usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

Tianqi Chen and Thierry Moreau, *University of Washington;* Ziheng Jiang, *University of Washington, AWS;* Lianmin Zheng, *Shanghai Jiao Tong University;* Eddie Yan, Haichen Shen, and Meghan Cowan, *University of Washington;* Leyuan Wang, *UC Davis, AWS;* Yuwei Hu, *Cornell;* Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy, *University of Washington*

https://www.usenix.org/conference/osdi18/presentation/chen

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

Tianqi Chen[1], Thierry Moreau[1], Ziheng Jiang[1,2], Lianmin Zheng[3], Eddie Yan[1]

Meghan Cowan[1], Haichen Shen[1], Leyuan Wang[4,2], Yuwei Hu[5], Luis Ceze[1], Carlos Guestrin[1], Arvind Krishnamurthy[1]
[1]Paul G. Allen School of Computer Science & Engineering, University of Washington

[2] AWS, [3]Shanghai Jiao Tong University, [4]UC Davis, [5]Cornell

Abstract

There is an increasing need to bring machine learning to a wide diversity of hardware devices. Current frameworks rely on vendor-specific operator libraries and optimize for a narrow range of server-class GPUs. Deploying workloads to new platforms – such as mobile phones, embedded devices, and accelerators (e.g., FPGAs, ASICs) – requires significant manual effort. We propose TVM, a compiler that exposes graph-level and operator-level optimizations to provide performance portability to deep learning workloads across diverse hardware back-ends. TVM solves optimization challenges specific to deep learning, such as high-level operator fusion, mapping to arbitrary hardware primitives, and memory latency hiding. It also automates optimization of low-level programs to hardware characteristics by employing a novel, learning-based cost modeling method for rapid exploration of code optimizations. Experimental results show that TVM delivers performance across hardware back-ends that are competitive with state-of-the-art, hand-tuned libraries for low-power CPU, mobile GPU, and server-class GPUs. We also demonstrate TVM's ability to target new accelerator back-ends, such as the FPGA-based generic deep learning accelerator. The system is open sourced and in production use inside several major companies.

1 Introduction

Deep learning (DL) models can now recognize images, process natural language, and defeat humans in challenging strategy games. There is a growing demand to deploy smart applications to a wide spectrum of devices, ranging from cloud servers to self-driving cars and embedded devices. Mapping DL workloads to these devices is complicated by the diversity of hardware characteristics, including embedded CPUs, GPUs, FPGAs, and ASICs (e.g., the TPU [21]). These hardware targets diverge in terms of memory organization, compute functional units, etc., as shown in Figure 1.

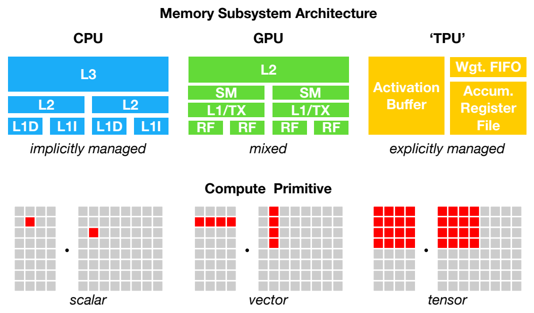


Figure 1: CPU, GPU and TPU-like accelerators require different on-chip memory architectures and compute primitives. This divergence must be addressed when generating optimized code.

Current DL frameworks, such as TensorFlow, MXNet, Caffe, and PyTorch, rely on a computational graph intermediate representation to implement optimizations, e.g., auto differentiation and dynamic memory management [3, 4, 9]. Graph-level optimizations, however, are often too high-level to handle hardware back-end-specific operator-level transformations. Most of these frameworks focus on a narrow class of server-class GPU devices and delegate target-specific optimizations to highly engineered and vendor-specific operator libraries. These operator-level libraries require significant manual tuning and hence are too specialized and opaque to be easily ported across hardware devices. Providing support in various DL frameworks for diverse hardware back-ends presently requires significant engineering effort. Even for supported back-ends, frameworks must make the difficult choice between: (1) avoiding graph optimizations that yield new operators not in the predefined operator library, and (2) using unoptimized implementations of these new operators.

To enable both graph- and operator-level optimizations for diverse hardware back-ends, we take a fundamentally different, end-to-end approach. *We built TVM, a compiler that takes a high-level specification of a deep learning program from existing frameworks and generates low-level optimized code for a diverse set of hardware back-ends.* To be attractive to users, TVM needs to offer performance competitive with the multitude of manually optimized operator libraries across diverse hardware back-ends. This goal requires addressing the key challenges described below.

Leveraging Specific Hardware Features and Abstractions. DL accelerators introduce optimized tensor compute primitives [1, 12, 21], while GPUs and CPUs continuously improve their processing elements. This poses a significant challenge in generating optimized code for a given operator description. The inputs to hardware instructions are multi-dimensional, with fixed or variable lengths; they dictate different data layouts; and they have special requirements for memory hierarchy. The system must effectively exploit these complex primitives to benefit from acceleration. Further, accelerator designs also commonly favor leaner control [21] and offload most scheduling complexity to the compiler stack. For specialized accelerators, the system now needs to generate code that explicitly controls pipeline dependencies to hide memory access latency – a job that hardware performs for CPUs and GPUs.

Large Search Space for Optimization Another challenge is producing efficient code without manually tuning operators. The combinatorial choices of memory access, threading pattern, and novel hardware primitives creates a huge configuration space for generated code (e.g., loop tiles and ordering, caching, unrolling) that would incur a large search cost if we implement black box auto-tuning. One could adopt a predefined cost model to guide the search, but building an accurate cost model is difficult due to the increasing complexity of modern hardware. Furthermore, such an approach would require us to build separate cost models for each hardware type.

TVM addresses these challenges with three key modules. (**1**) We introduce a *tensor expression language* to build operators and provide program transformation primitives that generate different versions of the program with various optimizations. This layer extends Halide [32]'s compute/schedule separation concept by also separating target hardware intrinsics from transformation primitives, which enables support for novel accelerators and their corresponding new intrinsics. Moreover, we introduce new transformation primitives to address GPU-related challenges and enable deployment to specialized accelerators. We can then apply different sequences of program transformations to form a rich space of valid programs for a given operator declaration. (**2**) We introduce an *automated program optimization framework* to find optimized tensor operators. The optimizer is guided by an ML-based cost model that adapts and improves as we collect more data from a hardware back-end. (**3**) On top of the automatic code generator, we introduce a *graph rewriter* that takes full advantage of high- and operator-level optimizations.

By combining these three modules, TVM can take model descriptions from existing deep learning frameworks, perform joint high- and low-level optimizations, and generate hardware-specific optimized code for back-ends, e.g., CPUs, GPUs, and FPGA-based specialized accelerators.

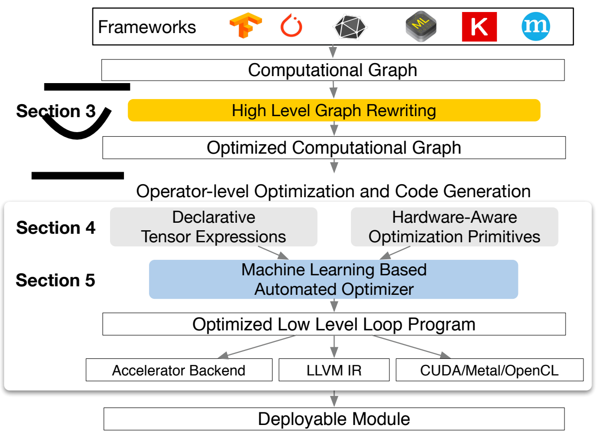
This paper makes the following contributions:

- We identify the major optimization challenges in providing performance portability to deep learning workloads across diverse hardware back-ends.
- We introduce novel schedule primitives that take advantage of cross-thread memory reuse, novel hardware intrinsics, and latency hiding.
- We propose and implement a machine learning based optimization system to automatically explore and search for optimized tensor operators.
- We build an end-to-end compilation and optimization stack that allows the deployment of deep learning workloads specified in high-level frameworks (including TensorFlow, MXNet, PyTorch, Keras, CNTK) to diverse hardware back-ends (including CPUs, server GPUs, mobile GPUs, and FPGA-based accelerators). The open-sourced TVM is in production use inside several major companies.

  We evaluated TVM using real world workloads on a server-class GPU, an embedded GPU, an embedded CPU, and a custom generic FPGA-based accelerator. Experimental results show that TVM offers portable performance across back-ends and achieves speedups ranging from 1.2× to 3.8× over existing frameworks backed by hand-optimized libraries.

