ImaGen：一种通用的高内存和功耗效率的图像处理加速器框架

Nisarg Ujjainkar
nujjaink
@ur.rochester.edu 罗切斯特
大学 美国纽约州罗切斯特

Jingwen Leng 上海交
通大学 中国上海
leng-jw@sjtu.edu.cn

Yuhao Zhu 罗切斯
特大学 美国纽约州罗切斯
特
yzhu@rochester.edu

arXiv:2304.03352v1 [cs.AR] 6 Apr 2023

摘要

图像处理算法通常用于资源和功耗受限的应用，因此是硬件加速的首要目标。如今的图像处理加速器设计会对算法结构和/或片上存储资源作出僵化的假设。因此，这些设计要么适用范围狭窄，要么效率低下。

本文提出了一种编译器框架，可自动生成高效利用存储器和功耗的图像处理加速器。我们允许程序员描述通用的图像处理算法（使用领域特定语言），并指定可用的片上存储结构。随后，我们的框架构建一个约束优化问题，在保持理论最大吞吐量的同时，最小化片上存储使用量。我们所解决的关键挑战是以分析方式表达吞吐量瓶颈——片上存储器争用，从而实现轻量级编译。FPGA原型验证和ASIC综合表明，与现有方法相比，由我们的框架生成的加速器可将片上存储使用量和/或功耗降低两位数。ImaGen代码可在以下地址获取：
https://github.com/horizon-research/imagen。

CCS 概念

• **计算机系统组织** → **体系结构**； • **硬件** → **功率与能量**。

关键词

加速器，行缓冲区，图像处理，约束优化，综合，编译器

ACM参考格式：
Nisarg Ujjainkar、Jingwen Leng 和 Yuhao Zhu。2023年。ImaGen：一种用于生成高内存和功耗效率的图像处理加速器的通用框架。载于第*50*届年度国际计算机体系结构研讨会论文集（*ISCA '23*），*2023*年*6*月*17*日至*21*日，美国佛罗里达州奥兰多。ACM，纽约州纽约，13 页。 https://doi.org/10.1145/3579371.3589076

特此授予许可，允许为个人或课堂用途免费制作本作品全部或部分内容的数字或纸质副本，但不得出于营利或商业目的制作或分发副本，且副本首页必须包含本声明和完整引文。对于本作品中由作者以外的其他方拥有版权的组件，必须尊重其版权。注明出处后可以摘录。若要以其他方式复制、再版、发布到服务器或分发给列表中的用户，必须事先获得具体许可，和/或支付费用。请通过 permissions@acm.org 申请许可。*ISCA '23*，*2023*年*6*月*17*日至*21*日，美国佛罗里达州奥兰多©2023年版权所有者/作者保留版权。出版权授权给 ACM。ACM
ISBN 979-8-4007-0095-8/23/06...$15.00 https://doi.org/10.1145/3579371.3589076

1 引言

随着增强现实/虚拟现实、计算摄影和智能相机等众多新兴视觉计算领域的发展，图像处理变得愈发重要。这些应用领域都面临着严格的资源和功耗限制，因此许多研究致力于构建用于图像处理的专用加速器 [6, 12, 17, 30, 31, 35]。然而，手动构建加速器不仅耗时且容易出错，还严重依赖经验启发式方法，而这些方法并不总能带来最优设计。

近期的一种趋势是根据高级算法描述自动生成加速器 [7, 16, 38]。现有的图像处理加速器生成方法要么适用范围狭窄，要么生成的设计效率低下，主要有两个原因（第3节）。首先，它们针对简单的单消费者算法进行优化，此类算法中的每个生产者阶段都只有一个消费者。面对诸如锐化滤波 [16] 和去噪 [7], 等多消费者算法时，这些方法要么必须人为地将多消费者算法转换为单消费者结构，从而增加片上存储使用量；要么增加片上存储器的总访问次数，进而增加功耗。

其次，在片上存储需求与功耗之间存在一个很大的、依赖于算法的权衡空间，而以往的研究未能探索这一空间。这是因为以往的研究假设只有一种存储器结构，而且关键的是，对所有算法以及某个算法中的所有阶段都使用相同的存储器结构。例如，FixyNN [38] 只能生成仅使用单端口SRAM的设计，而 SODA [7] 只能生成仅使用先进先出队列（双端口SRAM）的设计。实际的设计空间要大得多：对于具有 N 个阶段和 M 种存储器结构的算法，共有 M^N 个设计点，每个设计点都对应一种独特的功耗与面积权衡。

本文提出了一种编译器框架，可为图像处理生成节省存储器和功耗的加速器（以可综合的寄存器传输级形式呈现）（第 4 节）。我们不再人为限制算法和/或片上存储结构，而是允许指定通用算法和存储器配置（包括大小和端口数量）。给定算法和硬件规格后，我们的编译器会构建一个约束优化问题，在保持理论最大吞吐量的同时，最大限度地减少片上存储使用量并降低总功耗。

我们要解决的一个关键挑战是生成能够始终实现理论最大吞吐量（帧率）的加速器，适用于每一帧；毕竟，当图像处理加速器的帧率较低时，节省片上面积和功耗几乎没有意义。核心难点在于以解析方式表达吞吐量瓶颈，即片上存储器争用，而这

涉及集合计数，并且与数值优化不兼容。我们利用模板操作的数据访问模式，将集合计数转换为等价的算术运算，使其适用于数值优化（第5节）。

在此优化形式化的基础上，我们提出将行缓冲区中的多行巧妙地合并到单个存储块中，以进一步减少片上存储器的消耗。我们展示了这一技术相当于对算法有向无环图（DAG）进行静态重写，并且可以自然地集成到我们的编译器框架中（第 6节）。

我们证明，我们的优化问题属于整数线性规划（ILP），并且存在高效的求解器。因此，我们的编译器十分轻量；它能在毫秒内为常见的图像处理算法生成可综合的寄存器传输级代码。我们使用该框架生成了种类丰富的图像处理加速器，并通过专用集成电路流程和赛灵思 Spartan-7 现场可编程门阵列开发板对其进行评估。对于不同的输入尺寸，与包括 Darkroom [16], SODA [7], FixyNN [38]在内的现有方法生成的设计相比，我们的框架生成的加速器最多可将片上存储使用量和功耗分别降低 86.0% 和 62.9%。

我们使用我们的框架进行设计空间探索（DSE），探索多种存储器配置以生成帕累托最优设计。我们展示了面积与功耗之间的权衡会因算法而异，而我们的框架能够独特地实现针对特定算法的设计空间探索。

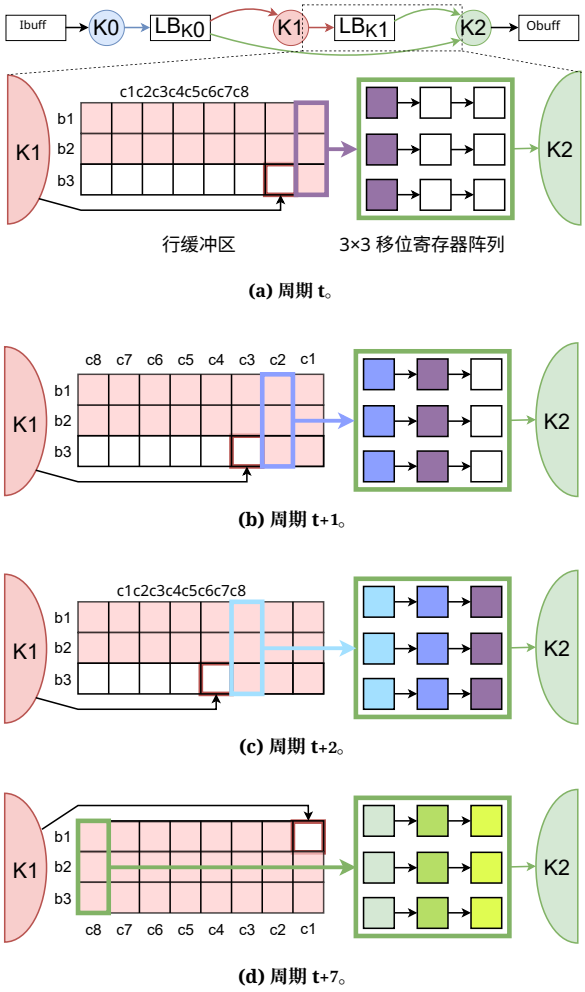
总而言之，本文作出了以下贡献：

- 我们提出了一种编译器框架，可根据通用算法和片上存储器规格生成高内存和功耗效率的图像处理加速器。这些加速器通过约束优化保证理论上的最大吞吐量。
- 我们提出了一种行合并算法，可将多条行缓冲行合并为一个存储器块，以进一步减少片上存储使用量。
- 与使用现有工具生成的加速器相比，我们的编译器生成的加速器消耗的片上存储器和功耗更低。我们的编译器集成在 DSE 流程中，可揭示算法相关的面积与功耗权衡，而以往的工具无法探索这些权衡。

2 背景

图像处理流水线由执行运算的阶段组成，运行于规则的二维像素阵列上。每个阶段执行一次模板操作，即对一个输入像素窗口进行处理，以生成一个输出像素。模板窗口按光栅扫描顺序移动。端到端算法通常由多个阶段级联而成。每个阶段都会生成一幅中间二维图像，供（可能有多个）消费者读取。常见的图像处理算法包括相机内图像信号处理 [16] 和利用连拍摄影实现的高动态范围成像 [15]。

范围。 我们的目标不是 能够运行多种算法的通用模板加速器。相反，我们专注于针对特定算法进行专门设计的加速器。这在以下两类系统中都很常见：1）基于现场可编程门阵列的加速系统，其中现场可编程门阵列可以针对特定算法重新编程；2）低功耗专用集成电路，例如现代相机中的图像信号处理器（如 Arm Mali [3]



