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.

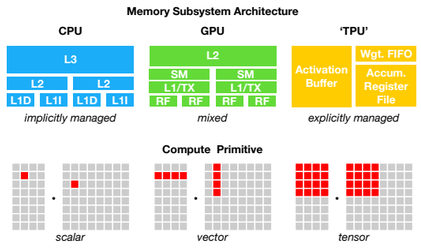


Figure 1: CPU, GPU and TPU-like accelerators require different on-chip memory architectures and compute primitives. This divergence must be addressed when generating optimized code.

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 optimiza-

TVM：一个自动 自动化端到端优化编译器用于深度学习

陈天奇[1]，Thierry Moreau[1]，蒋志恒[1,2]，郑连民[3]，Yan Eddie[1]

Meghan Cowan[1]，沈海晨[1]，王雷元[4,2]，胡宇伟[5]，Luis Ceze[1]，Carlos Guestrin[1]，ArvindKrishnamurthy[1]保罗·艾伦计算机科学与工程学院，华盛顿大学

[2] AWS，[3]上海交通大学，[4]UC Davis，[5]康奈尔大学

摘要

随着机器学习应用需求的增长，需要将其扩展到各种硬件设备上。当前框架依赖于特定供应商的算子库，并针对窄范围的伺服器级GPU进行优化。将工作负载部署到新平台——如手机、嵌入式设备和加速器（例如FPGA、ASIC）——需要大量手动工作。我们提出了TVM，一个编译器，它通过暴露图级和算子级的优化，为深度学习工作负载在不同硬件后端提供性能可移植性。TVM解决了深度学习特有的优化挑战，如高层算子融合、映射到任意硬件原语以及内存延迟隐藏。它还通过采用一种新颖的基于学习的成本建模方法来自动化低级程序的优化，以硬件特性，从而快速探索代码优化。实验结果表明，TVM在低功耗CPU、移动GPU和伺服器级GPU等硬件后端上提供的性能，与最先进的、手工调优的库具有竞争力。我们还展示了TVM针对新加速器后端的能力，例如基于FPGA的通用深度学习加速器。该系统已开源，并在多家主要公司中使用中。

1 简介

深度学习（DL）模型现在可以识别图像、处理自然语言，并在具有挑战性的策略游戏中击败人类。人们越来越需要将智能应用程序部署到各种设备上，从云服务器到自动驾驶汽车和嵌入式设备。将深度学习工作负载映射到这些设备上之所以复杂，是因为硬件特性的多样性，包括嵌入式CPU、GPU、FPGA和ASIC（例如TPU [21]）。这些硬件目标在内存组织、计算功能单元等方面存在差异，如图1所示。

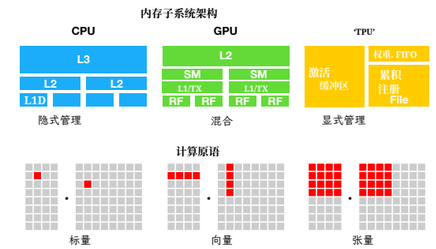


图1：CPU、GPU和TPU类似的加速器需要不同的片上内存架构和计算原语。在生成优化代码时必须解决这种差异。

内存组织、计算功能单元等方面存在差异，如图1所示。

当前的深度学习框架，如TensorFlow、MXNet、Caffe和PyTorch，依赖于计算图中间表示来实现优化，例如自动微分和动态内存管理 [3, 4, 9]。然而，图级别的优化通常过于高级，无法处理特定于硬件后端的算子级别转换。这些框架大多专注于服务器级GPU设备的一个狭窄类别，并将特定于目标的优化委托给高度工程化和特定于供应商的算子库。这些算子级库需要大量的手动调整，因此过于专业且不透明，难以跨硬件设备轻松移植。目前，在各种深度学习框架中为不同的硬件后端提供支持需要大量的工程工作。即使对于支持的后端，框架也必须在以下两者之间做出艰难的选择：(1)避免生成预定义算子库中不存在的新的算子的图优化，以及(2)使用这些新算子的未优化实现。

To 启用图级和算子级优化 miza-

tions 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-

针对不同的硬件后端，我们采用了一种根本不同的端到端方法。我们构建了*TVM*，这是一个编译器，它从现有框架中获取深度学习程序的高级规范，并为各种硬件后端生成低级优化代码。为了吸引用户，TVM需要提供与不同硬件后端中大量手动优化算子库具有竞争力的性能。这一目标需要解决下面描述的关键挑战。

利用特定硬件特性和抽象。深度学习加速器引入了优化的张量计算原语 [1, 12, 21], ，而 GPU 和 CPU 则持续改进其处理单元。这给针对给定算子描述生成优化代码带来了重大挑战。硬件指令的输入是多维的，具有固定或可变长度；它们决定了不同的数据布局；并且对内存层次结构有特殊要求。系统必须有效利用这些复杂原语以获得加速效果。此外，加速器设计通常也更青睐更精简的控制 [21] ，并将大部分调度复杂性卸载到编译器栈中。对于专用加速器，系统现在需要生成显式控制流水线依赖以隐藏内存访问延迟的代码——这项工作由硬件为 CPU 和 GPU 执行。

优化的大型搜索空间 另一个挑战是在不手动调整算子的情况下生成高效代码。内存访问、线程模式和新型硬件原语的组合选择为生成代码创造了巨大的配置空间（例如，循环瓦片和排序、缓存、展平），如果我们实现黑盒自动调优，这将导致巨大的搜索成本。我们可以采用预定义的成本模型来指导搜索，但由于现代硬件复杂性的不断增加，构建准确的成本模型很困难。此外，这种方法将要求我们为每种硬件类型构建单独的成本模型。

TVM 通过三个关键模块解决这些挑战。**(1)** 我们引入了一种 张量表达式语言 来构建算子，并提供程序转换原语，这些原语通过不同的优化生成程序的多个版本。这一层通过也将目标硬件内联与转换原语分离，扩展了 Halide [32] 的计算/调度分离概念，这支持了新型加速器及其对应的内联。此外，我们引入了新的转换原语来解决与 GPU 相关的挑战，并支持在专用加速器上部署。然后，我们可以应用不同序列的程序转换，以形成丰富的空间给定算子声明的有效程序。 **(2)** 我们引入一个 自动程序优化框架 来寻找优化的张量算子。优化器由一个 **基于机器学习的成本模型** 指导，该模型会随着我们从硬件后端收集更多数据而适应和改进。 **(3)** 在自动代码生成器的基础上，我们引入一个 图重写器，它充分利用了高层和算子级优化。

通过结合这三个模块，TVM可以从现有的深度学习框架中获取模型描述，执行高低级联合优化，并为后端（例如CPU、GPU和基于FPGA的专用加速器）生成硬件特定的优化代码。

本文做出了以下贡献：

- 我们识别了在为跨不同硬件后端的深度学习工作负载提供性能可移植性时面临的主要优化挑战。
- 我们引入了新的调度原语，这些原语利用了跨线程内存重用、新的硬件内建函数和延迟隐藏。
- 我们提出并实现了一个基于机器学习的优化系统，用于自动探索和搜索优化的张量算子。
- 我们构建了一个端到端的编译和优化栈，允许在高级框架（包括TensorFlow、MXNet、PyTorch、Keras、CNTK）中指定的深度学习工作负载部署到不同的硬件后端（包括CPU、服务器GPU、移动GPU和基于FPGA的加速器）。开源的TVM已在多家主要公司投入生产使用。

我们在服务器级GPU、嵌入式GPU、嵌入式CPU和基于自定义通用FPGA的加速器上使用真实世界工作负载评估了TVM。实验结果表明，TVM能够在不同后端上提供可移植的性能，并实现比现有基于手优化库的框架快1.2×到3.8×的速度提升。

2 概述

本节通过示例介绍 TVM 的组件。图 2 总结了 TVM 的执行步骤及其在论文中对应的章节。系统首先从现有框架中获取模型作为输入，并将其转换为计算图表示。然后执行高级数据流重写以生成优化后的图。算子级优化模块必须为该图中的每个融合算子生成高效代码。算子通过声明式张量表达式指定，

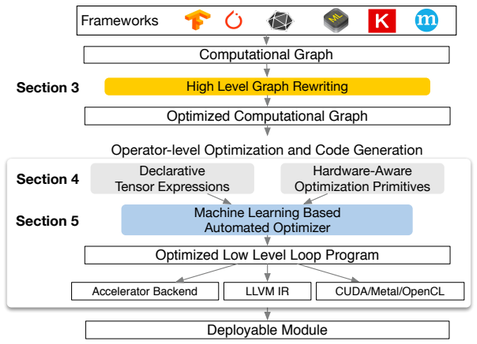


Figure 2: System overview of TVM. The current stack supports descriptions from many deep learning frameworks and exchange formats, such as CoreML and ONNX, to target major CPU, GPU and specialized accelerators.

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

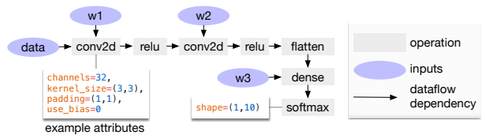


Figure 3: Example computational graph of a two-layer convolutional neural network. Each node in the graph represents an operation that consumes one or more tensors and produces one or more tensors. Tensor operations can be parameterized by attributes to configure their behavior (e.g., padding or strides).