2 Overview

This section describes TVM by using an example to walk through its components. Figure 2 summarizes execution steps in TVM and their corresponding sections in the paper. The system first takes as input a model from an existing framework and transforms it into a computational graph representation. It then performs high-level dataflow rewriting to generate an optimized graph. The operator-level optimization module must generate efficient code for each fused operator in this graph. Operators are specified in a declarative tensor expression lan-



Frameworks
Computational Graph
Section 3
High Level Graph Rewriting
Optimized Computational Graph
Operator-level Optimization and Code Generation
Section 4
Declarative Tensor Expressions
Hardware-Aware Optimization Primitives
Section 5
Machine Learning Based Automated Optimizer
Optimized Low Level Loop Program
Accelerator Backend
LLVM IR
CUDA/Metal/OpenCL
Deployable Module

Figure 2: System overview of TVM. The current stack supports descriptions from many deep learning frameworks and exchange formats, such as CoreML and ONNX, to target major CPU, GPU and specialized accelerators.

guage; execution details are unspecified. TVM identifies a collection of possible code optimizations for a given hardware target's operators. Possible optimizations form a large space, so we use an ML-based cost model to find optimized operators. Finally, the system packs the generated code into a deployable module.

End-User Example. In a few lines of code, a user can take a model from existing deep learning frameworks and call the TVM API to get a deployable module:

```
import tvm as t
# Use keras framework as example, import model
graph, params = t.frontend.from_keras(keras_model)
target = t.target.cuda()
graph, lib, params = t.compiler.build(graph, target, params)
```

This compiled runtime module contains three components: the final optimized computational graph (`graph`), generated operators (`lib`), and module parameters (`params`). These components can then be used to deploy the model to the target back-end:

```
import tvm.runtime as t
module = runtime.create(graph, lib, t.cuda(0))
module.set_input(**params)
module.run(data=data_array)
output = tvm.nd.empty(out_shape, ctx=t.cuda(0))
module.get_output(0, output)
```

TVM supports multiple deployment back-ends in languages such as C++, Java and Python. The rest of this paper describes TVM's architecture and how a system programmer can extend it to support new back-ends.

3 Optimizing Computational Graphs

Computational graphs are a common way to represent programs in DL frameworks [3, 4, 7, 9]. Figure 3 shows

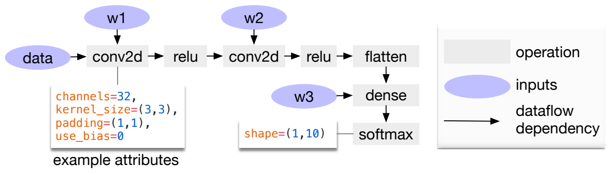


Figure 3: Example computational graph of a two-layer convolutional neural network. Each node in the graph represents an operation that consumes one or more tensors and produces one or more tensors. Tensor operations can be parameterized by attributes to configure their behavior (e.g., padding or strides).

an example computational graph representation of a two-layer convolutional neural network. The main difference between this high-level representation and a low-level compiler intermediate representation (IR), such as LLVM, is that the intermediate data items are large, multi-dimensional tensors. Computational graphs provide a global view of operators, but they avoid specifying how each operator must be implemented. Like LLVM IRs, a computational graph can be transformed into functionally equivalent graphs to apply optimizations. We also take advantage of shape specificity in common DL workloads to optimize for a fixed set of input shapes.

TVM exploits a computational graph representation to apply high-level optimizations: a node represents an operation on tensors or program inputs, and edges represent data dependencies between operations. It implements many graph-level optimizations, including: *operator fusion*, which fuses multiple small operations together; *constant-folding*, which pre-computes graph parts that can be determined statically, saving execution costs; a *static memory planning pass*, which pre-allocates memory to hold each intermediate tensor; and *data layout transformations*, which transform internal data layouts into back-end-friendly forms. We now discuss operator fusion and the data layout transformation.

Operator Fusion. Operator fusion combines multiple operators into a single kernel without saving the intermediate results in memory. This optimization can greatly reduce execution time, particularly in GPUs and specialized accelerators. Specifically, we recognize four categories of graph operators: (1) injective (one-to-one map, e.g., add), (2) reduction (e.g., sum), (3) complex-out-fusable (can fuse element-wise map to output, e.g., conv2d), and (4) opaque (cannot be fused, e.g., sort). We provide generic rules to fuse these operators, as follows. Multiple injective operators can be fused into another injective operator. A reduction operator can be fused with input injective operators (e.g., fuse scale and sum). Operators such as conv2d are complex-out-fusable, and we

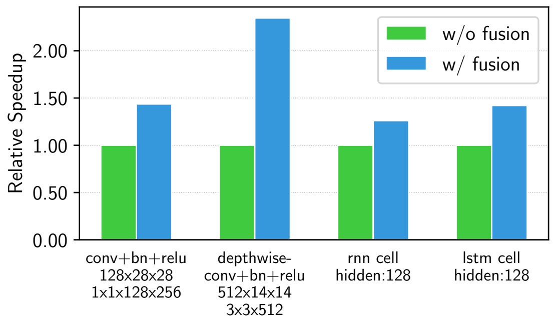


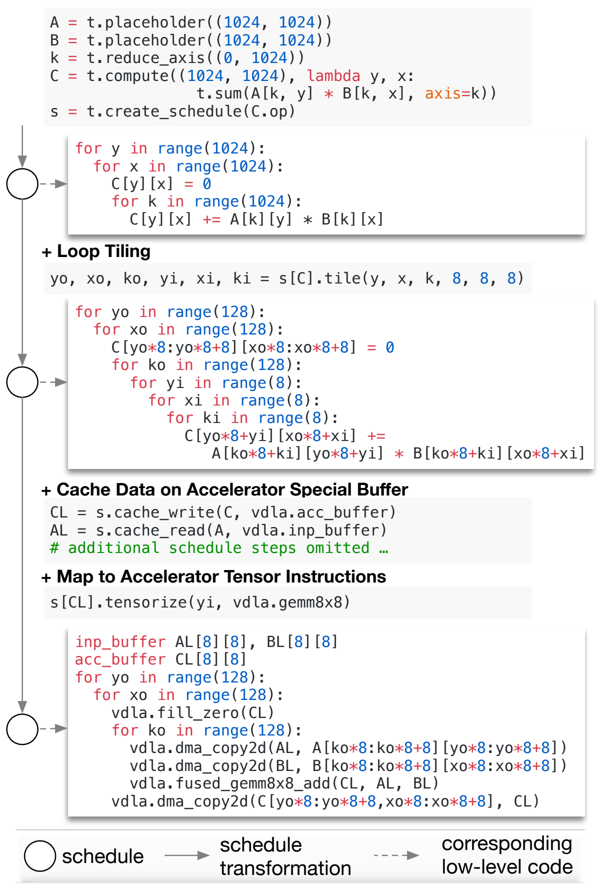
Figure 4: Performance comparison between fused and non-fused operations. TVM generates both operations. Tested on NVIDIA Titan X.

can fuse element-wise operators to its output. We can apply these rules to transform the computational graph into a fused version. Figure 4 demonstrates the impact of this optimization on different workloads. We find that fused operators generate up to a 1.2× to 2× speedup by reducing memory accesses.

Data Layout Transformation. There are multiple ways to store a given tensor in the computational graph. The most common data layout choices are column major and row major. In practice, we may prefer to use even more complicated data layouts. For instance, a DL accelerator might exploit 4×4 matrix operations, requiring data to be tiled into 4×4 chunks to optimize for access locality.

Data layout optimization converts a computational graph into one that can use better internal data layouts for execution on the target hardware. It starts by specifying the preferred data layout for each operator given the constraints dictated by memory hierarchies. We then perform the proper layout transformation between a producer and a consumer if their preferred data layouts do not match.

While high-level graph optimizations can greatly improve the efficiency of DL workloads, they are only as effective as what the operator library provides. Currently, the few DL frameworks that support operator fusion require the operator library to provide an implementation of the fused patterns. With more network operators introduced on a regular basis, the number of possible fused kernels can grow dramatically. This approach is no longer sustainable when targeting an increasing number of hardware back-ends since the required number of fused pattern implementations grows combinatorially with the number of data layouts, data types, and accelerator intrinsics that must be supported. It is not feasible to handcraft operator kernels for the various operations desired by a program and for each back-end. To this end, we next propose a code generation approach that can generate various possible implementations for a given model's operators.


```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024), lambda y, x:
              t.sum(A[k, y] * B[k, x], axis=k))
s = t.create_schedule(C.op)
```

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

+ Loop Tiling

```
yo, xo, ko, yi, xi, ki = s[C].tile(y, x, k, 8, 8, 8)
```

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
               A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

+ Cache Data on Accelerator Special Buffer

```
CL = s.cache_write(C, vdla.acc_buffer)
AL = s.cache_read(A, vdla.inp_buffer)
# additional schedule steps omitted …
```

+ Map to Accelerator Tensor Instructions

```
s[CL].tensorize(yi, vdla.gemm8x8)
```

```
inp_buffer AL[8][8], BL[8][8]
acc_buffer CL[8][8]
for yo in range(128):
  for xo in range(128):
    vdla.fill_zero(CL)
    for ko in range(128):
      vdla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
      vdla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
      vdla.fused_gemm8x8_add(CL, AL, BL)
    vdla.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
```

Figure 5: Example schedule transformations that optimize a matrix multiplication on a specialized accelerator.