(a) 周期 t。

(b) 周期 t+1。

(c) 周期 t+2。

(d) 周期 t+7。

图 1：行缓冲加速器。每个阶段都有专用的行缓冲区（以及相关的移位寄存器阵列）用于写入。我们使用阶段 $K1$ 之后的行缓冲区来说明行缓冲操作。行缓冲区存储三行像素，每行存储在一个双端口静态随机存取存储器中。当生产者将第二个元素写入 b3 时，c1 列中的像素会移入移位寄存器阵列。经过三个周期后，移位寄存器阵列中包含来自 3×3 模板窗口的数据，并且 $K2$ 开始。当 $K1$ 完成向 b3 的写入后，它会转而写入 b1，因为消费者将不再需要其中的像素。

以及 QualcommSpectra [4])、嵌入式计算机视觉处理-器 [11, 24, 38, 40], 和机器人加速器 [25, 32, 34], , 其中极高的效率要求使用专用算法加速器。

图像处理加速器。 图 1 展示了一个采用简单流水线的典型图像处理加速器示例。每个算法阶段都映射到专用的硬件级。输入和输出阶段（$K0$ 和 $K2$ 此处）通过输入/输出缓冲区与片外存储器通信。正如已有大量研究

所示，这些缓冲区通常采用双缓冲设计，并具有较高的访问带宽，因此并非本文的关注重点。

各阶段通过另一组片上缓冲区传递中间数据，这些缓冲区构成了片上存储使用量的主要部分。具体而言，每个生产者阶段都有专用缓冲区来写入其中间数据；该阶段的所有消费者都从此缓冲区读取数据。这与所有图像处理加速器设计一致 [5, 7, 16, 38]。

理想情况下，阶段间的数据通信流量应完全在片上完成；否则，片外存储器访问会使流水线停顿，而这需要复杂的硬件逻辑来进行动态调度，会导致帧率不确定、降低峰值吞吐量并消耗大量功耗。一种简单的方法是在片上缓冲阶段间的所有中间数据，但这有两个缺点。首先，中间数据可能很大，超出典型片上存储器的容量。例如，阶段间传递的每张 $1080p$ 图像会消耗 6 MB 数据。其次，这会人为地迫使消费者等待生产者生成完整图像后才能继续。

行缓冲区。解决这些问题的一种常见策略是使用一种称为“行缓冲区”的特殊片上缓冲区结构。关键观察是，每个流水线阶段在任意时刻都只对一个小型局部窗口中的数据进行操作。因此，只要模板窗口中的数据可用，消费者阶段就可以立即开始工作，实质上是在生产者生成像素的同时逐步消费这些像素。当消费者不再需要缓冲区中的某个像素时（即模板窗口已完全越过该像素），便可以覆写该像素，从而降低片上存储器的需求。

考虑图 $K1$ 和 $K2$ 1 中的$K2$ 每个周期都对 3×3 输出中的一个模板窗口进行操作，而该输出来自 $K1$。为支持这种数据通信模式，硬件使用一个行缓冲区，用于存储由 $K1$生成的三行像素；该行缓冲区连接到一个 3×3 移位寄存器阵列，该阵列保存模板窗口中的数据，并由 $K2$读取。

生产者从第一行开始写入，每个周期写入一个像素。当生产者完成两行加一个元素的生成后（图1a），消费者便可以将第一列的

像素（此处为 c1 ）从行缓冲区移至移位寄存器阵列。在下一个周期（图1b）中，随着生产者写入 b3 的第三个元素，c2 列中的像素被移至移位寄存器阵列。三个周期后（图1c），移位寄存器阵列中包含一个模板窗口的数据，此时 $K2$ 便可以开始生成其输出。一旦 $K1$ 完成向 b3 的写入，它将不再写入新的一行，而是写入行缓冲区中的第一行（b1），覆盖b1中的数据，因为消费者已不再需要这些像素（图1d）。因此，行缓冲区只需存储三行像素。

实现。在线缓冲区的实际硬件实现中，可以使用3 个独立的静态随机存取存储器，每个存储器存储一行像素。在任意给定周期内，都会从全部3个静态随机存取存储器中读取数据，同时其中一个静态随机存取存储器还会被写入。被写入的静态随机存取存储器将轮流变换。因此，在此示例中，所有静态随机存取存储器都必须至少具有两个端口。请注意，当图像行大于 SRAM块大小时，可以将该行拆分为多个 SRAM 块，而无需改变行缓冲区的工作原理。

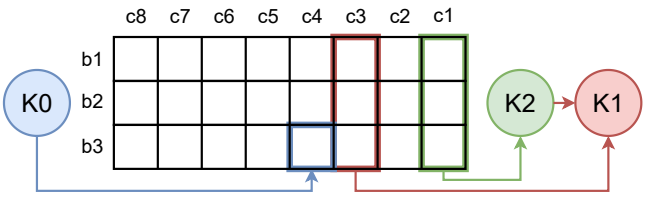


图2：具有多消费者的行缓冲区示意图。$K0$ 是正在向块 b3 中的单元 c4 写入数据的生产者。$K1$ 和 $K2$ 是分别读取 c3 列和 c1 列的消费者。每一行位于一个静态随机存取存储器块中。假设每个静态随机存取存储器具有两个端口这一常见设置，由于 b3 必须处理三次访问次数，流水线会停顿。

表1：以往工作对内存结构作出了僵化的假设，导致设计未达最优，和/或仅适用于特定形式的算法。

	Darkroom [16]	FixyNN [38]	SODA [7]	Ours
片上存储器假设	双端口	单端口	双端口（先进先出队列）	**通用**
算法适用性	单消费者	单消费者	通用	**通用**

3 动机

加速器设计决策必须根据具体的算法流水线和可用的存储器资源来制定。先前的研究对算法结构和/或可用的存储器资源作出了严格假设。因此，它们要么适用范围狭窄，要么导致设计效率低下。我们在表中总结了先前的研究。 1 并将在下文详细阐述。

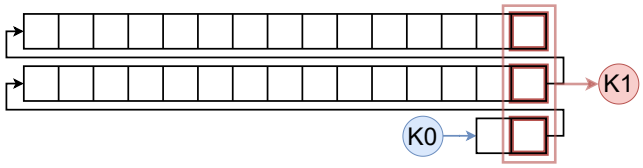
3.1 算法局限性

第 2 节中的基本设计假设使用双端口SRAM和“单消费者”流水线，即每个生产者只有一个消费者。非锐化掩模 [16] 和去噪 2D [7], 等多消费者流水线中，多个消费者会读取同一个生产者的输出，这对简单设计提出了挑战。在多消费者场景下，同一行会被多个硬件阶段访问。因此，对 SRAM 块的访问次数可能超过其端口数，从而导致流水线停顿。

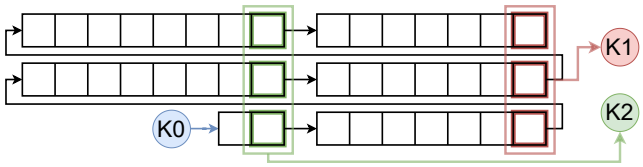
考虑图 1中的算法，其中 $K0$ 有两个消费者 $K1$ 和 $K2$。之后的行缓冲区 $K0$ 同时被三个阶段访问（生产者 $K0$ 和两个消费者）。图 2展示了一种朴素的行缓冲区设计，其中 $K0$ 正在写入（b3、c4），而 $K1$ 和 $K2$ 分别读取列 c3 和 c1。因此，三个不同的阶段正在访问块 b3。如果每个静态随机存取存储器块只有两个端口，b3 将无法处理全部三次访问。简单地增加静态随机存取存储器端口的数量会导致面积效率低下，因为静态随机存取存储器面积会随着端口数量的增加而呈二次增长 [37]。以往的工作主要通过两种方式尝试支持多消费者流水线，但每种方式都有其缺点，下面将对此进行解释。。



图 3：算法线性化示例。我们需要在 $K11$ 之后增加一个行缓冲区。请注意，尽管 $K1$ 和 $K11$ 都是 $K0$ 的消费者，但它们消耗数据的模式完全相同，实际上相当于一个消费者。



(a) 一种用于支持单消费者的先进先出队列实现 (*K* 1)



(b) 用于支持多消费者（$K1$ 和 $K2$）的先进先出队列实现。

图 4：使用先进先出队列实现的行缓冲区。图 (a)展示了单消费者的情况，(b) 展示了多消费者的情况，假设每个存储块有两个端口。在多消费者的情况下，为了同时满足两个消费者的需求，每个先进先出队列都拆分为两个较小的先进先出队列。先进先出队列通常实现为双端口 SRAM 块。生产者正在写入的行通常只包含少量元素（此处为 2 个），因此可以使用 D触发器将其实现为移位寄存器，以节省片上存储使用量。

算法线性化。 可以将具有多消费者阶段的算法转换为另一个功能等价的算法，使其仅包含单消费者阶段；Darkroom 将这一过程称为“线性化” [16]。

我们使用图 3 来说明线性化。为了将流水线线性化，我们在 $K11$ 和 $K0$ 之间添加一个虚拟阶段 $K2$。每个周期，$K11$以 $K0$ 中完全相同的模式读取数据，$K1$ 但不对数据执行任何计算。从本质上说，$K11$ 的唯一目的就是将数据从 $K0$ 中转发到 $K2$。因此，$K2$ 从$K11$ 读取数据，而不是直接从 $K0$读取数据。关键在于，尽管 $K0$仍有两个消费者 $K1$ 和 $K11$，这两个消费者每个周期都以完全相同的模式从 $K0$ 读取数据，因此它们实际上充当单消费者，无需额外的存储器端口。

这种方法的缺点是，线性化算法比原始算法增加了片上存储使用量，因为每个虚拟阶段都需要一个专用行缓冲区。在这种情况下，与 $K11$ 关联的额外行缓冲区缓存了与 $K1$ 相同的数据，因此是冗余的。

拆分行缓冲区。或者，可以将一个由多个消费者共用的行缓冲区拆分为多个较小的行缓冲区，每个仅服务于一个消费者。这种方法意味着必须将数据从一个行缓冲区传输到另一个行缓冲区，因此每个行缓冲区都必须实现为先进先出队列。SODA [7] 和赛灵思的行缓冲设计 [5]就是这种方法的例子。

