用于高效处理基于逻辑的神经网络的算法与硬件

Jingkai Hong*、Arash Fayyazi*、Amirhossein Esmaili、Mahdi Nazemi 和 Massoud Pedram，南加州大学电气与计算机工程系，美国加利福尼亚州洛杉矶{jingkaih,fayyazi,esmailid,nazemi,pedram}@usc.edu

arXiv:2304.06299v1 [cs.AR] 13 Apr 2023

摘要—近期，为提升神经网络（NN）加速器的性能以满足当今应用需求，人们付出了诸多努力，由此催生了一种依赖固定函数组合逻辑（FFCL）的基于逻辑的神经网络推理新趋势。本文提出了一种创新的优化方法，用于将采用 FFCL 的神经网络编译并映射到逻辑处理器中。所提出的方法将 FFCL 模块映射到一组布尔函数，其中每个函数中的布尔运算均映射到高性能、低延迟、并行化的处理单元。本文还提出了图划分与调度算法，以处理无法直接适配逻辑处理器的 FFCL 模块。我们在多个数据集和神经网络上进行的实验评估表明，与现有神经网络加速器相比，我们的框架在推理吞吐量方面性能更优。对于 VGG16 模型，与基于异或非门的加速器相比，我们实现了 25 倍的吞吐量提升；采用图划分与合并算法后，吞吐量还可提升 5 倍。

I. 引言

深度神经网络（DNN）在各种人工智能应用中提供最先进的性能，并在包括计算机视觉 [7], [21]与自然语言处理 [4], [6] 在内的许多具有挑战性的领域中超越了传统机器学习模型的准确率。诸如 *Transformer* [18] 和*MLPMixer*[15] 等更复杂的深度神经网络模型的出现，是深度神经网络在许多应用领域取得显著性能的关键原因，但这些模型对计算和存储器资源的需求极其庞大。

为提高深度神经网络（DNN）的效率，已有一些工作将神经网络高效处理的问题表述为布尔逻辑最小化问题，其中最终由逻辑表达式计算各种滤波器/神经元的输出。NullaNet [10] 针对给定数据集优化目标DNN，并将DNN计算中的关键部分映射到逻辑块，例如现场可编程门阵列上的查找表（LUT）。在大多数情况下，包括本研究所采用的情况，二值化实现的平均准确率下降小于4%。

问题在于，针对特定神经网络模型合成出目标固定功能组合逻辑（FFCL）后，合成得到的逻辑结构只能用于该神经网络模型的推理任务，这使得 FFCL 的 ASIC实现并不切实际。此外，由于为 SoA 神经网络设计的神经元包含数十到数百个输入，得到的布尔逻辑表达式可能非常庞大。我们的实验表明，某些生成的布尔函数无法映射到单个现场可编程门阵列上，因为查找表（LUT）已耗尽，无法容纳如此庞大的布尔函数。

由布尔逻辑单元组成的逻辑处理器，其中布尔逻辑单元负责执行每个布尔函数的布尔运算，这些布尔函数与从二值神经网络（BNN）中提取的 FFCL 模块相关联，如 [11], 中所述。这对于使用不同的 BNN 进行推理至关重要。因此，设计可配置的高效逻辑处理器作为基于逻辑的推理引擎，它可用于采用不同 DNN 模型的多种应用场景，因此极具价值。

从编译器设计的角度来看，将与布尔函数相关联的任意逻辑图编译并调度到要映射至逻辑处理器上的任务极具挑战性。编译器需要检测并分组所有能够同时执行的逻辑门操作，同时考虑硬件资源限制（即每个逻辑处理器中的布尔逻辑单元数量）。为应对这些挑战，我们设计了一种众核逻辑处理器，并提出了用于编译和映射利用 FFCL 的二值神经网络至该逻辑处理器的新技术。此外，我们还提出了一种原创的图划分算法，用于处理规模极大的逻辑图。本文的贡献如下：

- 我们介绍了一种逻辑处理器的设计，该处理器能够处理大型布尔函数。
- 我们提出了一种创新的优化方法，用于将利用 FFCL 的二值神经网络编译并映射到该逻辑处理器中。所提出的编译器会生成定制指令，以便在推理过程中对逻辑图中的所有操作进行静态调度。
- 我们提出了一种将 FFCL 模块映射为一组布尔函数的方法，其中每个函数中的布尔运算都映射到高性能、低延迟且可并行执行的处理单元。
- 我们提出了一种图划分算法，以高效处理超大型布尔逻辑图。
- 我们在多个数据集和神经网络上进行的实验评估表明，与现有神经网络加速器相比，我们的框架在推理吞吐量方面性能更优。对于 VGG16 模型，我们的吞吐量比基于异或非门的加速器高出 25 倍 [13] ；采用图划分与合并算法后，吞吐量还可再提升 5 倍。

虽然本研究主要关注如何通过算法在逻辑处理器架构上执行任意规模的 FFCL 模块，但我们也展示了在现场可编程门阵列上应用这些算法和硬件设计的结果，以评估其有效性。这些算法和硬件同样可用于专用集成电路设计。

