The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically

NICOLAS VASILACHE, Facebook AI Research, NY, USA
OLEKSANDR ZINENKO, Inria and ENS, France
THEODOROS THEODORIDIS, ETH Zürich, Switzerland
PRIYA GOYAL, Facebook AI Research, NY, USA
ZACHARY DEVITO, Facebook AI Research, CA, USA
WILLIAM S. MOSES, MIT CSAIL, MA, USA
SVEN VERDOOLAEGE, Polly Labs & Facebook AI Research, Belgium
ANDREW ADAMS, Facebook AI Research, CA, USA
ALBERT COHEN, Inria, ENS and Facebook AI Research, France

Deep learning frameworks automate the deployment, distribution, synchronization, memory allocation, and hardware acceleration of models represented as graphs of computational operators. These operators wrap high-performance libraries such as cuDNN or NNPACK. When the computation does not match any predefined library call, custom operators must be implemented, often at high engineering cost and performance penalty, limiting the pace of innovation. To address this productivity gap, we propose and evaluate: (1) a domain-specific language with a tensor notation close to the mathematics of deep learning; (2) a Just-In-Time optimizing compiler based on the polyhedral framework; (3) carefully coordinated linear optimization and evolutionary algorithms to synthesize high-performance CUDA kernels; (4) the transparent integration of our flow into PyTorch and Caffe2, providing the fully automatic synthesis of high-performance GPU kernels from simple tensor algebra. The performance is comparable to, and often exceeds the performance of, highly tuned libraries.

CCS Concepts: • **Software and its engineering** → *Compilers*;

Additional Key Words and Phrases: Deep learning layers, polyhedral compilation, GPU acceleration

N. Vasilache, O. Zinenko, and A. Cohen are with Google AI at the time of publication.
S. Verdoolaege is with Cerebras at the time of publication.
A. Adams is with Adobe at the time of publication.
This work was partly supported by a grant from Facebook to ETH Zürich and by the European Commission through the MNEMOSENE project ID 780215. It would not have been possible without the long-term committment of ARM funding much of the development of isl over the past four years in the context of the Polly Labs initiative of ARM and Inria in collaboration with ETH Zürich.
Authors' addresses: N. Vasilache, Facebook AI Research, New York City, NY, USA; email: nicolas.vasilache@gmail.com; O. Zinenko, Inria and ENS, Paris, France; email: oleksandr.zinenko@inria.fr; T. Theodoridis, ETH Zürich, Zürich, Switzerland; email: theodort@student.ethz.ch; P. Goyal, Facebook AI Research, New York City, NY, USA; email: prigoyal@fb.com; Z. DeVito, Facebook AI Research, Menlo Park, CA, USA; email: zdevito@fb.com; W. S. Moses, MIT CSAIL, Cambridge, MA, USA; email: wmoses@mit.edu; S. Verdoolaege, Polly Labs & Facebook AI Research, Leuven, Belgium; email: skimo@kotnet.org; A. Adams, Facebook AI Research, Menlo Park, CA, USA; email: andrew.b.adams@gmail.com ; A. Cohen, Inria, ENS and Facebook AI Research, Paris, France; email: albert.cohen@inria.fr.

1544-3566/2019/10-ART38
https://doi.org/10.1145/3355606

ACM Reference format:
Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4, Article 38 (October 2019), 26 pages.
https://doi.org/10.1145/3355606

1 INTRODUCTION

Deep neural networks trained with back-propagation learning [41] are a method of choice to solve complex problems with sufficient data. Popular graph computation engines [1, 16, 19, 48, 61] offer high-level abstractions for optimizing and executing deep neural networks expressed as graphs of tensor operations. These frameworks make transparent use of heterogeneous computing systems, leveraging highly optimized routines for individual operators. While these operators are sufficient for many applications, they fall short in a number of instances. Developing a novel type of layer or network architecture incurs high engineering cost or performance penalty. Even if a new layer may be expressed in terms of existing library primitives, performance is often far from peak for two reasons: missed optimizations across operators; and no tuning for its specific size, shape, and data flow [65]. Our work aims at addressing this *productivity gap*.[1]

In parallel to the software problem, a hardware race has begun, fueled by the needs for energy-efficient computing. With Google's TPU [34] and Microsoft's Brainwave project [44] on the bleeding edge, many large tech companies are pursuing their own hardware. At Google I/O 2018, Turing-award recipient John Hennessy called for fully rethinking our hardware, compilers, and language support for domain-specific properties [31], citing orders of magnitude speedup opportunities and power constraints caused by the advent of dark silicon [21].

With the increasing problem complexity and hardware limitations, growing the size of manually optimized libraries will not scale to future demands. To address these challenges, we present a novel *domain-specific flow* capable of generating highly optimized kernels for tensor expressions. It leverages optimizations across operators and takes into account the size and shape of data. The polyhedral framework of compilation emerged as a natural candidate to design a versatile optimization flow satisfying the needs of the domain and target hardware. It has demonstrated strong results in domain-specific optimization [5, 9, 20, 46], expert-driven meta-programming [6, 15, 26], embedding of third-party library code [40], and automatic generation of efficient code for heterogeneous targets [5, 7, 43, 51, 70, 77]. We attempt to take the best of both worlds, defining a domain-specific language rich enough to capture full sub-graphs of modern Machine Learning (ML) models while enabling aggressive compilation competitive to native libraries. In doing so, we may temporarily sacrifice some of the performance of über-optimized large matrix multiplications (e.g., compared to the recent Diesel polyhedral compiler [20]) while providing full automation and ML framework integration. Note that there is no fundamental difficulty in combining both approaches, recognizing and linking external library kernels when appropriate, as illustrated in Section 3.7.

Our contributions are the following:

(1) the Tensor Comprehensions (TC) Domain-Specific Language (DSL) with a tensor notation close to the mathematics of deep learning, with an emphasis on improving productivity while maintaining a direct lowering path to the intermediate representation of a parallelizing compiler for GPU acceleration;

[1] The "700 layers" is a reference to a seminal paper on programming languages by P. J. Landin: "The next 700 programming languages." *Communications of the ACM*, 9(3):157–166, 1966.

(2) an intermediate representation and Just-in-Time optimizing compiler based on the polyhedral framework, enabling complex program transformations and levels of automation unmatched by any other compiler for the acceleration of computational sub-graphs of neural networks;
(3) coordinated optimization algorithms with integrated functional correctness, profitability modeling, domain and target specialization; we propose a layered approach, relying on integer linear programming and other polyhedral algorithms to address the core program optimization and synthesis challenges, while resorting to evolutionary algorithms as a higher level of control, to select high-level strategies and fine-tune transformation parameters;
(4) the transparent integration of our flow into PyTorch [48] and Caffe2 [29], providing the fully automatic synthesis of high-performance GPU kernels from simple tensor algebra.

The TC flow is also portable to other ML frameworks with a few lines of code. While our initial implementation focuses on Nvidia GPUs, the core technology applies to other types of accelerators with shared or partitioned memory [43, 51, 70, 76]; these include vector and SIMD accelerators and also the generation of computational patterns suitable for ASICs with systolic designs and efficient storage management involving non-volatile memory technologies.

2 TENSOR COMPREHENSIONS

Tensor Comprehensions (TC) are an algorithmic notation for computing on multi-dimensional arrays. It borrows from the Einstein notation, a.k.a. summation convention: (1) index variables are defined implicitly, and their range is inferred from what they index; (2) indices that only appear on the right-hand side of a statement are assumed to be reduction dimensions; (3) the evaluation order of points in the iteration space does not affect the output.

A *tensor comprehension function* (or tensor comprehension for short) defines output tensors from *pointwise* and *reduction* operations over input tensors. These operations are defined declaratively as a sequence of pointwise equations or reductions, called *tensor comprehension statements* (or statements for short).

Let us consider matrix-vector product as a simple example of a tensor comprehension with two statements:

```
def mv(float(M,K) A, float(K) x) → (C) {
  C(i)  = 0
  C(i) += A(i,k) * x(k)
}
```

This defines the function `mv` with `A` and `x` as input tensors and `C` as an output. The shapes of `A` and `X` are of size (M, K) and (K), respectively. The shape of `C` is inferred automatically. The statements introduce two indices "`i`" and "`k`." Variables not defined in the function signature implicitly become indices. Their range is inferred based on how they are used in indexing (see Section 3.1); here, we will discover `i` $\in [0, M)$ and `k` $\in [0, K)$. Because `k` only appears on the right-hand side, stores into `C` will *reduce* over `k` with the reduction operator +.

Intuitively, a tensor comprehension may be thought of as the *body* of a loop whose control flow is inferred from context. The equivalent C-style pseudo-code is:

```
tensor C({M}).zero(); // 0-filled single-dim tensor
parallel for (int i = 0; i < M; i++)
  reduction for (int k = 0; k < K; k++)
    C(i) += A(i,k) * x(k);
```

Importantly, the nesting order (`i` then `k`) is arbitrary: The semantics of a tensor comprehension is always invariant to loop permutation.[2] TC allows in-place updates while preserving a functional

[2]Nested reductions over multiple variables are supported as long as they involve a single reduction operator, as commutation does not hold across reduction operators, e.g., $\min(\max(f(.))) \neq \max(\min(f(.)))$.

semantics that is atomic on full tensors: *RHS expressions are read in full before assigning any element on the LHS*. This specification is important in case the LHS tensor also occurs in the RHS [24]: The compiler is responsible for checking the causality of in-place updates on element-wise dependences, currently allowing only pointwise updates. Also, to enable in-place updates across TC functions, outputs of a TC statement can also be used as inputs.

