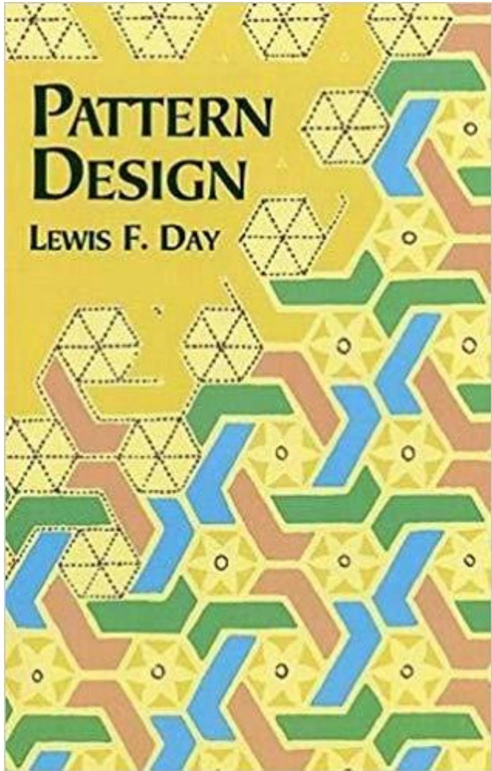
Polyhedral Compilation as a Design Pattern for Compiler Construction

PLISS, May 19-24, 2019

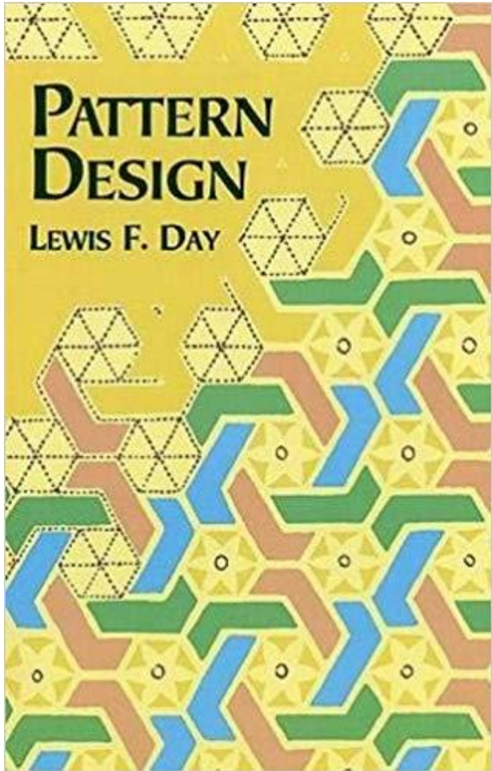
albertcohen@google.com

Polyhedra? Example: Tiles



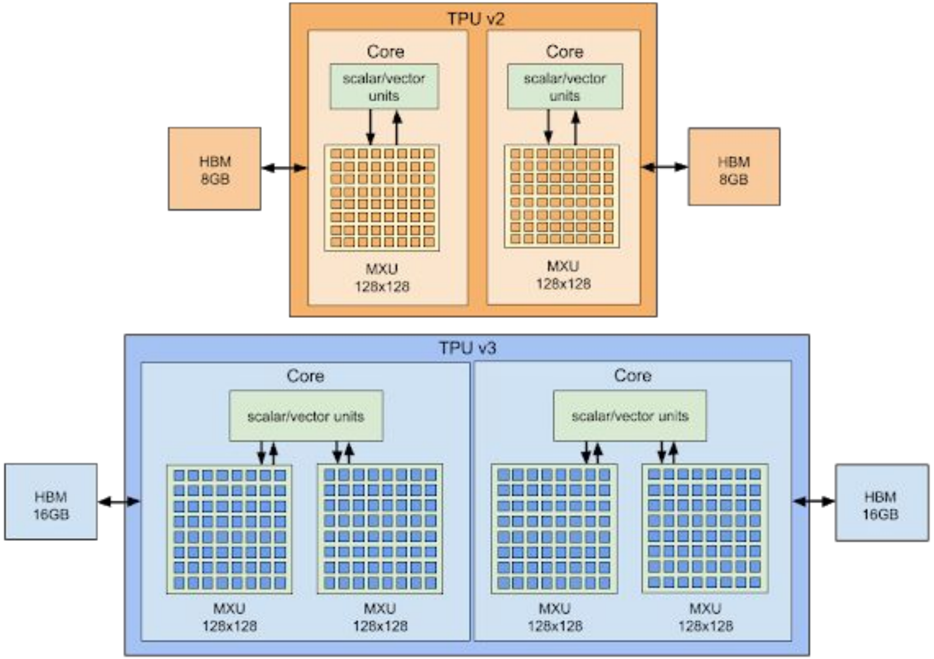
Polyhedra? Example: Tiles

How many of you read **“Design Pattern”**? →



Tiles Everywhere

1. Hardware



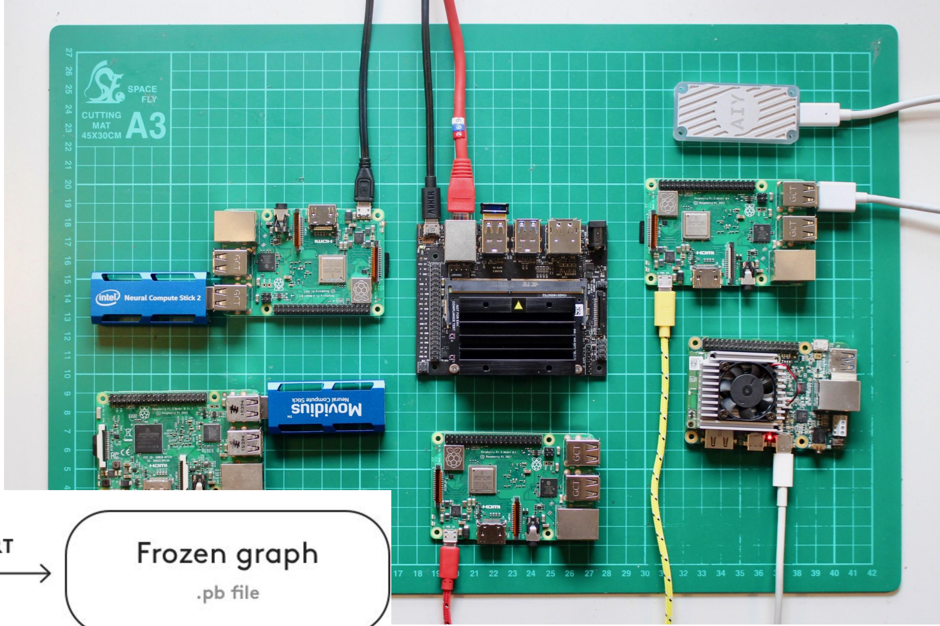
Example: Google Cloud TPU

Architectural Scalability With Tiling

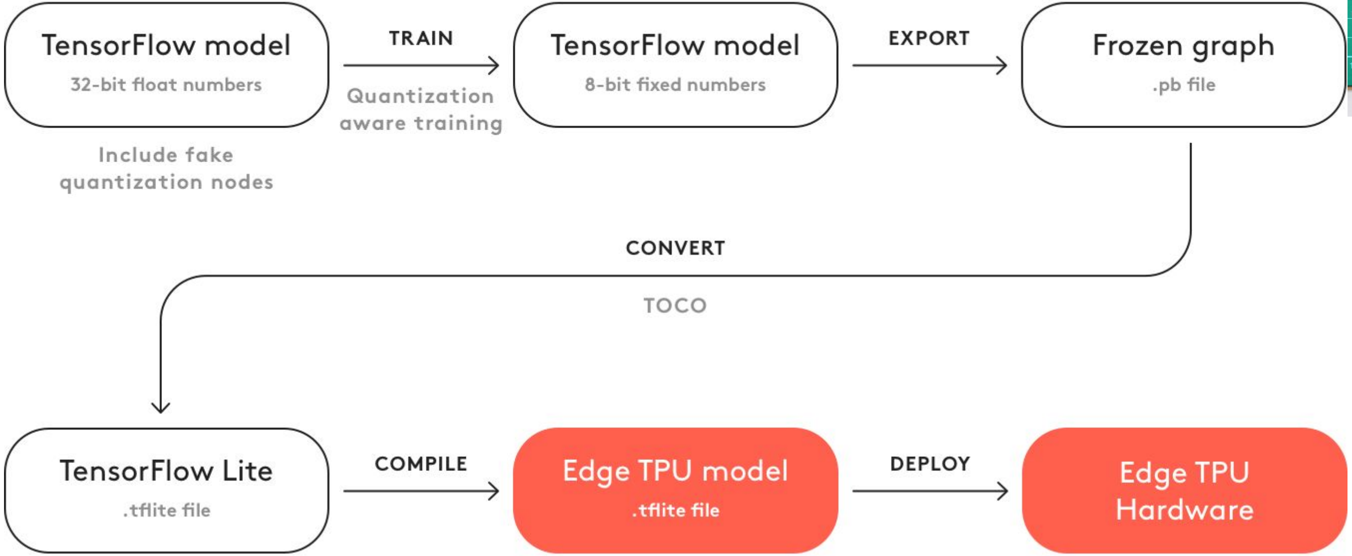
Tiles Everywhere

1. Hardware

Google Edge TPU

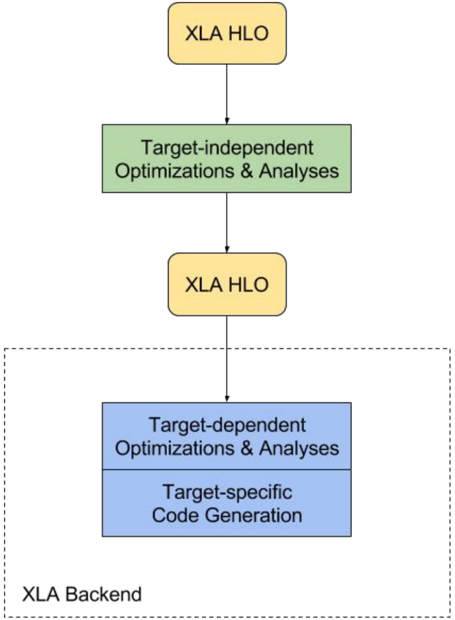


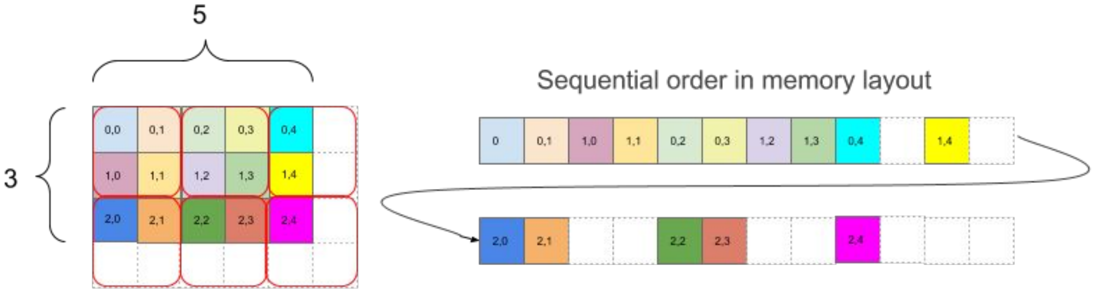
Edge computing zoo



Tiles Everywhere

1. Hardware
2. Data Layout





F32[3,5] with tile size of 2x2

Example: XLA compiler, Tiled data layout

Repeated/Hierarchical Tiling
e.g., BF16 (bfloat16)
on Cloud TPU
(should be 8x128 then 2x1)

Tiles Everywhere

1. Hardware
2. Data Layout
3. Control Flow
4. Data Flow
5. Data Parallelism

Example: Halide for image processing pipelines
https://halide-lang.org

Meta-programming API and domain-specific language (DSL) for loo[redacted]mputing kernels

[redacted]

Tiling in Halide

Tiled schedule:
- strip-mine (a.k.a. split)
- permute (a.k.a. reorder)

[redacted]

Vectorized schedule:
- strip-mine
- vectorize inner loop

[redacted]

Non-divisible bounds/extent:
- strip-mine
- shift left/up
- redundant computation

(also forward substitute/inline operand)

Tiles Everywhere

1. Hardware
2. Data Layout
3. Control Flow
4. Data Flow
5. Data Parallelism

Example: Halide for image processing pipelines
https://halide-lang.org

And also TVM for neural networks
https://tvm.ai

TVM example: scan cell (RNN)

```
m = tvm.var("m")
n = tvm.var("n")
X = tvm.placeholder((m,n), name="X")
s_state = tvm.placeholder((m,n))
s_init = tvm.compute((1,n), lambda _,i: X[0,i])
s_do = tvm.compute((m,n), lambda t,i: s_state[t-1,i] + X[t,i])
s_scan = tvm.scan(s_init, s_do, s_state, inputs=[X])

s = tvm.create_schedule(s_scan.op)

// Schedule to run the scan cell on a CUDA device
block_x = tvm.thread_axis("blockIdx.x")
thread_x = tvm.thread_axis("threadIdx.x")
xo,xi = s[s_init].split(s_init.op.axis[1], factor=num_thread)
s[s_init].bind(xo, block_x)
s[s_init].bind(xi, thread_x)
xo,xi = s[s_do].split(s_do.op.axis[1], factor=num_thread)
s[s_do].bind(xo, block_x)
s[s_do].bind(xi, thread_x)
print(tvm.lower(s, [X, s_scan], simple_mode=True))
```

Tiling and Beyond

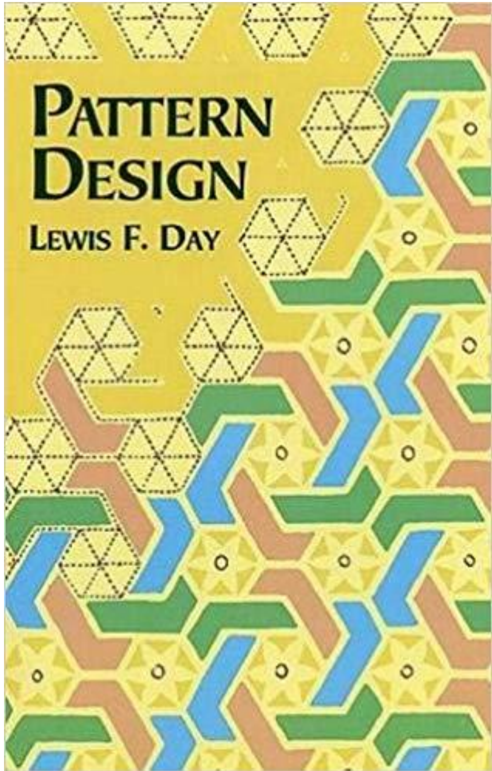
1. But what about **symbolic bounds, sizes, shapes**?
2. **Other transformations**: fusion, fission, pipelining, unrolling… ?
3. **Composition** with other **transformations** and **mapping** decisions?
4. **Consistency** with … ?
5. **Evaluating** cost functions, **enforcing** resource constraints?

→ Impact on compiler construction,
intermediate representations,
program analyses and transformations?

→ Polyhedral Compilation as a Design Pattern

- Tiles tend to be **hyper-rectangles** and occasionally **parallelograms**, **trapezoids**
- **Compose** tile patterns with fusion, fission, pipelining and nested tile patterns

More generally: **Polyhedral Compilation** =
*a **geometric, affine, periodic** view of*
program transformations
*along **time**: sequence, dependences*
*and **space**: parallelism, memory, processors*



Polyhedral Compilation in a Nutshell

with Alex Zinenko

Based on “A Performance Vocabulary for Affine Loop Transformations”
by Martin Kong, Louis-Noel Pouchet; and a dozen of other papers

Architectural Effects to Consider

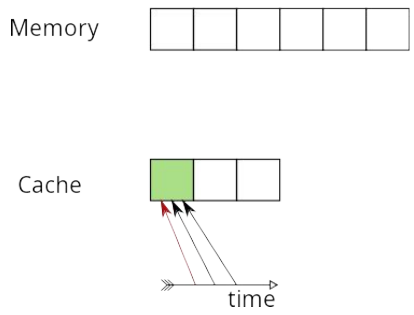
- **Multi-level parallelism**
  - **CPU** — typically 3 levels: system threads or finer grain tasks, vectors, instruction-level parallelism
  - **GPU** — 2 to 8 levels: work groups, work items, warps and vectors, instruction-level parallelism

and related features on other HW accelerators

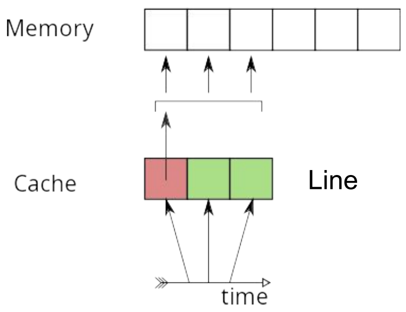
- **Deep memory hierarchies**
  - Positive effects: temporal and spatial locality, coalescing, latency hiding through multithreading
  - Negative effects: cache conflicts, false sharing
  - Many other concerns: capacity constraints, alignment, exposed pipelines

Architectural Effects to Consider

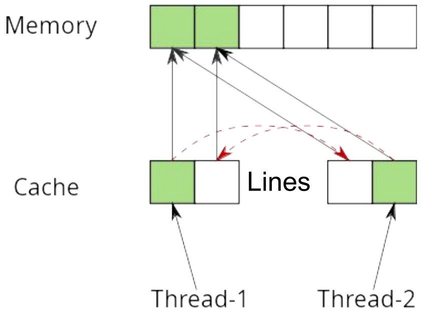
Temporal Locality



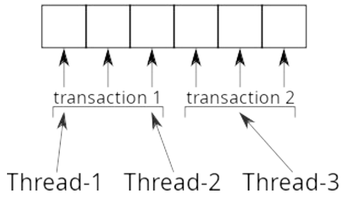
Spatial Locality



False Sharing

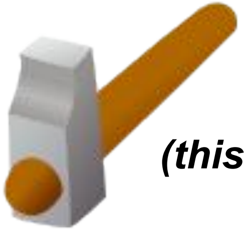


Memory Coalescing



- Need a program representation to reason about individual array elements, individual iterations, relations among these, and with hardware resources
    - Programming languages may provide high level abstractions for nested loops and arrays, tensor algebra, graphics...
    - The need for performance portability leads to domain-specific approaches
      E.g., for ML high-performance kernels alone:
      XLA, TVM, Tensor Comprehensions, Glow, Tiramisu, etc.

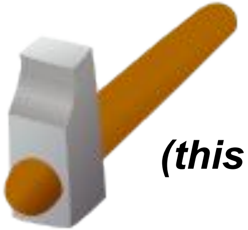
- Yet few compiler intermediate representations reconcile these with
    1. the ability to model hardware features
    2. while capturing complex transformations
    3. supporting both general-purpose domain-specific optimizers



(this is a hammer)

Generic Loop Nest Optimizer

E.g. Intel ICC, Pluto, PPCG, LLVM/Polly



(this is a hammer)

Generic Loop Nest Optimizer

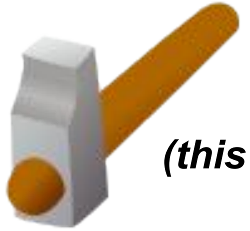
E.g. Intel ICC, Pluto, PPCG, LLVM/Polly

E.g., XLA, Halide, Polymage

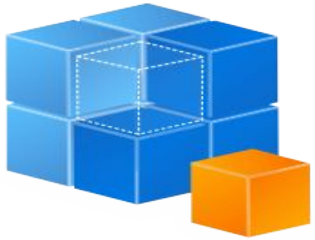
Domain-Specific Optimizer and Code Generator

(these are not nails)

E.g., XLA, Halide, Polymage



(this is a hammer)



Domain-Specific Optimizer and Code Generator

Generic Loop Nest Optimizer

Polyhedral Framework =
semantical and algorithmic design pattern
for multi-purpose representation, analysis, transformation, optimization, code generation

E.g. Intel ICC, Pluto, PPCG, LLVM/Polly

(these are not nails)

Polyhedral Representation Example: 2D convolution

```
for (int i = 0; i < H - KH; ++i)
  for (int j = 0; j < W - KW; ++j) {
    C[i][j] = 0.;
    for (int k = 0; k < KH; ++k)
      for (int l = 0; l < KW; ++l)
        C[i][j] += A[i+k][j+l] * M[k][l];
  }
```

Polyhedral Representation Example: 2D convolution

```
for (int i = 0; i < H - KH; ++i)
  for (int j = 0; j < W - KW; ++j) {
    C[i][j] = 0.;
    for (int k = 0; k < KH; ++k)
      for (int l = 0; l < KW; ++l)
        C[i][j] += A[i+k][j+l] * M[k][l];
  }
```

Statement S1

Polyhedral Representation Example: 2D convolution

```
/* i=0 */
  for (int j = 0; j < W - KW; ++j) {
    C[0][j] = 0.;
    /*...*/
  }

/* i=1 */
  for (int j = 0; j < W - KW; ++j) {
    C[1][j] = 0.;
    /*...*/
  }
/*...*/
```

Polyhedral Representation Example: 2D convolution

```
/* i=0 */
  /* and j=0 */
    C[0][0] = 0.;
  /* and j=1 */
    C[0][1] = 0.;
  /*...*/
/* i=1 */
  /* and j=0 */
    C[1][0] = 0.;
  /*...*/

/*...*/
```

Statement Instance

```
/* i=0 */
  /* and j=0 */
    C[0][0] = 0.;   // S1(0,0)
  /* and j=1 */
    C[0][1] = 0.;   // S1(0,1)
  /*...*/
/* i=1 */
  /* and j=0 */
    C[1][0] = 0.;   // S1(1,0)
  /*...*/

/*...*/
```

Statement *instance*

= specific execution of a statement in a loop

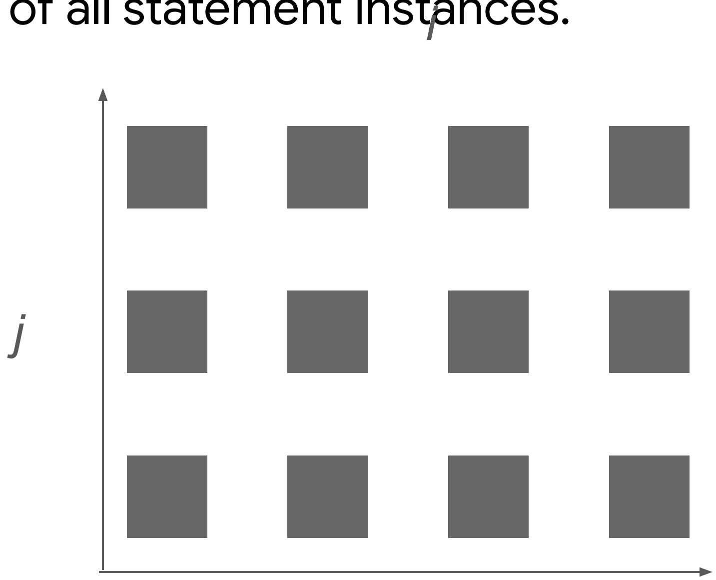
Iteration Domain

Iteration domain: set of all statement instances

```
C[0][0] = 0.;   C[0][1] = 0.;   C[0][2] = 0.;  //...

C[1][0] = 0.;   C[1][1] = 0.;   C[1][2] = 0.;  //…

C[2][0] = 0.;   C[2][1] = 0.;   C[2][2] = 0.;  //…

/*...*/
```

Iteration Domain

Iteration domain: set of all statement instances.



