignment to scalar or scalar array elements, executed over integer points in polyhedral domain

statement instance: instance of statement for one integer point

$$\underbrace{u[t+2, x+1] = 2 * u[t+1, x+1] + \dots}_{\text{statement instance}} \quad \text{at } x = 5, t = 6$$

statement

- Define dependencies as ‘happens-before’ relationship between statement instances
 - IR: maps involving statement instances, lexicographical orderings; similar to lit¹³

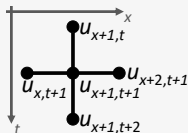
¹³ [GGL12], [VJC⁺13], [YGK⁺13], [YBG⁺12]

Formal, Verifiable Dependency Semantics

Dependency: $D_{m,n} \subseteq S_m \times S_n$ $s_m = (m, i_0, \dots, i_k) \in S_m$

S_m, S_n : Sets of all statement instances for statements m, n

s_m : Statement instance (unique statement ID and set of integer values for sequential loop indices)



Dependency for wave stencil:

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

↖ In polyhedral representation

Formal Statement Ordering Semantics

To **check dependency** $D_{m,n}$, construct pairwise **statement instance ordering** (SIO):

$$O^{P_{m,n}} \subseteq S_m \times S_n,$$

mapping each instance of S_m to all instances of S_n executing later in linearized program

Dependency satisfaction check:

$$D_{m,n} \stackrel{?}{\subseteq} O^{P_{m,n}}$$

Formal Statement Ordering Semantics



To construct SIO, first construct three pairwise **program schedules**

- *Intra-thread*, *intra-group*, *global*
- Map statement instances to points in lexicographical space

Intra-thread schedule: execution within *same work-item* (thread)

$$P_{m,n}^{\text{thread}} \subseteq \{S_m \cup S_n\} \times L^{\text{thread}} \quad I^* = (l_0, \dots, l_{d_{\text{thread}}-1}) \in L^{\text{thread}}$$

Intra-thread SIO:

$$O_{m,n}^{P^{\text{thread}}} = P_m^{\text{thread}} \circ O^{L^{\text{thread}}} \circ (P_n^{\text{thread}})^{-1} \subseteq S_m \times S_n$$

$O^{L^{\text{thread}}}$: maps each point in lex. ordering to every *later* point

Formal Statement Ordering Semantics



Programmatic construction of *intra-group* schedule, SIO

```

s0          (0, 0, 0)
s1          (0, 0, 0)
< barrier >
s2          (1, 0, 0)
s3          (1, 0, 0)
for i = 0 : n
  s4          (2, i, 0)
  s5          (2, i, 0)
  < barrier >
  s6          (2, i, 1)
  s7          (2, i, 1)
end
s8          (3, 0, 0)
s9          (3, 0, 0)
< barrier >
s10         (4, 0, 0)
s11         (4, 0, 0)
for j
  s12         (4, 0, 0)
  s13         (4, 0, 0)
end
...

```

- Statement instances executed within *different* work-items in *same* work-group
 - Ordering mediated by local barriers
 - **Map statement instances to barrier-delimited lex.-numbered program section**
- Avoid unwanted before-after SIO pairs at top, bottom of loops
 - Combine $P_{m,n}^{\text{group}}$ with specialized lex. order mapping O^L_{group}

Formal Statement Ordering Semantics



Start with standard lex. ordering O_{full}^{Lgroup}

Subtract subset of before-after pairs R^{group}
to get O^{Lgroup}

$$R^{group} = \left(\bigcup_i (R_{i-enter}^{group} \cup R_{i-exit}^{group} \cup R_{i-step}^{group}) \right)^+$$

$$O^{Lgroup} = O_{full}^{Lgroup} \setminus R^{group}$$

```

...
< barrier >
s2
s3
for i = 0 : n
  s4
  s5
  < barrier >
  s6
  s7
end
s8
s9
< barrier >
...

```

$R_{i-enter}^{group}$ (1, 0, 0)
 (1, 0, 0)
 R_{i-step}^{group} (2, i, 0)
 (2, i, 0)
 R_{i-exit}^{group} (2, i, 1)
 (2, i, 1)
 (3, 0, 0)
 (3, 0, 0)

Intra-group SIO:

$$O_{m,n}^{Pgroup} = P_m^{group} \circ O^{Lgroup} \circ (P_n^{group})^{-1} \subseteq S_m \times S_n$$

Transitive closure not guaranteed to be quasi-affine; when necessary, use over-approximation¹⁴

¹⁴ [VCB11]

Formal Statement Ordering Semantics



Complete pairwise **SIO**:

$$\begin{aligned}
 O^{P_{m,n}} &= \left(O^{P_{m,n}^{\text{thread}}} \cap T_{m,n}^{\text{GID}} \cap T_{m,n}^{\text{LID}} \right) \\
 &\cup \left(O^{P_{m,n}^{\text{group}}} \cap T_{m,n}^{\text{GID}} \right) \\
 &\cup \left(O^{P_{m,n}^{\text{global}}} \right) \\
 &\subseteq S_m \times S_n,
 \end{aligned}$$

Dependency check:

$$D_{m,n} \stackrel{?}{\subseteq} O^{P_{m,n}}$$

$T_{m,n}^{\text{GID}}$ maps all instances of m to all instances of n assigned to same **work-group**

$T_{m,n}^{\text{LID}}$ maps all instances of m to all instances of n assigned to same **work-item**

Dependencies and Program Linearization

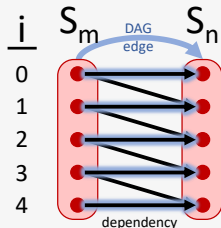
Linearizer: Convert unordered statements into fully ordered program

Traditional polyhedral scheduling

- Obey all deps and minimize objective function
- Expensive; opaque to user

Instead

- Avoid enumerating all possible orderings
- Guide linearization w/ coarse-grained representation of deps: statement DAG¹⁵
 - Full deps form graph on statement *instances*
- Report dependency violation



¹⁵ DAG formation details in Section 3.2.2 of [dissertation]

Formal, Enforceable Loop Nest Structure Semantics

- Within bounds of correctness, program order is performance concern
- Key ordering concern: nesting structure of loops
 - Affects performance: cache, TLB hit rates
 - Loop structure may be prerequisite for transformation, e.g., vectorize

```
for i
  for j
    for k
      ...
    for g
      ...
  for h
    for r
      ...
```

Formal, Enforceable Loop Nest Structure Semantics

- Nesting constraint properties: well-defined, 'survive' transformations, enforceable, concisely express 'innermost'
- **Must-nest** set of loop pairs C_m and **must-not-nest** set C_n satisfied by program w/nesting pairs N if:

$$C_m \subseteq N \quad \wedge \quad C_n \cap N = \emptyset$$

- "Innermost": $C_n = (k, \neg k)$

```

for i
  for j
    for k
      ...
    for g
      ...
    for h
      for r
        ...
  
```

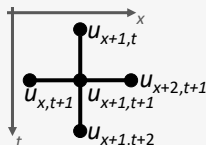
Details in Appendix 3 of presentation and Section 3.2.3 of [dissertation]

Experimental Results: FD Wave Equation Solution



Finite Differences Solution to the Wave Equation with Diamond Tiling

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



```
# Make kernel
```

```
knl = lp.make_kernel(...)
```

```
# Specify dependency
```

```
dep = make_dep_map(
```

```
    "{ [t', x'] -> [t, x] : "
```

```
    " (t = t' + 2 and x = x') or "
```

```
    " (t = 1 + t' and x' - 1 <= x <= x' + 1) }",
```

```
    self_dep=True, # Statement depends on itself
```

```
    knl_with_domains=knl) # Provide kernel w/ relevant loop domains
```

```
knl = lp.add_dependency_v2(knl, "stmt", "stmt", dep) # stmt <- stmt
```