We provide a short-cut for an *initializing reduction*, where the result is initialized to the operator's neutral element before reduction by appending "!" to the operator, e.g., "+=!" instead of "+=". A one-line definition of the matrix-vector product **mv** is given below; and common ML kernels can be written in just a few lines, such as the **sgemm** function from BLAS:

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

def sgemm(float a, float b, float(N,M) A, float(M,K) B) → (C) {
  C(i,j)  = b * C(i,j)           # initialization
  C(i,j) += a * A(i,k) * B(k,j)  # accumulation
}
```

Expressing general tensor contractions is equally easy. A fully connected layer followed by a rectified linear unit takes the form of a transposed matrix multiplication initialized to a broadcast bias term and followed by pointwise clamping (applying the built-in scalar function `fmaxf` with 0):

```
def fcrelu(float(B,I) in, float(O,I) weight,
           float(O) bias) → (out) {
  out(b,o)  = bias(o) where b in 0:B
  out(b,o) += in(b,i) * weight(o,i)
  out(b,o)  = fmaxf(out(b,o), 0)
}
```

The `where` annotation informs the inference algorithm of the intended index variable ranges when they cannot be unambiguously inferred. In this case, "b" indexes only "out" whose size *also* needs to be inferred. Unlike tensor kernel libraries with predefined layout conventions, notice that TC lets the user control data layout through the order of tensor indexing dimensions. Here, we chose to reuse the out tensor across all comprehensions, indicating the absence of temporary storage.

Similarly, the `where` clause serves to indicate ranges of kh and kw in the max pooling layer, which would otherwise be under-constrained:

```
def maxpool2x2(float(B,C,H,W) in) → (out) {
  out(b,c,i,j) max=! in(b,c, 2 * i + kh, 2 * j + kw) where kh in 0:2, kw in 0:2
}
```

A 2-D convolution is also simple. Its reduction is initialized to 0 (note the use of +=!) with reduction dimensions kh, kw:

```
def conv2d(float(B,IP,H,W) in, float(OP,IP,KH,KW) weight) → (out) {
  out(b,op,h,w) +=! in(b,ip, h + kh, w + kw) * weight(op,ip,kh,kw)
}
```

Subscript expressions can be any affine function of iterators, or subscript-of-subscript expressions (a tensor element indexing another), and combinations thereof. The latter capture data-dependent accesses such as a gather operation:

```
def gather(float(N) X, int(A,B) I) → (Z) { Z(i,j) = X(I(i,j)) }
```

TC algorithmic notation differs from today's prominent frameworks where most operators are defined as black-box functions. The design of TC makes it easy to experiment with small layer variations while preserving a concise, in-place expression. Thus, a strided convolution is easily created as a tweak on convolution, e.g., strided by 2 along h and 3 along w is:

```
num ::= <decimal number literal>
id ::= <C identifier>
binop ::= '+' | '-' | '*' | '/' | '=' | '≠' | ...
exp ::= num
  | ( '-' | '!' ) exp
  | exp binop exp
  | exp '?' exp ':' exp
  | id '.' num    # range of num-th dimension of id
  | id '(' exp_list ')'     # call or tensor access

reduction ::= '+='  | '*='  | 'min='  | 'max='
            | '+=!' | '*=!' | 'min=!' | 'max=!'

range_constraint ::= id '=' exp '..' exp
                   | id '=' exp

stmt ::= id '(' id_list ')' ( '=' | reduction )
         [ 'where' range_constraint_list ]
  | id_list = id '('id_list ')' # TC function call
```

```
arg ::= type id
return ::= id # inferred return type and range

scalar_type ::= 'double' | 'float' | 'half'
              | 'int' | 'byte' | 'uint32' | ...
type ::= scalar_type [ '(' id_list ')' ]

func ::= # TC function definition
  'def' id '(' arg_list ')' '⟶' '(' return_list ')' '{'
    stmt_list
  '}'

id_list ::= <comma separated id list>
exp_list ::= <comma separated exp list>
arg_list ::= <comma separated arg list>
stmt_list ::= <whitespace separated stmt list>
return_list ::= <comma separated return list>
range_constraint_list ::= <non-empty comma separated
                           range_constraint list>
```

Fig. 1. Simplified EBNF syntax for core TC. Parentheses denote inline alternatives, brackets denote optional clauses, and angle brackets contain textual descriptions used for simplicity.

```
def sconv2d(float(N,C,H,W) I, float(F,C,KH,KW) W, float(F) B) → (O) {
  O(n,f,h,w) +=! I(n,c, 2 * h + kh, 3 * w + kw) * W(f,c,kh,kw)
  O(n,f,h,w) +=  B(f)
}
```

Figure 1 shows the grammar of the Tensor Comprehension language in EBNF notation.

2.1 Data Layout

TC makes data layout explicit and easy to reason about. It supports generalized tensor transpositions (i.e., applying an n-D permutation matrix where $n > 2$), and data tiling can be achieved by reshaping tensors and adjusting the index expressions. Range inference and checking guarantees such reshaping will always be consistent throughout the statements of a tensor comprehension. For instance, *NCHW* convolution operates on an explicit input, declared as `float I(N, C, H, W)`, with the layout matching the expected row-major semantics.

In addition, the TC compiler may transparently apply layout transformations, e.g., when mapping tensor tiles to GPU shared memory.

2.2 Automatic Differentiation

TC does not natively deal with automatic differentiation, but we aim to add TC support to an existing differentiation tool in the future. DSLs like PlaidML [49] already demonstrated this.

However, backward passes can readily be implemented in TC as a few lines of code. Here is the backward pass of matrix multiplication:

```
def matmul_grad(float(M,N) A, float(N,K) B, float(M,K) d_O) → (d_A,d_B) {
  d_A(m,n) +=! d_O(m,r_k) * B(n,r_k)
  d_B(n,k) +=! d_O(r_m,k) * A(r_m,n)
}
```

3 TENSOR COMPREHENSIONS WORKFLOW

The Tensor Comprehensions workflow consists of several stages, progressively lowering the level of abstraction (Figure 2). Given a TC with specialized tensor sizes and strides,[3] we lower it to a parametric Halide-IR expression, which is further lowered to a polyhedral representation where most transformations are applied. The output of the polyhedral flow is CUDA code that can be further

[3]Our toolchain supports parametric specifications, yet we have found early specialization to be beneficial in driving profitability decisions during polyhedral scheduling.

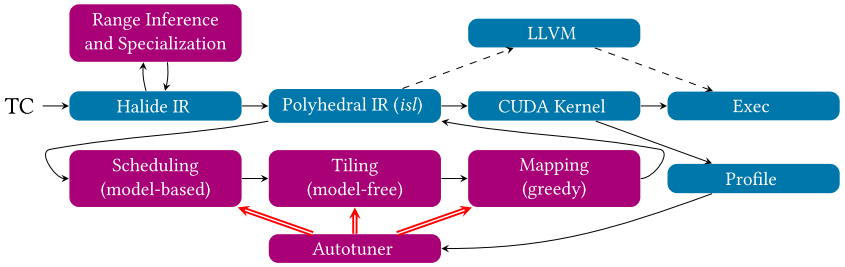

Fig. 2. The JIT compilation flow lowers TC to Halide-IR, then to Polyhedral-IR, followed by optimization, code generation, and execution.

JIT-compiled with NVRTC and executed. Complementing this flow, an autotuner and serializable compilation engine interacts with scheduling and mapping strategies to search the optimization space.

Much of TC's versatility and effectiveness resides in its embedding of a polyhedral compiler as the main optimization engine. The polyhedral framework is an algebraic representation of "sufficiently regular" program parts, covering arithmetic expressions on arrays surrounded by static control flow [23]. It has been a cornerstone of loop optimization in the past three decades [3, 8, 14, 22, 32, 70] and is integrated into production compilers [13, 30, 43, 62]. Despite its deceiving apparent simplicity, it covers a large class of computationally intensive kernels. It is parametric on loop bounds and array sizes and captures more transformations of the control and data flow than domain-specific representations such as Halide [55] or TVM [17]. The use of the polyhedral model by TC is derived from that of PPCG [70], and this section only provides a general overview. Our transformation engine is composed of the following specially adapted or algorithmically novel components:

(1) range inference and lowering from high-level TC abstraction to the polyhedral representation;
(2) core affine scheduling adapted from *isl* that automatically optimizes for (outer) loop parallelism and locality, tuned towards folding a complete TC function into a single GPU kernel;
(3) the schedule is further tiled to facilitate the mapping and temporal reuse on the deep parallelism and memory hierarchy of GPUs [72];
(4) mapping to GPUs borrows from PPCG [70] with extensions to support the more complex and imperfectly nested control structures of ML kernels;
(5) memory promotion deals with explicit data transfers to and from shared and private memory.

This work demonstrates that the polyhedral framework is particularly well suited for deep neural networks, featuring large and deeply nested loops with long dependence chains and non-uniform or all-to-all patterns—arising from fully connected layers and tensor contractions, and transpositions. These features push the optimization problem into a different heuristic space than Halide's for image processing, and a wider space than linear algebra alone.

3.1 Range Inference

TC loops are implicit and output tensor sizes are inferred from index ranges, which themselves may also be inferred. Our algorithm infers the largest rectangular ranges that avoid out-of-bounds reads on inputs. A `where` clause allows for disambiguation if multiple such ranges exist.

Consider the conv2d kernel on page four. The sizes of the input tensors, in and weight, are known from the function signature. The algorithm needs to infer the ranges of the iterators and the size of the output tensor out. The iterators b, op, kh, and kw appear only once on the RHS and their ranges are therefore $[0, B)$, $[0, OP)$, $[0, KH)$, $[0, KW)$ so they index the input tensors maximally. The iterator ip appears twice, but indexes the dimension of the same size, so its range is $[0, IP)$. Had it been indexing dimensions of different sizes, its range would have been the intersection of all size-imposed ranges. Once the ranges of kh and kw are known, it is possible to infer those of h and w: We require $h + kh \leq H$ and $w + kw \leq W$, which leads to the maximal ranges of $[0, H - KH)$ and $[0, W - KW)$, respectively. Finally, the size of out can be inferred given the ranges of the iterators that index it, yielding float$(B, OP, H - KW, W - KW)$. The user of TC is able to inspect the symbolic sizes inferred for the output tensors using a command-line flag.

Consider now a typical stencil operation A(i)+=B(i + k) ∗ K(k): There are multiple ways to maximize the ranges of i and k. To disambiguate without annotations, range inference proceeds in rounds. It maintains a set of index variables whose ranges are not yet resolved. Initially, it contains all variables not in any where clause. Each step considers argument expressions that contain a single unresolved variable and constructs a Boolean condition stating the accesses are within bounds. Using Halide [55] mechanisms, range inference computes the maximal range that satisfies this condition given the already known ranges of other variables. If different ranges are computed for the same variable, they are then intersected. For the stencil above, in the first round, we ignore the expression B(i + k), because it contains multiple unresolved variables. We use K(k) to deduce a range for k. In the second round, B(i + k) contains a single unresolved variable, and we use the already-inferred range of k to deduce a maximal range for i.

3.2 Lowering to the Polyhedral Representation

The role of lowering is to bridge the impedance mismatch between the logical layout of high-level tensor operations (dimension ordering) and the data format the polyhedral code generator expects (C-style row-major arrays). It ensures the absence of aliasing and performs range inference for output tensors. Based on range inference, TC differs from NumPy-style implicit "broadcast" semantics (non-trivial tensor dimensionality extension) adopted by XLA, PyTorch, and MXNet.

Our representation derives from schedule trees [71], implemented in the *isl* library [68], and uses a set of node types. Each TC-statement corresponds to multiple runtime statement *instances*—one for every valuation of the index variables. The root *domain node* defines the set of statement instances to be executed. Due to the nature of the TC-language, the constraints on the index variables are always affine, resulting in an exact representation of the set of operations. A *band node* defines a *partial* execution order through one or multiple piecewise affine functions defined over iteration domains. The name refers to the notion of a *permutable schedule band*, a tuple of one-dimensional schedule functions that can be freely interchanged while preserving the semantics of the program. A *filter node* partitions the iteration space, binding its sub-tree to a subset of the iteration domain. It can be arranged into *set or sequence nodes* depending on whether or not the order of execution must be serialized. *Context nodes* provide additional information on the parameters, e.g., tensor extents or GPU grid/block sizes. Finally, *extension nodes* introduce auxiliary computations that are not part of the original iteration domain, which is useful for, e.g., introducing data-copy statements.

A *canonical* schedule tree for a TC is defined by an outer *sequence* node, followed by *filter* nodes for each TC statement. Inside each filtered branch, *band* nodes define an identity schedule with as many one-dimensional schedule functions as loop iterators for the statement. The implicit loops form a permutable band as per TC semantics.

In addition to the schedule tree, our representation includes tensor access functions that map the index variables to the subscripts of tensors they access. These subscripts are not necessarily affine,

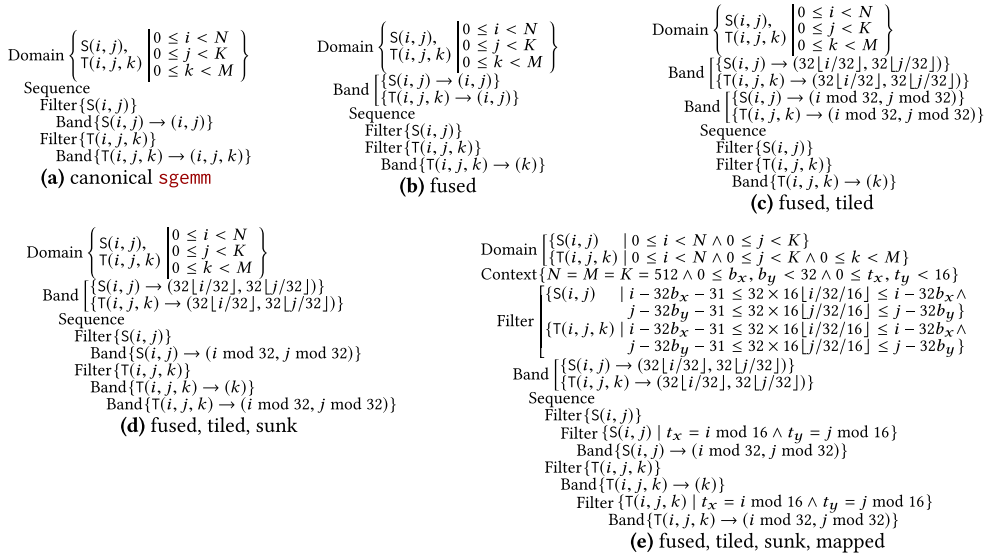

Fig. 3. Optimization steps for sgemm.

in which case over-approximations are used [11]: A non-affine access is assumed to potentially access *all* values along the given dimension. After the polyhedral representation is constructed, dependence analysis can be used to ensure the absence of out-of-bounds accesses [53].

Additional lowering steps include forward substitution of convolution expressions (storage/computation trade-off), padding, mirroring, and clipping. The process is analogous to Halide's [55].

Example. Figure 3(a) shows the canonical schedule tree for unions of relations where tuples of iterators are guarded with syntactic identifiers [53][4] for the sgemm TC defined on page 4. One recognizes a 2-D nest from the initialization statement followed by a 3-D nest for the update statement. The schedule can be either parametric in input sizes or have extra context information on the tensor sizes. In cases where *band* nodes do not define an injective schedule, the statement instances are scheduled following the lexicographical order of their domain coordinates.

3.3 Tunable Polyhedral Scheduling

Program transformation in the polyhedral model involves defining a different schedule, which corresponds to a different (partial or total) order of traversing the iteration domain. The instances of all statements are scheduled completely automatically [14] using one of several scheduling strategies with which we extended the *isl* scheduler [72].

The *isl* scheduler iteratively solves integer linear programming problems to compute piece-wise affine functions that form new schedule *band* nodes. Internally, it operates on a data dependence graph where nodes correspond to statements and edges express dependences between them. It introduces the *affine clustering* technique that is based on computing the schedule bands separately for individual strongly connected components of the dependence graph and then clustering these components iteratively and scheduling them with respect to each other. Clustering not only decreases the size of the linear problems the scheduler has to solve, but also serves as a basis for *isl*'s loop fusion heuristic.

[4]We use the *named relation notation* of iscc [69]. The declaration of parameters $(N, M, K) \rightarrow \{\ldots\}$ is omitted hereinafter for brevity.

We extended *isl* to provide finer-grained control over the scheduling process. For affine transformations, the user can set additional scheduling options. For clustering, the user can supply a decision function for pairwise dependence graph component combination, after this combination was demonstrated to be valid by the scheduler. These configuration points serve as a basis for both fixed scheduling choices made by TC and *scheduling strategies*. In particular, TC tells the scheduler to produce schedules with only non-negative coefficients and without any skewing. Clustering decisions allow TC to control the conventional minimum and maximum fusion targets, and additionally, maximum fusion that preserves at least three nested parallel loops (to be mapped to CUDA blocks and threads). With the scheduling strategies, one may optionally enable point band rescheduling (i.e., scheduling the inner dimensions after tiling). In particular, two fusion strategies can be specified, one for the global schedule and one for the point band. If these fusion strategies are different, then the point band (along with all its descendants) is rescheduled after tiling, preserving only the outer tile band of the original schedule. Scheduling strategies can be selected through the autotuning process. In all cases, we enforce that a single GPU kernel is generated.

Example. Observing that the `C` tensor in `sgemm` (see page four) is reused between two nests, the scheduler constructs the tree in Figure 3(b) to leverage access locality and improve performance. This tree features an outer band node with `i` and `j` loops that became common to both statements, which corresponds to *loop fusion*. The sequence node ensures that instances of `S` are executed before respective instances of `T` enabling proper initialization. The second band is only applicable to `T` and corresponds to the innermost (reduction) loop `k`.

Overall, the tuning process is greatly simplified compared to Halide and TVM. Relying on a heavy-duty, well-understood analytical optimization framework based on integer linear programming, TC exposes a small, dedicated search space of high-level strategies and block-size parameters. Beyond guaranteeing the validity of the transformation, dependences can be used to explore parallelization opportunities (independent instances can be executed in parallel), to improve data access locality (dependent instances executed close in time) or to automate vectorization [14, 50, 66, 72, 77].

3.4 Imperfectly Nested Loop Tiling

Let us first describe the general setting for loop tiling on schedule trees, before developing the TC-specific specialization and extensions.

Tiling Permutable Bands. Pluto has been very successful at decoupling the actual implementation of loop tiling from the preparation of an affine schedule exposing permutable loops amenable to tiling [14]. This design allows exploring locality and parallelization tradeoffs without bloating the schedule representation with complex quasi-affine forms capturing the precise distribution of iterations into tile and point loops. Schedule trees ease the implementation of such a decoupled design, capturing tiling as the conversion of a permutable schedule band into a chain of two bands, with the outer band containing tile loops and the inner band containing point loops with fixed trip count. This can be seen as a conventional strip-mine and sink transformation.

In addition to conventional loop tiling, the schedule tree representation allows tiling imperfectly nested loops. The technique is based on the following observation: If a loop does not carry dependences, it can be sunk below any other loop. In valid schedules, all dependences are carried (or satisfied) by some loop, along which they feature a positive distance. A dependence is only violated if it has a negative distance along some loop *before* it is carried by another loop [35]. Parallel loops do not carry dependences by definition and therefore do not affect dependence satisfaction or violation. Therefore, imperfectly nested tiling may be implemented by first tiling bands in isolation and then sinking parallel point loops in the tree. During this process, the point band is replicated in

each sub-tree below a sequence (or set) node and its schedule is restricted to only map the relevant points in the iteration domain. Such an extension is particularly helpful in Pluto, where bands of permutable loops are rediscovered through a post-pass traversal of the affine schedule.

Parallelism and Locality Trade-offs. TC applies two tiling schemes with complementary purposes.