Iteration Domain

Iteration domain: set of all statement instances.

```
for (int i = 0; i < H - KH; ++i)
  for (int j = 0; j < W - KW; ++j) {
    C[i][j] = 0.;
    for (int k = 0; k < KH; ++k)
      for (int l = 0; l < KW; ++l)
S2      C[i][j] += A[i+k][j+l] * M[k][l];
  }
```

need to represent and operate on "finitely presented" integer sets (and relations), and solve optimization problems:

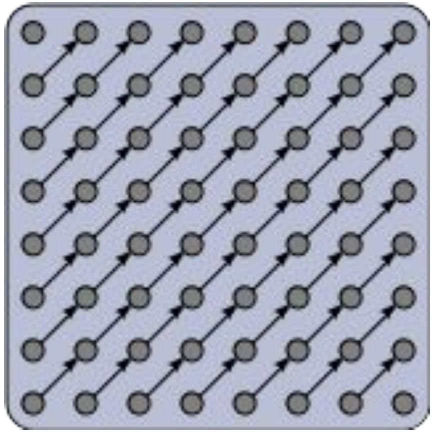
isl http://repo.or.cz/w/isl.git

Sven Verdoolaege

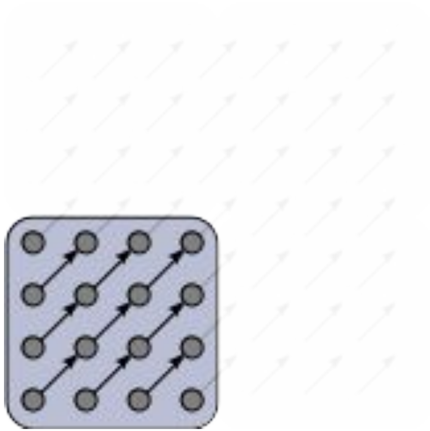
```
D_S2 = (H,W,KH,KW) ->
  {S2(i,j,k,l): 0 <= i < H-KH &&
                0 <= j < W-KW &&
                0 <= k < KH &&
                0 <= l < KW}
```

Program Transformations

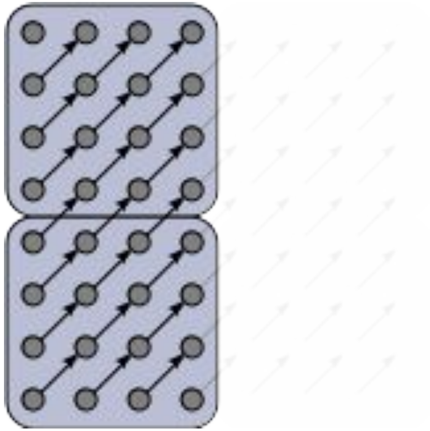
Tiling



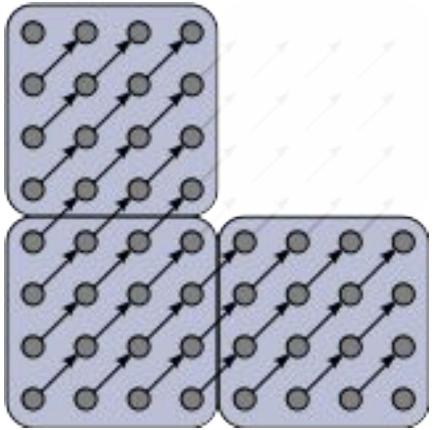
Tiling



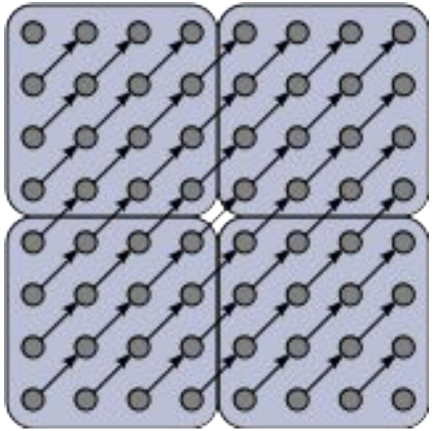
Tiling



Tiling



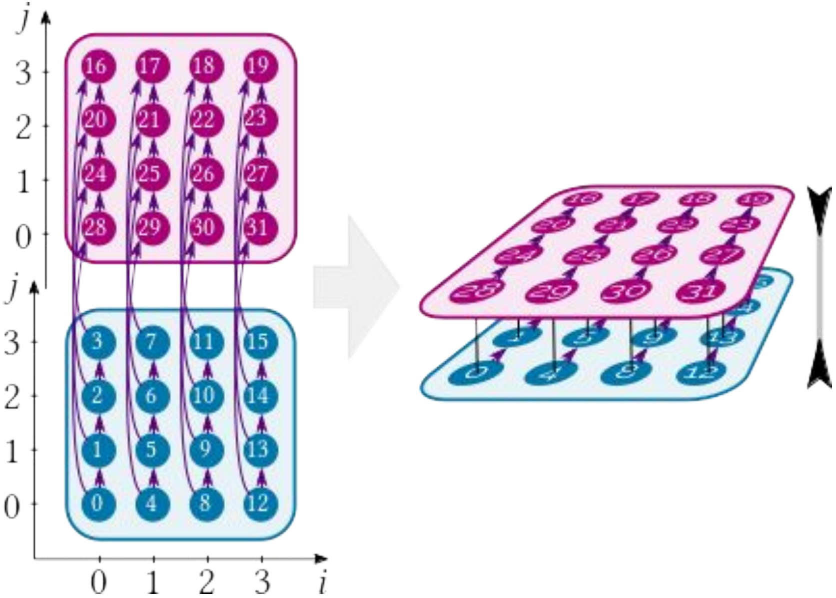
Tiling



May be performed as part of scheduling,
or separately from affine scheduling
if the scheduler ensures *permutability*

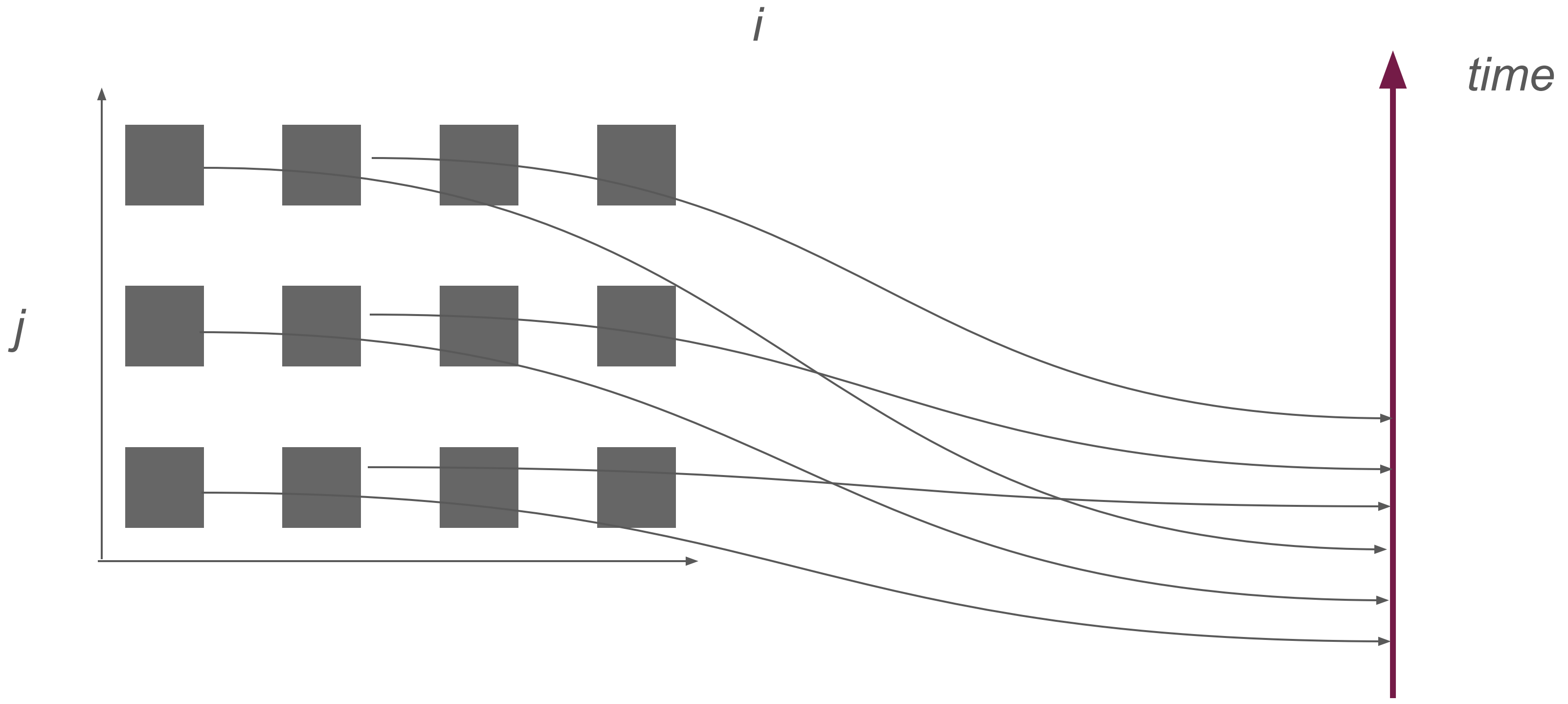
Loop Fusion and Fission

Look for a single-loop schedule respecting all dependences



Multiple fusion heuristics: min, max, “smart”, “wise”...

Affine Schedules



Affine Schedules

Schedules assign logical execution dates to elements of the iteration domain

We are interested* only in *affine* schedules, that is of the shape

$$t = c0 + c1 * i1 + c2 * i2 + c3 * i3 + \ldots + k1 * W + k2 * H + k3 * KH + k4 * KW$$

where c0,c1,c2,...,k1,k2,k3,k4 are constants, and i1,i2,i3... are iteration variables

* We will explain later why

Multi-Dimensional Schedules

To obtain nested loop structure like

```
for (int i = 0; i < H - KH; ++i)
  for (int j = 0; j < W - KW; ++j)
    C[i][j] = 0.;
```

we need to execute the instances C[x][*] after all of C[x-1][*] + induction, but

$$t = \mathbf{(W - KW) * i} + j$$

does not match our affine schedule pattern

Multi-Dimensional Schedules

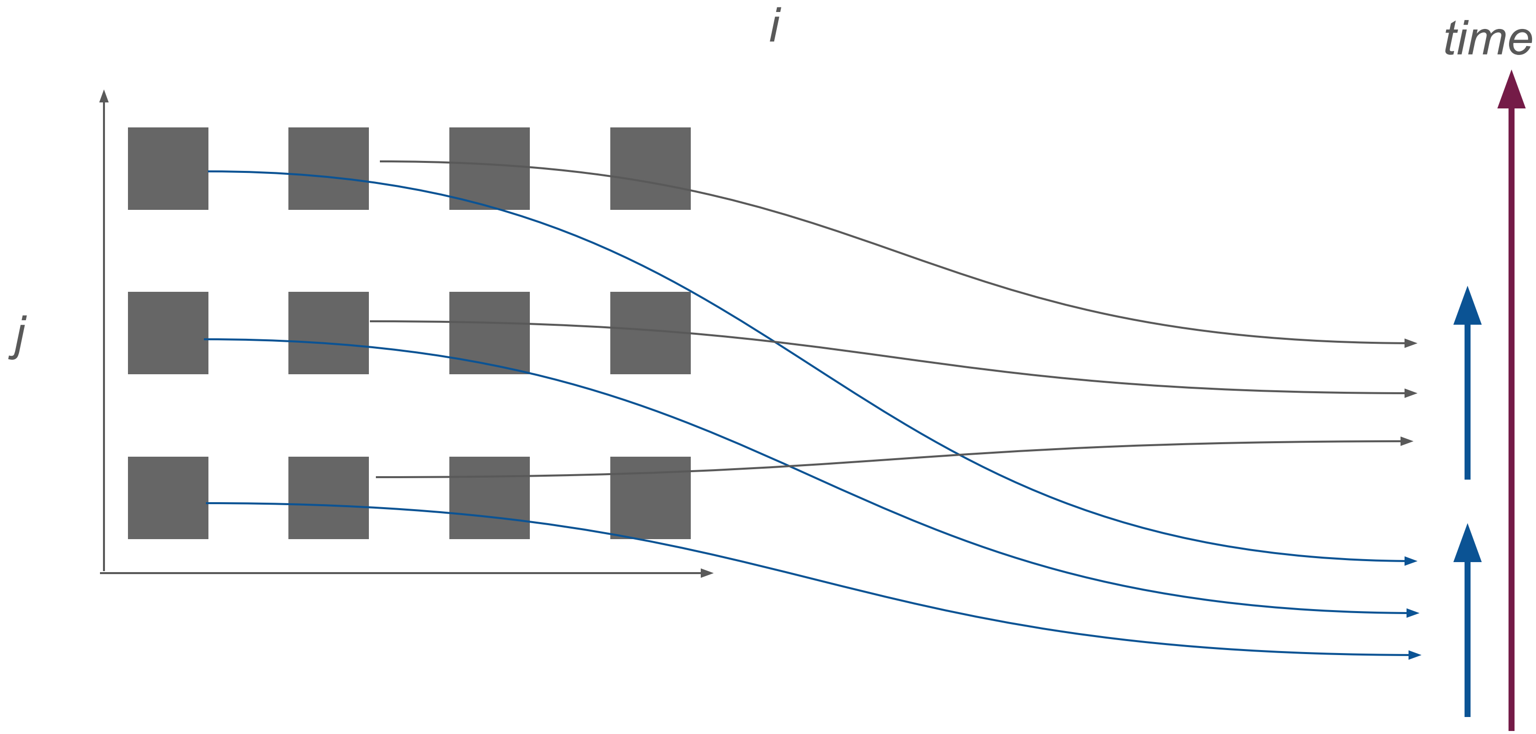
Solution: use multi-dimensional schedules and execute statement instances following the **lexicographical order** of their logical dates:

(0,0) << (0,1) << (0,5672658425682435) << (1,0) << (1,1).

We can then get the original order using a two-dimensional schedule

(i,j) ⟶ (i,j)

Affine Schedules



Multi-Statement Schedules

Similar problem exists: for the same i,j, we want to execute all instances of S2 after all instances of S1

```
for (int i = 0; i < H - KH; ++i)
  for (int j = 0; j < W - KW; ++j) {
    C[i][j] = 0.;                                   // S1
    for (int k = 0; k < KH; ++k)
      for (int l = 0; l < KW; ++l)
        C[i][j] += A[i + k][j + l] * M[k][l];  // S2
  }
```

Note: in this particular case, we can use (i,j,-1,-1) for S1 and (i,j,k,l) for S2 because the lower bound is constant, this trick no longer works for bounds that depend on outer iterators

Multi-Statement Schedules

General solution: introduce auxiliary scalar dimensions to the schedule to separate the statements thanks to the lexicographical order

(i, j, **0**, *, *) for S1;
(i, j, **1**, k, l) for S2.

Any constant can be used in place of *, or these dimensions can be omitted if we extend lexicographical order to vectors of different size with shorter vector preceding longer vectors with the same prefix

Multi-Statement Schedules

```
void 2mm(double alpha, double beta,
         double A[NI][NK], double B[NK][NJ],
         double C[NJ][NL], double D[NI][NL]) {
  double tmp[NI][NJ];
  for (i = 0; i < NI; i++)
    for (j = 0; j < NJ; j++) {
S1:   tmp[i][j] = 0.0;
      for (k = 0; k < NK; ++k)
S2:     tmp[i][j] += alpha * A[i][k] * B[k][j];
    }
  for (i = 0; i < NI; i++)
    for (j = 0; j < NL; j++) {
S3:   D[i][j] *= beta;
      for (k = 0; k < NJ; ++k)
S4:     D[i][j] += tmp[i][k] * C[k][j];
    }
}
```

fuse outer, permute

S1→(0,i,j,1,0)
S2→(1,i,j,0,k)
S3→(0,i,j,0,0)
S4→(1,i,k,1,j)

permute, fuse inner

S1→(**0**,i,0,0,j)
S2→(**0**,i,**k**,1,**j**)
S3→(**1**,i,0,0,j)
S4→(1,i,k,1,j)

> 2x faster
on 4-core CPU

Schedule Functions or Relations

In the simplest case, we use *affine functions* to assign logical execution dates, e.g

$t = f(i, j, k) = i.$

In some cases, we want to relax that and use *relations constrained by affine inequalities* instead. For example, the non-affine function

$t1 = g(i, j, k) = \text{floor}(i / 42)$

can be transformed into an affine relation

$\{(i, j, k) \rightarrow (t1) : 42*t1 <= i <= 42*t1 + 41\}$

Polyhedral/Affine Scheduling

- Iteratively produce affine schedule functions such that:
- dependence distances are *lexicographically* positive
- dependence distances are small ⇒ locality
- dependence distances are zero ⇒ parallelism
- dependences have non-negative distances along consecutive dimensions ⇒ permutability (which enables tiling)

permutable	permutable	
(0,**1**,0,0)	(0,**1**,-2,3)	(0,0,**-1**,42)
valid	also valid	violated

Generally, dependences = RAW + WAR + WAW

Polyhedral/Affine Scheduling

- *Iteratively* produce affine scheduling functions of shape:

$$t_{S,k} = \vec{a} \cdot \vec{i} + \vec{b} \cdot \vec{P} + d$$

Statement S, scheduling step k
a,b,d – coefficients
i – original loop iterators
P – symbolic parameters

minimize $(t_{S,k} - t_{R,k})$

for every dependence R⟶S

Polyhedral/Affine Scheduling

- *Iteratively* produce affine scheduling functions of shape:

$$t_{S,k} = \vec{a} \cdot \vec{i} + \vec{b} \cdot \vec{P} + d$$

Statement S, scheduling step k
a,b,d – coefficients
i – original loop iterators
P – symbolic parameters

minimize $(t_{S,k} - t_{R,k}) \leq \vec{u} \cdot \vec{P} + w$

for every dependence R⟶S

use the affine form of the Farkas lemma to linearize the inequality

$$\text{lexmin } \vec{u}, w, \vec{a}, \vec{b}, d \ \text{ s.t. } \ \vec{u} \succ \vec{0}$$

⟶ Integer Linear Programming (ILP) problem

isl Schedules Trees: Compose w/ Imperative Semantics

Domain $\left[\begin{array}{l} \{\mathsf{S}(i,j) \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathsf{T}(i,j,k) \mid 0 \le i < N \\ \qquad \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array}\right.$