Experimental Results: FD Wave Equation Solution

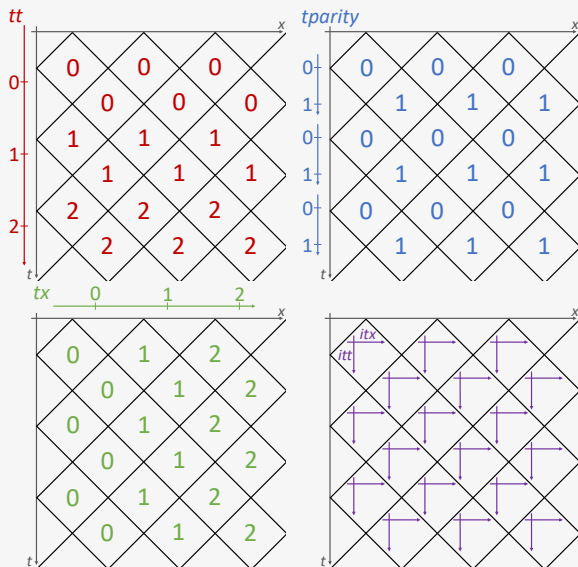


Diamond tiling transformation w/map from current indices to new indices:

```
transform_map = isl.BasicMap(
  f"""
  {[t, x] -> [tt, tparity, tx, itt, itx]:
  {tile_sz}*(tx - tt) - {tile_sz//2} + itx - itt = x - t and
  {tile_sz}*(tx + tt + tparity) - {tile_sz//2} + itt + itx = x + t and
  0 <= itx - itt < {tile_sz} and 0 <= itt + itx < {tile_sz} and
  0 <= tparity < 2 }}
  """)

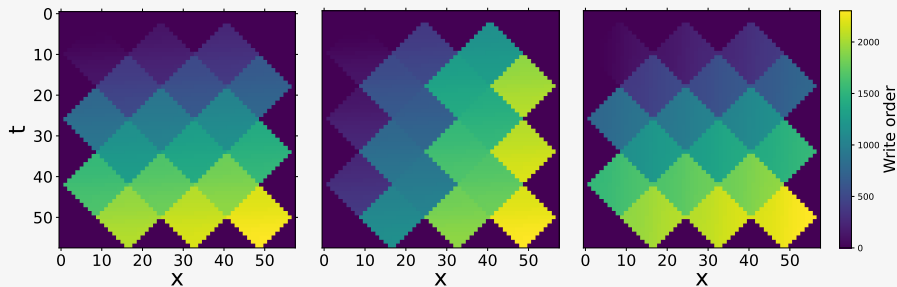
knl = lp.map_domain(knl, transform_map)
```

Experimental Results: FD Wave Equation Solution



Transformed loop indices
for diamond tiling

Experimental Results: FD Wave Equation Solution



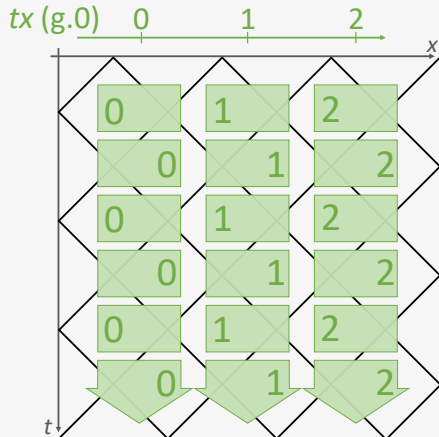
Correct loop nest:
(tt, tparity, tx, itt, itx)

Incorrect loop nest:
(tx, tt, tparity, itt, itx)

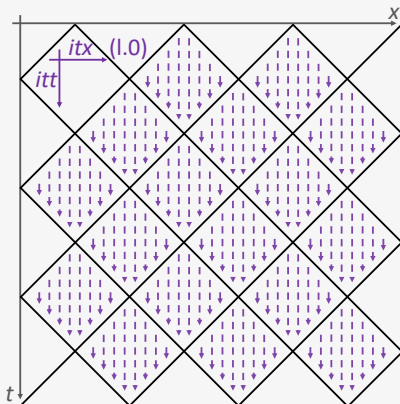
Incorrect loop nest:
(tt, tparity, tx, itx, itt)

Access pattern visualization showing index write orders in solution array

Experimental Results: FD Wave Equation Solution



```
# Parallelize tx across groups; add barrier
knl = lp.tag_inames(knl, {"tx": "g.0"})
knl = lp.add_barrier(knl,
    within_inames=frozenset(["tt", "tparity"]),
    synchronization_kind="global", ...)
```



```
# Parallelize itx across work-items within group
knl = lp.tag_inames(knl, {"itx": "l.0"})
knl = lp.add_barrier(knl,
    within_inames=frozenset(["tt", "tparity", "itt"]),
    synchronization_kind="local", ...)
```

Experimental Results: FD Wave Equation Solution



Add constraints on loop nest structure

```
# Add constraints on loop nest structure  
knl = lp.constrain_loop_nesting(knl, must_nest="tt,tparity,itt")
```

Check for dependency violations

```
# Linearize kernel  
lin_knl = lp.get_one_linearized_kernel(lp.preprocess_kernel(knl))  
  
# Find any dependency violations  
unsatisfied_deps = lp.find_unsatisfied_dependencies(lin_knl)
```

View transformed dependency

```
# Print dependency  
print(knl.id_to_insn["stmt"].dependencies["stmt"][0])
```

```
[..., tt', tparity', tx', itt', itx'] -> [..., tt, tparity, tx, itt, itx]: ...
```

Experimental Results: FD Wave Equation Solution



nx	nt	Tile Width	Wall Time (ms)	GFLOP/s	Bandwidth (GB/s) ¹⁶	Bandwidth % of Peak
36896	36896	64	43.3	617.7	251.0	38.5
40992	40992	64	52.8	625.3	254.1	38.9
45088	45088	64	63.7	627.7	255.1	39.1
36928	36928	128	42.8	629.8	253.9	38.9
41024	41024	128	48.8	681.6	274.8	42.1
45120	45120	128	55.2	729.5	294.1	45.1
36992	36992	256	39.0	694.7	279.0	42.7
41088	41088	256	43.8	762.2	306.1	46.9
45184	45184	256	50.0	809.4	325.0	49.8

Performance of transformed kernel on Nvidia Titan V GPU

Theoretical peak bandwidth (GB/s): 653

Peak 32-bit GFLOP/s: 12,288

Op/mem. stats gathered using counting mechanisms discussed in next section

¹⁶ Lower bound calculated using footprint of accessed data

Experimental Results: LU NAS Parallel Benchmark



Lower-Upper Symmetric Gauss-Seidel NASA Advanced Supercomputing Parallel Benchmark¹⁷

- Simulated CFD application
- Successive over-relaxation to solve block-diagonal system from FD discretization of Navier-Stokes equations in 3-D
 - Six 5×5 off-diagonal blocks, three left, three right

```

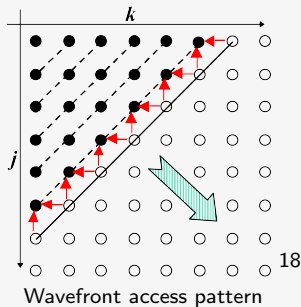
do k = kst, kend
  ...
  do j = jst, jend
    ! Form lower triangular part of Jacobian
    call jacl(j, k)
    ! Compute lower triangular solution
    call blts(..., j, k)
  end do
  ...
end do

```

Demonstrated subcomponent of LU benchmark

¹⁷ [JFY99] ¹⁸ [JFY99]

- Each Newton step, solve $Ax = b$ w/SSOR
- $A = M - N$; $M = (1/\omega)\dot{D} + L$;
 $N = (1/\omega - 1)\dot{D} - U$
- Each SSOR step:
block- $\{lower, upper\}$ -triangular solve



