面向多核NPU的张量切片与优化

Rafael Sousa 巴西坎皮纳斯，坎皮纳斯州立大学计算机研究所 rafael.sousa@ic.unicamp.br

Marcio Pereira 巴西坎皮纳斯，坎皮纳斯州立大学计算机研究所 mpereira@ic.unicamp.br

Yongin Kwon 和 Taeho Kim 韩国电子通信研究院（ETRI）韩国首尔{yongin.kwon, taehokim}@etri.re.kr

Namsoon Jung 和 Chang Soo Kim SilicoNeuro/AiM Future 韩国首尔{yongin.kwon, changsoo.kim}@siliconeuro.com

Michael Frank MagiCore 美国圣克拉拉 michael@magicore.com

GuidoAraujo IC-UNICAMP 坎皮纳斯，巴西 guido@unicamp.br

arXiv:2304.03013v1 [cs.PF] 6 Apr 2023

摘要—尽管卷积神经网络（CNN）模型的代码生成已得到广泛研究，但对于高度受限的多核神经处理单元（NPU）执行高效的数据切片和并行化仍然是一个具有挑战性的问题。鉴于卷积输入/输出张量的规模以及NPU片上存储器的有限容量，在最大化并行性和MAC利用率的同时最小化内存事务，是任何有效解决方案的核心。本文提出了一种面向多核NPU的TensorFlow XLA/LLVM编译器优化过程，称为张量切片优化（TSO），其功能包括：（a）最大化NPU核心之间的卷积并行性和存储器使用率；以及（b）利用动态随机存取存储器（DRAM）内存突发时间估算来指导张量切片，从而减少主机与NPU片上存储器之间的数据传输。为评估所提出的方法，研究人员使用神经形态处理器（NMP）开展了一组实验。该处理器是一种包含32个经过新型CNN指令扩展的RISC-V核心的多核NPU。实验结果表明，TSO能够识别出使一组CNN模型的执行时间最小化的最佳张量切片方案。与无突发数据切片方法相比，采用基于突发的TSO技术时，速度最高提升了21.7%。为验证TSO方法的通用性，还将该算法移植到了Glow机器学习框架中。在Glow和TensorFlowXLA/LLVM编译器上分别测量了这些模型的性能，结果显示出相似的表现。

索引词—**基于突发传输的模型、卷积神经网络、神经处理单元、映射策略**

I. 引言

使用卷积神经网络（卷积神经网络）进行深度学习已成为一种重要的架构模型技术，显著提高了许多现代人工智能应用的准确率。卷积神经网络的采用率稳步增长，主要由计算机视觉领域的应用推动，这些应用解决了对象识别 [1]–[3],对象检测 [4], [5], 和视频分类 [6], [7]等问题。语音识别和自然语言处理（NLP）等其他领域也受益于CNN模型的应用 [8], [9]。

随着准确率的提升，最先进卷积神经网络的规模和复杂度也显著增长。例如，LeNet-5 [10], 是一种用于识别手写数字的模型，参数量不到 100 万；而更复杂的模型（如 InceptionV3 [11] ）可对数千种不同的对象类别进行分类，参数量超过 2,300 万。模型复杂度和参数规模的增长不仅需要更强大的计算能力，还会显著增加主机（片外）与人工智能加速器（片上）存储器之间的数据移动，从而大幅影响能耗 [12]和内存流量。

众所周知，卷积是卷积神经网络中开销最大的操作，占据卷积神经网络执行过程中的最大比重。鉴于其张量输入的大小以及配置参数的多样性（例如内核大小、步幅等），选择能够最大化卷积并行度、同时最小化内存事务的最佳数据映射，是影响任何人工智能加速器性能的关键因素。对于多核神经处理单元（NPU）而言，这一点尤为重要，因为它们受到严格的（片上）存储器限制，并且需要实现较高的推理吞吐量。

为此，需要将卷积输入张量和权重划分为适合神经处理单元片上存储器的切片。切片从（较慢的）外部 DRAM 搬移到（较快的）片上存储器。随后，每次使用一组切片来执行卷积。切片形状和大小的选择方式不同，卷积执行时间可能会有显著差异。例如，参见图 1，该图展示了使用不同形状的切片执行卷积所需的时间。在此示例中，输入张量是单通道，包含 128×128 16 位定点元素（行主序），并通过单个大小为 1×1 的卷积核进行计算。图中，切片以浅灰色和深灰色区域表示，每个红点代表一次对 DRAM 的 (128B)存储器突发 访问。DRAM 的访问时间可分为两个部分：(a) 列地址选通延迟，即从动态随机存取存储器行缓冲区读取存储器突发的第一个字节所需的时间；(b) 访问延迟，即读取突发传输后续字节所需的时间。例如，读取典型 DDR3 内存 128B 突发传输的第一个字节需要 128×128 1614ns，读取该突发传输剩余的 127 个字节所需的时间也相同。

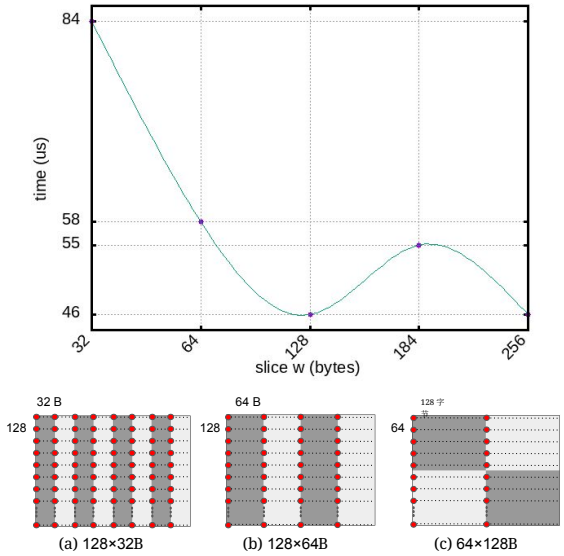


图 1：不同切片形状下的内存访问。

在典型 DDR3 内存中，从 128B 突发传输读取第一个字节需要 ~14ns，读取该突发传输剩余的 127 个字节所需的时间也相同。数据切片方式不同，存储器突发传输可能会对执行时间产生巨大影响。例如，在图 1 中，卷积可分为：(a)8 $128 \times 32B$ 个切片，对应 1024 次突发传输（红点），执行时间为 84us；(b) 4 $128 \times 64B$ 个切片，对应 512 次突发传输，执行时间缩短至 58us；以及 (c) 4$64 \times 128B$ 个切片，需要 256 次突发传输，执行时间为 46us；与 (a) 中的切片方式相比，卷积时间减少了 45%。在图 1 中，切片方式 (c) 在 w = 128B 时的内存访问时间最短。从此处开始，随着切片宽度 (w) 持续增加，内存访问时间会变长，随后在下一个存储器突发对齐点 (w = 256) 再次缩短。

尽管内存访问合并是图形处理器代码生成中的常见问题，但在多核NPU并行性的背景下尚未得到研究。本文提出了一种面向多核NPU的编译器优化方法，称为张量切片优化（TSO），其目标有两个：（a）最大限度地利用可用NPU核心的存储器并行执行卷积；（b）减少主机与各核心片上内存之间的数据传输。该方法通过在编译时对各NPU核心的存储器利用率进行建模，搜索能够最大限度减少主机与NPU核心存储器之间数据传输的最佳输入/输出张量切片方案来实现。为评估该方法，我们使用包含32个核心的多核NPU NeuroMorphic处理器（NMP）以及TensorFlow XLA LLVM编译工具链进行了一系列实验。

本文的组织如下。第二节介绍背景综述。第三节详细介绍 NMP加速器。第四节展示如何使用 TSO 算法将卷积层映射到 NMP 上。第五节介绍

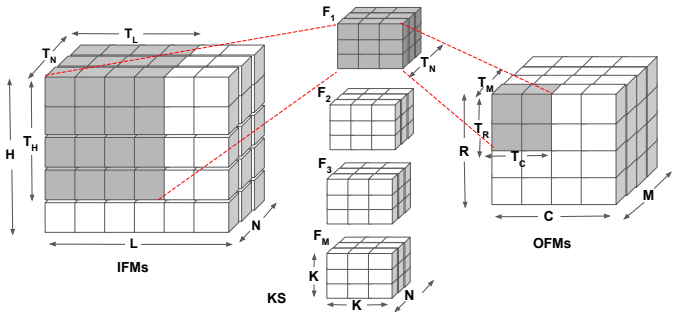


图 2：分块卷积层。

使用 TF-XLA编译器的编译流程。第六节展示实验结果。第七节分析与本文相关的工作，最后，第八节总结本文工作。

II. 背景

CNN模型可视为一个有向无环图，由多个运算层组成，用于处理一组输入通道（例如图像）。经过非常深的层级结构处理后，会得到输出。输出结果通常是一个概率分布数组，用于将给定图像分类到一组类别中（例如狗、汽车等）。本工作的重点是对已经预训练的模型进行推理。

在组成CNN的所有可能层中，卷积层占模型执行时间的90%以上 [13]，并产生大量数据移动。对于采用小型片上存储器的架构而言，这一点尤为关键，例如用于在移动和嵌入式设备上执行推理的NPU。因此，本文提出的数据分块等方法是减少此类设备内存事务的关键技术。

如图 2 所示，一个卷积层使用大小为 H （高度） $\times$ L （宽度）的 N 输入特征图（输入特征图）以及一组预训练权重。权重集合（KS）是一组大小为$N \times K \times K$的M 多维（例如三维）数组卷积核/滤波器。每个滤波器以 S的步长因子在输入特征图上滑动，执行三维卷积。在整个输入图像（输入特征图）上滑动后，会生成一个 $R \times C$ 输出特征图（输出特征图）。将 M 中的所有 KS 滤波器应用于输入特征图后，会得到一组 M 输出特征图。

卷积神经网络中的不同卷积层通常具有不同的内核大小、数量各异且尺寸不同的输入特征图和输出特征图，以及可变的步幅。例如，Inception-V3 [11] 的第一个卷积层的输入包含 $3 \times 299 \times 299$ 个元素。若将输入的数据类型视为 16 位定点，则输入大小为 524KB。对于典型 NPU 中有限的片上存储器，一次性存储这么多数据并不现实。因此，数据分块是卷积层计算中的必要步骤。

卷积层的数据分块包括将其输入特征图和输出特征图数据映射划分为较小的分块，并将滤波器划分为 KS，以便每个 IN、OUT 和 W 分块能够同时装入 NPU 的片上存储器。图 2 展示了输入特征图和输出特征图数据映射如何分别划分为$IN = (T_N, T_H, T_L)$个分块和 $OUT = (T_M, T_R, T_C)$个分块。请注意，因此可以通过公式 1，利用输出特征图的维度 T_R 和 T_C 计算输入特征图的维度 T_H 和 T_L。

$$\begin{aligned} T_H &= (T_R - 1)S + K \\ T_L &= (T_C - 1)S + K \end{aligned} \tag{1}$$