II. 术语和符号

本节介绍本文中使用的术语和符号：

- **固定功能组合逻辑（FFCL）模块：**以硬件描述语言（如 Verilog）编写的组合逻辑电路网表。
- **逻辑处理单元（LPE）：**一种可编程硬件模块，可执行与、或、异或等（二输入）逻辑运算。

本研究部分由美国国家科学基金会软件与硬件基础项目的资助支持。
*Jingkai Hong 和 Arash Fayyazi 对本项工作贡献相同。

- **逻辑处理向量（LPV）：** 包含固定数量 LPE 的硬件模块。LPV 彼此之间按线性顺序排列。
- **逻辑处理单元（LPU）：** 一种包含固定数量 LPV 的硬件模块，在本文中也称为**逻辑处理器**。
- **最大可行子图（MFG）：** 从 FFCL 中贪心提取的有向无环图（其中节点为布尔运算，边缘端表示数据依赖关系），映射到 LPU 时不会超出其容量。
- **全路径平衡（FPB）：** 使从电路输入（即主输入）到电路输出（即主输出）的所有传播路径具有相同的逻辑深度。这可确保所有输入-输出路径上的逻辑门数量相同。

III. 所提设计流程

所提框架的整体流程如图1所示。该流程的输入是用 Verilog 语言描述的 FFCL 模块。请注意，该框架可以采用任何 FFCL 模块规范作为输入。Yosys[19] 和 ABC [3] 可用于根据任何行为规范生成可综合的 Verilog 代码。NullaNet [10]以 Verilog 格式生成 FFCL 模块，并将用作上游引擎。我们首先解析 Verilog 网表，使用标准逻辑优化技术综合电路，主要目标是减少电路的总门数量和逻辑深度，然后将电路映射到定制单元库。请注意，单元库中逻辑门支持的布尔运算（例如双输入与、或和异或运算）必须受到 LPE 支持。接下来，根据第 II 节中的定义对映射后的电路进行层级化。逻辑综合和层级化与文献 [12] 中介绍的方法相同。

由于目标电路中处于特定逻辑级别的逻辑门与同一逻辑级别的其他逻辑门没有连接，因此同一逻辑级别的所有逻辑门操作都可以同时执行。然而，由于硬件资源限制，即每个 LPU 中 LPV 的数量固定（深度问题），或每个 LPV 中 LPE 的数量固定（宽度问题），这些操作可能必须分配到不同的计算周期。对于较大的图，可以将多个 LPU 以并联或串联配置组合起来，以完成给定逻辑图所需的计算，但会增加面积和功耗成本。我们的编译器能够将任意逻辑图映射到任意规模的 LPU。为处理宽度问题，

编译器将滤波器/神经元函数分解为多个 MFG，每个 MFG 随后依次映射到 LPU 上（更多详情见第 V 节）。

IV. 逻辑处理器架构

逻辑处理器的架构如图 2 所示。逻辑处理器是一种数据驱动架构，因为

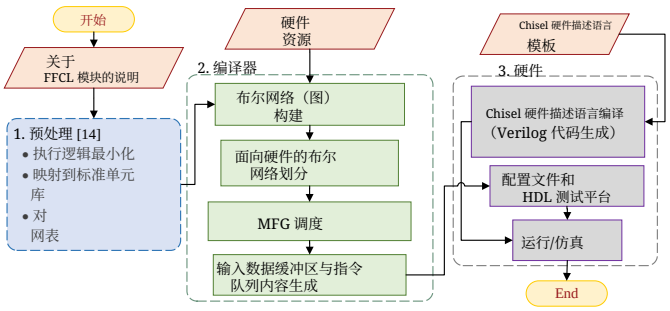


图1：所提出框架的概览。

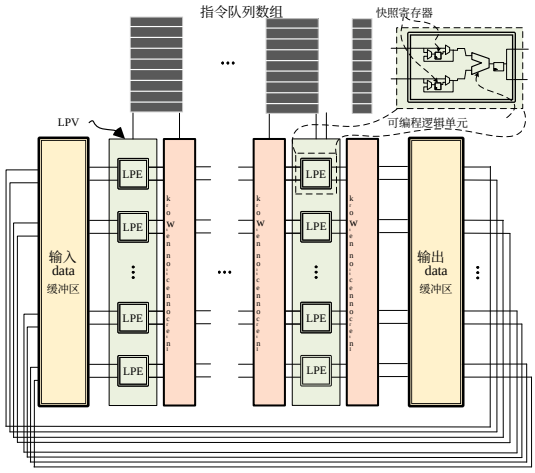


图2：LPU 架构。

流式数据由多个 LPV 阶段接收并处理，无需将任何中间结果存储在暂存存储器中。更准确地说，逻辑处理器由一组按线性顺序排列的 LPV 组成。每个 LPV 包含 m 个 LPE，每个 LPE 接收两个输入并产生一个输出，类似于逻辑门。因此，每个 LPV 最多接收 $2m$ 个输入操作数，并产生一个最多包含 m 个输出结果的向量。为支持逻辑运算打包并提高硬件效率，每个操作数的位宽为 $2m$ 位，相当于 $2m$ 个布尔变量或 m 4 值逻辑变量等。在处理从卷积神经网络（CNN）模型中提取的 FFCL 块时， $2m$ 位数据来自 CNN 输入特征体中的不同图像块，或来自用于批量推理任务的不同图像。为将数据从第 i 个 LPV 传递到第 $i+1$ 个 LPV，我们使用一种非阻塞式多播多级交换网络。