Experimental Results: LU NAS Parallel Benchmark

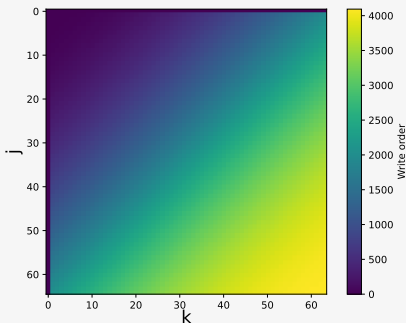


- LOOPY ingests Fortran subroutines
- Over 500 unique dependency pairings in JACLD+BLTS; examples:

```
# Create dependency map for BLTS usage of JACLD results
dep_ijk_eq = make_dep_map(
  "[... { [i_jacld', j', k'] -> [i_blts, j, k, m] : " # Map space
  "i_blts = i_jacld' and j = j' and k = k' }", # Constraints
  knl_with_domains=knl)
```

```
# Create dependency map for BLTS usage of result at previous k index
dep_k_incr = make_dep_map(
  "... { [i_blts', j', k'] -> [i_blts, j, k, m] : " # Map space
  "i_blts = i_blts' and j = j' and k = k' + 1 }", # Constraints
  knl_with_domains=knl)
```

Experimental Results: LU NAS Parallel Benchmark



```
# Create mapping from current indices to indices in wavefront ordering
transform_map = isl.BasicMap(
    "[jend, kend] -> "
    "{ [j,k] -> [wave, wave_inner] : wave = j + k and wave_inner = j }")
knl = lp.map_domain(knl, transform_map)

# Ensure no loop nests outside 'wave' loop
knl = lp.constrain_loop_nesting(knl, must_not_nest="~wave, wave")
```

Experimental Results: LU NAS Parallel Benchmark



```
# Parallelize diagonal wave front across work-groups
knl = lp.tag_inames(knl, "wave_inner:g.0")

# Add barrier after last statement in BLTS, within wave loop
# (keeps groups synchronized in lock-step)
knl = lp.add_barrier(knl,
    within_inames=frozenset(["wave", ]), synchronization_kind="global",
    insn_before="id:s69_write_v_ijk_blts", insn_after=None)

... # (further transformation)
```

Experimental Results: LU NAS Parallel Benchmark



Domain Width $n_x = n_y = n_z$	Wall Time (ms)	GFLOP/s	Bandwidth (GB/s) ¹⁹	Bandwidth % of Peak
224	501.0	20.5	72.3	11.1
256	662.3	23.2	82.0	12.6
288	847.5	24.2	91.4	14.0
320	1060.5	28.4	100.4	15.4

Nvidia Titan V GPU

Theoretical peak bandwidth (GB/s): 653 Peak 32-bit GFLOP/s: 6,144

Examples demonstrated:

- How dependency representation is exposed to user
- Automatic transformation of deps as transformations applied
- Dependency, program order semantics enable violation detection
- Expression, enforcement of precise constraints on loop structure
- Ingestion of complex, application-level benchmark
 - Application of semantics therein

¹⁹ Apply different perf. model than stencil example: model each access as it happens

A Mechanism for Balancing Accuracy and Scope in Cross-Machine Black-Box GPU Performance Modeling

Publication: [SK20]

Goal

Model, interpret, predict execution times in an automated,
architecture-independent fashion

Approach

Model execution time, or any *kernel feature*, as *user-defined* function of kernel features and parameters

$$T_{\text{wall}}(\mathbf{n}) = \text{feat}^{\text{out}}(\mathbf{n}) \approx g(\text{feat}_0^{\text{in}}(\mathbf{n}), \dots, \text{feat}_j^{\text{in}}(\mathbf{n}), p_0, \dots, p_k)$$

- \mathbf{n} : (Runtime-constant) domain size parameters
- $\text{feat}_i^{\text{in}}(\mathbf{n})$: Count of quantitative kernel characteristic
(e.g., number of 32-bit floating point multiplications)
- p_i : Machine-dependent parameter relating features to exec time,
found by fitting model to microbenchmark data
- g : Function provided by user; differentiable w.r.t. parameters

Summary of Methodology and Results

Three workload cost categories, sums of features weighted by cost params:

c_{gmem} : global memory access

$c_{\text{on-chip}}$: on-chip work, i.e., local/scratchpad memory access and arithmetic

c_{overhead} : barrier, kernel launch, and work-group launch costs

Linear model:

$$t \approx c_{\text{overhead}} + c_{\text{gmem}} + c_{\text{on-chip}}$$

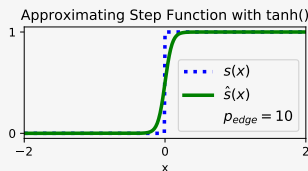
Nonlinear model: Introduced approach to model overlap of on-chip and global memory operation costs

$$t \approx c_{\text{overhead}} + c_{\text{gmem}} \cdot \hat{S}(c_{\text{gmem}} - c_{\text{on-chip}}) + c_{\text{on-chip}} \cdot \hat{S}(c_{\text{on-chip}} - c_{\text{gmem}})$$

Summary of Methodology and Results

$$\hat{s}(x) = (\tanh(p_{\text{edge}} \cdot x) + 1)/2$$

$$\approx s(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$



p_{edge} : regulates “abruptness” of step; determined with other params

$$t \approx c_{\text{overhead}} + c_{\text{gmem}} \cdot \hat{s}(c_{\text{gmem}} - c_{\text{on-chip}}) + c_{\text{on-chip}} \cdot \hat{s}(c_{\text{on-chip}} - c_{\text{gmem}})$$

$$\approx c_{\text{overhead}} + \max(c_{\text{gmem}}, c_{\text{on-chip}})$$

Modeling Overlap of Local, Global Memory Transactions

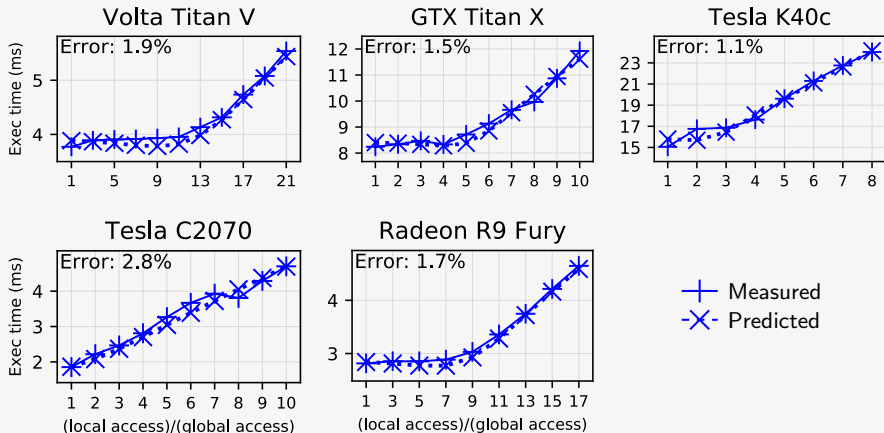


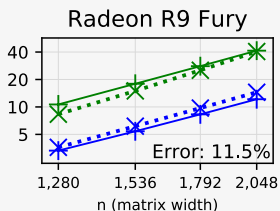
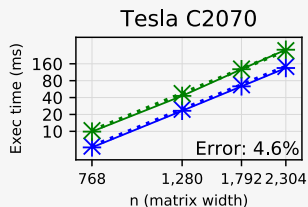
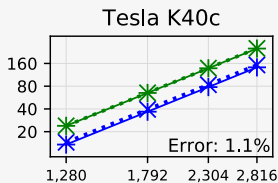
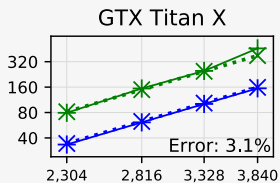
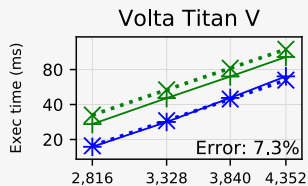
Table displays the geometric mean of relative error (%). Array size differs across GPUs.

