Polyhedral Compilation as a Design Pattern for Compiler Construction

PLISS, May 19-24, 2019
albertcohen@google.com
Polyhedra? Example: Tiles
How many of you read “Design Pattern”? → Polyhedra? Example: Tiles
Tiles Everywhere

1. Hardware

Example: Google Cloud TPU
Architectural Scalability With Tiling
Tiles Everywhere

1. Hardware

Google Edge TPU

- TensorFlow model
  - 32-bit float numbers
  - Include fake quantization nodes
- TRAIN
- Quantization aware training
- TensorFlow model
  - 8-bit fixed numbers
- EXPORT
- Frozen graph
  - .pb file
- CONVERT
- TOCO
- TensorFlow Lite
  - .tflite file
- COMPILE
- Edge TPU model
  - .tflite file
- DEPLOY
- Edge TPU Hardware

Edge computing zoo
Tiles Everywhere

1. Hardware
2. Data Layout

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 loop unrolling and computing kernels

Tiling in Halide

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

Vectorized schedule:
- strip-mine
- vectorize inner loop

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)**

```python
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
Generic Loop Nest Optimizer

E.g. Intel ICC, Pluto, PPCG, LLVM/Polly
Generic Loop Nest Optimizer

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

Domain-Specific Optimizer and Code Generator

E.g., XLA, Halide, Polymage

(this is a hammer)

(these are not nails)
Polyhedral Framework =
semantical and algorithmic design pattern
for multi-purpose representation, analysis,
transformation, optimization, code generation

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

Domain-Specific
Optimizer and
Code Generator

E.g., XLA, Halide, Polymage

(this is a hammer)

(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.;                          // Statement 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];
    }
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

\[ C[0][0] = 0.; \quad C[0][1] = 0.; \quad C[0][2] = 0.; \quad //... \]

\[ C[1][0] = 0.; \quad C[1][1] = 0.; \quad C[1][2] = 0.; \quad //... \]

\[ C[2][0] = 0.; \quad C[2][1] = 0.; \quad C[2][2] = 0.; \quad //... \]

/*...*/
Iteration Domain

*Iteration domain*: set of all statement instances.
Iteration Domain

*Iteration domain*: set of all statement instances.

```c
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](http://repo.or.cz/w/isl.git)

Sven Verdoolaege

\[
D_{S2} = (H,W,KH,KW) \rightarrow \\
\{ S2(i,j,k,l): 0 \leq i < H-KH && \\
    0 \leq j < W-KW && \\
    0 \leq k < KH && \\
    0 \leq l < KW \}
\]
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

\[ t = c_0 + c_1 \cdot i_1 + c_2 \cdot i_2 + c_3 \cdot i_3 + ... + k_1 \cdot W + k_2 \cdot H + k_3 \cdot KH + k_4 \cdot KW \]

where \( c_0, c_1, c_2, ..., k_1, k_2, k_3, k_4 \) are constants, and \( i_1, i_2, i_3, ... \) are iteration variables

* We will explain later why
Multi-Dimensional Schedules

To obtain nested loop structure like

```c
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 $C[x-1][] + \text{induction}$, but

$$t = (W - KW) \times i + j$$

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

\[(i,j) \rightarrow (i,j)\]
Affine Schedules

$i$

$j$

$time$
Multi-Statement Schedules

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

