SMaLL：可移植机器学习库的软件框架

Upasana Sridhar
Nicholai Tukanov Elliott
Binder Tze Meng Low
upasanas@cmu.edu
ntukanov@cmu.edu
ebinder@cmu.edu
lowts@cmu.edu 电气与计算机工程系，卡内基梅隆大学，美国宾夕法尼亚州匹兹堡

Scott McMillan
smcmillan@sei.cmu.edu 软件工程研究所，卡内基梅隆大学，美国宾夕法尼亚州匹兹堡

Martin D. Schatz
mschatz@fb.com
Meta，美国

arXiv:2303.04769v1 [cs.MS] 8 Mar 2023

摘要

在边缘设备上部署深度神经网络（DNN）推理的需求，促使可用硬件平台的数量和类型激增。尽管 TensorFlow 等高级编程接口可以轻松移植到不同设备上，但高性能推理实现依赖于将高级接口良好映射到目标硬件平台。通常，这种映射可以通过优化编译器在编译时生成代码，也可以使用针对目标平台专门优化的高性能供应商库。

这两种方法都依赖专业知识来生成映射，这可能既耗时，也难以扩展到新的架构。

在这项工作中，我们提出了一个深度神经网络库框架 SMaLL，可轻松扩展到新的架构。该框架在所有中间层中采用统一的循环结构和共享的缓存友好数据格式，从而消除层间数据转换所带来的时间和内存开销。只需指定层的维度和 内核 ——即每层的关键计算操作——即可实现各层。

统一循环结构和内核抽象使我们能够在不同层和计算平台之间复用代码。对于新的架构，只需重新设计内核中的数百行代码。为了展示我们方法的优势，我们开发了一款软件，支持多种层类型和计算平台，并且易于扩展，可快速实例化高性能深度神经网络库。

我们通过在 5 个 ARM 平台和 1 个 x86 平台（AMD Zen 2）上实例化 TinyMLPerf基准测试套件中的神经网络，评估了我们的软件。我们的框架在端到端性能方面与 TensorFlow、TVM 和 LibTorch 等机器学习框架相当或更优。

ACM 引用格式：
Upasana Sridhar、Nicholai Tukanov、ElliottBinder、Tze Meng Low、Scott McMillan 和 Martin D. Schatz。2023年。SMaLL：便携式机器学习库的软件框架。载于*ACM Conference* 论文集（*Conference'17*）。ACM，纽约州纽约市，14 页。https://doi.org/10. 1145/nnnnnnn.nnnnnnn

1 引言

机器学习领域的快速发展，促使深度学习模型得以部署在数量激增的各类计算平台上。如今，市面上有多种 GPU、CPU 和 FPGA，它们配备了专用硬件单元（也称为加速器），例如矩阵引擎 [1] 和张量核心 [2]，可加快深度学习层的计算。面向芯片级的专用机器学习系统也在迅速推出 [3]。最后但同样重要的是，机器学习模型也越来越多地部署在计算能力和内存资源都有限的嵌入式平台上。

与快速变化的计算平台不同，编程接口一直相对稳定，TensorFlow [4]和 PyTorch [5]等机器学习框架仍是当下的主流接口。高级编程框架与底层计算平台的演进速度并不一致，这意味着我们需要一种方法，能够快速开发高性能、针对特定机器的高级接口实现。

将高级框架映射到硬件特定后端的两种主要方法是：1）使用面向特定平台的定制高性能库；或 2）使用在编译时生成平台特定代码的优化编译器。开发高性能后端（例如 oneDNN [6],CMSIS-NN [7]）需要专家开发者投入大量时间和人力，因此难以适配快速变化的平台。因此，许多微型机器学习设备都没有针对特定设备优化的高性能后端，

允许免费制作本作品的全部或部分内容的数字副本或纸质副本，供个人或课堂使用，前提是这些副本不用于或分发以获取利润或商业利益，并且副本首页须包含本声明和完整引文。对于本作品中归 ACM 以外的其他方所有的组件，必须遵守其版权规定。注明出处后允许摘录。若要以其他方式复制、重新发布、发布到服务器或向列表重新分发，须事先获得特定许可和/或支付费用。请通过 permissions@acm.org 申请许可。*Conference'17*，*2017*年7月，美国华盛顿特区
© 2023 Association for Computing Machinery。ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

这些后端专为特定的嵌入式设备设计。开发优化编译器同样需要大量专业知识。然而，编译器中的优化启发式与机器相关，需要经过调优，以避免做出不适用于特定设备的决策（例如，对于微型机器学习设备，数据重排会额外占用内存）。从本质上说，这两种方法都是黑箱式方案，需要具备领域知识的高技能专家。因此，难以快速扩展到新的架构。

本文通过引入一个开放且可移植的框架，填补了高层机器学习框架与平台特定代码之间的鸿沟，该框架可用作高层机器学习框架的后端。具体而言，我们的 SMaLL 框架背后的关键观察是，许多常见的机器学习层可以通过一个通用抽象层统一起来。这使我们能够开发一个单一的循环嵌套，并通过一小组参数使其适用于各种机器学习层，同时为每一层提供专用函数。这个通用循环嵌套得益于在许多不同层之间采用统一的数据布局，并将性能相关或平台特定的代码隔离成小段代码（内核）。不同层采用统一的数据布局，消除了数据重塑和打包的需求，避免增加机器学习网络的内存占用；这种额外开销不利于在边缘设备上部署机器学习网络。明确所需的内核后，实现整个高性能库的工作量便可缩减为仅实现这些内核，这使开发者更容易将框架移植到边缘设备上，因为这类设备通常没有可用作计算后端的高性能库。此外，所提出框架的开放性使其他人能够开发针对特定体系结构特性的定制内核。我们表明，尽管我们的方法开放且模块化，该框架的性能仍能与采用不同机器学习框架和库的实现相媲美。此外，我们通过在 TinyML、移动设备和常规 CPU 设备上使用同一框架（搭配不同的定制内核），展示了性能可移植性。

2 背景

2.1 专家机器学习库

许多机器学习库，例如 oneDNN [6], CMSIS-NN [7], ，都是由专家开发者团队历经多年开发的高性能库，并支持众多常用的机器学习层。

然而，其中许多层彼此独立实现。通常，每个层都有自己的独立循环嵌套，并且针对不同层，代码可能会被重复多次。这会产生庞大而复杂的代码库，使其难以以高性能方式移植到不同平台。

此外，每个库通常会为每个独立的层提供多种算法和实现。这些实现往往对数据布局有不同的要求，这意味着通常需要数据重塑例程，以在连续层的不同布局要求之间进行数据转换。一个经典示例是执行 im2col 转换（可以隐式执行，也可以显式执行） [8, 9], ，以利用高性能矩阵乘法例程。这些例程需要支持不同的布局，而用于在不同布局之间进行转换的例程会进一步增加需要维护的代码库规模。

相反，SMaLL 框架认识到，其中许多层可以使用相同的循环嵌套结构和相同的数据格式来实现。这降低了整体代码库的复杂性，并消除了在连续层之间支持不同数据格式的需求。将性能关键代码隔离到内核中，也简化了专家开发者的工作，因为只需针对不同平台更新性能关键代码。

2.2 编译器方法

许多先进的机器学习框架选择使用优化编译器或即时编译代码生成，以生成高性能机器学习代码。这种方法旨在快速支持新的架构，而不受供应商专用库是否可用的限制。TensorFlow [4], 、PyTorch [5] 和 TVM [10] 均在不同程度上采用了这种方法。

通常，这一过程从生成神经网络的计算图开始。例如，TensorFlow 使用加速线性代数（XLA） [11] 优化编译器，对计算图进行高层分析。图中以节点表示运算，以边表示输入和输出依赖关系。图中还可能包含张量的大小及其数据类型。随后，编译器会对计算图执行优化遍。这些优化遍可能会重写计算图，加入高性能层实现所需的数据转换；如果存在融合实现且被认为有益，也可能将多个节点融合。TVM 广泛采用融合，并利用硬件特定参数来决定如何调度神经网络的计算——例如是否并行化某一层、分块的大小等。

几乎所有自动化工具都会进行的一项关键优化，是识别将一种数据布局重塑为另一种更适合后续层的数据布局的机会。这类转换的一个例子是，将原始数据打包成能够利用高性能矩阵乘法例程（例如 im2col）的格式，从而隐藏打包带来的开销。我们的框架旨在让所有层共用数据布局，以免进行数据重新打包或重塑。这非常适合微型机器学习设备，因为这类设备的内存资源有限，难以承受不必要的内存开销。

编译器方法有时会完成从计算图到可执行文件的所有降级步骤。XLA、PyTorch 的优化编译器 Glow [12], 和 TVM 具有抽象后端，可针对不同硬件平台进行专门化。这使它们能够在有供应商库可用时使用高性能供应商库，否则则使用默认实现。这种设计可能会导致在供应商支持程度不同的平台之间，性能特征存在较大差异。SMaLL 也提供不同层的参考实现，但编写专用库所需的工作量已减少到编写内核。

2.3 可移植线性代数库实例化

我们用于实例化机器学习库的 SMaLL 框架采用了与类 BLAS 库实例化软件（BLIS）相同的方法。BLIS 是一个开源框架 [13]，用于快速实例化经过优化的基本线性代数子程序 [14–16] 库。具体而言，BLIS 将用于高性能矩阵乘法 [17] 的事实标准 Goto 算法提炼为一个特定于架构的计算微内核和两个打包例程 [18]。通过为微内核和/或打包例程提供自定义实现，并为框架指定 5 个特定架构参数，即可为特定架构生成完整的 BLAS 库。