- Sequence
  - Filter $\{\mathsf{S}(i,j)\}$
    - Band $\{\mathsf{S}(i,j) \to (i,j)\}$
  - Filter $\{\mathsf{T}(i,j,k)\}$
    - Band $\{\mathsf{T}(i,j,k) \to (i,j,k)\}$

(a) canonical `sgemm`

Domain $\left[\begin{array}{l} \{\mathsf{S}(i,j) \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathsf{T}(i,j,k) \mid 0 \le i < N \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array}\right.$

Context $\{N = M = 16 \wedge K > 1000\}$

- Band $\left[\begin{array}{ll} \{\mathsf{S}(i,j) & \to (i,j)\} \\ \{\mathsf{T}(i,j,k) & \to (i,j)\} \end{array}\right.$
  - Sequence
    - Filter $\{\mathsf{S}(i,j)\}$
    - Filter $\{\mathsf{T}(i,j,k)\}$
      - Band $\{\mathsf{T}(i,j,k) \to (k)\}$

(b) fused

Domain $\left[\begin{array}{l} \{\mathsf{S}(i,j) \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathsf{T}(i,j,k) \mid 0 \le i < N \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array}\right.$

Context $\{N = M = K = 512 \wedge 0 \le b_x, b_y < 32 \wedge 0 \le t_x, t_y < 16\}$

Filter $\left[\begin{array}{l} \{\mathsf{S}(i,j) \mid i - 32b_x - 31 \le 32 \times 16\lfloor i/32/16 \rfloor \le i - 32b_x \wedge \\ \qquad j - 32b_y - 31 \le 32 \times 16\lfloor j/32/16 \rfloor \le j - 32b_y\} \\ \{\mathsf{T}(i,j,k) \mid i - 32b_x - 31 \le 32 \times 16\lfloor i/32/16 \rfloor \le i - 32b_x \wedge \\ \qquad j - 32b_y - 31 \le 32 \times 16\lfloor j/32/16 \rfloor \le j - 32b_y\} \end{array}\right.$

- Band $\left[\begin{array}{ll} \{\mathsf{S}(i,j) & \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \\ \{\mathsf{T}(i,j,k) & \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \end{array}\right.$
  - Sequence
    - Filter $\{\mathsf{S}(i,j)\}$
      - Filter $\{\mathsf{S}(i,j) \mid (t_x - i) = 0 \bmod 16 \wedge (t_y - j) = 0 \bmod 16\}$
        - Band $\{\mathsf{S}(i,j) \to (i \bmod 32, j \bmod 32)\}$
    - Filter $\{\mathsf{T}(i,j,k)\}$
      - Band $\{\mathsf{T}(i,j,k) \to (32\lfloor k/32 \rfloor)\}$
        - Band $\{\mathsf{T}(i,j,k) \to (k \bmod 32)\}$
          - Filter $\{\mathsf{T}(i,j,k) \mid (t_x - i) = 0 \bmod 16 \wedge (t_y - j) = 0 \bmod 16\}$
            - Band $\{\mathsf{T}(i,j,k) \to (i \bmod 32, j \bmod 32)\}$

(e) fused, tiled, sunk and mapped

Domain $\left[\begin{array}{l} \{\mathsf{S}(i,j) \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathsf{T}(i,j,k) \mid 0 \le i < N \\ \qquad \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array}\right.$

- Band $\left[\begin{array}{ll} \{\mathsf{S}(i,j) & \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \\ \{\mathsf{T}(i,j,k) & \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \end{array}\right.$
  - Band $\left[\begin{array}{ll} \{\mathsf{S}(i,j) & \to (i \bmod 32, j \bmod 32)\} \\ \{\mathsf{T}(i,j,k) & \to (i \bmod 32, j \bmod 32)\} \end{array}\right.$
    - Sequence
      - Filter $\{\mathsf{S}(i,j)\}$
      - Filter $\{\mathsf{T}(i,j,k)\}$
        - Band $\{\mathsf{T}(i,j,k) \to (k)\}$

(c) fused and tiled

Domain $\left[\begin{array}{l} \{\mathsf{S}(i,j) \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathsf{T}(i,j,k) \mid 0 \le i < N \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array}\right.$

- Band $\left[\begin{array}{ll} \{\mathsf{S}(i,j) & \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \\ \{\mathsf{T}(i,j,k) & \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \end{array}\right.$
  - Sequence
    - Filter $\{\mathsf{S}(i,j)\}$
      - Band $\{\mathsf{S}(i,j) \to (i \bmod 32, j \bmod 32)\}$
    - Filter $\{\mathsf{T}(i,j,k)\}$
      - Band $\{\mathsf{T}(i,j,k) \to (32\lfloor k/32 \rfloor)\}$
        - Band $\{\mathsf{T}(i,j,k) \to (k \bmod 32)\}$
          - Band $\{\mathsf{T}(i,j,k0 \to (i \bmod 32, j \bmod 32)\}$

(d) fused, tiled and sunk

Optimization steps for `sgemm`

Code Generation (simplified)

Given iteration domains and schedules, generate the code that traverses all statement instances in the order defined by the schedules:

- each schedule dimension is (potentially) a loop
- compute loop bounds
- eliminate single-iteration loops and other redundant control flow
- rewrite access subscripts

See [Bastoul, 2004], [Vasilache et.al, 2006], [Grosser et.al, 2015]

Why Affine Schedules?

Code generation requires us to invert the schedule, that is express original loop iterators (i,j,...) in terms of new ones (t…). For example, to rewrite A[i][j].

Please solve:

```
t0 = i*i*j*N + j*k;
t1 = k*j - i;
t2 = j*k*(i-1);
```

in integers for i,j,k.

Hilbert's tenth problem: devise a procedure that decides, for any Diophantine equation, if it has an integer solution. [Hilbert, 1900]
Demonstrated that such general procedure does not exist [Matiyasevich, 1970].

Why Affine Schedules?

Systems of *affine* (i.e., linear) Diophantine equations can be solved, e.g. by Gaussian elimination

Systems of affine equations and inequalities can be solved for some optimum using integer linear programming (ILP) or Fourier-Motzkin elimination (FM)

Note: there may be other special cases of Diophantine equations that can be solved, see [Featurier, 2015], [Yuki, 2019]

Correctness Guarantees

Validity of a Loop Transformation

A code transformation is valid if the transformed code produces the same result as the original code.

If we restrict observable results to memory modifications, the transformed code must write the same data to the same addresses in the same order.

Access Functions

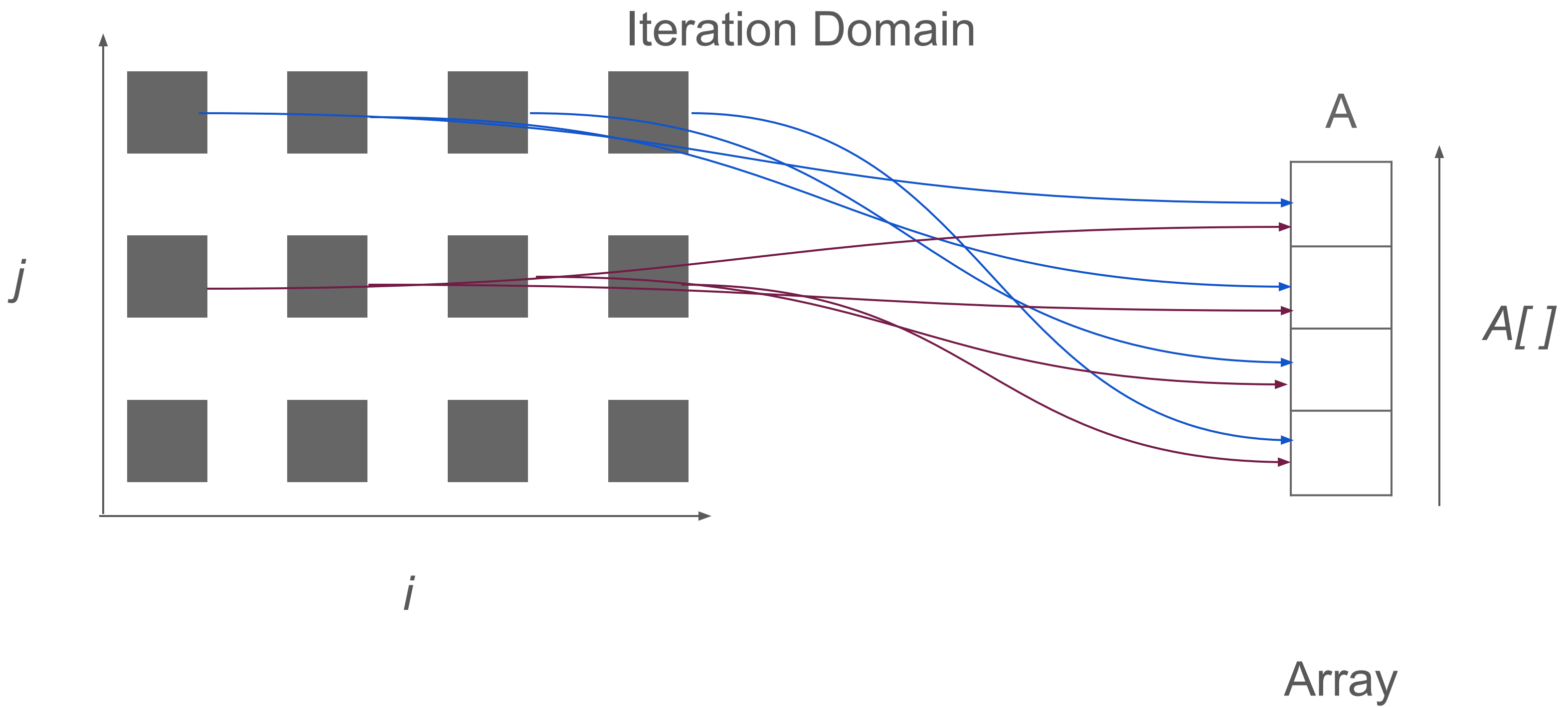
Each statement reads and/or writes to some addresses.

Assuming no aliasing, address = array id + subscripts.

Characterize each access by an affine function:

- arguments: loop induction variables ;
- value: vector of array subscripts.

Access Functions



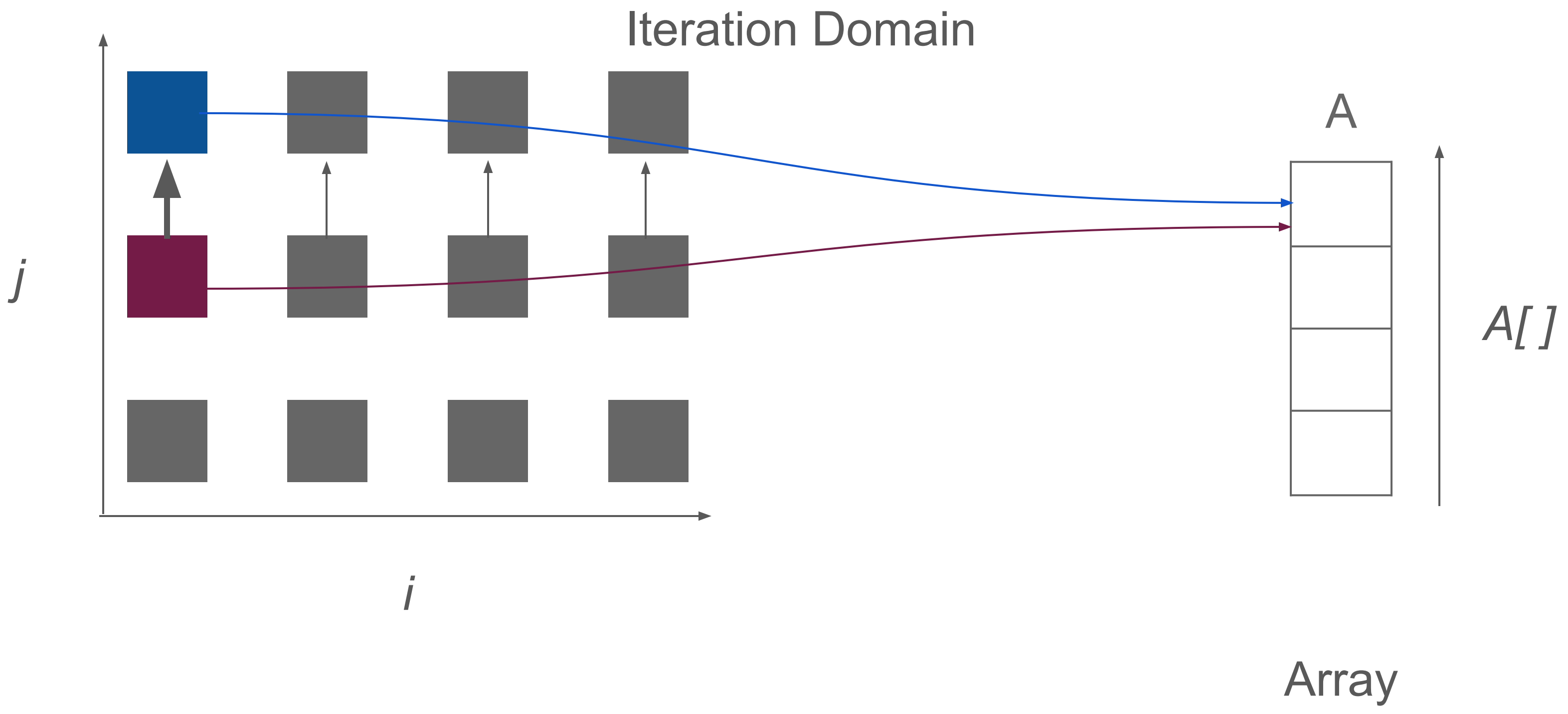
Data Dependences

Statement instances that access the same data in some order are *dependant*.

Define a dependence relation:

- statement instance exists
  (belongs to its iteration domain)
- two statement instances access the same array element
  (equality of access functions)
- one of the instances is executed before another
  (lexicographic order of schedules)

Data Dependences



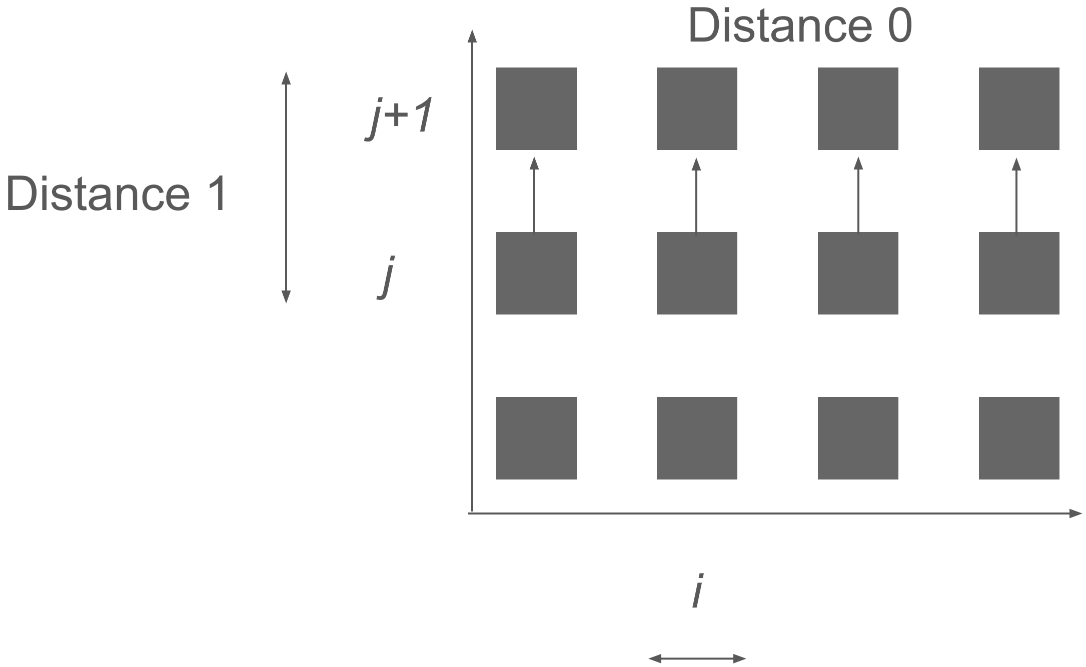
Dependence Distance and Satisfaction

Dependence distance : the difference between logical dates of dependent statement instances.

Dependence is satisfied if its distance is *lexicographically positive*,
i.e. the leading non-zero element is positive => guaranteed order of execution.

Correctness guarantee: all dependences are satisfied.

Dependence Distance and Satisfaction



Affine Scheduling

Basic Principles of Affine Scheduling

Find coefficients of affine schedules for each statement such that:

- dependences are satisfied (correctness);
- schedule is invertible to unambiguously generate code.

Dependences and invertibility define a space of valid schedules. Explore it:

- as an optimization problem given some objective function (ILP);
- exhaustively or sampling + evaluation (e.g., evolutionary methods).

Ensuring Schedule Invertibility

The matrix of coefficients must be non-singular

Iterative approach:

- look for matrix rows iteratively, in separate ILP problems;
- include constraints that guarantee the matrix still has full row rank

One-shot approach:

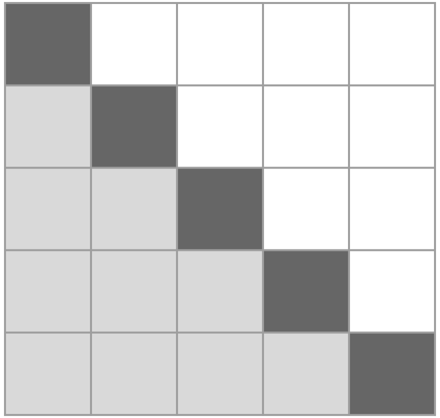
- look for the entire matrix of coefficients in a single ILP problem;
- restrict the matrix to forms that are guaranteed to have full row rank

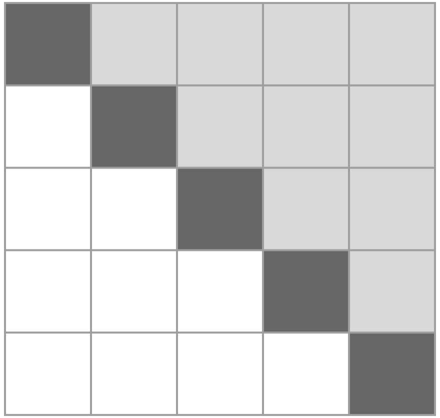
Ensuring Schedule Invertibility

Lower triangular

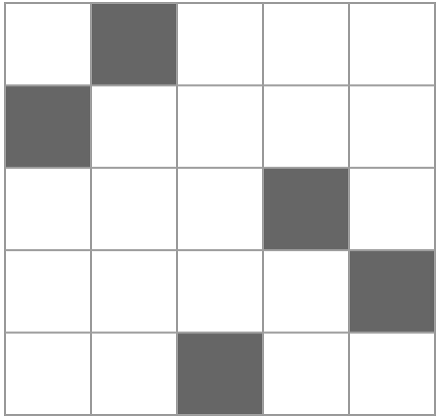
Upper triangular

with no zeros on the main diagonal





One-per-row/column



Objective Functions

Ingredients of the Objective Function

- Access functions
  - controlling the order of access
  - extended forms, such as “vectorized” access functions
- Dependence distances*
  - dependence satisfaction (positive distance)
  - **In**dependence (zero distance = parallelism)
  - proximity (possible reuse)
- Binary decision variables
  - fusion/fission
  - access or dependence properties like 0/1-stride
- Lexicographical order

* cannot use distances directly because non-affine,
  instead, convert into affine constraints on the schedule coefficients using the Farkas lemma

Some Existing Objective Functions

- Farkas-based scheduler [Feautrier 1992]
  - Iterative; maximize the number of satisfied dependences
    ⇒ validity, inner loop parallelism
- Pluto [Bondhugula et.al, 2008]
  - Iterative; minimize the upper bound of the dependence distances
    ⇒ outer loop parallelism + tiling + possible reuse
- Consecutivity [Vasilache et.al, 2012]
  - One-shot: maximize the number of stride-0/1 accesses in the last “dimension”
    ⇒ vectorization
- Spatial locality [Zinenko et.al, 2018]
  - Iterative: Pluto + maximize the “cache-line-sized” dependence distances
    ⇒ cache locality + possible vectorization

Objective Vocabulary: Parallelism

Encode dependence satisfaction as binary variable (related to distance)

```
for (int i = 0; i < NI; ++i)   // distance 0 => parallel
  for (int j = 0; j < NJ; ++j) // distance * => sequential
    A[i] += 42.;
```

OP: Outer Parallelism

- for the outermost linear dimension, minimize the # of satisfied dependences

IP: Inner Parallelism

- for the innermost linear dimension, minimize the # of satisfied dependences

Objective Vocabulary: Vectorization

Define penalty for each iterator appearing in the fastest-varying array subscript, and not appearing in other subscripts (breaks vectorization).

```
for (int i = 0; i < NI; ++i)
  for (int j = 0; j < NJ; ++j)
    for (int k = 0; k < NK; ++k)
      A[i][j] = 0.;          // reuse
      B[i][k] = 0.;          // vectorization
      C[k][j] = 0.;          // nothing => penalty
```