```c
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

```c
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**
  - $S1 \rightarrow (0,i,j,1,0)$
  - $S2 \rightarrow (1,i,j,0,k)$
  - $S3 \rightarrow (0,i,j,0,0)$
  - $S4 \rightarrow (1,i,k,1,j)$

- **Permute, fuse inner**
  - $S1 \rightarrow (0,i,0,0,j)$
  - $S2 \rightarrow (0,i,k,1,j)$
  - $S3 \rightarrow (1,i,0,0,j)$
  - $S4 \rightarrow (1,i,k,1,j)$

> 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
\[ t_1 = g(i, j, k) = \text{floor}(i / 42) \]
can be transformed into an affine relation
\[ \{(i, j, k) \rightarrow (t_1) : 42t_1 \leq i \leq 42t_1 + 41\} \]
Polyhedral/Affine Scheduling

- Iteratively produce affine schedule functions such that:
- Dependence distances are *lexicographically* positive
- Dependence distances are small $\Rightarrow$ locality
- Dependence distances are zero $\Rightarrow$ parallelism
- Dependences have non-negative distances along consecutive dimensions $\Rightarrow$ permutability (which enables tiling)

\[
\begin{align*}
\text{permutable} & \quad \text{permutable} \\
(0,1,0,0) & \quad (0,1,-2,3) \\
\text{valid} & \quad \text{also valid} \\
& \quad (0,0,-1,42) \quad \text{violated}
\end{align*}
\]

Generally, dependences $= \text{RAW} + \text{WAR} + \text{WAW}$
Polyhedral/Affine Scheduling

- Iteratively produce affine scheduling functions of shape:

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

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

for every dependence \(R \rightarrow S\)
Polyhedral/Affine Scheduling

- Iteratively produce affine scheduling functions of shape:

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

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

for every dependence \( R \rightarrow S \)

\[ \text{lexmin } \vec{u}, w, \vec{a}, \vec{b}, d \text{ s.t. } \vec{u} \geq \vec{0} \]

\[ \rightarrow \text{ Integer Linear Programming (ILP) problem} \]

use the affine form of the Farkas lemma to linearize the inequality
isl Schedules Trees: Compose w/ Imperative Semantics

(a) canonical \texttt{sgemm} 

\begin{align*}
\text{Domain} & : \{S(i, j) \mid 0 \leq i < N \land 0 \leq j < K\} \\
& \quad \cup \{T(i, j, k) \mid 0 \leq i < N \land 0 \leq j < K \land 0 \leq k < M\}
\end{align*}

\begin{align*}
\text{Sequence} & : \text{Filter}\{S(i, j)\} \\
& \quad \cup \text{Band}\{S(i, j) \rightarrow (i, j)\} \\
& \quad \cup \text{Filter}\{T(i, j, k)\} \\
& \quad \cup \text{Band}\{T(i, j, k) \rightarrow (i, j, k)\}
\end{align*}

(b) fused 

\begin{align*}
\text{Domain} & : \{S(i, j) \mid 0 \leq i < N \land 0 \leq j < K\} \\
& \quad \cup N = M = 16 \land K = 1000
\end{align*}

\begin{align*}
\text{Context} & : N = M = 16 \land K = 1000 \\
& \quad \cup \{S(i, j) \mid i = 32 \times 2 - 31 \leq j < 32 \times 16 \mid i \} \leq i - 32 \times 32 \land j < 32 \times 16 \mid j \} \leq j - 32 \times 32 \\
& \quad \cup \{T(i, j, k) \mid i = 32 \times 2 - 31 \leq j < 32 \times 16 \mid j \} \leq j - 32 \times 32 \\
& \quad \cup \text{Band}\{S(i, j) \rightarrow (i, j)\} \\
& \quad \cup \text{Filter}\{T(i, j, k)\} \\
& \quad \cup \text{Band}\{T(i, j, k) \rightarrow (i, j, k)\}
\end{align*}

(c) fused and tiled 

\begin{align*}
\text{Domain} & : \{S(i, j) \mid 0 \leq i < N \land 0 \leq j < K\} \\
& \quad \cup N = M = 256 \land K = 1 \leq k < M
\end{align*}

\begin{align*}
\text{Context} & : N = M = 256 \land K = 1 \leq k < M \\
& \quad \cup \{S(i, j) \mid i = 32 \times 2 - 31 \leq j < 32 \times 16 \mid i \} \leq i - 32 \times 32 \land j < 32 \times 16 \mid j \} \leq j - 32 \times 32 \\
& \quad \cup \{T(i, j, k) \mid i = 32 \times 2 - 31 \leq j < 32 \times 16 \mid j \} \leq j - 32 \times 32 \\
& \quad \cup \text{Band}\{S(i, j) \rightarrow (i, j)\} \\
& \quad \cup \text{Filter}\{T(i, j, k)\} \\
& \quad \cup \text{Band}\{T(i, j, k) \rightarrow (i, j, k)\}
\end{align*}

(d) fused, tiled and sunk 

\begin{align*}
\text{Domain} & : \{S(i, j) \mid 0 \leq i < N \land 0 \leq j < K\} \\
& \quad \cup N = M = 1 \leq k < M
\end{align*}

\begin{align*}
\text{Context} & : N = M = 1 \leq k < M \\
& \quad \cup \{S(i, j) \mid i = 32 \times 2 - 31 \leq j < 32 \times 16 \mid i \} \leq i - 32 \times 32 \land j < 32 \times 16 \mid j \} \leq j - 32 \times 32 \\
& \quad \cup \{T(i, j, k) \mid i = 32 \times 2 - 31 \leq j < 32 \times 16 \mid j \} \leq j - 32 \times 32 \\
& \quad \cup \text{Band}\{S(i, j) \rightarrow (i, j)\} \\
& \quad \cup \text{Filter}\{T(i, j, k)\} \\
& \quad \cup \text{Band}\{T(i, j, k) \rightarrow (i, j, k)\}
\end{align*}

(e) fused, tiled, sunk and mapped

Optimization steps for \texttt{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:

\[
\begin{align*}
t_0 &= i^2jN + jk; \\
t_1 &= k^j - i; \\
t_2 &= j^k(i-1);
\end{align*}
\]

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

Iteration Domain

Array
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

Iteration Domain

Array

\[ A[\] \]
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

Distance 1

Distance 0

\[ j+1 \]

\[ j \]

\[ i \]
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
  with no zeros on the main diagonal

- Upper triangular

- 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)
  - *Independence* (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)

\[
\text{for (int } i = 0; i < NI; ++i) \quad \text{// distance } 0 \Rightarrow \text{ parallel}
\]
\[
\text{for (int } j = 0; j < NJ; ++j) \quad \text{// distance } \ast \Rightarrow \text{ sequential}
\]
\[
A[i] \, += \, 42.;
\]

\textbf{OP: Outer Parallelism}

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

\textbf{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).

