Meta-programming for Cross-Domain Tensor Optimizations

Adilla Susungi\textsuperscript{1}, Norman A. Rink\textsuperscript{2}, Albert Cohen\textsuperscript{3}, Jerónimo Castrillón\textsuperscript{2}, Claude Tadonki\textsuperscript{1}

\textsuperscript{1}MINES ParisTech, PSL Research University
\textsuperscript{2}Chair for Compiler Construction, Technische Universität Dresden
\textsuperscript{3}Inria, Ecole normale supérieure

17th International Conference on Generative Programming: Concepts & Experiences (GPCE’18)
Boston, USA
November 5, 2018
Tensor optimizations and frameworks

Tensors
- Fundamental algebraic structure with applications to many domains
- Operations on multi-dimensional and computationally intense loop nests
- Involves multiple optimization strategies: loop, data layout, algebraic transformations, mapping decisions, etc.

Existing optimizing frameworks
- Built-in strategies do not always generalize well
- Lack of flexibility in composing finely tuned, target-specific optimizations
Transformation meta-languages

Meta-languages offering transformation heuristics as first-class citizens

URUK (Cohen et al., ICS’05)

CHiLL (Chen et al., 2008)

Halide (Ragan-Kelley et al., PLDI’14)

Clay (Bagnères et al., CGO’16)

Lift (Steuwer et al., CGO’17)

TVM (Chen et al., 2018)
Transformation meta-languages

We are interested in meta-languages for program transformation, because

- They help increasing expert productivity when hand-writing optimizations
- They ease the composition and cancellation of transformations
- They make the optimization paths explicit and future-proof

Strong allies for building adaptive, portable and efficient compiler infrastructures to face the complexity of parallel architectures
Contributions outline

Keys to

- Widen optimization search space
- Enhance the ability to flexibly compose optimization paths
- Formally characterize their semantics

Design and semantics of a tensor optimizations meta-language (TeML)
TeML overview

Every function returns either
- Tensors
- Loops

Operations on tensors
- Computation specification
- Layout transformations
- Data initialization, mapping

Operations on loops
- Expansion from tensor computation
- Transformation

\[
\begin{align*}
\langle \text{program} \rangle & \quad ::= \quad \langle \text{stmt} \rangle \langle \text{program} \rangle \\
& \quad \quad | \quad \varepsilon \\
\langle \text{stmt} \rangle & \quad ::= \quad \langle \text{id} \rangle = \langle \text{expression} \rangle \\
& \quad \quad | \quad \langle \text{id} \rangle = @\langle \text{id} \rangle : \langle \text{expression} \rangle \\
& \quad \quad | \quad \text{codegen} (\langle \text{ids} \rangle) \\
& \quad \quad | \quad \text{init} (...) \\
\langle \text{expression} \rangle & \quad ::= \quad \langle \text{Texpression} \rangle \\
& \quad \quad | \quad \langle \text{Lexpression} \rangle \\
\langle \text{Texpression} \rangle & \quad ::= \quad \text{scalar} () \\
& \quad \quad | \quad \text{tensor} (\langle \text{ints} \rangle) \\
& \quad \quad | \quad \text{eq} (\langle \text{id} \rangle, \langle \text{iters} \rangle? \rightarrow \langle \text{iters} \rangle) \\
& \quad \quad | \quad \text{vop} (\langle \text{id} \rangle, \langle \text{id} \rangle, [\langle \text{iters} \rangle?, \langle \text{iters} \rangle?]) \\
& \quad \quad | \quad \text{op} (\langle \text{id} \rangle, \langle \text{id} \rangle, [\langle \text{iters} \rangle?, \langle \text{iters} \rangle?] \rightarrow \langle \text{iters} \rangle) \\
\langle \text{Lexpression} \rangle & \quad ::= \quad \text{build} (\langle \text{id} \rangle) \\
& \quad \quad | \quad \text{stripmine} (\langle \text{id} \rangle, \langle \text{int} \rangle, \langle \text{int} \rangle) \\
& \quad \quad | \quad \text{interchange} (\langle \text{id} \rangle, \langle \text{int} \rangle, \langle \text{int} \rangle) \\
& \quad \quad | \quad \text{fuse} (\langle \text{id} \rangle, \langle \text{id} \rangle, \langle \text{int} \rangle) \\
& \quad \quad | \quad \text{unroll} (\langle \text{id} \rangle, \langle \text{int} \rangle) \\
\langle \text{iters} \rangle & \quad ::= \quad [\langle \text{ids} \rangle] \\
\langle \text{ids} \rangle & \quad ::= \quad \langle \text{id} \rangle (, \langle \text{id} \rangle)^* \\
\langle \text{ints} \rangle & \quad ::= \quad \langle \text{int} \rangle (, \langle \text{int} \rangle)^* 
\end{align*}
\]
TeML overview
Raising the level of abstraction