其中，K 和 S 分别是卷积核的内核大小和步幅。此外，如果 KS 中的滤波器/卷积核无法装入各自神经处理单元的片上存储器，则 KS 也会被划分为$W = (T_M, T_N, K, K)$个分块，其中T_M是滤波器的数量，T_N 是每个滤波器中要加载的通道数量。

在神经处理单元上执行卷积时，可以探索不同的分块形状。每种分块形状都会导致不同的内存访问模式，并影响神经处理单元可用资源的使用情况。此外，还可以探索不同的调度策略；每种策略都有特定的内存访问模式，从而带来不同的数据复用方式。计算映射方式会显著影响主机与神经处理单元存储器之间的数据移动，进而导致数据复用率低下。此外，如果分块处理不当，还会导致神经处理单元的乘累加器（MAC）单元利用率低下，使其在计算过程中处于闲置状态（更多详情见第 IV 节）。

III. NMP架构

虽然已有方案提出利用多核并行和 SIMD 指令 [14], [15],来加速卷积神经网络，但现代 CNN 模型中的层数量和复杂度使得在 CPU 上运行整个网络变得非常困难。为提高推理吞吐量，已有方案提出使用（快速）图形处理器处理大量数据 [16],[17]。另一方面，现场可编程门阵列（FPGA）因具备良好的性能和可重构性[18]–[22]，已被广泛用作解决这一问题的替代方案。然而，对于监控摄像头和手机人脸识别等对执行效率和功耗有严格要求的关键边缘应用而言，这些架构并非高能效的解决方案。已有多种类型的加速器被提出，用于以高能效方式加速卷积神经网络。专用集成电路[23], 、神经处理单元（NPU） [24], [25], 和张量

处理单元（TPU） [26] 是一些示例。

本文以 LG Electronics（LGE）的神经形态处理器（NMP）作为编译目标。NMP架构的核心思想是使用 RISC-V指令集扩展来设计相关的卷积神经网络指令，例如卷积层、全连接层、池化层、逐元素运算等。

NMP架构（图 3）是一种多核NPU，包含一个 ARM57处理器，作为一组多个 Tile（TLE）处理器的主机，每个处理器都包含一组 Tilelet

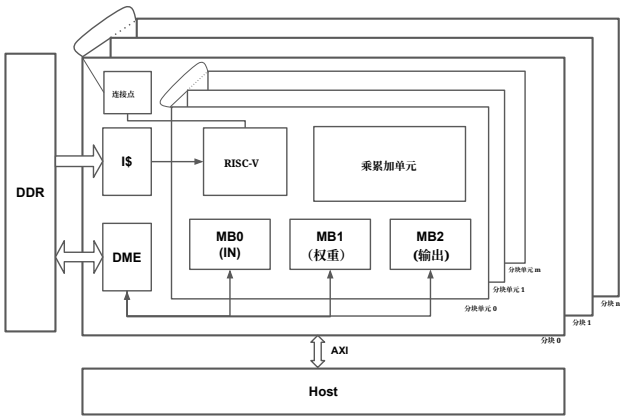


图 3：NMP架构。

(TLT) 核心。每个 TLT 都有一个 RISC-V核心和三个片上（暂存）存储器，即 MB0、MB1 和 MB2，它们分别存储来自输入特征图、卷积核和输出特征图数据映射的 IN、W 和 OUT 分块。此外，每个 TLT 还配备了一个乘加加速单元，用于执行卷积神经网络运算。乘累加单元由 RISC-V核心触发执行，并能够执行 8 位和 16 位定点运算，存储器布局采用 NCHW格式。乘累加单元与 TLT 片上存储器（MBLOB，即 MB0、MB1 和 MB2）之间的数据通路宽度为 128 位，这意味着对于 8 位定点，每个周期最多可执行 16 次乘加运算；对于 16 位定点，每个周期最多可执行 8 次乘加运算。

NMP 与主机之间的数据传输通过数据搬移引擎（DME）模块完成，如图 3 所示。主机通过 AXI接口与 NMP 通信，不同 TLE 的 TLT 之间可通过全局环形总线共享数据。一个 TLE 中的 TLT 也有各自的环形总线，用于彼此传输数据。RISC-V核心执行的指令从主机内存中获取，并存入指令缓存；同一 TLE 中的 TLT 共用该指令缓存。

要在 NMP 上执行任何计算，首先要使用 TF-XLA编译器编译模型。编译后的模型在每个 TLT 的 RISC-V核心上启动执行，各核心彼此独立运行。要调用乘累加单元执行计算，RISC-V核心会运行以下扩展指令之一：nmp conv2d、nmp veop、nmp pool、nmp activation 和 nmp percept。要将数据从动态随机存取存储器加载到 TLT 的片上内存，RISC-V核心必须执行数据搬移指令（nmp load 和 nmp load3d），从而调用 DME 完成这项工作。执行 RISC-V 扩展指令 nmp store 和 nmp store3d，可将数据从各 TLT 的片上内存传回主机动态随机存取存储器。

尽管单个 TLE 中的每个 TLT 都独立于其他 TLT 执行计算，但 NMP 支持一种特殊的多播加载指令。该指令仅在单个 TLT 上执行，却能使该 TLE 中的所有 TLT 加载

来自动态随机存取存储器的相同数据。例如，假设给定 TLE 中的所有 TLT 都处理相同的输入特征图，但各自使用不同的滤波器。无需多次加载（输入特征图）输入瓦片（即为 TLE 中的每个 TLT 分别加载一次），可以使用多播加载将相同的输入瓦片加载到所有 TLT 中，从而使其能够由不同的滤波器并行处理。请注意，来自不同 TLE 的 TLT 不能使用同一个多播来加载相同的数据。在这种情况下，需要为每个 TLE 分别执行多次多播加载，以将相同的数据加载到其中的所有 TLT。

NMP 中提供了三级基于硬件的信号量，用于在 TLE 和 TLT 之间进行同步。它们支持以下操作：(a) 同步 TLT 内部的计算（例如，调用乘累加单元执行操作后，可以阻塞 RISC-V 的执行，直到该单元完成工作）；(b) 同步同一 TLE 中不同 TLT 之间的计算（例如，在多播加载期间，特定 TLE 中的所有 TLT 都会被阻塞，直到各自对应的片上存储器从动态随机存取存储器接收到数据）；以及 (c) 同步不同 TLE 中的 TLT 之间的计算（例如，假设对于给定模型中的某一层，它依赖于前一层；如果某个 TLT 在同一层上的其他所有 TLT 尚未完成计算时就已完成计算，则必须将其阻塞，使其无法继续执行下一层）。

本文使用的 NMP架构由 4 个 TLE 组成，每个 TLE 包含 8 个 TLT （ RISC-V + 乘累加单元）。每个 TLT 都有三个片上存储器，每个容量为 8KB。在 1GHz 的工作频率下，NMP（全部 32 个 TLT）执行 8 位或 16 位定点运算时，理论峰值性能分别为 512 或 256 GMAC/s。动态随机存取存储器采用 DDR3，时钟频率为 1066MHz（DDR3-2133，17GB/s）。对于此版本的架构，NMP 不支持乘加/加载重叠。对于未来的 NMP架构，我们预计将增加额外的片上（双端口）存储器，使编译器能够对乘加/加载操作进行软件流水线处理。

IV. 近存计算映射策略

在卷积层执行期间，将数据从主机传输到加速器存储器有多种方式。如果处理不当，数据移动次数可能会大幅增加，从而影响性能和能耗 [12]。本文提出了一种名为张量切片优化 (TSO) 的搜索空间探索优化算法，旨在确定最佳的 TLE/TLT 数据划分，以最大限度地减少卷积层执行期间的内存传输次数。

A. TSO 算法

TSO 通过穷举搜索解空间，寻找能够最大限度缩短执行时间的最佳卷积平铺/调度策略。它首先将卷积的输入张量（输入特征图）及其对应的滤波器（卷积核）划分到 NMP 的各个 TLE 处理器上，使每个 TLE 计算卷积层输出的不同切片。之后，再将每个 TLE 切片进一步划分为多个分块

算法 1 选择最佳 TLE/TLT 映射

```
1: 函数 TSO(CONVS, #TLE, #TLT)
2:     #pragmaomp parallel
3:     #pragmaomp single
4:     对每个 conv ∈ CONVS 执行
5:         #pragmaomp task
6:             ▷令 conv =(IFM, KS, OFM)
7:             map[conv].bestTile.time ← ∞
8:             ▷令 PART_TLE = {KS, KS&OFM, OFM}
9:             对每个 p ∈ PART_TLE 执行
10:                ▷设 slice =(TLE_R, TLE_W)
11:                slice ← TLE切片(p, conv, #TLE)
12:                ▷令 PART_TLT = {IS, OS, WS}
13:                对每个 q ∈ PART_TLT 执行
14:                    ▷令 tile =(IN, W, OUT, time, schedule)
15:                    tile ← TLTTILING(q, conv, slice, #TLT)
16:                    如果 tile.time < map[conv].bestTile.time
   then
17:                        map[conv].bestSlice ← slice
18:                        map[conv].bestTile ← tile
19:    返回 map
```

以便将计算分配到相应 TLE 处理器的多个 TLT 核心上。下面将详细介绍 TSO 算法的这两个步骤。继续阅读之前，请注意，从现在起，凡是提到 切片 ，均指 TLE 数据划分；凡是提到 分块 ，均指 TLT 划分。

首先（参见算法 1），TSO 将模型的卷积集合（$CONVS$）以及架构的 TLE 数量（#TLE）和 TLT 数量（#TLT）作为输入（第 1 行）。随后，它遍历所有卷积（第 4 行），并初始化一个 map，用于存储该卷积对应的最佳 TLE 切片和 TLT 分块（第 7 行）。接着，对于 $PART_{TLE}$ 中所有可用的 TLE 切片策略 p（第 9 行，详情参见 IV-B 小节），算法调用函数 $TLESlicing$，将卷积的输入特征图和卷积核数据分配到各个 TLE 上。$TLESlicing$返回元组 $slice = (TLE_R, TLE_W)$，其中TLE_R 指由分配给 TLE 处理器的输入特征图切片生成的输出特征图部分（行），而 TLE_W 则是将在该 TLE 处理器上运行的卷积核滤波器子集。

请记住，NMP 中的每个 TLE 处理器都有一组 TLT核心，因此，对于第11 行生成的每个 TLE 切片，都需要将切片数据划分到其对应的TLT 核心之间。因此，对于每种 TLT 调度策略 q（第13行，详情请参见 IV-C 小节），TSO 通过调用 $TLTTiling$（第 15 行）计算当前 TLE 数据切片的最佳 TLT分块。此函数的输入为 TLT调度策略 q、卷积数据（$conv$）、当前 TLE $slice$ 以及 TLT 的数量（#TLT）。随后，该函数确定 TLE 数据在 TLT 核心之间的最佳分块方式。 $TLTTiling$函数返回元组 $tile =(IN, W, OUT, time, schedule)$，其中 IN、 OUT分别是分配给该 TLE 的TLT的 IFM和 OFM数据图的分块，而 W 是包含 KS中部分滤波器的一个分块。