在 BLIS 框架中，5 个特定于架构的参数用于定制中间数据格式，将问题划分为适合特定计算平台不同存储层次结构 [19] 的子问题。矩阵的所有其他细节（例如对称性、对角线上的值）以及边缘情况的处理均由 BLIS 框架管理，从而使开发人员无需为不同情况编写代码。

类似地，我们的 SMaLL 框架会识别需要实现高性能的关键代码段，而内核周围的循环则以平台无关的方式编写，以确保在不同架构之间具有可移植性。我们的框架还采用一种可根据架构特征进行参数化的通用布局。所有受支持的层都使用这种通用布局，从而减少支持不同数据布局所需的编程工作。

3 机器学习层的统一视图

SMaLL 框架基于一项关键观察构建而成：许多常用的深度神经网络（DNN）层都是模板运算，它们通过在一组输入值上使用滑动窗口来计算每个输出元素。这项观察表明，不同层可以使用相同的循环嵌套实现，从而减少支持新层所需编写的代码量。本节将介绍 DNN 工作负载中常见不同层的统一视图。

3.1 抽象深度神经网络层

我们首先介绍一种可专门化为具体深度神经网络层的抽象深度神经网络层。抽象层由一个输入张量（I）、一个输出张量（O）以及一个可选的权重张量（W）组成。其计算由映射函数 F 表示，该函数将 I 中的一组输入元素（以及 W 中一组可选的对应元素）归约为 O 中的单个输出元素。输入元素通过 F 中指定的运算进行归约。

3.1.1 输入张量和输出张量。 我们的抽象层的输入张量和输出张量是三维的，即具有 I_H、I_W 和 I_C 这几个维度，分别表示高度、宽度和通道。请注意，输出张量的维度也采用类似的标记方式，其中 O_H、O_W 和 O_C 分别表示输出高度、宽度和通道。通过增加多个输入张量和/或输出张量，可以支持多个输入和/或输出张量增加第四个维度 I_B，即可支持输入批次。为便于讨论，我们将作出简化假设，即 $\mathrm{I}_B = 1$。

3.1.2 从输入到输出张量的映射。映射函数 F 指定一个输入元素窗口，这些元素必须经过归约才能生成每个输出元素。该窗口的大小由参数 F_H、F_W 和 F_C 描述，用于指定每个维度上参与归约的元素数量。输出的高度和宽度方向上的元素，是根据高度和宽度维度上的步幅（分别记为 S_H 和 S_W），通过在输入张量上"滑动"窗口计算得出的。输出的高度和宽度方向上的元素数量为

$$O_H = \frac{\mathcal{I}_H - \mathcal{F}_H}{S_H} + 1 \qquad O_W = \frac{\mathcal{I}_W - \mathcal{F}_W}{S_W} + 1$$

类似地，通道维度上的元素也可以通过按照步幅 S_C 沿通道维度滑动窗口来生成。此滑动操作生成的通道数由参数 G（即组数）给出。

$$G = \frac{\mathcal{I}_C - F_C}{S_C} + 1$$

F 中指定的归约操作可能涉及使用一组称为权重或滤波器的学习值对输入张量中的元素进行缩放。因此需要一个权重张量，具体如下所述。

此外，也可以使用不同的一组权重对输入中的同一窗口进行缩放。我们用参数 K 表示应用于同一组输入元素、以便在输出的通道维度上生成不同元素的不同滤波器数量。通过这 7 个参数——3 个归约窗口参数 F_H、F_W、F_C，它们对应的步幅 F_H、F_W、F_C，以及应用于输入同一窗口的滤波器的批量大小 K——我们便能完整描述从输入张量到输出张量的映射。

考虑到复用输入元素所产生的元素，以及沿通道维度滑动归约窗口所产生的元素，输出通道总数等于组数、 G 和每组滤波器数量 K 的乘积。

将归约操作作为映射函数 F 的参数后，具有相同窗口参数 $\mathrm{F}_H \times \mathrm{F}_W \times \mathrm{F}_C$ 的层可以归为一组。

$$O_C = K \times G$$

3.1.3 权重张量。 权重张量 W 是可选的，其维度取决于 F 中的归约操作。在常用的机器学习层中，W 有五个维度 $\mathrm{F}_H \times \mathrm{F}_W \times \mathrm{F}_C \times K \times G$。$\mathrm{F}_H \times \mathrm{F}_W \times \mathrm{F}_C$ 元素描述滤波器值。输入张量的归约窗口中的元素会按比例缩放。 K 表示应用于同一窗口的滤波器元素批次。我们特别关注 G，这是一个由窗口大小和通道维度中的步幅推导出的参数。通过在输入的高度和宽度上滑动归约窗口生成的元素使用相同的滤波器，而不同组中的元素可能会由不同的滤波器缩放。直观地说，

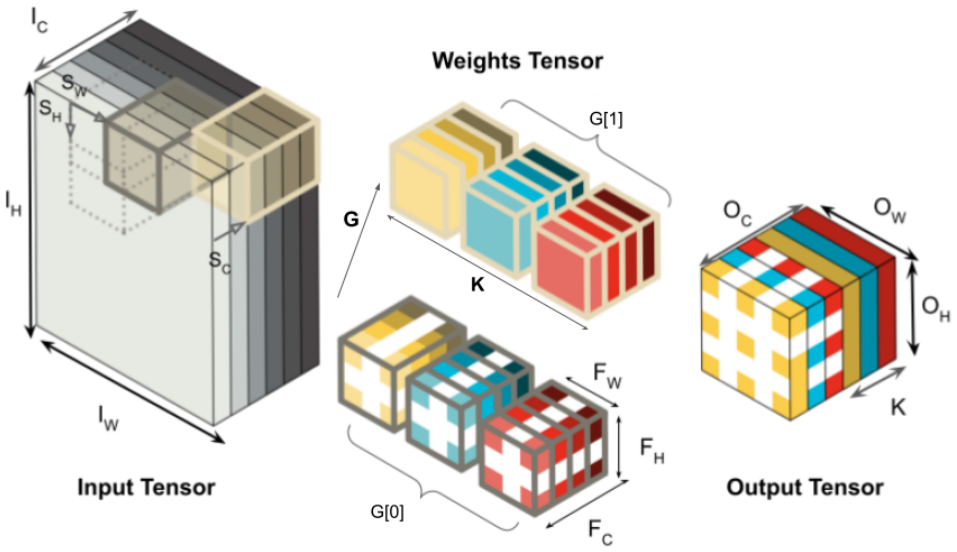


图 1：抽象层的可视化。滤波器张量 $\mathcal{W}$ 包含 $G = 2$ 组滤波器，每组有 $K = 3$ 个滤波器，共产生 $O_C = 6$ 个输出通道。每个滤波器 $\mathcal{F}$ 对 $\mathcal{F}_W \times \mathcal{F}_H = 9$ 个输入元素进行归约，这些元素位于 $\mathcal{F}_C = 4$ 个输入通道中。

通道维度表示输入的不同属性，因此，权重张量包含 G 维度。

3.1.4 输入到输出张量的映射可视化。 图 1 直观展示了抽象深度学习层的各个组成部分。输入张量是一个三维张量，大小为 $\mathrm{I}_H \times \mathrm{I}_W \times \mathrm{I}_C$。权重张量分为两组：本例中，一组为 G_0（中央白色十字），另一组为 G_1（纯色），每组包含 $K = 3$ 个滤波器。因此，输出张量有 6 个通道，其中 3 个为纯色，另外 3 个带有白色十字图案。每个滤波器的深度也为 F_C，用于归约输入通道。在输入中，我们可以看到组间的步幅为 S_C。图中展示的情况是 $S_C < F_C$，因此两个滤波器组所使用的输入通道存在一定重叠。如上所述，通过改变归约

3.2 层的类别

利用抽象深度神经网络层，我们可以将其细分为 5 种主要层类别，这些类别常见于许多深度学习模型中。具体而言，可以根据归约窗口的维度 F 来区分这些层。五种专用层如下：

(1) 单元素归约

单元素归约也称为逐点操作。这类层从输入张量中读取一个元素，并生成一个输出元素。这些层可由以下滤波器参数进行描述：

$$\mathcal{F}_W = \mathcal{F}_H = \mathcal{F}_C = 1.$$

使用 ReLU 或 Sigmoid 的激活层就是这类层的示例。逐点层可能涉及权重张量，其元素会参与该层的计算。加法和批量归一化是单元素归约层的示例。

(2) 单通道归约

这些层在计算单个输出元素时，会对来自一个输入通道的元素进行归约。具体而言，输入张量的高度和宽度上的元素数量满足

$$\begin{aligned} \mathcal{F}_C &= 1 \\ \mathcal{F}_H &< \mathcal{I}_H, \text{ and} \\ \mathcal{F}_W &< \mathcal{I}_W. \end{aligned}$$

输入通道维度上的归约大小为 1。深度卷积和池化都是这类归约的示例，其中深度卷积会使用权重张量，而池化则不会。

二维最大池化层读取单个输入通道中的元素，并返回一个输出元素，该元素是全部 $\mathrm{F}_H \times \mathrm{F}_W$ 个输入元素中的最大值。深度卷积具有 $F_C = 1$，并采用类似的模式，将单个通道中的 $\mathrm{F}_H \times \mathrm{F}_W$ 个输入元素归约为一个输出元素。该归约操作需要一个权重张量，其中每个输入通道对应一组权重，即 $G = \mathrm{I}_C$。在大多数情况下，每个输入通道都用于生成一个输出通道，因此 $K = 1$。