The first one takes place immediately after affine scheduling. It aims at exposing a sufficient number of parallel dimensions, some of which amenable to memory coalescing, and some better suited to block-level parallelism. It also aims at exploiting data locality within thread blocks (through shared memory) and individual threads (through register reuse). This tiling scheme is influenced by the strong emphasis on loop fusion in the affine scheduling heuristic (to enforce that the generated code runs as a single GPU kernel). In this context, conventional loop nest tiling—considering a single band at a time—appears to be sufficient. This is the hypothesis we make in this article.[5]

The second tiling scheme takes place in the block and thread mapping algorithm, which is the topic of the next sub-section.

Example. Figure 3(c) shows the schedule tree for the fused and tiled `sgemm`. It purposely has two imperfectly nested bands. Dependence analysis shows that loops `i` and `j` are parallel. Therefore, we can tile them and sink the point loops below the band of the reduction `k` loop, resulting in the schedule tree in Figure 3(d). Innermost nested bands with point loops can be joined together into a single band after checking for permutability. As indicated earlier, TC implements the fusion and tiling scheme of Figure 3(c) but not the sunk, imperfect scheme of Figure 3(d).

3.5 Mapping to Blocks and Threads

A schedule tree can also be used to represent the *mapping* to an accelerator, in particular a GPU with multiple blocks and threads. This operation is performed by associating certain schedule band members, and the corresponding loops, to thread or block indices. The polyhedral code generator then omits the loops, if possible, and rewrites the index expressions accordingly. Building on PPCG, our mapping approach is decoupled from tiling for data locality: Grid and block sizes are specified independently from tile sizes and are exposed as tunable parameters. Due to the semantics of blocks and threads, only parallel loops that belong to a permutable schedule band can be mapped. If point loops are mapped to threads, the ratio between tile sizes and block sizes controls the number of iterations executed by each thread. Note that tile sizes smaller than the block sizes lead to some threads not performing any computation.

Contrary to PPCG, which may generate multiple kernels for a given input program, our mapping approach handles imperfectly nested loops in a way that generates a single kernel as expected by ML frameworks. We require the schedule tree to have at least an outermost band with outer parallel dimensions. The parallel dimensions of the (single) outermost band are mapped to GPU blocks. In each schedule tree branch, the innermost permutable band, typically consisting of point loops, is mapped to GPU threads with the following restrictions: The number of mapped dimensions must be equal across branches, and on each branch, there must be exactly one band mapped to threads. The mapping is performed bottom-up, first attempting to map the leaf bands to threads, before moving to a parent band only if none of the children could be mapped to threads.

Thread mapping can be extended to imperfectly nested loops, following the same principle as imperfect loop tiling. Within a given thread block, one may sink parallel point loops so multiple bands in a sequence (or set) may be equalized in depth and mapped together. However, TC currently does not perform any such sinking.

[5]The TC implementation supporting our experiments does not implement imperfect loop tiling after affine scheduling.

Example. Our mapping strategy produces the schedule tree in Figure 3(e). We introduced a context node in the schedule tree to indicate the effective sizes of the parameters as well as the grid and block sizes (denoted as b_x, b_y and t_x, t_y, respectively, standing for the values eventually taken by `blockIdx.x`, `blockIdx.x` and `threadIdx.x`, `threadIdx.y`). This insertion is performed just in time, when the effective tensor sizes are known. Also notice the filter nodes referring to the b_x, b_y, t_x, and t_y parameters: these nodes express the *mapping* to the GPU.

3.6 Memory Promotion

We are interested in promoting parts of tensors into shared or private GPU memory. While the promotion decision is taken by a heuristic and the corresponding imperative code is generated at a later stage, schedule trees offer a convenient interface for attaching memory-related information. Memory promotion is based on the notion of an *array tile*, a form of data tiling for software-controlled local memories. It is a constant-size potentially strided block in the array that covers all elements accessed by within a given (schedule) tile. We build upon and extend PPCG's support for memory promotion [70, 72] and expose the promotion to shared and private memory as Boolean options for the autotuner.

Promotion of Indirectly Accessed Arrays. Memory promotion is also applicable to indirectly accessed arrays. These frequently occur when modeling variable length data through *embedding layers* such as word embeddings in natural language processing. This is particularly important in the case of latency-bound benchmarks where there is little computational or additional data processing work to hide global memory latency. Indirect arrays used to be promoted in the initial TC implementation based on PPCG. When implementing parallel reductions, working towards the first released version of TC, we realized that parallelizing reductions was sufficient to deliver comparable or higher speedups in our word-embedding benchmarks. For this reason, indirect array promotion was dropped from the publicly available version of TC. We still report on the design, for it remains interesting to describe how the polyhedral TC flow may optimize non-affine data flow.

Without loss of generality, consider the access `O[l + Idx[i][j]][k]`. We refer to `O` as the outer array and to `Idx` as the index array. In case of nested indirections, outer/index pairs are processed iteratively from innermost to outermost. While the values taken by the first index expression of the outer array are unknown statically, we can still cache them locally as `shared_O[l][i][j][k]= O[l + Idx[i][j]][k]`. Because some values can be duplicated, indirect promotion is only possible if both the outer and the index arrays are only read, since writing to them could result in different values that cannot be trivially merged. In general, we require the index array to have an array tile, i.e., only a fixed-sized block of it is accessed. When computing the array tile for the outer array, we ignore the indirect parts of the subscript (affine parts are treated as usual). We then introduce as many additional index expressions in the promoted outer array as are associated to the index array. Extents of the array along these new dimensions correspond exactly to the array tile sizes of the index array. Hence, an element of the promoted array contains a copy of the global array element that would be accessed with the given index array. Indirect subscripts are only used when copying from global memory, while all other accesses are rewritten through code generation. In presence of multiple indirect index expressions that share sub-expressions and have equal tile sizes along the corresponding dimensions, it is sufficient to introduce a single index expression in the promoted array for all identical sub-expressions.

Promotion Heuristics. Directly accessed arrays are promoted to shared memory if there exists an array tile of fixed size, if individual elements are accessed more than once, and if at least one of the accesses does not feature memory coalescing. The latter is visible from the access relation with the schedule applied to the domain: The last access dimension should be aligned with the schedule dimension mapped to `x` threads.

For indirect arrays, the coalescing requirement may be dropped because of the presence of additional long memory dependences that these cases entail. The total amount of shared memory being fixed, one may follow a simple greedy heuristic, refusing promotion if the required amount of shared memory would outgrow the available resources.

3.7 Matching Library Calls

While TC aims at generating code for any computational kernel expressible in the DSL, if (part of) a kernel happens to match a pattern that is heavily optimized by some library, then it may as well be handled by that library. In particular, and as a proof of concept, TC looks for opportunities for letting CUB handle specific forms of reductions [57]. It is currently restricted to single-dimensional addition reductions.

A reduction is represented in TC by a binary relation between updated tensor elements and the statement instances that perform the corresponding updates.[6] Right before the mapping to threads, each permutable band with a sufficient number of parallel members is checked for reductions. In particular, the band should have at least one non-parallel member and the number of parallel members plus one (corresponding to the non-parallel member) should be greater than or equal to the number of dimensions that will be mapped to threads. If the band schedules instances of exactly one reduction statement and if the instances of any other statement scheduled by the band can be moved before or after the reduction instances, taking into account the active dependence at (the top of) the band, then the remaining band (involving only reduction statement instances) will be considered for replacement by a library call during thread mapping.

When a band marked for replacement is considered during thread mapping, full/partial tile separation is applied—using the block size tuning parameter—since only the full tiles can be handled directly by CUB. Furthermore, the condition separating full tiles from partial tiles should be simple enough, as otherwise the cost of determining when to invoke CUB would outweigh any possible benefit obtained from the invocation. If the condition is too complicated, the separation is discarded and the band is treated in the same way as bands that were not marked for replacement. Otherwise, the collection of full tiles is tiled along the parallel dimensions, since a single scalar variable is used to hold the result of the reduction mapped to CUB. Synchronization and a special marking is then inserted around the point band of this tiling, which is later used during code generation to replace each full tile by a call to CUB. Finally, since CUB uses some shared memory, its consumption is taken into account during the downstream memory promotion step.

3.8 Autotuning and Caching

While the polyhedral core of TC is capable of optimizing and generating code for any TC function, it is well known that the state-of-the-art linear optimization heuristics are not sufficient to account for all performance anomalies and interactions with downstream program transformations [39, 77]. Different kernels need different, target-specific optimization trade-offs. We thus complement our flow by an autotuner that varies the options of the polyhedral JIT compiler marked as *tunable* in the previous section. These options can be stored and reused for similar operations/kernels (similar shapes, target architecture), since autotuning may require significantly more time than compilation.

The tuning session is defined by a list of parameters to tune and their admissible values, initial values, and the search strategy. We currently implement a genetic search strategy [27]. It runs for multiple steps, each one evaluating multiple candidate values. Each candidate is assigned a fitness value inversely proportional to its runtime. The pool is updated on each generation by

[6] This description is based on commit TC commit `8cfdd5764`, which is slightly ahead of the commit used in the experiments, but is easier to explain.

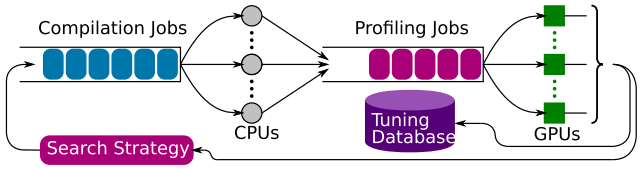

Fig. 4. Multithreaded autotuning pipeline for kernels.

cross-breeding three candidates, chosen from the pool at random, with fitter candidates having a higher chance of being chosen, such that each candidate's value is inherited from one of its parents. A subsequent mutation phase can change the candidate's values at random with some low probability. Much of the autotuning effort resides in tile size selection, for which no linear objective functions exist in polyhedral compilers. Genetic approaches have been used successfully to explore such spaces, performing better than random search due to the strong coupling of optimization decisions—including tile sizes bound by the limits of the memory hierarchy—[18, 50].

Autotuning evaluates hundreds to thousands of versions for each kernel. We devise a generic multi-threaded, multi-GPU autotuner. It maintains a queue of candidates to compile with the polyhedral flow and a queue of compiled kernels ready to be profiled on the GPU (see Figure 4). Candidates or kernels are picked up by available worker threads and compiled or profiled concurrently. Profiling results are accumulated in the tuning database and used for setting up successive search steps.

Each generated version is "warmed up" by a few executions before being profiled. Without any performance guarantees, autotuning needs to quickly prune poor candidates. Because CUDA kernels cannot be stopped once launched, we rely on the following pruning heuristics to decrease the autotuning time by an order of magnitude. (1) Parameter specialization allows the exact number of active threads and blocks to be computed beforehand. Kernels with fewer threads than some configurable threshold (e.g., 256) are not launched. (2) If during the first run, a kernel is more than 100× slower than the best version so far, or it is 5× slower after warmup, it is pruned immediately.

While autotuning time may become significant, compilation and autotuning time is not a fundamental limit to TC's applicability. In training scenarios, a significant amount of time is spent on computing the same kernel repeatedly over different data during the (stochastic) gradient descent. In inference scenarios, the network is optimized ahead of time. As a result, although TC operates as a JIT compiler, it only marginally hits the typical compilation/run-time trade-offs of JIT compilers. Autotuning time may become an issue in specific training scenarios where hyper-parameters would need to be frequently updated, but in such a case one may leverage TC's intrinsic handling of dynamic shapes and generate a single version of each operator or fused operators to handle all hyper-parameter configurations.

4 INTEGRATION WITH ML FRAMEWORKS

TC is designed to optimize individual layers or small subgraphs of an ML model. The entire model is not only computationally expensive, but often leads to most transformations being hindered by a large number of data dependences. Furthermore, ML frameworks perform work distribution and placement at the model level, treating a layer as a unit of work; extremely large layers could interfere with the framework operation.

Unlike XLA or Glow, TC supports completely custom layers. In TC, *layer fusion* is merely pasting the code that constitutes the layers into a single function, or inlining TC functions at the AST level. Unlike Halide and TVM, the polyhedral backbone of TC includes instance-wise dependence analysis, capturing dependences and tensor access relations at the level of individual loop iterations and tensor elements. This allows TC to fuse operations without introducing