该元组还会返回使用该特定组合计算卷积所需时间的估计值

用于实现最佳调度（$schedule$）策略的 TLE 切片和 TLT 分块（详情请参见第 IV-E 小节）。为此，它会考虑将IN和 W 分块从动态随机存取存储器加载到（片上）TLT 存储器 MB0（输入）和 MB1（权重）的开销，以及将 OUT 从 MB2 TLT（输出）存储器写回主机动态随机存取存储器所需的时间。此外，$time$ 还包括使用各种调度方案时，每种经评估的分区在乘累加单元上运行所需的时间（详情请参见第 IV-D 小节）。

从 $TLTTiling$ 返回后，TSO 会比较（第 16 行）当前评估的划分所估算的 $time$ 与迄今为止针对该特定卷积找到的最佳时间（$map[conv].bestTile.time$）。随后，它会将其存入相应的 map条目（即 $map[conv]$），并记录对应的 TLE 切片（第 17 行）和 TLT 分块（第 18 行）。最后，返回包含每个卷积最佳切片/分块的 map （第 19 行），以便代码生成器稍后据此为 TLT 核心生成并调度代码。

TSO 是一种算法，会针对模型中的每个卷积，穷举探索所有可能的张量切片方案所构成的解空间。因此，执行该算法可能需要很长时间，尤其是在卷积神经网络包含大量卷积时。由于估算一个卷积的执行时间与其他卷积相互独立，探索解空间的过程具有高度并行性。在本研究中，我们使用 OpenMP 任务并行来加速这一探索，通过并行模拟所有卷积的执行时间来实现。并行执行从第 2 行开始，此处创建一个线程池。执行到此处时，会从线程池中选取一个唯一的线程（第 3 行），为每个卷积创建一个任务（第 5 行）。随后，这些任务会分配给线程池中的各个线程，以并行计算卷积的 TLE/TLT 数据划分和调度。

B. TLE 划分

TSO 优化的第一步是将 KS 中的滤波器划分给各个 TLE，并确定所选滤波器将计算 OFM（行）的哪一部分。具体做法是按照算法 1 中定义的划分集合进行划分 – $PART_{TLE}$ ={$KS, KS\&OFM,\ OFM$}，其中 KS、KS& OFM 和 OFM 是由算法 2 计算得出的划分策略。

KS 划分 – 在第一种划分方案中（第 4 行），仅将 KS 中的卷积滤波器划分成多个切片，分配给各个 TLE（第 5 行）。就数据复制而言，OFM 的所有 $R \times C$元素都必须由 TLE 计算，因此运行时需要在每个 TLE 上加载完整的输入特征图。这种划分方案通常适用于 CNN模型的最后几个卷积层，因为这些层的滤波器数量以及通道深度都会增加。因此，划分滤波器可以减少动态随机存取存储器与 NMP 之间的数据传输。

卷积核和输出特征图分区– 第二种分区方案（第 7 行）将滤波器和输出特征图行都分配给各个 TLE。对于本文使用的 NMP（#TLE = 4），该方案将输出特征图行和卷积核中的滤波器分别切分为两组，然后将两者组合，为每个 TLE 生成一个切片。

算法 2 TLE 切片

```
1: 函数 TLESLICING(p, 卷积, #TLE)
2:     rows ← conv.R
3:     filters ← conv.M
4:     如果 p = KS 则
5:         TLE_W = ⌈filters/#TLE⌉
6:         TLE_R = rows
7:     如果 p = KS&OFM 则
8:         TLE_W = ⌈filters/(#TLE/2)⌉
9:         TLE_R = ⌈rows/(#TLE/2)⌉
10:    如果 p = OFM 那么
11:        TLE_W = filters
12:        TLE_R = ⌈rows/#TLE⌉
13:    return (TLE_R, TLE_W)
```

卷积核中的滤波器随后会与输出特征图行组合，为每个 TLE 生成一个切片。与第一种 TLE 划分方案相比，这种 TLE 分区方案减少了输入特征图的数据传输，但由于分配给各个切片的滤波器更多，滤波器的负载会增加。当卷积核和输入特征图的数据量相近时，这种方案通常效果更好。

输出特征图分区– 最后，第三种分区方式仅将输出特征图行分配给各个 TLE（第 12行）。鉴于卷积核中的滤波器必须由每个 TLE 加载，因此，与卷积核中的滤波器相比，输入特征图更大时，这种分区方案通常在前几个卷积层上效果更好。

由于用于收集本文实验数据的 NMP 板没有全局共享缓冲区（由 TLE 共享），因此我们在设计 TSO 时未考虑此功能。不过，由于不同 TLE 的输入特征图/卷积核的不同切片可能相同，因此可以轻松扩展 TSO 以考虑全局共享缓冲区。

C. TLT 划分

选择 TLE 分区方案后，通过调用算法 1 第 12 行的函数 $TLTTiling$ ，将每个 TLE 的工作负载分配给相应的 TLT。 $TLTTiling$ 将 TLT 调度策略（q）、卷积数据（$conv$）、算法 1 第 8 行得到的 TLE 切片（$slice$）以及每个 TLE 上的 TLT 数量（$\#TLT$）作为输入。随后，它生成元组（$IN,\ W,\ OUT,\ time,\ schedule$）作为输出，用于为 TLT 生成代码。

最初（参见算法 3），$TLTTiling$ 将变量 $bestTile.time$ 初始化为无穷大，因为它将存储该函数访问的所有可能分块中最小的（估计）执行时间。为此，由三个嵌套循环组成的序列（第4–6 行）生成值 T_R、 T_C 和 T_N ，用于探索可由 TLE 切片形成的所有可能的输入、权重和输出块形状。但在计算该TLE切片的输入块和输出块之前，需要将TLE_W 中的卷积滤波器分配到各个 TLT之间。这在第7 行完成，该行还确定能够装入 TLT的 MB1W存储器中的最大滤波器数量（T_M）；第 8 行则生成相应的 W 块。

算法 3 TLT 分块

```
1: function TLTTILING(q,卷积,slice,#TLT)
2:     ▷令 tile = (IN, W, OUT, time, schedule)
3:     bestTile.time ← ∞
4:     对于 T_R ← 1 至 slice.TLE_R 执行
5:         从 T_C ← 1 至 conv.C 执行
6:             对于 T_N ← 1 至 conv.N 做
7:                 T_M ← GETFILTERS(T_R, T_C, q, slice.TLE_W, #TLT)
8:                 W ← 詹蒂莱_W(T_M, T_N, conv, q)
9:                 OUT ← GENTILE_OUT(T_M, T_R, T_C)
10:                IN ← 詹蒂莱_IN(T_N, T_R, T_C, conv)
11:                (time, schedule) ← CALCTIME(IN, W, OUT, q)
12:                if time < bestTile.time then
13:                    bestTile ← (IN, W, OUT, time, schedule)
14:    return bestTile
```

在滤波器划分不均衡的情况下，剩余的滤波器会分配给 ID 最低的TLT。随后调用函数生成 OUT 分块（第 9行的 $GenTILE_{OUT}$）和 IN分块（第10行的$GenTILE_{IN}$）。如公式2所示，这两个函数还会检查分块 OUT 和 IN是否分别适合放入 TLT 的存储器 MB2（输出）和MB0（输入）中，其中类型表示8 位或16位定点。第8–10 行之间的函数还会计算需要从动态随机存取存储器中加载或将其存储回动态随机存取存储器的输入、输出和权重瓦片的次数，以覆盖TLE 切片的全部工作负载（更多详细信息请参见第 IV-C 小节）。

$$\begin{cases} IN = T_N \times T_H \times T_L \times type \le MB0 \\ W = T_M \times T_N \times K \times K \times type \le MB1 \\ OUT = T_M \times T_R \times T_C \times type \le MB2 \end{cases} \tag{2}$$

分块 IN, W, OUT和分块策略 q随后被传递给函数 $CalcTime$ （第 11行），以便其估计使用生成的分块计算 TLE 切片所需的最佳 $schedule$ 和 $time$ （更多详情见第 IV-E小节）。最后，算法测试当前分块所计算的 $time$ 是否小于迄今为止看到的 $bestTile.time$ ，如果是，则更新 $bestTile$。

D. 调度

由于数据已在 TLE/TLT 之间进行划分，TLT核心可以采用不同的调度策略——输入驻留（IS）、输出驻留（OS）和权重驻留（WS）——来执行卷积，每种策略都会产生不同的内存访问模式（$schedule$）。鉴于本文所述映射策略的数据传输可以静态确定，我们根据各自的数据流模式，计算每种策略所需的动态随机存取存储器访问次数，从而确定哪种策略能够实现最佳的数据复用。

输入驻留（IS）– 是一种专注于重用输入瓦片的调度策略。图 4 展示了 IS 的执行流程。第一步（1 在图4中）是将输入瓦片从动态随机存取存储器加载到 NMP MB0 片上内存中；然后，W 块也被加载 2 从动态随机存取存储器加载到 MB1。为了使

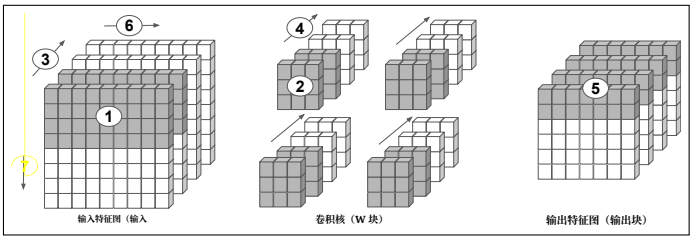


图 4：输入驻留。

为了完整复用输入瓦片，W 块必须包含分配给 TLT的所有滤波器——即使每个滤波器只包含其中很小的一部分。输入瓦片和 W块已加载后，乘累加单元在其上执行卷积。结果存储到 MB2（输出）存储器中，此时其中仅包含卷积的部分和——只有在计算完输入特征图深度方向上的所有元素后，输出块的最终结果才会生成。为此，在沿通道（深度）方向遍历时，可能需要加载多个输入 3 和 W 块 4 可能需要加载。计算并累加结果后，输出块即可存储到动态随机存取存储器5 中。之后，会加载一个新的输入瓦片，先沿输入特征图的宽度6 ，然后沿高度 7 方向移动——对于每个输入瓦片，相同的 W 块都会被反复重新加载。

根据输入驻留从动态随机存取存储器加载数据或向其存储数据时采用的访问模式，可以使用公式 3 确定，对于每个分块（输入、权重和输出块），为完成 TLE/TLT 上整个卷积层的计算，需要加载或存储该分块的次数。符号 α_{in} 和 α_w 表示 TLE/TLT 为计算整个卷积层，需要从动态随机存取存储器加载输入和权重分块的次数。 α_{out} 表示将输出瓦片存储到动态随机存取存储器的次数。

$$\begin{cases} \alpha_{in} = \#TLEs \times \left\lceil \frac{TLE_R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \\ \alpha_w = \#TLEs \times \left\lceil \frac{TLE_R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \\ \alpha_{out} = \left\lceil \frac{R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \end{cases} \tag{3}$$