Summary of Methodology and Results

Evaluation:

- DG, matrix-matrix multiplication, finite differences computations
- 3 computations, 5 GPUs, 6.4% geomean relative error
- Predictions correctly rank all variants by execution time in 13/15 cases
- Demonstrated insight into computation cost gained from modelling (see [SK20])

Matrix Multiplication Model Accuracy

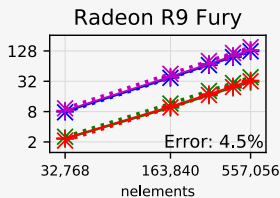
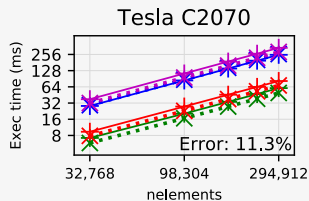
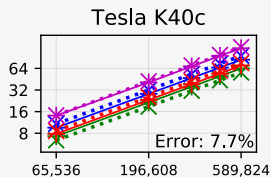
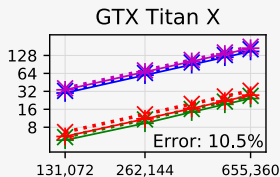
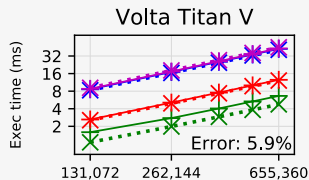


Variant	TiV	TiX	K40	2070	Fury
PF	3.0	2.6	7.4	6.0	14.1
NoPF	18.0	3.7	0.2	3.6	9.3
Mean	7.3	3.1	1.1	4.6	11.5

— Prefetch — No prefetch
 + Measured -X- Predicted

Table displays the geometric mean of relative error (%).

DG Differentiation Model Accuracy

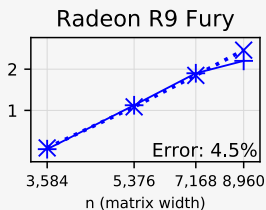
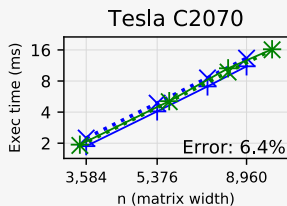
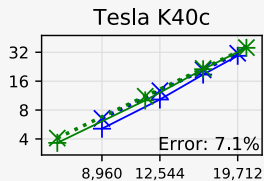
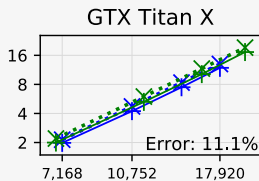
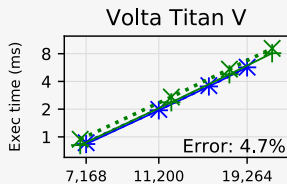


Variant	TiV	TiX	K40	2070	Fury
PFdm	3.6	8.6	13.3	3.3	0.4
PFdm ^T	29.1	12.4	12.3	18.5	19.8
PFu	2.8 ^L	19.1	9.4 ^L	16.9 ^L	4.5
NoPF	4.2	6.0	2.3	15.8	13.2
Mean	5.9	10.5	7.7	11.3	4.5

— No prefetch — Prefetch diff mat
— Prefetch u — Prefetch diff mat^T
+ Measured X Predicted

Table displays the geometric mean of relative error (%). ^TElement data transposed. ^LLinear model used (otherwise nonlinear).

Finite Differences Model Accuracy



Variant	TiV	TiX	K40	2070	Fury
16x16	1.6	10.1	14.8	20.2	4.5
18x18	14.0	12.2	3.4	2.1	
Mean	4.7	11.1	7.1	6.4	4.5

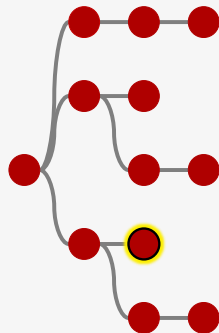
— 16x16 — 18x18
 + Measured -X- Predicted

Table displays the geometric mean of relative error (%).

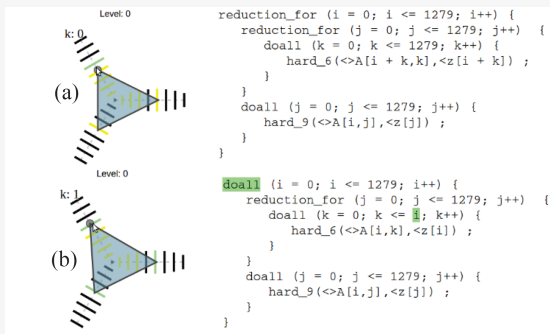
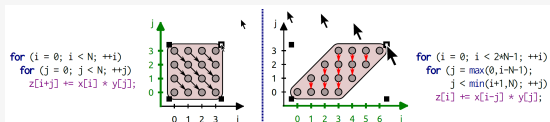
A Visual User Interface for Code Transformation, Analysis, and Optimization

Motivation

- Composing transform chain is technical and involves experimentation
- Goal: Make this easier and more interactive



Related Work



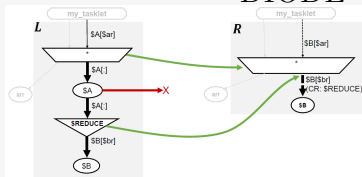
PUMA-V²⁰

Related work details in Appendix 6 of presentation and Section 5.1.1 of [dissertation]

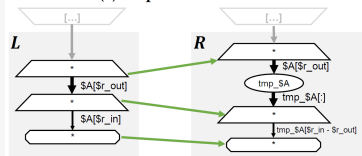
²⁰ [PML⁺16, PLM⁺19] ²¹ [ZHB14], [ZHB18] ²² [BNdFLZ⁺19]

CLINT²¹

DIODE²²



(a) Map-Reduce Fusion



(b) Local Storage

Building on Related Work

Our approach:

- Fine-grained search space w/ integrated stats, transform info
 - FLOP/s, mem. throughput, exec. time
- Transform via direct source manipulation (without manual editing)
 - Representation already understood
 - Observations of source often impetus for transformation
- Easy deployment of (maintainable) transformed result
 - Inherits advantages of LOOPY system, representation

Related work details in Appendix 6 of presentation and Section 5.1.1 of [dissertation]

Key System Criteria



C-level source code, transformation code should not be manually written

- Reduce development time, errors; focus on higher-level decisions

Approach:

- Describe program intent with high level math
- Use LOOPY's automated source code generation
- Transform via direct source interaction without manual coding
- Deploy source and (maintainable) source-generating transform script

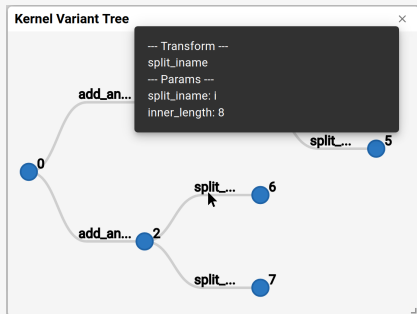
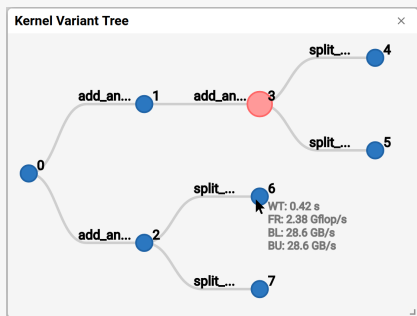
The screenshot displays the LOOPY interface with three main panels:

- Kernel Define:** Contains buttons for 'RESTART' and 'NEW KERNEL'.
- Transform Kernel:** Features a dropdown menu for 'add_and_infer_dtypes', input fields for 'variable' and 'data type', and a 'SUBMIT' button.
- Target Source Code:** Shows the generated C code for a kernel. A red box highlights the 'split_iname' function call in the loop, with a dropdown menu showing options: 'split_iname', 'tag_inames', 'duplicate_inames', and 'rename_iname'. A mouse cursor is pointing at 'split_iname'.
- Kernel Variants:** A graph showing nodes for 'add_an...', 'split...', and '+2', connected by arrows.