每个 LPV 中的 LPE 数量以及每个 LPU 中的 LPV 数量，决定了 LPU 能够处理的逻辑图规模。根据前文所述的参数值，LPU 能够处理的逻辑图最大宽度为 m，最大深度为 n。其中，宽度指图中任意逻辑深度上的逻辑运算数量，深度则指从任意图输入到任意图输出的逻辑深度。

由于逻辑图中相邻两个逻辑层级之间存在数据依赖，直观的图计算方式是逐层进行。然而，考虑到硬件预算紧张且需要较高吞吐量，使用暂存存储器保存两个逻辑层级之间的临时结果（即已生成但尚未被使用的结果）并非理想选择。因此，我们提出采用分布式快照寄存器 来存储临时结果，如图2所示。此外，我们还提出了逐 MFG 的计算范式和调度算法，详见第 V 节。该范式进一步利用了设置在每个 LPE 中的快照寄存器 所带来的局部性。

每个 LPE 都包含一个逻辑单元，可在其中执行基本布尔运算；还包含两个快照寄存器，可根据编译器确定的数据生命周期，暂时存储 LPE 的每个输入。此外，借助指令集配置分配给每个 LPE 的运算。请注意，可执行两种类型的逻辑运算：多输入单输出（MISO）运算，包括与、或、异或/同或；以及单输入单输出（SISO）运算，包括非/缓冲。缓冲表示向有向无环图（DAG）中添加一个节点，使任意两个

相连节点之间的所有路径具有相同的拓扑长度（即，连接两个节点的所有路径上都包含相同数量的逻辑门）。缓冲器插入是全路径平衡步骤的一部分。全路径平衡可确保不同逻辑层级的逻辑门之间不存在数据依赖，从而简化逻辑图到我们流水线架构上的映射。

V. 编译器

本文介绍了一种专为逻辑处理器设计的编译器。该编译器解析门级 Verilog 网表，提取电路网表中每个逻辑层级执行的操作集合，创建有向无环图来表示这些门操作及其方向性数据依赖，然后将该有向无环图划分为多个 MFG，使每个 MFG 都能装入给定的 LPU。随后，编译器对生成的 MFG 进行调度，并将其映射到不同计算周期中的不同 LPV。这样一来，LPU 就能在每个 MFG 内逐层处理，并以逐个 MFG 的方式执行大型逻辑图。

A. 布尔网络划分

考虑一个子图 $H=(V',E')$，它属于一个完全路径平衡的布尔有向无环图 $G=(V,E)$。子图 H 中节点的逻辑层级从 $L_{bottom}(H)$ 到 $L_{top}(H)$。现在，给定一个划分方案 P，该方案针对图 G，使得 P 由 $|P|$ 个子图组成，我们有：

$$\forall H \in P, \quad 0 \le L_{bottom}(H) \le L_{top}(H) \le L_{max}$$

其中，0 和 L_{max} 分别表示 G的主输入和主输出所在的层级。我们将逻辑层级 l 中的节点集记为 $node\ set(l)$。此外，给定一个节点集 S，令 $input(S)$表示输入到节点集 S的不同节点所构成的集合。

MFG 必须满足以下条件。首先，相应的子图 H 必须满足：

$$\forall l \in [L_{bottom}(H)+1, L_{top}(H)] \implies input(node_set(l)) \in H \tag{1}$$

这意味着，子图中除最底层以外所有层级的输入也必须包含在同一子图中。换言之，来自子图外部节点、指向子图内部节点的入向连接，只允许出现在子图的最底层。其次，子图在其每一层中至多包含 m个节点（请记住，所提出的逻辑处理器中的每个 LPV 都有 m 个 LPE）：

$$\forall l \in [L_{bottom}(H), L_{top}(H)], |node_set(l)| \le m \tag{2}$$

请注意，节点不必只属于某一个子图，即 MFG 可以具有重叠的节点集。更准确地说，

$$\exists l \in H, \exists l' \in H', node_set(l) \cap node_set(l') \neq \emptyset \tag{3}$$

最后，MFG 最底层的节点，其输入节点数必须大于 m，但输入为 G的 PI 的子图除外：

$$\forall H \in P \setminus \{H_0 | L_{bottom}(H_0)=0\}, |input(node_set(L_{bottom}(H)))| > m \tag{4}$$