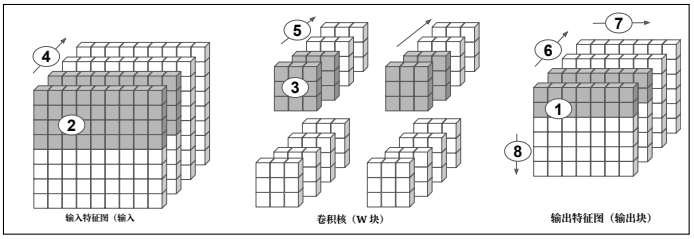


图 5：输出驻留。

输出驻留 (OS) – 是一种映射策略，优先生成输出瓦片，即使相同的输入和权重分块必须从动态随机存取存储器多次加载到各自的片上存储器中也是如此。图 5 展示了 OS 的执行流程。首先，根据输出块 1 的尺寸，将相应的输入 2 和权重 3 分块

从动态随机存取存储器加载到各自的片上存储器中，然后使用 TLT 的乘累加单元对它们进行卷积计算。由于片上存储器通常没有足够的空间容纳所有所需的输入数据，因此必须沿通道（深度）方向加载多个输入 4 和权重 5 分块。完成一个输出块的计算后，将其存储到动态随机存取存储器中，并沿通道（深度）方向开始计算新的输出块 6。之后，先沿输出特征图宽度 7 再沿其高度 8 依次计算其他输出瓦片。每个输入、权重和输出分块需要从动态随机存取存储器加载或存储的次数由公式 4 给出。

$$\begin{cases} \alpha_{in} = \#TLEs \times \left\lceil \frac{TLE_R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \times \left\lceil \frac{TLE_W}{T_M} \right\rceil \\ \alpha_{w} = \#TLEs \times \left\lceil \frac{TLE_R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \times \left\lceil \frac{TLE_W}{T_M} \right\rceil \\ \alpha_{out} = \left\lceil \frac{R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{M}{T_M} \right\rceil \end{cases} \tag{4}$$

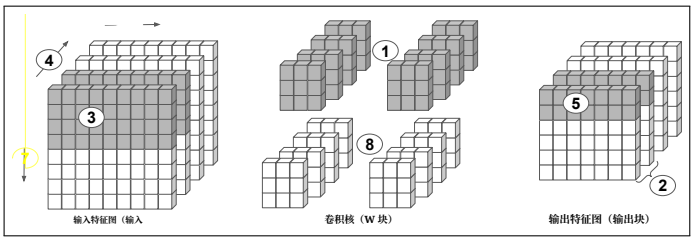


图 6：权重驻留。

权重驻留（WS）——这是一种映射策略，重点在于仅从动态随机存取存储器加载一次滤波器，并在计算卷积分块时重复使用这些滤波器。图 6 展示了 WS 的执行流程。首先，T_M 将 W 块中的滤波器从动态随机存取存储器加载到 MB1 片上内存中 1 。在此策略中，每个已加载的滤波器都包含其全部 N 通道。随后重复使用这些已加载的滤波器，直至计算出相应的输出特征图 2 。执行 nmp conv2d 指令之前，先从动态随机存取存储器加载一个输入瓦片 3。沿通道深度可能需要多次加载输入瓦片 4，每次计算并存储部分结果，之后再将其累加以形成最终的输出块 5，然后将其写回动态随机存取存储器。接着依次沿宽度加载其他输入瓦片 6 ，然后沿高度加载 7 。这一过程持续进行，直至计算出 W 块中相应滤波器的所有输出特征图。之后，可能需要加载包含其他滤波器的新 W 块 8 ，以计算这些滤波器对应的输出特征图——此时，每次迭代都会再次加载相同的输入瓦片。公式 5 给出了覆盖整个卷积层所需的动态随机存取存储器数据传输次数。

$$\begin{cases} \alpha_{in} = \#TLEs \times \left\lceil \frac{TLE_R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \times \left\lceil \frac{TLE_W}{T_M} \right\rceil \\ \alpha_{w} = \#TLEs \times \left\lceil \frac{TLE_W}{T_M} \right\rceil \\ \alpha_{out} = \left\lceil \frac{R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{M}{T_M} \right\rceil \end{cases} \tag{5}$$

NMP架构支持本文所提出方案之外的其他分区策略。该方案利用多播指令，将输入特征图切片并行加载到 8 个 TLT/TLE 中。例如，可以考虑一种将输入特征图（通道）分配给各个 TLE 的方法，使它们计算部分输出特征图和，之后再归约得到最终的输出特征图结果。遗憾的是，在 NMP 中，归约部分和的过程需要在 TLE 之间传递大量环形网络消息，从而影响性能，并导致某些 TLT 在其他 TLT 执行归约树时处于空闲状态。此外，本文提出的分区策略不仅利用多播加载减少数据传输，还能确保 TLE/TLT 之间实现负载均衡。

E. 时间估算

为了确定在上述各节讨论的切片策略中哪一种最佳，对于每个卷积，TSO 会从搜索空间<$PART_{TLE}$、$PART_{TLT}$、T_M、T_N、T_R、T_C>中组合出多个解决方案，并估算每种有效组合所需的时间，从而选出性能最佳的组合。该估算包含以下组成部分，按其对卷积执行时间的贡献由小到大排列：(a) 在每个 TLT 上运行 RISC-V 指令所需的时间；(b) 对切片执行乘累加单元操作所需的时间；以及 (c) 在动态随机存取存储器与神经处理单元片上存储器之间加载/存储数据所需的时间。执行时间由函数 *CalcTime* （定义于算法 3——第 11 行）根据公式 6 对执行的各个组成部分的时间求和后计算得出。

$$T_{CONV} = T_{MAC} + T_{DRAM} + T_{SW} \tag{6}$$

其中，T_{MAC}、T_{DRAM} 和 T_{SW} 分别表示乘累加单元所用的时间、在神经处理单元的片上存储器与动态随机存取存储器之间传输数据所用的时间，以及执行 RISC-V 指令所用的时间。由于 T_{SW} 并不显著（通常不到总执行时间的 5%），本文将不对此进行详细讨论。

时间 T_{MAC} 根据卷积层的乘加运算（MAC operations）数量计算得出。在第一步中（见公式 7），首先确定在输入特征图上计算一个通道所需的乘加运算数量，然后将其除以一个乘累加单元在每个周期内能够执行的乘加运算数量。之后，再考虑其他特征图，以估算整个分块（$Tile_{MAC}$）所需的时间。已知单个分块所需的时间后，便可估算计算整个卷积层工作负载所需的总时间；该工作负载分配到 NMP 中所有 TLT 核心（$\#TLEs \times \#TLTs$）上（见公式 8）。

$$Tile_{MAC} = T_N \times T_M \times \left(\left\lceil \frac{T_R \times T_C \times K \times K}{\#MACs} \right\rceil \right) \times \frac{1}{Freq} \tag{7}$$

算法 4 估算数据传输所需的时间

```
1: 函数 CALCDATATRANSFER(tile, conv, arch)
2:     type ← arch.type
3:     BW ← arch.BW
4:     如果 model = TSO-burst 则
5:         nbursts ← CALBURSTCOUNT(tile, conv, type)
6:         cas_latency ← nbursts ∗ CAS
7:         tile_size ← GETTILESIZE(tile, type)
8:         transfer_time ← tile_size/BW
9:         total_time ← transfer_time + CAS_latency
10:    else
11:        ▷ TSO-noburst
12:        tile_size ← GETTILESIZE(tile, type)
13:        total_time ← tile_size/BW
14:    返回 total time
```

$$T_{MAC} = \frac{1}{\#TLTs} \times \left\lceil \frac{M}{T_M} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \times \left\lceil \frac{R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times Tile_{MAC} \tag{8}$$

为评估片上存储器与 DDR存储器 之间的数据传输，TSO 采用两种方法来估算加载/存储输入、W 和输出分块所需的时间，分别为：（a）TSO-burst，一种基于突发传输的模型，通过确定存储器突发传输的数量以及所需的列地址选通和访问时间总和，来估算分块的 DRAM 传输时间；（b）TSO-noburst，一种基于数据量的模型，仅根据分块数据的大小和存储器带宽来估算 DRAM 传输时间。算法 4 描述了如何使用这两种方法计算 DRAM 传输时间。为便于解释算法 4，假设使用具有 CAS 延迟（微秒）、 BW 带宽（字节/秒）和 $BURST$ 大小（以字节计，例如 128B）的 DDR3 内存

为了估算两种方法（TSO-burst 和 TSO-noburst）中数据传输所需的时间（见公式 9），需要将每个分块所需的内存访问次数（α_{in}、 α_w 和 α_{out}）与对函数 *CalcData-Transfer*（定义见算法 4，详情如下）的调用结合起来。随后，将每个分块的估算时间合并，得到 T_{DRAM}。

$$\begin{aligned} T_{DRAM} = {} & \alpha_{in} * CalcDataTransfer(IN, conv, arch) + \\ & \alpha_w * CalcDataTransfer(W, conv, arch) + \\ & \alpha_{out} * CalcDataTransfer(OUT, conv, arch) \end{aligned} \tag{9}$$

基于突发传输的数据传输（TSO-burst）– TSO-burst 的核心思想是确定每次访问瓦片行所需的突发传输次数，并据此估算瓦片的动态随机存取存储器访问时间。例如，考虑一个包含 T_N（通道）、 $\times T_H$（行）和 $\times T_L$（列）的输入瓦片，其中每个元素为 16 位（2B）。假设通道按行主序排列，加载一行的第一个元素需要 CAS 时间，而加载其余元素$\sim (2 \times (T_L - 1))/BW$。因此，加载一行输入需要 $CAS + \sim (T_L - 1)/BW$。当行大小（$2 \times T_L$）小于 $BURST$ 字节时，情况确实如此。否则，加载该行时可能会产生其他内存突发传输，额外的 CAS惩罚也会影响所需时间。

算法 4 用于估算在架构 $arch$ 上将卷积 $conv$ 划分为分块 $tile$ 时的执行时间。首先，从 $arch$ 数据结构中提取分块数据大小 $type$（例如，16 位定点）（第 2 行）和动态随机存取存储器带宽 BW（第 3 行）。接着（第 4 行），算法选择存储器传输模型（例如，TSO-burst），并调用函数 *CalBurstCount*（第 5 行），以确定加载一个输入瓦片的所有 T_H 行所需的突发传输次数（$nburts$）。随后，将列地址选通延迟的影响计入 $cas\ latency$（第 6 行），并在第 7 行调用函数 *GetTileSize* 来确定分块大小（$tile\ size$）。然后确定传输分块中的所有数据所需的时间（$transfer\ time$）（第 8 行），最后计算加载该分块所需的总时间（第 9 行）并将其返回（第 14 行）。