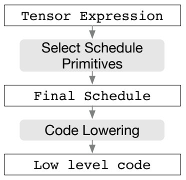
4 Generating Tensor Operations

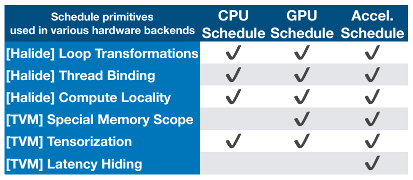
TVM produces efficient code for each operator by generating many valid implementations on each hardware back-end and choosing an optimized implementation. This process builds on Halide's idea of decoupling descriptions from computation rules (or *schedule optimizations*) [32] and extends it to support new optimizations (nested parallelism, tensorization, and latency hiding) and a wide array of hardware back-ends. We now highlight TVM-specific features.

4.1 Tensor Expression and Schedule Space

We introduce a tensor expression language to support automatic code generation. Unlike high-level computation graph representations, where the implementation of tensor operations is opaque, each operation is described in

not able to be seen through; not transparent





Schedule primitives used in various hardware backends	CPU Schedule	GPU Schedule	Accel. Schedule
[Halide] Loop Transformations	✓	✓	✓
[Halide] Thread Binding	✓	✓	✓
[Halide] Compute Locality	✓	✓	✓
[TVM] Special Memory Scope		✓	✓
[TVM] Tensorization	✓	✓	✓
[TVM] Latency Hiding			✓

Figure 6: TVM schedule lowering and code generation process. The table lists existing Halide and novel TVM scheduling primitives being used to optimize schedules for CPUs, GPUs and accelerator back-ends. Tensorization is essential for accelerators, but it can also be used for CPUs and GPUs. Special memory-scope enables memory reuse in GPUs and explicit management of on-chip memory in accelerators. Latency hiding is specific to TPU-like accelerators.

an index formula expression language. The following code shows an example tensor expression to compute transposed matrix multiplication:

```
m, n, h = t.var('m'), t.var('n'), t.var('h')
A = t.placeholder((m, h), name='A')
B = t.placeholder((n, h), name='B')
k = t.reduce_axis((0, h), name='k')
C = t.compute((m, n), lambda y, x:
               t.sum(A[k, y] * B[k, x], axis=k))
```

computing rule

result shape

Each compute operation specifies both the shape of the output tensor and an expression describing how to compute each element of it. Our tensor expression language supports common arithmetic and math operations and covers common DL operator patterns. The language does not specify the loop structure and many other execution details, and it provides flexibility for adding hardware-aware optimizations for various back-ends. Adopting the decoupled compute/schedule principle from Halide [32], we use a schedule to denote a specific mapping from a tensor expression to low-level code. Many possible schedules can perform this function.

We build a schedule by incrementally applying basic transformations (schedule primitives) that preserve the program's logical equivalence. Figure 5 shows an example of scheduling matrix multiplication on a specialized accelerator. Internally, TVM uses a data structure to keep track of the loop structure and other information as we apply schedule transformations. This information can then help generate low-level code for a given final schedule.

Our tensor expression takes cues from Halide [32], Darkroom [17], and TACO [23]. Its primary enhancements include support for the new schedule optimizations discussed below. To achieve high performance on many back-ends, we must support enough schedule primitives to cover a diverse set of optimizations on different hardware back-ends. Figure 6 summarizes the operation code generation process and schedule primi-

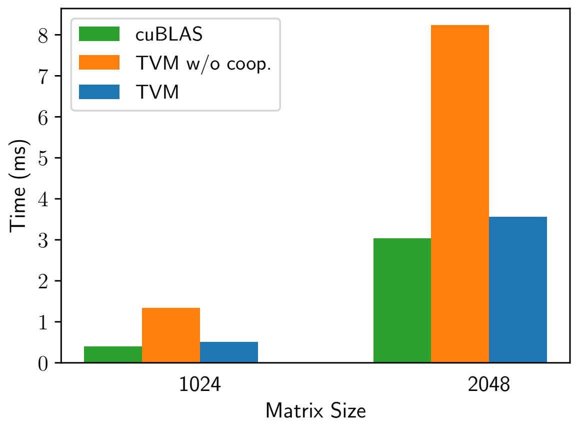


Figure 7: Performance comparison between TVM with and without cooperative shared memory fetching on matrix multiplication workloads. Tested on an NVIDIA Titan X.

tives that TVM supports. We reuse helpful primitives and the low-level loop program AST from Halide, and we introduce new primitives to optimize GPU and accelerator performance. The new primitives are necessary to achieve optimal GPU performance and essential for accelerators. CPU, GPU, TPU-like accelerators are three important types of hardware for deep learning. This section describes new optimization primitives for CPUs, GPUs and TPU-like accelerators, while section 5 explains how to automatically derive efficient schedules.

4.2 Nested Parallelism with Cooperation

Parallelism is key to improving the efficiency of compute-intensive kernels in DL workloads. Modern GPUs offer massive parallelism, requiring us to bake parallel patterns into schedule transformations. Most existing solutions adopt a model called *nested parallelism*, a form of fork–join. This model requires a parallel schedule primitive to parallelize a data parallel task; each task can be further recursively subdivided into subtasks to exploit the target architecture's multi-level thread hierarchy (e.g., thread groups in GPU). We call this model *shared-nothing nested parallelism* because one working thread cannot look at the data of its sibling within the same parallel computation stage.

An alternative to the shared-nothing approach is to fetch data cooperatively. Specifically, groups of threads can cooperatively fetch the data they all need and place it into a shared memory space. [1] This optimization can take advantage of the GPU memory hierarchy and en-

[1] Halide recently added shared memory support but without general memory scope for accelerators.

able data reuse across threads through shared memory regions. TVM supports this well-known GPU optimization using a schedule primitive to achieve optimal performance. The following GPU code example optimizes matrix multiplication.

```
for thread_group (by, bx) in cross(64, 64):
  for thread_item (ty, tx) in cross(2, 2):
    local CL[8][8] = 0
    shared AS[2][8], BS[2][8]
    for k in range(1024):
      for i in range(4):
        AS[ty][i*4+tx] = A[k][by*64+ty*8+i*4+tx]
      for each i in 0..4:
        BS[ty][i*4+tx] = B[k][bx*64+ty*8+i*4+tx]
      memory_barrier_among_threads()
      for yi in range(8):
        for xi in range(8):
          CL[yi][xi] += AS[yi] * BS[xi]
      for yi in range(8):
        for xi in range(8):
          C[yo*8+yi][xo*8+xi] = CL[yi][xi]
```

All threads cooperatively load AS and BS in different parallel patterns

Barrier inserted automatically by compiler

Figure 7 demonstrates the impact of this optimization. We introduce the concept of *memory scopes* to the schedule space so that a compute stage (AS and BS in the code) can be marked as shared. Without explicit memory scopes, automatic scope inference will mark compute stages as thread-local. The shared task must compute the dependencies of all working threads in the group. Additionally, memory synchronization barriers must be properly inserted to guarantee that shared loaded data is visible to consumers. Finally, in addition to being useful to GPUs, memory scopes let us tag special memory buffers and create special lowering rules when targeting specialized DL accelerators.

4.3 Tensorization

DL workloads have high arithmetic intensity, which can typically be decomposed into tensor operators like matrix-matrix multiplication or 1D convolution. These natural decompositions have led to the recent trend of adding tensor compute primitives [1, 12, 21]. These new primitives create both opportunities and challenges for schedule-based compilation; while using them can improve performance, the compilation framework must seamlessly integrate them. We dub this *tensorization*: it is analogous to vectorization for SIMD architectures but has significant differences. Instruction inputs are multi-dimensional, with fixed or variable lengths, and each has different data layouts. More importantly, we cannot support a fixed set of primitives since new accelerators are emerging with their own variations of tensor instructions. We therefore need an *extensible* solution.

We make tensorization extensible by separating the target hardware intrinsic from the schedule with a mechanism for tensor-intrinsic declaration. We use the same tensor expression language to declare both the behavior of each new hardware intrinsic and the lowering rule associated with it. The following code shows how to declare an 8×8 tensor hardware intrinsic.

```
w, x = t.placeholder((8, 8)), t.placeholder((8, 8))
k = t.reduce_axis((0, 8))
y = t.compute((8, 8), lambda i, j:
               t.sum(w[i, k] * x[j, k], axis=k))

def gemm_intrin_lower(inputs, outputs):
   ww_ptr = inputs[0].access_ptr(“r")
   xx_ptr = inputs[1].access_ptr("r")
   zz_ptr = outputs[0].access_ptr("w")
   compute = t.hardware_intrin("gemm8x8", ww_ptr, xx_ptr, zz_ptr)
   reset = t.hardware_intrin("fill_zero", zz_ptr)
   update = t.hardware_intrin("fuse_gemm8x8_add", ww_ptr, xx_ptr, zz_ptr)
   return compute, reset, update

gemm8x8 = t.decl_tensor_intrin(y.op, gemm_intrin_lower)
```