```
string tc = R"TC(some_tc_for_conv)TC";
auto I = makeATenTensor<CudaBackend>({N, C, H, W});
auto W = makeATenTensor<CudaBackend>({F, C, KH, KW});
ATenAutotuner<CudaBackend> tuner(tc);
auto best = tuner.tune("conv", {I, W});
auto pExecutor =
    compile<CudaBackend>(tc, "conv", {I, W}, best[0]);
auto out = prepareOutputs(tc, "conv", {I, W});
auto times = profile(*pExecutor, {I, W}, outs);
```

```
import torch
import tensor_comprehensions as tc
tcdef = """...some_tc_for_conv..."""
T_I = torch.randn(N, C, H, W).cuda()
T_W = torch.randn(F, C, KH, KW).cuda()
# register the TC string
conv = tc.define(tcdef, name="conv")
# autotune the kernel
best = conv.autotune(T_I, T_W)
# run with best option and cache the binary
T_O = conv(T_I, T_W, options=best)
```

Fig. 5. Example of embedded usage in C++/ATen (top) and PyTorch (bottom).

redundant computation, and to combine fusion with enabling transformations such as shifting (for convolutions) or scaling (for pooling layers). TC's polyhedral representation also enables it to automatically infer sizes, and to discover parallelism and locality-parallelism trade-offs beyond a predefined collection of map/reduce/scan combinators.

Let us now describe the transparent integration into a ML framework, from a user perspective. Until now, such levels of integration had only been demonstrated on operator graph compilers such as XLA [28] and Glow [58], starting from a lower level of abstraction than TC, and missing the genericity and high reusability of a polyhedral framework as well as feedback-directed autotuning.

We opted for an "in process" implementation, streamlining the interaction with computation graph engines and ML applications built on top of them, a unique feature for a fully automated scheduling and mapping flow. TC is integrated into any ML framework as follows: We provide a thin API that translates the specific tensor object model to our own (see Figure 5). Operator definitions are overridden to generate TC rather than the framework's backend implementation, as well as provide users the ability to write their own TC. A single TC may correspond to a DAG of operators in the ML framework. The tensor comprehensions are then JIT-compiled as shown in Figure 2. DAG partitioning, matching, and rewriting (like, e.g., TensorRT [47]) is currently not part of the flow, although this would make an interesting future combination, with feedback from the compiler.

5 PERFORMANCE RESULTS

We evaluate our framework on two systems: (1) Nvidia Pascal nodes with 2 socket, 14 core Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, with two Quadro P100-12GB; and (2) Nvidia Volta nodes with 2 socket, 20 core Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz, with eight Tesla V100-SXM2-16GB. Both systems use CUDA 9.0 and cuDNN 7.0.

We report results for nine TC functions ranging from a simple matrix multiplication kernel to a full WaveNet cell [64]. The individual benchmarks are described below: Figure 6 and Figure 7 show the complete source code. The matrix multiplication and convolution kernels were selected for their dominance of the training and inference time of the most classical networks [4, 75]. The other kernels bring interesting computation patterns to enable expressiveness and performance comparisons in more diverse network architectures.

These results are all based on TC commit, `2e1a0dc54850` available at https://github.com/nicolasvasilache/TensorComprehensions.

Running the autotuner for 25 generations of 100 candidates, the (parallel) autotuning process takes up to 1h on the longest running kernels, and 6h in total.[7]

The relative performance of kernels automatically generated with TC compared to Caffe2 is shown in Figure 8 and Figure 9.[8] Caffe2 provides a very strong baseline by wrapping tuned

[7]Classical strategies exist to accelerate autotuning, such as predictive modeling and search space pruning [2], but this was not the focus of this article.

[8]We compile Caffe2 and PyTorch from source (commit `6223bfdb1d32`) and integrate it in the TC testing flow for proper benchmarking.

```
def tmm(float(M,K) A, float(N,K) B) → (C) { C(m,n) +=! A(m,r_k) * B(n,r_k) }
def tbmm(float(B,N,M) X, float(B,K,M) Y) → (Z) { Z(b,n,k) +=! X(b,n,r_m) * Y(b,k,r_m) }

def 1LUT(float(E1,D) LUT1, int(B,L1) I1) → (O1) { O1(i,j) +=! LUT1(I1(i,r_k),j) }
def 2LUT(float(E1,D) LUT1, int(B,L1) I1, float(E1,D) LUT1, int(B,L1) I1) → (O1, O2) {
  O1(i,j) +=! LUT1(I1(i,r_k),j)
  O2(i,j) +=! LUT2(I2(i,r_k),j)
}

def MLP3(float(B,M) I, float(O,N) W2, float(O) B2, float(P,O) W3, float(P) B3, float(Q,P) W4,
   float(Q) B4) → (O1,O2,O3,O4) {
  O2(b,o)  = B2(o)
  O2(b,o) += O1(b,r_n) * W2(o,r_n)
  O2(b,o)  = fmaxf(O2(b,o), 0)
  O3(b,p)  = B3(p)
  O3(b,p) += O2(b,r_o) * W3(p,r_o)
  O3(b,p)  = fmaxf(O3(b,p), 0)
  O4(b,q)  = B4(q)
  O4(b,q) += O3(b,r_p) * W4(q,r_p)
  O4(b,q)  = fmaxf(O4(b,q), 0)
}

def kronecker3(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,r_n2) * W2(d2,r_n2)
  XW1(m,n0,d1,d2) +=! XW2(m,n0,r_n1,d2) * W1(d1,r_n1)
    Y(m,d0,d1,d2) +=! XW1(m,r_n0,d1,d2) * W0(d0,r_n0)
}

def group_convolution(float(N,G,C,H,W) I, float(G,F,C,KH,KW) W1, float(M) B) → (O) {
  O(n,g,o,h,w)  = O(n,g,o,h,w) + B(m)
  O(n,g,o,h,w) += I(n,g,r_i, h + r_kh, w + r_kw) * W1(g,o,r_i,r_kh,r_kw)
}

def moments2_2d_1D(float(N,K) I) → (mean,var) {
# var = E(x^2) - mean^2
  mean(n) +=! I(n,r_k)
  var(n)  +=! I(n,r_k) * I(n,r_k)
  mean(n)  =  mean(n) / K
   var(n)  =  var(n) / K - mean(n) * mean(n)
}

def group_normalization(float(N,G,D,H,W) I, float(G,D) gamma, float(G,D) beta,
      float(N,G) mean, float(N,G) var) → (O,mean,var) {
  O(n,g,d,h,w) = gamma(g,d) * (I(n,g,d,h,w) - mean(n,g))
         * rsqrt(var(n,g) - mean(n,g) * mean(n,g) + 1e-5) + beta(g,d)
}

def group_normalization_single_kernel(float(N,G,D,H,W) I, float(G,D) gamma,
              float(G,D) beta) → (O,mean,var) {
  mean(n,g) +=! I(n,g,r_d,r_h,r_w)
  var(n,g)  +=! I(n,g,r_d,r_h,r_w) * I(n,g,r_d,r_h,r_w)
  O(n,g,d,h,w) = gamma(g,d) * (I(n,g,d,h,w) - mean(n,g) / (D * H * W)) * rsqrt(var(n,g)/(D * H * W)
                 - mean(n,g)/(D * H * W) * mean(n,g)/(D * H * W) + 1e-5) + beta(g,d)
}
```

Fig. 6. TC Benchmarks used in the experiments. Evaluated sizes are available in Table 1.

implementations, which originate from either hand-tuned libraries or other high-performance code generators.[9] *We chose to compare against Caffe2 rather than against other optimization flows due to expressivity and automation limitations*: XLA or Glow do not support custom layers, and Halide or TVM lack range inference and automatic parallelism discovery, which significantly complicates the expression of new layers such as KRU and WaveNet. The common set of comparable layers would be limited to matrix multiplications and convolutions, while one of the main contributions of TC is to enable exploration of new unconventional layers *before super-optimized implementations are available*.

In addition, Figure 10 brings together the performance of TC-compiled kernels on both GPU systems, normalized to Caffe2 on P100. This consolidated graph conveys three classes of information in a common context: (1) speedup of Caffe2 V100 over Caffe2 P100 to illustrate the out-of-the-box benefits (or lack thereof) of a faster GPU; (2) speedup of TC over Caffe2 on P100 (main comparison);

[9] A recent unification effort [59] made Caffe2 the backend for PyTorch 1.0.

```
def wavenet1(float(B, RESIDUAL_C, RECEPTIVE_FIELD) Data,
             float(DILATION_C, RESIDUAL_C, 2) FilterWeight, float(DILATION_C) FilterBias,
             float(DILATION_C, RESIDUAL_C, 2) GateWeight, float(DILATION_C) GateBias,
             float(RESIDUAL_C, DILATION_C) ResWeight, float(RESIDUAL_C) ResBias,
             float(SKIP_C, DILATION_C) SkipWeight, float(SKIP_C) SkipBias,
             float(DILATION_FACTOR) Dilation)
    → (FilterOut, GateOut, NonLin, Res, Skip) {
  FilterOut(b, dilation_c, rf)  = FilterBias(dilation_c)
    where b in 0:B, dilation_c in 0:DILATION_C, rf in 0:RECEPTIVE_FIELD
  FilterOut(b, dilation_c, rf) += Data(b, r_residual_c, rf)
      * FilterWeight(dilation_c, r_residual_c, 1) + ((rf - DILATION_FACTOR ≥ 0)
        ? Data(b, r_residual_c, rf - DILATION_FACTOR) * FilterWeight(dilation_c, r_residual_c, 0)
        : float(0))
      where rf in 0:RECEPTIVE_FIELD

  GateOut(b, dilation_c, rf)  = GateBias(dilation_c)
    where b in 0:B, dilation_c in 0:DILATION_C, rf in 0:RECEPTIVE_FIELD
  GateOut(b, dilation_c, rf) += Data(b, r_residual_c, rf)
      * GateWeight(dilation_c, r_residual_c, 1) + ((rf - DILATION_FACTOR ≥ 0)
        ? Data(b, r_residual_c, rf - DILATION_FACTOR) * GateWeight(dilation_c, r_residual_c, 0)
        : float(0))
      where rf in 0:RECEPTIVE_FIELD

  NonLin(b, dilation_c, rf)  = tanh(FilterOut(b, dilation_c, rf))
    where rf in 0:RECEPTIVE_FIELD
  NonLin(b, dilation_c, rf) *= 1 / (1 + exp(-GateOut(b, dilation_c, rf)))
    where rf in 0:RECEPTIVE_FIELD

  Res(b, residual_c, rf)  = Data(b, residual_c, rf) + ResBias(residual_c)
  Res(b, residual_c, rf) += NonLin(b, r_dilation_c, rf) * ResWeight(residual_c, r_dilation_c)