SO: Stride Optimization

- minimize stride penalty

Objective Vocabulary: Parallelism+Reuse

Define a benefit for iterator appearance that favors parallelism, another for reuse, define a penalty for iterator appearance that breaks reuse or vectorization.

```
for (int i = 0; i < NI; ++i)          // parallel
  for (int k = 0; k < NK; ++k)        // reuse
    for (int j = 0; j < NJ; ++j)      // vectorizable
      C[i][j] += A[i][k] * B[k][j];
```

Locality: spatial temporal spatial

OPIR: Outer Parallelism Inner Reuse

- maximize total benefit, then minimize total penalty

Objective Vocabulary: Fusion/Fission

If two statements are in fused loops, the difference of the resp. scalar schedule is 0. Dependences exist between statements that reuse data.

```
for (i = 0; i < N; ++i) {
  A[i] = 42.;
  B[i] = 43.;}
```

```
for (i = 0; i < N; ++i)
  A[i] = 42.;
for (i = 0; i < N; ++i)
  B[i] = A[i];
```

DGF: Dependence-Guided Fusion

- Minimize the penalty for fissioned dependence-related statements

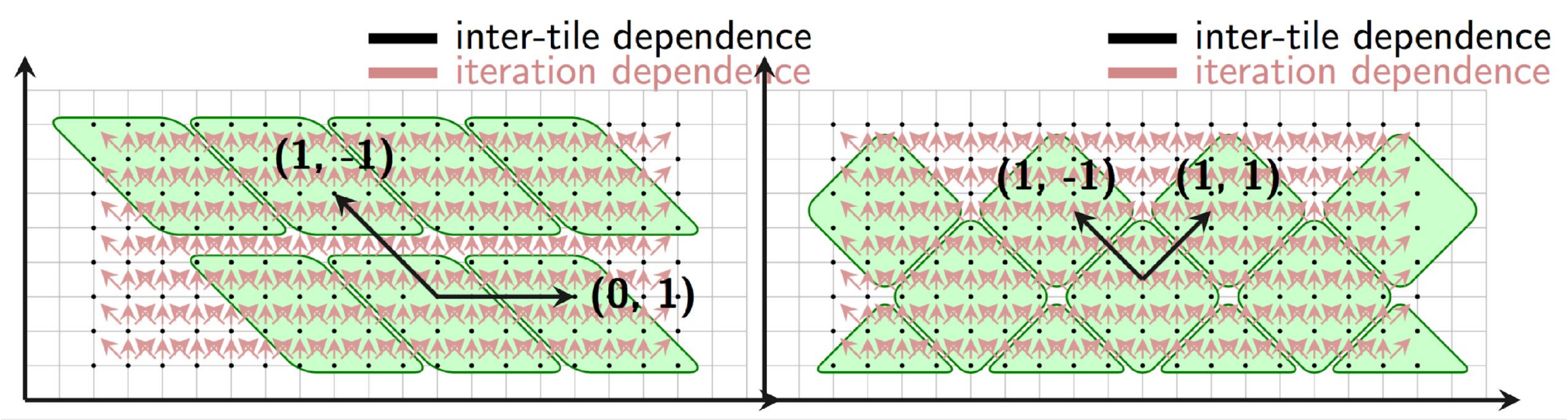
SIS: Separation of Independent Statements

- Minimize the penalty for fused independent statements

Objective Vocabulary: Stencils

Stencils access adjacent locations and are often wavefront-parallelized.

```
for (int t = 0; t < T; ++t)
  for (int i = 0; i < H; ++i)
    for (int j = 0; j < W; ++j)
      A[i][j] = A[i-1][j-1]*M[0][0] + A[i-1][j]*M[0][1] + A[i-1][j+1]*M[0][2]
              + A[i  ][j-1]*M[1][0] + A[i  ][j]*M[1][1] + A[i  ][j+1]*M[1][2]
              + A[i+1][j-1]*M[2][0] + A[i+1][j]*M[2][1] + A[i+1][j+1]*M[2][2];
```



Objective Vocabulary: Stencils

Stencils access adjacent locations and are often wavefront-parallelized.

```
for (int t = 0; t < T; ++t)
  for (int i = 0; i < H; ++i)
    for (int j = 0; j < W; ++j)
      A[i][j] = A[i-1][j-1]*M[0][0] + A[i-1][j]*M[0][1] + A[i-1][j+1]*M[0][2]
              + A[i  ][j-1]*M[1][0] + A[i  ][j]*M[1][1] + A[i  ][j+1]*M[1][2]
              + A[i+1][j-1]*M[2][0] + A[i+1][j]*M[2][1] + A[i+1][j+1]*M[2][2];
```

SPAR: Stencil Parallelization

- Since outer loop is not parallelizable, favor shifting/skewing in this loop only to expose inner parallelism. Penalize skewing coefficients.

SMVS: Stencils Minimization of Vector Skewing

- Further penalize skewing by the loop iterators that appear in fastest-varying array subscripts since this breaks vectorization.

Selecting Objectives

Analyze *iteration domains* and *access functions* to classify programs and select the corresponding sequence of cost functions.

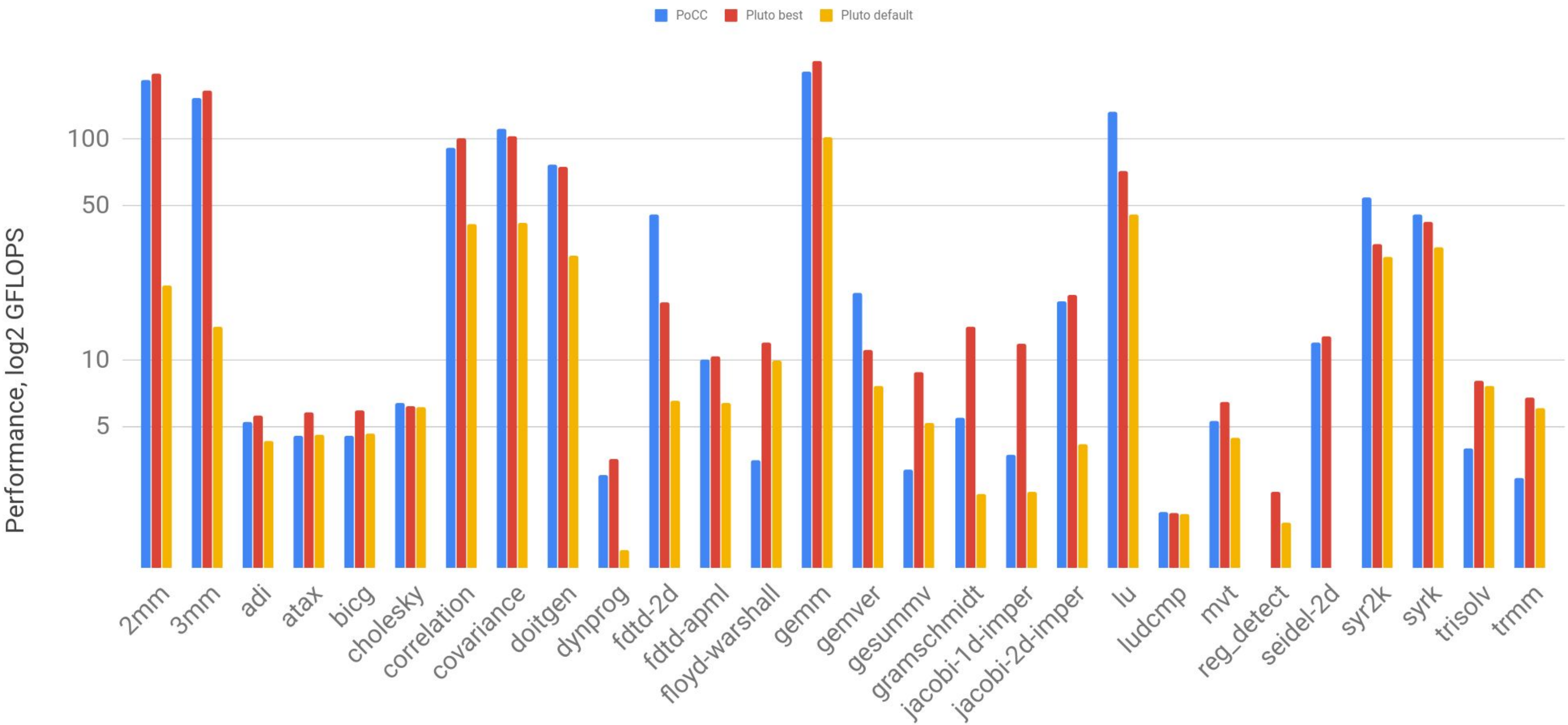
Program class	**Objective functions (lexicographical order)**
Stencils	(stencil-specific dependence analysis) + SMVS, SPAR
Max-2D-loops	SO, IP, OPIR, SIS, DGF, OP
Dense Linear Algebra	<SO, IR, OP>(conditioned on # of dep.), SIS, DGF, OP
Other	SO(conditioned on # of dep.), OP + restrict schedule coefficients

Selected Performance Results

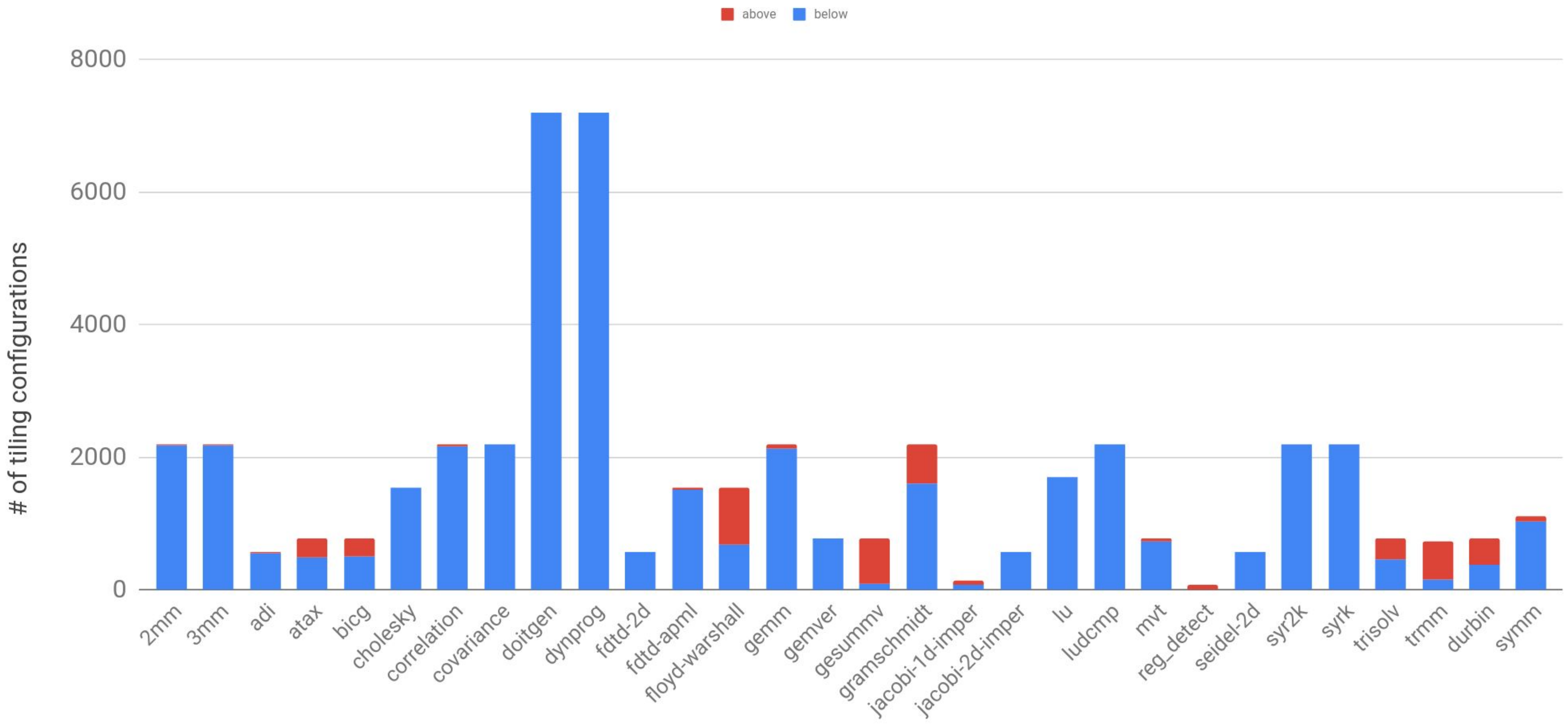
Experimental Methodology

- H/W: 3.3 GHz, 10-core Intel i9-7900X CPU
- Benchmarks: PolyBench/C 3.2 (30 polyhedral kernels)
- Conditions:
    - Baseline: Pluto 0.11.4 + autotuned tiling
    - Tested: PoCC = Pluto with scheduling improvements **without** autotuning

Performance Comparison



Autotuning Search Space



Refinements: Scalability and Customizable Incremental Scheduling

Scalability: Proximity Relation Grouping

- Grouping “sufficiently similar” accesses:
- Access *rank:*
  - prioritize array references with more subcripts.
- Access *multiplicity:*
  - prioritize repeated accesses.
- Iterative *grouping:*
  - Consider accesses contributing to each other’s multiplicity together.

- Each group optimized separately for temporal/spatial locality

Incremental Scheduling Policies

- *Sort* the groups in the ILP to give them more or less priority
- Default heuristic:
  - by rank
  - by multiplicity
  - temporal first

- May include external factors unavailable to a linear optimizer (types, memory characteristics, costs of cache misses, etc.)

Refined Scheduling Algorithm Template

- Consists of two parameterizable ILP problems:
  - carry as little *spatial proximity* relations as possible and produce coincident dimensions for parallelism
    (based on the Pluto algorithm [Bondhugula et.al 2008]);
  - carry multiple *spatial proximity* relations without skewing
    (based on the Feautrier algorithm [Feautrier 1992]).

Instantiation for CPUs

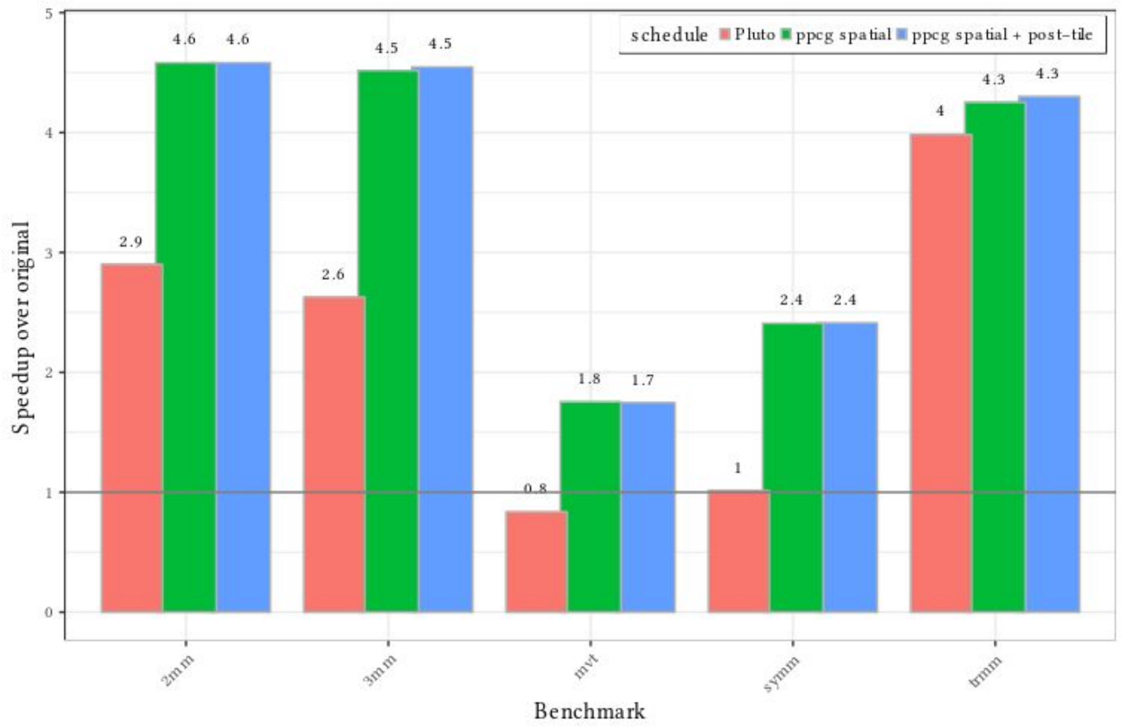
- *One level* of coarse-grain parallelism
  - avoid carrying coincident relations in the outer loops.
- Memory hierarchy favoring *adjacent accesses*
  - carry spatial proximity relations in the inner loops.
- False sharing effect
  - avoid carrying spatial proximity relations in outer || loops.
- High cost of barrier synchronization inside loops
  - if few loops remain to be scheduled, prefer carrying coincident
- Parallelism/Locality conflict
  - requires tiling and different schedule for point loops.

Instantiation for GPUs

- Multiple levels of parallelism
  - avoid carrying coincidence relations, communicate with block/thread mapper.
- Memory coalescing along one thread dimension
  - carry multiple spatial proximity relations while not carrying coincidence relations, ensure mapping to the right thread.
- High overhead of kernel launch
  - more aggressive fusion including (spatial) RAR relations

Polybench on Sequential CPU

Intel Core i7-6600u (Skylake) @ Ubuntu 17.04 + gcc 6.3



Two versions of ppcg: “post-tile” makes syntactic loop interchange after tiling

Example: 2 matrix multiplications

```
void 2mm(double alpha, double beta,
        double A[NI][NK], double B[NK][NJ],
        double C[NJ][NL], double D[NI][NL]) {
  double tmp[NI][NJ];
  for (i = 0; i < NI; i++)
    for (j = 0; j < NJ; j++) {
S1:   tmp[i][j] = 0.0;
     for (k = 0; k < NK; ++k)
S2:     tmp[i][j] += alpha * A[i][k] * B[k][j];
    }
  for (i = 0; i < NI; i++)
    for (j = 0; j < NL; j++) {
S3:   D[i][j] *= beta;
     for (k = 0; k < NJ; ++k)
S4:     D[i][j] += tmp[i][k] * C[k][j];
    }
}
```

Example: 2 matrix multiplications

```
void 2mm(double alpha, double beta,
        double A[NI][NK], double B[NK][NJ],
        double C[NJ][NL], double D[NI][NL]) {
  double tmp[NI][NJ];
  for (i = 0; i < NI; i++)
    for (j = 0; j < NJ; j++) {
S1:   tmp[i][j] = 0.0;
      for (k = 0; k < NK; ++k)
S2:     tmp[i][j] += alpha * A[i][k] * B[k][j];
    }
  for (i = 0; i < NI; i++)
    for (j = 0; j < NL; j++) {
S3:   D[i][j] *= beta;
      for (k = 0; k < NJ; ++k)
S4:     D[i][j] += tmp[i][k] * C[k][j];
    }
}
```

Pluto

- S1→(0,i,j,1,0)
- S2→(1,i,j,0,k)
- S3→(0,i,j,0,0)
- S4→(1,i,k,1,j)

Ppcg-spatial

- S1→(**0**,i,0,0,j)
- S2→(**0**,i,**k**,1,**j**)
- S3→(**1**,i,0,0,j)
- S4→(1,i,k,1,j)

Example: LU decomposition

```
void lu(double A[N][N]) {
 for (i = 0; i < N; i++) {
  for (j = 0; j < i; j++) {
   for (k = 0; k < j; k++)
S1:   A[i][j] -= A[i][k] * A[k][j];
S2:  A[i][j] /= A[j][j];
  }
  for (j = i; j < N; j++)
   for (k = 0; k < i; k++)
S3:   A[i][j] -= A[i][k] * A[k][j];
 }
}
```

Pluto

S1→tile(i,j,k); point(i,k,j)
S2→tile(i,j,j); point(i,j,j)
S3→tile(i,j,k); point(i,k,j)