图 4a展示了仅有单消费者时基于先进先出队列的行缓冲区实现。通常，先进先出队列由双端口 SRAM/BRAM 存储块实现。因此，每个周期都会对每个存储块进行一次读访问和一次写访问。当两个消费者试图访问行缓冲区时，必须将每个先进先出队列拆分为两个先进先出队列，并分别置于独立的存储块中，以满足两次访问的需求；如图 4b 所示。请注意，如果先进先出队列变得非常小（例如，只有几个元素），这通常是生产者正在写入的那一行的情况，那么可以使用 D 触发器将其实现为移位寄存器，以减少存储器用量。

先进先出队列的缺点是功耗较高，因为先进先出队列的特性决定了每个周期都会始终对每个静态随机存取存储器块进行两次访问，而在经典实现中，只有三行中的一行需要处理两次访问；其他行每个周期仅需访问一次。在我们的现场可编程门阵列测量中（详见第7节），每个周期进行两次访问的BRAM比每个周期仅进行一次访问的BRAM功耗高约35%。

3.2 硬件限制

以往方法的一个根本限制是，它们对所有算法以及算法中的所有行缓冲区都使用相同形式的存储器，例如 Darkroom 中的双端口静态随机存取存储器 [16] 或 FixyNN 中的单端口静态随机存取存储器 [38]。然而，实际的设计空间要大得多。对于包含 N 个阶段且其中的行缓冲区具有 M 种实现选项的算法，共有 M^N 个设计点。由存储器资源所决定的大型设计空间的探索尤其重要，这对专用集成电路设计而言更是如此：不同于存储器资源固定的现场可编程门阵列，专用集成电路设计可以根据面积和/或功耗目标灵活定制存储器（例如大小和端口）。

至关重要的是，在如此庞大的设计空间中进行探索时，存在功耗与面积之间的权衡。例如，增加存储器端口的数量会增加每个静态随机存取存储器的面积和功耗，但也会减少所需的 SRAM 块数量。正如我们将在第8.5节中展示的那样，确切的帕累托最优前沿因算法而异，而现有方法无法进行这样的设计空间探索。

最后值得注意的是，当可用存储器资源无法满足需求时，先前的方法根本无法生成任何硬件设计。例如，SODA 提出的先进先出队列方法 [7] 假设使用双端口存储器。因此，在仅有单端口存储器可用时，该方法无法运行，这进一步凸显了考虑任意存储器配置的必要性，而我们的框架正提供了这一能力。

4 框架概述

图 5展示了我们框架的整体工作流程。该框架接收以领域特定语言（DSL）描述的图像处理流水线，以及可用存储器资源的描述（即大小和端口数量），并生成可综合的Verilog代码。

前端。 文献中有许多用于表达图像处理流水线的领域特定语言 [16, 29], 这并非 本文的重点。为简单起见，我们使用一种类似于 Darkroom 的领域特定语言 [16]。代码

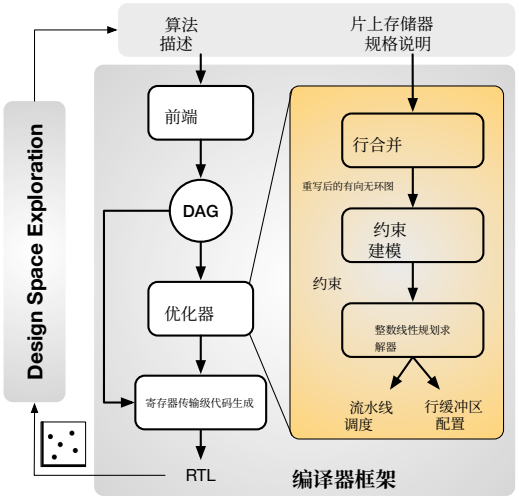


图5：编译器框架，根据图像处理算法和片上存储器规格生成可综合的Verilog代码。该框架可集成到设计空间探索工具中，以生成帕累托最优设计。

下面展示了图 1 中的算法。每个阶段都定义在 `im` 块中。 `input` 和 `output` 表示流水线的输入阶段和输出阶段，这些阶段允许访问片外存储器。前端将算法解析为有向无环图，作为中间表示。有向无环图中的每个节点都是一个流水线阶段，每条边连接一个生产者-消费者对。模板窗口大小编码在有向无环图节点中。

输入 *K0*；*//K1* 从 *K0* 读取 *3x3* 窗口 $K1 = im(x,y)K0(x-1,y-1)+...+K0(x+1,y+1)$ 结束 *//K2* 从 *K1* 读取 *3x3* 窗口，并从 *K0* 读取 *2x2* 窗口 输出 $K2 = im(x,y)K0(x,y)+...+K0(x+1,y+1)+K1(x-1,y-1)+...+K1(x+1,y+1)$ 结束

优化器。 我们工作的核心贡献是优化器，它以有向无环图和存储器资源规格为输入，生成加速器流水线调度和片上存储器配置，然后由代码生成器据此生成可综合的寄存器传输级代码。

优化器构建一个约束优化问题，以最小化片上存储使用量（第 5 节）。不同于传统的行缓冲区综合器 [7, 16, 38]，编译器通过简单的有向无环图重写，利用将图像行合并到同一缓冲区的机会，从而进一步降低片上存储使用量（第 6 节）。

寄存器传输级代码生成。 给定流水线调度和行缓冲区配置，代码生成器会生成可综合的寄存器传输级代码。生成的代码包含执行模板操作的计算单元、实现行缓冲区的存储器模块，以及用于控制硬件执行顺序的逻辑。

值得注意的是，代码生成基本上是将每个流水线阶段中的算术运算机械地翻译为寄存器传输级代码，因此并非本文的贡献。事实上，任何现有的 HLS 工具（例如 Vivado HLS）可用于代码生成：我们可以使用我们的优化器生成最优的行缓冲区配置，然后在 HLS 代码中使用该配置标注存储器大小。据我们所知，目前的 HLS 工具（如 Vivado HLS）要求程序员显式指定行缓冲区大小，而我们的优化器可以自动生成这些大小。

表 2：我们公式中使用的符号列表。下标i、 j、 p、 c用于表示流水线阶段； p和 c分别表示生产者和消费者。黑板粗体符号 $\mathbb{N}$、 $\mathbb{C}$、 $\mathbb{A}$ 表示集合。

符号	含义
N	流水线中的阶段总数
LB_i	与阶段 i关联的行缓冲区大小
S_i	阶段的启动周期 i
$\mathbb{N}_i$	访问阶段 i的行缓冲区的阶段集合
SH_i	阶段 i[1]的模板高度
$B_{l,t}$	周期 t 对一行 l的访问总次数
P	SRAM 端口数量
W	每个阶段输入图像的宽度[2]
$\mathbb{C}_p$	生产者阶段的消费者阶段集合 p
$\mathbb{A}_{i,t}$	阶段 i 在周期 t[3]访问的行集合
$L_{i,t}$	阶段 i 在周期 t访问的第一行

5 生成行缓冲流水线

我们首先介绍这一总体思路背后的直觉（第 5.1 节），随后给出严格的优化形式化（第 5.2 节）。然后，我们讨论如何对片上存储器争用进行建模，这是我们的优化形式化的关键（第 5.3 节），随后介绍一种消除冗余硬件约束的技术（第 5.4 节）。最后，我们将说明我们的形式化最终构成一个整数线性规划问题（第 5.5 节）。

5.1 总体思路与直觉

目标。 我们的目标是在 保持理论最大吞吐量的同时，尽量减小片上存储器的总大小，该吞吐量通过每周期生成的像素数量来量化。理论最大吞吐量从根本上受硬件能够承担的功能单元数量限制。与几乎所有先前的工作一样， [7, 16, 38]，我们假设理论最大吞吐量为每周期一个像素。提高吞吐量只需增加计算资源，而这并非 本文 的重点。不过请注意，对于现实世界的应用而言，每周期一个像素的假设是合理的。假设

[1]一个消费者阶段可以访问来自多个生产者的数据，因此可能有多个模板高度。为简化起见，我们在符号中省略生产者阶段，但在每个出现 h_i 的上下文中，生产者阶段都显而易见。[2]当前系统本身能够处理不带填充的阶段；在这种情况下，各阶段的输入图像尺寸不同，并且可以根据模板窗口大小轻松计算得出。为使论述简洁，本文假设使用填充，并在全文采用相同的 W。[3]与 SH_i 类似，$\mathbb{A}_{i,t}$ 在此与 i 的某个特定生产者绑定，而 i 可能有多个生产者。为简化起见，符号中省略了生产者，但结合上下文应能明确其所指。

100 MHz 时钟频率下，每周期产生一个像素，相当于为 1080p 图像提供每秒 50 帧（FPS）的帧率，足以满足实时运行的需求。

我们强调，加速器必须持续在各帧之间提供规定的吞吐量。即使平均帧率令人满意，如果某些帧的吞吐量低于其他帧，也是不被接受的，因为变化的帧率会带来迟缓的用户体验。为确保吞吐量一致，加速器流水线不得停顿（一旦流水线启动，在所有输入像素处理完之前都不会停顿），这意味着必须满足三个要求：

$R1$ 数据依赖（因果关系）：任何像素在被消费者读取之前，都必须已由生产者生成并存放在线缓冲区；

$R2$ 无需中间片外存储器访问次数：仅当某个像素不再被其任何消费者需要时，才会将其从行缓冲区中逐出（以避免之后的 *DRAM* 访问次数）；

$R3$ 无片上存储器访问停顿：在任意周期内，对任一片上存储块的访问次数不得超过其端口数量。

解决方案直觉。直观而言，同时满足这两项存储器要求的通用解决方案是延迟某些消费者的启动时间。例如，当同一生产者的两个消费者被允许在同一周期启动时，每个存储块都必须服务两次同时发生的读取访问。现在，如果将其中一个消费者延迟W个周期，其中W是图像的宽度，那么两个消费者将从两个不同的存储块（图像行）读取数据，从而避免存储器端口争用。延迟消费者的启动时间还可以为生产者生成中间数据提供更多时间。