除了假设存在这些存储器中常见的基于突发的访问外，TSO-burst 不对外部动态随机存取存储器或内存控制器的设计作任何假设。NMP 板上的内存控制器遵循 ARM 总线协议。除 CAS 延迟外，还可以纳入其他动态随机存取存储器参数（例如 Trcd/Trp/Tras），以提高数据传输建模的精度。不过，由于 CAS 延迟是这些动态随机存取存储器参数中最相关的一项，算法 4 仅对此进行了讨论。

基于容量的数据传输（TSO-noburst）–这种方法通常被所有解决该问题的先前研究所采用。如算法 4的第 12-13行所示，该方法仅考虑传输分块数据所需的时间（第 12 行），而不考虑分块的存储器突发传输的列地址选通延迟所带来的影响，从而估算分块时间。

V. NMP XLA 编译器

在本工作中，我们使用了TensorFlowXLA（TF-XLA） [27]，它是用于线性代数的领域专用编译器。TF-XLA 预先（AOT）编译被用于为 NMP架构上的机器学习模型生成可执行二进制文件。

编译流程 – TF-XLA编译器接收一个*protobuf* 文件作为输入，其中包含网络的定义（操作及其连接）和权重。随后，它会对*protobuf*执行一些转换（使用TFGraph TransformTool），（例如，将批归一化折叠到卷积中），这些转换对于后续的量化阶段很有用。应用转换后，会生成一个新的protobuf，随后用于创建 XLA HLO 中间表示（见图 7）。从初始的XLA HLO表示开始，会应用目标无关优化（例如，死代码消除、公共子表达式消除等），从而生成优化后的HLO 表示。之后，会执行目标相关优化（如量化），随后执行其他特定阶段，以便于代码生成（例如，操作融合）。TSO 是应用于 HLO中间表示的最后一个阶段，紧接在 LLVM IR 降级之前。之后，编译器从 HLO 中间表示生成 LLVM IR，该 LLVM IR调用 NMP 内建函数，将计算降级为 TLT RISC-V 加速代码。

量化阶段 – 在此阶段，我们的 TF/XLA 编译器将权重从 32 位浮点数转换为 8 位或 16 位

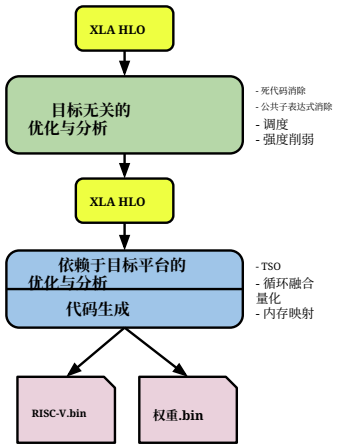


图 7：TensorFlow XLA 流程。

定点。量化的目标是确定输入张量、中间张量和权重的 *Q* 点 。Q 点是一个基数点，用于确定为数据的整数部分和小数部分分别分配多少位。其目标是确保整数部分具有足够的位数，以表示某个操作产生的最小和最大输出值。为了获取模型张量的数据范围，需要执行一些步骤。首先，需要确定对于一组不同的输入张量，卷积神经网络的每一层可能产生的最小值和最大值。为此，使用 TF Graph Transform Tool 中提供的插入日志记录 *method* 功能，在所有模型操作的输入/输出处插入探针，从而生成经过检测的*protobuf*。之后，编译器执行校准步骤，即对经过检测的*protobuf* 执行数百或数千张图像，以捕获每个记录操作输出的最小值和最大值。随后使用这些值确定每个记录操作的 Q 点。由于某些操作在日志记录过程中未被标记（例如 Pool-Layer），因此需要在 TF Graph 上执行两次遍历。在第一次遍历（正向遍历）中，将已计算出的 Q 点从图的输入向输出传播到各个运算符。在第二次遍历（反向遍历）中，从图的输出向输入调整 Q 点——例如，Concatenate 或 Add 操作的输入必须具有相同的 Q 点。然后，编译器将布局从 NHWC（TensorFlow 中的默认布局）修改为 NCHW，这是 NMP架构所要求的布局，随后生成包含量化权重的文件。此外，还会生成一个内存映射，运行时每个 TLT 的二进制代码会使用该内存映射来调度分块，使其能够在动态随机存取存储器和 TLT 片上存储器中加载/存储数据。

优化 HLO中间表示后，我们的 TF-XLA编译器将 HLO中间表示降低为 LLVM IR。在代码生成过程中，HLO指令映射到经过优化的 NMP 库中的内建函数，该库包含大多数典型的 C NN运算。从 LLVM IR 中，编译器生成一个 RISC-V 可执行文件

模型	准确率 (TF-XLA)				TSO (微秒)		TLE 分区 固定 (微秒)			TLT 调度 固定（微秒）		
	CPU 浮点 (%)		NMP (%)									
	Top-1	Top-5	Top-1	Top-5	突发	No 突发	KS	KS& OFM	OFM	IS	OS	WS
InceptionV3	75.1	92	76.9	93.4	72686	88478	82370	81367	91408	73706	93731	101641
LeNet	99.9	100	99.9	100	199	231	202	224	231	199	233	228
MobileNetV2	70.2	89.6	70.5	89.8	14030	15470	17765	16696	17571	14084	17387	15155
ResNet-50	70.6	89.9	70.7	89.9	55927	62375	62077	63118	78844	59714	71905	77535
SqueezeNet	47.1	71	47.1	71	12504	13713	16579	14134	12993	12908	15840	13452
YOLO	-	-	-	-	53271	57225	56591	55713	63550	68530	58592	55375

表 I：模型在 CPU（FP32）和 NMP（16 位定点）上的准确率，以及 TSO（突发和非突发）与固定策略的执行时间。YOLO 使用不同的准确率指标，它衡量检测的精确率；NMP 上为 93.53%，而 CPU 上为 93.03%。

(RISCV.bin)，供 TLT 使用，与量化权重文件(Weight.bin)一起执行模型。

其他人工智能编译器也被用于为人工智能加速器生成代码。例如，Glow [28] 可为 Intel Habana 生成代码，而 onnx-mlir [29] 可为集成在 IBM z16 处理器中的人工智能加速器生成代码。

TSO 的执行需要几分钟。为了便于重新适配，我们计划未来将 TSO 制作为通用的 MLIR通道，以便以与机器无关的方式用于 ONNX 和 XLA等流程。用户将指定存储器层级、处理器/核心数量，MLIR-TSO 通道将输出最佳的分块/调度方案，以便 LLVM 对其进行降级。

VI. 实验结果

为验证 TSO 方法，我们在一块配备 4 个 TLE 的 NMP 板上执行了一组实验，每个 TLE 配有 8 个 TLT。每个 TLT 包含三个 8KB 的 MB 片上存储器（ MB0–MB2 ）。为了评估 TSO，我们使用了 5 个用于图像分类的卷积神经网络模型：InceptionV3 [11], LeNet[10],MobileNetV2 [30], ResNetV50 [2], 和 SqueezeNet[31]。我们还将 TSO 应用于一个目标检测应用——一个基于 YOLO 的模型 [32] ，用于识别汽车牌照。所选模型包含数量不等、形状各异的卷积，其输入（输入特征图）、权重（卷积核）和输出（输出特征图）也各不相同。

所有模型均由我们的 TF-XLA编译器编译，并将量化阶段设为 16 位定点。表 1 展示了各图像分类模型在 NMP 上达到的准确率。对于 LeNet 和其他图像分类模型，Top-1 和 Top-5 准确率分别通过在验证数据集 MNIST [33] 和 ImageNet (ILSVRC2012) [34], 上运行所有图像进行测量。我们还使用相同的数据集测量了 CPU 上原始浮点模型（FP 32 位）的准确率。准确率下降幅度介于 0.1% 至 1.8% 之间。对于基于 YOLO 的模型，NMP 在汽车牌照数据集 [35] 上使用 16 位量化数据运行时，检测精度达到 93.53%；同一模型在 CPU 上的检测精度为 93.03%。

本文采用的量化方案仅将卷积数据量化为 16 位定点。但对于更低精度，例如 8 位定点，TSO 可能会选择与处理同一卷积并将其量化为 16 位定点时不同的方案。例如，

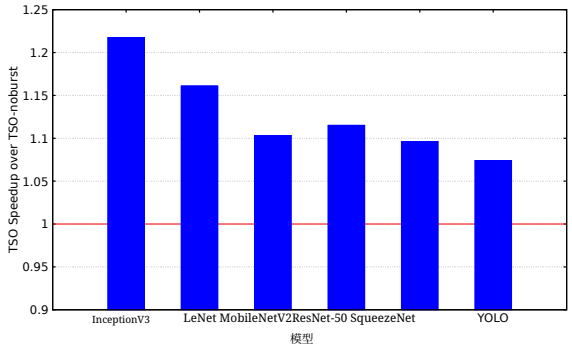


图8：TSO-burst 相较于 TSO-noburst 的加速比。

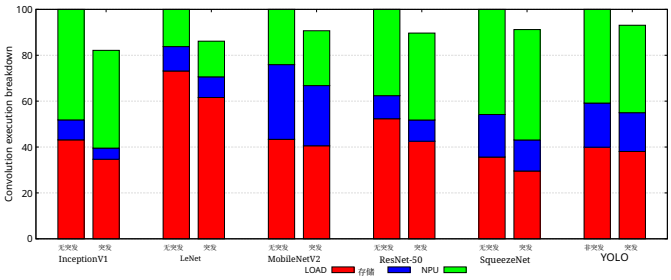


图 9：以TSO-burst 相对于 TSO-noburst 的比例表示的卷积执行分解。

假设只有在将给定卷积量化为 8 位后，其中一个卷积组成部分（例如输入特征图）才变得比对应的 MBLOB 更小。将该组成部分保持驻留，只需加载一次该组成部分及其他组成部分的分块（卷积核和输出特征图），即可完成卷积。因此，这会显著减少动态随机存取存储器与 NMP 之间的数据传输。因此，TSO 倾向于选择此方案，因为与其他方案相比，它能减少数据传输。

TSO-burst 与 TSO-noburst

我们比较了启用基于突发传输的建模的 TSO（TSO-burst）和未启用该建模的 TSO（TSO-noburst），后者是大多数先前研究中常见的方法。如图 8 所示，与 TSO-noburst 相比，TSO-burst 的加速比从 YOLO 的 7.4% 到 InceptionV3 的 21.7% 不等。采用 TSO-burst 带来的主要改进出现在输入特征图被划分为输入瓦片时。这是因为 TSO 倾向于沿宽度（行主序）方向选择较大的分块。通过选择较大的分块，TSO 可最大限度地减少所需突发传输的次数，从而降低列地址选通延迟对内存访问时间的影响。通过优先沿宽度方向进行突发传输，TSO 可提高突发传输的利用率，因为这有助于改善内存访问合并。对于 TSO-noburst，选择分块的目的是减少从动态随机存取存储器加载到 NMP 的字节数。迄今为止，文献中提出的大多数解决方案都采用了这种方法 [36]–[39]。与这些方法不同，本文提出的 TSO-burst 技术在估算从存储器加载/存储数据所需的时间时，会考虑动态随机存取存储器访问合并，从而实现更优的分区并提升性能。各模型的执行时间如表 I 所示。请注意，TSO 始终能获得最短的执行时间。

