Meta-programming for Cross-Domain Tensor Optimizations

Adilla Susungi[1], Norman A. Rink[2], Albert Cohen[3], Jerónimo Castrillón[2], Claude Tadonki[1]

[1]MINES ParisTech, PSL Research University
[2]Chair for Compiler Construction, Technische Universität Dresden
[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

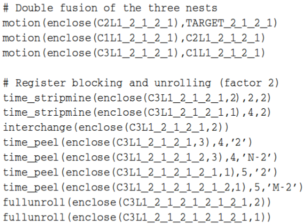
- Fundamental algebraic structure with applications to many domains
- Operations on multi-dimensional and computationally intense loop nests
- Involves multiple optimization strategies: loop, data layout, algebraic transformations, mapping decisions, etc.

Existing optimizing frameworks

- Built-in strategies do not always generalize well
- Lack of flexibility in composing finely tuned, target-specific optimizations

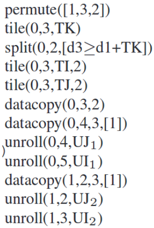
Transformation meta-languages

Meta-languages offering transformation heuristics as first-class citizens



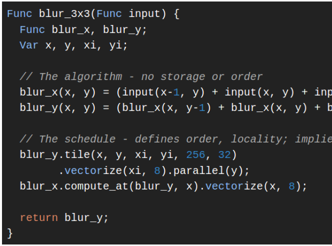
```
# Double fusion of the three nests
motion(enclose(C2L1_2_1_2_1),TARGET_2_1_2_1)
motion(enclose(C1L1_2_1_2_1),C2L1_2_1_2_1)
motion(enclose(C3L1_2_1_2_1),C1L1_2_1_2_1)

# Register blocking and unrolling (factor 2)
time_stripmine(enclose(C3L1_2_1_2_1,2),2,2)
time_stripmine(enclose(C3L1_2_1_2_1,1),4,2)
interchange(enclose(C3L1_2_1_2_1,2))
time_peel(enclose(C3L1_2_1_2_1,3),4,'2')
time_peel(enclose(C3L1_2_1_2_1_2,3),4,'N-2')
time_peel(enclose(C3L1_2_1_2_1_2_1,1),5,'2')
time_peel(enclose(C3L1_2_1_2_1_2_1_2,1),5,'M-2')
fullunroll(enclose(C3L1_2_1_2_1_2_1_2_1,2))
fullunroll(enclose(C3L1_2_1_2_1_2_1_2_1,1))
```

URUK (Cohen et al., ICS'05)



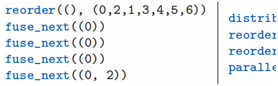
```
permute([1,3,2])
tile(0,3,TK)
split(0,2,[d3≥d1+TK])
tile(0,3,TI,2)
tile(0,3,TJ,2)
datacopy(0,3,2)
datacopy(0,4,3,[1])
unroll(0,4,UJ_1)
unroll(0,5,UI_1)
datacopy(1,2,3,[1])
unroll(1,2,UJ_2)
unroll(1,3,UI_2)
```

CHiLL (Chen et al., 2008)



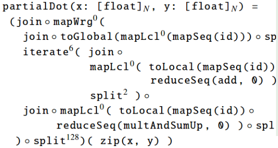
```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // The algorithm - no storage or order
  blur_x(x, y) = (input(x-1, y) + input(x, y) + inp
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + b

  // The schedule - defines order, locality; implie
  blur_y.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);

  return blur_y;
}
```

Halide (Ragan-Kelley et al., PLDI'14)



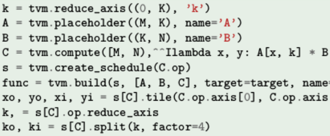
```
reorder((), (0,2,1,3,4,5,6))      distrit
fuse_next((0))                    reorder
fuse_next((0))                    reorder
fuse_next((0))                    paralle
fuse_next((0, 2))
```