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

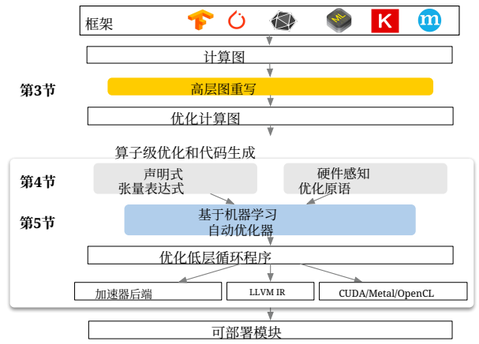


图2：TVM的系统概述。当前栈支持从许多深度学习框架和交换格式（如CoreML和ONNX）描述，以针对主要的CPU、GPU和专用加速器进行目标。

语言；执行细节未指定。TVM 识别出针对给定硬件目标的算子集合的可能代码优化。可能的优化形成一个很大的空间，因此我们使用基于机器学习的成本模型来找到优化的算子。最后，系统将生成的代码打包成一个可部署的模块。

用户示例。 几行代码内，用户可以从现有的深度学习框架中获取模型，并调用TVM API以获得可部署的模块：

```
import tvm as t# 以 keras 框架为例，导入模型graph, params =
t.frontend.from_keras(keras_model) target = t.target.cuda() graph, lib,
params = t.compiler.build(graph, target, params)
```

这个编译后的运行时模块包含三个组件：最终的优化计算图（graph）、生成的算子（lib）和模块参数（params）。这些组件随后可用于将模型部署到目标后端：

```
import tvm.runtime as t模块 =
runtime.create(graph, lib, t.cuda(0)) 模块.set_
input(**参数) 模块.run(数据=数据_数组) 输出 =
tvm.nd.empty(out_形状, 上下文=t.cuda(0)) 模块.get_输
出(0, 输出)
```

TVM 支持多种语言（如 C++、Java 和 Python）的部署后端。本文其余部分描述了 TVM 的架构以及系统程序员如何扩展它以支持新的后端。

3 优化计算图

计算图是表示深度学习框架中程序的一种常见方式 [3, 4, 7, 9]。图3展示了

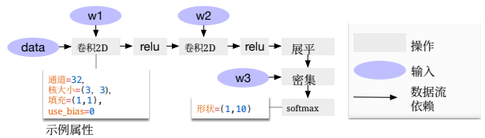


图 3：一个两层卷积神经网络的示例计算图。图中的每个节点代表一个操作，该操作消耗一个或多个张量并产生一个或多个张量。张量操作可以通过属性进行参数化以配置其行为（例如，填充或步长）。

一个双层卷积神经网络的计算图表示示例。这种高级表示与低级编译器中间表示（IR），例如LLVM，之间的主要区别在于中间数据项是大型多维张量。计算图提供了算子的全局视图，但它们避免了指定每个算子必须如何实现。与LLVM IR类似，计算图可以转换为功能等价的图以应用优化。我们还利用了常见深度学习工作负载中的形状特定性来针对一组固定的输入形状进行优化。

TVM 利用计算图表示来应用高级优化：节点表示张量或程序输入上的操作，边表示操作之间的数据依赖。它实现了许多图级优化，包括：**算子融合**，它将多个小操作融合在一起；常量折叠，它预先计算可以静态确定的图部分，节省执行成本；一个静态内存规划阶段，它预先分配内存以保存每个中间张量；以及 数据布局转换，它将内部数据布局转换为后端友好的形式。我们现在讨论算子融合和数据布局转换。

算子融合。 算子融合将多个算子组合成一个内核，而不会将中间结果保存在内存中。这种优化可以大大减少执行时间，特别是在 GPU 和专用加速器中。具体来说，我们识别了四种图算子类别：(1) 单射（一对一映射，例如 add），(2) 归约（例如 sum），(3) 复杂输出可融合（可以将其逐元素映射融合到输出，例如 conv2d），以及 (4) 不透明（无法融合，例如 sort）。我们提供了融合这些算子的通用规则，如下所示。多个单射算子可以融合成另一个单射算子。一个归约算子可以与输入单射算子融合（例如，融合 scale 和 sum）。像 conv2d 这样的算子是复杂输出可融合的，我们

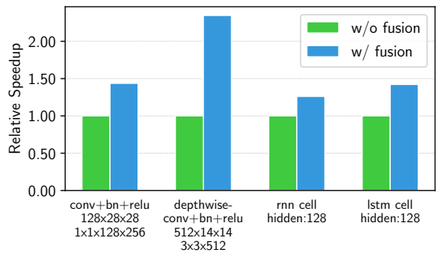


Figure 4: Performance comparison between fused and non-fused operations. TVM generates both operations. Tested on NVIDIA Titan X.

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

```
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)
```

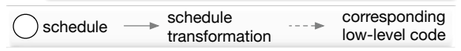


Figure 5: Example schedule transformations that optimize a matrix multiplication on a specialized accelerator.

this end, we next propose a code generation approach that can generate various possible implementations for a given model's operators.

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

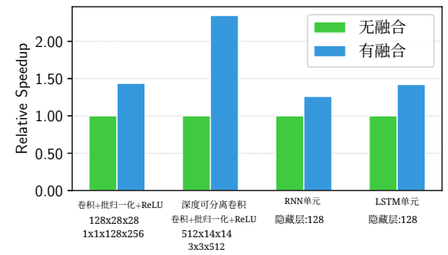


图4：融合操作与非融合操作的性能比较。TVM生成这两种操作。在NVIDIA Titan X上测试。

可以将逐元素算子融合到其输出。我们可以应用这些规则将计算图转换为融合版本。图 4 展示了这种优化对不同工作负载的影响。我们发现，融合算子通过减少内存访问，最多可产生 1.2× 到 2× 的加速比。

数据布局转换。 计算图中存储给定张量的方式有多种。最常见的数据布局选择是列主序和行主序。在实践中，我们可能更喜欢使用更复杂的数据布局。例如，深度学习加速器可能会利用 4×4 矩阵运算，需要将数据分块成 4×4 以优化访问局部性。

数据布局优化将计算图转换为可以在目标硬件上使用更好的内部数据布局执行的形式。它首先根据内存层次结构约束指定每个算子的首选数据布局。然后，如果生产者和消费者的首选数据布局不匹配，我们就会在它们之间执行适当的布局转换。

虽然高级图优化可以极大地提高深度学习工作负载的效率，但它们的效果仅取决于算子库提供的内容。目前，少数支持算子融合的深度学习框架需要算子库提供融合模式的实现。随着网络算子定期引入，可能的融合核的数量可能会急剧增长。当针对越来越多的硬件后端时，这种方法不再可持续，因为所需的融合模式实现数量会随着必须支持的数据布局、数据类型和加速器内联函数数量的增加而呈组合式增长。不可能为程序所需的各种操作和每个后端手工制作算子核。为此，我们接下来提出一种代码生成方法，该方法可以为给定模型的算子生成各种可能的实现。

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
核大小 = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024), lambda y, x:
                t求和(A[k, y] * B[k, x], axis=k))
s = t创建_调度(C.op)
```

```
对于 y 在 范围内(1024):
  对于 x 在 范围(1024):
    C[y][x] = 0
    for 核大小 in range(1024):
      C[y][x] += A[核大小][y] * B[核大小][x]
```

+ 循环分块

```
哟, 嗬, 科, 依, 希, 基 = s[C]。核大小(y, x, 核大小, 8, 8, 8)
```

```
对于 yo 在 范围(128):
  对于 xo 在 范围(128):
    C[悠*8:悠*8+8][悠*8:悠*8+8] = 0
    for核大小in range(128):
      对于每一个i 在(8)范围内:
        对于 xi 在 范围(8):
          for核大小in range(8):
            C[优*8+艺][学*8+习] +=
              A[核大小*8+核大小[步长*8+步长] * B[核大小*8+核大小][步长*8+步长]
```

+ 在加速器特殊缓冲区缓存数据

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

+ 映射到加速器张量指令

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