然而，推迟消费者会增大行缓冲区的大小，因为只有当所有消费者都不再需要某个元素时，它才会从行缓冲区中移除；推迟消费者意味着数据必须在行缓冲区中保留更长时间，从而增大行缓冲区的大小。因此，核心挑战在于如何进行最优的移位：如何调度不同的流水线阶段，在满足数据和硬件约束的同时，将行缓冲区的总大小降至最低。

5.2 优化形式化

形式上，我们的硬件生成器的任务可以描述为一种受约束的优化形式化：

$$\min_{\phi} LB(\phi) = \sum_{i=0}^{N-1} LB_i(\phi),$$

$$where\ \phi = \{S_i\}, i \in [0, 1, \cdots, N-1] \tag{1a}$$

$$s.t.\ \forall (p,c)\ S_c - S_p \geq (SH_c - 1) \times W + 1, \tag{1b}$$

$$\forall l \forall t\ B_{l,t}(\phi) \leq P. \tag{1c}$$

优化目标。公式1a 给出了优化目标。ϕ表示调度，即所有阶段启动周期的集合$\{S_i\}$（i是介于0和 $N-1$之间的整数，其中N是流水线阶段的数量）。$LB(\phi)$ 表示行缓冲区的总大小，即N个独立行缓冲区大小之和。回顾图1 可知，每个阶段都对应一个行缓冲区，因此行缓冲区的数量与流水线阶段的数量相同，N。

每个行缓冲区的大小由生产者阶段和消费者阶段的启动周期决定。例如，对于流水线中的一个生产者-消费者对（p，c），其启动周期分别为S_p和S_c，消费者与生产者之间存在（$S_c - S_p$）个周期的延迟。根据上述 ~~$R2$~~，在消费者开始处理之前，行缓冲区必须至少能够缓冲（$S_c - S_p$）个像素，因为每个像素只有在被消费之后才能从行缓冲区中移除。

考虑到同一个生产者可能有多个消费者，并且只有在最后一个消费者读完某个元素后，才能将该元素从行缓冲区中移除，因此特定阶段 p 的行缓冲区大小为：

$$LB_p = \left\lceil \frac{\max\{S_c - S_p\}}{W} \right\rceil \times W,\ \forall c \in \mathbb{C}_p, \tag{2}$$

其中，c 是 p的一个消费者，记为 $\mathbb{C}_p$。向上取整操作用于确保行缓冲区始终存储整数行，因此，实际行缓冲区大小必须是W的整数倍，其中W是图像宽度。

数据依赖。公式。1b 规定了数据依赖要求 $R1$：一个元素必须位于行缓冲区中，之后才能被其消费者读取。由于模板计算的特性，消费者读取的数据可能跨越多个图像行。因此，消费者必须等待行缓冲区中包含一定数量的像素。例如，如果我们有一个模板大小为 3×3 的消费者，则该消费者必须等到行缓冲区包含两整行像素以及第三行中的一个像素（见图 1a）。

通常，对于任意生产者-消费者对（p，c），消费者的启动周期必须延迟（$SH_c - 1$）$\times W + 1$ 个周期，即在生产者启动后再启动，其中 SH_c 是消费者读取的模板窗口的高度。

公式 1c 给出了片上存储器约束 $R3$，该约束的表达复杂得多，下面将对此进行讨论。

5.3 片上存储器争用建模

公式 1c 指出，在任意给定周期 t，行缓冲区中任意一行 l的访问次数 $B_{l,t}$不得超过包含该行 l[4]的 SRAM 块的端口数（P）。关键在于如何以数学方式表达 $B_{l,t}$。为此，我们首先表示每个周期各阶段访问的行集合。

考虑在周期 t访问行缓冲区的流水线阶段 i。阶段 i在 t访问的第一行，记为 $L_{i,t}$，如下所示：

$$L_{i,t} = \left\lceil \frac{t - S_i}{W} \right\rceil. \tag{3}$$

因此，阶段 i在周期 t 的访问集合，即阶段 i访问的行集合，为[5]：

$$\mathbb{A}_{i,t} = \left\{ L_{i,t},\ L_{i,t} + 1,\ \cdots,\ L_{i,t} + SH_i - 1 \right\}. \tag{4}$$

为满足硬件约束，任何行都不能属于超过 P 个集合的交集。这等价于说，

[4]理论上，P应表示为 P_l，以表明每个静态随机存取存储器都可以具有不同数量的端口。为简化起见，我们假设硬件中的所有静态随机存取存储器都具有相同数量的端口。不过，我们的公式很容易扩展，以支持不同的端口数量。[5]这里有一点需要注意：向行缓冲区写入的阶段，其模板高度始终为 1。

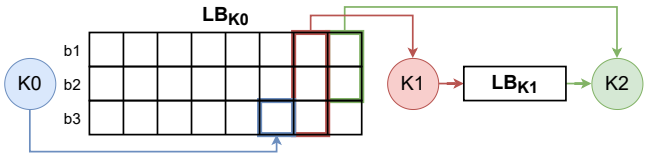


图6：说明如何计算访问集合的示例。在图中所示的当前周期 t，$K0$的访问集合为$A_{0,t}$=(b3)，$K1$的访问集合为 $A_{1,t}$=(b1,b2,b3)，而 $K2$的访问集合为 $A_{2,t}$=(b1,b2)。假设每行存储在双端口静态随机存取存储器中，硬件约束将为$A_{0,t} \cap A_{1,t} \cap A_{2,t} = \emptyset$(公式6)，经过约束剪枝后缩减为$A_{0,t} \cap A_{2,t} = \emptyset$ (第 5.4 节)。

任何 $(P+1)$-组合的交集始终为空集。因此，每个阶段 i 的硬件约束可用数学形式表示为：

$$\forall t \; \forall \mathcal{T} \in \binom{\mathbb{N}_i}{P+1} \bigcap_{i \in \mathcal{T}} \mathbb{A}_{i,t} = \emptyset, \tag{5}$$

其中，$\binom{\mathbb{N}_i}{P+1}$表示 $\mathbb{N}_i$ 的所有$(P+1)$-组合的集合，而 $\mathbb{N}_i$ 本身是访问阶段 i 的行缓冲区的所有阶段的集合。

为了具体说明公式5，请看图6中的示例，其中行缓冲区 LB_{K0} 由一个生产者（$K0$）和两个消费者（$K1$ 和 $K2$）访问。假设静态随机存取存储器有两个端口，这是静态随机存取存储器/BRAM 块中的常见配置，则LB_{K0} 的硬件约束表示为：

$$\forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{1,t} \cap \mathbb{A}_{2,t} = \emptyset \tag{6}$$

其中 $\mathbb{A}_{0,t}$, $\mathbb{A}_{1,t}$, 以及 $\mathbb{A}_{2,t}$分别是周期 t 时阶段 $K0, K1,$ 和 $K2$ 的访问集合。

为了施加这一约束，可以施加以下三个约束中的任意一个，其直觉在于：如果任意两个集合的交集为空，那么三个集合的交集必然为空：

$$\forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{1,t} = \emptyset, \tag{7a}$$
$$\forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{2,t} = \emptyset, \tag{7b}$$
$$\forall t \; \mathbb{A}_{1,t} \cap \mathbb{A}_{2,t} = \emptyset. \tag{7c}$$

公式7a – 公式7c 涉及计算集合交集（或等价地，进行计数），这不适用于通常的数值优化。我们必须将它们转换为等价的数值表达式。

不失一般性，考虑约束 $\forall t \; \mathbb{A}_{i,t} \cap \mathbb{A}_{j,t} = \emptyset$。施加该约束等价于要求 s_i 最后写入的行（在任意周期 t）必须位于 s_j 首次读取的行之上。也就是说：

$$\forall t \; L_{i,t} + SH_i - 1 < L_{j,t}, \tag{8}$$

应用公式 3后，可得：

$$\forall t \left\lceil \frac{t - S_i}{W} \right\rceil + SH_i - 1 < \left\lceil \frac{t - S_j}{W} \right\rceil. \tag{9}$$

方程 9 取决于 t，而它没有上界。因此，t必须消除，才能使该约束可用。我们的策略是（以某种方式）移除上取整运算符（$\lceil \rceil$），这样便可以使 t 能够从方程 9两边同时消去。为此，注意：

$$x \leq \lceil x \rceil < x + 1, \tag{10}$$

由此可以推导出：

$$\left\lceil \frac{t - S_i}{W} \right\rceil < \left(\frac{t - S_i}{W} \right) + 1, \; and$$
$$\left(\frac{t - S_j}{W} \right) \leq \left\lceil \frac{t - S_j}{W} \right\rceil. \tag{11}$$

将公式 11 与公式 9结合起来，可以将公式9 转换为以下约束：

$$\forall t \left(\frac{t - S_i}{W} \right) + 1 + SH_i - 1 \leq \left(\frac{t - S_j}{W} \right)$$
$$\equiv S_i - S_j \geq W \times SH_j. \tag{12}$$

式 12 现在独立于 t。注意，式 12 是比式 9更严格的约束，这意味着通过式 12 获得的解是通过式 9获得的解的子集，从而牺牲了解的最优性。然而，这种权衡带来的好处是，该约束独立于 t。随后应用式 12 重写式 7a、式 7b和式 7c。

5.4 约束剪枝

一个潜在问题是，式 7a、式 7b和式 7c 中的约束需要通过“或”连接；也就是说，三个约束中只需满足一个。通常，这需要我们构建三个不同的子优化问题，分别单独考虑其中一个约束。当流水线包含多个阶段，且每个阶段都有需要通过“或”连接的约束时，子问题的总数会呈组合式增长。

为减少优化时间，我们发现公式中的约束 7a、公式 7b和公式 7c 并非相互排斥，因此可以剪除其中一些约束。我们以图 6中的示例来直观说明约束剪枝，然后讨论如何将其扩展到一般情况。