Ppcg-spatial

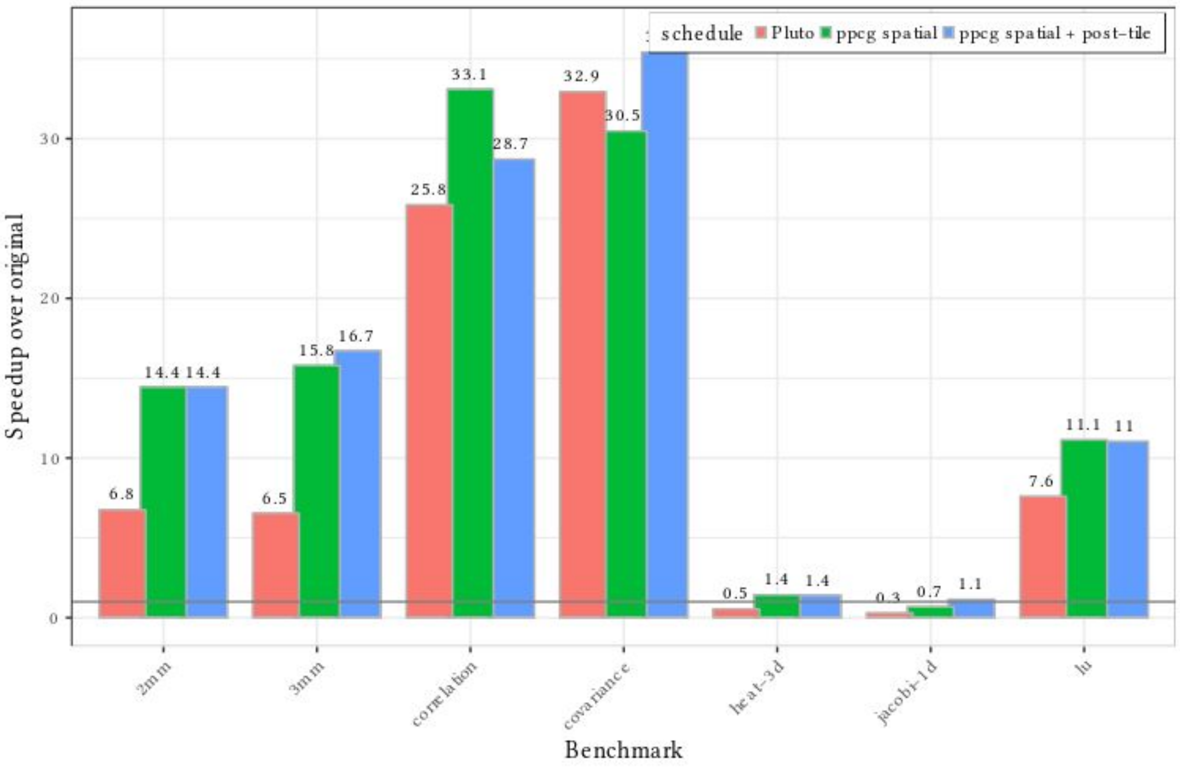
S1→tile(i,**k,j**); point(i,k,j)
S2→tile(i,j,j); point(i,j,j)
S3→tile(i,**k,j**); point(i,k,j)

+wavefront parallelism
(i,k,j) → (i+k,*k*,j)

Reduces false sharing

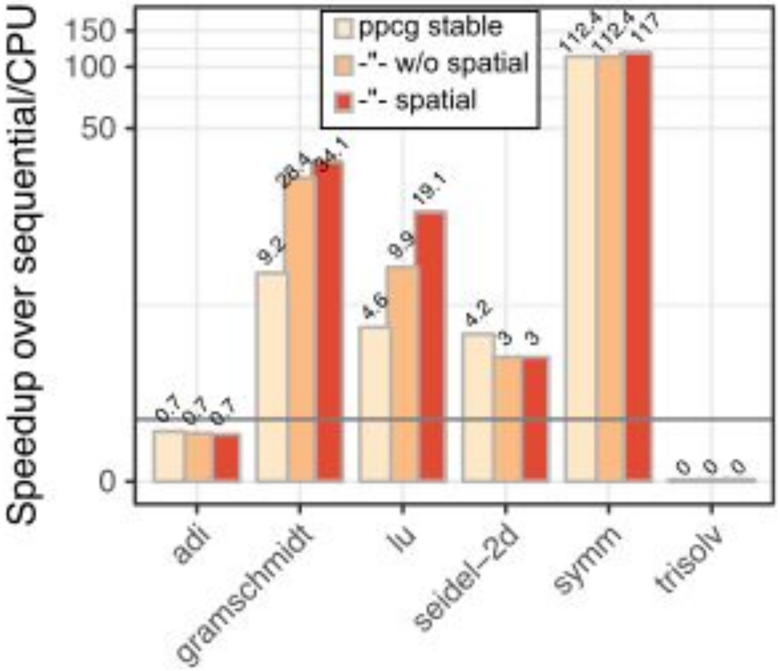
Parallel CPU

4x Intel Xeon E5-2630 (Ivy Bridge) @ CentOS 7.2.1511 + gcc 6.3



Parallel GPU

Nvidia Quadro K4000 (Kepler) @ CentOS 7.2.1511 + CUDA 8.0r1



GPU performance on Polybench is dominated by efficient parallelism extraction

Affine Scheduling Lessons

- “One-size-fits-all” heuristics don’t really fit all.

- Various optimization objectives can be expressed in polyhedral scheduling.

- Different kernels require different optimization strategies.

- The cost of tile size autotuning can be decreased by picking the right cost function, but guessing the best tile sizes remains a challenge.

Some Polyhedral Compilation References

[Bastoul 2004] Bastoul “Code generation in the polyhedral model is easier than you think” @ PACT’04

[Bondhugula et.al 2008] Bondhugula, Hartono, Ramanujam, Sadayappan “A practical automatic polyhedral parallelizer and locality optimizer” @ PLDI 2008

[Feautrier 1992] Feautrier “Some efficient solutions to the affine scheduling problem” (part I and II) @ Intl. Journal of Parallel Programming 21(5) and 21(6).

[Grosser et.al 2014] Grosser, Verdoolaege, Cohen “Polyhedral AST generation is more than scanning polyhedra” @ TOPLAS 2015

[Matiyasevich 1970] Matiyasevich “The Diophantineness of enumerable sets” @ Reports of the URSS Academy of Sciences 191(2) (in Russian).

[Vasilache et.al 2012] Vasilache, Meister, Baskaran, Lethin “Joint scheduling and layout optimization to enable multi-level vectorization” @ IMPACT 2012

[Zinenko et.al 2018] Zinenko, Verdoolaege, Reddy, Shirako, Grosser, Sarkar, Cohen “Modeling the conflicting demands of parallelism and Temporal/Spatial locality in affine scheduling” @ CC 2018

Subjective References in Affine Scheduling

- [Feautrier 1992] first Farkas-based approach to affine scheduling: carry dependences early
- [Bondhugula et al. 2008] Pluto algorithm: outer parallelism and locality in one incremental ILP-based optimization problem
- [Vasilache et al. 2012] introduced access consecutivity constraints and a “one-shot” ILP scheduler
- [Verdoolaege and Isoard 2018] extended Vasilache et al. approach to incremental ILP scheduling
- [Zinenko et al. 2018] unified polyhedral flow for temporal and spatial locality and incremental orchestration of constraints and objectives

Polyhedral Compilation in the Real World

Where From?

Mathematical core: “**isl**”

parametric linear optimization, Presburger arithmetic

- used in **GCC’s Graphite** and **LLVM Polly**
- and many research projects including **Pluto**, **PoCC**, **PPCG**...

Building on **12 years of collaboration**

- ARM, Inria, ETH Zürich (Tobias Grosser)
- AMD, Qualcomm, Xilinx, Facebook
- IISc (Uday Bondhugula)
- IIT Hyderabad (Ramakrishna Upadrasta)
- Ohio State University, Colorado State University, Rice University
- Google Summer of Code

Research and Industry Transfer - Virtual Lab
https://www.pollylabs.org

- Bilateral contracts ARM, Facebook (FAIR), Xilinx, Inria, ETHZ
  in cooperation with Qualcomm (QuIC)

- Focus on LLVM ecosystem: http://llvm.org
  → exploitation & developer community

- Mutualization of core polyhedral compilation infrastructure
- Contributing to domain-specific – **deep learning, image processing, solvers, linear algebra** – and research compilers
- Training and tutorials

Timeline

- **isl** started in 2008, licensed under **LGPLv2.1**
  Used by GCC as its polyhedral library since 5.0
  http://repo.or.cz/w/isl.git
- 2013: Relicensed under **MIT**, through **CARP EU project**
  Used by LLVM through the Polly project

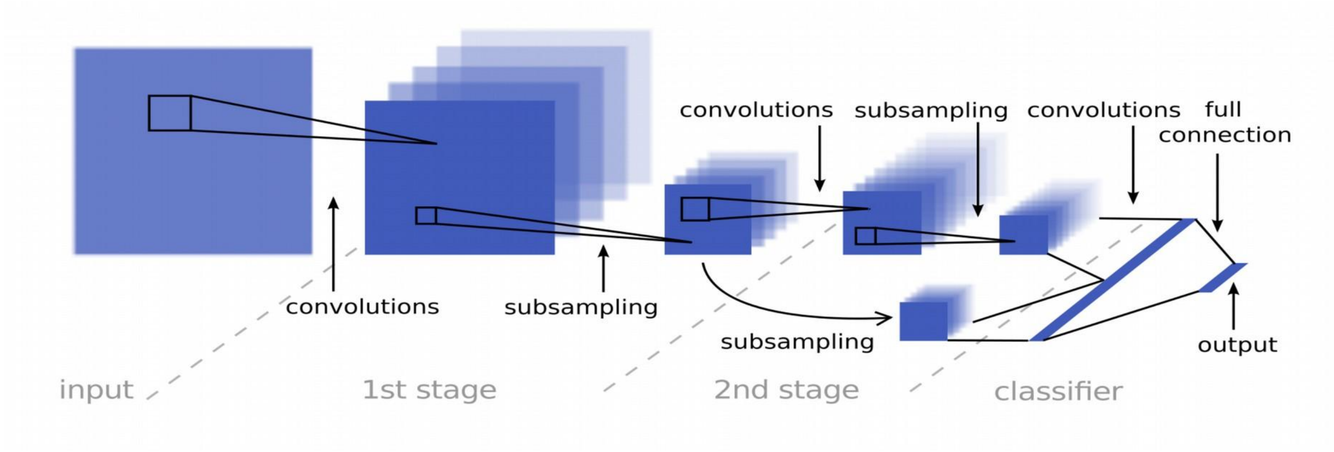
- 2014: Triggered **ARM to release tools** to generate linear algebra kernels
- 2014: **Qualcomm**, then **ARM**, push for ***Polly Labs***
- 2015: Qualcomm Snapdragon LLVM uses Polly, **compiles Android OSP**
- 2016: **Xilinx** starts an isl-based project within **Vivado HLS**
- 2017: **Facebook** works on a **deep learning compiler** using isl
  → **Tensor Comprehensions** project

Tensor Comprehensions

Nicolas Vasilache, Oleksandr Zinenko,
Theodoros Theodoridis, Priya Goyal, Zach DeVito,
William S. Moses, Sven Verdoolaege,
Andrew Adams, Albert Cohen

Facebook AI Research

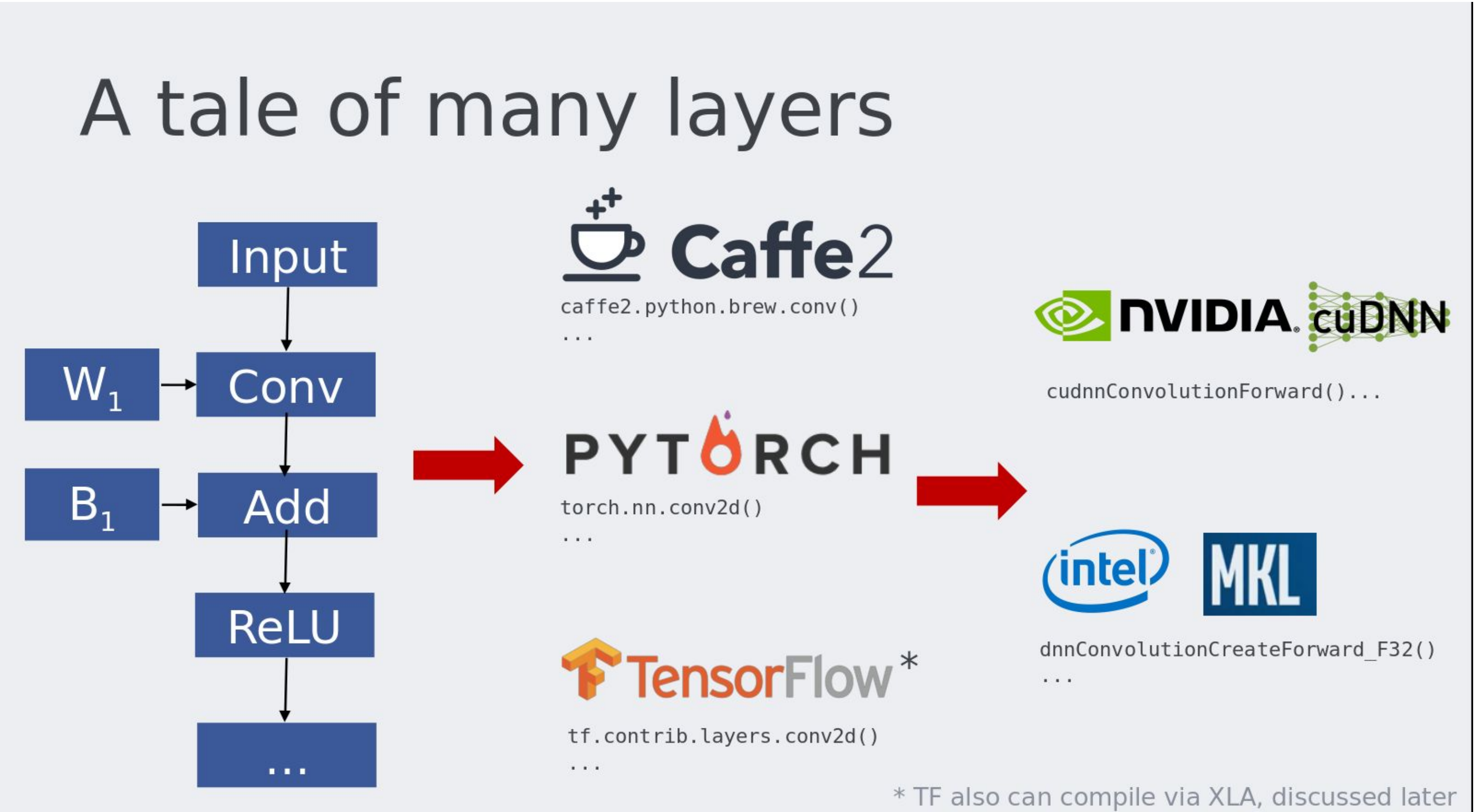
Programming Language & Systems Research… for Deep Learning?



Deep learning has become massively popular over the last 10 years
Machine Learning (ML) frameworks are all over the place

Is this good enough?

Programming Language & Systems Research... for Deep Learning?



Writing “Good” Neural Network Layers

Tedious experience
out of reach from Machine Learning (ML) users and researchers

- **Existing layers** in vendor libraries from Intel, Nvidia, etc.
  - Can reach great performance, 95%+ efficiency on a few, ideal kernels
  - But the practice is often far from machine peak
- **New layer or architecture → performance bottleneck**
  - High-performance library coders are scarce, not all genius, don’t scale
  - *ML differs from scientific/high-performance computing: variety of hardware, data layouts & types, need for symbolic manipulation (automatic differentiation, quantization), and programmer expertise levels*

Our Approach

Don’t write programs, synthesize them

Derive ML-specific techniques from software synthesis and compilers

- Mini-language, **close to mathematics** and **easy to manage by automatic tools**
- Compiler for **algorithmic optimization** and **automatic differentiation**
- Compiler for **“polyhedral” scheduling and mapping**
- Just-in-time **specialization** of the model (hyper)parameters for **efficient kernel implementations**, e.g., for GPU acceleration

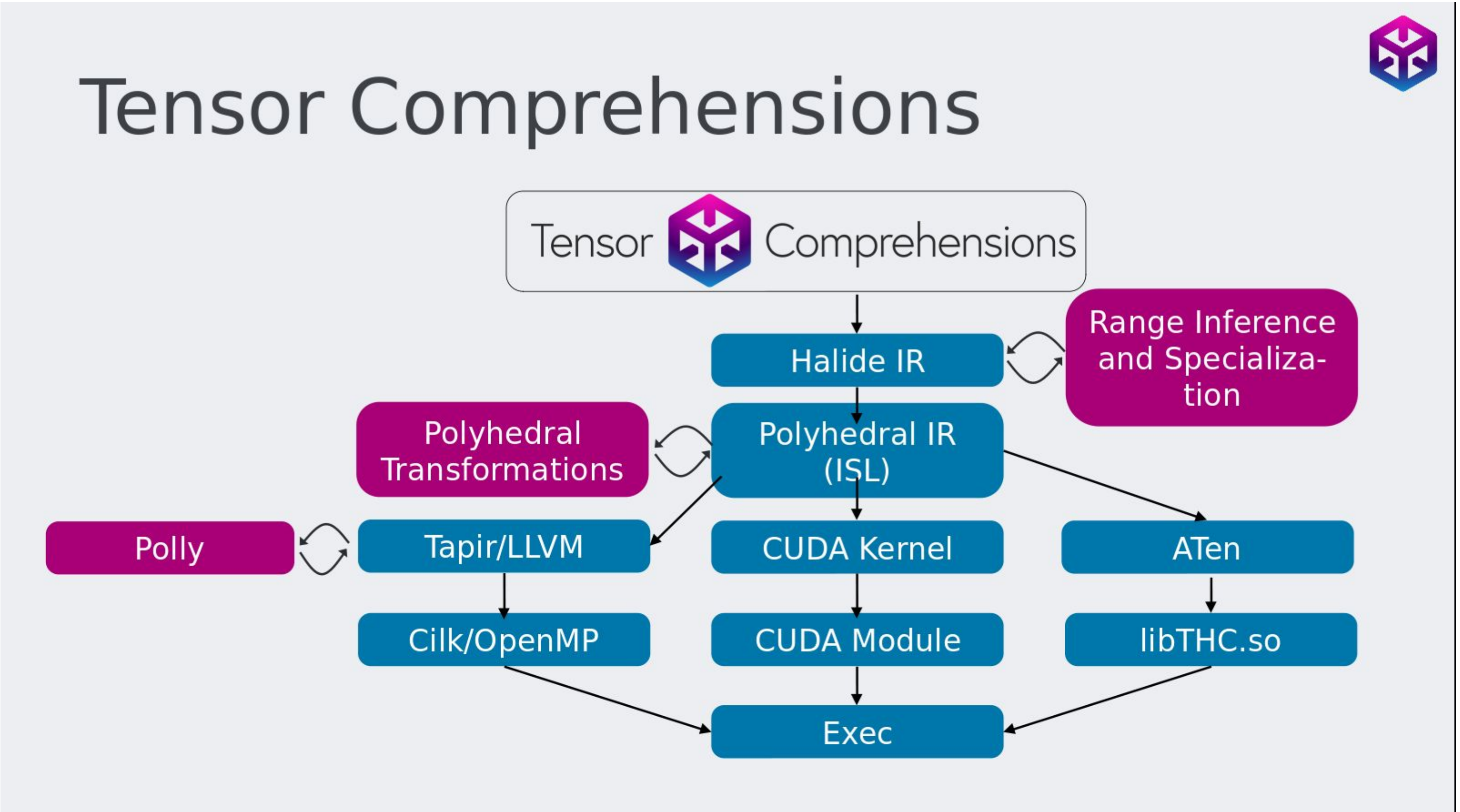
- Works transparently: “A New Op” for machine learning and applications
- Integrated with industry-standard frameworks (Caffe2, PyTorch)

Prior Work

- "Direct generation" such as active library [2] or built-to-order (BTO) [3] provide effective performance, but miss global optimization
- DSLs such as Halide [4] provide usability, and permit scheduling transformations, though manually specify.
- Compilers like XLA [5] or Latte [6] optimize and fuse operators, though performance lacking as the language can't represent complex schedules crucial to GPU/others.

[2] Run-time code generation in C++ as a foundation for domain-specific optimization, 2003
[3] Automating the generation of composed linear algebra kernels, 2009
[4] Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines, 2013
[5] Xla:domain-specific compiler for linear algebra to optimizes tensorflow computations, 2017
[6] Latte: A language, compiler, and runtime for elegant and efficient deep neural networks, 2016

Our Approach



Our Approach

TC language

Concise, emits 1000’s of optimized LOC

```
def mv(float(M,K) A, float(K) x) -> (C)
{
  C(i) +=! A(i,k) * x(k)
}

def conv3(float(N,C,H,W) I, float(O,C,H,W) W1, float(D,O,H,W) W2, float(E,D,H,W) W3)
    -> (O1, O2, O3) {
  O1(n, o, h, w) +=!  I(n, c, h + kh, w + kw) * W1(o, c, kh, kw)
  O1(n, o, h, w) = fmax(O1(n, o, h, w), 0) // relu
  O2(n, d, h, w) +=! O1(n, d, h + kh, w + kw) * W2(d, o, kh, kw)
  O2(n, d, h, w) = fmax(O2(n, d, h, w), 0)
  O3(n, e, h, w) +=! O2(n, c, h + kh, w + kw) * W3(e, d, kh, kw)
  O3(n, e, h, w) = fmax(O3(n, e, h, w), 0)
}
```

Iteration bounds inferred

Variables only on one side are reduced

Synthesize From Model...