Key System Criteria

Navigable search space of program variants

- Compare *multiple* optimization paths
- Reason about performance consequences



Approach:

- Fine-grained, interactive space including performance stats, transform info
- Stats: Exec. time, FLOP/s, memory throughput

Evaluation of Design Objectives

The image displays a software interface for kernel transformation, divided into several panels:

- Kernel Define:** Contains buttons for "RESTART" and "NEW KERNEL".
- Transform Kernel:** A dropdown menu is open, showing options like "add_and_infer_dtypes", "Frequently Used", "Dealing with Parameters", "Wrangling Inames" (highlighted), "split_iname" (highlighted), "tag_inames", "duplicate_inames", "prioritize_loops", "rename_iname", "split_reduction_inward", "split_reduction_outward", "Caching, Precomputation and Prefetching", "Modifying Arguments", "Influencing Data Access", "Padding Data", "Substitution Rules", "Manipulating Instructions", and "Other".
- Target Source Code (Left):** Shows C-level code with a loop: `for ^ (int i = 0; i <= -1 + n; ++i) a[i] = b[i]`. A red box highlights the loop index `i`.
- Target Source Code (Right):** Shows the same code after transformation, with a "loop optimizations" label and the transformed loop: `for ^ (int i = 0; i <= -1 + n; ++i) a[split_iname]`. A red box highlights the transformed loop index `i`.
- Kernel Variant Tree (Bottom):** A graph showing the transformation process. A red node labeled "add_an..." with a "1" is connected to two blue nodes labeled "split..." with "2" and "7". The "2" node is further connected to a node with "+2", and the "7" node is connected to a node labeled "assu".

Immediacy of Interaction with C-level Source Code

- Lower action counts for transformation via direct code manipulation

Segregated Transform Menu and Direct Code Manipulation

Transform Kernel

- add_and_infer_dtypes
- > Frequently Used
- > Dealing with Parameters
- > Wrangling Inames
 - split_iname
 - tag_inames
 - duplicate_inames
 - prioritize_loops
 - rename_iname
 - split_reduction_inward
 - split_reduction_outward
- > Caching, Precomputation and Prefetching
- > Modifying Arguments
- > Influencing Data Access

```

_kernel void __attribute__((reqd_work_group_size(1, 1, 1)))
example(__global float *__restrict__ a,
        __global float const *__restrict__ b, int
        const n)
{
    for ^ (int i = 0; i <= -1 + n; ++i )
        a[i] = b[i]
}

```

Kernel Variant Tree

```

_kernel void __attribute__((reqd_work_group_size(1, 1, 1)))
example(__global float *__restrict__ a,
        __global float const *__restrict__ b, int
        const n)
{
    loop optimizations
    for ^ (int i = 0; i <= -1 + n; ++i )
        a[i]
}

```

Kernel Variant Tree

Transform Kernel

split_iname

iname

inner length

16

SUBMIT

Gather Stats

paramDict

{'n': 25600}

```

_kernel void __attribute__((reqd_work_group_size(1, 1, 1)))
example(__global float *__restrict__ a,
        __global float const *__restrict__ b, int
        const n)
{
    for ^ (int i = 0; i <= -1 + n; ++i )
        a[i] = b[i]
}

```

Kernel Variant Tree

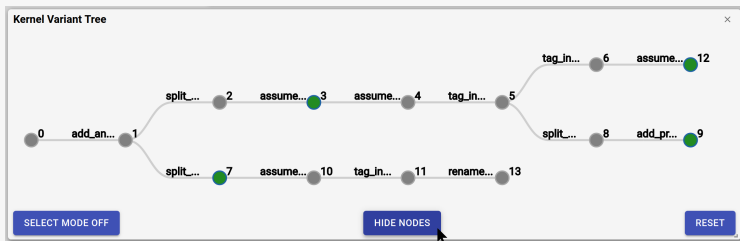
```

_kernel void __attribute__((reqd_work_group_size(1, 1, 1)))
example(__global float *__restrict__ a,
        __global float const *__restrict__ b, int
        const n)
{
    loop optimizations
    for ^ (int i = 0; i <= -1 + n; ++i )
        a[i]
}

```

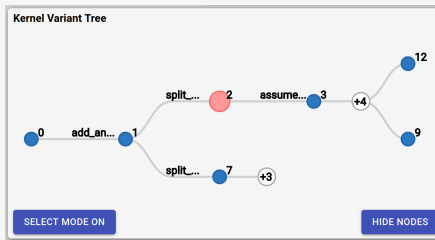
Kernel Variant Tree

Evaluation of Design Objectives



v : variant ct

v^* : unhidden
variant ct



Scalability w.r.t. Variant/Line Count

- Collapse segments to reduce action count scaling factor: $\mathcal{O}(v^*) \ll \mathcal{O}(v)$
- Collapse source code blocks $\mathcal{O}(l^*) \ll \mathcal{O}(l)$

Evaluation of Design Objectives

Applicability to Programs In Situ

```
from app import LoopyUIApp
LoopyUIApp(initial_kernel=kn1, open_browser=True)
```

Reproducibility of Experiments

- Save, reload complete search space of program variants

Deployability to Production Code

- Deploy full transformation script
 - Enables further modification of transform chain
- Deploy source code

Evaluation of Design Objectives



Research Goal	Function	Action Count
Immediacy of interaction with the code	Bring arbitrary non-hidden code element into view	$\mathcal{O}(l^*) \cdot q + p$
	Unhide arbitrary previously hidden code section	$\mathcal{O}(l^*) \cdot q + 2p$
	<i>Once the relevant code element is in view:</i>	
	Apply tag-iname transform via segregated input form	$8 \cdot (p_{IOVR} + s) + 2t$
	Apply tag-iname transform via code manipulation	$5 \cdot (p + s) + t$
	Apply split-iname transform via segregated input form	$7 \cdot (p_{IOVR} + s) + 2t$
	Apply split-iname transform via code manipulation	$4 \cdot (p + s) + t$
Immediacy of interaction with the search space	Bring arbitrary non-hidden kernel node into view	$(2p + s) \cdot \mathcal{O}(v^*)$
	<i>Once the relevant kernel node or set of nodes is in view:</i>	
	Select non-hidden kernel node	$p + s$
	Unhide a hidden subset of search space represented by proxy node	$p + s$
	Display parent transform details for non-hidden kernel node	p
	Display previously gathered statistics for non-hidden kernel node	p
	Hide contiguous section of search tree bounded by k kernel nodes	$2(p_{IOVR} + s) + k \cdot (p + s)$
Collect statistics for selected code variant	$2(p_{IOVR} + s) + t$	
Reproducibility of experiments	Save search space	$2p_{IOVR} + s + t$
	Load saved search space	$2p_{IOVR} + 3s$
Ease of deployability to production code	Deploy Python script	$2p_{IOVR} + s + t$

p : Position action; s : Select action; t : Text action; q : Quantify action; (Action types commonly counted in HCI²³)

p_{IOVR} : Independent-of-visual-representation position action;

l : Lines of source code in selected variant pre-actions; l^* : Non-hidden lines of source code in selected variant pre-actions ($l^* \leq l$);

v : Number of variants pre-actions; v^* : Number of un-hidden variants pre-actions ($v^* \leq v$);

Action counts required to perform selected UI functions ²³ [FVVD⁺96]
 (starting in *worst* scenario, e.g., target in a list always requires typing to bring into view)

A Real-World Use Case



- Optimize kernel presented in [KWW16]
- Computes volume term in semi-discretization of Euler's equations in weather model
- Complex transformation chain
 - Kernel fusion, vectorization, prefetching, parallelization, more
 - 50 statements; 125 transformations (14 different kinds)

A Real-World Use Case



N_q	N_e	Wall Time (ms)	GFLOP/s	Bandwidth (GB/s) ²⁴	Bandwidth % of Peak
8	6910	0.8	761.3	552.1	84.6
8	13820	1.5	795.5	576.9	88.4
8	20730	2.2	810.5	587.8	90.0
8	27640	3.0	818.3	593.4	90.9
8	34550	3.7	822.1	596.2	91.3

Performance of (optimization level 8) kernel for different numbers of elements

Peak 32-bit GFLOP/s: 12,288

Theoretical peak bandwidth (GB/s): 653

²⁴ Lower bound calculated using footprint of accessed data



Loopy UI



Reset panel positions: All Define Transform Code Stats Tree

Kernel Define

RESTART NEW KERNEL

Transform Kernel

add_prefetch

fetched var.
b

sweep iname(s)
l_inner x

default tag
Select...