```
输入_缓冲区 AL[8][8], BL[8][8]
acc_buffer CL[8][8]
对于 yo 在 范围(128):
  对于 xo 在 范围(128):
    vdla.fill_zero(CL)
    for核大小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)
```

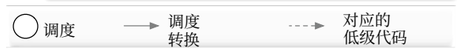


图 5：示例调度转换，优化在专用加速器上的矩阵乘法。

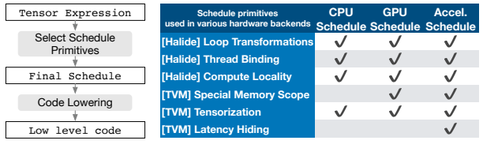
为此，我们接下来提出一种代码生成方法，该方法可以为给定模型的算子生成各种可能的实现。

4 生成 TensorOperations

TVM 通过在每个硬件后端生成许多有效的实现并选择优化的实现来为每个算子生成高效的代码。此过程基于 Halide 的描述与计算规则（或调度优化）[32]解耦的思想，并将其扩展以支持新的优化（嵌套并行、张量化、延迟隐藏）以及广泛的硬件后端。我们现在突出 TVM 特有的功能。

4.1 张量表达式和调度空间

我们介绍了一种张量表达式语言，用于支持自动代码生成。与高级计算图表示不同，其中张量操作的实现不透明，每个操作都用一个索引公式表达式语言描述。



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 primitives 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.

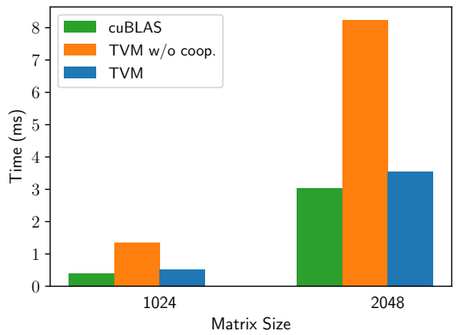


Figure 7: Performance comparison between TVM with and without cooperative shared memory fetching on matrix multiplication workloads. Tested on an NVIDIA Titan X.

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.

调度原语 用于各种硬件后端	CPU 调度	GPU 调度	加速 调度
[Halide] 循环变换	✓	✓	✓
[Halide] 线程绑定	✓	✓	✓
[Halide] 计算局部性	✓	✓	✓
[TVM] 特殊内存作用域		✓	✓
[TVM] 张量化	✓	✓	✓
[TVM] 延迟隐藏			✓

图6：TVM调度降低和代码生成过程。该表格列出了现有的Halide和新的TVM调度原语，用于优化CPU、GPU和加速器后端的调度。张量化对于加速器至关重要，但也可以用于CPU和GPU。特殊的内存范围能够在GPU中实现内存重用，并在加速器中显式管理片上内存。延迟隐藏是TPU类加速器的特定功能。

以下代码展示了一个计算转置矩阵乘法的张量表达式示例：

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

计算规则

结果形状

每个计算操作都指定了输出张量的形状，以及一个描述如何计算其每个元素的公式。我们的张量表达式语言支持常见的算术和数学操作，并涵盖了常见的深度学习算子模式。该语言不指定循环结构和其他许多执行细节，并为为各种后端添加硬件感知优化提供了灵活性。采用Halide [32], 的解耦计算/调度原则，我们使用调度来表示从张量表达式到低级代码的特定映射。许多可能的调度都可以执行此功能。

我们通过逐步应用保留程序逻辑等价性的基本转换（调度原语）来构建调度。图5展示了一个在专用加速器上调度矩阵乘法的示例。内部，TVM使用数据结构来跟踪循环结构和我们在应用调度转换时获取的其他信息。这些信息随后可以帮助为给定的最终调度生成低级代码。

我们的张量表达式借鉴了 Halide [32], Darkroom [17], 和 TACO [23]。其主要增强功能包括支持下面讨论的新调度优化。为了在许多后端上实现高性能，我们必须支持足够的调度原语，以涵盖不同硬件后端上的各种优化。图6总结了操作码生成过程和TVM支持的调度原语。

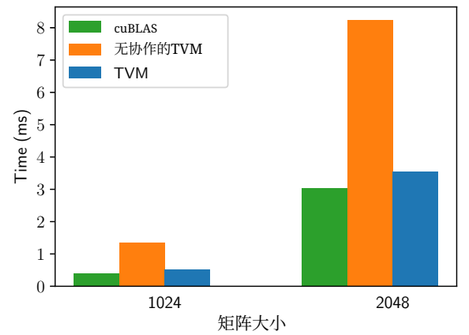


图7：TVM在矩阵乘法工作负载上使用和未使用协作共享内存获取的性能比较。测试于NVIDIA Titan X。

我们重用了Halide的有用原语和底层循环程序抽象语法树，并引入了新的原语来优化GPU和加速器的性能。新的原语对于实现最佳GPU性能是必要的，对于加速器也是必不可少的。CPU、GPU、TPU类加速器是深度学习的三种重要硬件类型。本节描述了CPU、GPU和TPU类加速器的新优化原语，而第5节解释了如何自动推导高效的调度。

4.2 带协作的嵌套并行

并行性是提高深度学习工作负载中计算密集型内核效率的关键。现代GPU提供了巨大的并行性，要求我们将并行模式嵌入调度转换中。大多数现有解决方案采用一种称为<样式id='1'>嵌套并行</样式>的模型，这是一种分叉-合并形式。该模型需要一个并行调度原语来并行化数据并行任务；每个任务可以进一步递归地细分为子任务，以利用目标架构的多级线程层次结构（例如GPU中的线程组）。我们称此模型为<样式id='3'>无共享嵌套并行</样式>，因为同一并行计算阶段中的一个工作线程无法查看其兄弟线程的数据。

共享无数据方法的替代方案是协同获取数据。具体来说，线程组可以协同获取它们都需要的数据，并将其放置到共享内存空间中。 [1] 这种优化可以利用GPU内存层次结构，并通过共享内存区域实现跨线程的数据重用。

[1] Halide最近增加了共享内存支持，但为加速器提供了通用的内存作用域。

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× 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-

TVM支持使用调度原语来实现这种众所周知的GPU优化，以获得最佳性能。以下GPU代码示例优化了矩阵乘法。

```
for 线程_组 (by, bx) in cross(64, 64):
  for 线程_项 (ty, tx) in cross(2, 2):
    局部 CL[8][8] = 0
    共享 AS[2][8], BS[2][8]
    for 核大小 in range(1024):
      对于 i 在 范围(4):
        自[动][化*4+文] = 自[动][化*64+动*8+化*4+文]
      对于 每个 i 在 0..4:
        BS[ty][i*4+tx] = B[k][bx*64+ty*8+i*4+tx]
      memory_屏障_among_线程()
      对于 yi 在 range(8):
        对于 xi 在 range(8):
          CL[步长][系] += AS[步长] * BS[系]
      对于每一个在范围(8):
        对于 xi 在 range(8):
          C[友*8+艺][晓*8+晓] = CL[艺][晓]
```

所有线程协作地在不同的并行模式中加载 AS 和 BS

并行模式

插入屏障自动地由编译器

图7展示了这种优化的影响。我们将内存作用域的概念引入调度空间，以便可以将计算阶段（代码中的AS和BS）标记为共享。如果没有显式的内存作用域，自动作用域推断会将计算阶段标记为线程本地。共享任务必须计算组中所有工作线程的依赖关系。此外，必须正确插入内存同步屏障，以保证共享加载的数据对消费者可见。最后，除了对GPU有用之外，内存作用域还允许我们标记特殊的内存缓冲区，并在针对专门的深度学习加速器时创建特殊的优化规则。

4.3 张量化

深度学习工作负载具有高算术强度，通常可以分解为矩阵乘法或1D卷积等张量算子。这些自然分解导致了近期添加张量计算原语 [1, 12, 21] 的趋势。这些新原语为基于调度的编译带来了机遇和挑战；虽然使用它们可以提高性能，但编译框架必须无缝地集成它们。我们将此称为 张量化：它类似于SIMD架构的向量化，但存在显著差异。指令输入是多维的，具有固定或可变长度，并且每个输入具有不同的数据布局。更重要的是，我们不能支持一组固定的原语，因为新的加速器正在涌现，它们有自己的张量指令变体。因此，我们需要一个 可扩展 的解决方案。

我们通过将目标硬件内建函数与调度分离，并引入张量内建函数声明机制，使张量化具有可扩展性。我们使用相同的张量表达式语言来声明每个新硬件内建函数的行为以及与之关联的下放规则。以下代码展示了如何声明一个 8×8 张量硬件内建函数。

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

def gemm_intrin_lower(输入, 输出):
  ww_ptr = 输入[0].access_ptr("r")
  xx_ptr = 输入[1].access_ptr("r")
  zz_指针 = 输出[0]. 访问_指针("w") 计算 = t硬件_内联("gemm8x8", ww_指针, xx_指针, zz_指针)

  重置 = t硬件_intrin("填充_零, zz_ptr)
  更新 = t硬件_intrin("融合_gemm8x8_加", ww_指针, xx_指针, zz_指针)
  返回计算、重置、更新

gemm8x8 = t声明_张量_intrin(y操作, gemm_intrin_降级)
```

声明行为

降级规则以生成硬件intrin以承载输出计算

此外，我们引入了 *tensorize* 调度原语，用对应的内建函数替换一个计算单元。编译器将计算模式与硬件声明进行匹配，并将其下放到对应的硬件内建函数。

张量化将调度与特定的硬件原语解耦，使得能够轻松扩展 TVM 以支持新的硬件架构。张量化调度的生成代码符合高性能计算中的实践：将复杂操作分解为一系列微内核调用。我们还可以使用 张量化 原语来利用手工制作的微内核，这在某些平台上可能有益。例如，我们通过利用位串行矩阵向量乘法微内核，为移动 CPU 实现了超低精度算子，这些算子操作的数据类型宽度为一比特或两比特。该微内核将结果累积到越来越大的数据类型中，以最小化内存占用。将微内核作为 TVM 的张量内禀呈现，相较于非张量化版本可带来高达 1.5× 的加速。

4.4 显式内存延迟隐藏

延迟隐藏是指将内存操作与计算重叠，以最大化内存和计算资源的利用率。它需要根据目标硬件后端采用不同的策略。在CPU上，内存延迟隐藏通过同时多线程 [14] 或硬件预取 [10, 20]隐式实现。GPU依赖于许多线程束的快速上下文切换 [44]。相比之下，TPU [21] 等专用深度学习加速器通常倾向于更精简的控制，采用<样式 id='7'>解耦访问-执行</样式>(DAE)架构 [35]，并将细粒度同步问题卸载到软件。

图9展示了一个数据并行硬件流水线，该流水线可降低运行时延迟。与单体硬件设计相比，该流水线能够隐藏大部分内存访问开销，并几乎完全利用计算资源。为达到更高的利用率，指令流必须增强细粒度同步操作。没有它们，依赖关系无法强制执行，导致错误执行。因此，数据并行硬件流水线需要在流水线阶段之间进行细粒度依赖入队/出队操作，以确保正确执行，如图9中的指令流所示。

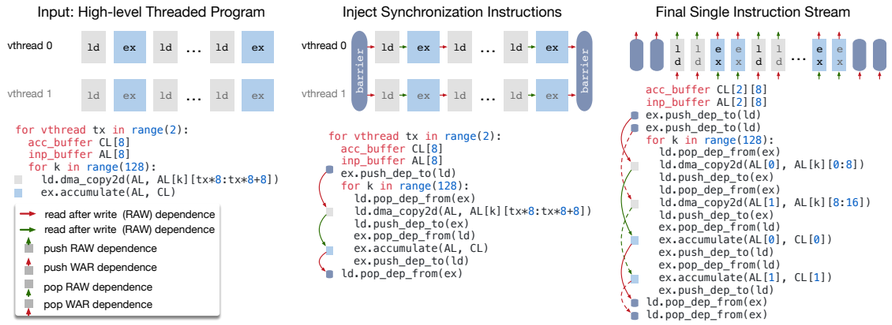


Figure 8: TVM virtual thread lowering transforms a virtual thread-parallel program to a single instruction stream; the stream contains explicit low-level synchronizations that the hardware can interpret to recover the pipeline parallelism required to hide memory access latency.

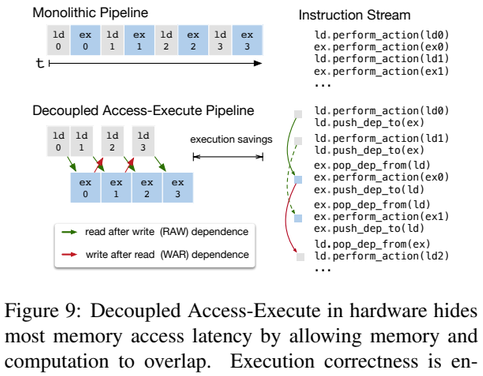


Figure 9: Decoupled Access-Execute in hardware hides most memory access latency by allowing memory and computation to overlap. Execution correctness is enforced by low-level synchronization in the form of dependence token enqueueing/dequeuing actions, which the compiler stack must insert in the instruction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

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

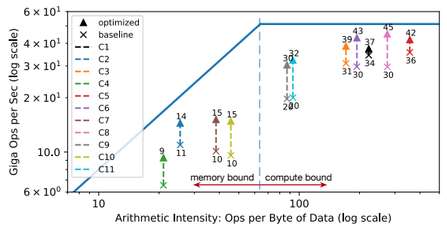


Figure 10: Roofline [47] of an FPGA-based DL accelerator running ResNet inference. With latency hiding enabled by TVM, performance of the benchmarks is brought closer to the roofline, demonstrating higher compute and memory bandwidth efficiency.