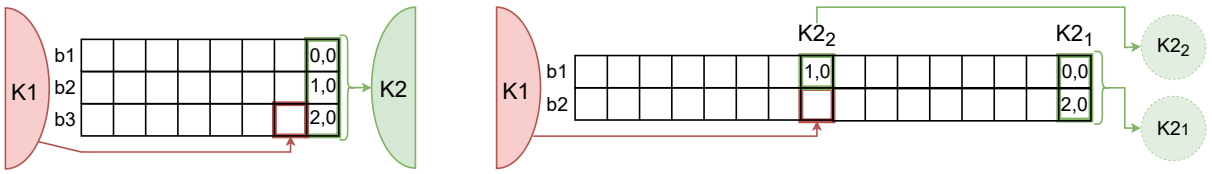
示例。 注意，公式 7b 中的约束比公式 更宽松7a 以及公式 7c。也就是说，如果公式 7a （或公式7c）成立，则公式7b必然成立：

$$\forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{1,t} = \emptyset \implies \forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{2,t} = \emptyset, \tag{13a}$$
$$\forall t \; \mathbb{A}_{1,t} \cap \mathbb{A}_{2,t} = \emptyset \implies \forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{2,t} = \emptyset \tag{13b}$$

其中 $A \implies B$ 读作“A 意味着 B。”

直观而言，等式 13a 成立，是因为阶段 $K2$ 依赖于阶段 $K1$的数据，这意味着 $K2$ 必须在 $K1$之后开始。因此，在任意时刻，$K2$ 写入的第一行都必须位于 $K1$ 写入的第一行下方（等式 14a）；而根据 $\forall t \; \mathbb{A}_{0,t} \cap \mathbb{A}_{1,t} = \emptyset$，后者又必须位于 $K0$ 写入的最后一行下方（等式 14b）。因此，根据传递性，$K2$ 写入的第一行位于 $K0$写入的最后一行下方

[6]该证明是对“小于”（$<$）关系的传递性的简单应用，受篇幅限制，此处略去。



(a) 未进行行合并。 (b) 使用行合并。

图 7：行合并优化示意图。图 (a) 展示了采用行合并的实现，(b) 展示了采用行合并的实现，并假设每个存储块有两个端口。在行合并中，我们将连续的两行放在同一个存储块中。实际上，DAG 中的消费者阶段 $K2$ 被替换为两个"虚拟"阶段，$K2_1$ 和 $K2_2$，其模板高度分别为 2 和 1。

写入（等式 14c），因此 $\forall t\ \mathbb{A}_{0,\ t} \cap \mathbb{A}_{2,\ t} = \emptyset$。

$$First(\mathbb{A}_{2,t}) > First(\mathbb{A}_{1,t}), \tag{14a}$$
$$First(\mathbb{A}_{1,t}) > Last(\mathbb{A}_{0,t}) \tag{14b}$$
$$\implies First(\mathbb{A}_{2,t}) > Last(\mathbb{A}_{0,t}) \tag{14c}$$

公式的有效性13b 同样可以根据以下事实进行推理：$K1$ 数据依赖于 $K0$。

一般来说，给定两个约束 A 和 B，需要将它们进行"或"运算，若 A 比 B更宽松，则可以安全地删除 B，而不会牺牲最优性，因为任何满足 B 的解也必定满足 A。因此，在上面的示例中，我们可以删除公式7a 和公式7c中的约束。

泛化。 上述示例表明，数据依赖关系是消除冗余约束的关键。具体而言，数据依赖关系使我们能够建立阶段之间的偏序 $\preccurlyeq$。如果有一条从阶段 i到阶段 j的有向无环图路径，即阶段 j（直接或间接）依赖于阶段 i，则有一个偏序 $i \preccurlyeq j$。偏序关系具有自反性：$i \preccurlyeq i$。我们有以下定理。

定理。给定两个一般约束 C_1 和 C_2:

$$C_1 : \forall t\ \mathbb{A}_{x,t} \cap \mathbb{A}_{y,t} = \emptyset \tag{15}$$
$$C_2 : \forall t\ \mathbb{A}_{z,t} \cap \mathbb{A}_{w,t} = \emptyset \tag{16}$$

如果 $x \preccurlyeq z$、$w \preccurlyeq y$ 和 $SH_x \le SH_z$，则 C_1 比 C_2 更宽松（即，C_2 蕴含 C_1）。

证明。$x \preccurlyeq z$导出等式 17a，结合 $SH_x \le SH_z$可得等式 17b；$w \preccurlyeq y$可得等式 17c；C_2可得等式 17d；由传递性可得等式 17e，由此得到 C_1。因此，C_2意味着 C_1，且根据 C_1可将其消去。

$$First(\mathbb{A}_{x,t}) < First(\mathbb{A}_{z,t}), \tag{17a}$$
$$Last(\mathbb{A}_{x,t}) < Last(\mathbb{A}_{z,t}), \tag{17b}$$
$$First(\mathbb{A}_{w,t}) < First(\mathbb{A}_{y,t}), \tag{17c}$$
$$Last(\mathbb{A}_{z,t}) < First(\mathbb{A}_{w,t}), \tag{17d}$$
$$\implies Last(\mathbb{A}_{x,t}) < First(\mathbb{A}_{y,t}) \tag{17e}$$

该定理是一条剪枝规则：我们检查每一对约束，并使用剪枝规则消除其中较严格的约束（如果存在）。请注意，给定两个约束时，可能无法判断哪一个更宽松或更严格，因为有向无环图中的两个阶段之间不一定总是存在偏序关系。

5.5 问题结构与求解器

我们所构建的优化问题是一个整数线性规划问题：优化变量是各阶段的起始周期，且均为整数；目标函数（式 1a）和约束（式 1b和式 12）均为线性。请注意，目标函数子项（式 2）中的向上取整操作可以去除而不影响解的最优性，因为最小化 $f(\lceil x \rceil)$ 等价于最小化 $f(x)$，条件是 f单调递增于 [7]。整数线性规划建模使我们能够使用成熟的求解器，快速得出最优的行缓冲器设计。

6 行合并优化

到目前为止，我们假设每个存储器（例如，静态随机存取存储器/BRAM）存储块包含一行。然而，存储块的容量可能足以容纳多行，在这种情况下，将多行合并到一个存储块中可以进一步降低存储器需求。挑战在于如何在行合并下生成行缓冲流水线。我们表明，上述优化形式化可以自然地扩展，以支持最优的行合并。

请看图 7中的示例，其中包含两个阶段：一个生产者 $K1$ 和一个消费者 $K2$；该消费者使用的模板高度为 3。暂且假设每个存储块有两个端口。图 7a 和图 7b 分别展示了未采用和采用行合并的行缓冲区实现。由于每个存储块有两个端口，我们最多可以将两行合并到一个存储块中，如图 7b所示。现在，$K2$ 访问的三个元素分布在两个而非三个存储块中，因为元素 (0,0) 和 (1,0) 位于同一个存储块中。

为了向优化器表达行合并后的流水线，我们观察到，行合并等效于对有向无环图进行变换：将原始阶段 $K2$ 替换为两个新的"虚拟"阶段 $K2_1$ 和 $K2_2$。在此示例中，两个虚拟阶段同时访问 b1，而只有 $K2_1$（以及生产者 $K1$）访问 b2。因此，虚拟阶段 $K2_1$ 的有效模板高度现在为 2，而虚拟阶段 $K2_2$ 的有效模板高度为 1。两个虚拟阶段都继承了原始阶段 $K2$ 的生产者和消费者。

[7]更严格地说，我们有：$\operatorname{argmin} f(x) \in \operatorname{argmin} f(\lceil x \rceil)$。也就是说，对于任意单调递增函数 f，使 $f(x)$ 最小的解必然也是使 $f(\lceil x \rceil)$ 最小的解。为证明这一点，令 $x_0 = \operatorname{argmin} f(x)$；则 $\forall x\ f(x) \ge f(x_0)$，因此 $\forall x\ x \ge x_0$（因为 f是单调递增的）。于是，$\forall x\ \lceil x \rceil \ge \lceil x_0 \rceil$，这意味着 $\forall x\ f(\lceil x \rceil) \ge f(\lceil x_0 \rceil)$，即 $x_0 \in \operatorname{argmin} f(\lceil x \rceil)$。

我们可以将行合并推广到具有 P 个端口的存储块，在这种情况下，我们最多可以合并同一存储块中的 P 行，并用 P 个虚拟阶段替换原始消费者阶段。此变换可以离线完成，因为它仅取决于算法DAG 和模板大小。算法1 描述了通用变换算法。

算法 1： 通过DAG重写实现的行合并算法。符号说明： P 表示端口数， SH_i 表示阶段 i读取的模板高度。

数据： 原始DAG
结果： 变换后的DAG
i = 原始DAG的输入节点;
当i 不是空节点**时**
  如果i 不是输入节点，**则**
    $K = \min(P, SH_i)$;
    将 i 拆分为 K 虚拟阶段;
    对于 从 i 拆分出的每个虚拟阶段 v **执行**
      将 v 的生产者设置为 i 的生产者;
      将 v 的消费者设置为 i 的消费者;
    end
  end
  i =通过广度优先搜索获取下一个节点;
end
返回 新的有向无环图;

从优化器的角度来看，变换后的有向无环图不过是另一条流水线，只是属于同一物理阶段的所有虚拟阶段必须共享相同的启动周期，因为从逻辑上讲，它们必须同步运行。使用方程式 1 中的优化公式，优化器会为新有向无环图中的每个阶段生成最优的启动周期。代码生成器特别注意的一点是，属于同一物理阶段的虚拟阶段会从不同的、但由离线预先确定的偏移量处读取数据。例如，在图 7 中， $K2_2$ 将始终从偏移量 W（图像宽度）处读取 b1，其中 $K2_1$ 和 $K1$ 的偏移量为 0。

备注。 请注意，行合并优化从根本上与基于 FIFO 的方法不兼容 [7]，也不兼容假设使用单端口存储器的设计 [38]——只需观察图7b中的数据访问行为即可。

行合并对现场可编程门阵列和 ASIC 后端都有益处。对于现场可编程门阵列，任何特定开发板上的 BRAM 存储块大小都是固定的；强制每个存储块仅容纳一行，可能导致 BRAM 存储块出现内部碎片。ASIC 设计人员可以针对算法定制存储器；他们可以适当设定存储块的大小，以便进行行合并，从而减小总体面积（第 8.5节）。

7 实验方法

编译器实现。 我们使用 Python 实现了编译器，代码约有 1,500 行。我们使用 Google 的优化库“or-tools” [2] 来求解整数线性规划问题。