其中，$\setminus$ 表示集合差运算。请注意，子图 H 最底层的输入不包含在 H中。该方程是算法 2 中 while 循环的跳出条件。

布尔网络划分伪代码如算法 1 所示。该算法从给定图的主输出（PO）开始进行广度优先搜索遍历，以找到所有 MFG。以主输出（PO）为根的 MFG 通过使用 $findMFG()$ 过程获得，具体如算法 2 所述。算法 1 随后继续查找以刚提取的 MFG 的输入节点为根的 MFG。遍历持续进行，直到到达布尔网络的 PI。构造以某个节点 V 为根的 MFG 时，会不断向 MFG 中添加节点（同样采用广度优先搜索遍历的方式），直至到达其传递扇入锥中的某个逻辑层，该层的节点数超过 m，此层称为停止层。请注意，如算法 2 所示且图 3 中也有所展示，以 V 为根的 MFG 不包含停止层节点。由于违反条件 1，MFG 不能包含停止层节点的任何子集。

算法 1 返回一个名为 allTempMFGs 的 MFG 集合，其中包含顶层只有一个节点的 MFG，如图 3 所示。BNN 推理任务的运行时间主要受 MFG 总数影响。因此，我们提出了一种贪心合并算法（见算法 3），将输入同一个 MFG 且底层相同的一组单输出 MFG 合并，生成一个多输出 MFG（参见图 3）。checkLevel 函数用于检查两个 MFG 在每个逻辑层级上的节点是否满足约束 $|\text{nodes(MFG1, } l_i) \cup \text{nodes(MFG2, } l_i)| \le m$。请注意，由于违反了条件 1 中所述的性质，不能合并底层不同的两个 MFG。接下来，我们将讨论编译器设计中面临的主要挑战。

B. 解决宽度问题

对于 NullaNet [10], 生成的大多数布尔网络，映射后的网表规模超过 n 个逻辑层级，且每个逻辑层级包含的节点远多于 m 个。我们将前者称为深度问题，后者称为宽度问题，MFG-by-MFG 计算范式通过以下方式解决这一问题。

借助快照寄存器存储生成的中间结果，MFG 可以按顺序调度和处理。每个 MFG 的 $node\ set(L_{top})$ 产生。在流水线方式下，每个 MFG 的计算需要 $L_{top}-L_{bottom}+1$ 个 LPV。具体而言，分配给 MFG H 的计算资源为 $LPV \in [L_{bottom}, L_{top}]$。计算周期数为 $(L_{top}-L_{bottom}+1)\times t_c$ 个时钟周期，其中 t_c 是 LPE 内部计算所需的一个周期与在已部署的交换网络中进行数据路由（转向）所需的 t_{sw} 个周期之和。请注意，对于不同的二值神经网络推理任务，t_c 是固定的。在本文中，$t_c=6$，因为 $t_{sw}=5$，其中使用了一个 5 级无阻塞多播交换网络 [20]。与 MFG 的待路由 $L_{top}+1$（例如 H_i）相关联的 LPV 快照寄存器存储给定 MFG 生成的中间结果。父 MFG（即 H_j，其输入来自 H_i 的 MFG）使用 快照寄存器中存储的数据，于 $L_{top}+1$ 开始计算。

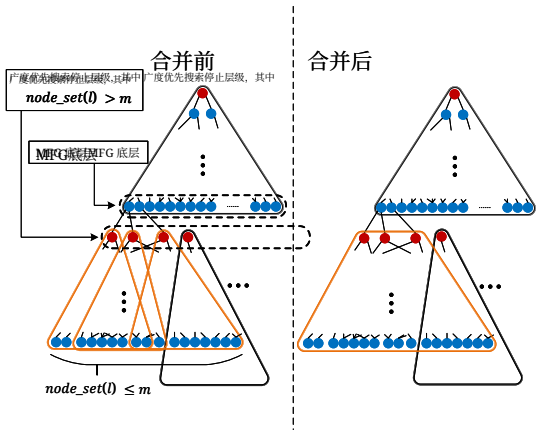


图 3：合并过程示例。

算法 1 单输出布尔网络划分

输入：布尔网络的 PO，m每个 LPV 的 LPE 数量
输出：覆盖布尔网络的一组 MFG

```
allTempMFGs = []                                  // 所有 MFG 的集合
MFG=findMFG(PO,m)                                 // 调用算法 2
队列 = []
queue.append(MFG)
当队列不为空 执行
    当前MFG = queue.pop(0)                        // 弹出第一个元素
    allTempMFGs.append(curMFG)
    对于inputNode in input(curMFG) 执行
        childMFG=findMFG(inputNode,m)             // 调用算法 2
        queue.append(childMFG)
返回allTempMFGs
```