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.

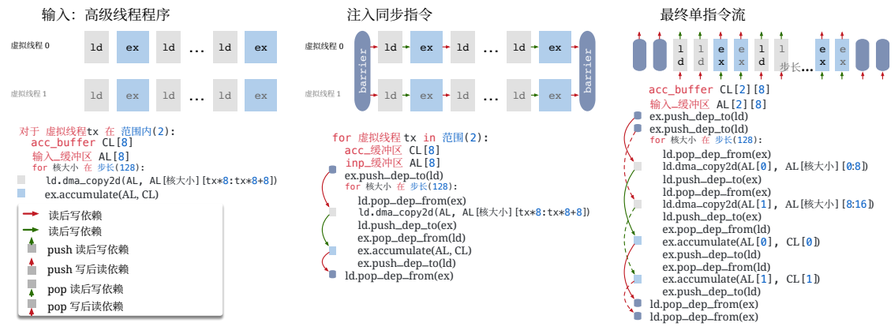


图8：TVM虚拟线程降低将虚拟线程并行程序转换为单指令流；该流包含硬件可解释的显式低级同步，以恢复隐藏内存访问延迟所需的流水线并行性。

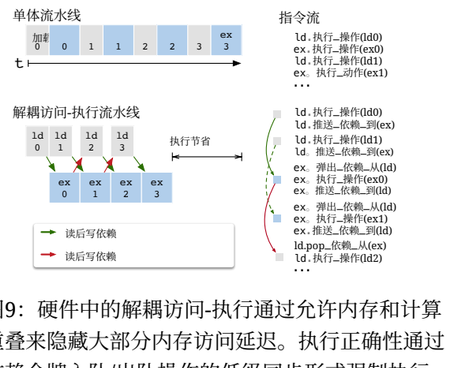


图9：硬件中的解耦访问-执行通过允许内存和计算重叠来隐藏大部分内存访问延迟。执行正确性通过依赖令牌入队/出队操作的低级同步形式强制执行，编译器栈必须在指令流中插入这些操作。

ante 正确执行，如图 9 的指令流所示。

编写需要显式低级同步的 DAE 加速器很困难。为了减轻编程负担，我们引入了一种虚拟线程调度原语，让程序员可以像编写支持多线程的硬件后端一样指定高级数据并行程序。TVM 然后自动将程序降低为具有低级显式同步的单个指令流，如图 8 所示。该算法从高级多线程程序调度开始，然后插入必要的低级同步操作，以确保每个线程内的正确执行。接下来，它将所有虚拟线程的操作交错到一个指令流中。最后，硬件恢复

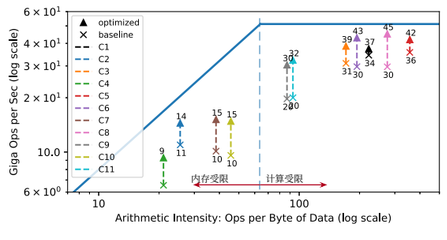


图10：基于FPGA的深度学习加速器运行ResNet推理时的Roofline [47]。通过TVM启用的延迟隐藏，基准测试的性能更接近Roofline，展示了更高的计算和内存带宽效率。

由指令流中的低级同步所决定的可用流水线并行性。

硬件延迟隐藏评估。我们现在在一个基于FPGA的自定义加速器设计上展示了延迟隐藏的有效性，该设计将在第6.4节中详细描述。我们在加速器上运行了ResNet的每一层，并使用TVM生成了两个调度：一个带有延迟隐藏，一个没有。带有延迟隐藏的调度使用虚拟线程并行化程序以暴露流水线并行性，从而隐藏内存访问延迟。结果以Roofline图的形式显示在图10中 [47]；Roofline性能图提供了关于给定系统如何在不同基准测试中使用计算和内存资源的洞察。总体而言，延迟隐藏提高了所有ResNet层的性能。峰值计算利用率从没有延迟隐藏的70%增加到带有延迟隐藏的88%。

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).

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 thread-

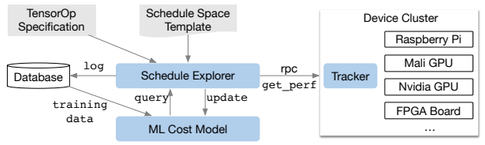


Figure 11: Overview of automated optimization framework. A schedule explorer examines the schedule space using an ML-based cost model and chooses experiments to run on a distributed device cluster via RPC. To improve its predictive power, the ML model is updated periodically using collected data recorded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

ing 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-

5 自动化优化

鉴于调度原语的丰富性，我们剩余的问题是找到深度学习模型每一层的最佳算子实现。在这里，TVM为与每一层相关的特定输入形状和布局创建一个专门的算子。这种专业化提供了显著的性能优势（与针对较小形状和布局多样性的手工艺代码相比），但它也提出了自动化挑战。系统需要选择调度优化——例如修改循环顺序或针对内存层次结构优化——以及调度特定参数，例如瓦片大小和循环展开因子。这种组合选择为每个硬件后端创造了大量的算子实现搜索空间。为了应对这一挑战，我们构建了一个自动调度优化器，它有两个主要组件：一个调度探索器，它提出 有希望的配置，以及一个机器学习成本模型，它预测 给定配置的性能。本节描述了这些组件以及TVM的自动优化流程（图11）。

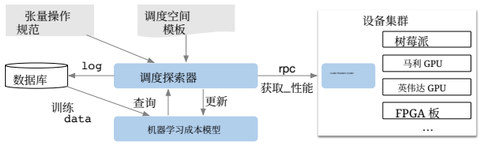


图11：自动化优化框架概述。调度探索器使用基于机器学习的成本模型检查调度空间，并通过远程过程调用RPC在分布式设备集群上选择要运行的实验。为了提高其预测能力，机器学习模型会定期使用记录在数据库中的收集数据更新。

5.1 调度空间规范

我们构建了一个 调度模板规范 *API*，让开发者能在调度空间中声明旋钮。模板规范允许在指定可能调度时，根据需要结合开发者的特定领域知识。我们还创建了一个 针对每个硬件后端的通用主模板，该模板能基于使用张量表达式语言表达的计算描述自动提取可能的旋钮。从宏观层面来看，我们希望尽可能多地考虑配置，并让优化器承担选择负担。因此，优化器必须为实验中使用的真实世界深度学习工作负载搜索数十亿种可能的配置。

5.2 基于机器学习的成本模型

一种从大型配置空间中找到最佳调度的方法是黑盒优化，即自动调优。该方法用于调优高性能计算库 [15, 46]。然而，自动调优需要大量实验来识别良好配置。

另一种方法是构建一个预定义的成本模型来指导针对特定硬件后端的搜索，而不是运行所有可能性并测量它们的性能。理想情况下，一个完美的成本模型可以

影响性能的所有因素：内存访问模式、数据重用、流水线依赖以及线程-

方法类别	Data Cost	模型 Bias	Need 硬件 Info	学习 from History
黑盒自动调优	high	none	no	no
预定义成本模型	none	high	yes	no
基于机器学习的成本模型	**low**	**low**	**no**	**yes**

表1：自动化方法比较。模型偏差是指由于建模导致的误差。

等。这种方法，不幸的是，由于现代硬件复杂性的增加而变得繁重。此外，每个新的硬件目标都需要一个新的（预定义的）成本模型。

我们采用统计方法来解决成本建模问题。在这种方法中，调度探索器会提出可能提升算子性能的配置。对于每个调度配置，我们使用一个机器学习模型，该模型以降级后的循环程序为输入，并预测其在给定硬件后端的运行时间。该模型使用在探索过程中收集的运行时测量数据训练，无需用户输入详细的硬件信息。在优化过程中，我们随着探索更多配置而定期更新模型，这提高了对其他相关工作负载的准确性。通过这种方式，机器学习模型的质量随着更多实验试验而提升。表1总结了自动化方法之间的主要差异。基于机器学习的成本模型在自动调优和预定义成本建模之间取得了平衡，并且可以从相关工作负载的历史性能数据中获益。

机器学习模型设计选择。 在选择调度探索器将使用的机器学习模型时，我们必须考虑两个关键因素：质量 和 速度。调度探索器频繁查询成本模型，这会由于模型预测时间和模型重新拟合时间而产生开销。为了有用，这些开销必须小于测量每个...的时间。

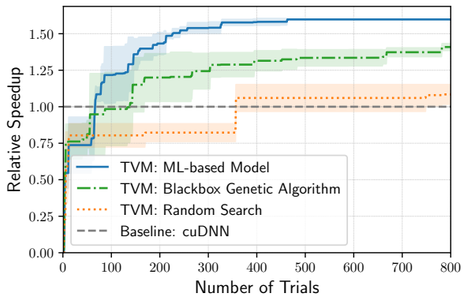


Figure 12: Comparison of different automation methods for a conv2d operator in ResNet-18 on TITAN X. The ML-based model starts with no training data and uses the collected data to improve itself. The Y-axis is the speedup relative to cuDNN. We observe a similar trend for other workloads.

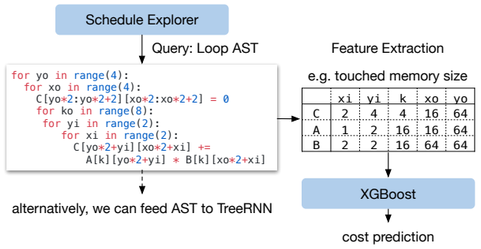


Figure 13: Example workflow of ML cost models. XGBoost predicts costs based on loop program features. TreeRNN directly summarizes the AST.