Clay (Bagnères et al., CGO'16)



```
partialDot(x: [float]_N, y: [float]_N) =
 (join ∘ mapWrg^0(
   join ∘ toGlobal(mapLcl^0(mapSeq(id))) ∘ sp
   iterate^6( join ∘
                mapLcl^0( toLocal(mapSeq(id))
                          reduceSeq(add, 0) )
                split^2 ) ∘
   join ∘ mapLcl^0( toLocal(mapSeq(id)) ∘
          reduceSeq(multAndSumUp, 0) ) ∘ spl
 ) ∘ split^128)( zip(x, y) )
```

Lift (Steuwer et al., CGO'17)



```
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
C = tvm.compute([M, N),^^Ilambda x, y: A[x, k] * B
s = tvm.create_schedule(C.op)
func = tvm.build(s, [A, B, C], target=target, name=
xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis
k, = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=4)
```

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

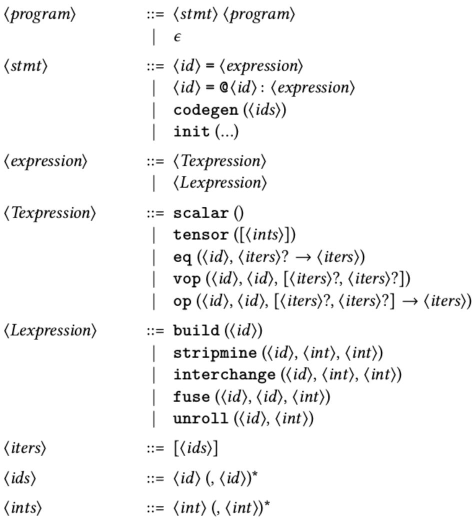
Contributions outline

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



```
⟨program⟩       ::= ⟨stmt⟩ ⟨program⟩
                  | ϵ
⟨stmt⟩          ::= ⟨id⟩ = ⟨expression⟩
                  | ⟨id⟩ = @⟨id⟩ : ⟨expression⟩
                  | codegen (⟨ids⟩)
                  | init (...)
⟨expression⟩    ::= ⟨Texpression⟩
                  | ⟨Lexpression⟩
⟨Texpression⟩   ::= scalar ()
                  | tensor ([⟨ints⟩])
                  | eq (⟨id⟩, ⟨iters⟩? → ⟨iters⟩)
                  | vop (⟨id⟩, ⟨id⟩, [⟨iters⟩?, ⟨iters⟩?])
                  | op (⟨id⟩, ⟨id⟩, [⟨iters⟩?, ⟨iters⟩?] → ⟨iters⟩)
⟨Lexpression⟩   ::= build (⟨id⟩)
                  | stripmine (⟨id⟩, ⟨int⟩, ⟨int⟩)
                  | interchange (⟨id⟩, ⟨int⟩, ⟨int⟩)
                  | fuse (⟨id⟩, ⟨id⟩, ⟨int⟩)
                  | unroll (⟨id⟩, ⟨int⟩)
⟨iters⟩         ::= [⟨ids⟩]
⟨ids⟩           ::= ⟨id⟩ (, ⟨id⟩)*
⟨ints⟩          ::= ⟨int⟩ (, ⟨int⟩)*
```

Every function returns either

- Tensors
- Loops

Operations on tensors

- Computation specification
- Layout transformations
- Data initialization, mapping