例如，考虑 InceptionV3 的第5 个卷积层，该层具有 80 个大小均为 73x73 的输入特征图，以及192 个大小为 80x3x3的滤波器。TSO-burst 选取的输入瓦片形状大小为14x4x73 ($T_N \times T_H \times T_L$)。由于 TSO-burst 的输入瓦片涵盖了通道的整个宽度 L (73)，因此会在动态随机存取存储器上连续对齐为一系列 584字节的数据($4 \times 73 \times 2B$)，这要求每个分块的通道进行 5 次存储器突发传输。总的来说，考虑该分块的 14 个通道时，第 5 个卷积层总共需要 70 次存储器突发传输。另一方面，TSO-noburst 选择了大小为 16x11x20 的分块，该分块在动态随机存取存储器上仅对齐 20 字节，因此每行需要一次存储器突发传输。鉴于输入分块包含 16 个通道，每个通道有 11 行，因此总共需要 176 次存储器突发传输，是加载 TSO-burst 分块所需次数的两倍多。对于这个特定的第 5 个卷积层，TSO-burst 将分块执行时间缩短了 28%。

TSO-burst 提供的解决方案旨在减少数据传输操作所需的总时间。为说明这一点，请参阅图9，该图针对每个模型显示了两个柱状图，分别表示相对于总执行时间，花费在 LOAD、STORE和乘加运算上的计算时间百分比细分。对于 TSO-burst 的柱状图，执行时间百分比是相对于 TSO-noburst 的总时间计算的。如图 9 所示，当使用 TSO-noburst 时，LOAD+STORE传输时间的百分比范围为 51.83%（InceptionV3）至 83.80%（LeNet）。另一方面，当在 TF-XLA 编译期间使用 TSO-burst对模型的内存访问进行建模时，YOLO和InceptionV3 的LOAD+STORE 操作所需时间分别从 7.08% 降至 23.71%。

加速 *TSO* 解空间探索

本实验旨在评估算法 1（第 2、3 和 5 行）中的OpenMP任务并行注释对TSO切片空间探索总体时间的影响。我们在一台配备 16个物理核心和 64GB存储器的英特尔 XeonE5-2620上进行了该实验。结果如图10所示，其中每条线对应一个模型，纵轴表示相对于顺序执行的加速比，横轴表示OpenMP 使用的线程数。值得注意的是，与串行执行相比，对于大多数模型，多线程执行几乎呈线性提升。对于具有 94 个卷积的 InceptionV3，多线程执行几乎呈线性提升。另一方面，对于仅具有2 个卷积的 LeNet，2 个线程就足以加速执行，因此当线程数从这一点继续增加时，会出现性能下降。就时间而言，TSO 的串行执行时间从LeNet网络的 28 毫秒到InceptionV3 网络的6 分钟不等，而采用多线程执行后，该时间分别缩短至 17 毫秒和 59 秒。总时间

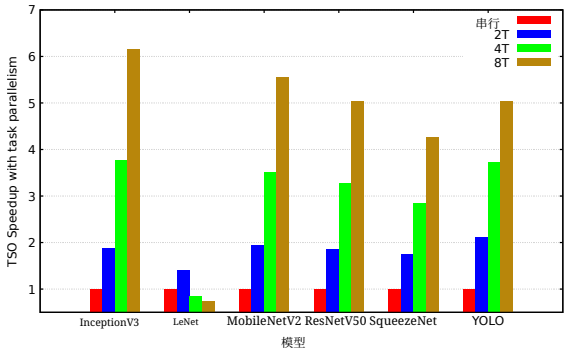


图 10：评估 OpenMP 并行化对 TSO 解空间探索的效果。

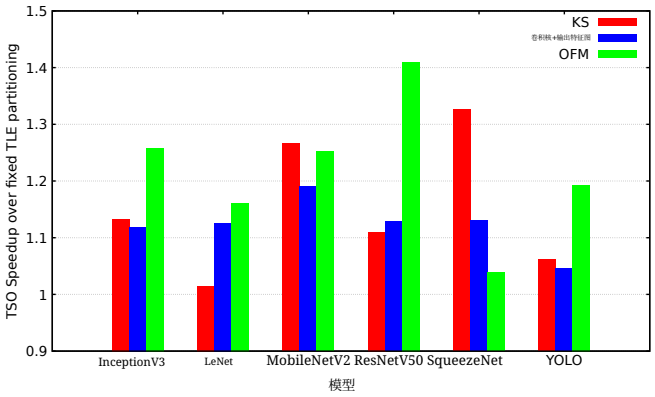


图 12：固定 TLE 切片。

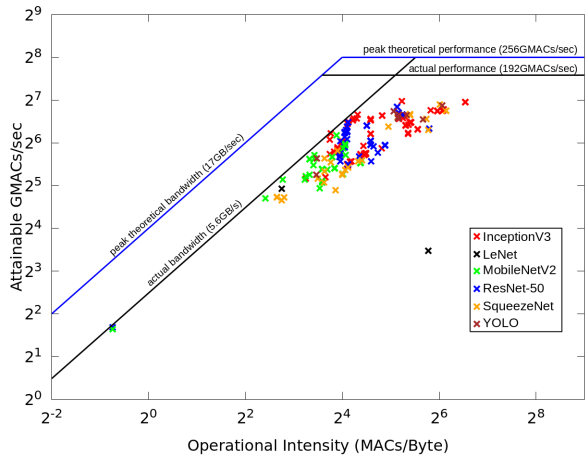


图 11：NMP架构的 Roofline模型（TSO-burst）。

编译器从开始到生成二进制文件所耗费的时间，对于相同的模型而言，分别从20秒到7分钟不等，而其中大部分时间都消耗在校准/量化步骤上。

*Roofline*模型

为了评估在 NMP 上执行的各模型的卷积层采用 TSO 后所得代码的性能（TSO-burst），我们使用了图 11 所示的 Roofline模型。图中，y 轴表示架构和卷积执行所达到的乘加运算（MAC）吞吐量（单位：GMAC/s）。x 轴表示运算强度，即每从动态随机存取存储器加载一个字节所执行的乘加运算次数。图中的蓝线表示 MAC 吞吐量（水平线）和动态随机存取存储器带宽（斜线）的理论上限，分别对应 NMP 引擎和存储器系统所能达到的数值。为了更好地评估系统的实际性能，我们另外进行了两项实验来测量这些参数。之所以需要这样做，是因为架构中的其他组件也会影响其数值。如图所示，测得的 MAC 吞吐量……

吞吐量达到 192 GMACs/sec 的上限，以水平黑线表示。造成这一降低的原因可能有很多。例如，本工作中使用的 NMP 设备采用单端口片上存储器，因此，等待数据的 TLT 会处于空闲状态，无法使用其乘累加单元。至于存储器带宽（斜黑线），测量值也有所降低。这可以解释为：动态随机存取存储器带宽受到每个TLE 配备的单个 DMA 引擎的限制，而该引擎必须同时为所有 8个TLT 核心提供服务。为了评估TSO 所带来的性能，我们在图11 中为模型中的所有卷积绘制了一个点。如图所示，大多数卷积要么达到由（测量的）存储器带宽（斜黑线）所限制的上限，要么接近由乘累加吞吐量（水平黑线）定义的上限。这清楚地表明，TSO 生成的代码能够接近该架构的最大性能。

固定 *TLE/TLT* 划分

固定 TLE 划分 – 在此实验中，编译器被设置为针对模型的所有卷积层生成代码，并固定每个 TLE 切片策略，这些策略如第 IV-B 小节所述。实验流程如下：对于固定的 TLE 切片，编译器会找出最佳的 TLT 分块/调度策略（IS、OS 和 WS）。此实验的结果如图 12 所示，图中报告了采用 TSO（突发模式）编译的模型相较于采用固定 TLE 切片编译的模型的加速比。对于 卷积核 情形，TSO 对 SqueezeNet 的加速比最高可达 32.6%。SqueezeNet 的 TLE 切片效果不佳，因为其大多数卷积层的输入特征图都大于滤波器组（卷积核）的尺寸。对于卷积核&输出特征图，TSO 对MobileNetV2的加速比最高可达 19%。这种 TLE 策略通常更适用于输入特征图与权重（卷积核）尺寸相近的卷积层。对于 输出特征图，TSO 对 ResNet-50 的加速比为 41.0%。对于 ResNet-50 的大多数卷积层，权重集合卷积核的尺寸大于输入特征图数据图的尺寸。因此，TSO 的表现优于最佳固定 TLE 切片策略。

固定 TLT 划分——在此实验中，编译器被设置为生成代码，为所有模型的卷积层固定三种 TLT

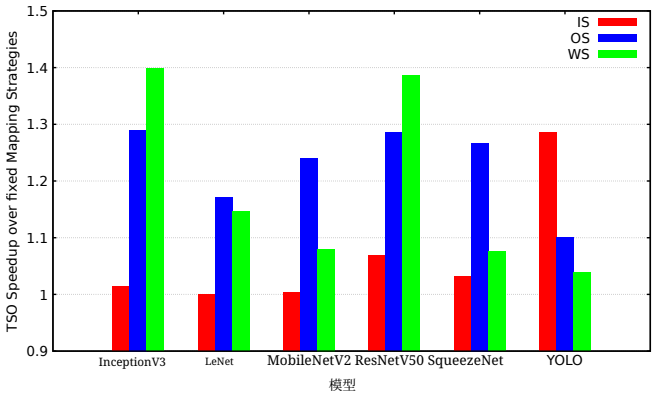


图 13：固定 TLT 调度策略。

模型	架构	Top-1	Top-5
MNIST 1.3	CPU（浮点）	98.9%	100%
	NMP	99.2%	99.8%
LeNet	CPU（浮点）	94.8%	99.9%
	NMP	95.2%	99.9%
ResNet18 1.2.1	CPU（浮点）	69.9%	89.3%
	NMP	66.4%	87.3%
SqueezeNet 48.7	CPU（浮点）	49.0%	72.9%
	NMP	47.1%	71.0%
Mobilenet 2.1	CPU（浮点）	71.8%	90.6%
	NMP	70.9%	89.4%

表 II：Glow 在 CPU（FP32）和 NMP（16 位定点）上的模型准确率。

调度策略（IS、OS 和 WS）之一。编译器将固定的调度策略应用于所有可能的 TLE 切片选项（卷积核、卷积核与输出特征图，以及输出特征图），以搜索最佳性能。图 13 展示了 TSO 相较于最佳固定 TLT 策略的加速比。在 TF-XLA 代码生成期间固定采用 IS，TSO 在 YOLO 上的加速比可达 28.6%。对于 YOLO 网络，IS 的表现不佳，因为该网络有多个卷积层，其输入特征图从 102×102 ($H \times L$) 变化到416×416——这导致滤波器需要多次加载，因为需要多个输入瓦片。TSO 相较于固定 OS 的加速比在 ResNet-50 上最高可达 28.6%。与固定 WS 相比，TSO 在 InceptionV3 上的加速比为 39.8%。对于 InceptionV3，使用 WS 时，需要多个权重瓦片来处理这些切片，因此每个权重瓦片都需要多次加载输入特征图，从而增加数据传输量。再次证明，TSO 优于最佳固定 TLT 调度策略。