declare behavior

lowering rule to generate hardware intrinsics to carry out the computation

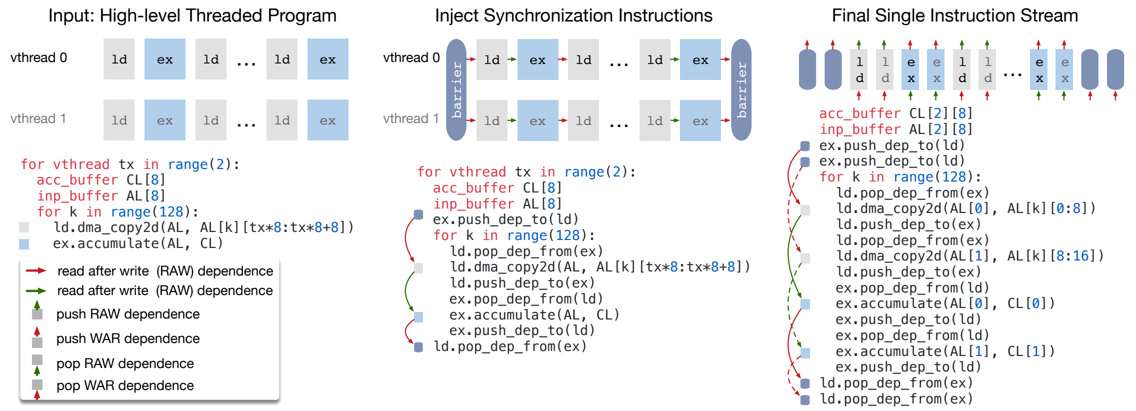
Additionally, we introduce a *tensorize* schedule primitive to replace a unit of computation with the corresponding intrinsics. The compiler matches the computation pattern with a hardware declaration and lowers it to the corresponding hardware intrinsic.

Tensorization decouples the schedule from specific hardware primitives, making it easy to extend TVM to support new hardware architectures. The generated code of tensorized schedules aligns with practices in high-performance computing: break complex operations into a sequence of micro-kernel calls. We can also use the *tensorize* primitive to take advantage of handcrafted micro-kernels, which can be beneficial in some platforms. For example, we implement ultra low precision operators for mobile CPUs that operate on data types that are one- or two-bits wide by leveraging a bit-serial matrix vector multiplication micro-kernel. This micro-kernel accumulates results into progressively larger data types to minimize the memory footprint. Presenting the micro-kernel as a tensor intrinsic to TVM yields up to a $1.5\times$ speedup over the non-tensorized version.

4.4 Explicit Memory Latency Hiding

Latency hiding refers to the process of overlapping memory operations with computation to maximize utilization of memory and compute resources. It requires different strategies depending on the target hardware back-end. On CPUs, memory latency hiding is achieved implicitly with simultaneous multithreading [14] or hardware prefetching [10, 20]. GPUs rely on rapid context switching of many warps of threads [44]. In contrast, specialized DL accelerators such as the TPU [21] usually favor leaner control with a *decoupled access-execute* (DAE) architecture [35] and offload the problem of fine-grained synchronization to software.

Figure 9 shows a DAE hardware pipeline that reduces runtime latency. Compared to a monolithic hardware design, the pipeline can hide most memory access overheads and almost fully utilize compute resources. To achieve higher utilization, the instruction stream must be augmented with fine-grained synchronization operations. Without them, dependencies cannot be enforced, leading to erroneous execution. Consequently, DAE hardware pipelines require fine-grained dependence enqueuing/dequeuing operations between the pipeline stages to guar-

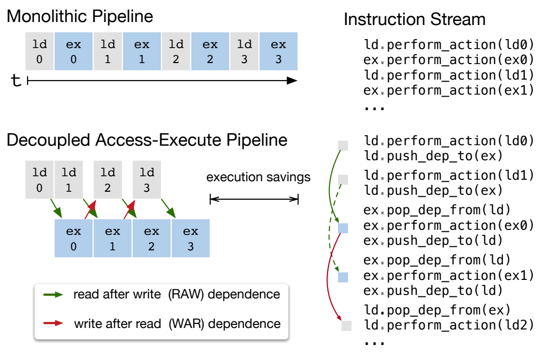

Input: High-level Threaded Program
vthread 0
vthread 1
ld
ex
Inject Synchronization Instructions
barrier
Final Single Instruction Stream
read after write (RAW) dependence
read after write (RAW) dependence
push RAW dependence
push WAR dependence
pop RAW dependence
pop WAR dependence

```
for vthread tx in range(2):
  acc_buffer CL[8]
  inp_buffer AL[8]
  for k in range(128):
    ld.dma_copy2d(AL, AL[k][tx*8:tx*8+8])
    ex.accumulate(AL, CL)
```

```
for vthread tx in range(2):
  acc_buffer CL[8]
  inp_buffer AL[8]
  ex.push_dep_to(ld)
  for k in range(128):
    ld.pop_dep_from(ex)
    ld.dma_copy2d(AL, AL[k][tx*8:tx*8+8])
    ld.push_dep_to(ex)
    ex.pop_dep_from(ld)
    ex.accumulate(AL, CL)
    ex.push_dep_to(ld)
  ld.pop_dep_from(ex)
```

```
acc_buffer CL[2][8]
inp_buffer AL[2][8]
ex.push_dep_to(ld)
ex.push_dep_to(ld)
for k in range(128):
  ld.pop_dep_from(ex)
  ld.dma_copy2d(AL[0], AL[k][0:8])
  ld.push_dep_to(ex)
  ld.pop_dep_from(ex)
  ld.dma_copy2d(AL[1], AL[k][8:16])
  ld.push_dep_to(ex)
  ex.pop_dep_from(ld)
  ex.accumulate(AL[0], CL[0])
  ex.push_dep_to(ld)
  ex.pop_dep_from(ld)
  ex.accumulate(AL[1], CL[1])
  ex.push_dep_to(ld)
ld.pop_dep_from(ex)
ld.pop_dep_from(ex)
```

Figure 8: TVM virtual thread lowering transforms a virtual thread-parallel program to a single instruction stream; the stream contains explicit low-level synchronizations that the hardware can interpret to recover the pipeline parallelism required to hide memory access latency.


Monolithic Pipeline
ld 0
ex 0
ld 1
ex 1
ld 2
ex 2
ld 3
ex 3
t
Instruction Stream
Decoupled Access-Execute Pipeline
execution savings
read after write (RAW) dependence
write after read (WAR) dependence

```
ld.perform_action(ld0)
ex.perform_action(ex0)
ld.perform_action(ld1)
ex.perform_action(ex1)
...
```

```
ld.perform_action(ld0)
ld.push_dep_to(ex)
ld.perform_action(ld1)
ld.push_dep_to(ex)
ex.pop_dep_from(ld)
ex.perform_action(ex0)
ex.push_dep_to(ld)
ex.pop_dep_from(ld)
ex.perform_action(ex1)
ex.push_dep_to(ld)
ld.pop_dep_from(ex)
ld.perform_action(ld2)
...
```

Figure 9: Decoupled Access-Execute in hardware hides most memory access latency by allowing memory and computation to overlap. Execution correctness is enforced by low-level synchronization in the form of dependence token enqueueing/dequeuing actions, which the compiler stack must insert in the instruction stream.

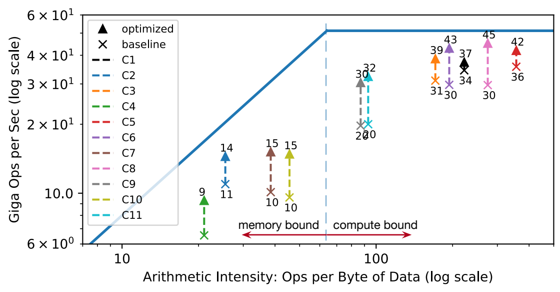


Figure 10: Roofline [47] of an FPGA-based DL accelerator running ResNet inference. With latency hiding enabled by TVM, performance of the benchmarks is brought closer to the roofline, demonstrating higher compute and memory bandwidth efficiency.

antee correct execution, as shown in Figure 9's instruction stream.

Programming DAE accelerators that require explicit low-level synchronization is difficult. To reduce the programming burden, we introduce a virtual threading scheduling primitive that lets programmers specify a high-level data parallel program as they would a hardware back-end with support for multithreading. TVM then automatically lowers the program to a single instruction stream with low-level explicit synchronization, as shown in Figure 8. The algorithm starts with a high-level multi-threaded program schedule and then inserts the necessary low-level synchronization operations to guarantee correct execution within each thread. Next, it interleaves operations of all virtual threads into a single instruction stream. Finally, the hardware recovers the available pipeline parallelism dictated by the low-level synchronizations in the instruction stream.

Hardware Evaluation of Latency Hiding. We now demonstrate the effectiveness of latency hiding on a custom FPGA-based accelerator design, which we describe in depth in subsection 6.4. We ran each layer of ResNet on the accelerator and used TVM to generate two schedules: one with latency hiding, and one without. The schedule with latency hiding parallelized the program with virtuals threads to expose pipeline parallelism and therefore hide memory access latency. Results are shown in Figure 10 as a roofline diagram [47]; roofline performance diagrams provide insight into how well a given system uses computation and memory resources for different benchmarks. Overall, latency hiding improved performance on all ResNet layers. Peak compute utilization increased from 70% with no latency hiding to 88% with latency hiding.