A contraction chain

\[ v_{ijk} = \sum_{l,m,n} A_{kn} \cdot A_{jm} \cdot A_{il} \cdot u_{lmn} \]

Control the evaluation order

\[ v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot (A_{jm} \cdot (A_{il} \cdot u_{lmn}))) \]

\[ v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot A_{jm}) \cdot (A_{il} \cdot u_{lmn}) \]

\[ v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot ((A_{jm} \cdot A_{il}) \cdot u_{lmn})) \]

- The evaluation order may dramatically impact execution time
- May be combined with other transformation heuristics
Tensor-algebraic transformations are essential for some applications:

- Out of the scope of polyhedral-based meta-languages.
- Or requires additional analyses to (re)discover algebraic tensor properties.

```
# -- Begin program specification
w = tensor(double, [13])
u = tensor(double, [13, 13, 13])
L = tensor(double, [13, 13])
M_ = outerproduct([w, w, w])
Lh = div(L, w, [[i1, i2], [i2]]) ->
    [i1, i2])
M = entrywise_mul(M_, u)
r1 = contract(Lh, M, [[2, 1]])
r2 = contract(Lh, M, [[2, 2]])
r3 = contract(Lh, M, [[2, 3]])
# -- End program specification
```

- We want such characterizations to be native to the language.
- Provides room for encoding algebraic properties.
TeML overview
By example: facilitating transformation composition

- Existing meta-languages are either fully imperative or mix a functional specification of the computation with an imperative transformation sequence
- We use a functional style for both program stages

```plaintext
# -- Begin program specification
w = tensor(double, [13])
u = tensor(double, [13, 13, 13])
L = tensor(double, [13, 13])
M_ = outerproduct([w, w, w])
Lh = div(L, w, [[i1, i2], [i2]] ->
    [i1, i2])
M = entrywise_mul(M_, u)
r1 = contract(Lh, M, [[2, 1]])
r2 = contract(Lh, M, [[2, 2]])
r3 = contract(Lh, M, [[2, 3]])
# -- End program specification

# Code generation without transformations
codegen([l1, l2, l3, l4, l5, l6])
17 = fuse(14, 15, 3)
18 = fuse(17, 16, 3)
# Code generation with loop fusions only
codegen([l1, l2, l3, l7])
19 = parallelize(11, 1, None)
l10 = parallelize(12, 1, None)
l11 = parallelize(13, 1, None)
l12 = parallelize(18, 1, None)
l13 = vectorize(l9, 3)
l14 = vectorize(l10, 2)
l15 = vectorize(l11, 3)
# Code generation with fusion, parallelism and vectorization
codegen([l13, l14, l15, l12])
```
Denotational semantics

Domains of trees for tensors ($T$) and loops ($L$)

State

- A state in a TeML meta-program maps identifiers to trees representing either tensors or loops

$$S = \text{identifier} \rightarrow (T + L)$$

$$\sigma : \text{identifier} \rightarrow (T + L)$$

Valuation functions

- Different manipulations of a state $\sigma$ for each syntactic entity

$$P_{\text{prog}} : \text{program} \rightarrow (S \rightarrow S)$$

$$P_{\text{stmt}} : \text{stmt} \rightarrow (S \rightarrow S)$$

$$E_t : \text{Texpression} \rightarrow (S \rightarrow T)$$

$$E_l : \text{Lexpression} \rightarrow (S \rightarrow L)$$
for (int i1 = 0; i1 < (N-1); i1++)
    for (int i2 = 0; i2 < (N-1); i2++)
        E[i1][i2] = C[i1][i2] * (A[i1][i2] + B[i1][i2]);

A = tensor([N, N])
B = tensor([N, N])
C = tensor([N, N])
D = vadd(A, B, [[i1, i2], [i1, i2]])
E = mul(C, D, [[i1, i2], ] -> [i1, i2])

- We use virtual operators (vops) to compose beyond 3-address expressions
- Tensors returned by vops only hold subexpressions eventually expanded recursively at instances of ops
- Tensors returned by vops do not have shapes of their own
- Others have their shape inferred, as well as their loop domains
Semantics of tensor expressions