在 *Glow* 框架上的评估

我们还进行了一系列实验，以评估 TSO 移植到其他机器学习框架的可行性。此次评估选用了 Facebook 的 Glow 工具链，并将其性能与 TensorFlow/XLA 编译器生成的精度进行了比较。由于 Glow 使用的模型与 ONNX 模型库中的模型略有不同，因此未能对其性能与 TensorFlow/XLA 进行全面比较。

部分模型需要从 Google TFLite Hub 转换为 ONNX 格式（例如 Lenet、Squeezenet 和 Mobilenet），而其他模型则直接使用 ONNX Model Zoo 中的模型（例如 MNIST 和 Resnet18 ）。准确率与相应代码库中列出的 CPU 32 位浮点精度结果进行了比较。为此，根据模型的不同，在 ImageNet 和 MNIST 验证数据集上使用 NMP 架构执行了各模型的推理，并测量了 Top-1 和 Top-5 准确率。如表 II 所示，Glow 在 NMP 上生成的代码所达到的准确率接近 TensorFlow/XLA 编译器的结果（参见表 I）。

VII. 相关工作

Maestro [40] 和 Timeloop [41] 采用解析建模，评估不同的映射配置——数据流策略、数据复用、分块大小等，以估算不同配置的运行时间。Maestro 设计了一些注释，将循环归类为时间循环或空间循环；Timeloop 则分析嵌套循环，并对其应用变换。对于这两种方法，给定一个 DNN 层（例如卷积及其相关信息）、硬件配置（PE 数量、片上存储器大小等）和数据流策略，它们都可以估算运行时间、能耗和功耗。由于 Maestro 易于使用，许多方案采用其注释来估算计算量 [42]，[43]。例如，Marvel [42] 采用 Maestro 的表示法，其主要目标是通过将片上/片外子空间访问成本模型的分析解耦来缩小搜索空间。Timeloop 和 Maestro 对空间型 DNN 加速器进行建模，也就是基于 FPGA 的架构；在这些架构中，卷积的内层循环会展开，并综合为 PE 阵列（乘累加单元），这些单元以同步方式运行，以利用它们之间通过 PE间通信实现的数据共享。与 Marvel 和 Timeloop 类似，我们的工作也进行成本建模和设计空间探索，但不同之处在于，我们对多核NPU架构上的执行进行建模，而非 FPGA 设计。

Tu 等。 [37] 和 Hu等。 [36] 提出了一种基于FPGA的加速器，能够重新配置其资源以提高数据复用率。他们采用了输入驻留（IS）、权重驻留（WS）和输出驻留（OS）的概念。此外，他们提出了一种名为混合驻留（HS）的新方法，利用这些概念为每个卷积层寻找最优配置。尽管他们的工作与我们的工作存在一定相似之处，但我们并不是将操作映射到PE 阵列上，而是考虑一种具有多个核心的架构（NMP），其中每个核心都有一个独立于其他核心运行的加速器。此外，我们的搜索空间探索算法会根据内存突发传输考虑不同的分块形状，而不仅仅是考虑适配硬件拓扑结构的方形形状。

要选择不同的分块大小、循环顺序、展开因子等，TVM [44] 使用机器学习成本模型，该模型不需要硬件信息，并会定期从先前的预测中学习，以搜索更优的划分方式。

据我们所知，并根据现有的公开文献，TVM 尚未展示针对 NMP 等多核NPU 的任何结果，而仅为 FPGA、嵌入式 CPU 和服务器CPU 生成代码。

为提高数据复用率，一些工作采用了多面体优化技术 [19], [38]。Ma 等。 [45] 描述了一种实现输出驻留（OS）的性能模型。Chen 等。 [20] 描述了一种称为行驻留（RS）的方法，该方法通过利用PE间通信实现数据复用，从而最大限度地减少数据移动。这些工作中的分块选择通常基于基于屋顶线的模型 [39], [46]。与我们的工作相比，它们的Roofline模型仅考虑内存访问次数，而未将内存突发纳入考量。另一方面，Stoutchinin 等。 [47], 采用了一种称为复用距离的技术，旨在确定容纳卷积数据所需的内存占用，该占用量最高可达512KB。他的工作仅考虑片上存储器中的数据复用，而未考虑DRAM访问。

Caffeine [48]是一个能够将全连接层（FCL）转换为卷积层的库。转换过程会调整数据布局，以减少对DRAM的访问次数，从而增加突发长度。Qiu 等。 [49] 同样修改了数据布局，并应用量化以改善内存访问。Putra 等。 [50] 对DRAM中的数据进行映射，以减少行缓冲区冲突。我们的工作采用了类似的思路来增加突发长度，但我们不会重新排列数据布局。 [48]中提出的布局修改过程会在写入层的输出时引入一定的复杂性，因为层的输出数据必须再次重新排列，才能适应下一层的输入配置（例如分块大小）。

Alwani 等人 [51] 和 Xiao 等人 [52]的工作着重研究多个卷积层之间的数据流。通常情况下，处理完一层后再处理下一层，而他们关注的是同时处理多层，且不在层间生成中间数据。他们通过融合多个层，使这些层形成一个计算金字塔。为保存每个金字塔的中间数据，他们使用了一些复杂的数据结构。总体而言，尽管像他们那样减少了FPGA 与主机之间的存储器传输，但其加速器仍需要大量存储器来存放不同金字塔的所有中间数据。另一方面，King 等人[53],提出了一种算法，用于评估从分支起点到合并处的多个卷积层的调度，从而尽可能将每层的输入/输出数据保留在片上内存中。在 [53],他们使用了一种配备 1MB 片上内存的神经处理单元，这对许多情况来说已经足够，但对于边缘推理AI加速器而言，成本略高。

还可以通过应用数据压缩来减少数据传输。NullHop [54] 通过硬件实现这一点，而 Han等人 [55] 则通过应用哈夫曼编码来实现。稀疏性是另一种用于避免计算零元素的技术，因此，除了避免不必要的计算外，还能减少不必要数据的数据传输。若干研究采用了这种技术 [55]–[57]。这些技术也都可以用于改进本文提出的方法，不过它们并非本文的重点。

VIII. 结论

鉴于神经处理单元架构的片上内存容量有限，高效的数据分块与调度技术对于最大限度地降低内存访问开销至关重要。本文提出了 TSO，这是一种用于 TF-XLA编译器的优化流程，可确定数据分块、调度和乘加运算的最佳组合，从而最大限度地减少 CNN模型中的卷积执行开销。为此，TSO 对内存突发进行了精确建模；与不采用突发建模相比，在一些典型 CNN模型上实现了最高 21.7% 的加速。与固定的 TLE 切片相比，TSO 的加速幅度最高可达 41.0%；与固定的 TLT 分块相比，最高可达 39.8%。此外，还通过将 TSO 移植到 Glow 工具链并运行，对其通用性进行了评估。

致谢

本研究获得了韩国政府（MSIT）资助的信息与通信技术规划评估院（IITP）项目（编号：2022年-0-00769，人工智能系统神经形态计算软件平台）的支持，同时也获得了电子与电信研究院（ETRI）以及 SilicoNeuro/Ai M Fu-ture 的支持。

参考文献

[1] K. Simonyan 和 A. Zisserman，《用于大规模图像识别的超深卷积网络》，*arXiv*预印本 *arXiv:1409.1556*，2014年。[2] K. He、X. Zhang、S. Ren 和 J. Sun，《用于图像识别的深度残差学习》，载于*IEEE*计算机视觉与模式识别会议论文集，2016年，第770–778页。[3] B. Zhou、A. Lapedriza、J. Xiao、A. Torralba 和 A. Oliva，《使用Places数据库学习用于场景识别的深度特征》，载于神经信息处理系统进展，2014年，第487–495页。[4] R. Girshick、J. Donahue、T. Darrell 和 J. Malik，《用于精确目标检测与语义分割的丰富特征层次结构》，载于*IEEE*计算机视觉与模式识别会议论文集，2014年，第580–587页。[5] P. Sermanet、D. Eigen、X. Zhang、M. Mathieu、R. Fergus 和 Y. Le- Cun，《Overfeat：使用卷积网络实现集成识别、定位与检测》，*arXiv*预印本 *arXiv:1312.6229*，2013年。[6] A. Karpathy、G. Toderici、S. Shetty、T. Leung、R. Sukthankar 和 L. Fei-Fei，《使用卷积神经网络进行大规模视频分类》，载于*IEEE*计算机视觉与模式识别会议论文集，2014年，第1725–1732页。[7] P. Sermanet、K. Kavukcuoglu、S. Chintala 和 Y. LeCun，《使用无监督多阶段特征学习进行行人检测》，载于*IEEE*计算机视觉与模式识别会议论文集，2013年，第3626–3633页。[8] O. Abdel-Hamid、A.-r. Mohamed、H. Jiang、L. Deng、G. Penn 和 D. Yu，《用于语音识别的卷积神经网络》，*IEEE/ACM*音频、语音与语言处理汇刊，第22卷，第10期，第1533–1545页，2014年。[9] Y. Kim，《用于句子分类的卷积神经网络》，*arXiv*预印本 *arXiv:1408.5882*，2014年。[10] Y. LeCun、L. Bottou、Y. Bengio、P. Haffner 等，《将基于梯度的学习应用于文档识别》，*IEEE*会刊，第86卷，第11期，第2278–2324页，1998年。