硬件平台。 我们评估了 ASIC 流程和 FPGA 流程。我们使用一块赛灵思 Spartan-7 现场可编程门阵列开发板（xa7s100fgga488- 2I）进行评估。该开发板有 120 个 BRAM 块，每个块的容量为 36 Kbits。每个块都可配置为单端口或双端口存储块。我们假设 ASIC 后端提供的 SRAM 块容量为

表 3：评估算法。-s 或 -m 分别表示算法仅包含单消费者阶段，或至少包含一个多消费者阶段。最后一列显示多消费者（MC）阶段的数量。

算法	描述	# 阶段	# 个 MC 阶段
Canny-s	Canny 边缘检测	9	0
Canny-m		10	1
Harris-s	Harris 角点检测	7	0
Harris-m		7	1
非锐化掩模-m	非锐化掩模处理	5	1
Xcorr-m	互相关	3	1
Denoise-m	图像去噪	5	2

64KB，用于行缓冲区。我们评估两种图像分辨率： $320p$（480×320）和 $1080p$（1920×1080）。SRAM 和 BRAM 块的大小确保行合并适用于 $320p$，但不适用于 $1080p$，因为对于 $1080p$ 图像，块的容量不足以容纳多行。

对于 FPGA 后端，生成的 Verilog 代码将使用 Vivado Design Suite 2021.1 进行 FPGA 综合和布局流程。我们使用 Vivado 的资源监视器报告 BRAM 的使用量。FPGA 通过 AXI DMA 与主机通信。我们首先通过 DMA 将输入图像从主机内存加载到 BRAM 中。报告的帧率（每周期 1 个像素）是加速器启动后的吞吐量，即稳态吞吐量。对于每种设计，我们都会进行实现后功能仿真以获取开关活动，然后使用 Vivado 的功耗分析工具估算功耗。

对于 ASIC 后端，我们构建了一个周期级模拟器来模拟行缓冲流水线。我们使用开源存储器编译器 OpenRAM[14] 和 FreePDK45 [1] 估算每次访问的 SRAM 功耗，再结合模拟器给出的访问次数，估算总存储器功耗。

由于本文旨在减少片上存储器的大小和能耗，因此我们主要报告与片上存储器相关的结果，但也会展示整个加速器的节省情况。请注意，存储器面积占加速器面积的主要部分，因此预计存储器面积/功耗的节省将转化为类似的总面积/功耗节省。在 ASIC 后端，静态随机存取存储器面积在所有算法中平均分别占 $320p$ 和 $1080p$ 图像对应的整个加速器面积的 79.8% 和 92.7%。存储器面积占主导地位的原因在于，行缓冲加速器中的处理单元非常少：为了执行 3×3 卷积，无论输入是什么，我们只需要 3×3 个乘加单元（见图 1）。对于两种分辨率，总面积平均分别为 0。65 mm^2 和 1。84 mm^2，而总平均功耗分别为 72。9 mW和 98。3 mW。

算法。 我们评估了表中列出的常见图像处理算法。 3，每种算法的名称要么以“-s”结尾，表示它只有单消费者阶段；要么以“-m”结尾，表示它至少有一个多消费者阶段。根据实现细节，Canny 和 Harris 各有两个版本。

基线与变体。 我们将其与三种常见的行缓冲图像处理加速器进行比较。

• FixyNN [38], 其基于第 2 节中描述的相同设计，但仅使用单端口SRAM。
• SODA [7],，它使用先进先出队列实现行缓冲区，并拆分先进先出队列以支持多消费者阶段（第3节）。这些先进先出队列使用双端口SRAM实现（而非移位寄存器）。
• Darkroom [16],，它将多消费者流水线线性化（第3节），并使用双端口静态随机存取存储器。

我们考虑框架的两个变体：Ours 和 Ours+LC。后者在前者的基础上增加了行合并。

8 评估结果

我们首先展示，我们的编译器能够保持理论最大吞吐量（第 8.1 节），并且执行速度很快（第 8.2节）。然后我们展示，与现有方法相比，我们的框架生成的硬件消耗更少的存储器资源（第8.3 节），并且具有更低的功耗（第8.4 节）。最后，我们展示我们的框架可以帮助针对单个算法定制存储器模块（第 8.5 节）。

8.1 吞吐量和延迟

对于所有算法，我们的编译器生成的硬件都能保持每周期处理一个像素的恒定吞吐量，这一目标已在第 5.1 节中阐述并论证。 我们的方法相比 Darkroom 和 SODA，平均端到端延迟仅增加 0.01%。因此，后文展示的存储器和功耗节省不会造成任何速度下降。

8.2 编译速度

平均而言，我们的编译器在所有算法上生成Verilog代码需要 14.5 ms。对于多消费者算法，约束剪枝（第 5.4 节）平均可将编译时间缩短4×。这是通过剪除原本会导致许多子优化问题的冗余约束实现的。与Darkroom的线性化编译器相比，我们的编译器平均快 37.4%。这是因为线性化会添加虚拟阶段，从而向整数线性规划中加入更多约束。

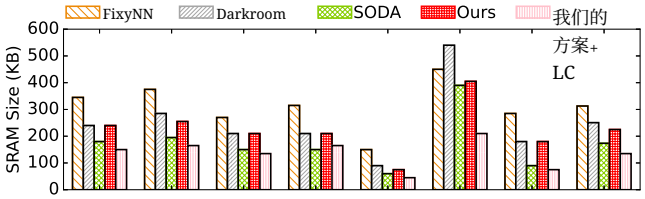
可扩展性。我们还对长度从 9 到 60 不等的不同流水线进行了测试。在每种算法中，三分之一的阶段有多个消费者。编译器编译 9 阶段流水线耗时 8.7 毫秒，编译 60 阶段流水线耗时 8.1 秒，这表明我们的编译器具有良好的可扩展性。

8.3 片上存储器需求降低

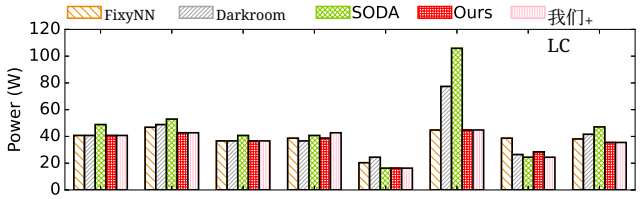
我们的框架生成的加速器显著减少了片上存储器的容量。图 8a 比较了我们的框架与三个基线方案生成的硬件的静态随机存取存储器容量，测试对象为 320p 张图像。对所有算法取平均值后，

与 FixyNN 和 Darkroom 相比，我们分别将静态随机存取存储器的大小降低了 28.0% 和 10.2%。应用行合并优化后，相较于基准方法，静态随机存取存储器节省量分别提升至 86.0% 和 56.8%。

在多消费者算法中，静态随机存取存储器的节省量明显高于单消费者算法，这凸显了我们的框架在前者中的优势。对于多消费者算法，算法线性化会增加虚拟阶段，并增大行缓冲区。即使在单消费者算法中，FixyNN 的静态随机存取存储器需求也始终高于我们的方法，因为它只使用单端口存储块，不允许两个阶段重叠

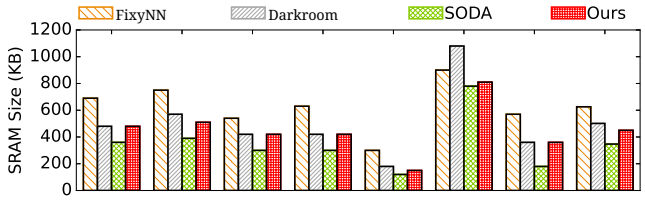


（a）静态随机存取存储器大小。

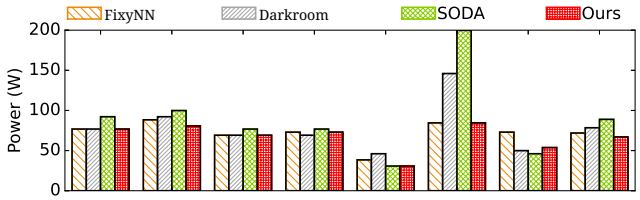


（b）功耗。

图 8：在 320p 图像上的静态随机存取存储器和功耗对比。



（a）静态随机存取存储器大小。



（b）功耗。

图 9：1080p图像上的静态随机存取存储器和功耗对比。请注意，未显示 Ours+LC，因为静态随机存取存储器容量不足以容纳多行。

。静态随机存取存储器的节省在 Xcorr-m 上尤为显著。这是因为对 Xcorr-m 进行线性化时，复制的阶段之一会处理一个较高的模板窗口（18×1）；复制该阶段会增加大量额外的静态随机存取存储器块。

Ours 的静态随机存取存储器需求 比 SODA 高 31.0%，因为 SODA 采用基于 FIFO 的方法，因此能够将行缓冲区中最后一行（即生产者正在写入的行）实现为 D触发器（图 4a）。通过行合并，Ours+LC 可将静态随机存取存储器需求降低 28.5%，相比 SODA 而言。

图 9a 显示， 1080p 输入的静态随机存取存储器节省趋势与 320p 输入相似，但行合并优化无法应用于 1080p 图像，因为静态随机存取存储器块的大小不足以容纳多于一行，正如第 7 节所述。

加速器结果。 如第 7 节所述，存储器面积占加速器面积的主要部分。因此，存储器大小的节省会转化为近似的加速器总面积节省。例如，与 FixyNN 和 Darkroom 相比，在 320p 图像上，我们的方案+LC 分别将总面积减少了 51.2% 和 41.9 %。在 1080p 图像上，节省量分别为 27.9% 和 12.9%。

现场可编程门阵列结果。 由于篇幅限制，我们总结现场可编程门阵列实现的主要结果。在 1080p 图像上，与 FixyNN 和 Darkroom 相比，我们分别将 BRAM 大小减少了 28.1% 和 10.2%；由于上述相同原因，与 SODA 相比，BRAM 使用量增加了 22.8%。在我们的现场可编程门阵列上，我们使用了 37.5% 的 BRAM 块，而 Darkroom 使用了 41.8%。