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

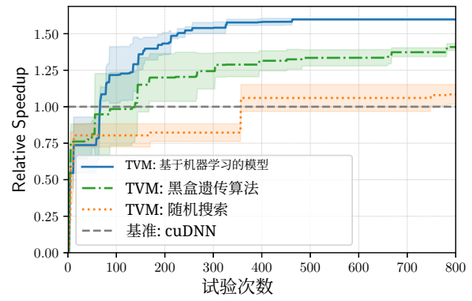


图12：在TITAN X上对ResNet-18中conv2d算子的不同自动化方法的比较。基于机器学习的模型从没有训练数据开始，并使用收集到的数据来提升自身。Y轴是相对于cuDNN的速度提升。我们观察到其他工作负载也有类似的趋势。

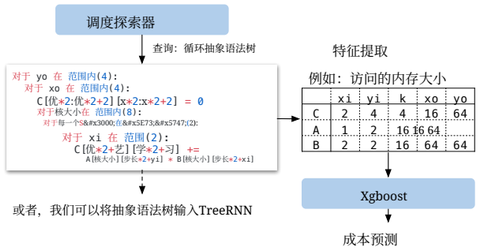


图13：机器学习成本模型的示例工作流程。XGBoost根据循环程序特征预测成本。TreeRNN直接总结抽象语法树。

性能在真实硬件上，这取决于具体的工作负载/硬件目标，可能需要秒级的时间。这种速度要求将我们的问题与传统的超参数调整问题区分开来，在后者中，执行测量的成本相对于模型开销非常高，并且可以使用更昂贵的模型。除了模型的选择之外，我们还需要选择一个目标函数来训练模型，例如配置预测运行时间的误差。然而，由于探索器仅根据预测的相对顺序选择顶级候选者（A比B运行得更快），因此我们不必直接预测绝对执行时间。相反，我们使用一个排序目标来预测运行成本之间的相对顺序。

我们在我们的机器学习优化器中实现了多种类型的模型。我们采用了一种<code>梯度树提升模型</code>(基于XGBoost [8])，该模型基于从循环程序中提取的特征进行预测；这些特征包括每个内存缓冲区在每个循环级别的内存访问计数和重用率，以及"向量化"、"展平"和"并行"等循环注释的独热编码。

我们还评估了一种使用TreeRNN [38] 来总结循环程序的抽象语法树而不进行特征工程的<code>神经网络模型</code>。图13总结了成本模型的流程。我们发现树提升和TreeRNN具有相似的预测质量。然而，前者预测速度是后者的两倍，且训练成本要低得多。因此，在我们的实验中，我们选择了梯度树提升作为默认成本模型。尽管如此，我们相信这两种方法都是有价值的，并期待未来对此问题的更多研究。

平均而言，树提升模型在0.67毫秒内完成预测，比运行真实测量快数千倍。图12比较了基于机器学习的优化器和黑盒自动调优方法；前者比后者更快地找到更好的配置。

5.3 调度探索

一旦我们选择了一个成本模型，就可以用它来选择有希望的配置，并在这些配置上迭代运行真实测量。在每次迭代中，探索器使用机器学习模型的预测来选择一批候选配置进行测量。收集到的数据随后被用作训练数据来更新模型。如果没有初始训练数据存在，探索器会随机选择候选配置进行测量。

最简单的探索算法会枚举并通过成本模型运行每个配置，选择预测表现最好的前k 个配置。然而，这种策略在搜索空间很大时变得难以处理。相反，我们运行一个并行模拟退火算法 [22]。探索器从随机配置开始，在每一步随机走到一个附近的配置。如果成本模型预测成本会降低，这种转换就是成功的。如果目标配置的成本更高，则很可能失败（拒绝）。随机游走倾向于收敛到成本模型预测成本较低的配置。探索状态在成本模型更新时保持不变；我们会在这些更新后从最后一个配置继续。

5.4 分布式 DevicePool 和 RPC

一个 分布式设备池 可以扩展硬件试验的运行，并允许多个优化作业之间进行细粒度资源共享。TVM 实现了一个基于 RPC 的定制分布式设备池，允许客户端在特定类型的设备上运行程序。我们可以使用这个接口在主机编译器上编译程序，请求远程设备，并在主机上运行

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?

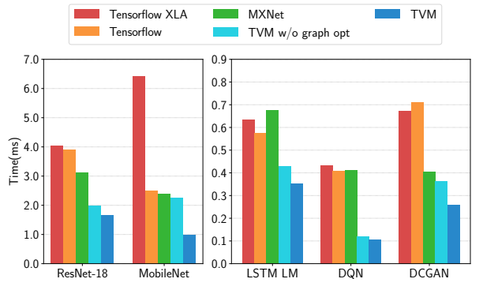


Figure 14: GPU end-to-end evaluation for TVM, MXNet, Tensorflow, and Tensorflow XLA. Tested on the NVIDIA Titan X.

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 operations, 并访问主机上同一脚本中的结果。TVM 的 RPC 支持动态上传，并运行使用其运行时约定的交叉编译模块和函数。因此，相同的 infrastructure 可以执行单个工作负载优化和端到端图推理。我们的方法跨多个设备自动执行编译、运行和分析步骤。这种基础设施对于嵌入式设备尤其关键，因为嵌入式设备传统上需要繁琐的手动工作来进行交叉编译、代码部署和测量。

Name	算子	H, W	输入通道,输出通道	K, S
C1	卷积2D	224, 224	3,64	7, 2
C2	卷积2D	56, 56	64,64	3, 1
C3	卷积2D	56, 56	64,64	1, 1
C4	卷积2D	56, 56	64,128	3, 2
C5	卷积2D	56, 56	64,128	1, 2
C6	卷积2D	28, 28	128,128	3, 1
C7	卷积2D	28, 28	128,256	3, 2
C8	卷积2D	28, 28	128,256	1, 2
C9	卷积2D	14, 14	256,256	3, 1
C10	卷积2D	14, 14	256,512	3, 2
C11	卷积2D	14, 14	256,512	1, 2
C12	卷积2D	7, 7	512,512	3, 1

Name	算子	H, W	IC	K, S
D1	深度卷积2D	112, 112	32	3, 1
D2	深度卷积2D	112, 112	64	3, 2
D3	深度卷积2D	56, 56	128	3, 1
D4	深度卷积2D	56, 56	128	3, 2
D5	深度卷积2D	28, 28	256	3, 1
D6	深度卷积2D	28, 28	256	3, 2
D7	深度卷积2D	14, 14	512	3, 1
D8	深度卷积2D	14, 14	512	3, 2
D9	深度卷积2D	7, 7	1024	3, 1

表2：ResNet-18中所有conv2D算子的配置以及单核实验中MobileNet中所有深度卷积2D算子的配置。高/宽表示高和宽，输入通道表示IC，输出通道表示OC，核大小表示K，步长表示S。所有操作使用"相同"填充。所有深度卷积2D操作都有通道乘数为1。

6 评估

TVM的核心用C++ (∼50k LoC)实现。我们提供了Python和Java的语言绑定。本文的早期部分评估了几个单独优化的影响以及TVM的组件，即图4中的<样式id='3'>算子融合</样式>，图10中的<样式id='5'>延迟隐藏</样式>，以及图12中的<样式id='7'>基于机器学习的成本模型</样式>。我们现在专注于端到端的评估，旨在回答以下问题：

- TVM 能否在多个平台上优化深度学习工作负载?
- TVM 与现有的深度学习框架相比，在各个后端上的表现如何?

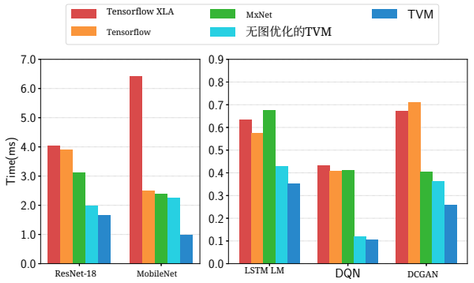


图 14：TVM、MXNet、Tensorflow 和 Tensorflow XLA 的 GPU 端到端评估。在 NVIDIA Titan X 上测试。

（这些框架依赖于高度优化的库）

- TVM 能否支持新的、新兴的深度学习工作负载（例如，深度卷积、低精度运算）？
- TVM 能否支持并优化新的专用加速器?

为了回答这些问题，我们在四种类型的平台上评估了 TVM：（1）服务器级 GPU，（2）嵌入式 GPU，（3）嵌入式 CPU，以及（4）在低功耗 FPGA SoC 上实现的深度学习加速器。基准测试基于真实的深度学习推理工作负载，包括 ResNet [16], MobileNet [19], LSTM 语言模型 [48], 深度 Q 网络 (DQN) [28] 以及深度卷积生成对抗网络 (DCGAN) [31]。我们将我们的方法与现有的深度学习框架进行比较，包括依赖高度工程化、供应商特定库的 MxNet [9] 和 Tensorflow [2], 。TVM 执行端到端自动优化和代码生成 无需外部算子库。

6.1 服务器级 GPU 评估

我们首先比较了在 NVIDIA Titan X 上深度神经网络的端到端性能，包括 TVM、MXNet (v1.1)、TensorFlow (v1.7) 和 Tensorflow XLA。MXNet 和 TensorFlow 都使用 cuDNN v7 进行卷积算子；由于深度卷积相对较新且最新库尚未支持，它们实现了自己的深度卷积版本。它们还使用 cuBLAS v8 进行矩阵乘法。另一方面，Tensorflow XLA 使用即时编译。

图14显示TVM的性能优于基线，由于联合图优化和自动优化器的共同作用，速度提升范围从1.6×到3.8×，自动优化器生成了高性能的融合算子。

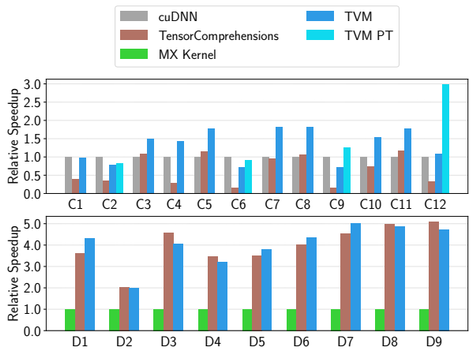


Figure 15: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in MobileNet. Tested on a TITAN X. See Table 2 for operator configurations. We also include a weight pre-transformed Winograd [25] for 3x3 conv2d (TVM PT).