SUBMIT

Gather Stats

paramDict
{'n': 25600}

SUBMIT

Target Source Code

IR SOURCE SAVE OPENCL GENERATE PYTHON

```

__kernel void __attribute__((reqd_work_group_size(1, 1, 1)))
example(__global float *__restrict__ a, __global float const *__restrict__
b, int const n)
{
    for ^ (int i_outer = 0; i_outer <= ((-16 + n) / (16)); ++i_outer )
        for ^ (int i_inner = 0; i_inner <= 15; ++i_inner )
            a[16*i_outer + i_inner] = b[16*i_outer + i_inner]
}

```

loop optimizations

- split_iname
- tag_inames
- duplicate_inames
- rename_iname

Kernel Variant Tree

SELECT MODE ON HIDE NODES RESET

Summary of Key Contributions



Programming System Semantics

- Human-comprehensible, verifiable, statement-instance-level deps
- Novel procedure for verifying correctness of generated prog. in generalized OpenCL comp. abstraction w/local, global barriers
- Linearization proc. using coarse-grained, statement-level ordering heuristic
- Human-comprehensible, enforceable, loop-nesting semantics





Transformation UI

- Navigable search space representation w/integrated transform, perf. info
- Transformation via direct code interaction
- Deployment of modifiable transformation, generation script
- Applicability to program 'in situ'




Black-Box Performance Modeling

- Broad customization of mathematical model not available in previous work
- Broad customization of set of measurement computations used to calibrate model
- Automated gathering of precise pre-compilation, parameterized operation counts, kernel features
- Hardware-agnostic modeling




Bibliography I

-  Peter Adshead, John T. Giblin, Mauro Pieroni, and Zachary J. Weiner, *Constraining axion inflation with gravitational waves across 29 decades in frequency*, Phys. Rev. Lett. **124** (2020), 171301.
-  _____, *Constraining axion inflation with gravitational waves from preheating*, Phys. Rev. D **101** (2020), 083534.
-  Cédric Bastoul, *Code generation in the polyhedral model is easier than you think*, PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques (Juan-les-Pins, France), September 2004, pp. 7–16.
-  U. Bondhugula, V. Bandishti, and I. Pananilath, *Diamond tiling: Tiling techniques to maximize parallelism for stencil computations*, IEEE Transactions on Parallel and Distributed Systems **28** (2017), no. 5, 1285–1298.


Bibliography II

-  Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam, *Putting polyhedral loop transformations to work*, LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958 (College Station, Texas), October 2003, pp. 209–225.
-  Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan, *A practical automatic polyhedral parallelizer and locality optimizer*, SIGPLAN Not. **43** (2008), no. 6, 101–113.
-  Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler, *Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures*, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (New York, NY, USA), SC '19, Association for Computing Machinery, 2019.




Bibliography III

-  Ian J. Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L. Chamberlain, David G. Wonnacott, and Michelle Mills Strout, *Parameterized diamond tiling for stencil computations with chapel parallel iterators*, Proceedings of the 29th ACM on International Conference on Supercomputing (New York, NY, USA), ICS '15, Association for Computing Machinery, 2015, p. 197–206.
-  V. Bandishti, I. Pananilath, and U. Bondhugula, *Tiling stencil computations to maximize parallelism*, SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1–11.
-  Chun Chen, Jacqueline Chame, and Mary Hall, *CHiLL: A framework for composing high-level loop transformations*, Tech. report, Citeseer, 2008.




Bibliography IV

-  Nicholas Christensen, *High Performance Discontinuous Galerkin with Grudge*, (in progress) (2021).
-  Nicholas J. Curtis, Kyle E. Niemeyer, and Chih-Jen Sung, *Using simd and simt vectorization to evaluate sparse chemical kinetic jacobian matrices and thermochemical source terms*, *Combustion and Flame* **198** (2018), 186 – 204.
-  Isuru Fernando, *Automatic Synthesis of Low Complexity Translation Operators for the Fast Multipole Method*, (in progress) (2021).
-  James D Foley, Foley Dan Van, Andries Van Dam, Steven K Feiner, John F Hughes, Edward Angel, and J Hughes, *Computer graphics: principles and practice*, vol. 12110, Addison-Wesley Professional, 1996.




Bibliography V

-  Tobias Grosser, Armin Groesslinger, and Christian Lengauer, *Polly—performing polyhedral optimizations on a low-level intermediate representation*, *Parallel Processing Letters* **22** (2012), no. 04, 1250010.
-  François Irigoin, Pierre Jouvelot, and Rémi Triolet, *Semantical interprocedural parallelization: An overview of the pips project*, *Proceedings of the 5th International Conference on Supercomputing (New York, NY, USA), ICS '91*, Association for Computing Machinery, 1991, p. 244–251.
-  Haoqiang Jin, Michael Frumkin, and Jerry Yan, *The openmp implementation of nas parallel benchmarks and its performance*, Tech. report, Citeseer, 1999.





Bibliography VI

-  Dominic Kempf, René Heß, Steffen Müthing, and Peter Bastian, *Automatic code generation for high-performance discontinuous galerkin methods on modern architectures*, 2018.
-  Andreas Kloeckner, *Loo.Py: Transformation-based Code Generation for GPUs and CPUs*, Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (New York, NY, USA), ARRAY'14, ACM, 2014, DOI: 10.1145/2627373.2627387, pp. 82:82–82:87.
-  _____, *Loo.Py: From Fortran to Performance via Transformation and Substitution Rules*, Proceedings of the 2Nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (New York, NY, USA), ARRAY 2015, ACM, 2015, DOI: 10.1145/2774959.2774969, pp. 1–6.




Bibliography VII

-  Kaushik Kulkarni, *UFL to GPU Near the Roofline*, (in progress) (2021).
-  Andreas Klöckner, Lucas C. Wilcox, and T. Warburton, *Array program transformation with loo.py by example: High-order finite elements*, Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (New York, NY, USA), ARRAY 2016, ACM, 2016, DOI: 10.1145/2935323.2935325, pp. 9–16.
-  U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso, *A survey of performance modeling and simulation techniques for accelerator-based computing*, IEEE Transactions on Parallel and Distributed Systems **26** (2015), no. 1, 272–281.

Bibliography VIII

-  Christophe Mauras, *Alpha: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*, Ph.D. thesis, L'Université de Rennes 1, December 1989.
-  Benoit Meister, Allen Leung, Nicolas Vasilache, David Wohlford, Cédric Bastoul, and Richard Lethin, *Productivity via automatic code generation for pgas platforms with the r-stream compiler*, Workshop on Asynchrony in the PGAS Programming Model, 2009.
-  Souley Madougou, Ana Varbanescu, Cees de Laat, and Rob van Nieuwpoort, *The landscape of gpgpu performance modeling tools*, *Parallel Computing* **56** (2016), 18 – 33.
-  Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin, *R-stream compiler.*, 2011.

Bibliography IX

-  Louis-Noel Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, R. Ramanujam, and P. Sadayappan, *Hybrid Iterative and Model-Driven Optimization in the Polyhedral Model*, Research Report RR-6962, INRIA, 2009.
-  E. Papenhausen, M. H. Langston, B. Meister, R. A. Lethin, and K. Mueller, *Puma-v: Optimizing parallel code performance through interactive visualization*, IEEE Computer Graphics and Applications **39** (2019), no. 1, 84–99.
-  E. Papenhausen, K. Mueller, H. Langston, B. Meister, and R. Lethin, *Puma-v: An interactive visual tool for code optimization and parallelization based on the polyhedral model*, 2016 New York Scientific Data Summit (NYSDS), 2016, pp. 1–4.




Bibliography X

-  James Stevens and Andreas Klöckner, *A mechanism for balancing accuracy and scope in cross-machine black-box gpu performance modeling*, The International Journal of High Performance Computing Applications (2020), 589–614.
-  Tianjiao Sun, Lawrence Mitchell, Kaushik Kulkarni, Andreas Klöckner, David A Ham, and Paul HJ Kelly, *A study of vectorization for matrix-free finite element methods*, The International Journal of High Performance Computing Applications **34** (2020), no. 6, 629–644.
-  Sven Verdoolaege, Albert Cohen, and Anna Beletka, *Transitive closures of affine integer tuple relations and their overapproximations*, Proceedings of the 18th International Conference on Static Analysis (Berlin, Heidelberg), SAS'11, Springer-Verlag, 2011, pp. 216–232.