多种算法。 我们的目标不是 一种能够运行多种算法的通用模板加速器。相反，我们专注于针对特定算法进行专门设计的加速器。不过，通过减少存储器使用量，我们的编译器也能帮助只有一个存储器系统的通用模板加速器，使其能够同时容纳更多算法。例如，在拥有 120 个 BRAM 块的 FPGA 上，由于 BRAM 约束，FixyNN 和 Darkroom 即使在 320p 分辨率下也无法同时执行全部六种算法。然而，借助 Ours+LC，该 FPGA 仅使用 84 个 BRAM 块即可容纳全部六种算法。

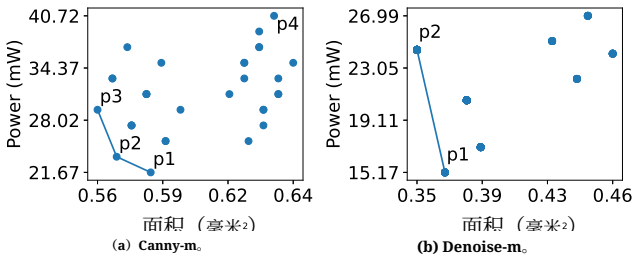
8.4 功耗降低

与所有基线方法相比，我们生成的加速器还具有更低的存储器功耗。图 8b 比较了在 320p 张图像上的功耗。平均而言，Ours相比 FixyNN、Darkroom 和 SODA 的功耗分别降低了 7.8%、13.8% 和 56.0%。行合并不会显著改变功耗，因为总访问次数基本保持不变。相比 Darkroom 和 FixyNN 的功耗节省源于静态随机存取存储器大小的减少。例如，FixyNN 仅使用单端口存储器，因此每次访问的功耗较低，但使用单端口存储器会导致静态随机存取存储器数量增加，从而提高总功耗。

值得注意的是，我们设计的功耗低于 SODA，尽管我们的设计所需的静态随机存取存储器阵列多于 SODA（图 8a）。这是因为 SODA 使用先进先出队列，每个周期都必须满足两次访问。我们的设计中，除一个静态随机存取存储器阵列外，其余阵列每周期只需满足一次访问，因此整体功耗更低。我们的设计+LC 相较于 SODA 的功耗优势，既源于减少了静态随机存取存储器需求，也源于避免使用功耗较高的先进先出队列。

图 9b 比较了使用 1080p 图像时的功耗。我们的设计的功耗分别比 FixyNN、Darkroom 和 SODA 低 7.8%、13.8% 和 56.0%。同样，尽管我们的设计使用的静态随机存取存储器多于 SODA，但由于避免了功耗较高的先进先出队列，因此功耗更低。

加速器结果。 存储器功耗节省转化为相近的加速器级节省。在 320p images 上，与 Darkroom 和 SODA 相比，我们的功耗分别降低了 11.7% 和 15.2%；在 1080p images 上，分别降低了 11.9% 和 18.3%。



(a) Canny-m。 (b) Denoise-m。

图 10：Canny-m和 Denoise-m在 320p下具有代表性的功耗-面积权衡。对于每种算法，我们扫描每个阶段的存储器配置，并生成相应的最优设计。每个阶段都可以使用双端口存储器（DP），或使用带行合并的 DP（DPLC）。帕累托最优设计因算法而异。

FPGA 结果。 FPGA 上的功耗节省趋势类似。在 1080p 输入上，我们的功耗分别比 FixyNN、Darkroom 和 SODA 低 19.7%、5.8% 和 17.7%。FPGA 的功耗节省较低，因为 FPGA 会消耗不可忽略的静态功耗。

8.5 设计空间探索

目前的评估假设所有算法的所有阶段都使用同一种行缓冲区，这是先前工作中唯一的选择 [7, 16, 38]。由于我们的框架允许指定任意存储器配置，因此可供专用集成电路设计人员用来创建定制的存储器模块，以探索针对特定算法的功耗与面积权衡。具体而言，我们允许将算法中的每个行缓冲区实现为双端口存储器（DP）或带行合并的双端口存储器（DPLC）。随后，我们会遍历每种算法所有可能的存储器配置组合，并为每种组合生成相应的最优设计。例如，如果某个算法包含四个阶段，我们最终会得到 16 种不同的设计。

我们观察到，帕累托最优设计会随算法而变化。以 320p 为示例，图10a 展示了Canny-m的功耗与面积对比，其中有三个帕累托最优设计。P1 对所有阶段均采用DP配置；在 P2 中，一个阶段采用 DPLC 配置；在 P3 中，两个阶段采用 DPLC 配置。相比之下，图10b 展示了Denoise-m所具有的另一种权衡模式，其中只有两个帕累托最优配置。在这种情况下，P1 对所有阶段仅采用 DP，而 P2 对所有阶段仅采用 DPLC。

我们特别指出，对于 Canny-m，所有阶段都使用 DPLC 的设计是图 10a中的 P4，远逊于 Canny-m的三个帕累托最优设计。DPLC 减少了 SRAM阵列的数量和总面积，但每次访问的功耗也随之增加。因此，总功耗取决于存储器访问总次数，而这必然因算法而异；我们的工具独有地支持了对此进行探索。

9 相关工作

敏捷加速器设计受到了广泛关注。近期的一个主题是弥合高级算法语义与硬件设计之间差距的语言 [20, 21, 26, 33]。这些语言支持对算法进行高级描述，并公开

将硬件视为一组基本组件。我们的工作聚焦于一种特定的算法领域（图像处理），以及硬件组件的一个特定方面：行缓冲片上存储器。

Dahlia [26]使用类型系统来拒绝那些在硬件中可能产生不可预测行为的程序——存储器争用就是其中之一。我们的工作将存储器端口争用编码为约束，并生成能够避免争用的（资源最优）硬件。Aetherling [8] 是一种领域特定语言，可将高级图像处理算法编译为硬件，重点在于探索资源与吞吐量权衡。它无法保证将存储器资源消耗降至最低，而我们可以做到这一点。HeteroHalide [22]和 HeteroCL [21] 可综合硬件加速器，并依靠 SODA [7] 生成片上存储器系统。HalideHLS [28]可为图像处理算法生成加速器，但依赖用户优化片上存储器。DSAGEN [36]使用编译指示注释算法，并自动为一系列算法搜索庞大的架构设计空间。

另一项彼此独立的工作是将算法映射到固定的硬件基底上并进行调度。先前工作采用约束优化方法 [11, 19, 27], , 主要针对深度学习工作负载。这些工作利用了这样一个事实：加速器设计空间通常可以参数化，而感兴趣算法的行为也可以通过机械方式建模；我们的工作同样利用了这两个特性。

10 结论与未来工作

本文提出了一个可自动合成图像处理加速器的框架。其关键在于一种优化公式化方法，能够将存储器争用表示为通用约束。显式的存储器约束优化使我们能够避免人工启发式方法，并针对特定应用定制设计。我们表明，与最先进方法相比，由该框架生成的加速器可分别将片上存储使用量和功耗最多降低 86.0% 和 62.9%。

我们以图像处理为例展示该框架，因为图像处理是自主机器等新兴应用的核心。不过，从根本上说，我们的框架并不限于图像处理，而是可推广至所有模板算法；模板算法是科学计算的核心，其中许多算法处理的是通用网格，而非图像 [18]。我们的主要技术创新，即以适用于数值优化的方式表达片上存储器争用，可推广至任何访问任意片上存储器的规则算法，而不仅限于行缓冲区。

未来工作的有趣方向包括自动综合稀疏图像处理加速器 [13, 23]，以及处理网格和点云等不规则视觉数据的加速器 [9, 10, 39]。这些应用领域中的加速器几乎完全由人工设计。

11 致谢

我们感谢 HPCA 2023 和 ISCA 2023 的匿名审稿人提出的宝贵意见。Jingwen Leng 和 Yuhao Zhu 为共同通讯作者。本研究部分获得了 NSF 资助，项目编号为 #2044963 和 #2126642。

参考文献

[1] [未注明日期]。FreePDK45。https://eda.ncsu.edu/freepdk/freepdk45/ .