Tight mathematical model, emits 1000s optimized lines of code

- Group Convolution

```
def g_conv_hwcgn(float I(H, W, C, G, N), float W(O, G, C, KH, KW)) -> (O) {
  O(h, w, c, g, n) += I(h + kh, w + kw, c, g, n) * W(o, g, c, kh, kw)
}
```

- Kronecker Recurrent Units (KRU)

```
def 3KRU_v0(float(D0,N0) W0, float(D1,N1) W1, float(D2,N2) W2,
            float(M,N0,N1,N2) X) → (Y) {
  Y(m,d0,d1,d2) +=! X(m,n0_r,n1_r,n2_r) * W2(d2,n2_r)
                  * (W1(d1,n1_r) * W0(d0,n0_r))
}

def 3KRU_v1(float(D0,N0) W0, float(D1,N1) W1, float(D2,N2) W2,
            float(M,N0,N1,N2) X) → (Y,XW2) {
  XW2(m,n0,n1,d2) +=!   X(m,n0,n1,r2) * W2(d2,r2)
    Y(m,d0,d1,d2) +=! XW2(m,r0,r1,d2) * W1(d1,r1) * W0(d0,r0)
}

def 3KRU(float(D0,N0) W0, float(D1,N1) W1, float(D2,N2) W2,
         float(M,N0,N1,N2) X) → (Y,XW1,XW2) {
  XW2(m,n0,n1,d2) +=!   X(m,n0,n1,r2) * W2(d2,r2)
  XW1(m,n0,d1,d2) +=! XW2(m,n0,r1,d2) * W1(d1,r1)
    Y(m,d0,d1,d2) +=! XW1(m,r0,d1,d2) * W0(d0,r0)
}
```

→ algorithmic exploration of storage/recompute tradeoffs

Synthesize From Model...

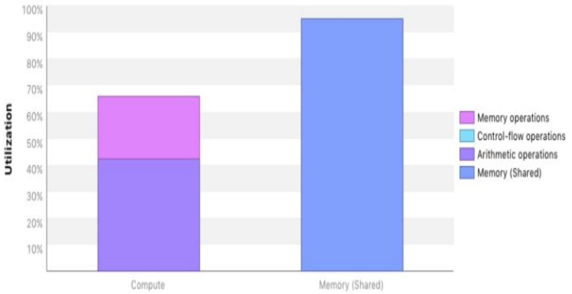
- Production MLP from Facebook
- And more complex examples, including Google WaveNet

```
def 2LUT(float(E1,D) LUT1, int(B,L1) I1,
         float(E2,D) LUT2, int(B,L2) I2) -> (O1,O2) {
  O1(i,j) +=! LUT1(I1(i,k),j)
  O2(i,j) +=! LUT2(I2(i,k),j)
}
def MLP1(float(B,M) I, float(O,N) W1, float(O) B1) -> (O1) {
  O1(b,n)  = B1(n)
  O1(b,n) += I(b,m) * W1(n,m)
  O1(b,n)  = fmaxf(O1(b,n), 0)
}
def MLP3(float(B,M) I, float(O,N) W2, float(O) B2,
         float(P,O) W3, float(P) B3, float(Q,P) W4,
         float(Q) B4) -> (O1,O2,O3,O4) {
  O2(b,o)  = B2(o)
  O2(b,o) += O1(b,n) * W2(o,n)
  O2(b,o)  = fmaxf(O2(b,o), 0)
  O3(b,p)  = B3(p)
  O3(b,p) += O2(b,o) * W3(p,o)
  O3(b,p)  = fmaxf(O3(b,p), 0)
  O4(b,q)  = B4(q)
  O4(b,q) += O3(b,p) * W4(q,p)
  O4(b,q)  = fmaxf(O4(b,q), 0)
}

def prodModel(float(E1,D) LUT1, int(B,L1) I1,
              float(E2,D) LUT2, int(B,L2) I2,
              float(B,WX) I3, float(WY,WX) W,
              float(N,M) W1, float(N) B1,
              float(O,N) W2, float(O) B2,
              float(P,O) W3, float(P) B3,
              float(Q,P) W4, float(Q) B4)
    -> (C1,C2,C3,I,O1,O2,O3,O4) {
  (C1,C2)     = 2LUT(LUT1,I1,LUT2,I2)
  C3(b,wy)   += I3(b,wxx) * W(wy,wxx)
  I(b,m)      = concat(C1, C2, C3) # not implemented yet
  O1          = MLP1(I, W1, B1)
  (O2,O3,O4) = MLP3(O1,W2,B2,W3,B3,W4,B4)
  # O4 goes out to binary classifier, omitted here
}
```

… Rather Than Write Accelerator Code

```
__global__ void lut_100_64_50_vE1_50_vE2(int B, int D, int L1, int E1, int L2, int E2, float* __restrict__ O1, f
{
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    __shared__ int shared_Idx1[1][51];
    __shared__ int shared_Idx2[1][51];
    __shared__ float shared_LUT1[50][64];
    // __shared__ float shared_LUT2[50][64];
    float private_O1[1][1];
    float private_O2[1][1];

    {
      shared_Idx1[0][t1] = Idx1[b0 * 50 + t1];
      if (t1 <= 17)
        shared_Idx1[0][t1 + 32] = Idx1[b0 * 50 + (t1 + 32)];
      shared_Idx2[0][t1] = Idx2[b0 * 50 + t1];
      if (t1 <= 17)
        shared_Idx2[0][t1 + 32] = Idx2[b0 * 50 + (t1 + 32)];
      __syncthreads();
      for (int c4 = 0; c4 <= 49; c4 += 1) {
        shared_LUT1[c4][32 * b1 + t1] = LUT1[(0 * vE1 + shared_Idx1[0][c4]) * 64 + (32 * b1 + t1)];
        // shared_LUT2[c4][32 * b1 + t1] = LUT2[(0 * vE2 + shared_Idx2[0][c4]) * 64 + (32 * b1 + t1)];
      }
      __syncthreads();
      private_O1[0][0] = 0.00000f;
      private_O2[0][0] = 0.00000f;
      for (int c4 = 0; c4 <= 49; c4 += 1) {
        private_O1[0][0] = (private_O1[0][0] + shared_LUT1[c4][(32 * b1 + t1)]);
        // private_O1[0][0] = (private_O1[0][0] + LUT1[(0 * vE1 + shared_Idx1[0][c4]) * 64 + (32 * b1 + t1)]);
        // private_O2[0][0] = (private_O2[0][0] + shared_LUT2[c4][(32 * b1 + t1)]);
        private_O2[0][0] = (private_O2[0][0] + LUT2[(0 * vE2 + shared_Idx2[0][c4]) * 64 + (32 * b1 + t1)]);
      }
      __syncthreads();
      O2[b0 * 64 + (32 * b1 + t1)] = private_O2[0][0];
      O1[b0 * 64 + (32 * b1 + t1)] = private_O1[0][0];
      __syncthreads();
    }
}
```

Shared Memory			
Shared Loads	1147666432	2,967.871 GB/s	
Shared Stores	19464192	50.335 GB/s	
Shared Total	1167130624	3,018.206 GB/s	Idle Low Medium High Max

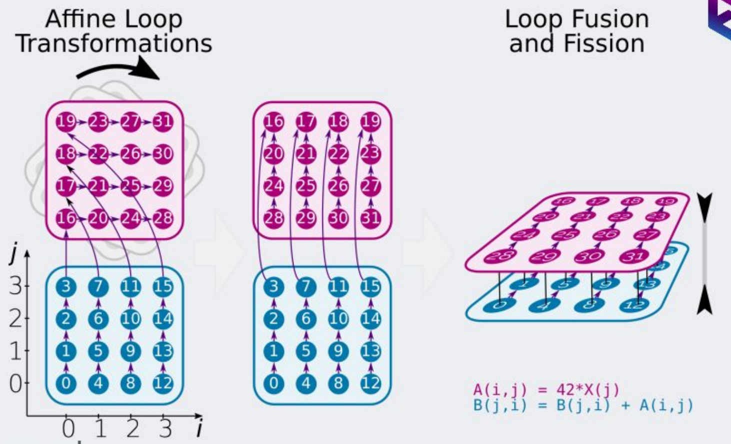



```
{       MOV R9, R3;
LDG.E.CI R22, [R22];      }
{       SHL R3, R0, 0x2;
LDG.E.CI R17, [R8];      }
ISETP.EQ.AND P0, PT, R12, RZ, PT;
STS [R3+0x324], R18;
STS [R3+0x42c], R20;
DEPBAR.LE SB5, 0x7;
STS [R3+0x534], R26;
DEPBAR.LE SB5, 0x4;
STS [R3+0x198], R24;
STS [R3+0x2a0], R25;
STS [R3+0x21c], R6;
DEPBAR.LE SB5, 0x1;
STS [R3+0x63c], R29;
STS [R3+0x3a8], R16;
STS [R3+0x4b0], R22;
STS [R3+0x5b8], R17;
LDS.U.64 R4, [0x28];
LDS.U.128 R8, [0x30];
```

Polyhedral Compilation to the Rescue

Polyhedral + TC

- High Level Polyhedral IR (ISL) =>
  Easy Transformations



- Schedule heuristic folds into a single kernel
- Schedule tiled to facilitate the mapping and reuse of memory hierarchy of GPU/CPU
- GPU mapping borrows from PPCG, with extensions for more complex/imperfectly nested structures
- Memory promotion into shared cache

Polyhedral Compilation to the Rescue

ISL scheduling

```
def sgemm(float a, float b float(N,M) A, float(M,K) B) -> (C) {
  C(i,j)  = b                          // S(i,j)
  C(i,j) += a * A(i,k) * B(k,j)   // T(i,j,k)
}
```

Domain $\left[\begin{array}{ll}\{\mathtt{S}(i,j) & \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathtt{T}(i,j,k) & \mid 0 \le i < N \\ & \quad \wedge 0 \le j < K \wedge 0 \le k < M\}\end{array}\right.$

Sequence
- Filter$\{\mathtt{S}(i,j)\}$
  - Band$\{\mathtt{S}(i,j) \to (i,j)\}$
- Filter$\{\mathtt{T}(i,j,k)\}$
  - Band$\{\mathtt{T}(i,j,k) \to (i,j,k)\}$

Tile

Domain $\left[\begin{array}{ll}\{\mathtt{S}(i,j) & \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathtt{T}(i,j,k) & \mid 0 \le i < N \wedge 0 \le j < K \wedge 0 \le k < M\}\end{array}\right.$

Band $\left[\begin{array}{ll}\{\mathtt{S}(i,j) & \to (i,j)\} \\ \{\mathtt{T}(i,j,k) & \to (i,j)\}\end{array}\right.$

Sequence
- Filter$\{\mathtt{S}(i,j)\}$
- Filter$\{\mathtt{T}(i,j,k)\}$
  - Band$\{\mathtt{T}(i,j,k) \to (k)\}$

Sequence node: order-dependent collection of nodes

Band node: (partial) execution

Filter node: partition iteration space

Heard That Before?

- 30 years of parallelizing and optimizing compiler research
- … wrapped into a robust, automated tool, with domain specialization
- … with modern C++ interface and tight ML framework integration

- Embed the most complex compositions of loop nest and tensor optimizations

$$\begin{array}{l}
\text{Domain}\left[\begin{array}{ll}\{\mathsf{S}(i,j) & \mid 0\le i<N\wedge 0\le j<K\}\\ \{\mathsf{T}(i,j,k) & \mid 0\le i<N\\ & \wedge 0\le j<K\wedge 0\le k<M\}\end{array}\right.\\
\quad\text{Sequence}\\
\qquad\text{Filter}\{\mathsf{S}(i,j)\}\\
\qquad\quad\text{Band}\{\mathsf{S}(i,j)\to(i,j)\}\\
\qquad\text{Filter}\{\mathsf{T}(i,j,k)\}\\
\qquad\quad\text{Band}\{\mathsf{T}(i,j,k)\to(i,j,k)\}
\end{array}$$

(a) canonical `sgemm`

$$\begin{array}{l}
\text{Domain}\left[\begin{array}{ll}\{\mathsf{S}(i,j) & \mid 0\le i<N\wedge 0\le j<K\}\\ \{\mathsf{T}(i,j,k) & \mid 0\le i<N\wedge 0\le j<K\wedge 0\le k<M\}\end{array}\right.\\
\quad\text{Context}\{N=M=16\wedge K>1000\}\\
\quad\text{Band}\left[\begin{array}{ll}\{\mathsf{S}(i,j) & \to(i,j)\}\\ \{\mathsf{T}(i,j,k) & \to(i,j)\}\end{array}\right.\\
\qquad\text{Sequence}\\
\qquad\quad\text{Filter}\{\mathsf{S}(i,j)\}\\
\qquad\quad\text{Filter}\{\mathsf{T}(i,j,k)\}\\
\qquad\qquad\text{Band}\{\mathsf{T}(i,j,k)\to(k)\}
\end{array}$$

(b) fused

$$\begin{array}{l}
\text{Domain}\left[\begin{array}{ll}\{\mathsf{S}(i,j) & \mid 0\le i<N\wedge 0\le j<K\}\\ \{\mathsf{T}(i,j,k) & \mid 0\le i<N\\ & \wedge 0\le j<K\wedge 0\le k<M\}\end{array}\right.\\
\quad\text{Band}\left[\begin{array}{ll}\{\mathsf{S}(i,j) & \to(32\lfloor i/32\rfloor,32\lfloor j/32\rfloor)\}\\ \{\mathsf{T}(i,j,k) & \to(32\lfloor i/32\rfloor,32\lfloor j/32\rfloor)\}\end{array}\right.\\
\qquad\text{Band}\left[\begin{array}{ll}\{\mathsf{S}(i,j) & \to(i \bmod 32, j \bmod 32)\}\\ \{\mathsf{T}(i,j,k) & \to(i \bmod 32, j \bmod 32)\}\end{array}\right.\\
\qquad\quad\text{Sequence}\\
\qquad\qquad\text{Filter}\{\mathsf{S}(i,j)\}\\
\qquad\qquad\text{Filter}\{\mathsf{T}(i,j,k)\}\\
\qquad\qquad\quad\text{Band}\{\mathsf{T}(i,j,k)\to(k)\}
\end{array}$$

(c) fused and tiled

$$\begin{array}{l}
\text{Domain}\left[\begin{array}{ll}\{\mathsf{S}(i,j) & \mid 0\le i<N\wedge 0\le j<K\}\\ \{\mathsf{T}(i,j,k) & \mid 0\le i<N\wedge 0\le j<K\wedge 0\le k<M\}\end{array}\right.\\
\quad\text{Band}\left[\begin{array}{ll}\{\mathsf{S}(i,j) & \to(32\lfloor i/32\rfloor,32\lfloor j/32\rfloor)\}\\ \{\mathsf{T}(i,j,k) & \to(32\lfloor i/32\rfloor,32\lfloor j/32\rfloor)\}\end{array}\right.\\
\qquad\text{Sequence}\\
\qquad\quad\text{Filter}\{\mathsf{S}(i,j)\}\\
\qquad\qquad\text{Band}\{\mathsf{S}(i,j)\to(i \bmod 32, j \bmod 32)\}\\
\qquad\quad\text{Filter}\{\mathsf{T}(i,j,k)\}\\
\qquad\qquad\text{Band}\{\mathsf{T}(i,j,k)\to(32\lfloor k/32\rfloor)\}\\
\qquad\qquad\quad\text{Band}\{\mathsf{T}(i,j,k)\to(k \bmod 32)\}\\
\qquad\qquad\qquad\text{Band}\{\mathsf{T}(i,j,k0\to(i \bmod 32, j \bmod 32)\}
\end{array}$$

(d) fused, tiled and sunk

$$\begin{array}{l}
\text{Domain}\left[\begin{array}{ll}\{\mathsf{S}(i,j) & \mid 0\le i<N\wedge 0\le j<K\}\\ \{\mathsf{T}(i,j,k) & \mid 0\le i<N\wedge 0\le j<K\wedge 0\le k<M\}\end{array}\right.\\
\text{Context}\{N=M=K=512\wedge 0\le b_x,b_y<32\wedge 0\le t_x,t_y<16\}\\
\quad\text{Filter}\left[\begin{array}{ll}\{\mathsf{S}(i,j) & \mid i-32b_x-31\le 32\times16\lfloor i/32/16\rfloor\le i-32b_x\wedge\\ & \quad j-32b_y-31\le 32\times16\lfloor j/32/16\rfloor\le j-32b_y\}\\ \{\mathsf{T}(i,j,k) & \mid i-32b_x-31\le 32\times16\lfloor i/32/16\rfloor\le i-32b_x\wedge\\ & \quad j-32b_y-31\le 32\times16\lfloor j/32/16\rfloor\le j-32b_y\}\end{array}\right.\\
\quad\text{Band}\left[\begin{array}{ll}\{\mathsf{S}(i,j) & \to(32\lfloor i/32\rfloor,32\lfloor j/32\rfloor)\}\\ \{\mathsf{T}(i,j,k) & \to(32\lfloor i/32\rfloor,32\lfloor j/32\rfloor)\}\end{array}\right.\\
\qquad\text{Sequence}\\
\qquad\quad\text{Filter}\{\mathsf{S}(i,j)\}\\
\qquad\qquad\text{Filter}\ \{\mathsf{S}(i,j) \mid (t_x-i)=0 \bmod 16\wedge (t_y-j)=0 \bmod 16\}\\
\qquad\qquad\quad\text{Band}\{\mathsf{S}(i,j)\to(i \bmod 32, j \bmod 32)\}\\
\qquad\quad\text{Filter}\{\mathsf{T}(i,j,k)\}\\
\qquad\qquad\text{Band}\{\mathsf{T}(i,j,k)\to(32\lfloor k/32\rfloor)\}\\
\qquad\qquad\quad\text{Band}\{\mathsf{T}(i,j,k)\to(k \bmod 32)\}\\
\qquad\qquad\qquad\text{Filter}\ \{\mathsf{T}(i,j,k) \mid (t_x-i)=0 \bmod 16\wedge (t_y-j)=0 \bmod 16\}\\
\qquad\qquad\qquad\quad\text{Band}\{\mathsf{T}(i,j,k)\to(i \bmod 32, j \bmod 32)\}
\end{array}$$

(e) fused, tiled, sunk and mapped

Figure 3: Optimization steps for `sgemm` from Figure 1

Algorithmic Contributions

Extending ISL scheduling

- Extended ISL's scheduler to allow additional constraints
  - Affine constraint added to the LP
  - Supply clustering decision for graph component combining
- Clustering allows for conventional minimum and maximum fusion targets AND maximum fusion that preserves at least three nested parallel loops (i.e. for mapping to CUDA blocks / threads)

Memory Promotion

```
O[l+Idx[i][j]][k] => shared_O[l][i][j][k]
```

- Cache indirectly accessed arrays
- Only when O and Idx are only read (not written)

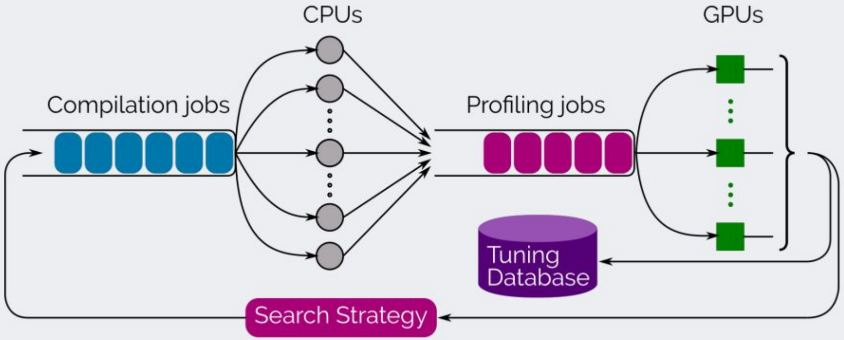
- Promote direct accesses if tile of fixed size, elements reused, and >= 1 access without memory coalescing
- Promote indirect accesses in same way (ignore coalescing)

- Heuristics for register promotion as well

Performance?

Autotuning

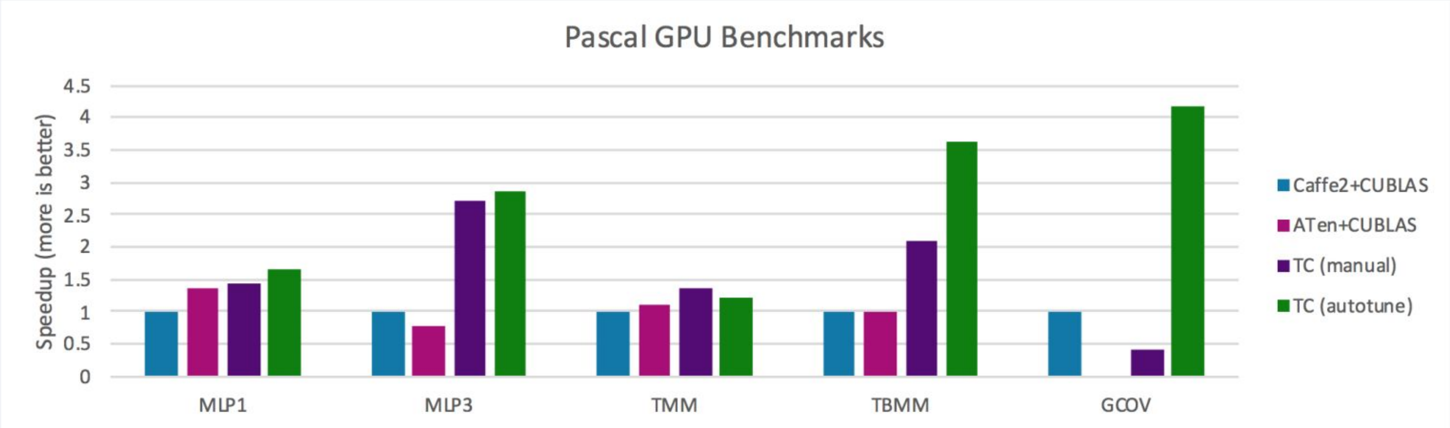
- Even with heuristics, there's a large space of options
- Derive schedule (and other parameters) by searching via genetic algorithm with fixed search-time.