(3) 部分通道归约

通道维度上的归约大小是部分归约，同时高度和宽度维度上也是如此：

$$\begin{aligned} 1 < \mathcal{F}_W &< \mathcal{I}_W, \\ 1 < \mathcal{F}_H &< \mathcal{I}_H, \text{ and} \\ 1 < \mathcal{F}_C &< \mathcal{I}_C \end{aligned}$$

分组卷积就是这种归约的一个示例。输入通道会被划分为若干组，使得组内的 F_C 个通道能够被重复利用，以生成 K 个输出通道。

分组卷积通常具有等于通道缩减 $S_C = F_C$ 的步幅，因此各组之间完全不相交。

(4) 全通道缩减
这些层使用来自所有输入通道的输入元素来计算一个输出元素，即

$$\mathcal{F}_C = \mathcal{I}_C$$

这类归约的例子包括传统卷积，以及 VGG [20] 和 ResNet [21]等神经网络中使用的 1×1 卷积。传统二维卷积会对所有输入通道上的一个小型空间窗口进行归约，以得到 1 个输出像素。使用不同的滤波器处理同一空间窗口，即可得到不同的输出通道。因此，滤波器的通道维度必须等于输入中的通道总数，即 $\mathrm{F}_C = \mathrm{I}_C$。输出通道数 O_C 是一个超参数。

1×1 卷积 [22] 只需对输入中对应于一个像素的所有输入通道进行归约，即可得到每个输出元素。请注意，这意味着除滤波器的空间维度现在固定为 1 且等于 1（$F_H = F_W = 1$）之外，所有滑动窗口参数都与传统卷积相同。这种配置还意味着空间维度的计算完全相互独立，彼此之间没有重叠。

(5) 完全缩减
此操作将输入张量中的所有元素归约为一个输出元素，即 $F_W = \mathrm{I}_W, F_H = \mathrm{I}_H$ 和 $F_C = \mathrm{I}_C$。全连接层或线性层就是完全归约的一个示例。输入可以视为一个展平向量 $1\times(\mathrm{I}_H\cdot\mathrm{I}_W\cdot\mathrm{I}_C)$，与一个 $(\mathrm{I}_H\cdot\mathrm{I}_W\cdot\mathrm{I}_C)\times K$ 权重张量相乘，从而生成 K 输出元素。

表1：根据被归约以获得单个输出的输入元素的数量和位置对层进行分类。$*$ 表示参数可能因层类型而异。部分通道归约在将输入映射到输出时具有最多的自由度。

层类别	$\mathcal{F}_H$	$\mathcal{F}_W$	$\mathcal{F}_C$	K	G
单元素归约	1	1	1	1	1
单通道归约	*	*	1	1	$\mathcal{I}_C$
部分通道归约	*	*	*	*	*
全通道归约	*	*	$\mathcal{I}_C$	O_C	1
完全归约	$\mathcal{I}_H$	$\mathcal{I}_W$	$\mathcal{I}_C$	O_C	1

3.3 将专用层映射到实际的深度神经网络层

上述 5 种层类别可进一步细分为常见机器学习网络中的实际深度神经网络层。这种映射的核心在于指定归约函数，该函数构成了操作的核心，用于处理输入张量和权重张量中的元素。在表 2 中，我们给出了每种专用层类别中一些具有代表性的深度神经网络层的映射。

4 实现机器学习层的框架

统一许多常见机器学习层的关键观察是：通过抽象层，可以为所有不同的层类别设计单一的循环嵌套。此外，为每一层提供专用的归约函数作为计算内核，就能得到不同的实际机器学习层。我们将探讨如何在通用框架中落实这些见解，从而支持在不同架构上的高性能、可移植实现。

4.1 常见高性能循环嵌套

我们首先讨论用于抽象机器学习层的常见高性能循环嵌套的设计。请注意，为了计算每个输出元素，我们首先需要遍历 $\mathrm{F}_H\times\mathrm{F}_W\times\mathrm{F}_C$ 个输入元素。总共有 $\mathrm{O}_H\times\mathrm{O}_W\times\mathrm{O}_C$ 个输出。因此，计算一个抽象层至少需要 6 层嵌套循环。

已有研究表明，可以对这 6 层嵌套循环重新组织，从而以高性能方式实现直接卷积，其性能可与基于矩阵乘法的实现相媲美，且无需额外内存进行数据打包 [23]。高性能循环嵌套的核心在于使用额外循环对当前循环进行分块/分块化，使其更好地映射到可用的硬件资源（例如寄存器数量）。这些额外循环嵌套中的参数值可根据特定架构推导得出。

采用相同的方法，我们可以定义 SMaLL 的循环嵌套，如算法 1 所示。由于直接卷积是抽象层的一种特化形式，只需对高性能直接卷积的循环结构稍作修改，即可将其泛化，以适用于不同的层类别。大部分性能分析仍然相同：最内层循环遍历彼此独立的输出元素，与归约类型无关。在所有层中，最内层的层边界表示

Zhang 等人提出的循环嵌套 [23] 与算法 1 之间的关键区别在于，Zhang 等人将遍历输出通道中各元素的循环拆分为一个遍历组参数 G的循环，以及一个遍历滤波器数量K的循环。该循环嵌套使我们能够计算单通道归约，因为要计算单个输出元素，这一特定层需要遍历 G 输入通道。将输出通道循环拆分为遍历维度 G 和 K的循环，也意味着输出通道现在可以属于同一组，也可以分布在多个组中。这意味着，原先将输出通道划分为每块包含 C_b 个通道的通道块的分块参数 C_b，也被拆分为两个参数：一个用于对组内通道进行分块 K_b，另一个用于对通道组进行分块G_b。类似地，块 C_b 输入通道由一个新的分块参数 F_{Cb} 与组分块参数 G_b的乘积得出，该新参数作用于归约的通道维度。

表 2：各专用层的代表性层。归约函数体现了不同层之间的计算差异。步长维度通常是由机器学习网络确定的输入参数。它们决定哪些输入元素映射到输出元素

层类型	专用层	$\mathcal{S}_H$	$\mathcal{S}_W$	$\mathcal{S}_C$	归约函数
激活函数（ReLU）	单元素	1	1	1	Max
批量归一化		1	1	1	FMA
上采样		*	*	1	最近邻插值
深度卷积	单通道	*	*	1	FMA
最大池化		*	*	1	Max
分组卷积	部分通道	*	*	*	FMA
卷积	全通道	*	*	1	FMA
全连接	Full	1	1	1	FMA

算法 1抽象层循环嵌套

```
1: 对于 g ≤ G，以步长 G_b 并行执行 执行
2:     对于 k ≤ K，以步长 K_b 并行执行 执行
3:         对于 i ≤ F_C，以步长 F_Cb 执行
4:             对于 j ≤ O_H 执行
5:                 对于 l ≤ O_W 以步长 O_Wb 执行
6:                                                  ▷ 内核
7:                     对于 x ≤ F_H 执行
8:                         对于 y ≤ F_W 执行
9:                             对于 ii ≤ F_Cb 执行
10:                                对于 ll ≤ O_wb 执行
11:                                    对于 gg ≤ G_b 执行
12:                                        对于 kk ≤ K_b 执行
13:                                            归约操作(O, W, I,
14:                                                S_H, S_W, S_C)
15:                                        结束循环
16:                                    结束循环
17:                                结束循环
18:                            结束循环
19:                        结束循环
20:                    结束循环
21:                结束循环
22:            结束循环
23:        结束循环
24:    结束循环
25: 结束循环
```

4.2 面向不同层的专用核

采用统一的循环结构，使我们能够以统一的方式遍历所有层类别中的输入、输出和权重元素。然而，要将它们实例化为实际的深度神经网络层，我们需要为每个需要实例化的实际层识别并实现专用的归约函数，我们将其称为内核。此外，为了实现实际层的高效实现，这些内核必须以高性能、针对特定硬件的方式实现。

从算法 1 可以看出，计算（即归约函数）是在最内层循环中执行的。此外，紧邻计算的外层循环是针对不同输出元素和/或输入元素的循环。请回想一下，每个专用层之间的差异仅体现在 5 个不同参数上；这些参数都由算法 1 的第 7–12 行涵盖。这意味着，要将抽象层专门化为不同的类别，我们只需编写替换第 7–12 行的代码。我们将这段专用代码称为每个专用层的内核。

4.2.1 内核设计。 内核被定义为算法 1 的最内层循环。为了获得高性能，这些内核以平台特定的方式编写。内核本身只对整体输出[1]中的一个小片段执行部分更新。具体而言，每个内核的核心设计基于需要执行的计算，以部分更新 $O_{wb} \times G_b \times K_b$ 个输出元素。对于每个正在计算的输出元素，都会读取 $F_H \times F_W \times F_{Cb}$ 个输入元素（以及可能数量相同的权重）。内核周围的循环使我们能够使用固定大小的内核，并将其用于处理不同的工作负载形状和大小。边界情况由内核周围的代码处理。

定义内核的参数 O_{wb}、G_b、F_{Cb} 和 K_b 的具体值通常基于机器参数，例如延迟、特定指令的吞吐量、可用寄存器的数量，以及每条单指令多数据（SIMD）指令能够计算的元素数量。为了确定特定架构的取值，我们采用 Zhang 等人 [23] 和 Low 等人 [24] 所使用的方法。由于每个内核计算 $O_{wb} \times G_b \times K_b$ 个相互独立的输出，因此内核的参数大小经过设计，可适配可用 SIMD 寄存器的数量。这确保计算能够在不发生寄存器溢出的情况下执行，而寄存器溢出会带来额外的数据移动开销。此外，由于计算针对相互独立的输出执行，最内层循环中会包含多个相互独立的指令，从而利用可用的指令级并行。对寄存器数量的这一要求通常足以填满流水线，使指令延迟得以隐藏。