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.

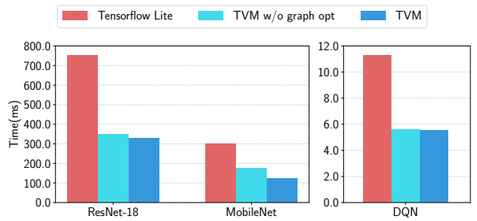


Figure 16: ARM A53 end-to-end evaluation of TVM and TFLite.

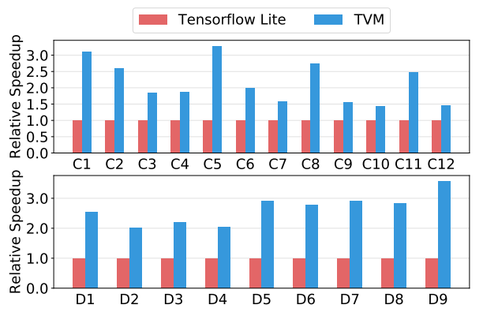


Figure 17: Relative speedup of all conv2d operators in ResNet-18 and all depthwise conv2d operators in mobilenet. Tested on ARM A53. See Table 2 for the configurations of these operators.

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.

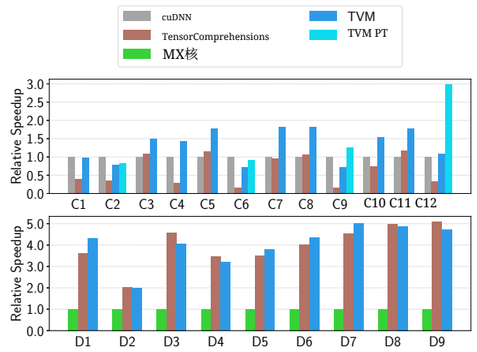


图15：ResNet-18中所有卷积2D算子的相对加速比和MobileNet中所有深度卷积2D算子的相对加速比。在TITAN X上测试。算子配置请参见表2。我们还包含了一个3x3卷积的预变换Winograd [25]（TVM PT）。

DQN的3.8倍速度提升源于其使用了非常规算子(4×4 卷积2D，步长=2)，这些算子未被cuDNN充分优化；而ResNet工作负载则更为常规。TVM在这两种情况下都能自动找到优化算子。

为评估算子级优化的有效性，我们还对ResNet和MobileNet中每个张量算子进行了分解对比，如图15所示。我们还将TensorComprehension（TC，提交：ef644ba） [42], 作为最近引入的自动调优框架，作为额外的基线。 [2] TC结果包括它在每个算子的10代 × 100种群 × 2 随机种子中找到的最佳内核（即每个算子2000次试验）。二维卷积作为最重要的深度学习算子之一，被cuDNN高度优化。然而，TVM仍然可以为大多数层生成更好的GPU内核。深度卷积是一个结构更简单的最新引入算子 [19]。在这种情况下，与MXNet的手工内核相比，TVM和TC都能找到更快的内核。TVM的改进主要归功于它对大规模调度空间的探索和有效的基于机器学习的搜索算法。

6.2 嵌入式 CPU 评估

我们在 ARM Cortex A53（四核 1.2GHz）上评估了 TVM 的性能。我们使用 Tensorflow Lite (TFLite, 提交: 7558b085) 作为我们的基准系统。图 17 比较了 TVM 算子与手工优化的

[2]根据个人沟通 [41], TC 目前尚不打算用于计算密集型问题。然而，它仍然是一个很好的参考基线，可以包含在比较中。

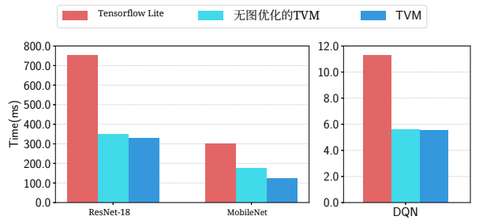


图 16：ARM A53 上 TVM 和 TFLite 的端到端评估。

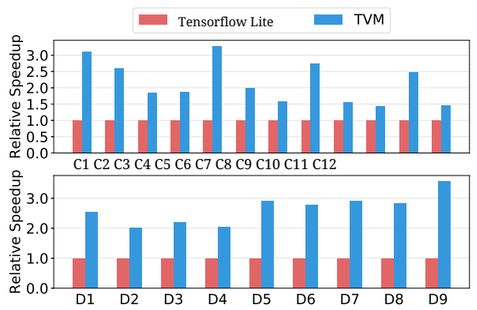


图 17：ResNet-18 中所有 conv2d 算子的相对加速比，以及 mobilenet 中所有深度卷积 2D 算子的相对加速比。在 ARM A53 上测试。有关这些算子的配置，请参见表 2。

ResNet 和 MobileNet 的手工优化算子。我们观察到 TVM 生成的算子在两种神经网络工作负载上都优于手工优化的 TFLite 版本。这一结果还展示了 TVM 快速优化新兴张量算子的能力，例如深度卷积算子。最后，图 16 展示了三个工作负载的端到端比较，其中 TVM 的性能优于 TFLite 基准系统。[3]

超低精度算子 我们展示了 TVM 支持超低精度推理的能力 [13, 33]，通过为小于 8 位的定点数据类型生成高度优化的算子。低精度网络用矢量化位串行乘法替换昂贵的乘法运算，后者由位与和 popcount 还原组成 [39]。实现高效的低精度推理需要将量化数据类型打包到更宽的标准数据类型中，例如 int8 或 int32。我们的系统生成的代码性能优于 Caffe2 的手优化库（提交：39e07f7）

[3]DCGAN 和 LSTM 结果未呈现，因为基线尚未支持它们。

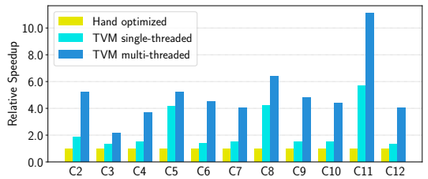


Figure 18: Relative speedup of single- and multi-threaded low-precision conv2d operators in ResNet. Baseline was a single-threaded, hand-optimized implementation from Caffe2 (commit: 39e07f7). C5, C3 are 1x1 convolutions that have less compute intensity, resulting in less speedup by multi-threading.

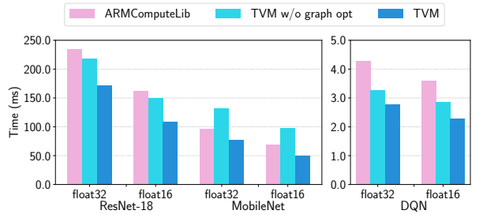


Figure 19: End-to-end experiment results on Mali-T860MP4. Two data types, float32 and float16, were evaluated.

[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× to 1.6×.

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 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.

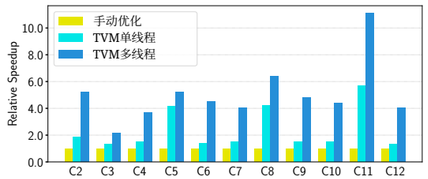


图18：ResNet中单线程和多线程低精度conv2d算子的相对加速比。基线是一个来自Caffe2的单线程手动优化实现（提交：39e07f7）。C5、C3是计算强度较低的1x1卷积，因此多线程带来的加速比较小。

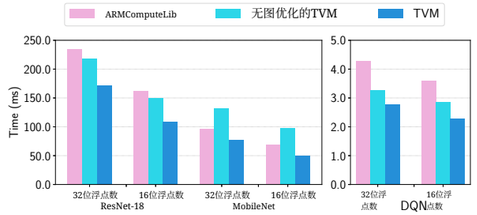


图19：Mali-T860MP4上的端到端实验结果。评估了两种数据类型，float32和float16。

[39]。我们实现了一个针对 ARM 的 张量化内建函数，该函数利用 ARM 指令构建高效的低精度矩阵向量微内核。然后我们使用 TVM 的自动优化器探索调度空间。

图18比较了TVM与Caffe2超低精度库在ResNet上的表现，用于2位激活值、1位权重推理。由于基线是单线程的，我们还将其与单线程TVM版本进行了比较。单线程TVM优于基线，尤其是在C5、C8和C11层上；这些是核大小为 1×1 、步长为2的卷积层，超低精度基线库未对其进行优化。此外，我们利用TVM的额外能力生成了一个并行库实现，其性能优于基线。除了2位+1位配置外，TVM还可以生成和优化基线库不支持的其它精度配置，提供了更好的灵活性。

6.3 嵌入式GPU评估

在我们的移动GPU实验中，我们在配备ARM Mali-T860MP4 GPU的Firefly-RK3399开发板上运行了我们的端到端流程。基线是一个供应商提供的库，即ARM Compute Library (v18.03)。如图19所示，我们在float16和float32（DCGAN和LSTM尚未被基线支持）的三个可用模型上均优于基线。加速比范围在1.2×到1.6×之间。

6.4 FPGA加速器评估

香草深度学习加速器我们现在阐述了TVM如何处理针对通用推理加速器设计的加速器特定代码生成，该设计我们在FPGA上进行了原型实现。在此评估中，我们使用了VanillaDeep

VDLA是作为张量处理器编程的，以高效执行具有高计算强度的操作（例如，矩阵乘法、高维卷积）。它可以执行加载/存储操作，将分块的3维张量从DRAM传输到SRAM的连续区域。它还提供专用片上内存，用于网络参数、层输入（窄数据类型）和层输出（宽数据类型）。最后，VDLA在连续的加载、计算和存储之间提供显式同步控制，以最大化内存和计算操作的重叠。

方法。 我们在一个集成了ARM Cortex A9双核CPU（主频667MHz）和基于Artix-7的FPGA布料的低功耗PYNQ开发板上实现了VDLA设计。在这些有限的FPGA资源上，我们实现了一个 16×16 矩阵向量单元，该单元以200MHz的频率执行8位值的乘法，并将它们在每个周期累积到一个32位寄存器中。该VDLA设计的理论峰值吞吐量约为102.4GOPS/s。我们为激活存储分配了32kB资源，为参数存储分配了32kB，为微代码缓冲区分配了32kB，为寄存器文件分配了128kB。这些片上缓冲区显然不足以为ResNet的单层提供足够的片上存储，因此能够进行有效内存重用和延迟隐藏的案例研究。

我们为VDLA构建了一个驱动库，该库使用C运行时API构建指令并将它们推送到目标加速器以执行。然后，我们的代码生成算法将加速器程序转换为一系列对运行时API的调用。添加专用的加速器后端需要 ∼2k行Python代码。

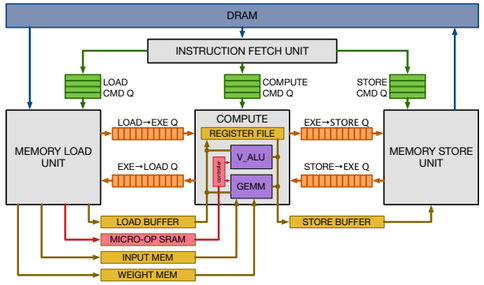


Figure 20: VDLA Hardware design overview.