算法 2 函数*findMFG()*

输入：节点 V, m 每个 LPV 的 LPE 数量
输出：一个维护以 V为根的 MFG 的图对象

```
MFG =图（根=V）                                   // 使用根=V 初始化图
队列 = []
queue.append(V)
visited =emptyset()
visited.add(V)
当queue 不为空 时
    curNode = 队列.pop(0)
    MFG.节点计数[当前节点.层级]+=1
    visited.add(curNode)
    如果 |MFG.nodeCount[curNode.level| ≥ m则
        stopLevel = curNode.level
        跳出
    对于 child in curNode.children执行
        如果 child not in visited则
            visited.add(child)
            queue.append(child)
MFG.bottomLevel = stopLevel + 1
返回MFG
```

算法 3 MFG合并算法

输入：一组 MFG
输出：精简后的 MFG 集合

```
所有合并后的MFG = []
根 MFG =包含 PO 的 MFG
队列 = []
queue.append(topMFG)
当队列不为空时执行
    当前MFG = 队列.pop(0)                         // 弹出第一个元素
    所有合并后的MFG.append(当前MFG)
    对于所有MFG1 与 MFG2，位于当前MFG.子节点中 执行
        合并 = False
        如果MFG1.bottomLevel == MFG2.bottomLevel则
            合并 = 检查层级(MFG1, MFG2)
        如果 合并，则
            合并后的子MFG = MFG1 ∪ MFG2
            队列.append(mergedChildMFG)
            对所有MFG1 和 MFG2 的子节点中的孙节点 执行
                将孙节点.parent 更新为 mergedChildMFG
        else
            queue.append(MFG1, MFG2)
return所有合并后的MFG
```

图 4 展示了一个图划分方案示例。在此示例中，一个布尔网络被划分为 10 个 MFG，并以不同颜色显示；每个 MFG 均按照图 5 的时空图所示进行调度和处理。不同逻辑层级的计算周期显示在 MFG 名称之后的 l_i 中。例如，$A1$ 表示 MFG A 中逻辑层级 1 的所有节点的计算，耗时 1 个周期。它们还与另外两个变量相关联：计算周期（即 C j）和 LPV（LPVk），用于表示 MFG 的第 l_i 个逻辑层级的计算在时钟周期 j 由 LPV k 执行。

在 LPU 中，一个 LPV 阶段和后续交换

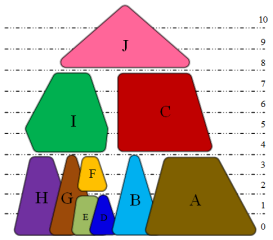


图 4：一个分区示例，其中 $L_{max} = 1$ 0.

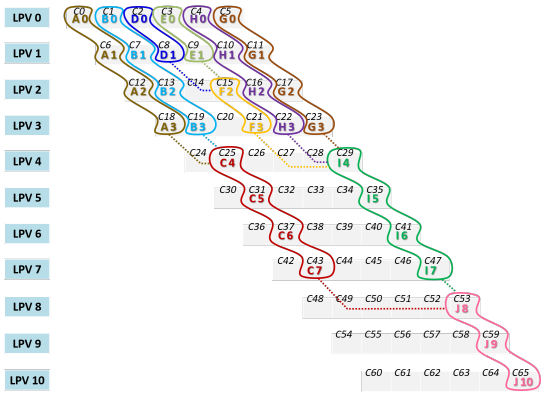


图 5：调度示例。（注：$C0$ 表示时钟周期 0）

网络的 5 个阶段构成一个由 6 个指令队列组成的区块，其中每个存储器每个周期都会从其前驱存储器获取读地址。可通过读地址移位寄存器访问指令队列。对应于某个 MFG 的指令会写入所有参与该 MFG 计算的 LPV 所关联的指令队列中的同一地址。例如，用于计算 MFG J 的指令（在图 5 中以粉色突出显示）会写入图 6 所示、与 LPV #8、9、10 关联的指令队列中的内存位置5。

在图 6 中，展示了对所有 LPV 进行编程的指令队列的存储位置。存储位置的颜色对应于 MFG 的颜色（参见图 4）。所有带有 $L_{bottom} = 0$ 的 MFG 都从输入数据缓冲区接收 $node\ set$（L_{bottom}）所需的 PI 值。编译器使用计数器，确保所需的 PI 值正确存储在输入数据缓冲区的不同位置，从而每个周期都能正确访问所需的数据。与随机访问寻址系统相比，此方案简化了地址生成。其他带有 L_{bottom} ≠0 的 MFG 的输入由至少两个更早计算的子 MFG 提供。因此，对 MFG 进行调度等同于安排指令并将它们放置在适当的存储位置。