Operations on loops

- Expansion from tensor computation
- Transformation

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

TeML overview

Raising the level of abstraction

Tensor-algebraic transformations are essential 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

```
  # -- 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

  # Generate loops
  l1 = build(M_)
  l2 = build(Lh)
  l3 = build(M)
  l4 = build(r1)
  l5 = build(r2)
  l6 = build(r3)
```

```
  # Code generation without
↪   transformations
  codegen([l1, l2, l3, l4, l5, l6])

  l7 = fuse(l4, l5, 3)
  l8 = fuse(l7, l6, 3)
  # Code generation with loop fusions
↪   only
  codegen([l1, l2, l3, l7])

  l9 = parallelize(l1, 1, None)
  l10 = parallelize(l2, 1, None)
  l11 = parallelize(l3, 1, None)
  l12 = parallelize(l8, 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 ($\mathbf{T}$) and loops ($\mathbf{L}$)

State

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

$$\begin{aligned} \mathbf{S} &= identifier \rightarrow (\mathbf{T} + \mathbf{L}) \\ \sigma &: identifier \rightarrow (\mathbf{T} + \mathbf{L}) \end{aligned}$$

Valuation functions

- Different manipulations of a state σ for each syntactic entity

$$\begin{aligned} &\boldsymbol{\mathcal{P}}_{prog} : program \rightarrow (\mathbf{S} \rightarrow \mathbf{S}) \\ &\boldsymbol{\mathcal{P}}_{stmt} : stmt \rightarrow (\mathbf{S} \rightarrow \mathbf{S}) \\ &\boldsymbol{\mathcal{E}}_{t} : Texpression \rightarrow (\mathbf{S} \rightarrow \mathbf{T}) \\ &\boldsymbol{\mathcal{E}}_{l} : Lexpression \rightarrow (\mathbf{S} \rightarrow \mathbf{L}) \end{aligned}$$

Semantics of tensor expressions

Subtleties

```
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])
```

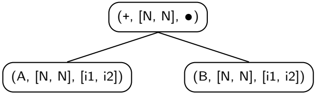
- We use virtual operators (`vops`) to compose beyond 3-address expressions
- Tensors returned by `vops` only hold subexpressions eventually expanded recursively at instances of ops
- Tensors returned by `vops` do not have shapes of their own
- Others have their shape inferred, as well as their loop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

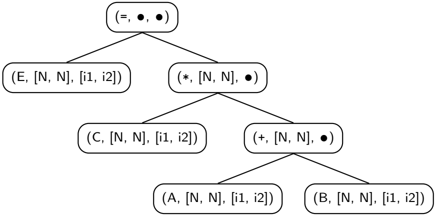
Semantics of tensor expressions

Low-level operations

Essential informations to capture

- Shape
- Expression tree
- Associated list of iterators





```
A = 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])
```

$$\begin{aligned}\sigma_1 &= \boldsymbol{\mathcal{P}}_{\textit{stmt}}[\![A\ \texttt{=}\ \texttt{tensor}([N, N])]\!]\emptyset \\ &= \{\ A \mapsto \langle (A, [N, N], \epsilon), [] \rangle\ \}\end{aligned}$$

$$\begin{aligned}\sigma_2 &= \boldsymbol{\mathcal{P}}_{\textit{stmt}}[\![B\ \texttt{=}\ \texttt{tensor}([N, N])]\!]\sigma_1 \\ &= \{\ A \mapsto \langle (A, [N, N], \epsilon), [] \rangle\ ,\ \ B \mapsto \langle (B, [N, N], \epsilon), [] \rangle\ \}\end{aligned}$$

$$\sigma_3 = \cdots$$

Semantics of tensor expressions

High-level operations

The example of tensor contraction

$$\boldsymbol{\mathcal{P}}_{\textit{stmt}}[\![t' \texttt{ = } \texttt{contract}(t_0, t_1, [r_0, r_1])]\!] =$$

$$\boldsymbol{\mathcal{P}}_{\textit{prog}} \left[\!\!\left[\begin{array}{l} t_2 \texttt{ = } \texttt{vmul}(t_0, t_1, [I, J]) \\ t' \texttt{ = } \texttt{add}(t', t_2, [I', \epsilon] \to I') \end{array} \right]\!\!\right]$$

where

$$\begin{aligned} I &= [\texttt{i0}, \ldots, \texttt{i}(r_0 - 1), \texttt{k}, \texttt{i}(r_0 + 1), \ldots, \texttt{i}s_0]\,, \\ J &= [\texttt{j0}, \ldots, \texttt{j}(r_1 - 1), \texttt{k}, \texttt{j}(r_1 + 1), \ldots, \texttt{j}s_1]\,, \\ I' &= (I \backslash \{\texttt{k}\}) \mathbin{||} (J \backslash \{\texttt{k}\})\,. \end{aligned}$$

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

```
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
 }
}
```

$$
\begin{aligned}
&\langle \mathtt{i1}, [\mathtt{tC}, \\
&\qquad \langle i2, [\mathtt{tE}, \mathtt{tF}]\rangle, \\
&\qquad \langle i3, [\mathtt{tG}]\rangle \\
&\qquad]\rangle
\end{aligned}
$$

Loop creation from tensor expressions

- The semantics of `build`