此外，$G_b \times K_b$ 的组合维度表示输出 O_{cb}在通道维度上的分块。这两个维度的乘积在不同层类型中保持一致，通常设置为 SIMD 长度的倍数。具体

[1]相比之下，GPU 内核要求将整个计算放在 GPU 上。同样，TFLite 委托/内核模型要求将整个计算子图放在 GPU/加速器上。

值 G_b 和 K_b 取决于 K 和 G 的可能取值。例如，对于深度卷积，给定 $K = 1$ $G_b = O_{cb}$

虽然不同层所需的计算可能有所不同，但我们可以将内核的设计简化为 3 个独立阶段。首先，必须初始化用于存放更新后输出元素的内存（加载阶段）。然后，必须执行归约操作，其中可能包括加载相应的输入和权重元素（计算阶段）。最后，必须将更新后的元素存回输出张量（存储阶段）。

计算阶段中对输出的每次更新，都由算法 1 第 10–12 行的循环描述。第 7–9 行的循环会更新输出中的同一分块，因此我们将加载阶段和存储阶段移到这些循环之外。算法2 展示了卷积内核的结构。

算法 2 卷积内核规范（突出显示平台特定的向量化阶段）

```
1: 函数 Kernel_Convolution(O, W, I, S_H, S_W, S_C)
2:     加载_向量_分块(O O_Wb, G_b, K_b)          ▷ 设置向量使用输出分块
3:     为 x ≤ F_H 执行
4:         为 y ≤ F_W 执行
5:             为 ii ≤ F_Cb 执行
6:                 FMA_向量_分块(I,W)▷ 特定于平台计算阶段
7:             结束循环
8:         结束循环
9:     结束循环
10:    存储_向量_分块(O_Wb, G_b, K_b, O)          ▷ 存储部分更新输出
11: 结束函数
```

4.3 性能可移植性

我们的一项关键发现是，这三个阶段的实现通常可以在不同的实际层之间复用。例如，加载阶段会将数据从主存加载到 SIMD/向量寄存器中。这一操作通常会在不同层之间重复执行，且与所执行的归约函数无关。因此，对于同一层类别中的机器学习层，只需实例化一个高性能、特定于平台的加载阶段即可。存储阶段也有类似的情况。

我们发现，不同层中的许多阶段本质上完全相同，因此可以大幅减少为实例化 SMaLL 框架支持的所有层而需要编写的代码量。具体而言，这一发现使我们能够利用该框架，通过复用代码（使用模板和宏）快速实例化机器学习库；对于一个包含 10k 行代码的机器学习库，针对特定架构仅需重写约数百行代码。

4.4 通用数据格式

尽管算法 1 中的循环嵌套与 Zhang等人提出的循环嵌套略有不同，[23], 我们仍采用 Zhang 等人提出的相同数据布局。具体而言，先前的工作提出了两种不同的布局：一种用于输入/输出张量，另一种用于权重张量。我们将概述这两种数据布局，但有关采用不同布局的原因，请读者参阅 Zhang 等人。 [23],

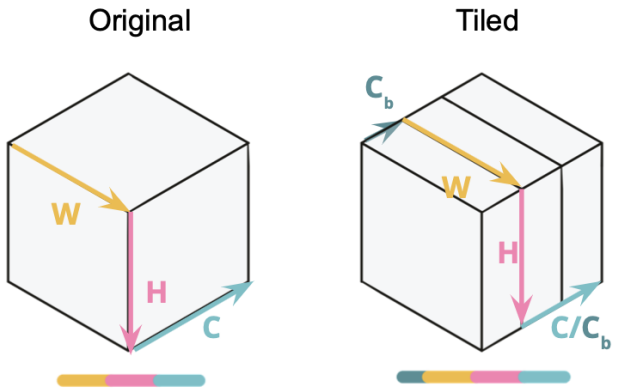


图 2：各层输入和输出的通用数据布局。底部的条带表示维度从最快到最慢的顺序。C_b 是机器特定参数。其余参数均由问题规模决定

我们采用的数据布局对输入张量和输出张量的通道维度进行了分块。最快的维度是通道数（C_b），如图 2 所示，这是一个机器特定参数。接下来两个最快的维度依次是宽度维度和高度维度。最慢的维度对应输出通道块。输出张量的数据布局与输入张量相同，因为该工作的作者观察到，后一层的输入通常是前一层的输出。

随后，对权重张量的输入通道维度和输出通道维度进行分块，以确保兼容。数据格式中各维度的顺序与算法 1 中的循环结构一致。

5 结果

5.1 实验设置

实验在多种边缘设备以及一个桌面平台上进行。表 3 汇总了运行自定义 SMaLL 实现的各种架构。此外，表中还介绍了与其他机器学习库和/或框架的比较。

基于 Armv8 的树莓派 4（一种单板计算机（SBC））、基于 x86 的 AMD Zen2 （一款消费级桌面中央处理器），以及骁龙 888 移动系统级芯片 [26] 上 Kryo 680 CPU 的 3 个平台 [25]。

我们的实现采用 C++ 编写，并在 AMD 中央处理器上使用 gcc 9.4.0 编译，在树莓派上使用 gcc 10.2.1 编译。我们以 TensorFlow 2.10 作为桌面 CPU 的基线。TensorFlow 结合 XLA 的优化代表了当前最先进的性能水平，因此我们还将自己的库与通过 XLA 即时编译、以即时执行方式运行的 TensorFlow 实现进行比较；该实现使用 AVX2 和 FMA 指令集 [27]。在本节其余部分，我们将这一组合简称为 Tensor- Flow。LibTorch（v1.8）是 PyTorch 的 C++ 接口

表 3：使用高性能 SMaLL 内核进行测试的微架构和平台汇总，以及比较过程中使用的数据类型。"-"表示该库/框架不支持相应的架构。

平台	微架构	SMaLL	TensorFlow	TFLite	LibTorch	TVM/ uTVM	SNPE
树莓派 4B	ARM Cortex-A72	浮点数	浮点数	浮点数	-	浮点型（调优）	-
Ryzen 3600x	x86 AMD Zen2	浮点数	浮点数	浮点数	浮点数	Not completed	-
骁龙 888 Kryo 680（Silver）	ARM Cortex-A55	浮点数	-	-	-	-	浮点数
骁龙 888 Kryo 680（金核）	ARM Cortex-A78	浮点数	-	-	-	-	浮点数
骁龙 888 Kryo 680（超大核）	ARM Cortex-X1	浮点数	-	-	-	-	浮点数
Arduino Nano 33 BLE	ARM Cortex-M4	量化的 uint8	-	失败	-	量化的 uint8（un-调优）	-

，我们用它来比较自己的 SMaLL 实现。单指令多数据指令集。

对于树莓派，我们将 TensorFlow 和轻量级的 tflite_runtime 包作为基线进行比较。据我们所知，LibTorch 不支持 ARM 平台，因此不报告与其的比较。我们还展示了树莓派上经过调优的 TVM [10, 28]推理模块的比较结果。我们也将 TFLite [29] 纳入比较。TFLite 使用神经网络的预编译计算图，并能够在该计算图上执行层融合等高级优化。

对于构成 Snapdragon 888 Kryo 680 的不同 CPU，我们在一台 Linux 设备上使用适用于 ARM 平台的 GNU C++ 编译器 9.4.0 版（aarch64- linux-gnu-g++）以及 Ubuntu 9.4.0-1ubuntu1 20.04.1 编译了 SMaLL 实现。我们使用 Snapdragon Neural Processing Engine (SNPE) 2.5.0.4052 [30] 版作为比较基准。与 AMD CPU 和树莓派不同，这些实现是在外部设备上编写的，并针对特定目标进行了交叉编译。

此外，我们通过展示 Arduino Nano 33 BLE 上的量化推理结果，说明该框架能够轻松适配不同数据类型的推理。Arduino Nano 33 BLE 是一款运行频率为 64 MHz 的 Cortex M4 设备。我们使用与上文相同的小型模型进行端到端推理，并将推理量化为整数（无符号 8 位）。SMaLL 中的模型使用 Arduino IDE 2.0.4 [31] 版本编译。TVM 使用 Arduino CLI 0.30.0 [32] 版本将模型传输到 Nano。

5.1.1 MLPerf:Tiny 推理模型。MLPerf:Tiny [33] 是面向 TinyML 系统的基准测试套件，包含 4 个模型，代表了可能卸载到嵌入式边缘设备上的各类机器学习任务。除深度自编码器外，所有模型均以传统卷积层开头。神经网络中的后续层通常会采用重复的层结构，并将这些层堆叠起来构成神经网络。在这 4 个模型中，有 2 个采用相同的构建模块——深度可分离卷积模块 [34]。它们是 MobileNet [35] 和 DS-CNN [36]，，使用深度卷积和 1×1 卷积。ResNet [21] 使用残差块，其中将输入累加到该块的输出上。每个块由两个 3×3 卷积层堆叠而成。如果

模型	用例	输入至模型	参数
MobileNet	视觉 Wake 单词（VWW）	(96, 96, 3)	3201472
ResNet	图像分类分类 (IC)	(32, 32, 3)	77744
DS-CNN	关键词检测 (KS)	(25, 5, 3)	20288
Deep 自动-编码器	异常检测 (AD)	(1, 1, 128)	133120