Bibliography XI

-  Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor, *Polyhedral parallel code generation for cuda*, ACM Trans. Archit. Code Optim. **9** (2013), no. 4, 54:1–54:23.
-  Tomofumi Yuki, Vamshi Basupalli, Gautam Gupta, Guillaume looss, D Kim, Tanveer Pathan, Pradeep Srinivasa, Yun Zou, and Sanjay Rajopadhye, *Alphaz: A system for analysis, transformation, and code generation in the polyhedral equational model*, Colorado State University, Tech. Rep (2012).
-  Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye, *Alphaz: A system for design space exploration in the polyhedral model*, Languages and Compilers for Parallel Computing (Berlin, Heidelberg) (Hironori Kasahara and Keiji Kimura, eds.), Springer Berlin Heidelberg, 2013, pp. 17–31.

Bibliography XII

-  O. Zinenko, S. Huot, and C. Bastoul, *Clint: A direct manipulation tool for parallelizing compute-intensive program parts*, 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2014, pp. 109–112.
-  Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul, *Visual program manipulation in the polyhedral model*, ACM Trans. Archit. Code Optim. **15** (2018), no. 1, 1–25.
-  Huihui Zhang, Anand Venkat, Protonu Basu, and Mary Hall, *Combining Polyhedral and AST Transformations in CHiLL*, Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques, IMPACT, vol. 16, 2016.

Additional Image Sources

Introduction: [1], [2], [3]

LOOPY's Model of a Program


- Unordered list of **statements**, which operate on multidimensional arrays
 - Assignment to array entry; RHS expression containing arithmetic, function calls
 - Parameterized by set of loop variables: **inames**
 - Executed once per integer point in iteration domain defined by *domain forest* for inames
 - Function calls may return tuples
 - LOOPY defines callable functions; additional funcs may be defined
 - Assignments may be atomic; recursion not permitted
- **Arrays**
 - n -D array shape defined by n -D tuple of expressions
 - Affine in size parameters, which are fixed during execution
 - Argument (accessible outside prog.); temporary var. (live only in prog.)
- **Domain Forest**
 - Made of *domains*, sets defined by conjunctions of inequalities of quasi-affine expressions of parameters or inames
 - Domains may have *parent domains*
 - Enable imperfectly nested and data-dependent loops
 - Domain without parent: parameters passed as program arguments, fixed during execution
 - Domain with parent: params may be inames from parent, or scalar, integer temp vars written by statements within domains defined by parent domain

More info in documentation, and [Klo14], [Klo15], and [KWW16]

Examples of Successful LOOPY Application

LOOPY's efficacy in enabling high-performance transformation and code generation demonstrated in multiple previous and in-progress applications:

- For cross-element²⁵ and intra-element²⁶ vectorization in FIREDRAKE, automated system for portable solution of PDEs using finite element method
- For intra-element vectorization in DUNE PDE software framework²⁷
- For solving PDEs using finite differences method in PDE solving framework PYTELLA²⁸
- For chemical kinetics in PYJAC²⁹, code-generating utility for analytically calculating chemical kinetics Jacobian matrices
- For solving PDEs using discontinuous Galerkin³⁰ (DG) method in GRUDGE, an unstructured, high-order, parallel DG solver, which is being used to facilitate high-performance scramjet simulations³¹
- For automatic synthesis of translation operators for fast multipole method³²

²⁵ [SMK⁺20] ²⁶ [Kul21] ²⁷ [KHMB18] ²⁸ [AGPW20a, AGPW20b] ²⁹ [CNS18]
³⁰  ³¹ [Chr21] ³² [Fer21]

Loop Nest Structure Semantics (further details)

- Within correctness constraints, program order is efficiency concern
- Key ordering concern: nesting structure of loops
 - Affects efficiency: cache, TLB hit rates
 - Loop structure may be prerequisite for transformation, e.g., vectorize
- Previous LOOPY: loop-structure semantics not expressible/enforceable

```
for i
  for j
    for k
      ...
    for g
      ...
  for h
    for r
      ...
```

Loop Nest Structure Semantics (further details)

Need loop nest structure requirements that:

- Are well-defined
- Can be expressed concisely without exhaustively defining full structure
- Can express 'innermost'
- 'Survive' transformations
- Can be checked and enforced

```
for i
  for j
    for k
      ...
    for g
      ...
  for h
    for r
      ...
```

Loop Nest Structure Semantics (further details)

- Express loop nesting structure of linearized program as set N of all nesting pairs (i, j) s.t. loop j nests inside i

$$N = \{(i, j), (i, k), (j, k), (i, g), (h, r)\}$$

- Must-nest** constraint set of pairs C_m and **must-not-nest** constraint set C_n satisfied if:

$$C_m \subseteq N \quad \wedge \quad C_n \cap N = \emptyset$$

```

for i
  for j
    for k
      ...
    for g
      ...
  for h
    for r
      ...

```

Loop Nest Structure Semantics (further details)

Constraint compactness for efficient storage, input, reasoning:

- Single nesting *tier* may contain set of loops; must-nest tuples may contain > 2 tiers

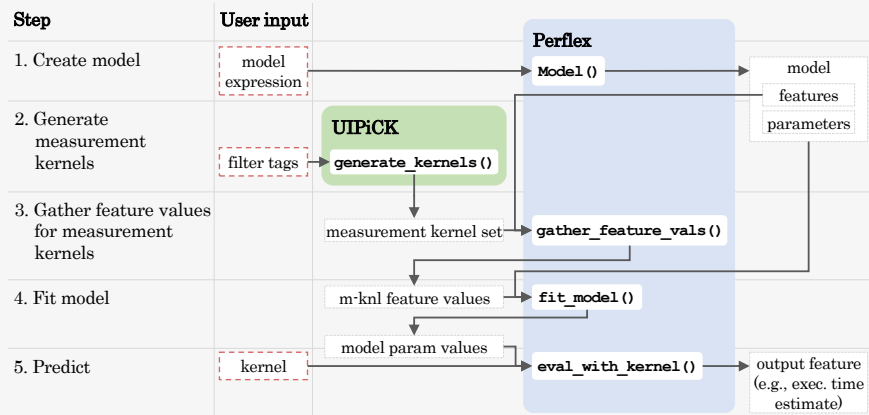
$$(l_1, l_2, \dots, l_n) = \{i_\mu, i_\nu : \mu < \nu, i_\mu \in l_\mu, i_\nu \in l_\nu\}$$

- In must-not-nest constraints, complement sets allowed

$$C_m = (i, \{j, k\}, g) = \{(i, j), (i, k), (i, g), (j, g), (k, g)\}$$

$$C_n = (k, \neg k) = \{(k, i), (k, j), (k, g), (k, h), (k, r)\}$$

Performance Modeling Process Overview



[SK20]

Transformation Interface Design Objectives

Jointly optimize:

- Immediacy of interaction with source
- Immediacy of interaction with search space
- Reproducibility of experiments
 - Search tree and associated metadata
- Ease of deployability without loss of information
- Ease of applicability to in-situ computation
- Scalability w.r.t. transformation count, program size

UI Action Counting

Quantitative, objective metrics for assessing research goals

- Actions³³:
 - *Position**, *Select*, *Text*, *Quantify*
 - *Introduce $position_{IOVR}$: independent of visual representation
- Scaling factors related to program, search space sizes