Low-level operations

Essential informations to capture

- Shape
- Expression tree
- Associated list of iterators

\[ A = \text{tensor}([N, N]) \]
\[ B = \text{tensor}([N, N]) \]
\[ C = \text{tensor}([N, N]) \]
\[ D = \text{vadd}(A, B, [[i1, i2], [i1, i2]]) \]
\[ E = \text{mul}(C, D, [[i1, i2], ] \rightarrow [i1, i2]) \]

\[ \sigma_1 = \mathcal{P}_{stmt}[A = \text{tensor}([N, N])] \emptyset \]
\[ = \{ A \mapsto \langle (A, [N, N], \epsilon), [] \rangle \} \]

\[ \sigma_2 = \mathcal{P}_{stmt}[B = \text{tensor}([N, N])] \sigma_1 \]
\[ = \{ A \mapsto \langle (A, [N, N], \epsilon), [] \rangle, \ B \mapsto \langle (B, [N, N], \epsilon), [] \rangle \} \}

\[ \sigma_3 = \cdots \]
Semantics of tensor expressions
High-level operations

The example of tensor contraction

$$P_{stmt}[t' = \text{contract}(t_0, t_1, [r_0, r_1])] =$$

$$P_{prog} \begin{bmatrix}
    t_2 = \text{vmul}(t_0, t_1, [I, J]) \\
    t' = \text{add}(t', t_2, [I', \epsilon] \rightarrow I')
\end{bmatrix}$$

where

$$I = [i_0, \ldots, i(r_0 - 1), k, i(r_0 + 1), \ldots, i s_0],$$

$$J = [j_0, \ldots, j(r_1 - 1), k, j(r_1 + 1), \ldots, j s_1],$$

$$I' = (I \setminus \{k\}) \parallel (J \setminus \{k\}).$$
Semantics of loop transformations

- Principles of loop transformations are quite well understood.
- The polyhedral model is a rich formalism to abstracts the effects of loop transformations
- The idea here is to formalize such principles in a meta-language context

Example

```c
for (int i1 = 0; i1 <= (N-1); i1++) {
    C[i1] = A[i1] - B[i1]; // tC
    for (int i2 = 0; i2 <= (N-1); i2++) {
        E[i1][i2] = D[i2] * C[i1]; // tE
        F[i1][i2] = E[i1][i2]; // tF
    }
    for (int i3 = 0; i3 <= (N-1); i3++) {
        G[i1] = G[i1] + F[i1][i3] // tG
    }
}
```
Loop creation from tensor expressions

- The semantics of build

\[ A = \text{tensor}([N, N]) \]
\[ B = \text{tensor}([N, N]) \]
\[ C = \text{tensor}([N, N]) \]
\[ D = \text{vadd}(A, B, [[i1, i2], [i1, i2]]) \]
\[ E = \text{mul}(C, D, [[i1, i2], ] \rightarrow [i1, i2]) \]

\[ \mathcal{E}_l[^{\text{build}}(E)]^5 \sigma_5 = \langle i1, [\langle i2, [\sigma_5(E)]\rangle]\rangle : \]
\[ \text{for } (i1 = 0; i1 <= (N-1); i1++) \]
\[ \text{for } (i2 = 0; i2 <= (N-1); i2++) \]
\[ E[i1][i2] = C[i1][i2] \times (A[i1][i2] + B[i1][i2]); \]
Semantics of loop expressions

Stripmining

- Divides an iteration space into smaller blocks

A = tensor([N, N])
B = tensor([N, N])
C = tensor([N, N])
D = vadd(A, B, [[[i1, i2], [i1, i2]]])
E = mul(C, D, [[[i1, i2], ] -> [i1, i2]])
L = build(E)
S = stripmine(L, 1, 32)

\[ \sigma_n = P_{stmt}[L = \text{build}(E)] \sigma_{n-1} \]
\[ = \{ L \mapsto \langle i1, [\langle i2, [\sigma_{n-1}(E)]]) \rangle \} \]

\[ \sigma_{n+1} = P_{stmt}[S = \text{stripmine}(L, 1, 32)] \sigma_n \]
\[ = \{ L \mapsto \langle i1, [\langle i2, [\sigma_n(E)]]) \rangle, S \mapsto \langle t1, [\langle i1, [\langle i2, [\sigma_n(E)]])]) \rangle \} \]

\[ E \llbracket \text{stripmine}(L, 1, 32) \rrbracket \sigma_n = \langle t1, [\langle i1, [\langle i2, [\sigma_n(E)]])]) \rangle : \]

for (int t1 = 0; t1 <= (N-1)/32; t1++)
for (int i1 = 32 * t1; i1 <= min((N-1), 32 * t1 + 31); i1++)
for (int i2 = 0; i2 <= (N-1); i2++)
    E[i1][i2] = C[i1][i2] * (A[i1][i2] + B[i1][i2]);
Semantics of loop expressions