算法 4 所示的调度算法用于确定写入每个 MFG 指令的存储位置。如图 5 所示，每个满足 L_{bottom} ≠0 的 MFG 至少有两个子 MFG，支持其 $node\ set(L_{bottom})$ 中的所有节点。对于 MFG H，我们将其子 MFG 称为 $H_c = \{H_{c1}, H_{c2}, ..., H_{cn}\}$。$H_c$ 中的每个 MFG 都会生成 $input(node\ set(L_{bottom}(H)))$的一个子集；因此，我们有 $\bigcup_{i=1}^{n} node\ set(L_{top}(H_{ci})) = input(node\ set(L_{bottom}(H)))$。我们将最后调度、且生成父节点输入节点子集（无需将其存储在快照寄存器中）的子 MFG 称为最近子节点。例如，MFG J的最近子节点是 MFG I（见图 5）。请注意，只要两个 MFG 中的一个是另一个的最近子节点，它们就可以使用相同的 memLoc 值（例如，MFG I, J），因为它们在不同的 LPV 上执行。因此，所需的指令队列大小得以缩减。

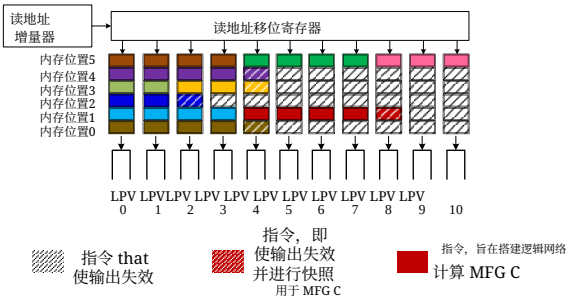


图 6：指令队列配置。

算法 4 MFG 调度算法

输入：一组已路由的 MFG
输出：指令写入的内存位置

```
内存位置 = INT_MAX
topMFG.内存位置() = MemLoc
栈 = []
栈.append(topMFG)
当栈非空 时
    当前MFG =栈.pop()
    当前MFG.memLocation() = 内存位置
    如果 当前MFG.parents ≠ None 那么
        对子MFG 执行 curMFG.children() 操作
            栈.append(子MFG)
    else
        内存位置 = 内存位置 - 1
    对所有MFG 执行
        MFG.内存位置() = MFG.内存位置() -
        内存位置
```

C. 解决深度问题

深度问题交由 LPU 硬件处理。如果某个 MFG 的L_{top} 大于固定大小 LPU 的 LPV 总数，输出数据缓冲区将充当 LPV $L_{top}+1$的快照寄存器，而该 LPV 实际上并不存在。此时，LPV 0 会采用循环机制来实现 LPV $L_{top}+1$的功能。编译器负责检测潜在的深度问题并相应地重新定位指令。编译器会通知硬件何时将存储在输出数据缓冲区中的中间数据送回 LPV 0。此类数据会经过流水线进行多轮计算，以完成给定 FFCL 的所有逻辑层级的计算。

VI. 实验结果

为进行评估，我们选择了一款高端 Virtex® Ultra-Scale+ 现场可编程门阵列（Xilinx VU9P FPGA，该器件可通过 AWS EC2 F1 实例在云端使用）。由于 SoA 实现采用相同的 FPGA，我们也纳入了 FPGA 原型测试结果。LPV 数量 = 16的硬件指标列于表 I。

我们基准测试的模型可分为两组：i）对准确率要求较高的模型（即大型模型），以及 ii）对吞吐量要求较高的模型（即微型模型）。在第一组中，我们研究了 VGG-16、用于 MNIST 数据集的 ChewBaccaNN[2], LENET5 中采用的 VGG 类模型，以及用于 CIFAR-10 数据集的 MLPMixer [15]。我们还针对物理学和网络安全领域对吞吐量要求极高的任务评估了我们的逻辑处理器，例如喷注子结构分类（JSC） [5] 和网络入侵检测（NID） [9]。我们使用 UNSWNB15 数据集，将我们的逻辑处理器与其他实现进行比较。我们采用了与 Murovic 等人 [9] 相同的预处理训练数据和测试数据，其中包含 593 个二元特征，这些特征对应于 49 个原始特征和两个输出类别。

对于 MLPMixer，输入图像的分辨率为 32*32，实验采用的图像块大小为 4*4。因此，我们有 64 个互不重叠的图像块，它们被映射到一个隐藏维度 C，小型设计（S）和基础设计（B）的该维度分别为 128 和 192。 D_S 和 D_C 分别是令牌混合 MLP 和通道混合 MLP 中可调的隐藏宽度。对于 S（B）设计， D_S 和D_C 分别为 64（96）和 512（768）。S 和 B 设计分别包含 8 层和 12 层混合层。

A. MFG 合并过程的效果

为展示第 V 节所述合并过程的有效性，我们比较了采用和未采用该合并过程时逻辑处理器的性能。结果如图7和图8所示。图7a展示了采用和未采用合并过程时，计算 VGG16 的层 [2:13] 所需的时钟周期数。图7b展示了采用和未采用合并过程时，通过所提出的算法获得的 MFG 总数。结果证实了应用合并过程后性能更优，也表明计算时间与 MFG数量之间高度相关。为进一步评估所提出合并算法的有效性，我们将其应用于本研究使用的所有模型，并在图8中汇总结果。如图所示，平均吞吐量提升了 5.2 倍，而 MFG数量最多可减少 9.4 倍。