表4：用于比较端到端推理性能的4个模型摘要。参数列列出了所有层中的权重总数

输出的大小与输入不同，则会在与输出累加之前，对输入额外进行一次1×1 卷积，以匹配大小。最后，深度自编码器使用堆叠的全连接层。这些模型总结见表4。

5.2 与 TensorFlow 相比，使用统一循环结构的开销

我们通过将 5 种不同层与 TensorFlow 中的等效实现进行比较，证明所提出的统一循环结构能够以最小的循环开销专门适配多种常用机器学习层：1）ReLU 和 2）最大池化，它们使用相同的最大值内核；3）标准的 3×3 卷积；4）特殊情况 1×1 卷积；以及 5）深度卷积（Dwise）。这 5 种层涵盖了表 1所述 5 种层类型中的 3 种。此外，这些层正是 MLPerf:Tiny基准测试 [33] 中各模型所使用的层。

使用 TensorFlow 时，我们采用 Keras 子模块[37]中提供的常见神经网络层实现。我们还通过加速线性代数（XLA）领域专用编译器[11] 运行每一层，以使用各层特定于平台的实现。

借助 SMaLL，我们评估了两种实现。一种实现使用显式 NEON 向量内在函数，针对树莓派的 ARMv8 微架构专门编写内核。除了硬件专用内核之外，我们还评估了纯 C++编写的参考内核所达到的性能。参考内核使用与硬件专用内核相同的参数， $O_{wb} \times K_b \times G_b$ 但内核是纯 C++代码，并未显式使用 NEON 内在函数。这两种实现之间的性能差异凸显了通过显式使用硬件专用指令来专门优化内核实现所带来的影响。

5.2.1 层类型和尺寸的选择。我们从 MobileNet 中选取了 2 种输入张量尺寸，作为每种层类型的代表性问题。这两种尺寸计算出的输出总数相同，但元素在输出的高度、宽度和通道维度上的分布不同。第一个问题尺寸为 $48 \times 48 \times 16$（高度 ×宽度 ×通道）输入，体现了空间维度（高度和宽度）较大、通道维度较小的情况。第二个问题尺寸 $6 \times 6 \times 512$ 则具有较大的通道维度和较小的空间维度。这可以考察框架如何处理深度神经网络中常见的维度尺寸变化。

5.2.2 逐层性能比较。图 3 展示了在树莓派 4 Model B 上，各框架经过 100 次试验后测得的最短执行周期。可以看到，使用 SMaLL 的硬件专用内核时，大多数层的速度都快得多。另一个值得注意的现象是，在许多情况下，仅参考实现就比 TensorFlow 更快。然而，专家编写的内核平均还能带来 2 倍的额外加速。这表明，TensorFlow 中许多单独的层并未以高性能方式实现。这可能是因为代码库规模庞大，导致专家开发者需要耗费大量时间才能以高性能方式实现所有层。大型 3×3 卷积是唯一一个 TensorFlow 实现优于 SMaLL 实现的情况，这一现象表明该实现可能经过了优化。

SMaLL 的层速度更快，另一个可能的原因是，SMaLL 框架采用的布局便于引入使用 SIMD 指令编写的内核。SMaLL 中的布局确保输入数据具有良好的空间局部性，同时也确保输出连续。因此，即使不使用 SIMD 指令，局部性的改善也能减少等待数据传入处理器所耗费的时间。

由于 Tensorflow 是使用 XLA 编译的，最终实现可能由多种不同的源实现组合而成，和/或经过优化。与优化编译器结合使用时，很难追溯推断时所用的具体配置。相比之下，由于 SMaLL 是为抽象层设计的，因此所有问题规模都只有一条代码路径。此外，每一层都受益于在抽象层面所做的性能优化决策。

深度卷积从 SMaLL 中获得的加速幅度最大，加速比分别达到 135 倍和 311 倍。在 XLA 中，深度卷积会被转换为传统卷积，导致所需计算量大幅增加。这可能是因为 XLA 的优化列表中没有针对深度卷积的专用实现。相比之下，SMaLL 的实现是通过抽象层实例化的深度卷积专用实现。这表明，借助通用的数据布局和循环结构，可以轻松支持新的层。更重要的是，这说明可以高性能地实现针对特定滤波器尺寸的专用实现。

5.3 内存使用量对比

我们将 SMaLL 实现的内存利用情况与 TensorFlow Lite 实现进行了比较。表 5 展示了 SMaLL 中 TinyML 模型分配的内存量，以及 TFLite 实现的峰值内存占用，该数据由 TFLite 模型基准测试工具 [38] 报告。此指标通过使用给定模型运行 100 次推理计算得出：每隔 10 毫秒检查一次内存占用，并报告观测到的最大内存占用。请注意，该工具指出此数值为近似值，因为工具本身会带来一定开销。

SMaLL 实现所分配的内存取决于模型所需的参数（见表 4）以及用于存放隐藏层输出的中间缓冲区大小。我们注意到，输出缓冲区可以在各层之间共享，因此只为所需的最大输入和输出缓冲区组合分配空间。对于浮点模型，所需内存（以字节为单位）为参数数量与中间缓冲区数量之和再乘以 4。对于量化模型，尽管输入和输出值为 uint8，但部分结果的累加使用 int32。因此，量化模型所需的内存（以字节为单位）为参数数量与中间数据大小之和，再加上任一中间数据的最大高度和宽度乘积的 4 倍。

表 5：SMaLL 与 TFLite 的内存使用量比较。内存使用量以兆字节（MB）为单位。与 TFLite 的整体峰值内存占用相比，SMaLL 使用的内存更少

模型	浮点数		量化的 uint8	
	SMaLL	TFLite	小型	TFLite
自编码器 深度可分离卷积神经网络	0.509	1.660	0.128	1.586
	0.144	1.602	0.065	1.742
ResNet	0.433	4.867	0.202	4.055
MobileNet	13.162	19.258	3.361	7.254

在所有情况下，SMaLL 实现所占用的内存都少于其 TFLite 对应实现。这是意料之中的，因为 SMaLL 框架围绕单一数据布局格式设计。这意味着 SMaLL 框架的实现无需额外内存来将数据打包为不同的数据格式。每一层都可以直接处理上一层的输出，无需进行任何数据重塑。与

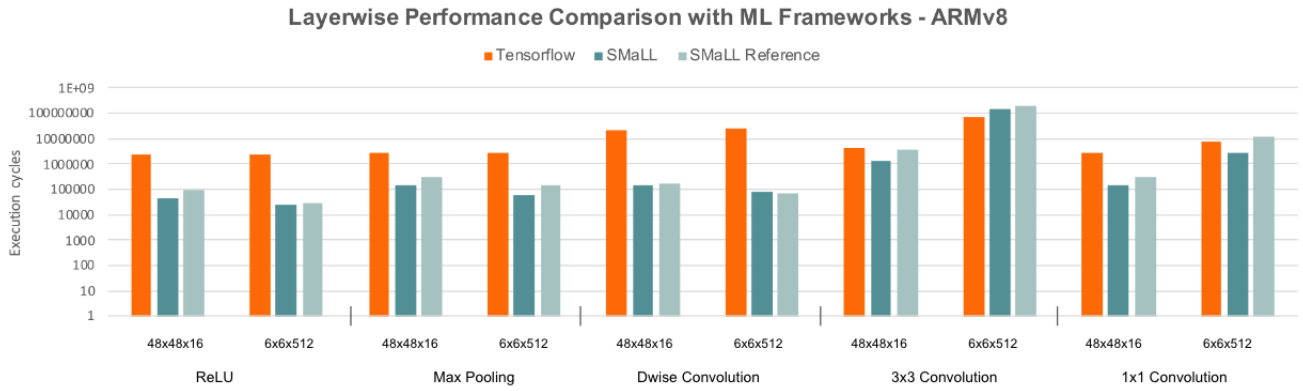


图 3：比较使用统一循环结构实现（SMaLL）的不同层执行周期与 TensorFlow 实现的执行周期。纵轴为对数刻度，数值越低越好。SMaLL 使用针对硬件优化的内核，在计算密集型的 3×3 卷积层上表现相当，在所有其他层类型上则显著更优。

可能使用临时缓冲区将数据打包为专用格式的框架相比，预计 SMaLL 在推理期间占用的内存更少。

5.4 端到端 DNN推理

我们在 6 个平台上展示了 4 项 MLPerf:Tiny Inference 基准测试。图 4 至图 5 中的柱形图展示了各框架达到的每秒帧数（FPS）。

在树莓派上，如图 4（左）所示，使用深度卷积的模型获得了最大的加速：修改版 DS-CNN 提速 2 倍，较大的 MobileNet 提速 4 倍。正如逐层比较所示，SMaLL 的深度卷积比 TensorFlow 快得多，因此这一结果在意料之中。自编码器由全连接层构成。在 SMaLL 中，这些层与 1×1 卷积非常相似，执行速度略快，因此相比 TFLite，我们看到性能提升了 28%。在所有模型中，ResNet 的 3×3 卷积最多，而 TensorFlow 的实现速度最高可比 SMaLL 快 2 倍。我们发现，SMaLL 的端到端性能较慢，帧率仅为经过调优的 TVM 实现的 44%。这可能是由于 SMaLL 当前的并行化方案所致：推理时我们仅使用 1 个线程。

请注意，在树莓派上，DS-CNN 第一层使用的矩形滤波器（3×1）似乎不受 TFLite 支持，输出始终为 0。DS-CNN 组中带虚线边框的 TFLite 柱状条显示了一个与 DS-CNN 类似的模型所达到的帧率，该模型将矩形滤波器替换为方形滤波器（3×3）。还请注意，TFLite 的速度往往比 TensorFlow 快一个数量级。这种差异可能是因为轻量级运行时系统降低了执行时间开销，并结合了图级优化。