  Skip(b, skip, rf) +=! NonLin(b, r_dilation_c, rf) * SkipWeight(skip, r_dilation_c)
    where rf in 0:RECEPTIVE_FIELD
  Skip(b, skip, rf)  =  Skip(b, skip, rf) + SkipBias(skip)
    where rf in 0:RECEPTIVE_FIELD
}
```

Fig. 7. Source of one full WaveNet cell.

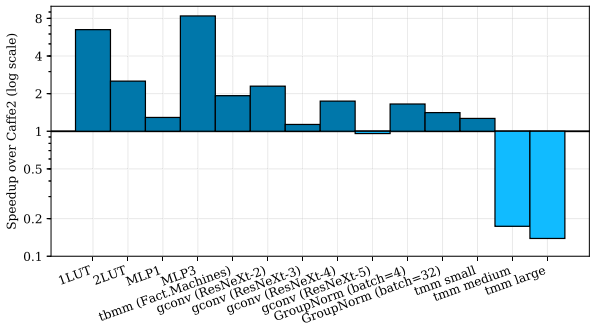

Fig. 8. Speedup of TC-generated kernels over Caffe2 hand-tuned kernels on Quadro P100-12GB.

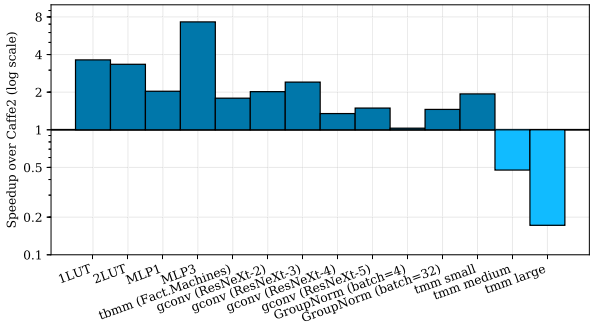

Fig. 9. Speedup of TC-generated kernels over Caffe2 hand-tuned kernels on Tesla V100-SXM2-16GB.

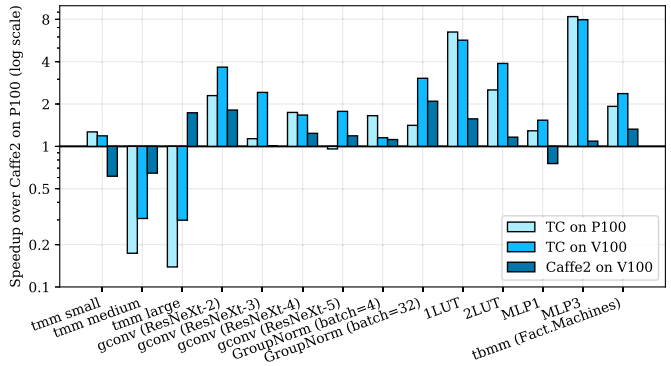

Fig. 10. Relative performance: baseline is Caffe2 performance on a P100 GPU.

and (3) speedup of TC V100 over Caffe2 P100. The last choice may seem surprising, but presented in the context of the other two, allows for relative comparisons: the height of the Caffe2 V100 and TC V100 captures the raw speedups of TC on V100. *We aim at compactly illustrating that TC provides a path to performance portability, improving on state-of-the-art frameworks and library primitives.*

Table 1 provides absolute runtime running TC and Caffe2; all values are reported in μs.

TMM: Transposed Matrix-Multiplication. On matrix multiplications of shapes and sizes relevant to deep learning workloads (i.e., small $128 \times 32 \times 256$, medium $128 \times 1{,}024 \times 1{,}024$ and large $128 \times 4{,}096 \times 16{,}384$), TC does not perform competitively, except in the low-latency small case. This is due to: (1) the lack of a target-specific register blocking optimization, making kernels bound by shared memory bandwidth that is an order of magnitude slower than register bandwidth; (2) the lack of target-specific, basic-block level optimizations including careful register allocation and instruction scheduling. Matrix multiplication is the most tuned computation kernel in history: The missing optimizations are all well known and may be found in use cases and open-source implementations such like CUTLASS [36]. Alternatively, polyhedral compilation has been shown to match or outperform cuBLAS, provided sufficient target- and operator-specific information has been captured in the optimization heuristic and code generator [20]. While our scientific focus was on covering a wide range of layers with TC, a production release would need to embed such operator-specific strategies as well. One strategy would be to follow the classification and heuristic steering of Kong et al. [39]. Also, TC does not replace all layers: It only acts as a custom operation in a graph; one may use TC concurrently with numerical libraries as well as custom implementations provided through TVM.

Group Convolution. Group convolution is expressible with two lines of TC. We report comparisons for sizes relevant to the ResNext model [75]. Despite not using either register optimizations, Fourier or Winograd domain convolutions, TC produces faster kernels than the cuDNN ones, with running times between $250\mu s$ and $750\mu s$. To check how TC fares w.r.t. recent advances in optimizing group convolutions, we performed an additional comparison with the PyTorch nightly package `py36_cuda9.0.176_cudnn7.1.2_1` with `torch.backends.cudnn.benchmark=True`. TC speedups range from -2% to $8\times$. We also observe PyTorch performance on V100 to be worse than on P100, while TC achieves performance portability.

Group Normalization. Group Normalization was recently proposed as a way to overcome limitations of Batch Normalization at smaller batch sizes and increase parallelism [74]. In TC, group normalization is a five-line function. TC performance is roughly 30% better than the hand-tuned Caffe2 implementation. Whereas Caffe2 uses four *handwritten* CUDA kernels, we chose to write the TC version as two separately compiled TC functions for better reuse and overall performance.

Table 1. Absolute Run Time in μs

		Pascal			Volta		
1LUT		p0	p50	p90	p0	p50	p90
$B = 128, D = 64, E1 = 10^7, L1 = 50$	TC	13	14	14	15	16	17
	Caffe2	85	91	95	56	58	63
2LUT		p0	p50	p90	p0	p50	p90
$B = 128, D = 64, E1, E2 = 10^7, L1, L2 = 50$	TC	52	54	57	35	35	37
	Caffe2	132	136	144	115	117	124
MLP1		p0	p50	p90	p0	p50	p90
$B = 128, M = 2000, N = 128$	TC	68	69	71	57	58	59
	Caffe2	87	89	91	116	118	123
MLP3		p0	p50	p90	p0	p50	p90
$B = 128, N = 128, O = 64, P = 32, Q = 2$	TC	18	19	19	20	20	21
	Caffe2	157	159	169	144	146	164
tbmm		p0	p50	p90	p0	p50	p90
$B = 500, K = 26, M = 72, N = 26$	TC	52	53	54	42	43	43
	Caffe2	94	102	103	76	77	78
Group Convolution		p0	p50	p90	p0	p50	p90
$C, F = 4, G, N = 32, H = 56, KH, KW = 3,$ $W = 56$	TC	696	701	704	435	440	443
	Caffe2	1,590	1,609	1,621	879	888	896
$C, F = 8, G, N = 32, H = 28, KH, KW = 3,$ $W = 28$	TC	574	576	578	269	270	272
	Caffe2	640	653	692	613	650	660
$C, F = 16, G, N = 32, H = 14, KH, KW = 3,$ $W = 14$	TC	265	272	276	274	284	287
	Caffe2	440	474	510	377	383	397
$C, F = 32, G, N = 32, H = 7, KH, KW = 3,$ $W = 7$	TC	463	481	491	259	260	264
	Caffe2	456	461	469	367	388	394
Group Normalization		p0	p50	p90	p0	p50	p90
$C = 512, G = 32, H = 12, N = 4, W = 12$	TC	22	23	24	32	33	35
	Caffe2	37	38	40	33	34	35
$C = 512, G = 32, H = 48, N = 32, W = 48$	TC	1,285	1,290	1,294	593	597	601
	Caffe2	1,814	1,819	1,823	865	869	871
tmm		p0	p50	p90	p0	p50	p90
$K = 32, M = 128, N = 256$	TC	15	15	15	15	16	17
	Caffe2	18	19	20	31	31	32
$K = 1{,}024, M = 128, N = 1{,}024$	TC	318	334	344	181	189	192
	Caffe2	55	58	64	89	90	91
$K = 4{,}096, M = 128, N = 16{,}384$	TC	17,168	17,209	17,270	7,937	8,004	8,096
	Caffe2	2,254	2,388	2,590	1,360	1,378	1,419

We also experimented with writing a single fused TC but performance degraded. This is mostly due to kernels requiring substantially different grid configurations, which makes their fusion unprofitable. A larger, graph-level compiler that decides on TC function granularity, informed by the TC mapper and the autotuner, is necessary to automate this decision process but is left for future work.

Production Model. The kernels `1LUT`, `2LUT`, `MLP1`, and `MLP3` are the backbone of a low-latency production model used at scale in a large company and correspond to (1) reductions over a large lookup table embedding (10M rows); (2) fused reduction over two large lookup table embeddings (10M rows); (3) small size Multi-Layer Perceptron (fully connected, bias, ReLU); and (4) very small size,

three consecutive Multi-Layer Perceptrons. Despite LUT sizes, this model is essentially latency-bound. Existing libraries are often not tuned for low-latency regimes and tend to perform poorly.

On these examples, the need for reuse and instruction-level parallelism is dwarfed by the need to quickly load data from the memory into registers. TC is able to adapt to the problem size, leveraging reduction parallelism to hide memory latency. This results in large speedups over Caffe2 with cuBLAS 9.0.

Transposed Batch MatMul. This kernel is meant as a case study to characterize performance benefits and losses in the current flow, compared with reference libraries. For the sizes relevant to Factorization Machines [56], ($500 \times 26 \times 72 \times 26$), Nvidia Profiler reports the TC autotuned kernel taking 56μs on the Nvidia Quadro P6000 GPU (Pascal), while both Pytorch and Caffe2 resort to the specialized cuBLAS function `maxwell_sgemm_128x64_nn` that takes 87μs. Beyond architecture mismatch indicated in the function name, a detailed performance comparison demonstrates that TC executes 500 blocks of $26 \times 13 = 338$ threads, compared to 500 blocks of 128 threads for cuBLAS, reaching 81.8% occupancy instead of 23.6%. Additionally, the cuBLAS kernel shows a large number of predicated-off instructions due to the block size not matching the problem size. Occupancy is limited by the number of registers in both cases (11,264 vs. 15,360), but the TC version can be distributed over five blocks instead of four.[10] TC promotes all tensors to shared memory, saturating its bandwidth, whereas arithmetic instructions are the performance limiter for cuBLAS. Given the large occupancy metric, performance can be further increased by promoting one tensor to registers instead, trading off lower occupancy for reduced pressure on memory bandwidth.

Kronecker Recurrent Units. These have been recently proposed as a solution to drastically reduce model sizes by replacing the weights matrix of a linear layer by a Kronecker product of much smaller matrices [33]. In TC, a Kronecker product of three matrices is easily written as shown in the `kronecker3` function in Figure 6. The following table shows the running time in μs—or out of memory (OOM)—of a large matrix multiplication in Caffe2 and the equivalent Kronecker product of three matrices. Note that *the performance difference mostly comes from using a different algorithm.* While no specialized GPU library primitives exist for Kronecker recurrent units, TC's automatic flow enabled rapid exploration and reached unprecedented levels of performance, as shown in Table 2. Clearly, this benchmark deserves a deeper discussion of the space of possible TC derivations, including memory/computation/parallelism trade-offs falling outside the scope of this article. The `kronecker3` function is one such possible implementation that performed well for the three selected matrix shapes; it avoids redundant computation at the expense of storage (two tensors for intermediate computations).

WaveNet. WaveNet [64] is a popular model that enables generation of realistic sounding voices as highlighted at Google I/O 2018. We encoded a full WaveNet cell using a single TC function and compared our generated kernel with a WaveNet layer from PyTorch. This experiment uses a batch size of 1, residual and dilation channels of 32, and 256 skip channels. With TC, we observe performance improvements up to 4× on Volta, as shown in Table 2.

6 RELATED WORK

Despite decades of progress in optimizing and parallelizing compilation, programmers of computationally intensive applications complain about the poor performance of optimizing compilers, often missing the machine peak by orders of magnitude. Among the reasons for this state of affairs, one may cite the complexity and dynamic behavior of modern processors, domain knowledge required to prove optimizations' validity or profitability being unavailable to the compiler, program

[10] Mapping to blocks of 32×13 threads to obtain full warps results in 60μs execution time and only four blocks due to the higher number of registers per block.

Table 2. Algorithmic Exploration of Kronecker Recurrent Units and optimization of a WaveNet Cell

Algorithmic exploration		**Pascal**			**Volta**		
of Kronecker Recurrent Units		p0	p50	p90	p0	p50	p90
	TC Kronecker	272	280	285	200	206	212
$256 \times 16^3 \times 32^3$	Caffe2 MatMul	7,714	8,158	8,216	4,946	5,065	5,466
	TC Kronecker	1,334	1,349	1,365	998	1,004	1,011
$256 \times 16^3 \times 64^3$	Caffe2 MatMul	64,499	64,765	65,659	38,280	39,307	39,327
	TC Kronecker	4,408	4,447	4,472	4,794	4,815	5,106
$256 \times 16^3 \times 64 \times 128^2$	Caffe2 MatMul	OOM	OOM	OOM	OOM	OOM	OOM
WaveNet Cell		p0	p50	p90	p0	p50	p90
receptive field = $4K$, dilation = 1	TC	457	466	477	253	255	257
receptive field = $4K$, dilation = 1	PyTorch	549	576	790	563	571	594
receptive field = $4K$, dilation = 32	TC	353	365	375	138	139	140
receptive field = $4K$, dilation = 32	PyTorch	551	574	630	562	569	585

transformations whose profitability is difficult to assess, and the intrinsic difficulty of composing complex transformations, particularly in the case of computationally intensive loop nests [6, 26].