```
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])
```

$$\underline{\mathcal{E}_l[\![\mathtt{build}(E)]\!]\sigma_5 = \langle \mathtt{i1}, [\langle \mathtt{i2}, [\sigma_5(E)]\rangle]\rangle\ :}$$

```
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]);
```

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)
```

$$\begin{aligned}\sigma_n &= \mathcal{P}_{\mathit{stmt}}[\![L = \texttt{build}(E)]\!]\sigma_{n-1} \\ &= \{\, L \mapsto \langle \texttt{i1}, [\langle \texttt{i2}, [\sigma_{n-1}(E)]\rangle]\rangle \,\}\end{aligned}$$

$$\begin{aligned}\sigma_{n+1} &= \mathcal{P}_{\mathit{stmt}}[\![S = \texttt{stripmine}(L, 1, 32)]\!]\sigma_n \\ &= \{\, L \mapsto \langle \texttt{i1}, [\langle \texttt{i2}, [\sigma_n(E)]\rangle]\rangle\ ,\ \ S \mapsto \langle \texttt{t1}, [\langle \texttt{i1}, [\langle \texttt{i2}, [\sigma_n(E)]\rangle]\rangle]\rangle \,\}\end{aligned}$$

$$\underline{\mathcal{E}_l[\![\texttt{stripmine}(L, 1, 32)]\!]\sigma_n = \langle \texttt{t1}, [\langle \texttt{i1}, [\langle \texttt{i2}, [\sigma_n(E)]\rangle]\rangle]\rangle\ :}$$

```
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

```
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)
I = interchange(L, [1, 2])
```

$$\begin{aligned}\sigma_n &= \mathcal{P}_{\mathit{stmt}}[\![L = \mathtt{build}(E)]\!]\sigma_{n-1}\\ &= \{\, L \mapsto \langle \mathtt{i1}, [\langle \mathtt{i2}, [\sigma_{n-1}(E)]\rangle]\rangle \,\}\end{aligned}$$

$$\begin{aligned}\sigma_{n+1} &= \mathcal{P}_{\mathit{stmt}}[\![I = \ = \mathtt{interchange}(L, [1, 2])]\!]\sigma_n\\ &= \{\, L \mapsto \langle \mathtt{i1}, [\langle \mathtt{i2}, [\sigma_n(E)]\rangle]\rangle\ ,\ \ I \mapsto \langle \mathtt{i2}, [\langle \mathtt{i1}, [\sigma_n(E)]\rangle]\rangle \,\}\end{aligned}$$

$$\underline{\mathcal{E}_l[\![\mathtt{interchange}(L, [1, 2])]\!]\sigma_n = \langle \mathtt{i2}, [\langle \mathtt{i1}, [\sigma_n(E)]\rangle]\rangle\ :}$$

```
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

```
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]);
```

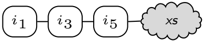
$$\mathcal{P}_{stmt}[\![l' = \texttt{tile}(l, v)]\!] =$$

$$\mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} l_0 = \texttt{stripmine\_n}(l, d, v) \\ l_1 = \texttt{interchange\_n}(l_0, 2, 2d-2) \\ l_2 = \texttt{interchange\_n}(l_1, 3, 2d-3) \\ \ldots \\ l_{d-1} = \texttt{interchange\_n}(l_{d-2}, d, d)) \\ l' = \texttt{interchange\_n}(l_{d-1}, d+1, d-1) \end{array}\right]\!\!\right]$$

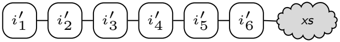
Semantics of loop expressions

Loop tiling in denotational semantics

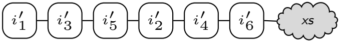
Initial loop nest



`stripmine_n`$(\_, 3, v)$ **has introduced** i'_2**,** i'_4**, and** i'_6



After triple application of `interchange_n`



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).