```c
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.

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

**DG F: 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.

```c
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]
```
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)

**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 \textit{iteration domains} and \textit{access functions} to classify programs and select the corresponding sequence of cost functions.

<table>
<thead>
<tr>
<th>Program class</th>
<th>Objective functions (lexicographical order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stencils</td>
<td>(stencil-specific dependence analysis) + SMVS, SPAR</td>
</tr>
<tr>
<td>Max-2D-loops</td>
<td>SO, IP, OPIR, SIS, DGF, OP</td>
</tr>
<tr>
<td>Dense Linear Algebra</td>
<td>\text{&lt;SO, IR, OP&gt;(conditioned on # of dep.)}, SIS, DGF, OP</td>
</tr>
<tr>
<td>Other</td>
<td>\text{SO(conditioned on # of dep.)}, OP + restrict schedule coefficients</td>
</tr>
</tbody>
</table>
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
Autotuning Search Space
Refinements: Scalability and Customizable Incremental Scheduling
Scalability: Proximity Relation Grouping

- Grouping “sufficiently similar” accesses:
- **Access rank:**
  - prioritize array references with more subscripts.
- **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 \parallel 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

```c
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

```c
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

```c
void lu(double A[N][N]) {
    for (i = 0; i < N; i++) {
        for (j = 0; j < i; j++) {
            for (k = 0; k < j; k++)
        }
        for (j = i; j < N; j++)
            for (k = 0; k < i; k++)
    }
}
```

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


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


[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](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
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?

A tale of many layers

Input

\( W_1 \) → Conv

\( B_1 \) → Add

ReLU

... → Caffe2

caffe2.python.brew.conv() ...

cudnnConvolutionForward()...

... → PyTorch

torch.nn.conv2d() ...

dnnConvolutionCreateForward_F32() ...

... → TensorFlow*

tf.contrib.layers.conv2d() ...

* TF also can compile via XLA, discussed later
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.