Several contributions have successfully addressed this issue, not by improving a general-purpose compiler, but through the design of application-specific program generators, a.k.a. active libraries [67]. Such generators often rely on feedback-directed optimization to select the best generation schema [60], as popularized by ATLAS [73] for dense matrix operations (and more recently BTO [10]) and FFTW [25] for the fast Fourier transform. Most of these generators use transformations previously proposed for traditional compilers, which fail to apply them for the aforementioned reasons. The SPIRAL project [54] made a quantum leap over these active libraries, operating on a domain-specific language (DSL) of digital signal processing formulas. Compilers for DSLs typically rely on domain-specific constructs to capture the intrinsic parallelism and locality of the application. Using such an approach, DSL compilers such as Halide [55] for image processing show impressive results. Its inputs are images defined on an infinite range while TC sets a fixed size for each dimension using range inference. This is better suited to ML applications, dominated by fixed-size tensors with higher temporal locality than 2-D images; it is also less verbose in the case of reductions and does not carry the syntactic burden of anticipating the declaration of stage names and free variables (Halide needs this as a C++ embedded DSL). OoLaLa [42] takes a similar approach for linear algebra, and TACO [37] and Simit [38] use a similar notation as TC but generate sparse matrix code for numerical solvers.

Following this trend in the context of deep neural networks, we not only design yet another DSL and compiler but propose a more generic code generation and optimization framework, bringing together decades of research in loop nest optimization and parallelization for high-performance computing. We also design the domain language to cover a variety of existing and emerging machine learning models. Our framework automates a combination of affine transformations involving hierarchical tiling, mapping, shifting, fusion, distribution, interchange, on either parametric or fully instantiated problems, that are not accessible to Halide [45, 55], Latte [63], or XLA's [28] representations of tensor operations.

The polyhedral framework is a powerful abstraction for the analysis and transformation of loop nests, and a number of tools and libraries have been developed to realize its benefits [12, 14, 22, 70, 77], including production compilers such as GCC (Graphite) and LLVM (Polly). Polyhedral techniques have also been tailored for domain-specific purposes. State-of-the-art examples include the PolyMage [46] DSL for image processing pipelines and the PENCIL approach to the

construction of parallelizing and compilers for DSLs [5, 9]. PolyMage is a clear illustration of the benefits of operating at a high level of abstraction, closer to the mathematics of the domain of interest: While GCC/Graphite and LLVM/Polly struggle to recover affine control and flow from low-level code, PolyMage natively captures patterns amenable to domain-specific optimization, such as stencil-specific overlapped tiling with or without recomputation, and cache-conscious fusion and tiling heuristics; it also offers a more productive programming experience for end-users. Interestingly, some techniques derived from PolyMage crossed out of polyhedral representations into Halide's automatic scheduler [45]. Back to deep learning frameworks, TVM extends Halide with recurrent (parallel scan) operators, support for ML accelerators, and tight integration with ML frameworks [17]. It also provides autotuning capabilities [18] and shares several engineering goals of TC, such as transparent ML framework integration. Much like PolyMage, TC implements optimizations well suited to the long-distance, non-uniform reuse patterns of deep learning models; these heuristics are not available in general-purpose compilers such as LLVM/Polly, Pluto, or PPCG, or semi-automatic frameworks such as Halide and TVM.

None of the aforementioned frameworks offer the complete transparency of TC's end-to-end compilation flow. TVM involves some level of manual intervention and/or feedback-directed optimization even for producing the most baseline GPU implementation, and it guarantees functional correctness for a subset of the scheduling primitives and tensor operations: e.g., convolutions can only be fused at the expense of introducing redundant computations or involving lower-level transformations that cannot be verified at compilation time. In addition, the balance between analytical objective functions (profitability heuristics) and feedback-directed autotuning is completely different: Halide and TVM auto-schedulers expose all scheduling decisions to the autotuner and infer most performance-related information from execution profiles, while TC's polyhedral flow reduces the autotuning space to a narrow set of optimization options and tile sizes.

TC also shares several motivations with Latte [63] and PlaidML [49], including a high-level domain-specific language and an end-to-end flow. TC provides elementwise access that is just as expressive when implementing custom layers, but unlike Latte it is more concise (thanks to type and shape inference), safer regarding static bound checking and graph connectivity, and more flexible by decoupling indexing from representation and layout choices. In addition, our framework implements more complex scheduling and mapping transformations than both Latte and PlaidML, some of which are essential to GPU targets with partitioned memory architectures. Unlike Latte, it is also designed as a JIT compilation library for seamless integration with deep learning frameworks. Unlike PlaidML, it is not limited to high-level patterns and rewrite rules, but captures complex affine transformations resulting from analytical modeling and autotuning. As a consequence, the TC compilation process takes generally more time than PlaidML, a price to pay for the ability to implement a wider range of optimizations.

Like TC, XLA [28] provides automatic shape and size inference, it may operate "in process" as a JIT compilation library, and it integrates into a production deep learning framework (TensorFlow, Caffe2 [29]). XLA shares many motivations with Latte, with a focus on integration and completeness of functionality rather than on the complexity of the optimizations and mapping strategies. Glow [58] is a recent domain-specific, retargetable compiler for PyTorch/Caffe2. It shares many of the motivations and capabilities of XLA, while emphasizing retargetability (CPUs as well GPUs and ML accelerators from multiple vendors) and the ability to differentiate, optimize, and lower operations and sub-graphs of operations within its own hierarchy of intermediate representations. It can leverage blackbox numerical libraries as well as generate custom vector processing kernels relying on LLVM. Our compiler design and algorithmic contributions would naturally fit XLA, Latte, or Glow, except for the following: TC remains independent from a specific computation graph while preserving tight integration with production frameworks; we did not use an embedded DSL

approach—keeping C++ as an interface for implementing optimization strategies only—isolating the user from complexity and debugging hurdles of embedded DSLs, and we leverage polyhedral techniques to factor out most of the optimization heavy-lifting, while XLA, Latte, and Glow resort to operation-specific emitters/lowering, optimization schemas, and heuristics.

Recently, R-Stream·TF [52] was presented as a proof-of-concept adaptation of the R-Stream polyhedral compiler to the automatic optimization of TensorFlow operators. Similarly to our approach, the generated code is wrapped as a custom operator of TensorFlow. The tool takes a computation graph as input and partitions it into sub-graphs amenable to tensor fusion, contraction, and layout optimization. R-Stream·TF also leverages the broadcast semantics of TensorFlow to maximize the operator's polymorphism w.r.t. input tensor dimension and shapes. This makes R-Stream·TF very aggressive in terms of static memory management and kernel partitioning. We made the more pragmatic choice of leaving most of these decisions to the level of tensor algebra, allowing a domain-specific optimizer or ML expert to rewrite declarative comprehensions into capacity- and layout-optimized ones. However, TC is more ambitious in its domain-specialization of affine scheduling and mapping, aiming for the generation of a single accelerated kernel, with heuristics adapted to the high-dimensional, non-uniform, long-distance reuse patterns of neural networks. The lack of algorithmic detail in the R-Stream·TF paper prevents us from comparing those affine transformation heuristics.

7 CONCLUSION

We presented and evaluated the first fully automatic, end-to-end flow, mapping a high-level mathematical language to high-performance accelerated GPU kernels. TC resembles the mathematical notation of a deep neural network and makes it easy to reason about, communicate, and to manually alter the computation and storage/computation trade-offs. Our flow leverages decades of progress in polyhedral compilation to implement the heavy-duty program transformations, analytical modeling of profitable optimizations, and code synthesis. It also implements domain-specific optimizations, code generation, autotuning with a compilation cache, and lightweight integration within Caffe2 and PyTorch. This unique combination differs from alternative proposals relying mainly on autotuning such as TVM [18], or pattern-based transformations such as PlaidML [49].

TC is capable of quickly synthesizing solid accelerated implementations that effectively lift bottlenecks in large training runs. In practice, such bottlenecks slow down ML research significantly, requiring substantial engineering efforts to be mobilized. Our contribution addresses this productivity gap; it brings more expressive power and control into the hands of domain experts, relieving ML frameworks' dependence on highly tuned vendor libraries without compromising performance. TC automates boilerplate optimization that has been replicated over the numerous deep learning frameworks and builds on a generic polyhedral intermediate representation and libraries shared with other domains (image processing, linear algebra) and general-purpose compilers (LLVM/Polly). Future work includes additional model-based domain-specific optimizations, CPU code generation, learning best mapping configurations automatically, automatic differentiation, interaction with the graph-level optimizer, and providing a path to emit a series of calls to a native library or hardware acceleration blocks.

ACKNOWLEDGMENTS

We are indebted to Léon Bottou for the constant support, feedback, and for providing much of the scientific inspiration and background for Tensor Comprehensions. We are also grateful to Cijo Jose and Moustapha Cissé, with whom we derived the Kronecker Research Unit experiment. We would also like to thank Antoine Bordes and Yann Lecun for making this collaboration possible and for their insights.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Vol. 16. 265–283.

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. 2006. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*. IEEE Computer Society. Washington, DC, 295–305. DOI: https://doi.org/10.1109/CGO.2006.37

[3] Corinne Ancourt and François Irigoin. 1991. Scanning polyhedra with DO loops. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 39–50.

[4] Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K. Panda. 2017. An in-depth performance characterization of CPU- and GPU-based DNN training on modern architectures. In *Proceedings of the Conference on Machine Learning on HPC Environments (MLHPC'17)*. ACM, New York, NY, Article 8, 8 pages. DOI: https://doi.org/10.1145/3146347.3146356

[5] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. V. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. 2015. PENCIL: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT'15)*. 138–149. DOI: https://doi.org/10.1109/PACT.2015.17

[6] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening polyhedral compiler's black box. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'16)*. ACM, New York, NY, 128–138. DOI: https://doi.org/10.1145/2854038.2854048