Interchange

- Swaps dimensions of a loop nest

\[
\begin{align*}
A &= \text{tensor}([N, N]) \\
B &= \text{tensor}([N, N]) \\
C &= \text{tensor}([N, N]) \\
D &= \text{vadd}(A, B, [[i1, i2], [i1, i2]]) \\
E &= \text{mul}(C, D, [[i1, i2], ] \rightarrow [i1, i2]) \\
L &= \text{build}(E) \\
I &= \text{interchange}(L, [1, 2])
\end{align*}
\]

\[
\begin{align*}
\sigma_n &= \mathcal{P}_{stmt}[L = \text{build}(E)] \sigma_{n-1} \\
&= \{ L \mapsto (i1, [i2, [\sigma_{n-1}(E)]]) \} \\
\sigma_{n+1} &= \mathcal{P}_{stmt}[I = = \text{interchange}(L,[1, 2])] \sigma_n \\
&= \{ L \mapsto (i1, [i2, [\sigma_n(E)]]) \}, I \mapsto (i2, [i1, [\sigma_n(E)]])) \}
\end{align*}
\]

\[
\begin{align*}
\mathcal{E}_l[\text{interchange}(L,[1, 2])] \sigma_n &= (i2, [i1, [\sigma_n(E)]])):
\end{align*}
\]

for (int i2 = 0; i2 <= (N-1); i2++)
for (int i1 = 0; i1 <= (N-1); i1++)
    E[i1][i2] = C[i1][i2] * (A[i1][i2] + B[i1][i2]);
Semantics of loop expressions
Loop tiling in denotational semantics

- Loop tiling is the composition of stripmining and interchange

```c
for (int t1 = 0; t1 <= (N-1)/32; t1++)
    for (int t2 = 0; t2 <= (N-1)/32; t2++)
        for (int i1 = 32 * t1; i1 <= min((N-1), 32 * t1 + 31); i1++)
            for (int i2 = 32 * t2; i2 <= min((N-1), 32 * t2 + 31); i2++)
                E[i1][i2] = C[i1][i2] * (A[i1][i2] + B[i1][i2]);
```

\[
P_{stmt}[l' = tile(l, v)] =
\begin{cases}
    l_0 = \text{stripmine}_n(l, d, v) \\
    l_1 = \text{interchange}_n(l_0, 2, 2d - 2) \\
    l_2 = \text{interchange}_n(l_1, 3, 2d - 3) \\
    \cdots \\
    l_{d-1} = \text{interchange}_n(l_{d-2}, d, d)) \\
    l' = \text{interchange}_n(l_{d-1}, d + 1, d - 1)
\end{cases}
\]
Semantics of loop expressions
Loop tiling in denotational semantics

Initial loop nest

\[ i_1 \rightarrow i_3 \rightarrow i_5 \rightarrow xs \]

\textit{stripmine}_n(\_, 3, v) \textbf{has introduced} \( i'_2, i'_4, \text{and} \ i'_6 \)

\[ i'_1 \rightarrow i'_2 \rightarrow i'_3 \rightarrow i'_4 \rightarrow i'_5 \rightarrow i'_6 \rightarrow xs \]

After triple application of \textit{interchange}_n

\[ i'_1 \rightarrow i'_3 \rightarrow i'_5 \rightarrow i'_2 \rightarrow i'_4 \rightarrow i'_6 \rightarrow xs \]
**TeML evaluation**

Expressing tensor computations in comparison to TensorFlow

Application domains: Linear Algebra (LA), Deep Learning (DL), Machine Learning (ML), Data Analytics (DA), Fluid Dynamics (FD), Image Processing (IP).