在 Zen 2 上（如图 4 右侧所示），我们看到 TFLite 比 TensorFlow 快的趋势类似。此外，除了 MobileNet 之外，TFLite 的速度往往也快于 LibTorch。对于 TFLite，Zen 2 似乎支持矩形滤波器。SMaLL 在各种开发板上都保持了良好的性能，在 4 个模型中的 3 个上实现了最高帧率。最大的加速再次出现在使用深度卷积的模型上。7.1 倍的

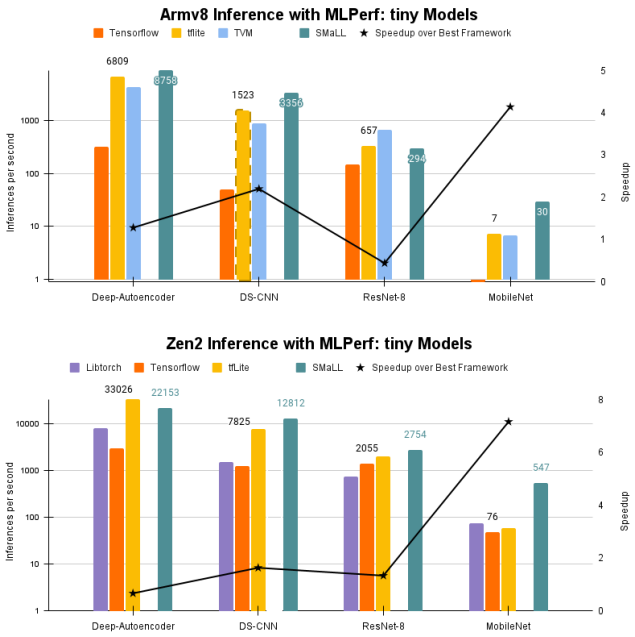


图 4：（左）与 TensorFlow TFLite 和 TVM 相比，SMaLL 的帧率相当或更高。虚线柱表示为 TFLite 修改后的神经网络，因为原始神经网络不受支持。TVM 在 ResNet-8 上取得了最佳性能，该神经网络主要由 3×3 卷积组成。（右）SMaLL 与最先进的库性能相当，但在许多情况下，SMaLL 的帧率最高。。

MobileNet（基于 LibTorch）和 DS-CNN 分别达到 1.67 倍的性能。使用 SMaLL 时，ResNet 比 TFLite 快 40%。自编码器模型比 TFLite 慢 33%。这可能是由于我们采用的简单并行化方案无法利用 Zen 2 上的 6 个线程。

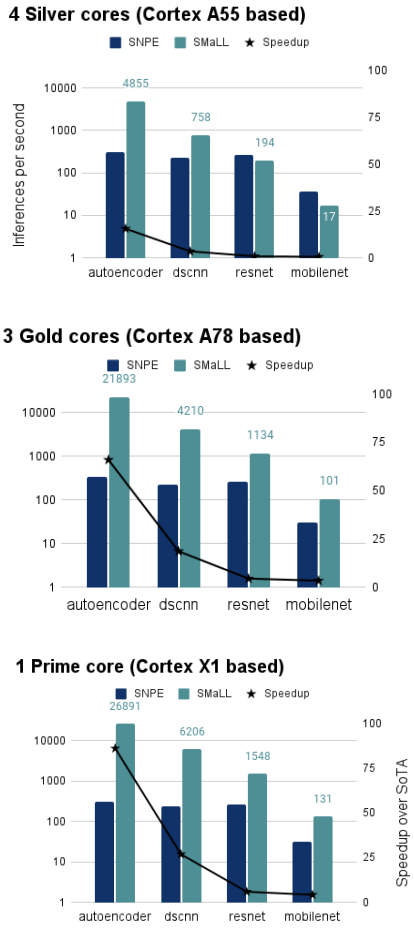


图 5：在 Kryo 680 的三种 CPU 上，SMaLL 的性能均优于 SNPE。SNPE 在 Kryo 680 的所有核心上的性能几乎相同。（左）与 SNPE 相比，SMaLL 在 4 个模型中的 2 个上速度较慢。这可能是因为该处理器采用顺序执行。（中）SMaLL 在黄金核心上实现了 3 到 65 倍的加速。（右）与 SNPE 的比较显示，SMaLL 在主核心上的性能比 SNPE 提高了 4 到 85 倍。

在移动中央处理器上，我们使用 SNPE 作为基线。SNPE 接受模型交换格式的预训练模型，然后将其优化并编译为适用于高通开发板的可执行文件（.dlc）。随后，我们使用 snpe-parallel-run 运行每个模型，并使用 -runtime-order cpu] 指定中央处理器为目标。

图 5 比较了 SMaLL 与 SNPE 在 3 种可用 CPU 上分别达到的帧率。在白银核心上（图 5 左），SMaLL 在自编码器和 DS-CNN 上的表现均优于 SNPE。较大的模型 ResNet 和 MobileNet 使用 SNPE 时速度更快。随着我们从吞吐量较低的核心转向吞吐量较高的核心，性能差距不断扩大，SMaLL 在所有模型上的表现都优于 SNPE。在黄金核心上，使用 SMaLL 时自编码器的帧率提高了 65 倍，MobileNet 的提升则较为温和，提速 3 倍。最后，在主核心上，SMaLL 的速度快 85 倍，MobileNet 的速度快 4 倍。

对逐层性能的分析表明，Cortex A55（顺序执行核心）的计算内核未能达到接近峰值的吞吐量，而 Cortex A78 和 Cortex X1 的表现则更接近峰值。这表明，重新设计 A55 的内核或许能够缩小 SMaLL 与 SNPE 之间的差距。具体而言，由于该处理器采用顺序执行，因此需要仔细安排指令，以实现更好的软件流水线 [39] 并隐藏延迟。理论上，编译器可以完成这项工作，但根据我们的经验，手动对内核进行流水线化通常会带来益处。

在全部 5 个平台上，我们看到 SMaLL 对几乎所有模型都能持续取得更好的性能；例外是树莓派上的 ResNet，使用单线程时，其每秒帧数为最快框架的 44%；以及 Zen 2 上的深度自编码器，其每秒帧数为 TFLite 的 67%。此外，即使模型架构具有相当大的层多样性，SMaLL 仍能取得良好的性能，这一优势可归因于抽象层的通用设计。

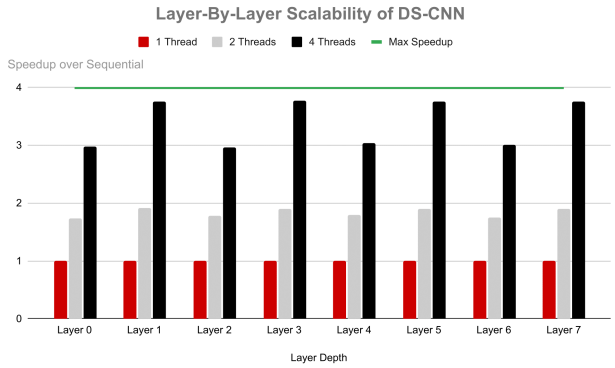
5.5 性能可扩展性

尽管本研究的重点并非对推理进行最优并行化，但边缘设备通常确实具备一定程度的多核并行能力，因此我们展示了 SMaLL 如何利用这种并行能力。我们评估的设备核心数各不相同：Zen 2 有 6 个核心，树莓派有 4 个核心，Kryo 680 CPU 总共有 8 个核心，其中包括 4 个大核、3 个小核和 1 个超大核；最后，Arduino Nano 有 1 个核心。

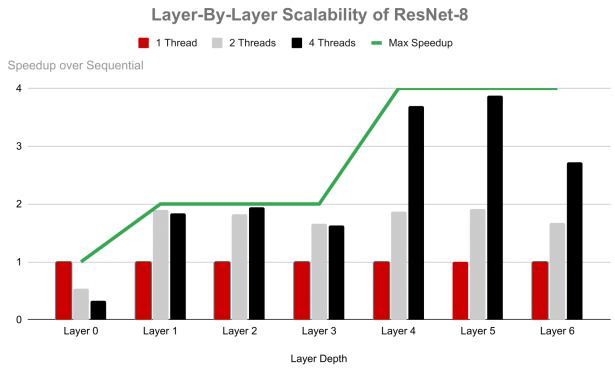
我们使用树莓派来展示 ResNet [21] 和 DS-CNN [36]（深度可分离卷积神经网络）中各层尺寸的可扩展性。此外，这两个神经网络的层数相近，问题规模也相似。DS-CNN 使用 3×3深度卷积（图 6a 中的偶数层）和1×1 卷积（图 6a 中的奇数层）。ResNet 主要由 3×3 卷积构成，第 3 层和第 6 层则使用 1×1 卷积。

我们并行化输出张量的（分块）通道维度。并行化所能带来的加速比受该维度大小的限制。图 6 中的“最大加速比”线展示了这一限制，该线假设单线程性能呈线性扩展。例如，在 DS-CNN（图 6a）中，所有层都具有 64 个通道。块大小为 16 时，树莓派上的 4 个核心都可以用于并行处理该问题，因此最大加速比为单线程执行时间的 4 倍。另一方面，ResNet 起始时通道数较少（16 个），因此无法并行化。第 1 至第 3 层有 32 个通道，最大加速比为 2 倍；第 4 至第 6 层有 64 个通道，最大加速比可达 4 倍。我们选择类似卷积的层来展示可扩展性，因为它们占端到端推理时间的大部分。不过，由于采用了统一循环，我们在所有层中使用相同的并行化方案。