[7] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proceedings of the 22nd International Conference on Supercomputing (ICS'08)*. ACM, New York, NY, 225–234. DOI: https://doi.org/10.1145/1375527.1375562

[8] Cédric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*. IEEE Computer Society, Washington, DC, 7–16. DOI: https://doi.org/10.1109/PACT.2004.11

[9] Ulysse Beaugnon, Alexey Kravets, Sven van Haastregt, Riyadh Baghdadi, David Tweed, Javed Absar, and Anton Lokhmotov. 2014. VOBLA: A vehicle for optimized basic linear algebra. In *Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'14)*. ACM, New York, NY, 115–124. DOI: https://doi.org/10.1145/2597809.2597818

[10] Geoffrey Belter, E. R. Jessup, Ian Karlin, and Jeremy G. Siek. 2009. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*. ACM, New York, NY, Article 59, 12 pages. DOI: https://doi.org/10.1145/1654059.1654119

[11] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, Rajiv Gupta (Ed.), Vol. 6011, Lecture Notes in Computer Science.Springer, 283–303.

[12] Uday Bondhugula, Aravind Acharya, and Albert Cohen. 2016. The Pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests. *ACM Trans. on Prog. Lang. Syst.* 38, 3 (Apr. 2016), 12:1–12:32. DOI: https://doi.org/10.1145/2896389

[13] Uday Bondhugula, Sanjeeb Dash, Oktay Gunluk, and Lakshminarayanan Renganarayanan. 2010. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. IEEE, 343–352.

[14] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[15] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A Framework for Composing High-level Loop Transformations*. Technical Report 08-897, University of Southern California.

[16] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. Retrieved from: http://arxiv.org/abs/1512.01274.

[17] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, 578–594. Retrieved from https://www.usenix.org/conference/osdi18/presentation/chen.

[18] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 3389–3400.

[19] R. Collobert, K. Kavukcuoglu, and C. Farabet. 2012. Implementing neural networks efficiently. In *Neural Networks: Tricks of the Trade*, G. Montavon, G. Orr, and K.-R. Muller (Eds.). Springer.

[20] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for linear algebra and neural net computations on GPUs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL'18)*. ACM, New York, NY, 42–51. DOI: https://doi.org/10.1145/3211346.3211354

[21] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA'11)*. 365–376. DOI: https://doi.org/10.1145/2000064.2000108

[22] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Prog.* 21, 6 (1992), 389–420.

[23] Paul Feautrier and Christian Lengauer. 2011. Polyhedron model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer, 1581–1592.

[24] Basilio B. Fraguela, Ganesh Bikshandi, Jia Guo, María J. Garzarán, David Padua, and Christoph von Praun. 2012. Optimization techniques for efficient HTA programs. *Parallel Comput.* 38, 9 (2012), 465–484. DOI: https://doi.org/10.1016/j.parco.2012.05.002

[25] Matteo Frigo and Steven G. Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 3. IEEE, 1381–1384.

[26] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Prog.* 34, 3 (July 2006), 261–317. DOI: https://doi.org/10.1007/s10766-006-0012-3

[27] David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

[28] Google 2017. XLA: Domain-Specific Compiler for Linear Algebra to Optimize TensorFlow Computations. Retrieved from https://www.tensorflow.org/performance/xla.

[29] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch SGD: Training ImageNet in 1 hour. Retrieved from http://arxiv.org/abs/1706.02677.

[30] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly-performing polyhedral optimizations on a low-level intermediate representation. *Parallel Proc. Lett.* 22, 04 (2012), 1250010.

[31] John Hennessy. 2018. The Future of Computing. *Google I/O presentation*. Retrieved on May 2018 from https://www.youtube.com/watch?v=Azt8Nc-mtKM.

[32] François Irigoin and Remi Triolet. 1988. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 319–329.

[33] Cijo Jose, Moustpaha Cisse, and François Fleuret. 2017. Kronecker recurrent units. Retrieved from http://arxiv.org/abs/1705.10142.

[34] Norman P. Jouppi et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA'17)*. 1–12. DOI: https://doi.org/10.1145/3079856.3080246

[35] Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA.

[36] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. 2017. CUTLASS: Fast Linear Algebra in CUDA C++. Retrieved from https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/.

[37] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. DOI: https://doi.org/10.1145/3133901

[38] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. 2016. Simit: A language for physical simulation. *ACM Trans. Graph.* 35, 2, Article 20 (Mar. 2016), 21 pages. DOI: https://doi.org/10.1145/2866569

[39] Martin Kong and Louis-Noël Pouchet. 2018. A performance vocabulary for affine loop transformations. Retrieved from: http://arxiv.org/abs/1811.06043.

[40] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. 127–138. DOI: https://doi.org/10.1145/2462156.2462187

[41] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. 1989. Handwritten digit recognition with a back-propagation network. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS'89)*. 396–404. Retrieved from http://papers.nips.cc/paper/293-handwritten-digit-recognition-with-a-back-propagation-network.

[42] Mikel Luján, T. L. Freeman, and John R. Gurd. 2000. OoLALA: An object oriented analysis and design of numerical linear algebra. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*. ACM, New York, NY, 229–252. DOI: https://doi.org/10.1145/353171.353187

[43] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. 2011. *R-Stream Compiler*. Springer, Boston, MA, 1756–1765. DOI: https://doi.org/10.1007/978-0-387-09766-4_515

[44] Microsoft 2017. Microsoft Unveils Project Brainwave for Real-time AI. Retrieved from https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave.

[45] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling Halide image processing pipelines. *ACM Trans. Graph.* 35, 4 (July 2016), 83:1–83:11. DOI: https://doi.org/10.1145/2897824.2925952

[46] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic optimization for image processing pipelines. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 429–443. DOI: https://doi.org/10.1145/2694344.2694364

[47] Nvidia 2017. Deploying Deep Neural Networks with Nvidia TensorRT. Retrieved from https://devblogs.nvidia.com/parallelforall/deploying-deep-learning-nvidia-tensorrt.

[48] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques, Long Beach, CA*.

[49] PlaidML 2018. PlaidML. Retrieved from https://www.intel.ai/plaidml/#gs.bBu0cF8W.

[50] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop transformations: Convexity, pruning and optimization. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*.

[51] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'13)*. ACM, New York, NY, 29–38. DOI: https://doi.org/10.1145/2435264.2435273

[52] Benoit Pradelle, Benoit Meister, Muthu Baskaran, Jonathan Springer, and Richard Lethin. 2017. Polyhedral optimization of TensorFlow computation graphs. In *Proceedings of the 6th Workshop on Extreme-scale Programming Tools (ESPT'17, associated with SC'17)*.

[53] William Pugh and David Wonnacott. 1994. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Trans. Prog. Lang. Syst.* 16, 4 (July 1994), 1248–1278. DOI: https://doi.org/10.1145/183432.183525

[54] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. 2004. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *Int. J. High Perf. Comput. Appl.* 18, 1 (2004), 21–45. DOI: https://doi.org/10.1177/1094342004041291

[55] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[56] Steffen Rendle. 2010. Factorization machines. In *Proceedings of the IEEE International Conference on Data Mining (ICDM'10)*. IEEE Computer Society, Washington, DC, 995–1000. DOI: https://doi.org/10.1109/ICDM.2010.127

[57] Nvidia Research. [n.d.]. CUB Documentation. Version 1.8.0. Retrieved from: https://nvlabs.github.io/cub.

[58] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph lowering compiler techniques for neural networks. Retrieved from http://arxiv.org/abs/1805.00907.

[59] Mike Schroepfer. 2018. Day 2 Keynote. *Facebook f8 presentation at McEnery Convention Center, San Jose, CA*. Retrieved from https://developers.facebook.com/videos/f8-2018/f8-2018-day-2-keynote/.

[60] Michael D. Smith. 2000. Overcoming the challenges to feedback-directed optimization (keynote talk). In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO'00)*. ACM, New York, NY, 1–11. DOI: https://doi.org/10.1145/351397.351408

[61] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. Retrieved from http://arxiv.org/abs/1605.02688.

[62] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. 2010. GRAPHITE two years after: First lessons learned from real-world polyhedral compilation. In *Proceedings of the GCC Research Opportunities Workshop (GROW'10)*.

[63] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. 2016. Latte: A language, compiler, and runtime for elegant and efficient deep neural networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. ACM, New York, NY, 209–223. DOI: https://doi.org/10.1145/2908080.2908105

[64] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. 2016. WaveNet: A generative model for raw audio. Retrieved from http://arxiv.org/abs/1609.03499.

[65] Nicolas Vasilache, Jeff Johnson, Michaël Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2014. Fast convolutional nets with fbfft: A GPU performance evaluation. Retrieved from http://arxiv.org/abs/1412.7580.

[66] Nicolas Vasilache, Benoît Meister, Muthu Baskaran, and Richard Lethin. 2012. Joint scheduling and layout optimization to enable multi-level vectorization. In *Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques*.

[67] T. Veldhuizen and E. Gannon. 1998. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop: Object Oriented Methods for Interoperable Scientific and Engineering Computing*, Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons (Eds.). SIAM Press, 286–295.

[68] Sven Verdoolaege. 2010. Isl: An integer set library for the polyhedral model. In *Proceedings of the 3rd International Conference on Mathematical Software (ICMS'10)*. Springer, Berlin, 299–302. Retrieved from http://dl.acm.org/citation.cfm?id=1888390.1888455.

[69] Sven Verdoolaege. 2011. Counting affine calculator and applications. In *Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*. DOI: https://doi.org/10.13140/RG.2.1.2959.5601

[70] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (Jan. 2013), 54:1–54:23. DOI: https://doi.org/10.1145/2400682.2400713

[71] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule trees. In *Proceedings of the 4th Workshop on Polyhedral Compilation Techniques (IMPACT'14, Associated with HiPEAC'14)*.

[72] Sven Verdoolaege and Gerda Janssens. 2017. *Scheduling for PPCG*. Report CW 706. Department of Computer Science, KU Leuven, Leuven, Belgium. DOI: https://doi.org/10.13140/RG.2.2.28998.68169

[73] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'98)*. IEEE Computer Society, Washington, DC, 1–27. Retrieved from http://dl.acm.org/citation.cfm?id=509058.509096.

[74] Yuxin Wu and Kaiming He. 2018. Group normalization. Retrieved from http://arxiv.org/abs/1803.08494.

[75] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2016. Aggregated residual transformations for deep neural networks. Retrieved from http://arxiv.org/abs/1611.05431.

[76] Tomofumi Yuki and Sanjay Rajopadhye. 2013. *Parametrically Tiled Distributed Memory Parallelization of Polyhedral Programs*. Technical Report CS13-105. Colorado State University. 19 pages.

[77] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. 2018. Modeling the conflicting demands of parallelism and Temporal/Spatial locality in affine scheduling. In *Proceedings of the 27th International Conference on Compiler Construction*. ACM, 3–13.

Received February 2019; revised July 2019; accepted August 2019