Our Approach

Tensor Comprehensions

- Polly
- Polyhedral Transformations
- Tapir/LLVM
- Cilk/OpenMP
- Halide IR
- Polyhedral IR (ISL)
- CUDA Kernel
- CUDA Module
- Exec
- ATen
- libTHC.so
- Range Inference and Specialization
TC Language

Concise, emits 1000’s of optimized LOC

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

    -> (01, 02, 03) {
    01(n, o, h, w) += I(n, c, h + kh, w + kw) * W1(o, c, kh, kw)
    01(n, o, h, w) = fmax(01(n, o, h, w), 0) // relu
    02(n, d, h, w) += 01(n, d, h + kh, w + kw) * W2(d, o, kh, kw)
    02(n, d, h, w) = fmax(02(n, d, h, w), 0)
    03(n, e, h, w) += 02(n, c, h + kh, w + kw) * W3(e, d, kh, kw)
    03(n, e, h, w) = fmax(03(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**

  ```python
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)**

  → algorithmic exploration of storage/recompute tradeoffs

  ```python
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,r1) * 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)
}
```
• Synthesize From Model...

- Production MLP from Facebook

- And more complex examples, including Google WaveNet
Rather Than Write Accelerator Code
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

```python
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**: 
  - \{S(i,j) \mid 0 \leq i < N \land 0 \leq j < K\}
  - \{T(i,j,k) \mid 0 \leq i < N \land 0 \leq j < K \land 0 \leq k < M\}

- **Sequence**
  - Filter\{S(i,j)\}
  - Band\{S(i,j) \rightarrow (i,j)\}
  - Filter\{T(i,j,k)\}
  - Band\{T(i,j,k) \rightarrow (i,j)\}

- **Tile**
  - 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

---

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

---

\[
\begin{align*}
\text{Domain: } & \quad [S(i, j) \mid 0 \leq i < N \land 0 \leq j < K] \\
& \quad \{T(i, j, k) \mid 0 \leq i < N \land 0 \leq j < K \land 0 \leq k < M\}
\end{align*}
\]

\[
\begin{align*}
\text{Sequence: } & \quad \text{Filter}\{S(i, j)\} \\
& \quad \text{Band}\{T(i, j, k) \rightarrow (i, j, k)\}
\end{align*}
\]

(a) canonical `sgemm`

\[
\begin{align*}
\text{Domain: } & \quad [S(i, j) \mid 0 \leq i < N \land 0 \leq j < K] \\
& \quad \{T(i, j, k) \mid 0 \leq i < N \land 0 \leq j < K \land 0 \leq k < M\}
\end{align*}
\]

\[
\begin{align*}
\text{Filter: } & \quad \{S(i, j)\} \\
\text{Band: } & \quad \{T(i, j, k) \rightarrow (i, j, k)\}
\end{align*}
\]

(b) fused

\[
\begin{align*}
\text{Domain: } & \quad [S(i, j) \mid 0 \leq i < N \land 0 \leq j < K] \\
& \quad \{T(i, j, k) \mid 0 \leq i < N \land 0 \leq j < K \land 0 \leq k < M\}
\end{align*}
\]

\[
\begin{align*}
\text{Filter: } & \quad \{S(i, j)\} \\
\text{Band: } & \quad \{T(i, j, k) \rightarrow (i \mod 32, j \mod 32)\}
\end{align*}
\]

(c) fused and tiled

\[
\begin{align*}
\text{Domain: } & \quad [S(i, j) \mid 0 \leq i < N \land 0 \leq j < K] \\
& \quad \{T(i, j, k) \mid 0 \leq i < N \land 0 \leq j < K \land 0 \leq k < M\}
\end{align*}
\]

\[
\begin{align*}
\text{Sequence: } & \quad \text{Filter}\{S(i, j)\} \\
& \quad \text{Band}\{T(i, j, k) \rightarrow (i \mod 32, j \mod 32)\}
\end{align*}
\]

(d) fused, tiled and sunk

\[
\begin{align*}
\text{Domain: } & \quad [S(i, j) \mid 0 \leq i < N \land 0 \leq j < K] \\
& \quad \{T(i, j, k) \mid 0 \leq i < N \land 0 \leq j < K \land 0 \leq k < M\}
\end{align*}
\]

\[
\begin{align*}
\text{Sequence: } & \quad \text{Filter}\{S(i, j)\} \\
& \quad \text{Band}\{T(i, j, k) \rightarrow (i \mod 32, j \mod 32)\}
\end{align*}
\]