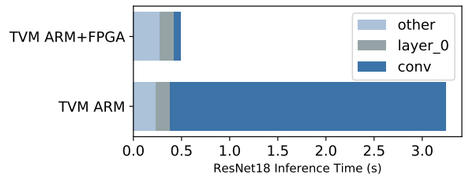


Figure 21: We offloaded convolutions in the ResNet workload to an FPGA-based accelerator. The grayed-out bars correspond to layers that could not be accelerated by the FPGA and therefore had to run on the CPU. The FPGA provided a 40x acceleration on offloaded convolution layers over the Cortex A9.

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.

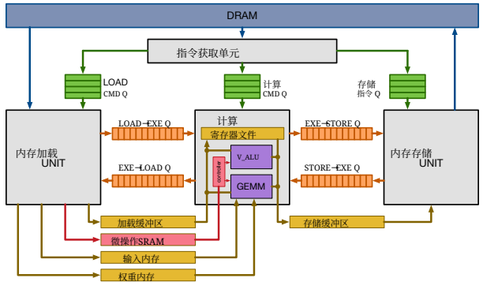


图20：vdla硬件设计概述。

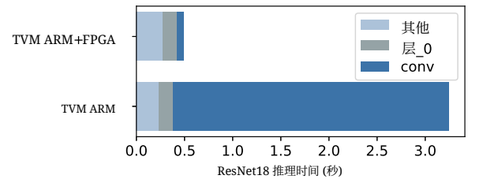


图21：我们将ResNet工作负载中的卷积卸载到了基于FPGA的加速器上。灰色的条形对应无法被FPGA加速且因此必须在CPU上运行的层。FPGA为卸载的卷积层提供了比Cortex A9高40倍的加速。

端到端ResNet评估。我们使用TVM在PYNQ平台上生成ResNet推理内核，并尽可能将尽可能多的层卸载到VDLA。我们还使用它来生成仅针对CPU的调度和CPU+FPGA实现调度。由于第一个ResNet卷积层的卷积深度较浅，它不能在FPGA上高效卸载，而是在CPU上计算。然而，ResNet中的所有其他卷积层都可以高效卸载。残差层和激活等操作也在CPU上执行，因为VDLA不支持这些操作。

图21将ResNet推理时间分解为仅CPU执行和CPU+FPGA执行。大部分计算时间都花在了可以卸载到vdla的卷积层上。对于这些卷积层，实现的加速比是 40×。不幸的是，由于阿姆达尔定律，FPGA加速系统的整体性能被必须由CPU执行的工作负载部分所瓶颈。我们设想扩展vdla设计以支持这些其他算子将有助于进一步降低成本。这个基于FPGA的实验展示了TVM适应新架构及其暴露的硬件内建函数的能力。

7 相关工作

深度学习框架 [3, 4, 7, 9] 为用户提供便捷的接口来表述深度学习工作负载，并轻松地部署到不同的硬件后端上。虽然现有框架目前依赖于特定供应商的张量算子库来执行其工作负载，但它们可以利用TVM的栈为更多硬件设备生成优化代码。

高层计算图DSL是一种典型的表示和执行高层优化的方法。Tensorflow的XLA [3] 和最近引入的DLVM [45]都属于此类。这些表示这些工作中的计算图表示方式相似，本文也使用了高层计算图DSL。虽然图级表示非常适合高层优化，但在多样化的硬件后端下，它们对于优化张量算子来说过于抽象。先前的工作依赖于特定的降低规则来直接生成低级LLVM，或者采用厂商定制的库。这些方法需要对每个硬件后端和算子变体组合投入大量的工程工作。

Halide [32] 引入了计算与调度分离的思想。我们采用了Halide的见解，并在我们的编译器中重用了其现有的有用调度原语。我们的张量算子调度也与其他关于GPU的DSL [18, 24, 36, 37] 和基于多边形的循环转换 [6,43]的工作相关。TACO [23]引入了一种生成CPU上稀疏张量算子的通用方法。Weld [30] 是一种用于数据处理任务的DSL。我们特别关注解决GPU和专用加速器上DL工作负载的新调度挑战。我们的新原语有可能被这些工作中的优化管道采用。

高性能库如ATLAS [46] 和FFTW [15] 使用自动调优来获得最佳性能。张量理解 [42]应用黑盒自动调优结合多面体优化来优化CUDA内核。OpenTuner [5] 和现有的超参数调优算法 [26] 应用领域无关的搜索。预定义的成本模型用于自动调度Halide [29]中的图像处理管道。TVM的机器学习模型使用有效的领域感知成本建模，考虑程序结构。基于分布式调度优化器扩展到更大的搜索空间，并且可以在支持的后端的大范围内找到最先进的内核。更重要的是，我们提供了一个端到端栈，可以直接从深度学习框架获取描述，并与图级栈联合优化。

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).

尽管深度学习加速器 [11, 21]，正在兴起，但如何构建编译栈以有效针对这些设备仍不明确。我们评估中使用的 VDLAdesign 提供了一种通用的方法来总结类似 TPU 的加速器的特性，并支持对如何为加速器编译代码的具体案例研究。我们的方法也可能惠及现有的将深度学习编译到 FPGA [34,40]，的系统。本文通过张量化和编译器驱动的延迟隐藏，提供了一种通用的解决方案来有效针对加速器。

8 结论

我们提出了一种端到端编译栈，以解决深度学习在不同硬件后端上的基本优化挑战。我们的系统包括自动端到端优化，这在历史上是一项劳动密集且高度专业化的任务。我们希望这项工作将鼓励对端到端编译方法进行更多研究，并为深度学习系统软硬件协同设计技术开辟新的机遇。

致谢

我们感谢Ras Bodik、James Bornholt、王希、Tom Anderson和张乔对本文早期版本的详细反馈。我们也感谢Sampa、SAMPL和Allen School系统小组的成员对这项工作和手稿提出的反馈。我们感谢匿名的OSDI审稿人和我们的指导Ranjita Bhagwan提供的宝贵意见。这项工作部分得到了Google博士奖学金（陈天奇）、ONR奖项#N00014-16-1-2795、NSF资助（CCF-1518703、CNS-1614717、CCF-1723352）以及英特尔（CAPA计划）、甲骨文、华为和匿名资助者的捐赠支持。

参考文献

[1] NVIDIA Tesla V100 GPU 架构：全球最先进的数据中心 GPU，2017年。[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., MAN´E, 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., VI´EGAS, F., VINYALS, O., WARDEN, P., WATTENBERG, M., WICKE, M., YU, Y., AND ZHENG, X. TensorFlow：在异构系统上的大规模机器学习，2015年。软件可在 tensorflow.org 获取。

[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.和ZHENG, X. TensorFlow：一个用于大规模机器学习的系统。在第*12*届*USENIX*操作系统设计与实现研讨会（*OSDI 16*）(2016年)，第265-283页。

[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. 计算网络与计算网络工具包的介绍。技术报告 MSR-TR-2014-112，2014年8月。

[5] ANSEL, J.，KAMIL, S.，VEERAMACHANENI, K.，RAGAN-KELLEY, J., BOSBOOM, J., O'REILLY, U.-M., AND AMARAS-INGHE, S. Opentuner：一个可扩展的程序自动调优框架。在《国际并行架构与编译技术会议》（加拿大埃德蒙顿，2014年8月）.

[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 HA- JIYEV, E.铅笔：一种平台无关的加速器编程计算中间语言。在《*2015*年并行架构与编译国际会议论文集（*PACT*）》（华盛顿特区，美国，2015年），PACT '15，IEEE 计算机协会，第138-149页。

[7] BASTIEN, F., LAMBLIN, P., PASCANU, R., BERGSTRA, J., GOODFELLOW, I. J., BERGERON, A., BOUCHARD, N., AND BENGIO, Y. Theano：新功能和速度改进。深度学习与无监督特征学习NIPS 2012研讨会，2012年。

[8] 陈天奇 ., AND 郭德信, C . Xgboost：可扩展的树提升系统。在 *ACM SIGKDD* 国际知识发现与数据挖掘会议论文集 (纽约，纽约州，美国，2016)，KDD '16，ACM，第785-794页。

[9] 陈天奇 .，李明 .，李阳 .，林明 .，王宁 .，王敏 .，肖腾.，许斌.，张超.，和 张智. MXNet: 一个灵活高效的机器学习库，用于异构分布式系统. 在神经信息处理系统、机器学习系统研讨会 *(LearningSys'15)* (2015).

[10] 陈天奇基于数据预取的高性能处理器. *IEEE* 计算机汇刊 *44*，5（1995年5月），609–623。

[11] 陈豫, 罗涛, 刘松, 张思, 何磊, 王健, 李丽, 陈天奇, 许哲, 孙宁, 和泰姆·奥达尼尔. 达达瑞: 一台机器学习超级计算机. 在 第*47*届 *IEEE/ACM*国际微架构研讨会论文集(华盛顿特区, 美国, 2014), MICRO-47, IEEE计算机协会, 第609-622页.[12] 陈豫-何, 艾默, 和蔡巍·伊尔瑞斯: 一种用于卷积神经网络的节能数据流空间架构. 在第 *43* 届国际计算机架构研讨会论文集 (美国新泽西州皮斯卡塔韦 , 2016), ISCA '16, IEEE 出版社 , 第 367-379 页 .[13] 库尔巴里亚克斯, 贝尼奥, 和大卫. 二进制连接: 在传播过程中使用二进制权重训练深度神经网络. *arXiv 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).