在 DS-CNN 中，所有层都能从使用 4 个线程中受益。在图 6a 中，1×1 卷积的性能与预期最大加速比相差不到 7%。深度卷积层的扩展速度较慢，使用 4 个线程时，平均加速比为 3 倍。这可能是因为



(a) DS-CNN 中所有层可利用的并行度都相同。



(b) 在 InResnet 中，不同层可供 SMaLL 利用的并行度各不相同

图 6：研究 SMaLL 在树莓派上利用 4 个可用核心的可扩展性。可用并行性能够得到利用的程度，取决于每一层输出的维度。

这些层的总体计算量较少，无法充分摊销并行化的成本。

对于预期加速比小于 4 的 ResNet 层，我们发现，尝试使用超出所需数量的线程会造成严重的负面影响。无论问题规模如何，3×3 卷积层的加速比都能达到最大加速比的 6% 以内。1×1卷积的扩展速度较慢，平均只能达到最大加速比的 75%。这可能是因为每个输出元素对应的 1×1 计算量少于 3×3 卷积。此外，层步幅为 2，导致输入元素的重复利用率较低，因此需要以更快的速率从较低层级的内存中加载输入，才能计算相同数量的输出元素。与 DS-CNN 中的深度卷积一样，并行化的成本会削弱其带来的收益。

我们发现，给定层所采用的并行度取决于输出张量的维度。此外，线程配置过多可能会降低层的性能。这表明，在端到端场景中，线程数量的选择需要权衡。我们将在第 5.4 小节中看到应用这些权衡后的结果。

5.5.1 整个模型的并行化。在 5.5 节中，我们展示了每个层都有可用的并行度上限，这限制了用于并行化该层的线程数。我们将这个上限称为 p_{avail}。我们还发现，单个神经网络中的不同层从不同程度的并行化中受益。在本小节中，我们研究可用并行度的这种差异如何影响端到端神经网络需要并行化的程度。为此，我们为一次推理可用的并行度设定上限 p_{max}。每个层使用 $p = min(p_{avail}, p_{max})$ 给定的线程数 p 进行并行化。图 7 展示了最低推理

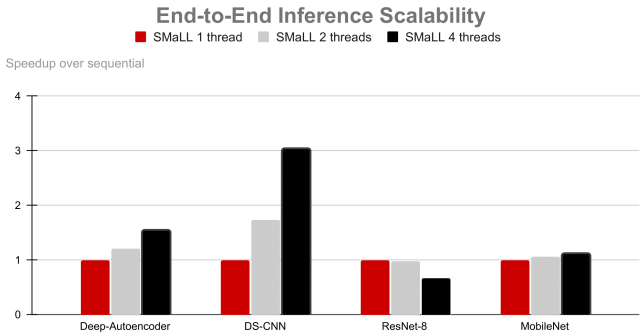


图 7：使用多个线程进行端到端推理所带来的加速。红色柱表示单线程。ResNet 中并行度的差异导致单线程推理速度最快。其他所有模型都能从使用全部 4 个线程中受益。

使用不同并行度上限运行 100 次所需的时间 $p_{max} = 1,2,4$。最快的配置以黑色轮廓突出显示。正如第 5.5 节所示，DS-CNN 在 4 个线程下性能最佳，因为所有层中的各线程通道数都相同。自编码器的结构也类似，其中 8 层中的 7 层通道数相同，均为 128。MobileNet 的通道数变化更大，从 32 到 1024 不等。不过，除第一层外，其余各层都有一个 $p_{avail} \geq 4$。因此，MobileNet 在 4 个线程上并行化时性能也最佳。例外是 ResNet：尽管 7 层中有 6 层可从不同程度的并行化中受益，但端到端性能最佳的配置却是单线程。这可能是因为每层的 p_{avail} 不同，导致各线程之间的工作负载不均衡。

5.6 量化推理

MLPerf:Tiny基准测试的重点是小型、资源受限的设备。本小节将展示，SMaLL 可轻松移植到采用基于整数的推理的微控制器平台。我们在一块 Arduino Nano 33 BLE 开发板上展示实验结果，该开发板搭载以 64MHz 运行的 32 位 Arm Cortex-M4 处理器，并配有 256KB 静态随机存取存储器。我们将 SMaLL 运行量化网络的性能与 TVM v0.10.0 的结果进行比较。请注意，图 8 未包含 MobileNet，这是因为其内存需求超过了 Nano 的 256K 限制。虽然 Nano 确实

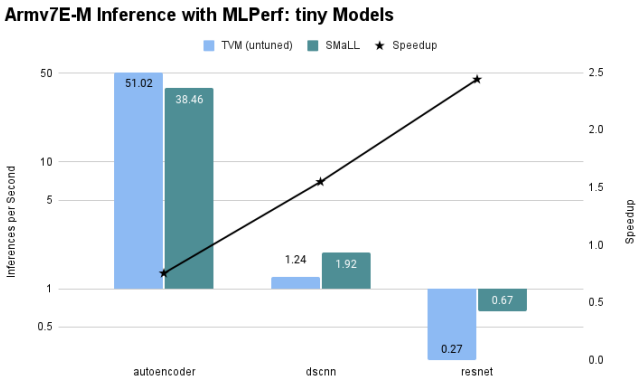


图 8：对于 Armv7E-M 平台，SMaLL 的帧率与 TVM 相当或更高

配有浮点单元（FPU），但浮点模型的内存需求往往会超过内存限制。

在所有情况下，SMaLL 的性能都与 TVM 相当；其中，自编码器的帧率达到 TVM 的 75%，而 MobileNet 的速度则快 2.4 倍。

我们还尝试与 TFLite 进行比较，但通过 Arduino IDE 上传量化模型后，开发板变得无响应。这可能与 TFLite 的整体峰值内存消耗有关；如表 5 所示，该消耗超过了 Nano 的 256KB SRAM。

6 结论与未来方向

在这项工作中，我们介绍了 SMaLL 框架，这是一个开放且可移植的框架，可快速为不同架构实例化常见的深度神经网络层。SMaLL 框架的核心是利用这样一个洞见：许多机器学习层可以通过单个抽象层统一起来。该抽象层可以通过单个循环嵌套和通用数据布局来实现，从而最大限度地减少为每个新层和不同架构编写的代码。各层采用通用布局，也消除了数据布局转换所需的内存开销，从而有助于在内存受限设备上部署更大的机器学习网络。

我们证明，尽管 SMaLL 框架具有模块化和高水平的代码复用特性，其实现的性能仍可与最先进的机器学习框架相媲美，无论是在单个层上，还是在真实模型的端到端推理中。此外，我们还展示了当前机器学习框架在不同架构上的性能特征差异很大，这可能是因为它们依赖供应商优化库。另一方面，我们的框架支持将实现快速移植到不同架构，从而让各种设备都能普遍使用先进的机器学习技术。

虽然我们展示了 6 种不同架构上的性能，但这份清单并不全面。正如我们在其他数值领域取得的成功所表明的那样，我们的方法可以轻松扩展到不同的平台 [13]。我们正在积极研究添加专用核以支持更多种类的边缘计算平台，例如 Nvidia Jetson [40] 和 Qualcomm Hexagon [41]。

我们计划利用该框架探索的一个未来方向，是实现跨层优化。请注意，所有层都具有相同的循环结构，并采用相同的数据布局。各层通用的数据格式尤其有利于识别和应用融合，因为它简化了融合数据依赖分析。抽象层的概念可以扩展为抽象的“融合”层。这样一来，凡是符合模板的多个层都可以融合，而不必为每种层组合编写不同的实现。

我们认为，SMaLL 框架所依据的原则（通用数据布局，以及围绕高性能内核构建的循环嵌套）能够支持其他神经网络中更复杂的层，例如 Transformer [42, 43] 中的注意力层。举例来说，注意力层可以表示为：先对查询和键进行缩放矩阵乘法（即一种专用的完整层），再执行 Softmax（由一个单元素层实现，后接一个用于计算指数值之和的单通道层），最后将结果与值向量进行另一次矩阵乘法（即另一个完整层）。这些层单独来看，都可以用 SMaLL 中的专用层来描述。然而，要实现高性能，仍需按照上一段所述跨多个层进行融合。

致谢

本项工作部分由美国国防高级研究计划局（DARPA）TRIAD 计划根据协议编号 HR00112190099 资助，并基于美国国防部根据合同编号 FA8702-15-D-0002 为卡内基梅隆大学运营软件工程研究所（一个由联邦政府资助的研究与发展中心）提供资金和支持的工作。本项工作还部分得到了 Meta 的赠款资助。

本材料所载观点、意见和／或研究结果仅代表作者的立场，不应被视为政府的官方立场、政策或决定，除非其他文件另有说明。

本文提及任何特定商业产品、流程或服务的商品名称、商标、制造商或其他信息，并不一定构成或暗示卡内基梅隆大学或其软件工程研究所对其认可、推荐或偏好。 [发行声明 A]本材料已获准公开发布和无限制传播。有关非美国政府用途和传播的信息，请参阅版权声明。 [DM22-1168]