5 Automating Optimization

Given the rich set of schedule primitives, our remaining problem is to find optimal operator implementations for each layer of a DL model. Here, TVM creates a specialized operator for the specific input shape and layout associated with each layer. Such specialization offers significant performance benefits (in contrast to handcrafted code that would target a smaller diversity of shapes and layouts), but it also raises automation challenges. The system needs to choose the schedule optimizations – such as modifying the loop order or optimizing for the memory hierarchy – as well as schedule-specific parameters, such as the tiling size and the loop unrolling factor. Such combinatorial choices create a large search space of operator implementations for each hardware back-end. To address this challenge, we built an *automated schedule optimizer* with two main components: a schedule explorer that *proposes* promising new configurations, and a machine learning cost model that *predicts* the performance of a given configuration. This section describes these components and TVM's automated optimization flow (Figure 11).

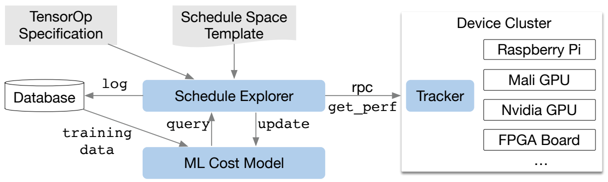


Figure 11: Overview of automated optimization framework. A schedule explorer examines the schedule space using an ML-based cost model and chooses experiments to run on a distributed device cluster via RPC. To improve its predictive power, the ML model is updated periodically using collected data recorded in a database.

Method Category	Data Cost	Model Bias	Need Hardware Info	Learn from History
Blackbox auto-tuning	high	none	no	no
Predefined cost model	none	high	yes	no
ML based cost model	**low**	**low**	**no**	**yes**

Table 1: Comparison of automation methods. Model bias refers to inaccuracy due to modeling.

5.1 Schedule Space Specification

We built a *schedule template specification API* to let a developer declare knobs in the schedule space. The template specification allows incorporation of a developer's domain-specific knowledge, as necessary, when specifying possible schedules. We also created a *generic master template for each hardware back-end* that automatically extracts possible knobs based on the computation description expressed using the tensor expression language. At a high level, we would like to consider as many configurations as possible and let the optimizer manage the selection burden. Consequently, the optimizer must search over *billions* of possible configurations for the real world DL workloads used in our experiments.

5.2 ML-Based Cost Model

One way to find the best schedule from a large configuration space is through blackbox optimization, i.e., auto-tuning. This method is used to tune high performance computing libraries [15, 46]. However, auto-tuning requires many experiments to identify a good configuration.

An alternate approach is to build a predefined cost model to guide the search for a particular hardware back-end instead of running all possibilities and measuring their performance. Ideally, a perfect cost model considers all factors affecting performance: memory access patterns, data reuse, pipeline dependencies, and threading patterns, among others. This approach, unfortunately, is burdensome due to the increasing complexity of modern hardware. Furthermore, every new hardware target requires a new (predefined) cost model.

We instead take a statistical approach to solve the cost modeling problem. In this approach, a schedule explorer proposes configurations that may improve an operator's performance. For each schedule configuration, we use an ML model that takes the lowered loop program as input and predicts its running time on a given hardware back-end. The model, trained using runtime measurement data collected during exploration, does not require the user to input detailed hardware information. We update the model periodically as we explore more configurations during optimization, which improves accuracy for other related workloads, as well. In this way, the quality of the ML model improves with more experimental trials. Table 1 summarizes the key differences between automation methods. ML-based cost models strike a balance between auto-tuning and predefined cost modeling and can benefit from the historical performance data of related workloads.

Machine Learning Model Design Choices. We must consider two key factors when choosing which ML model the schedule explorer will use: *quality* and *speed*. The schedule explorer queries the cost model frequently, which incurs overheads due to model prediction time and model refitting time. To be useful, these overheads must be smaller than the time it takes to measure per-

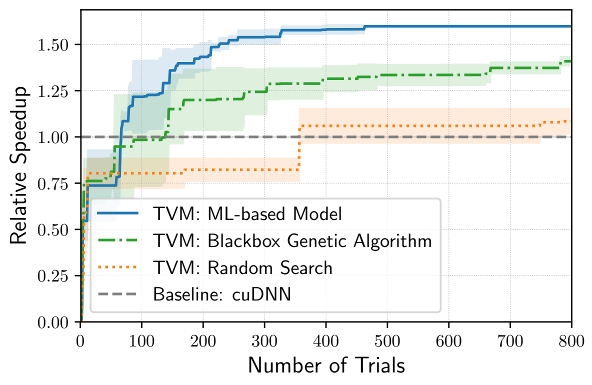
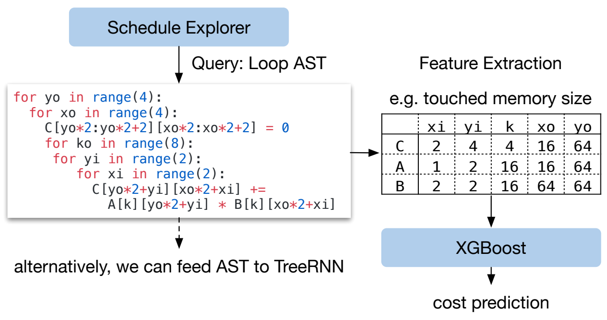


Figure 12: Comparison of different automation methods for a conv2d operator in ResNet-18 on TITAN X. The ML-based model starts with no training data and uses the collected data to improve itself. The Y-axis is the speedup relative to cuDNN. We observe a similar trend for other workloads.



```
for yo in range(4):
  for xo in range(4):
    C[yo*2:yo*2+2][xo*2:xo*2+2] = 0
    for ko in range(8):
     for yi in range(2):
        for xi in range(2):
          C[yo*2+yi][xo*2+xi] +=
            A[k][yo*2+yi] * B[k][xo*2+xi]
```

	xi	yi	k	xo	yo
C	2	4	4	16	64
A	1	2	16	16	64
B	2	2	16	64	64

Figure 13: Example workflow of ML cost models. XGBoost predicts costs based on loop program features. TreeRNN directly summarizes the AST.

formance on real hardware, which can be on the order of seconds depending on the specific workload/hardware target. This speed requirement differentiates our problem from traditional hyperparameter tuning problems, where the cost of performing measurements is very high relative to model overheads, and more expensive models can be used. In addition to the choice of model, we need to choose an objective function to train the model, such as the error in a configuration's predicted running time. However, since the explorer selects the top candidates based only on the relative order of the prediction (A runs faster than B), we need not predict the absolute execution times directly. Instead, we use a rank objective to predict the relative order of runtime costs.

We implement several types of models in our ML optimizer. We employ a *gradient tree boosting model* (based on XGBoost [8]), which makes predictions based on features extracted from the loop program; these features include the memory access count and reuse ratio of each memory buffer at each loop level, as well as a one-hot encoding of loop annotations such as "vectorize", "unroll", and "parallel." We also evaluate a *neural network model* that uses TreeRNN [38] to summarize the loop program's AST without feature engineering. Figure 13 summarizes the workflow of the cost models. We found that tree boosting and TreeRNN have similar predictive quality. However, the former performs prediction twice as fast and costs much less time to train. As a result, we chose gradient tree boosting as the default cost model in our experiments. Nevertheless, we believe that both approaches are valuable and expect more future research on this problem.

On average, the tree boosting model does prediction in 0.67 ms, thousands of times faster than running a real measurement. Figure 12 compares an ML-based optimizer to blackbox auto-tuning methods; the former finds better configurations much faster than the latter.

5.3 Schedule Exploration

Once we choose a cost model, we can use it to select promising configurations on which to iteratively run real measurements. In each iteration, the explorer uses the ML model's predictions to select a batch of candidates on which to run the measurements. The collected data is then used as training data to update the model. If no initial training data exists, the explorer picks random candidates to measure.

The simplest exploration algorithm enumerates and runs every configuration through the cost model, selecting the top-k predicted performers. However, this strategy becomes intractable with large search spaces. Instead, we run a parallel simulated annealing algorithm [22]. The explorer starts with random configurations, and, at each step, randomly walks to a nearby configuration. This transition is successful if cost decreases as predicted by the cost model. It is likely to fail (reject) if the target configuration has a higher cost. The random walk tends to converge on configurations that have lower costs as predicted by the cost model. Exploration states persist across cost model updates; we continue from the last configuration after these updates.