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
<th>LOC</th>
<th>TensorFlow Constructs used</th>
<th>LOC</th>
<th>TeML Constructs used</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Matrix Multiplication</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmm</td>
<td>LA</td>
<td>3</td>
<td>matmul</td>
<td>3</td>
<td>contract</td>
</tr>
<tr>
<td>tmm</td>
<td>DL</td>
<td>3</td>
<td>matmul:transpose=True</td>
<td>4</td>
<td>transpose, contract</td>
</tr>
<tr>
<td>bmm</td>
<td></td>
<td>3</td>
<td>einsum</td>
<td>3</td>
<td>mul, add</td>
</tr>
<tr>
<td><strong>Grouped Convolutions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gconv</td>
<td>N/A</td>
<td>5</td>
<td>Not implemented</td>
<td>5</td>
<td>vmul, add</td>
</tr>
<tr>
<td>mttkrp</td>
<td>DA</td>
<td>4</td>
<td>einsum or tensordot, multiply</td>
<td>5</td>
<td>vcontract, contract</td>
</tr>
<tr>
<td>sddmm</td>
<td>ML</td>
<td>4</td>
<td>einsum or tensordot, multiply</td>
<td>5</td>
<td>vcontract, entrywise_mul</td>
</tr>
<tr>
<td>interp</td>
<td>FD</td>
<td>3</td>
<td>einsum or tensordot</td>
<td>5</td>
<td>contract</td>
</tr>
<tr>
<td>helm</td>
<td>N/A</td>
<td>9</td>
<td>Required division not well supported</td>
<td>9</td>
<td>contract, outerproduct, div, entrywise_mul</td>
</tr>
<tr>
<td>blur</td>
<td>IP</td>
<td>N/A</td>
<td>No stencil support.</td>
<td>9</td>
<td>op, vop</td>
</tr>
<tr>
<td>coars</td>
<td></td>
<td>6</td>
<td>einsum or multiply, subtract</td>
<td>6</td>
<td>ventrywise_mul, entrywise_sub</td>
</tr>
</tbody>
</table>
TeML evaluation
Reproducing optimization paths of Pluto

Pluto

- Polyhedral automatic parallelizer
- Some flexibility in selecting optimizations and their parameters
- But quite rigid heuristics, mostly “black-box” optimizations

<table>
<thead>
<tr>
<th>mttkrp (250<em>250</em>250)</th>
<th>sddmm (4096*4096)</th>
<th>bmm (8192<em>72</em>26)</th>
<th>gconv (32<em>32</em>32<em>32</em>7*7)</th>
<th>interp (50000<em>7</em>7*7)</th>
<th>helm (5000<em>13</em>13*13)</th>
<th>coars (4096*4096)</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallelize(l, 1)</td>
<td>interchange(l, 2, 3)</td>
<td>tile(l, 32)</td>
<td>interchange(l1, 4, 5)</td>
<td>interchange(l1, 1, 5)</td>
<td>fuse_outer(l4, l5, 5)</td>
<td>tile(l, 32)</td>
</tr>
<tr>
<td>interchange(l, 1)</td>
<td>parallelize(l, 1)</td>
<td>interchange(l, 7,8)</td>
<td>vectorize(l1, 1)</td>
<td>vectorize(l2, 5)</td>
<td>parallelize(l1, 1)</td>
<td>parallelize(l, 1)</td>
</tr>
<tr>
<td>vectorize(l, 3)</td>
<td></td>
<td>parallelize(l, 1)</td>
<td>vectorize(l1, 9)</td>
<td>parallelize(l2, 1)</td>
<td>parallelize(l1, 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>vectorize(l, 8)</td>
<td>vectorize(l2, 9)</td>
<td>parallelize(l3, 1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

> Can we outperform Pluto?
TeML evaluation
Expressing transformations that outperform Pluto

- On Intel(R) Core(TM) i7-4910MQ CPU (2.90GHz, 8 hyperthreads, 8192KB of shared L3 cache), Ubuntu 16.04
- Generated C programs compiled with the Intel C compiler ICC 18.02 (flags: -O3 -xHost -qopenmp)
- TensorFlow version 1.6 with support for AVX, FMA, SSE, and multi-threading

![Graphs showing performance comparison between TensorFlow, Pluto, and TeML for mttkrp, interp, and helm tasks across different core counts.](chart.png)
We are able to express more efficient transformation paths
Conclusion

TeML

- Program construction and transformation phases are both functional
- Higher-level of abstractions for tensor computations
- Formal specification of program construction and transformation

Future work

- Extensions for parallelism support
- Abstractions for memory virtualization and corresponding semantics
- Type system
- High-level abstractions for stencil patterns, general convolutions, sparse tensors