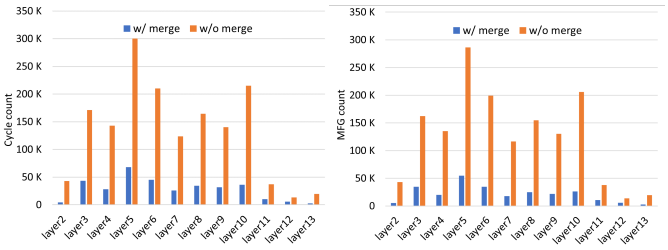
B. 基于 MAC、同或和 NullaNet 的神经网络现场可编程门阵列实现比较

VGG16 是一个庞大的网络，包含约 1.38 亿个参数。我们使用所提出的框架和固定功能组合逻辑函数，实现了 VGG16 中第 2 至第 13 个中间卷积层。作为采用传统 MAC 计算实现各层的 SoA 通用 MAC 阵列加速器基线，我们使用了 [14] 的开源实现，并加入了 [12] 中提出的一些改进。对于基于同或的基线，我们使用 FINN [16]，并通过打包操作改进了该实现。我们还使用 NullaDSP 模型 [12] 作为另一种基线，将 NullaNet 生成的 FFCL 映射到可编程 DSP 模块；该模型可适配任意大小的 FFCL。我们采用 [12] 中报告的各实现最佳结果，并在表 II 和表 III 中将其与所提出的架构进行比较。对于 JSC 和 NID，我们采用 LogicNets [17]，中报告的实现及其相应性能、Google 和欧洲核子研究中心的优化实现 [8], ，以及 [1] 中介绍的实现。

如表 II 所示，与其他实现相比，我们的实现表现更优。LPU

表 I：LPV 数量 = 16 的设计资源利用率。

FF(%)	LUT（%）	BRAM（%）	FREQ
478K(20.2%)	433K（36.7%）	12240K (15.8%)	333兆赫兹



(a) 计算时间　　(b) MFG数量

图7：a) 应用合并算法前后 VGG16 各层的计算时间；b) MFG数量。

与同或实现节省显著，尤其是在 VGG16 这类大型 DNN 模型上，因为我们将所有中间数据都保存在片上存储器中，无需承担片外存储器带来的开销，而基于 MAC 和 NullaDSP 的实现则并非如此。与基于 MAC、NullaDSP 和同或的实现相比，我们在 VGG16（LENET-5）推理性能上分别提升了 14.01 倍（33.43 倍）、4.86 倍（3.93 倍）和 1.95 倍（4.89 倍）。

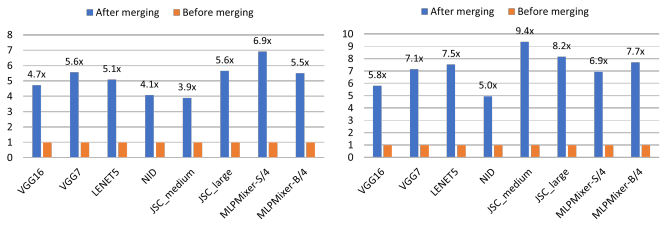
LogicNets [17] 的每秒帧数（FPS）高于我们的设计。然而，由于 LogicNets 将每个模型实现为定制的硬件逻辑网络（如随机逻辑块），因此无法让其他模型使用相同的硬件。相比之下，我们的设计提供了可编程逻辑处理器，能够执行任意逻辑（计算）图所需的逻辑门操作。前一种实现方式适合构建高效但无法更改的推理引擎，而后一种方式则适合加速训练过程，并构建部署到实际环境后仍可更新的推理引擎。请注意，NullaNet Tiny [11] 是我们生成 FFCL 逻辑块的上游模型，其实现方式与 LogicNets [17] 类似，并且在相同基准测试的类似设置下，性能优于 LogicNets。

C. LPV 数量消融研究

为确定 LPV 数量对所提出的逻辑处理器性能的影响，我们针对不同的 LPV 数量进行了实验。如图 9 所示，随着 LPV 数量增加，推理时间会缩短。LPV 数量的影响在一段时间后趋于饱和。为比较 NullaDSP [12] 与 LPU 的性能，我们对有效 LPV 阈值进行了基准测试。该阈值定义为 LPU 达到与 NullaDSP 相当的性能所需的最少 LPV 数量。如图 9 所示，对于 VGG16，至少需要 2 个 LPV 才能达到这一性能。

七、结论

所提出的逻辑处理器提供了一种新颖的硬件设计方法，这得益于本文介绍的编译器，能够将给定的神经网络模型高效地划分并映射为固定功能组合逻辑。未来，我们计划探索异构架构，其中每个 LPV 上的 LPE 数量