5.4 Distributed Device Pool and RPC

A *distributed device pool* scales up the running of on-hardware trials and enables fine-grained resource sharing among multiple optimization jobs. TVM implements a customized, RPC-based distributed device pool that enables clients to run programs on a specific type of device. We can use this interface to compile a program on the host compiler, request a remote device, run the

Name	Operator	H,W	IC,OC	K,S
C1	conv2d	224, 224	3,64	7, 2
C2	conv2d	56, 56	64,64	3, 1
C3	conv2d	56, 56	64,64	1, 1
C4	conv2d	56, 56	64,128	3, 2
C5	conv2d	56, 56	64,128	1, 2
C6	conv2d	28, 28	128,128	3, 1
C7	conv2d	28, 28	128,256	3, 2
C8	conv2d	28, 28	128,256	1, 2
C9	conv2d	14, 14	256,256	3, 1
C10	conv2d	14, 14	256,512	3, 2
C11	conv2d	14, 14	256,512	1, 2
C12	conv2d	7, 7	512,512	3, 1

Name	Operator	H,W	IC	K,S
D1	depthwise conv2d	112, 112	32	3, 1
D2	depthwise conv2d	112, 112	64	3, 2
D3	depthwise conv2d	56, 56	128	3, 1
D4	depthwise conv2d	56, 56	128	3, 2
D5	depthwise conv2d	28, 28	256	3, 1
D6	depthwise conv2d	28, 28	256	3, 2
D7	depthwise conv2d	14, 14	512	3, 1
D8	depthwise conv2d	14, 14	512	3, 2
D9	depthwise conv2d	7, 7	1024	3, 1

Table 2: Configurations of all conv2d operators in ResNet-18 and all depthwise conv2d operators in MobileNet used in the single kernel experiments. H/W denotes height and width, IC input channels, OC output channels, K kernel size, and S stride size. All ops use "SAME" padding. All depthwise conv2d operations have channel multipliers of 1.

function remotely, and access results in the same script on the host. TVM's RPC supports dynamic upload and runs cross-compiled modules and functions that use its runtime convention. As a result, the same infrastructure can perform a single workload optimization and end-to-end graph inference. Our approach automates the compile, run, and profile steps across multiple devices. This infrastructure is especially critical for embedded devices, which traditionally require tedious manual effort for cross-compilation, code deployment, and measurement.

6 Evaluation

TVM's core is implemented in C++ (∼50k LoC). We provide language bindings to Python and Java. Earlier sections of this paper evaluated the impact of several individual optimizations and components of TVM, namely, *operator fusion* in Figure 4, *latency hiding* in Figure 10, and the *ML-based cost model* in Figure 12. We now focus on an end-to-end evaluation that aims to answer the following questions:

- Can TVM optimize DL workloads over multiple platforms?
- How does TVM compare to existing DL frameworks (which rely on heavily optimized libraries) on each back-end?
- Can TVM support new, emerging DL workloads (e.g., depthwise convolution, low precision operations)?
- Can TVM support and optimize for new specialized accelerators?

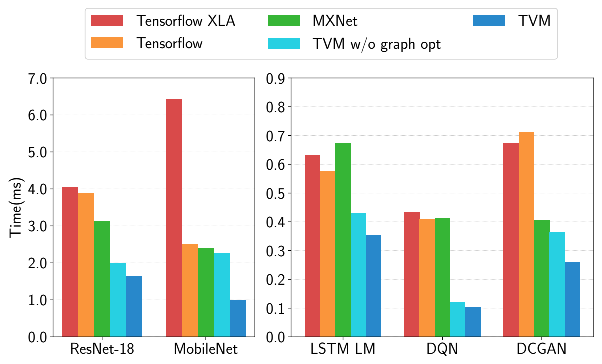


Figure 14: GPU end-to-end evaluation for TVM, MXNet, Tensorflow, and Tensorflow XLA. Tested on the NVIDIA Titan X.

To answer these questions, we evaluated TVM on four types of platforms: (1) a server-class GPU, (2) an embedded GPU, (3) an embedded CPU, and (4) a DL accelerator implemented on a low-power FPGA SoC. The benchmarks are based on real world DL inference workloads, including ResNet [16], MobileNet [19], the LSTM Language Model [48], the Deep Q Network (DQN) [28] and Deep Convolutional Generative Adversarial Networks (DCGAN) [31]. We compare our approach to existing DL frameworks, including MxNet [9] and TensorFlow [2], that rely on highly engineered, vendor-specific libraries. TVM performs end-to-end automatic optimization and code generation *without the need for an external operator library*.

6.1 Server-Class GPU Evaluation

We first compared the end-to-end performance of deep neural networks TVM, MXNet (v1.1), Tensorflow (v1.7), and Tensorflow XLA on an Nvidia Titan X. MXNet and Tensorflow both use cuDNN v7 for convolution operators; they implement their own versions of depthwise convolution since it is relatively new and not yet supported by the latest libraries. They also use cuBLAS v8 for matrix multiplications. On the other hand, Tensorflow XLA uses JIT compilation.

Figure 14 shows that TVM outperforms the baselines, with speedups ranging from 1.6× to 3.8× due to both joint graph optimization and the automatic optimizer, which generates high-performance fused opera-

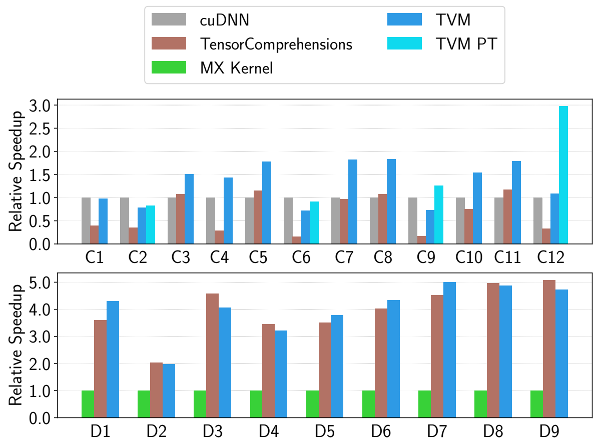


Figure 15: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in MobileNet. Tested on a TITAN X. See Table 2 for operator configurations. We also include a weight pre-transformed Winograd [25] for 3x3 conv2d (TVM PT).

tors. DQN's 3.8 x speedup results from its use of unconventional operators (4×4 conv2d, strides=2) that are not well optimized by cuDNN; the ResNet workloads are more conventional. TVM automatically finds optimized operators in both cases.

To evaluate the effectiveness of operator level optimization, we also perform a breakdown comparison for each tensor operator in ResNet and MobileNet, shown in Figure 15. We include TensorComprehension (TC, commit: ef644ba) [42], a recently introduced auto-tuning framework, as an additional baseline. [2] TC results include the best kernels it found in 10 generations × 100 population × 2 random seeds for each operator (i.e., 2000 trials per operator). 2D convolution, one of the most important DL operators, is heavily optimized by cuDNN. However, TVM can still generate better GPU kernels for most layers. Depthwise convolution is a newly introduced operator with a simpler structure [19]. In this case, both TVM and TC can find fast kernels compared to MXNet's handcrafted kernels. TVM's improvements are mainly due to its exploration of a large schedule space and an effective ML-based search algorithm.

6.2 Embedded CPU Evaluation

We evaluated the performance of TVM on an ARM Cortex A53 (Quad Core 1.2GHz). We used Tensorflow Lite (TFLite, commit: 7558b085) as our baseline system. Figure 17 compares TVM operators to hand-optimized

[2]According to personal communication [41], TC is not yet meant to be used for compute-bound problems. However, it is still a good reference baseline to include in the comparison.

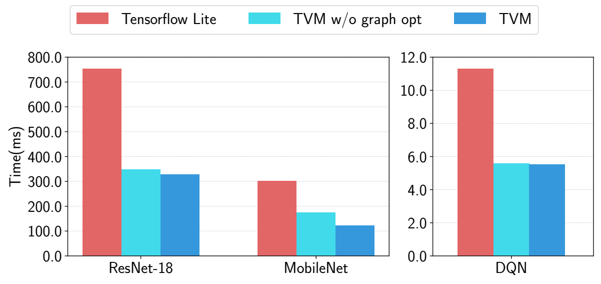


Figure 16: ARM A53 end-to-end evaluation of TVM and TFLite.

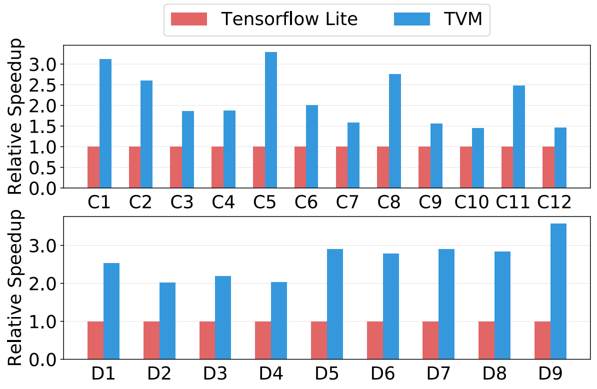


Figure 17: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in mobilenet. Tested on ARM A53. See Table 2 for the configurations of these operators.

ones for ResNet and MobileNet. We observe that TVM generates operators that outperform the hand-optimized TFLite versions for both neural network workloads. This result also demonstrates TVM's ability to quickly optimize emerging tensor operators, such as depthwise convolution operators. Finally, Figure 16 shows an end-to-end comparison of three workloads, where TVM outperforms the TFLite baseline.[3]

Ultra Low-Precision Operators We demonstrate TVM's ability to support ultra low-precision inference [13, 33] by generating highly optimized operators for fixed-point data types of less than 8-bits. Low-precision networks replace expensive multiplication with vectorized bit-serial multiplication that is composed of bitwise *and* popcount reductions [39]. Achieving efficient low-precision inference requires packing quantized data types into wider standard data types, such as `int8` or `int32`. Our system generates code that outperforms hand-optimized libraries from Caffe2 (commit: 39e07f7)

[3]DCGAN and LSTM results are not presented because they are not yet supported by the baseline.