[14] 伊格斯, S·J., 艾默, J·S., 莱维, H·M., 罗, J·L., 斯塔姆, R·L., 和图尔森, D·M. 同时多线程: 下一代处理器的平台. *IEEE Micro 17*, 第5期(1997年9月), 第12-19页.[15] 弗里戈, M., 和约翰逊, S·G. FFTW: 一种自适应的FFT软件架构. 在 *1998*年*IEEE*国际会议论文集·声学、语音和信号处理(1998年5月), 第3卷, 第1381-1384页, 第3卷.[16] 何凯明, 张晓, 郑强, 和孙剑. 深度残差网络中的恒等映射. *arXiv*预印本 *arXiv:1603.05027*(2016).[17] 黑格arty, J., 布鲁恩哈弗, Z., 德维托, Z., 拉根-凯利, J., 科恩, N., 贝尔, S., 瓦西列夫, A., 赫罗维茨, M., 和汉拉汉, P. 暗室: 将高级图像处理代码编译成硬件流水线. *ACM*图形学交易. *33*, 第4期(2014年7月), 第144:1-144:11.[18] 亨利克森, T., 塞鲁普, N·G·W., 埃尔曼, M., 恩格尔林, F., 和奥安塞亚, C·E. 富特阿尔克: 一种具有嵌套并行性和原地数组更新的纯函数GPU编程. 在 第*38*届*ACM SIGPLAN*编程语言设计和实现会议论文集(美国纽约, 2017), PLDI 2017, ACM, 第556-571页.[19] 霍华德, A·G., 朱铭, 陈博, 卡莱尼琴科, D., 王伟, 韦安德, 安德烈托, M., 和亚当, H. MobileNet: 用于移动视觉应用的高效卷积神经网络. *arXiv abs/1704.04861*(2017).[20] 约普皮, N·P. 通过添加一个小全关联缓存和预取缓冲区来提高直接映射缓存性能. 在 *[1990]* 第*17*届国际计算机架构研讨会论文集(1990年5月), 第364-373页.[21] 约普皮, N·P., 杨, C., 帕蒂尔, N., 帕特森, D., 阿格拉瓦尔, G., 巴贾瓦, R., 贝茨, S., 巴蒂亚, S., 博登, N., 博尔彻斯, A., 博伊尔, R., 坎廷, P-Л., 赵晨, 克拉克, C., 科里尔, J., 戴利, M., 戴, M., 迪恩, J., 格尔布, B., 格哈姆马吉米, T·V., 戈蒂帕蒂, R., 古兰德, W., 哈格曼, R., 霍, C·R., 霍伯格, D., 胡军, 亨特, R., 赫特, D., 伊巴兹, J., 贾菲, A., 雅沃尔斯基, A., 卡普兰, A., 卡海坦, H., 基尔布雷伍德, D., 科奇, A., 库马尔, N., 莱西, S., 劳登, J., 劳, J., 李, D., 李, C., 刘, Z., 卢克, K., 兰丁, A., 麦凯恩, G., 马吉奥雷, A., 马洪尼, M., 米勒, K., 纳加拉贾南, R., 奈拉亚南斯瓦米, R., 尼, R., 尼克斯, K., 诺里, T., 奥默尼克, M., 彭库翁达, N., 菲尔普斯, A., 罗斯, J., 罗斯, M., 塞莱克, A., 萨马迪亚尼, E., 塞文, C., 西齐科夫, G., 斯内尔汉, M., 苏特, J., 斯坦伯格, D., 斯温, A., 坦, M., 托尔森, G., 田博, 天, 汤姆, H., 塔特尔, E., 瓦苏德万, V., 瓦尔特, R., 王伟, 威尔科克斯, E, 和尤恩, D·H. 张量处理单元的数据中心性能分析. 在第*44*届国际计算机架构研讨会论文集(美国纽约, 2017), ISCA '17, ACM, 第1-12页.[22] 基尔帕特里克, S., 杰拉特, C·D., 和韦基, M·P. 模拟退火优化. 科学 *220*, 第4598期(1983年), 第671-680页.[23] 科奥尔斯塔德, F., 卡米尔, S., 周翔, 卢加托, D, 和阿马拉辛赫, S. 张量代数编译器. *ACM*程序语言过程. *1*, OOPSLA (2017年10月), 第 77:1-77:29.[24] 克洛克纳 , A·L. Loo.py: 基于转换的 GPU 和 CPU代码生成. 在 *ARRAY '14: ACM SIGPLAN*数组编程库、语言和编译器研讨会论文集(苏格兰爱丁堡, 2014), 计算机协会.[25] 拉文, A, 和格雷, S. 卷积神经网络的快速算法. 在*2016*年*IEEE*计算机视觉会议 *and* 模式识别，*CVPR 2016*，拉斯维加斯，*NV*，美国，*2016*年*6*月*27*日至*30*日 (2016)，pp. 4013–4021.[26] LI, L., JAMIESON, K. G., DESALVO, G., ROSTAMIZADEH, A., AND TALWALKAR, A. 高效超参数优化和无限多臂老虎机。 *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：一种多功能的机器学习加速器。在 第十九届编程语言与操作系统架构国际会议论文集 (纽约，NY，美国，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. 通过深度强化学习实现人类水平的控制。 *Nature 518*, 7540 (2015)，529.[29] MULLAPUDI, R. T., ADAMS, A., SHARLET, D., RAGAN-KELLEY, J., AND FATAHALIAN, K. 自动调度Halide图像处理管道。 *ACM Trans. Graph. 35*, 4 (2016年7月)，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：重新思考数据密集型应用程序之间的接口。 *CoRR abs/1709.06416* (2017).[31] RADFORD, A., METZ, L., AND CHINTALA, S. 基于深度卷积生成对抗网络的无监督表示学习。 *arXiv preprint arXiv:1511.06434* (2015).[32] RAGAN-KELLEY, J., BARNES, C., ADAMS, A., PARIS, S., DU- RAND, F., AND AMARASINGHE, S. Halide：一种用于优化图像处理管道中并行性、局部性和可重计算的编程语言和编译器。在 第*34*届*ACM SIGPLAN*编程语言设计与实现会议 (纽约，NY，美国，2013)，PLDI '13，ACM，pp. 519–530.[33] RASTEGARI, M., ORDONEZ, V., REDMON, J., AND FARHADI, A. Xnor-net：使用二进制卷积神经网络的ImageNet分类。在 欧洲计算机视觉会议 (2016)，Springer，pp. 525–542.[34] SHARMA, H., PARK, J., MAHAJAN, D., AMARO, E., KIM, J. K., SHAO, C., MISHRA, A., AND ESMAEILZADEH, H. 从高层深度神经网络到FPGA。在微架构 *(MI- CRO)*，*2016*年*49*届年*IEEE/ACM*国际研讨会 (2016)，IEEE，pp. 1–12.[35] SMITH, J. E. 解耦访问/执行计算机架构。在 第 *9* 届计算机架构国际会议论文集 (洛斯阿拉莫斯，CA，美国，1982)，ISCA '82，IEEE 计算机学会出版社，pp. 112–119.[36] STEUWER, M., REMMELG, T., AND DUBACH, C. Lift：一种用于高性能GPU代码生成的函数式数据并行中间表示。在 第*2017*届代码生成与优化国际研讨会(皮斯卡托威，NJ，美国，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：一种用于机器学习的隐式并行领域特定语言。在 第*28*届国际机器学习会议 (美国，2011)，ICML'11，pp. 609–616.[38] TAI, K. S., SOCHER, R., AND MANNING, C. D. 从树状长短期记忆网络中获得改进的语义表示。 *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).

[39] TULLOCH, A., 和 JIA, Y. 移动设备上的高性能超低精度卷积。*arXiv* 预印本 *arXiv:1712.02427*(2017)。[40] UMUROGLU, Y., FRASER, N. J., GAMBARDELLA, G., BLOTT, M., LEONG, P. H. W., JAHRE, M., 和 VISSERS, K. A. FINN: 一个用于快速、可扩展二值化神经网络推理的框架。 *CoRRabs/1612.07119* (2016)。[41] VASILACHE, N. 个人交流。[42] VASILACHE, N., ZINENKO, O., THEODORIDIS, T., GOYAL, P., DEVITO, Z., MOSES, W. S., VERDOOLAEGE, S., ADAMS, A., 和 COHEN, A. 张量解析：框架无关的高性能机器学习抽象。 *CoRR abs/1802.04730* (2018)。[43] VERDOOLAEGE, S., CARLOS JUEGA, J., COHEN, A., IGNA- CIO G ´OMEZ, J., TENLLADO, C., 和 CATTHOOR, F. 多面体并行代码生成 for cuda。 *ACM* 计算机架构代码优化汇刊 *9*, 4 (2013年1月), 54:1–54:23。[44] VOLKOV, V. 理解 *GPU* 延迟隐藏。博士论文，加州大学伯克利分校，2016。[45] WEI, R., ADVE, V., 和 SCHWARTZ, L. Dlvm: 用于深度学习系统的现代编译器基础设施。 *CoRR abs/1711.03016*(2017)。[46] WHALEY, R. C., 和 DONGARRA, J. J. 自动调优的线性代数软件。在 *1998*年 *ACM/IEEE* 超级计算会议 (华盛顿特区，美国，1998), SC '98, IEEE 计算机协会, 第1–27页。[47]WILLIAMS, S., WATERMAN, A., 和 PATTERSON, D. Roofline: 一个用于多核架构的有洞察力的可视化性能模型。 *ACM* 通讯 *52*, 4 (2009年4月), 65–76。[48] ZAREMBA, W., SUTSKEVER, I., 和 VINYALS, O. 循环神经网络正则化。 *arXiv* 预印本 *arXiv:1409.2329* (2014)。