(a) 吞吐量　　(b) MFG数量

图 8：应用合并算法前后的 a) 吞吐量和 b) MFG数量。

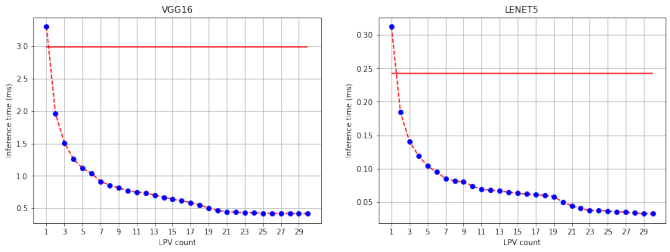


图 9：VGG16 和 LENET5 的推理时间。

表 II：对高精度要求模型的不同实现方式进行的每秒帧数比较。LPU 中的 LPV 数量为 16。

	MAC	NullaDSP	XNOR	LPU
VGG16	0.12K	0.33K	0.83K	**103.99K**
LENET5	0.48K	4.12K	3.31K	**1035.60K**
MLPMixer-S/4	4.17K	-	50.00K	**179.23K**
MLPMixer-B/4	0.88K	-	16.67K	**102.01K**

表 III：高吞吐量要求模型的不同实现方式之间的每秒帧数比较。LPU 中的 LPV 数量为 16。

	LogicNets [17]	Google+欧洲核子研究中心 [8]	[1]	LPU
NID	**95.24M**	-	49.58M	8.39M
JSC-M	**2995.00M**	-	-	0.69M
JSC-L	**76.92M**	76.92M	-	0.21M

以及其后续交换网络的数量不会在所有 LPV 上都相同。此外，也值得尝试将多个 LPU 以并联或串联配置进行组装。

参考文献

[1] S. A. Alam 等，《FINN 矩阵向量计算单元的 RTL 实现》，CoRR，第 abs/2201.11409 卷，2022 年。[2] R. Andri 等，《Chewbaccann：一款灵活的 223 TOPS/W 二值神经网络加速器》，载于 IEEE 国际电路与系统研讨会，2021 年。[3] R. K. Brayton 和 A. Mishchenko，《ABC：一种学术界与工业界通用的验证工具》，载于 CAV，第 6174 卷。施普林格，2010 年。[4] J. Devlin 等，《BERT：用于语言理解的深度双向变换器预训练》，载于北美计算语言学协会人类语言技术分会年会。计算语言学协会，2019 年。[5] J. M. Duarte 等，《用于粒子物理学的现场可编程门阵列深度神经网络快速推理》，CoRR，第 abs/1804.06913 卷，2018 年。[6] S. Hochreiter 和 J. Schmidhuber，《长短期记忆》，《神经计算》，1997 年。[7] G. Huang 等，《密集连接卷积网络》，载于计算机视觉与模式识别会议，2017 年。[8] C. J. N. C. Jr. 等，《面向粒子探测器边缘端低延迟推理的深度神经网络自动异构量化》，《自然·机器智能》，2021 年。[9] T. Murovic 和 A. Trost，《面向边缘端现场可编程门阵列系统中船舶检测的大规模并行二值神经网络推理》，载于第24届IEEE设计与诊断研讨会，2021 年。[10] M. Nazemi 等，《通过布尔逻辑最小化实现节能、低延迟的神经网络》，载于第24届ACM亚洲及南太平洋设计自动化会议，2019 年。[11] M. Nazemi 等，《Nullanet tiny：通过固定功能组合逻辑实现超低延迟深度神经网络推理》，载于第29届IEEE现场可编程定制计算机研讨会，2021 年。[12] S. N. Shahsavani 等，《固定功能组合逻辑在面向神经网络推理的数字信号处理器上的高效编译与映射，并利用高级综合》，《ACM可重构技术与系统汇刊》，2022 年。[13] K. Simonyan 和 A. 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。[14] A. Sohrabizadeh 等，《深度学习应用的端到端优化》，载于 2020 年 ACM/SIGDA现场可编程门阵列会议，2020 年。[15] I. O. Tolstikhin 等，《MLP-Mixer：一种用于视觉的全 MLP 架构》，载于第34届神经信息处理系统大会，2021 年。[16] Y. Umuroglu 等，《FINN：一种用于快速、可扩展二值神经网络推理的框架》，载于 2017 年 ACM/SIGDA现场可编程门阵列会议，2017 年。[17] Y. Umuroglu 等，《Logicnets：面向超高吞吐量应用的神经网络与电路协同设计》，载于 2020 年 IEEE第30届现场可编程逻辑国际会议，2020 年。[18] A. Vaswani 等，《注意力机制就是你所需要的一切》，CoRR，第 abs/1706.03762 卷，2017 年。[19] C. Wolf，《Yosys 开放式综合套件》，2016 年。[在线]。网址：http://www.clifford.at/yosys/ [20] Y. Yang 和 G. M. Masson，《非阻塞广播交换网络》，《IEEE计算机汇刊》，第40卷，第9期，1991 年。[21] S. Zagoruyko 和 N. Komodakis，《宽残差网络》，载于《2016年英国机器视觉会议论文集》。BMVA Press，2016 年。