Performance?

End-to-end benchmarks

Baseline CUDA 8.0, CUBLAS 8.0, CUDNN 6.0, CUB recent



8 Pascal nodes with 2 socket, 14 core Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, with 8 Tesla P100-SXM2 GPUs and 16GB of memory each.
Median runtime out of a batch of 1000

TC in a Nutshell

TC overview

"Natural ML math running faster than libraries"

- Available stand-alone and in Caffe2/PyTorch bindings [public in a few days]
- Open source: https://github.com/facebookresearch/tensorcomprehensions
- White Paper: https://arxiv.org/abs/1802.04730

Polyhedral Compilation in the Real World
… once again with broader ambitions

MLIR: Multi-Level Intermediate Representation for the End of Moore’s Law

From EuroLLVM 2019 keynote and tutorial

Presenting the work of many, many, people

TensorFlow

Huge machine learning community

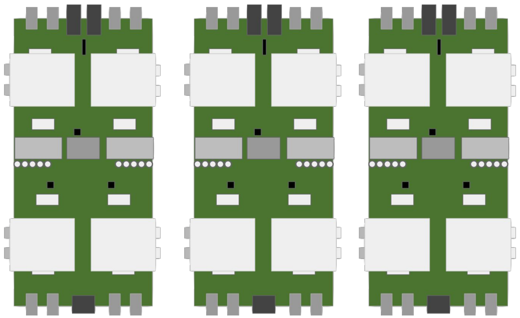
Programming APIs for many languages

Abstraction layer for accelerators:

- Heterogenous, distributed, mobile, custom ASICs…
- Urgency is driven by the “end of Moore’s law”

Open Source:

https://tensorflow.org
https://tensorflow.org/mlir



Why a new compiler infrastructure?

The LLVM Ecosystem: Clang Compiler

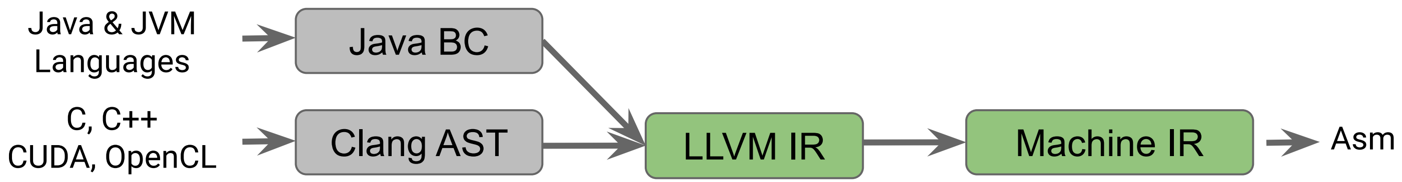
Green boxes are SSA IRs:

- Different levels of abstraction - operations and types are different
- Abstraction-specific optimization at both levels

Progressive lowering:

- Simpler lowering, reuse across other front/back ends

Azul Falcon JVM

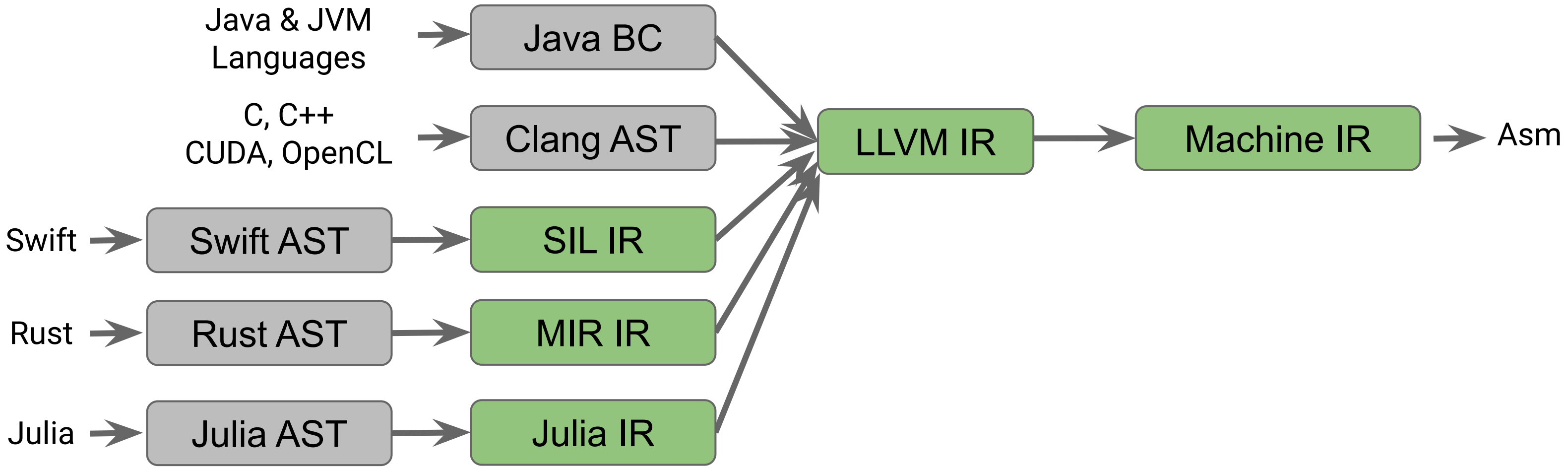


Uses LLVM IR for high level domain specific optimization:

- Encodes information in lots of ad-hoc ways: IR Metadata, well known functions, intrinsics, …
- Reuses LLVM infrastructure: pass manager, passes like inliner, etc.

"[Falcon: An Optimizing Java JIT]" - LLVM Developer Meeting Oct'2017

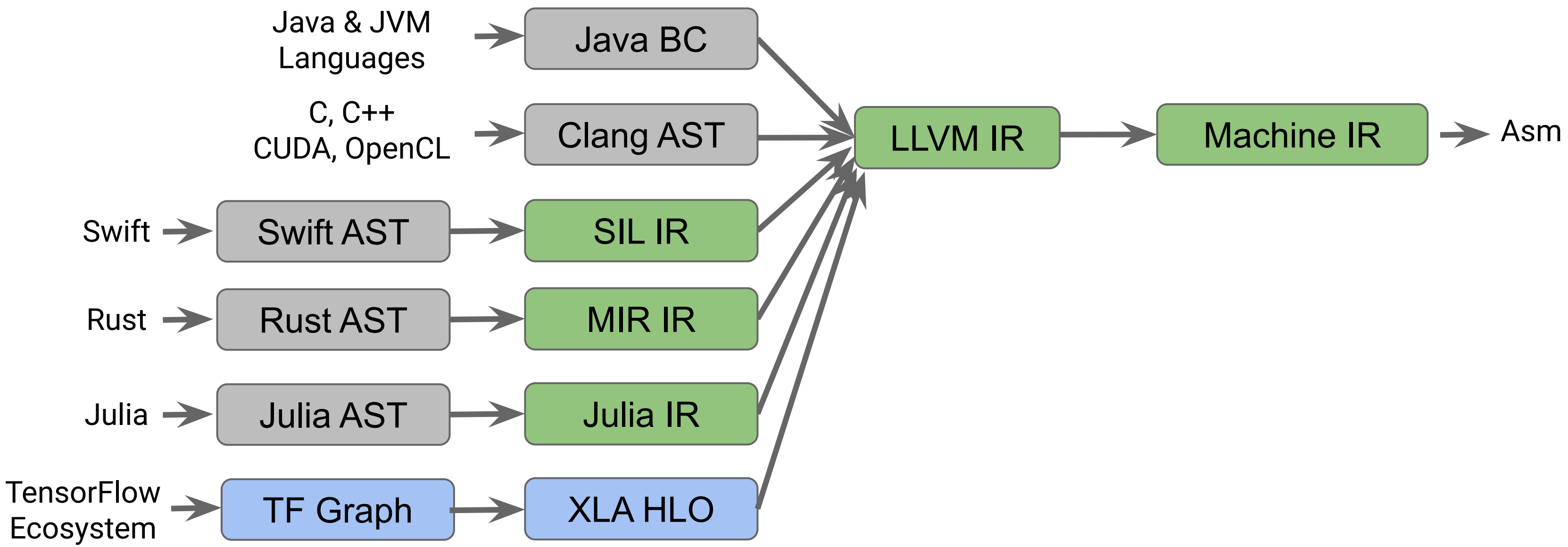
Swift, Rust and Julia have a high level IR - Not C and C++



- Domain specific optimizations: generic specialization, devirt, ref count optzns, library-specific optzns, etc
- Dataflow driven type checking - e.g. borrow checker in Rust
- Domain specific optimizations, progressive lowering

"Introducing MIR": Rust Language Blog, "Julia SSA-form IR": Julia docs

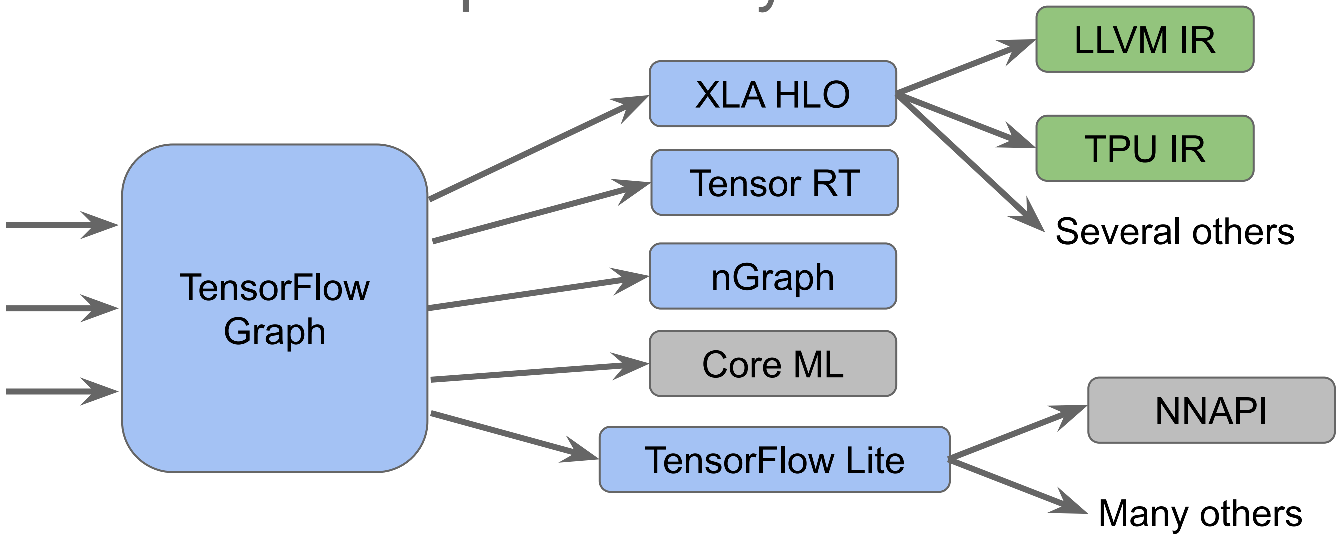
TensorFlow XLA Compiler



- Domain specific optimizations, progressive lowering, ad-hoc emitters

"XLA Overview": https://tensorflow.org/xla/overview (video overview)

The TensorFlow compiler ecosystem



Many “Graph IRs”, each with challenges:

- Similar-but-different (some, proprietary) technologies: not going away anytime soon
- Duplication of infrastructure at all levels

→ **Need SSA-based design to generalize and improve “ML graphs”**

Domain Specific IRs

Great!

- High-level domain-specific optimizations
- Progressive lowering encourages reuse between levels

Not great!

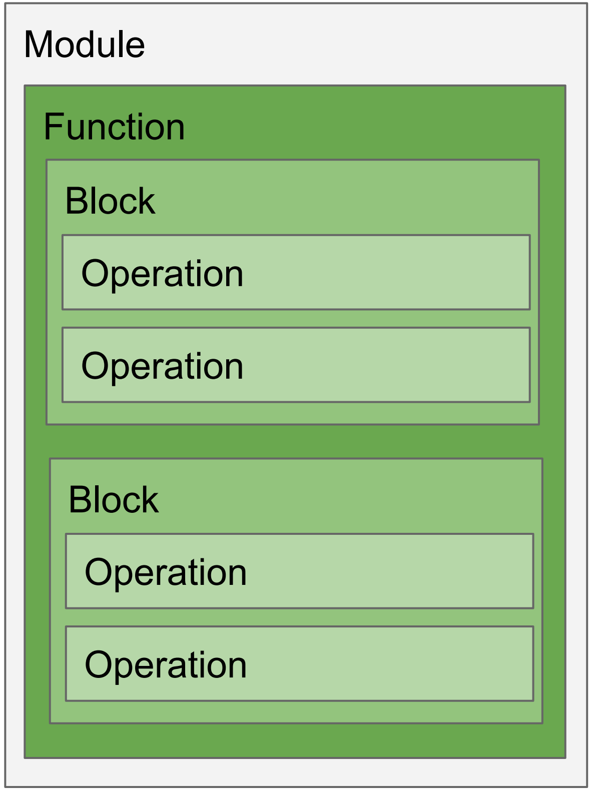
- Huge expense to build this infrastructure
- Reimplementation of all the same stuff
  - pass managers, location/error tracking, testing tools
  - inlining, use-def chains, constant folding, partial redundancy elimination, ...
- Innovations in one community don’t benefit the others

MLIR Primer

Many similarities to LLVM

- SSA, CFG, typed, three address
- Module/Function/Block/Operation structure
- Round trippable textual form
- Syntactically similar:

```
func @testFunction(%arg0: i32) {
  %x = call @thingToCall(%arg0) : (i32) -> i32
  br ^bb1
^bb1:
  %y = addi %x, %x : i32
  return %y : i32
}
```



MLIR Type System: some examples

Scalars:

- f16, bf16, f32, … i1, i8, i16, i32, … i3, i4, i7, i57, …

Vectors:

- vector<4 x f32>, vector<4x4 x f16>

Tensors, including dynamic shape and rank:

- tensor<4x4 x f32>, tensor<4x?x?x17x? x f32>, tensor<* x f32>

Others:

- functions/closures, memory buffers, quantized integers, TensorFlow stuff, ...

MLIR Operations: an open ecosystem

No fixed / builtin list of globally known operations:

- No “instruction” vs “target-indep intrinsic” vs “target-dep intrinsic” distinction
  - Why is “add” an instruction but “add with overflow” an intrinsic in LLVM? 😿

Passes are expected to conservatively handle unknown operations:

- just like LLVM does with unknown intrinsics

```
func @testFunction(%arg0: i32) -> i32 {
  %x = "any_unknown_operation_here"(%arg0, %arg0) : (i32, i32) -> i32
  %y = "my_increment"(%x) : (i32) -> i32
  return %y : i32
}
```

MLIR Operations Capabilities

Operations always have: opcode and source location info

Operations may have:

- Block arguments instead of PHI nodes
- Any number of SSA results and operands
- Attributes: constant values of custom syntax and type
- Regions: discussed in later slide
- Custom printing/parsing - or use the more verbose generic syntax

Extensible Operations Allow Multi-Level IR

TensorFlow

```
%x = "tf.Conv2d"(%input, %filter)
          {strides: [1,1,2,1], padding: "SAME", dilations: [2,1,1,1]}
    : (tensor<*xf32>, tensor<*xf32>) -> tensor<*xf32>
```

XLA HLO

```
%m = “xla.AllToAll"(%z)
          {split_dimension: 1, concat_dimension: 0, split_count: 2}
    : (memref<300x200x32xf32>) -> memref<600x100x32xf32>
```

LLVM IR

```
%f = "llvm.add"(%a, %b)
    : (f32, f32) -> f32
```

Also: TF-Lite, Core ML, other frontends, ...

😭 Don’t we end up with the JSON/XML of compiler IRs???

MLIR “Dialects”: Families of defined operations

Example Dialects:

- TensorFlow, XLA HLO, TF Lite, Swift SIL, ...
- linalg, affine, LLVM IR, ...

Dialects can define:

- Operations
- Custom type and attribute systems

Operation can define:

- Invariants on # operands, results, attributes, ...
- Custom parser, printer, verifier, ...
- Constant folding, canonicalization patterns, …

Operations in a Nutshell

- No predefined set of instructions
- Operations are like “opaque functions” to MLIR

Number of value returned

Dialect prefix

Op Id

Argument

Index in the producer’s results

List of attributes: constant named arguments

```
%res:2 = "mydialect.morph"(%input#3) { some.attribute : true, other_attribute : 1.5 }
          : (!mydialect<"custom_type">) -> (!mydialect<"other_type">, !mydialect<"other_type">)
                                      loc(callsite("foo" at "mysource.cc":10:8))
```

Name of the results

Dialect prefix for the type

Opaque string / Dialect specific type

Mandatory and Rich Location

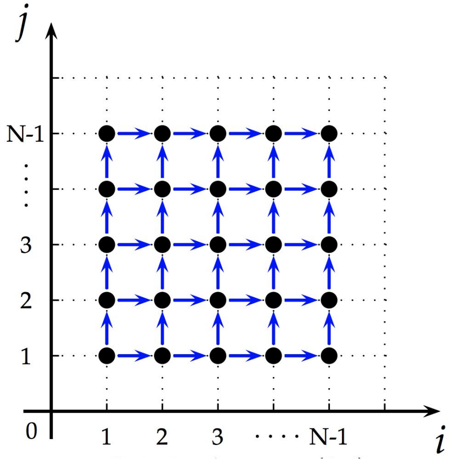
Nested Regions in a Linear IR

```
%2 = xla.fusion (%0 : tensor<f32>, %1 : tensor<f32>) : tensor<f32> {
^bb0(%a0 : tensor<f32>, %a1 : tensor<f32>):
  %x0 = xla.add %a0, %a1 : tensor<f32>
  %x1 = xla.relu %x0 : tensor<f32>
  return %x1
}

%7 = tf.If(%arg0 : tensor<i1>, %arg1 : tensor<2xf32>) -> tensor<2xf32> {
  … “then” code...
  return ...
} else {
  … “else” code...
  return ...
}
```

➔ Functional control flow, XLA fusion node, lambdas/closures, parallelism abstractions like OpenMP, etc.

Bigger Example: Polyhedral IR Dialect

```
func @matmul_square(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>) {
  %n = dim %A, 0 : memref<?x?xf32>

  affine.for %i = 0 to %n {
    affine.for %j = 0 to %n {
      store 0, %C[%i, %j]        : memref<?x?xf32>
      affine.for %k = 0 to %n {
        %a    = load %A[%i, %k] : memref<?x?xf32>
        %b    = load %B[%k, %j] : memref<?x?xf32>
        %prod = mulf %a, %b     : f32
        %c    = load %C[%i, %j] : memref<?x?xf32>
        %sum  = addf %c, %prod  : f32
        store %sum, %C[%i, %j]  : memref<?x?xf32>
      }
    }
  }
  return
}
```



affine.for and **affine.if** represent simplified polyhedral schedule trees:

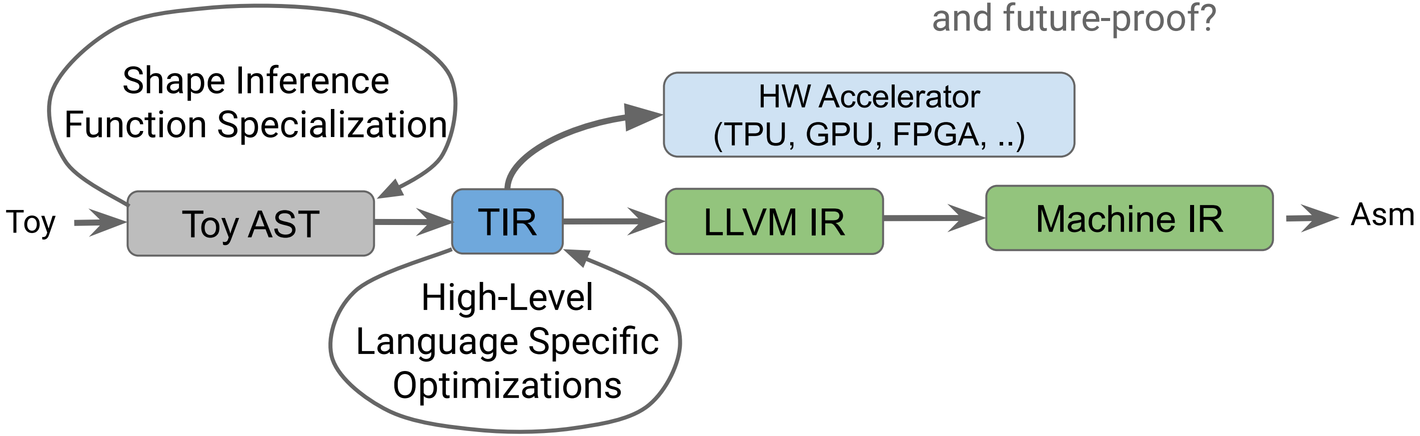
- Great match for ML kernels
- Includes systems of affine constraints, mappings, solvers, etc.