(e) fused, tiled, sunk and mapped
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

\[ O[l+Idx[i][j]][k] \Rightarrow \text{shared}_O[l][i][j][k] \]

- 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 \( \geq 1 \) 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.
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 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
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](https://tensorflow.org/xla/overview) (video overview)
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:

```swift
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

```swift
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**

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

**XLA HLO**

\[
\%m = "xla.AllToAll"(%z) \\
\quad \{\text{split_dimension: 1, concat_dimension: 0, split_count: 2}\} \\
\quad : (\text{memref<300x200x32xf32>}) \to \text{memref<600x100x32xf32>}
\]

**LLVM IR**

\[
\%f = "llvm.add"(%a, %b) \\
\quad : (\text{f32, f32}) \to \text{f32}
\]

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

%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
- Op Id
- Argument
- Index in the producer's results
- List of attributes: constant named arguments
- Number of value returned
- Dialect prefix
- Argument
- Opaque string / Dialect specific type
- Mandatory and Rich Location
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)**

```python
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))
```
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

Constraints?
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](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

Search?
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

- \( \text{order}(a, b) \in \{ \text{Before, After} \} \)
- \( \text{order}(a, c) \in \{ \text{Before} \} \)
- ...
Candidates and Constraints

Kernel

\%x = load X[0]
\%y = add \%x, 42
for \%d0 = 0 to 16 {
  \%z = add \%y, \%d0
}

Decisions

order(\%x, \%d0) \in \{ \text{Before, Inner} \} <- \text{decision}
order(\%x, \%y) \in \{ \text{Before} \}
order(\%y, \%d0) \in \{ \text{Before, Inner} \} <- \text{constraint propagation}
...

Enforce coherent decisions with constraints

order(x, d0) = Inner && order(x, y) = Before \Rightarrow order(y, d0) \in \{ \text{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

- Kernel Description (Rust API Calls)
- Implementation Space Description (Custom DSL)

Compiled Constraint Program

- Telamon + GPU Backend (Rust)
- Monte Carlo Tree Search & Program Synthesis

- Implementation (PTX)
- GPU Execution
Branch and Bound + Monte Carlo Tree Search (MCTS)

Performance model of a lower bound on the execution time

\[ \forall x \in S. \ Model(S) \leq \ Time(x) \]

- 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

### Results on Nvidia Kepler GPU

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Space Size</th>
<th>Avg. Runtime</th>
<th>Reference</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>axpy</td>
<td>$1.1\times10^{11} \pm 1.7\times10^{10}$</td>
<td>$7.05,\text{ms} \pm 0.005$</td>
<td>$10.3,\text{ms}$</td>
<td>$1.47 \pm 10^{-3}$</td>
</tr>
<tr>
<td>matmul 256</td>
<td>$1.83\times10^{21} \pm 3.3\times10^{20}$</td>
<td>$34.2,\mu\text{s} \pm 2.54$</td>
<td>$82.8,\mu\text{s}$</td>
<td>$2.42 \pm 0.18$</td>
</tr>
<tr>
<td>matmul 1024</td>
<td>$3.5\times10^{21} \pm 1.8\times10^{21}$</td>
<td>$4.81,\text{ms} \pm 0.06$</td>
<td>$3.75,\text{ms}$</td>
<td>$0.78 \pm 0.01$</td>
</tr>
<tr>
<td>strided matmul</td>
<td>$6.0\times10^{20} \pm 2.0\times10^{20}$</td>
<td>$10.1,\text{ms} \pm 0.59$</td>
<td>$637,\text{ms}$</td>
<td>$66.7 \pm 3.9$</td>
</tr>
<tr>
<td>batched matmul</td>
<td>$2.5\times10^{28} \pm 1.3\times10^{28}$</td>
<td>$177,\mu\text{s} \pm 37.6$</td>
<td>$433,\mu\text{s}$</td>
<td>$2.45 \pm 0.52$</td>
</tr>
<tr>
<td>reuse matmul</td>
<td>$7.1\times10^{25} \pm 4.5\times10^{25}$</td>
<td>$143,\mu\text{s} \pm 39.8$</td>
<td>$436,\mu\text{s}$</td>
<td>$3.05 \pm 0.85$</td>
</tr>
</tbody>
</table>
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