[2] [无日期。]。Google OR-Tools。https://developers.google.com/optimization。[3] [无日期。]。Mali-C55。https://developer.arm.com/Processors/Mali-C55。[4] [无日期。]。借助 Qualcomm Spectra 图像信号处理器和 Snapdragon Sight，Snapdragon 为移动相机实现重大飞跃。https://futurumresearch.com/snapdragon-makes-significant-leap-for-mobile-cameras-with-qualcomm-spectra-image-signal-processor-and-snapdragon-sight/。[5] Daniele Bagni、Pari Kannan 和 Stephen Neuendorffer。2017 年。使用 Vivado HLS 揭秘 Lucas-Kanade 光流算法。技术说明 XAPP1300。赛灵思（2017）。[6] Nanchini Chandramoorthy、Giuseppe Tagliavini、Kevin Irick、Antonio Pullini、Siddharth Advani、Sulaiman Al Habsi、Matthew Cotter、John Sampson、Vijaykr- ishnan Narayanan 和 Luca Benini。2015 年。探索智能视觉系统中的体系结构异构性。载于 HPCA 会议论文集。[7] Yuze Chi、Jason Cong、Peng Wei 和 Peipei Zhou。2018 年。SODA：具有优化数据流体系结构的模板。载于 2018 IEEE/ACM 计算机辅助设计国际会议（ICCAD）。IEEE，第 1–8 页。[8] David Durst、Matthew Feldman、Dillon Huff、David Akeley、Ross Daly、Gilbert Louis Bernstein、Marco Patrignani、Kayvon Fatahalian 和 Pat Han- rahan。2020 年。流式加速器的类型导向调度。载于第 41 届 ACM SIGPLAN 编程语言设计与实现会议论文集。第 408–422 页。[9] Yu Feng、Gunnar Hammonds、Yiming Gan 和 Yuhao Zhu。2022 年。Crescent：驯服存储器不规则性，加速深度点云分析。载于第 49 届年度国际计算机体系结构研讨会论文集。第 962–977 页。[10] Yu Feng、Boyuan Tian、Tiancheng Xu、Paul Whatmough 和 Yuhao Zhu。2020 年。Mesorasi：通过延迟聚合为点云分析提供体系结构支持。载于 2020 年第 53 届 IEEE/ACM 年度微体系结构国际研讨会（MICRO）。IEEE，第 1037–1050 页。[11] Yu Feng、Paul Whatmough 和 Yuhao Zhu。2019 年。Asv：加速立体视觉系统。载于第 52 届 IEEE/ACM 年度微体系结构国际研讨会论文集。第 643–656 页。[12] Yiming Gan、Yu Bo、Boyuan Tian、Leimeng Xu、Wei Hu、Shaoshan Liu、Qiang Liu、Yanjun Zhang、Jie Tang 和 Yuhao Zhu。2021 年。Eudoxus：表征并加速自主机器中的定位——产业赛道论文。载于 2021 年 IEEE 高性能计算机体系结构国际研讨会（HPCA）。IEEE，第 827–840 页。[13] Cong Guo、Bo Yang Hsueh、Jingwen Leng、Yuxian Qiu、Yue Guan、Zehuan Wang、Xiaoying Jia、Xipeng Li、Minyi Guo 和 Yuhao Zhu。2020 年。无需硬件支持，通过分块稀疏性加速稀疏 DNN 模型。载于 SC20：高性能计算、网络、存储与分析国际会议。IEEE，第 1–15 页。[14] Matthew R. Guthaus、James E. Stine、Samira Ataei、Brian Chen、Bin Wu 和 Mehedi Sarwar。2016 年。OpenRAM：开源存储器编译器。载于 2016 年 IEEE/ACM 计算机辅助设计国际会议（ICCAD）。第 1–6 页。https://doi.org/10.1145/2966986.2980098[15] Samuel W Hasinoff、Dillon Sharlet、Ryan Geiss、Andrew Adams、Jonathan T Barron、Florian Kainz、Jiawen Chen 和 Marc Levoy。2016 年。用于移动相机高动态范围和低照度成像的连拍摄影。ACM 图形学汇刊（ToG）35，6（2016），第 1–12 页。[16] James Hegarty、John Brunhaver、Zachary DeVito、Jonathan Ragan-Kelley、Noy Cohen、Steven Bell、Artem Vasilyev、Mark Horowitz 和 Pat Hanrahan。2014 年。Darkroom：将高级图像处理代码编译为硬件流水线。ACM 图形学汇刊。33，4（2014），144–1。[17] James Hegarty、Ross Daly、Zachary DeVito、Jonathan Ragan-Kelley、Mark Horowitz 和 Pat Hanrahan。2016 年。Rigel：灵活的多速率图像处理硬件。载于 SIGGRAPH 会议论文集。[18] Justin Holewinski、Louis-Noël Pouchet 和 Ponnuswamy Sadayappan。2012 年。面向 GPU 体系结构的模板计算高性能代码生成。载于第 26 届 ACM 超级计算国际会议论文集。第 311–320 页。[19] Qijing Huang、Aravind Kalaiah、Minwoo Kang、James Demmel、Grace Dinh、John Wawrzynek、Thomas Norell 和 Yakun Sophia Shao。2021 年。Cosa：通过约束优化为片上空间加速器进行调度。载于 2021 年 ACM/IEEE 第 48 届年度计算机体系结构国际研讨会（ISCA）。IEEE，第 554–566 页。[20] David Koeplinger、Matthew Feldman、Raghu Prabhakar、Yaqi Zhang、Stefan Hadjis、Ruben Fiszel、Tian Zhao、Luigi Nardi、Ardavan Pedram、Christos Kozyrakis 等。2018 年。Spatial：一种用于应用加速器的语言和编译器。载于第 39 届 ACM SIGPLAN 编程语言设计与实现会议论文集。第 296–311 页。[21] Yi-Hsiang Lai、Yuze Chi、Yuwei Hu、Jie Wang、Cody Hao Yu、Yuan Zhou、Jason Cong 和 Zhiru Zhang。2019 年。HeteroCL：面向软件定义可重构计算的多范式编程基础设施。载于 2019 年 ACM/SIGDA 现场可编程门阵列国际研讨会论文集。第 242–251 页。

[22] Jiajie Li、Yuze Chi 和 Jason Cong。2020 年。HeteroHalide：从图像处理领域特定语言到高效现场可编程门阵列加速。载于 *2020 年 ACM/SIGDA 现场可编程门阵列国际研讨会论文集*。第 51–57 页。[23] Zhi-Gang Liu、Paul N Whatmough、Yuhao Zhu 和 Matthew Mattina。2022 年。S2ta：利用结构化稀疏性实现高能效移动 CNN 加速。载于 *2022 年 IEEE 高性能计算机体系结构国际研讨会（HPCA）*。IEEE，第 573–586 页。[24] Mostafa Mahmoud、Bojian Zheng、Alberto Delmás Lascorz、Felix Heide、Jonathan Assouline、Paul Boucher、Emmanuel Onzon 和 Andreas Moshovos。2017 年。IDEAL：图像去噪加速器。载于 *2017 年第 50 届 IEEE/ACM 年度微体系结构国际研讨会（MICRO）*。IEEE，第 82–95 页。[25] Sean Murray、William Floyd-Jones、Ying Qi、George Konidaris 和 Daniel J Sorin。2016 年。实时机器人运动规划加速器的微体系结构。载于 *第 49 届 IEEE/ACM 年度微体系结构国际研讨会*。IEEE Press，第 45 页。[26] Rachit Nigam、Sachille Atapattu、Samuel Thomas、Zhijing Li、Theodore Bauer、Yuwei Ye、Apurva Koti、Adrian Sampson 和 Zhiru Zhang。2020 年。利用时间敏感仿射类型实现可预测的加速器设计。载于 *第 41 届 ACM SIGPLAN 编程语言设计与实现会议论文集*。第 393–407 页。[27] Tony Nowatzki、Michael Sartin-Tarm、Lorenzo De Carli、Karthikeyan Sankar- alingam、Cristian Estan 和 Behnam Robatmili。2013 年。面向空间体系结构的通用约束中心调度框架。*ACM SIGPLAN 通知* 48，6（2013），第 495–506 页。[28] Jing Pu、Steven Bell、Xuan Yang、Jeff Setter、Stephen Richardson、Jonathan Ragan- Kelley 和 Mark Horowitz。2017 年。从图像处理领域特定语言编程异构系统。*ACM 体系结构与代码优化汇刊（TACO）* 14，3（2017），第 1–25 页。[29] Jonathan Ragan-Kelley、Connelly Barnes、Andrew Adams、Sylvain Paris、Frédo Durand 和 Saman Amarasinghe。2013 年。Halide：一种用于优化图像处理流水线中的并行性、局部性和重新计算的语言与编译器。*ACM SIGPLAN 通知* 48，6（2013），第 519–530 页。[30] Parthasarathy Ranganathan、Daniel Stodolsky、Jeff Calow、Jeremy Dorfman、Marisabel Guevara、Clinton Wills Smullen IV、Aki Kuusela、Raghu Balasubra- manian、Sandeep Bhatia、Prakash Chauhan 等。2021 年。仓库规模的视频加速：真实环境中的协同设计与部署。载于 *第 26 届 ACM 编程语言与系统体系结构支持国际会议论文集*。操作系统。600–615。[31] Jason Redgrave、Albert Meixner、Nathan Goulding-Hotta、Artem Vasilyev 和 Ofer Shacham。2018。Pixel Visual Core：Google 面向移动设备的全可编程图像、视觉和 AI 处理器。载于*IEEE Hot Chips 研讨会论文集 (HCS)*。1–18。[32] Jacob Sacks、Divya Mahajan、Richard C Lawson 和 Hadi Esmaeilzadeh。2018。Robox：加速机器人自主控制的端到端解决方案。载于第 *45 届年度国际计算机体系结构研讨会论文集*。IEEE Press，479–490。[33] Robert Stewart、Kirsty Duncan、Greg Michaelson、Paulo Garcia、Deepayan Bhowmik 和 Andrew Wallace。2018。RIPL：一种面向现场可编程门阵列的并行图像处理语言。*ACM 可重构技术与系统汇刊 (TRETS)* 11, 1 (2018)，1–24。[34] Amr Suleiman、Zhengdong Zhang、Luca Carlone、Sertac Karaman 和 Vivienne Sze。2019。Navion：用于纳米无人机自主导航的 2 毫瓦全集成实时视觉惯性里程计加速器。*IEEE 固态电路期刊*54, 4 (2019)，1106–1119。[35] Artem Vasilyev、Nikhil Bhagdikar、Ardavan Pedram、Stephen Richardson、Shahar Kvatinsky 和 Mark Horowitz。2016。面向成像和视觉应用的可编程体系结构评估。载于*MICRO 论文集*。[36] Jian Weng、Sihao Liu、Vidushi Dadu、Zhengrong Wang、Preyas Shah 和 Tony Nowatzki。2020。Dsagen：可编程空间加速器的综合。载于*2020 ACM/IEEE 第 47 届年度国际计算机体系结构研讨会 (ISCA)*。IEEE，268–281。[37] Neil HE Weste 和 David Harris。2015。*CMOS VLSI 设计：电路与系统视角*。Pearson Education India。[38] Paul N Whatmough、Chuteng Zhou、Patrick Hansen、Shreyas Kolala Venkatara- manaiah、Jae-sun Seo 和 Matthew Mattina。2019。Fixynn：通过迁移学习实现移动计算机视觉的高效硬件。*arXiv 预印本 arXiv:1902.11128* (2019)。[39] Tiancheng Xu、Boyuan Tian 和 Yuhao Zhu。2019。Tigris：面向点云 3D 感知的体系结构与算法。载于第 *52 届年度 IEEE/ACM 国际微体系结构研讨会论文集*。629–642。[40] Yuhao Zhu、Anand Samajdar、Matthew Mattina 和 Paul Whatmough。2018。Euphrates：面向低功耗移动连续视觉的算法与片上系统协同设计。载于第 *45 届 ACM/IEEE 年度国际计算机体系结构研讨会论文集*。