More on MLIR: See the EuroLLVM'19 Tutorial

Example: DSL and Compiler for a Heterogeneous World

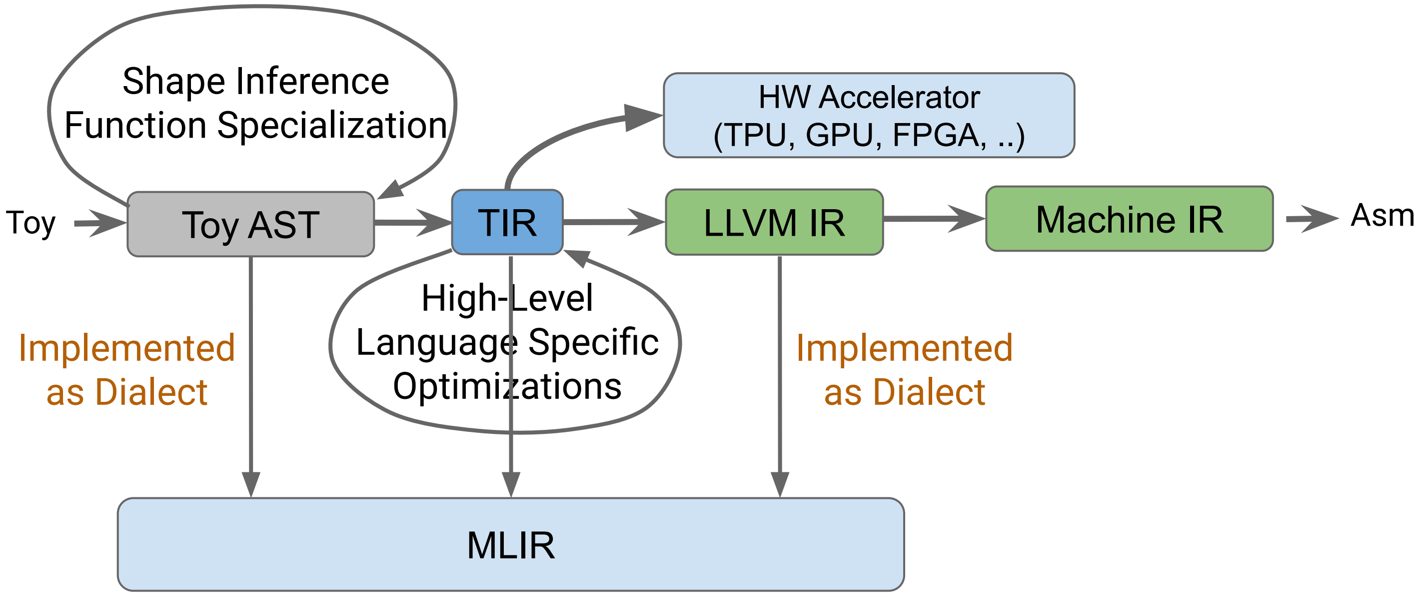
Need to analyze and transform the AST
→ heavy infrastructure
And is the AST really the most friendly
representation we can get?

New HW: are we extensible
and future-proof?



More on MLIR: See the EuroLLVM'19 Tutorial

It's All About Dialect(s)



MLIR is a Large Project...

Albert Cohen
Alex Zinenko
Alexandre Passos
Andrew Selle
Andy Davis
Bjarke Roune
Brian Patton
Chris Lattner
Cliff Young
David Majnemer
Daniel Killebrew
Dimitrios Vytiniotis
Feng Liu
Himabindu Pucha
Jacques Pienaar
James Molloy
Jeff Dean
Jianwei Xie
Lei Zhang
Mark Heffernan
Martin Wicke
Mehdi Amini
Michael Isard
Nicolas Vasilache
Paul Barham
Peter Hawkins
Rasmus Larsen
Richard Wei
River Riddle
Sanjoy Das
Sergei Lebedev
Skye Wanderman-Milne
Smit Hinsu
Sourabh Bajaj
Stella Laurenzo
Tatiana Shpeisman
Todd Wang
Uday Bondhugula
Ulysse Beaugnon
Yanan Cao

MLIR-Related Research Projects

Hackability & HW/SW Research

Aiming for a super-extensible system, catalyzing next-gen accelerator research:

- domain-specific languages / annotations lower naturally to MLIR
- domain-specific HW constructs are first-class operations
- extend type system: novel numerics, sparse tensors, (G)ADTs, …
- concurrency & parallel constructs, memory modeling
- many classes of transformations have *structured* search spaces: algorithmic rewriting, graph rewriting, memory-recompute, polyhedral, and synthesis

Accelerate innovation in hardware, compiler algorithms, and applications thereof

Compile to Learn → Learn to Compile

- Move past handwritten heuristics
  - NP complete problems
  - Cost models that are hard or infeasible to characterize
  - Hardware explosion, model diversity, problem diversity, … can't scale
- Autotuning, search and caching
  - Separate algorithms and policy
  - Exploit structure in search space

Two projects: past & future

Tensor Comprehensions & Telamon

Telamon: Commutative Optimizations on Partially Specified Implementations

with Ulysse Beaugnon, Basile Clément and Andi Drebes, Nicolas Tollenaere
ENS, Inria, Google

CC 2017: *Optimization space pruning without regrets*
Ulysse Beaugnon, Antoine Pouille, Marc Pouzet, Jacques Pienaar, Albert Cohen

arXiv preprint: *On the Representation of Partially Specified Implementations and its Application to the Optimization of Linear Algebra Kernels on GPU*
Ulysse Beaugnon, Basile Clément, Nicolas Tollenaere, Albert Cohen

Problem Statement

Context: “superoptimizing” loop nests in numerical kernels
Challenge: finding good/best combinations of implementation decisions is hard

- Optimizations may enable or disable others
- Transformations ordering affects performance
- Cannot infer precise performance estimation from intermediate compilation steps

Corollary: optimizing compilation never seems to catch up... new hardware, optimization tricks… effectively witnessing a widening performance portability gap

Telamon Approach

Candidates as Partially Specified Implementations

- Optimizations as independent, commutative decisions
  e.g., tile? unrolling? ordering?
- Vector of choices, listed upfront, decisions taken in any order
  defer any interference to search
- Synthesize imperative code from fixed/complete decision vectors
  e.g., infer buffers, control flow

Constraint Programming for Semantics and Resource Modeling

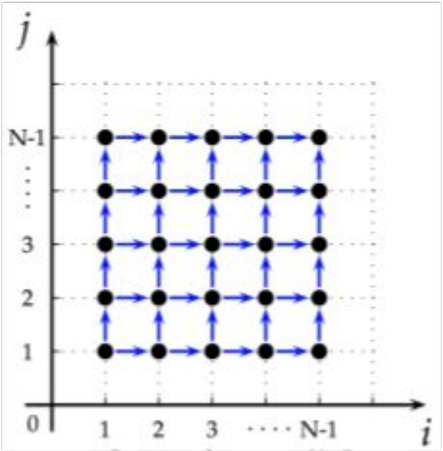
- Control structure
  e.g., loop nesting
- Semantics of the kernel
  e.g., def-use, array dependences
- Optimization interactions
  e.g., enabling transformations
- Resource constraints
  e.g., local memory

Branch-and-Bound- and MCTS-enabled **Search**

- Lower bound derived from orthogonal resource modeling
  inspired by roofline modeling
- Lower bound for a candidate = ideal performance for a set of potential implementations
- Empowered by structured, decision vector and CSP-based implementation space

Candidates?

Inspired From Polyhedral Compilation



- Polyhedral compilation
  - Affine scheduling
    e.g., ILP-based
  - Code generation
    from affine schedules to nested loops

- Meta-programming array processing code
  - Halide / TVM specific combinators and scheduling/mapping primitives
  - URUK, CHiLL with automatic schedule completion

TVM example: scan cell (RNN)

```
m = tvm.var("m")
n = tvm.var("n")
X = tvm.placeholder((m,n), name="X")
s_state = tvm.placeholder((m,n))
s_init = tvm.compute((1,n), lambda _,i: X[0,i])
s_do = tvm.compute((m,n), lambda t,i: s_state[t-1,i] + X[t,i])
s_scan = tvm.scan(s_init, s_update, s_state, inputs=[X])
s = tvm.create_schedule(s_scan.op)
// Schedule to run the scan cell on a CUDA device
block_x = tvm.thread_axis("blockIdx.x")
thread_x = tvm.thread_axis("threadIdx.x")
xo,xi = s[s_init].split(s_init.op.axis[1], factor=num_thread)
s[s_init].bind(xo, block_x)
s[s_init].bind(xi, thread_x)
xo,xi = s[s_do].split(s_do.op.axis[1], factor=num_thread)
s[s_do].bind(xo, block_x)
s[s_do].bind(xi, thread_x)
print(tvm.lower(s, [X, s_scan], simple_mode=True))
```

Constraints?

Inspired From Program Synthesis and Superoptimization

- Program synthesis
    - Start from denotational specification, possibly partial (sketching), or (counter-)examples
      Telamon ≈ Domain-specific denotations
    - Guess possible implementations by (guided) sampling lots of random ones
      Telamon ≈ Guess efficient implementations by (guided) sampling lots of stupid ones
    - Filter correct implementations using SMT solver or theorem prover
      Telamon ≈ Constraint programming to model both correctness and hardware mapping
- Superoptimization
    - Typically on basic blocks, with SAT solver or theorem prover and search
    - Architecture and performance modeling

Search?

Inspired From Adaptive Libraries and Autotuning

- Feedback-directed and iterative compiler optimization, lots of work since the late 90s
- Adaptive libraries
  - SPIRAL: *Domain-Specific Language (DSL) + Rewrite Rules + Multi-Armed Bandit or MCTS*
    http://www.spiral.net
  - ATLAS, FFTW, etc.: *hand-written fixed-size kernels + micro-benchmarks + meta-heuristics*
  - Pouchet et al. (affine), Park et al. (affine and CFG): *Genetic Algorithm, SVM, Graph Kernels*
- Telamon
  - vs. SPIRAL, FFTW: better structured, independent/commutative choices, branch-and-bound
  - vs. Pouchet and Park: finite space, bounded vectors

Candidates

Partially Instantiated Vector of Decisions

- Every choice is decision variable
- List the *domain* of variables: the values they can take
- Taking a decision = restricting a domain
- Fully specified implementation ⇔ All decision variables assigned a single value

```
- order(a, b) ∈ { Before, After }
- order(a, c) ∈ { Before }
- …
```

Candidates and Constraints

Kernel

```
%x = load %X[0]
%y = add %x, %d0
for %d0 = 0 to 1024 {
    %z = add %y, %d0
}
```

Decisions

```
order(%x, %d0) ∈ { Before, Inner } <- decision
order(%x, %y)  ∈ { Before }
order(%y, %d0) ∈ { Before, Inner } <- constraint propagation
...
```

Enforce coherent decisions with constraints

```
order(x, d0) = Inner && order(x, y) = Before => order(y, d0) ∈ { Inner, After }
```

Enabling Better Search Algorithms

Well Behaved Set of Actions

- Commute
- All decisions known upfront
- Constraint propagation almost never backtracks in practice

Flat, Fixed Sized, Ideal Environment for Reinforcement Learning (RL)

- Extract features from the decision vector
- Global heuristics, aware of all potential optimizations
- Infer all possible decisions (actions) and/or estimate performance

Constraint Satisfaction Problem (CSP)

Find an Assignment for Functions

```
kind: Dimension -> { Loop, Unrolled, Vector, Thread, Block }

order: Statements x Statements -> { Before, After, Inner, Outer, Fused }
```

That Respects Constraints

```
∀ a, b ∈ Dimension. order(a, b) = Fused => kind(a) = kind(b)
```

(a.k.a. typed fusion)

CSP Without the Optimization Aspect

- Finding an implementation is easy: random decisions + constraint propagation
- Use CSP to represent, not to solve the problem
- No analytical objective function
    - Analytical functions cannot model the complexity of the hardware
    - Hardware details are proprietary
    - Use actual evaluations
    - And external heuristics

Evaluation: Telamon on GPU

Generic loop nest and array optimizations + GPU-specific optimizations

Supported Decisions

- Strip mining factor
- Loop interchange
- Loop fusion
- Statement Scheduling
- Rematerialization

- Array placement in memory spaces
- Memory layout
- Copy to local memories
- Vectorization

Example: Vector Addition

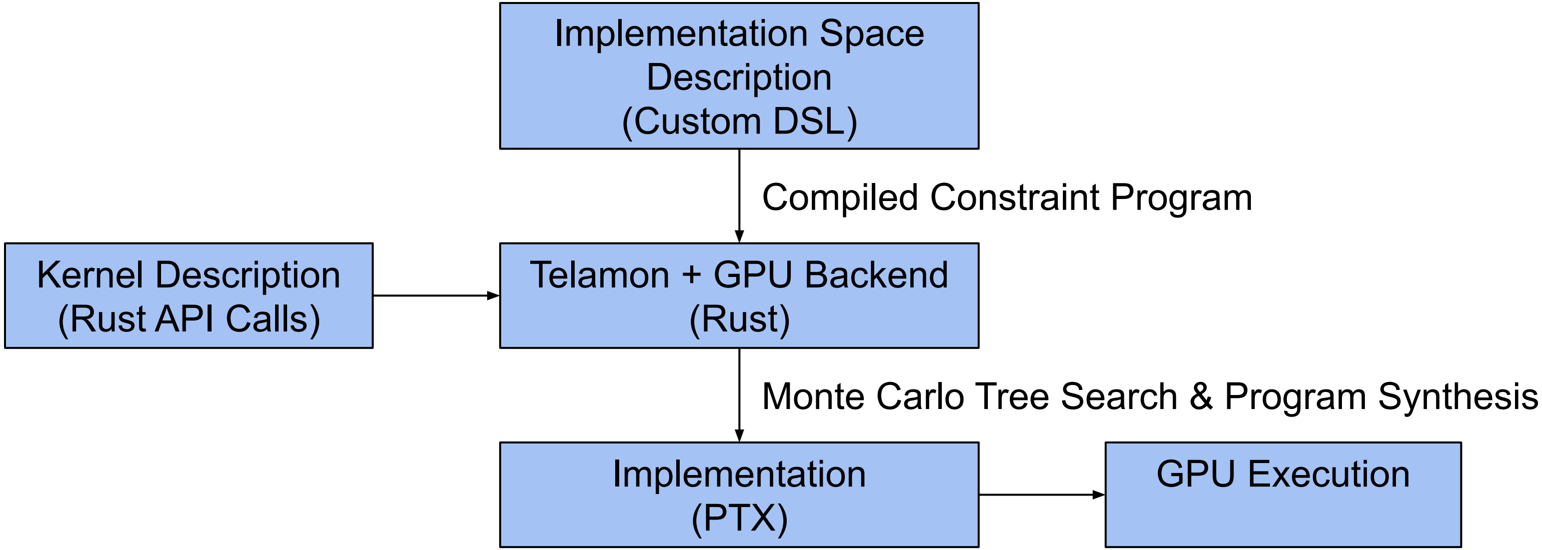
Computes Z = X + Y

```
-   load X
-   load Y
-   add X and Y into Z
-   store Z
```

Implementation space

- Each instruction in its own loop
- Strip-mined 3 times
- Can choose strip-mining factors
- Can fuse, interchange and unroll loops
- Can reorder instructions
- Can coalesce transfers across memory spaces

Telamon System Overview



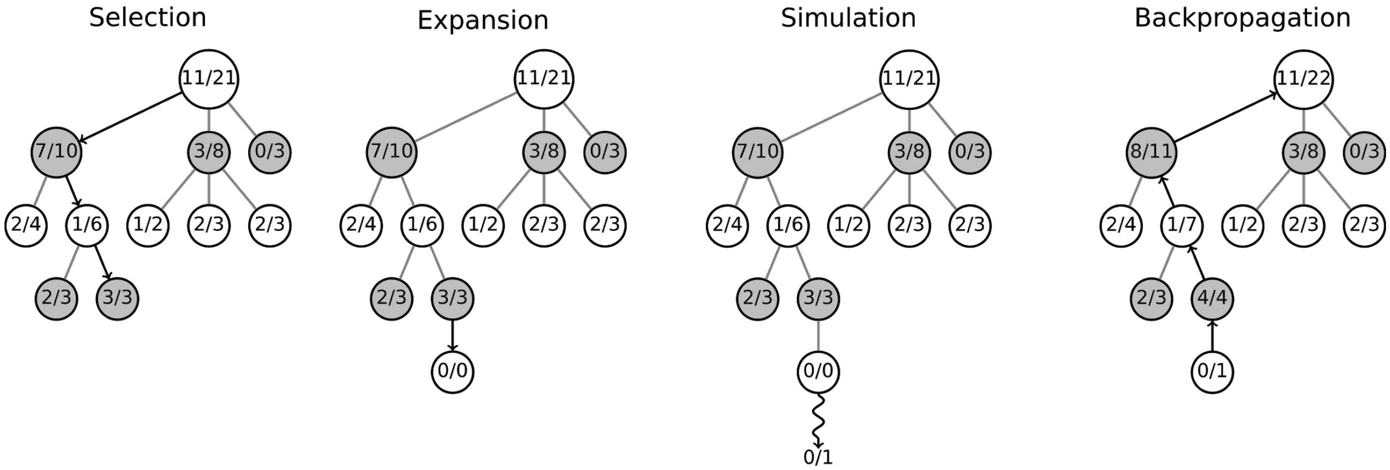
Branch and Bound + Monte Carlo Tree Search (MCTS)

Performance model of a lower bound on the execution time

$$\forall x \in S.\ \mathrm{Model}(S) \leq \mathrm{Time}(x)$$

- Enables Branch & Bound, with feedback from real executions
  - Reduces the search space by several orders of magnitude
  - Prunes early in the search tree (75% in the first two levels for matmul on GPU)
- Possible because it is aware of potential future decisions
- GPU model of block- and thread-level performance features, as well as single-thread microarchitecture
  - No cache and registers model (yet)
  - Coarse-grain model of the interaction between bottlenecks

Zooming in the MCTS-Based Search



Source: Wikipedia

Results on Nvidia Kepler GPU

Kernel	Space Size	Avg. Runtime	Reference	Speedup
axpy	$1.1e11 \pm 1.7e10$	$7.05ms \pm 0.005$	$10.3ms$	**1.47** $\pm 10^{-3}$
matmul 256	$1.83e21 \pm 3.3e20$	$34.2\mu s \pm 2.54$	$82.8\mu s$	**2.42** ± 0.18
matmul 1024	$3.5e21 \pm 1.8e21$	$4.81ms \pm 0.06$	$3.75ms$	**0.78** ± 0.01
strided matmul	$6.0e20 \pm 2.0e20$	$10.1ms \pm 0.59$	$637ms$	**66.7** ± 3.9
batched matmul	$2.5e28 \pm 1.3e28$	$177\mu s \pm 37.6$	$433\mu s$	**2.45** ± 0.52
reuse matmul	$7.1e25 \pm 4.5e25$	$143\mu s \pm 39.8$	$436\mu s$	**3.05** ± 0.85

Search Issues (Ongoing Research)

- **High variance of the search time (stuck in suboptimal areas)**
- **Lots of dead-ends**
  - Mostly due to performance model
  - ~20x more dead-ends than implementations
- **Non-stationary distribution due to cuts**
  - Somewhat intrinsic to MCTS
  - Branch & bound strategy makes it trickier

Take Home Message

In a Nutshell — Benefits of Polyhedral Compilation

Search Space
Abstract, Partially Specified Implementations

- Optimizations and lowering, choices and transformations
  e.g., tile? unrolling? ordering?
- Choice vector or sequence of transformations/rewrite rules
  combine with search
- Synthesize imperative code, API calls, assembly code
  infer buffers, control...

Constraints
Functional Semantics and Resource Modeling

- Control structure
  e.g., loop nesting
- Semantics of the kernel
  e.g., def-use, array dependences
- Optimization interactions
  e.g., enabling transformations
- Resource constraints
  e.g., local memory, DMA

Search Heuristics
Semi-automatic or Black-box Optimization

- Objective functions
  linear approximations, resource counting...
- Feedback from actual execution
  profile-directed, JIT, trace-based...
- Combinatorial optimization
  ILP, SMT, graph algorithms, reinforcement learning...

With numerous applications:
compiler construction, domain-specific optimization, performance portability