	Name	**Domain**	**TensorFlow**		**TeML**	
			LOC	Constructs used	LOC	Constructs used
Matrix Multiplication	*mm*	LA	3	matmul	3	contract
transposed	*tmm*	DL	3	matmul:transpose=True	4	transpose, contract
batched	*bmm*		3	einsum	3	mul, add
Grouped Convolutions	*gconv*		N/A	Not implemented Incompatible with einsum	5	vmul, add
Matricized Tensor Times Khatri-Rao product	*mttkrp*	DA	4	einsum or tensordot, multiply	5	vcontract, contract
Sampled Dense-Dense Matrix Product	*sddmm*	ML	4	einsum or tensordot, multiply	6	vcontract, entrywise_mul
Interpolation	*interp*	FD	3	einsum or tensordot	5	contract
Helmholtz	*helm*		N/A	Required division not well supported	9	contract, outerproduct, div, entrywise_mul
Blur	*blur*	IP	N/A	No stencil support.	9	op, vop
Coarsity	*coars*		6	einsum or multiply, subtract	6	ventrywise_mul, entrywise_sub

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

*mttkrp (250*250*250)*	*sddmm (4096*4096)*	*bmm (8192*72*26)*	*gconv (32*32*32*32*7*7)*	*interp (50000*7*7*7)*	*helm (5000*13*13*13)*	*coars (4096*4096)*
parallelize(l, 1) interchange(l, 2, 3)	interchange(l, 2, 3), parallelize(l, 1), vectorize(l, 3)	tile(l, 32) interchange(l, 7,8) paralellize(l, 1) vectorize(l, 8)	interchange(l1, 4, 5) interchange(l1, 5, 6) parallelize(l1, 1) vectorize(l1, 9) paralellize(l2, 1) vectorize(l2, 9)	interchange(l1, 4, 5), vectorize(l1, 5), interchange(l2, 4, 5), vectorize(l2, 5), parallelize(l1, 1), parallelize(l2,1), parallelize(l3, 1)	fuse_outer(l4, l5, 5), fuse_outer(l4, l6, 5), parallelize(l1, 1), parallelize(l2, 1), parallelize(l3, 1), parallelize(l4, 1), vectorize(l1, 2), vectorize(l2, 3), vectorize(l3, 4)	tile(l, 32) parallelize(l, 1) vectorize(l, 4)

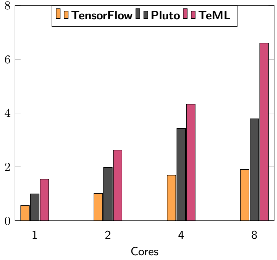
- Can we outperform Pluto?

TeML evaluation

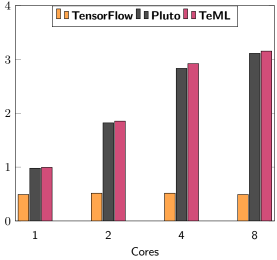
Expressing transformations that outperform Pluto

- 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

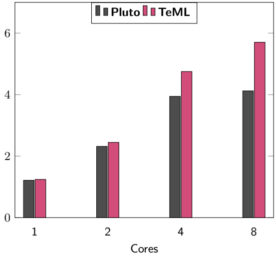
mttkrp



interp

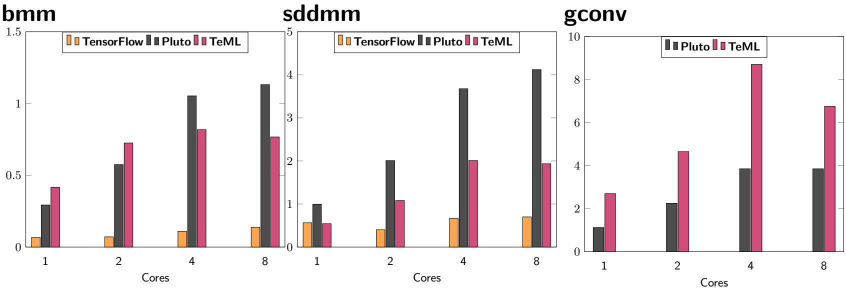


helm



TeML evaluation

Expressing transformations that outperform Pluto



- 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