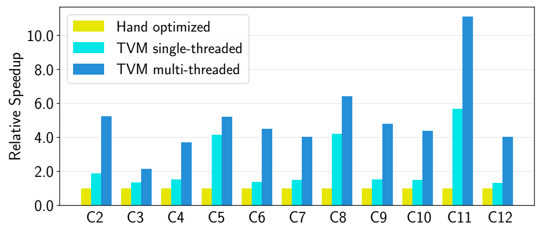


Figure 18: Relative speedup of single- and multi-threaded low-precision conv2d operators in ResNet. Baseline was a single-threaded, hand-optimized implementation from Caffe2 (commit: 39e07f7). C5, C3 are 1x1 convolutions that have less compute intensity, resulting in less speedup by multi-threading.

[39]. We implemented an ARM-specific *tensorization* intrinsic that leverages ARM instructions to build an efficient, low-precision matrix-vector microkernel.We then used TVM's automated optimizer to explore the scheduling space.

Figure 18 compares TVM to the Caffe2 ultra low-precision library on ResNet for 2-bit activations, 1-bit weights inference. Since the baseline is single threaded, we also compare it to a single-threaded TVM version. Single-threaded TVM outperforms the baseline, particularly for C5, C8, and C11 layers; these are convolution layers of kernel size 1×1 and stride of 2 for which the ultra low-precision baseline library is not optimized. Furthermore, we take advantage of additional TVM capabilities to produce a parallel library implementation that shows improvement over the baseline. In addition to the 2-bit+1-bit configuration, TVM can generate and optimize for other precision configurations that are unsupported by the baseline library, offering improved flexibility.

6.3 Embedded GPU Evaluation

For our mobile GPU experiments, we ran our end-to-end pipeline on a Firefly-RK3399 board equipped with an ARM Mali-T860MP4 GPU. The baseline was a vendor-provided library, the ARM Compute Library (v18.03). As shown in Figure 19, we outperformed the baseline on three available models for both float16 and float32 (DCGAN and LSTM are not yet supported by the baseline). The speedup ranged from $1.2\times$ to $1.6\times$.

6.4 FPGA Accelerator Evaluation

Vanilla Deep Learning Accelerator We now relate how TVM tackled accelerator-specific code generation on a generic inference accelerator design we prototyped on an FPGA. We used in this evaluation the Vanilla Deep

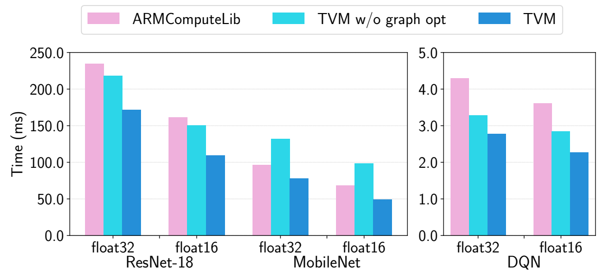


Figure 19: End-to-end experiment results on Mali-T860MP4. Two data types, float32 and float16, were evaluated.

Learning Accelerator (VDLA) – which distills characteristics from previous accelerator proposals [12, 21, 27] into a minimalist hardware architecture – to demonstrate TVM's ability to generate highly efficient schedules that can target specialized accelerators. Figure 20 shows the high-level hardware organization of the VDLA architecture. VDLA is programmed as a tensor processor to efficiently execute operations with high compute intensity (e.g, matrix multiplication, high dimensional convolution). It can perform load/store operations to bring blocked 3-dimensional tensors from DRAM into a contiguous region of SRAM. It also provides specialized on-chip memories for network parameters, layer inputs (narrow data type), and layer outputs (wide data type). Finally, VDLA provides explicit synchronization control over successive loads, computes, and stores to maximize the overlap between memory and compute operations.

Methodology. We implemented the VDLA design on a low-power PYNQ board that incorporates an ARM Cortex A9 dual core CPU clocked at 667MHz and an Artix-7 based FPGA fabric. On these modest FPGA resources, we implemented a 16×16 matrix-vector unit clocked at 200MHz that performs products of 8-bit values and accumulates them into a 32-bit register every cycle. The theoretical peak throughput of this VDLA design is about 102.4GOPS/s. We allocated 32kB of resources for activation storage, 32kB for parameter storage, 32kB for microcode buffers, and 128kB for the register file. These on-chip buffers are by no means large enough to provide sufficient on-chip storage for a single layer of ResNet and therefore enable a case study on effective memory reuse and latency hiding.

We built a driver library for VDLA with a C runtime API that constructs instructions and pushes them to the target accelerator for execution. Our code generation algorithm then translates the accelerator program to a series of calls into the runtime API. Adding the specialized accelerator back-end took ~2k LoC in Python.

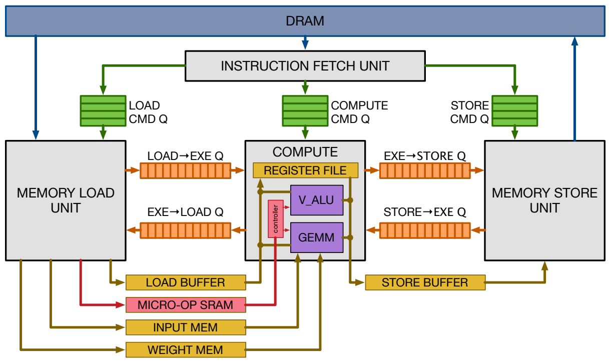


Figure 20: VDLA Hardware design overview.

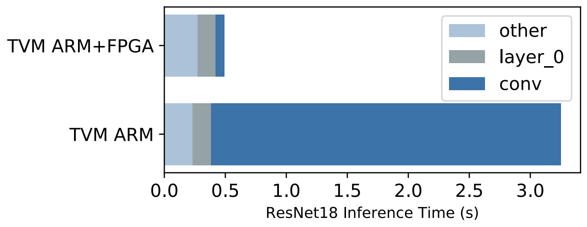


Figure 21: We offloaded convolutions in the ResNet workload to an FPGA-based accelerator. The grayed-out bars correspond to layers that could not be accelerated by the FPGA and therefore had to run on the CPU. The FPGA provided a 40x acceleration on offloaded convolution layers over the Cortex A9.

End-to-End ResNet Evaluation. We used TVM to generate ResNet inference kernels on the PYNQ platform and offloaded as many layers as possible to VDLA. We also used it to generate both schedules for the CPU only and CPU+FPGA implementation. Due to its shallow convolution depth, the first ResNet convolution layer could not be efficiently offloaded on the FPGA and was instead computed on the CPU. All other convolution layers in ResNet, however, were amenable to efficient offloading. Operations like residual layers and activations were also performed on the CPU since VDLA does not support these operations.

Figure 21 breaks down ResNet inference time into CPU-only execution and CPU+FPGA execution. Most computation was spent on the convolution layers that could be offloaded to VDLA. For those convolution layers, the achieved speedup was 40×. Unfortunately, due to Amdahl's law, the overall performance of the FPGA accelerated system was bottlenecked by the sections of the workload that had to be executed on the CPU. We envision that extending the VDLA design to support these other operators will help reduce cost even further. This FPGA-based experiment showcases TVM's ability to adapt to new architectures and the hardware intrinsics they expose.

7 Related Work

Deep learning frameworks [3, 4, 7, 9] provide convenient interfaces for users to express DL workloads and deploy them easily on different hardware back-ends. While existing frameworks currently depend on vendor-specific tensor operator libraries to execute their workloads, they can leverage TVM's stack to generate optimized code for a larger number of hardware devices.

High-level computation graph DSLs are a typical way to represent and perform high-level optimizations. Tensorflow's XLA [3] and the recently introduced DLVM [45] fall into this category. The representations of computation graphs in these works are similar, and a high-level computation graph DSL is also used in this paper. While graph-level representations are a good fit for high-level optimizations, they are too high level to optimize tensor operators under a diverse set of hardware back-ends. Prior work relies on specific lowering rules to directly generate low-level LLVM or resorts to vendor-crafted libraries. These approaches require significant engineering effort for each hardware back-end and operator-variant combination.

Halide [32] introduced the idea of separating computing and scheduling. We adopt Halide's insights and reuse its existing useful scheduling primitives in our compiler. Our tensor operator scheduling is also related to other work on DSL for GPUs [18, 24, 36, 37] and polyhedral-based loop transformation [6, 43]. TACO [23] introduces a generic way to generate sparse tensor operators on CPU. Weld [30] is a DSL for data processing tasks. We specifically focus on solving the new scheduling challenges of DL workloads for GPUs and specialized accelerators. Our new primitives can potentially be adopted by the optimization pipelines in these works.