参考文献

[1] José E. Moreira、Kit Barton、Steven Battle、Peter Bergner、Ramon Bertran、Puneeth Bhat、Pedro Caldeira、David Edelsohn、Gordon C. Fossum、Brad Frey、Nemanja Ivanovic、Chip Kerchner、Vincent Lim、Shakti Kapoor、Tulio Machado Filho、Sil- via Melitta Mueller、Brett Olsson、Satish Sadasivam、Baptiste Saleil、Bill Schmidt、Rajalakshmi Srinivasaraghavan、Shricharan Srivatsan、Brian W. Thompto、An- dreas Wagner 和 Nelson Wu。适用于 Power ISA(TM) 处- 理器的矩阵数学单元。*CoRR*，abs/2104.03142，2021年。[2] 英伟达公司。*NVIDIA A100 Tensor Core GPU* 架构。[3] Albert Reuther、Peter Michaleas、Michael Jones、Vijay Gadepally、Siddharth Samsi 和 Jeremy Kepner。机器学习加速器综述。*CoRR*，abs/2009.00993，2020年。

[4] Martín Abadi、Ashish Agarwal、Paul Barham、Eugene Brevdo、Zhifeng Chen、Craig Citro、Greg S. Corrado、Andy Davis、Jeffrey Dean、Matthieu Devin、San- jay Ghemawat、Ian Goodfellow、Andrew Harp、Geoffrey Irving、Michael Isard、Yangqing Jia、Rafal Jozefowicz、Lukasz Kaiser、Manjunath Kudlur、Josh Levenberg、Dandelion Mané、Rajat Monga、Sherry Moore、Derek Murray、Chris Olah、Mike Schuster、Jonathon Shlens、Benoit Steiner、Ilya Sutskever、Kunal Talwar、Paul Tucker、Vincent Vanhoucke、Vijay Vasudevan、Fernanda Viégas、Oriol Vinyals、Pete Warden、Martin Wattenberg、Martin Wicke、Yuan Yu 和 Xiaoqiang Zheng。TensorFlow：异构系统上的大规模机器学习，2015年。软- 件可从 tensorflow.org 获取。[5] Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gi melshein、Luca Antiga、Alban Des- maison、Andreas Kopf、Edward Yang、Zachary DeVito、Martin Raison、Alykhan Tejani、Sasank Chilamkurthy、Benoit Steiner、Lu Fang、Junjie Bai 和 Soumith Chintala。Pytorch：一种命令式、高性能深度学习库。载于《神经信息处理系统进展第32卷》，第8024–8035页。Curran Associates, Inc.，2019年。[6] Intel® oneAPI 深度神经网络库。可访问 https://www.intel. co m/content/www/us/en/developer/tools/oneapi/onednn.html（2022年10月31日）。[7] Liangzhen Lai、Naveen Suda 和 Vikas Chandra。CMSIS-NN：适用于 Arm Cortex-M CPU 的高效神经网络内核。CoRR，abs/1801.06601，2018年。[8] Sergio Barrachina、Adrián Castelló、Manuel F Dolz、Tze Meng Low、Héctor Martínez、Enrique S Quintana-Ortí、Upasana Sridhar 和 Andrés E Tomás。面向 Arm 处理器上高性能深度学习推理的直接卷积重- 构。《系统架构杂志》，135:102806，2023年。[9] Kumar Chellapilla、Sidd Puri 和 Patrice Simard。用于文档处理的高性能卷积- 神经网络。载于 Guy Lorette（编辑），第十届国际手写识别前沿研讨会，拉博勒（法国），2006年10月。雷恩第一大学，Suvisoft。http://www.suvisoft.com。[10] Tianqi Chen、Thierry Moreau、Ziheng Jiang、Lianmin Zheng、Eddie Yan、Meghan Cowan、Haichen Shen、Leyuan Wang、Yuwei Hu、Luis Ceze、Carlos Guestrin 和 Arvind Krishnamurthy。TVM：用于深度学习的自动化端到端优化编译器，2018年。[11] Chris Leary 和 Todd Wang。XLA：已编译的 TensorFlow。2017年。[12] Nadav Rotem、Jordan Fix、Saleem Abdulrasool 等。Glow：用于神经网络的图降阶编译器技术。arXiv 预印本 arXiv:1805.00907，2018年。[13] Field G. Van Zee 和 Robert A. van de Geijn。BLIS：用于快速实例化 BLAS 功能的框架。《ACM 数学软件汇刊》，41(3)，2015年6月。[14] C. L. Lawson、R. J. Hanson、F. T. Krogh 和 D. R. Kincaid。算法539：供 Fortran 使用的基本线性代数子程序 [f1]。《ACM 数学软件汇刊》，5(3):324–325，1979年9月。[15] Jack J. Dongarra、Jeremy Du Croz、Sven Hammarling 和 Richard J. Hanson。一组扩展的 Fortran 基本线性代数子程序。《ACM 数学软件汇刊》，14(1):1–17，1988年3月。[16] Jack J. Dongarra、Jeremy Du Croz、Sven Hammarling 和 Iain Duff。一组第3级基本线性代数子程序。《ACM 数学软件汇刊》，16(1):1–17，1990年3月。[17] Kazushige Goto 和 Robert A. Van De Geijn。高性能矩阵乘法剖析。《ACM 数学软件汇刊》，第1–25页，2008年–。[18] Field G. Van Zee、Tyler Smith、Francisco D. Igual、Mikhail Smelyanskiy、Xianyi Zhang、Michael Kistler、Vernon Austel、John Gunnels、Tze Meng Low、Bryan Marker、Lee Killough 和 Robert A. van de Geijn。BLIS 框架：可移植性实- 验。《ACM 数学软件汇刊》，42(2):12:1–12:19，2016年6月。[19] Tze Meng Low、Francisco D. Igual、Tyler M. Smith 和 Enrique S. Quintana-Ortí。解析建模足以实现高性能 BLIS。《ACM 数学软件汇刊》，43(2):12:1–12:18，2016年8月。[20] Karen Simonyan 和 Andrew Zisserman。用于大规模图像识别的超深卷积网络。载于国际学习表征会议，2015年。[21] Kaiming He、Xiangyu Zhang、Shaoqing Ren 和 Jian Sun。用于图像识别的深度残差学习。CoRR，abs/1512.03385，2015年。[22] Christian Szegedy、Wei Liu、Yangqing Jia、Pierre Sermanet、Scott Reed、Dragomir Anguelov、Dumitru Erhan、Vincent Vanhoucke 和 Andrew Rabinovich。通过卷积网络走向更深层。载于2015年 IEEE 计算机视觉与模式识别会议（CVPR），第1–9页，2015年。[23] Jiyuan Zhang、Franz Franchetti 和 Tze Meng Low。高性能零-内存开销直接卷积。载于国际机器学习会议，第5776–5785页。PMLR，2018年。[24] Tze Meng Low、Francisco D. Igual、Tyler M. Smith 和 Enrique S. Quintana-Ortí。解析建模足以实现高性能 BLIS。《ACM 数学软件汇刊》，43(2):12:1–12:18，2016年8月。[25] Robert Triggs。高通骁龙 888 深度解析：你需要了解的一切。[26] 高通公司。骁龙 888 规格。[27] Intel® Intrinsics 指南。可访问 https://www.intel.com/content/www/us/ en/docs/intrinsics-guide/index.html（2022年10月31日）。

[28] Tianqi Chen、Lianmin Zheng、Eddie Yan、Ziheng Jiang、Thierry Moreau、Luis Ceze、Carlos Guestrin 和 Arvind Krishnamurthy。学习优化张量程序。神经信息处理系统进展，31，2018年。[29] TensorFlow Lite | 移动端和边缘设备机器学习。网址：https://www. tensorflow.org/lite/（2022年2月4日）。[30] 高通公司。骁龙神经处理引擎 *SDK*。[31] Arduino。*Arduino IDE*。[32] Arduino。*Arduino* 命令行界面。[33] Colby Banbury、Vijay Janapa Reddi、Peter Torelli、Jeremy Holleman、Nat Jeffries、Csaba Kiraly、Pietro Montino、David Kanter、Sebastian Ahmed、Danilo Pau 等。MLPerf Tiny 基准测试。神经信息处理系统数据集与基准测试专题论文集，2021年。[34] L Sifre 和 Stéphane Mallat。用于纹理分类的刚性运动散射 [j]。计算机科学，3559：501–515，2014年。[35] Andrew G. Howard、Menglong Zhu、Bo Chen、Dmitry Kalenichenko、Weijun Wang、Tobias Weyand、Marco Andreetto 和 Hartwig Adam。MobileNets：面向移动视觉应用的高效卷积神经网络。*CoRR*，abs/1704.04861，2017年。[36] Peter Mølgaard Sørensen、Bastian Epp 和 Tobias May。一种用于嵌入式系统关键词检测的深度可分离卷积神经网络。*EURASIP* 音频、语音与音乐处理期刊，2020(1)：1–14，2020年。[37] Keras。网址：https://keras.io/（2022年10月31日）。[38] Tensorflow。使用 *TF-Lite* 进行性能测量。[39] Monica Lam。软件流水线：一种适用于 VLIW 机器的有效调度技术。载于*ACM SIGPLAN 1988*年程序设计语言设计与实现会议论文集，第318–328页，1988年。[40] Advanced AI Embedded Systems。网址：https://www.nvidia.com/en-us/ autonomous-machines/embedded-systems/（2022年10月31日）。[41] Hexagon DSP 处理器。网址：https://developer.qualcomm.com/software/ hexagon-dsp-sdk/dsp-processor（2022年10月31日）。[42] Jacob Devlin、Ming-Wei Chang、Kenton Lee 和 Kristina Toutanova。BERT：用于语言理解的深度双向变换器预训练。*arXiv* 预印本 *arXiv:1810.04805*，2018年。[43] Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N Gomez、Łukasz Kaiser 和 Illia Polosukhin。注意力机制就是你所需的一切。神经信息处理系统进展，30，2017年。