[11] C. Szegedy、V. Vanhoucke、S. Ioffe、J. Shlens 和 Z. Wojna，《重新思考用于计算机视觉的Inception架构》，载于IEEE计算机视觉与模式识别会议论文集，2016年，第2818–2826页。[12] F. Tu、W. Wu、S. Yin、L. Liu 和 S. Wei，《Rana：通过刷新优化型嵌入式DRAM实现高效神经网络加速》，载于第45届年度国际计算机体系结构研讨会论文集。IEEE出版社，2018年，第340–352页。[13] J. Cong 和 B. Xiao，《最大限度减少卷积神经网络中的计算》，载于国际人工神经网络会议。施普林格，2014年，第281–290页。[14] V. Vanhoucke、A. Senior 和 M. Z. Mao，《提升神经网络在CPU上的运行速度》，2011年。[15] S.-J. Lee、S.-S. Park 和 K.-S. Chung，《用于加速卷积神经网络的高效单指令多数据实现》，载于第4届国际通信与信息处理会议论文集，2018年，第174–179页。[16] S. Chetlur、C. Woolley、P. Vandermersch、J. Cohen、J. Tran、B. Catanzaro 和 E. Shelhamer，《cuDNN：用于深度学习的高效基本运算》，arXiv预印本 arXiv:1410.0759，2014年。[17] M. Song、Y. Hu、Y. Xu、C. Li、H. Chen、J. Yuan 和 T. Li，《弥合横向扩展卷积神经网络大数据处理中图形处理器加速的语义鸿沟：着眼宏观，洞察细节》，载于2016年国际并行体系结构与编译会议论文集。ACM，2016年，第315–326页。[18] M. Sankaradas、V. Jakkula、S. Cadambi、S. Chakradhar、I. Durdanovic、E. Cosatto 和 H. P. Graf，《用于卷积神经网络的大规模并行协处理器》，载于2009年第20届IEEE应用专用系统、体系结构与处理器国际会议。IEEE，2009年，第53–60页。[19] M. Peemen、A. A. Setio、B. Mesman 和 H. Corporaal，《面向存储器的卷积神经网络加速器设计》，载于2013年IEEE第31届国际计算机设计会议（ICCD）。IEEE，2013年，第13–19页。[20] Y.-H. Chen、J. Emer 和 V. Sze，《Eyeriss：用于卷积神经网络能效数据流的空间架构》，载于ACM SIGARCH计算机体系结构新闻，第44卷。IEEE出版社，2016年，第367–379页。[21] S. Chakradhar、M. Sankaradas、V. Jakkula 和 S. Cadambi，《用于卷积神经网络的动态可配置协处理器》，载于ACM SIGARCH计算机体系结构新闻，第38卷。ACM，2010年，第247–257页。[22] V. Gokhale、A. Zaidy、A. X. M. Chang 和 E. Culurciello，《Snowflake：面向深度卷积神经网络的模型无关加速器》，arXiv预印本 arXiv:1708.02579，2017年。[23] L. Cavigelli、D. Gschwend、C. Mayer、S. Willi、B. Muheim 和 L. Benini，《Origami：卷积网络加速器》，载于第25届大湖区超大规模集成电路研讨会论文集。ACM，2015年，第199–204页。[24] X. Liu、M. Mao、B. Liu、H. Li、Y. Chen、B. Li、Y. Wang、H. Jiang、M. Barnell、Q. Wu 等，《Reno：高效可重构神经形态计算加速器设计》，载于第52届年度设计自动化会议论文集，2015年，第1–6页。[25] D. Kim、J. Kung、S. Chai、S. Yalamanchili 和 S. Mukhopadhyay，《Neurocube：一种配备高密度三维存储器的可编程数字神经形态架构》，ACM SIGARCH计算机体系结构新闻，第44卷，第3期，第380–392页，2016年。[26] N. P. Jouppi、C. Young、N. Patil、D. Patterson、G. Agrawal、R. Bajwa、S. Bates、S. Bhatia、N. Boden、A. Borchers 等，《数据中心内张量处理单元的性能分析》，载于2017年ACM/IEEE第44届年度国际计算机体系结构研讨会（ISCA）。IEEE，2017年，第1–12页。[27]《XLA：面向机器学习的优化编译器》，https://www.tensorflow.org/xla，2022年，访问日期：2022年2月14日。[28] T. Jin、G.-T. Bercea、T. D. Le、T. Chen、G. Su、H. Imai、Y. Negishi、A. Leu、K. O'Brien、K. Kawachiya 等，《使用MLIR编译ONNX神经网络模型》，arXiv预印本 arXiv:2008.08272，2020年。[29] N. Rotem、J. Fix、S. Abdulrasool、G. Catron、S. Deng、R. Dzhabarov、N. Gibson、J. Hegeman、M. Lele、R. Levenstein 等，《Glow：用于神经网络的图降低编译器技术》，arXiv预印本 arXiv:1805.00907，2018年。[30] M. Sandler、A. Howard、M. Zhu、A. Zhmoginov 和 L.-C. Chen，《MobileNetV2：倒置残差与线性瓶颈》，载于论文集

IEEE计算机视觉与模式识别会议论文集，2018年，第4510–4520页。[31] F. N. Iandola、S. Han、M. W. Moskewicz、K. Ashraf、W. J. Dally 和 K. Keutzer，"SqueezeNet：以少50倍的参数量实现与 AlexNet 相当的准确率，且模型大小小于 0 07360，2016 5 1602 . mb"，arXiv预印本，arXiv: . .[32] S. M. Silva 和 C. R. Jung，"使用深度卷积神经网络实时检测与识别巴西车牌"，载于2017年第30届 SIBGRAPI图形、模式与图像会议（SIBGRAPI）。IEEE，2017年，第55–62页。[33] "MNIST数据库"，http://yann.lecun.com/exdb/mnist/，2022年，访问日期：2022年2月14日。[34] "ImageNet"，http://www.image-net.org/challenges/LSVRC/2012/，2022年，访问日期：2022年2月14日。[35] G. R. Gonc.alves、S. P. G. da Silva、D. Menotti 和 W. R. Schwartz，"车牌字符分割基准"，《电子成像期刊》，第25卷，第5期，页码053034，2016年。[36] X. Hu、Y. Zeng、Z. Li、X. Zheng、S. Cai 和 X. Xiong，"一种资源高效的可配置深度卷积神经网络加速器"，IEEE Access，2019年。[37] F. Tu、S. Yin、P. Ouyang、S. Tang、L. Liu 和 S. Wei，"具有可重构计算模式的深度卷积神经网络架构"，IEEE超大规模集成电路（VLSI）系统汇刊，第25卷，第8期，第2220–2233页，2017年。[38] C. Zhang、P. Li、G. Sun、Y. Guan、B. Xiao 和 J. Cong，"优化基于FPGA的深度卷积神经网络加速器设计"，载于2015年 ACM/SIGDA现场可编程门阵列国际研讨会论文集。ACM，2015年，第161–170页。[39] M. Motamedi、P. Gysel、V. Akella 和 S. Ghiasi，"基于FPGA的深度卷积神经网络设计空间探索"，载于2016年第21届亚洲及南太平洋设计自动化会议（ASP-DAC）。IEEE，2016年，第575–580页。[40] H. Kwon、P. Chatarasi、V. Sarkar、T. Krishna、M. Pellauer 和 A. Parashar，"Maestro：一种以数据为中心的方法，用于理解DNN映射中的复用、性能和硬件成本"，IEEE Micro，第40卷，第3期，第20–29页，2020年。[41] A. Parashar、P. Raina、Y. S. Shao、Y.-H. Chen、V. A. Ying、A. Mukkara、R. Venkatesan、B. Khailany、S. W. Keckler 和 J. Emer，"Timeloop：一种系统化的DNN加速器评估方法"，载于2019年IEEE系统与软件性能分析国际研讨会（ISPASS）。IEEE，2019年，第304–315页。[42] P. Chatarasi、H. Kwon、N. Raina、S. Malik、V. Haridas、A. Parashar、M. Pellauer、T. Krishna 和 V. Sarkar，"Marvel：一种面向空间加速器上DNN算子的以数据为中心的编译器"，arXiv预印本，arXiv:2002.07752，2020年。[43] S.-C. Kao 和 T. Krishna，"Gamma：通过遗传算法自动完成加速器上DNN模型的硬件映射"，载于2020年IEEE/ACM计算机辅助设计国际会议（ICCAD）。IEEE，2020年，第1–9页。[44] T. Chen、T. Moreau、Z. Jiang、L. Zheng、E. Yan、H. Shen、M. Cowan、L. Wang、Y. Hu、L. Ceze等，"{TVM}：一种用于深度学习的端到端自动优化编译器"，载于第13届 {USENIX} 操作系统设计与实现研讨会（{OSDI} *18*），2018年，第578–594页。[45] Y. Ma、Y. Cao、S. Vrudhula 和 J.-S. Seo，"面向FPGA上卷积神经网络推理加速器的性能建模"，IEEE集成电路与系统计算机辅助设计汇刊，第39卷，第4期，第843–856页，2019年。[46] C. Park、S. Park 和 C. S. Park，"基于屋顶线模型的卷积神经网络加速器数据流技术设计空间探索"，IEEE Access，第8卷，第172 509–172 523页，2020年。[47] A. Stoutchinin、F. Conti 和 L. Benini，"为实现高效内存访问而对卷积神经网络卷积进行最优调度"，arXiv预印本，arXiv:1902.01492，2019年。[48] C. Zhang、G. Sun、Z. Fang、P. Zhou、P. Pan 和 J. Cong，"Caffeine：面向深度卷积神经网络的统一表示与加速"，IEEE集成电路与系统计算机辅助设计汇刊，2018年。[49] J. Qiu、J. Wang、S. Yao、K. Guo、B. Li、E. Zhou、J. Yu、T. Tang、N. Xu、S. Song等，"借助嵌入式FPGA平台深入探索卷积神经网络"，载于2016年 ACM/SIGDA现场可编程门阵列国际研讨会论文集。ACM，2016年，第26–35页。

[50] R. V. W. Putra、M. A. Hanif 和 M. Shafique，“Drmap：一种用于卷积神经网络高能效处理的通用动态随机存取存储器数据映射策略”，*arXiv*预印本 *arXiv:2004.10341*，2020年。[51] M. Alwani、H. Chen、M. Ferdman 和 P. Milder，“融合层卷积神经网络加速器”，载于第*49*届 *IEEE/ACM* 微架构国际研讨会。IEEE出版社，2016年，第22页。[52] Q. Xiao、Y. Liang、L. Lu、S. Yan 和 Y.-W. Tai，“探索用于在 FPGA 上加速深度卷积神经网络的异构算法”，载于*2017*年第*54*届 *ACM/EDAC/IEEE* 设计自动化大会（*DAC*）。IEEE，2017年，第1–6页。[53] D. Kim、K.-Y. Kim、S. Ko 和 S. Ha，“一种减少卷积神经网络片外内存访问的简单方法”，*arXiv*预印本 *arXiv:1901.09614*，2019年。[54] A. Aimar、H. Mostafa、E. Calabrese、A. Rios-Navarro、R. Tapiador-Morales、I.-A. Lungu、M. B. Milde、F. Corradi、A. Linares-Barranco、S.-C. Liu 等，“Nullhop：一种基于特征图稀疏表示的灵活卷积神经网络加速器”，*IEEE* 神经网络与学习系统汇刊，第30卷，第3期，第644–656页，2018年。[55] S. Han、H. Mao 和 W. J. Dally，“深度压缩：通过剪枝、训练量化和霍夫曼编码压缩深度神经网络”，*arXiv*预印本 *arXiv:1510.00149*，2015年。[56] J. Li、S. Jiang、S. Gong、J. Wu、J. Yan、G. Yan 和 X. Li，“Squeezeflow：一种利用简洁卷积规则的稀疏卷积神经网络加速器”，*IEEE* 计算机汇刊，第68卷，第11期，第1663–1677页，2019年。[57] S. Han、X. Liu、H. Mao、J. Pu、A. Pedram、M. A. Horowitz 和 W. J. Dally，“EIE：压缩深度神经网络上的高效推理引擎”，载于*2016*年 *ACM/IEEE* 第*43*届年度计算机体系结构国际研讨会（*ISCA*）。IEEE，2016年，第243–254页。