High-performance libraries such as ATLAS [46] and FFTW [15] use auto-tuning to get the best performance. Tensor comprehension [42] applied black-box auto-tuning together with polyhedral optimizations to optimize CUDA kernels. OpenTuner [5] and existing hyper parameter-tuning algorithms [26] apply domain-agnostic search. A predefined cost model is used to automatically schedule image processing pipelines in Halide [29]. TVM's ML model uses effective domain-aware cost modeling that considers program structure. The based distributed schedule optimizer scales to a larger search space and can find state-of-the-art kernels on a large range of supported back-ends. More importantly, we provide an end-to-end stack that can take descriptions directly from DL frameworks and jointly optimize together with the graph-level stack.

Despite the emerging popularity of accelerators for deep learning [11, 21], it remains unclear how a compilation stack can be built to effectively target these devices. The VDLA design used in our evaluation provides a generic way to summarize the properties of TPU-like accelerators and enables a concrete case study on how to compile code for accelerators. Our approach could potentially benefit existing systems that compile deep learning to FPGA [34,40], as well. This paper provides a generic solution to effectively target accelerators via tensorization and compiler-driven latency hiding.

8 Conclusion

We proposed an end-to-end compilation stack to solve fundamental optimization challenges for deep learning across a diverse set of hardware back-ends. Our system includes automated end-to-end optimization, which is historically a labor-intensive and highly specialized task. We hope this work will encourage additional studies of end-to-end compilation approaches and open new opportunities for DL system software-hardware co-design techniques.

Acknowledgement

We would like to thank Ras Bodik, James Bornholt, Xi Wang, Tom Anderson and Qiao Zhang for their thorough feedback on earlier versions of this paper. We would also like to thank members of Sampa, SAMPL and Systems groups at the Allen School for their feedback on the work and manuscript. We would like to thank the anonymous OSDI reviewers, and our shepherd, Ranjita Bhagwan, for helpful feedbacks. This work was supported in part by a Google PhD Fellowship for Tianqi Chen, ONR award #N00014-16-1-2795, NSF under grants CCF-1518703, CNS-1614717, and CCF-1723352, and gifts from Intel (under the CAPA program), Oracle, Huawei and anonymous sources.

References

[1] NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Data Center GPU, 2017.

[2] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 265–283.

[4] Agarwal, A., Akchurin, E., Basoglu, C., Chen, G., Cyphers, S., Droppo, J., Eversole, A., Guenter, B., Hillebrand, M., Hoens, R., Huang, X., Huang, Z., Ivanov, V., Kamenev, A., Kranen, P., Kuchaiev, O., Manousek, W., May, A., Mitra, B., Nano, O., Navarro, G., Orlov, A., Padmilac, M., Parthasarathi, H., Peng, B., Reznichenko, A., Seide, F., Seltzer, M. L., Slaney, M., Stolcke, A., Wang, Y., Wang, H., Yao, K., Yu, D., Zhang, Y., and Zweig, G. An introduction to computational networks and the computational network toolkit. Tech. Rep. MSR-TR-2014-112, August 2014.

[5] Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques* (Edmonton, Canada, August 2014).

[6] Baghdadi, R., Beaugnon, U., Cohen, A., Grosser, T., Kruse, M., Reddy, C., Verdoolaege, S., Betts, A., Donaldson, A. F., Ketema, J., Absar, J., Haastregt, S. v., Kravets, A., Lokhmotov, A., David, R., and Hajiyev, E. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)* (Washington, DC, USA, 2015), PACT '15, IEEE Computer Society, pp. 138–149.

[7] Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., and Bengio, Y. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[8] Chen, T., and Guestrin, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2016), KDD '16, ACM, pp. 785–794.

[9] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., , and Zhang, Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems (LearningSys'15)* (2015).

[10] Chen, T.-F., and Baer, J.-L. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers 44*, 5 (May 1995), 609–623.

[11] Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., and Temam, O. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2014), MICRO-47, IEEE Computer Society, pp. 609–622.

[12] Chen, Y.-H., Emer, J., and Sze, V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2016), ISCA '16, IEEE Press, pp. 367–379.

[13] Courbariaux, M., Bengio, Y., and David, J. Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR abs/1511.00363* (2015).

[14] Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L., Stamm, R. L., and Tullsen, D. M. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro 17*, 5 (Sept 1997), 12–19.

[15] Frigo, M., and Johnson, S. G. Fftw: an adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on* (May 1998), vol. 3, pp. 1381–1384 vol.3.

[16] He, K., Zhang, X., Ren, S., and Sun, J. Identity mappings in deep residual networks. *arXiv preprint arXiv:1603.05027* (2016).

[17] Hegarty, J., Brunhaver, J., DeVito, Z., Ragan-Kelley, J., Cohen, N., Bell, S., Vasilyev, A., Horowitz, M., and Hanrahan, P. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph. 33*, 4 (July 2014), 144:1–144:11.

[18] Henriksen, T., Serup, N. G. W., Elsman, M., Henglein, F., and Oancea, C. E. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, ACM, pp. 556–571.

[19] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR abs/1704.04861* (2017).

[20] Jouppi, N. P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture* (May 1990), pp. 364–373.

[21] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA '17, ACM, pp. 1–12.

[22] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by simulated annealing. *Science 220*, 4598 (1983), 671–680.

[23] Kjolstad, F., Kamil, S., Chou, S., Lugato, D., and Amarasinghe, S. The tensor algebra compiler. *Proc. ACM Program. Lang. 1*, OOPSLA (Oct. 2017), 77:1–77:29.

[24] Klöckner, A. Loo.py: transformation-based code generation for GPUs and CPUs. In *Proceedings of ARRAY '14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming* (Edinburgh, Scotland., 2014), Association for Computing Machinery.

[25] Lavin, A., and Gray, S. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016* (2016), pp. 4013–4021.

[26] Li, L., Jamieson, K. G., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR abs/1603.06560* (2016).

[27] Liu, D., Chen, T., Liu, S., Zhou, J., Zhou, S., Teman, O., Feng, X., Zhou, X., and Chen, Y. Pudiannao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, pp. 369–381.

[28] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529.

[29] Mullapudi, R. T., Adams, A., Sharlet, D., Ragan-Kelley, J., and Fatahalian, K. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph. 35*, 4 (July 2016), 83:1–83:11.

[30] Palkar, S., Thomas, J. J., Narayanan, D., Shanbhag, A., Palamuttam, R., Pirk, H., Schwarzkopf, M., Amarasinghe, S. P., Madden, S., and Zaharia, M. Weld: Rethinking the interface between data-intensive applications. *CoRR abs/1709.06416* (2017).

[31] Radford, A., Metz, L., and Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[32] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. 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* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 519–530.

[33] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision* (2016), Springer, pp. 525–542.

[34] Sharma, H., Park, J., Mahajan, D., Amaro, E., Kim, J. K., Shao, C., Mishra, A., and Esmaeilzadeh, H. From high-level deep neural models to fpgas. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on* (2016), IEEE, pp. 1–12.

[35] Smith, J. E. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture* (Los Alamitos, CA, USA, 1982), ISCA '82, IEEE Computer Society Press, pp. 112–119.

[36] Steuwer, M., Remmelg, T., and Dubach, C. Lift: A functional data-parallel ir for high-performance gpu code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Piscataway, NJ, USA, 2017), CGO '17, IEEE Press, pp. 74–85.

[37] Sujeeth, A. K., Lee, H., Brown, K. J., Chafi, H., Wu, M., Atreya, A. R., Olukotun, K., Rompf, T., and Odersky, M. Optiml: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning* (USA, 2011), ICML'11, pp. 609–616.

[38] Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[39] TULLOCH, A., AND JIA, Y. High performance ultra-low-precision convolutions on mobile devices. *arXiv preprint arXiv:1712.02427* (2017).

[40] UMUROGLU, Y., FRASER, N. J., GAMBARDELLA, G., BLOTT, M., LEONG, P. H. W., JAHRE, M., AND VISSERS, K. A. FINN: A framework for fast, scalable binarized neural network inference. *CoRR abs/1612.07119* (2016).

[41] VASILACHE, N. personal communication.

[42] VASILACHE, N., ZINENKO, O., THEODORIDIS, T., GOYAL, P., DEVITO, Z., MOSES, W. S., VERDOOLAEGE, S., ADAMS, A., AND COHEN, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR abs/1802.04730* (2018).

[43] VERDOOLAEGE, S., CARLOS JUEGA, J., COHEN, A., IGNACIO GÓMEZ, J., TENLLADO, C., AND CATTHOOR, F. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim. 9*, 4 (Jan. 2013), 54:1–54:23.

[44] VOLKOV, V. *Understanding Latency Hiding on GPUs*. PhD thesis, University of California at Berkeley, 2016.

[45] WEI, R., ADVE, V., AND SCHWARTZ, L. Dlvm: A modern compiler infrastructure for deep learning systems. *CoRR abs/1711.03016* (2017).

[46] WHALEY, R. C., AND DONGARRA, J. J. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 1998), SC '98, IEEE Computer Society, pp. 1–27.

[47] WILLIAMS, S., WATERMAN, A., AND PATTERSON, D. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM 52*, 4 (Apr. 2009), 65–76.

[48] ZAREMBA, W., SUTSKEVER, I., AND VINYALS, O. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).