IR-Source Toggling with UI Application Example

Kernel IR

IR SOURCE SAVE OPENCIL GENERATE PYTHON

```

KERNEL: strongVolumeKernelIR_and_strongVolumeKernelS
-----
ARGUMENTS:
D: type: np.dtype('float32'), shape: (8, 8), dim_tags: (N0:stride:1, N1:stride:8) aspace: global
Q: type: np.dtype('float32'), shape: (i:8, j:8, k:8, field_inner:4, field_outer:2, e:elements), dim
elements: ValueArg, type: np.dtype('int32')
grad0: type: np.dtype('float32'), shape: (8, 8, 8, 3, elements), dim_tags: (N0:stride:1, N1:str
rhs0: type: np.dtype('float32'), shape: (i:8, j:8, k:8, field_inner:4, field_outer:2, e:elements),
volumeGeometricFactors: type: np.dtype('float32'), shape: (8, 8, 8, 11, elements), dim_tags: (N0:s
-----
DOMAINS:
[elements] -> { [i, j, e, k_outer, k_inner, jj, ii, kk, n_UFlux, n_VFlux, n_WFlux, n_RFlux, n_TFlux
{ [0 dim 0, 0 dim 1] : 0 <= 0 dim 0 <= 7 and 0 <= 0 dim 1 <= 7 }
-----
INAME IMPLEMENTATION TAGS:
D dim 0: L.0
D dim 1: L.1
Q dim field_inner: vec
Q dim field_outer: unr
Q dim k: unr
e: g.0
i: L.0
ii: L.0
j: L.1
jj: L.1
k_inner: ilp.unr
k_outer: None
kk: ilp.unr
n_QaFlux: None
n_QvFlux: None
n_RFlux: None
n_TFlux: None
n_UFlux: None
n_VFlux: None
n_WFlux: None
rhs0_init_field_inner: vec
rhs0_init_field_outer: unr
rhs0_init_k: unr
rhs0_store_field_inner: vec
rhs0_store_field_outer: unr
rhs0_store_k: unr
-----

```

Target Source Code

IR SOURCE SAVE OPENCIL GENERATE PYTHON

```

__kernel void __attribute__((reqd_work_group_size(8, 8, 1))) strongVol
const elements, __global float const *__restrict__ volumeGeometricFacto
__global float4 const *__restrict__ Q, __global float const *__restrict
rhsQ)
{
    __local char temp_storage [1024] __attribute__((aligned (8)));
    __local char temp_storage_0 [1024] __attribute__((aligned (8)));
    __local float Q_fetch[8 * 8];
    float Jrx_subst_0[2];
    float Jrz_subst_0[2];
    float Jrz_subst_0[2];
    float Jsx_subst_0[2];
    float Jsx_subst_0[2];
    float Jsx_subst_0[2];
    float Jsx_subst_0[2];
    double P_r_subst_0[2];
    float4 Q_fetch[2 * 2];
    float Qa_r_subst_0[2];
    float Qw_r_subst_0[2];
    float R_r_subst_0[2];
    float T_r_subst_0[2];
    float U_r_subst_0[2];
    float UdotGradR_subst_0[2];
    float UdotGradS_subst_0[2];
    float V_r_subst_0[2];
    float W_r_subst_0[2];
    __local double *const __restrict__ flux_store_0 = (__local double *co
    __local double *const __restrict__ flux_store_1 = (__local double *co
    __local float *const __restrict__ flux_store_10 = (__local float *cor

```

Transformation Framework Related Work (further details)

CHILL³⁴

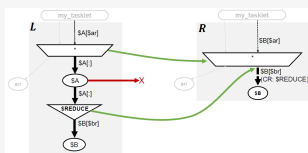
- Source-to-source; C, C++, Fortran
- Automatically generated data dependencies
- Scripting language is stateful, single-pass, w/ inflexible addressing

```

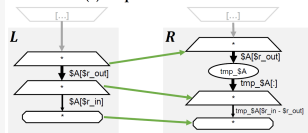
from chill import *
source('dist.c')
procedure('mm')
loop(0, 1)
known(['ambn > 0', 'an > 0', 'bm > 0'])
fuse([0,1], 1)
  
```

DACE³⁵

- Stateful DataFlow multiGraph IR expresses data deps, high-level control flow
 - Manipulate via graph transformations
- Oriented around data flow graph rather than memory, statements



(a) Map-Reduce Fusion



(b) Local Storage

Details in Section 3.1.1 of [dissertation]

³⁴ [CCH08], [ZVBH16] ³⁵ [BNdFLZ+19]

Transformation Framework Related Work (further details)

ALPHAZ³⁶

- Express program as ALPHA³⁷ expressions
 - No representation of state or execution order

```
for (t = 0; t < T; t++)
  for (i = 1; i < N-1; i++)
    A[i] = foo(B[i-1] + B[i] + B[i+1]);
for (i = 1; i < N-1; i++)
  B[i] = A[i];
```



$$A(t, i) = \begin{cases} t=0: & B_{\text{init}}(i); \\ t>0 \leq i < N-1: & \text{foo}(A(t-1, i-1), A(t-1, i), A(t-1, i+1)); \\ t>0 = i: & A(t-1, i); \\ t>0 \wedge i = N-1: & A(t-1, i); \end{cases}$$

- Transformations, array storage specified via multidimensional affine mappings
- Powerful; exert fine-grained control
- Nontrivial, error-prone tasks

Details in Section 3.1.1 of [dissertation]

³⁶ [YGK⁺13], [YBG⁺12] ³⁷ [Mau89]

Performance Modeling Related Work

Related work reviewed in Section 9 of [SK20]

- Two surveys of current GPU performance modeling landscape³⁸
 - Existing GPU performance models predict well for particular application or architecture, but not easily portable
 - Most require (manually gathered) architecture or application info
 - Significant effort to construct, use
- (Non-analytical) learning/statistical techniques more hardware-flexible
 - Less user-accessible design, interpretability
 - Assumptions/limitations about predictive power, fidelity, program/hardware scope less clear
- No significant control over model expression or benchmark design

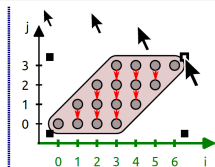
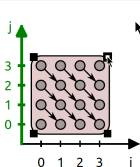
Details in [SK20] and Section 4.1.1 of [dissertation]

³⁸ [MVdLvN16], [LNMMA15]

UI Related Work (further details)

CLINT³⁹

```
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    z[i+j] += x[i] * y[j];
```



```
for (i = 0; i < 2*N-1; ++i)
  for (j = max(0, i-N+1);
       j < min(i+1, N); ++j)
    z[i] += x[i-j] * y[j];
```

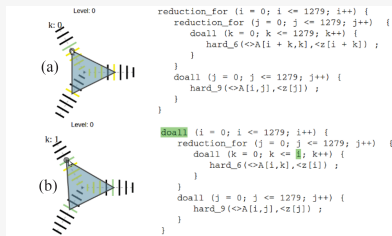
- Source-to-source, extracts polyhedral rep. using CLAN⁴⁰
- Manipulate visualization of iteration domains to transform
- Uses CLOGG⁴¹ to generate C w/OPENMP pragmas
- **1-D history extremely limited, no performance stats**
- **User must learn to interpret/use geometric visualization**
 - Diagram limited in scope
- Transformation via source limited to manual code editing
 - Code interaction has benefits; **possible without manual editing?**
 - Representation already understood
 - Observations of source often impetus for transformation

³⁹ [ZHB14], [ZHB18] ⁴⁰ [BCG+03] ⁴¹ [Bas04]

UI Related Work (further details)

PUMA-V⁴²

- Expose internal transformation process, heuristics of R-STREAM⁴³
- Select (boilerplate) 'tactic' from list
- Apply transformations via multiple geometric visualizations
- Measure execution time
 - Generate C code, outer loops automatically parallelized OPENMP
- **Relies on geometric visualizations for transformation**
 - No interaction with source code
- **History/search space not comprehensive**
 - Tactics only, course-grained
 - Stats (exec. **time only**) not integrated with tree

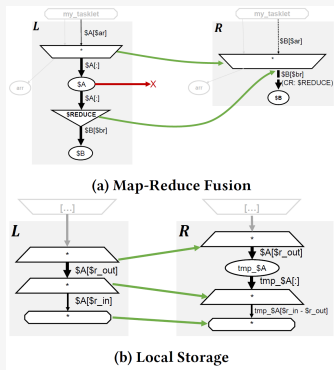


⁴² [PML⁺16, PLM⁺19] ⁴³ [MVW⁺11], [MLV⁺09]

UI Related Work (further details)

DIODE

- Expose Statement DataFlow multiGraph (SDFG) IR used by DACE^{44}
- Select transformation from list
- Select components from SDFG to enter transformation parameters and properties
- Source code not interactive
- 1-D, course-grained history excluding transformation property adjustments



⁴⁴ [BNdFLZ+19]