Pudding：面向使用处理器内存的DRAM系统的可重定向编译器

Tianrui Zheng*†
zhengtr1998@gmail.com达姆施塔特工业大学，华为希尔伯特研究中心，德国

JoãoPaulo Cardoso de Lima*†
joao.lima@fau.de埃朗根-纽伦堡弗里德里希-亚历山大大学，德国

Alexander K u snezoffalexander.kusnezoff@mailbox.tu-dresden.de德累斯顿工业大学，德国

Asif AliKhan†
asif.khan@uni-paderborn.de帕德博恩大学，德国

Jeronimo Castrillonjeronimo.castrillon@tu-dresden.de德累斯顿工业大学，ScaDS.AI，德国

arXiv:2611.30446v1 [cs.AR] 23 Nov 2026

摘要

使用处理器内存的DRAM（PUD）架构利用 DRAM 阵列的物理特性，直接在内存中执行布尔运算，从而消除内存密集型工作负载中的数据移动瓶颈。近期提出了多种 PUD 架构，它们的原语操作集、行分配模型和执行指标各不相同。如今，这些架构差异都被硬编码在新兴的 PUD 感知编译器中，使得跨架构复用成为不可能。我们提出了一种面向 PUD 架构的可重定向端到端编译框架。其核心思想是将架构描述作为编译流水线中的一等对象：要重定向到新的 PUD 架构，只需提供其描述。我们的编译器构建于 MLIR 框架之上，在两个相互关联的层级上运行。在张量层级，它通过位切片、混合精度分配以及基于整数线性规划（ILP）的放置来降低 PUD 命令数量和存储体间的数据移动。在逻辑层，它采用等式饱和来探索布尔网络实现，并直接依据架构描述推导重写规则和代价函数。我们使用相同的编译器遍，在多个 PUD 目标平台上对代表性工作负载进行评估。结果表明，该框架的表现可与架构特定的手工优化编译器媲美（在若干情况下甚至更优），同时具备可重定向性和可扩展性。

关键词

使用DRAM进行处理、编译器、MLIR、等式饱和、整数线性规划

1 引言

近期的实验工作证实，商用现成（COTS）DDR4芯片已经支持一组功能完备的布尔原语（即非以及多输入与、或和多数），无需进行硬件修改 [15, 16, 20, 24, 52, 53]。业界也已开始发表 PUD 相关成果，例如 SK海力士 [20, 24]。其他设计则提出对动态随机存取存储器阵列进行小幅修改，以支持更丰富的原语集，并实现更高效的存内计算 [3, 10, 12, 19, 31, 37, 38, 43, 44, 55]。

然而，每种设计都有各自的映射约束，对代码生成提出了截然不同的要求。例如，SIMDRAM 类目标平台 [19]将 MAJ3、非和 RowCopy 作为其原生原语，并在同一子阵列中将计算行与数据行分开。这种目标平台需要基于这些假设构建逻辑优化器和行分配器。相比之下，商用现成 DDR4 目标平台 [52, 53] 提供了更丰富的原语集，且每种原语都有特定的行使用规则。因此，同一程序在这两个目标平台上需要采用本质不同的实现，而目前每个目标平台的映射都是单独推导的 [17, 40]。因此，将应用移植到新的目标平台意味着需要重新进行映射；若要在同一应用上比较不同目标平台，也必须针对每个目标平台重复这一过程。

在编译器能够以仍在演进的 PUD 设计为目标之前，无法在完整应用上对其进行评估；数字存内计算领域也观察到了这一点 [36]。编译器技术往往先于其目标硬件开发，例如，面向 VLIW 机器的轨迹调度 [14], 、量子程序编译器 [22], 以及赛道存储器的数据放置 [26]。将高级程序映射到 PUD架构，需要在两个层级上解决优化问题。在张量层级，编译器必须确定操作的顺序和融合方式、张量如何分块并分布到各存储体和子阵列中，以及每个算子所需的位宽。位宽优化尤为关键，因为针对某项操作发出的 PUD命令数量与操作数位宽直接成正比。在逻辑层，编译器必须将每项算术运算分解为一系列特定于目标平台的原子操作（原语），将操作数位映射到存储行，并生成能够实现该计算的最优命令序列。一个关键观察是，这两个层级之间存在依赖关系，而这种依赖在传统编译中并不存在。例如，在张量层级做出的位宽决策，会决定布尔网络的行分配

逻辑优化器必须处理的内容，以及逻辑层级的布局约束，会反馈到张量层级中张量的划分方式。目标架构的变化（例如原语集、行预算和行译码方案）会同时贯穿这两个层级，这就是为什么可重定向框架必须将二者联合处理，而不能将其视为彼此独立的阶段。

*两位作者贡献相同。
†部分工作在德累斯顿工业大学完成。

表 1：存内计算的硬件框架与编译器对比。✓：支持；○：部分支持；✗：不支持。MT：多个目标平台；Synth.：将任意操作综合为目标原语；Row：行分配；Destr.：对破坏性操作建模；Tensor：张量层级映射/调度；Bitw.：位宽分析；Input：编程入口点。

Work	MT	合成	Row	析构	张量	按位	输入
PUD 硬件框架（指令集架构 + μ程序库）							
SIMDRAM [19]	✗	○	✓	✓	✗	✗	ISA
Proteus [12]	✗	✗	✓	✓	✗	✓	ISA
逻辑级 *PUD* 编译器							
CHOPPER [40]	√	✓	✓	✓	○	✗	Usuba
PIMsynth [17]	○	✓	✓	✓	✗	✗	Verilog
张量级*PIM*编译器							
OptiPIM [32]	√	✗	✗	✗	✓	✗	PyTorch
Meta-PIM [54]	○	✗	✗	✗	✓	✗	C
CINM [25]	✓	✗	✗	✗	✓	✗	MLIR
OCC [45]	✗	✗	✗	✗	✓	✗	TC
CIM-MLC [41]	✓	✗	✗	✗	✓	✗	ONNX
Pudding	✓	✓	✓	✓	✓	✓	PyTorch

现有框架将架构约束直接硬编码到后端中，使得跨目标平台复用在结构上不可能实现（表1）。SIMDRAM 及其后续工作以 多数/非 为规范基础，并在此之上构建编译流水线 [11, 12, 19, 27, 37]。CINM [25] 引入了通用中间表示（IR）和通用优化基础设施。然而，特定于技术的后端，即逻辑优化、布局和命令生成，均未作明确规定，也未在不同目标平台间共享。这些后端本应由硬件供应商提供。其他工具（例如 Primo、Chopper、OptiPIM、PIMSynth 和 Meta-PIM [1, 17,32, 40, 54]）针对孤立的子问题，采用特定于目标平台的启发式方法，且没有整合逻辑与布局的共享抽象。结果是生态系统支离破碎：每种新的 PUD 架构都需要一个新的编译器，常见优化也必须从头重新实现。这种割裂源于一个更深层的误解，即现有工具将 PUD 视为固定操作集的模拟器，而不是可编程逻辑底层平台。当算术运算并非常规形式，且必须针对目标平台特定的原语成本合成最优布尔实现时，这一区别至关重要。更重要的是，既没有能够描述位切片、破坏性操作语义和布局方向等跨架构特性的通用抽象，也没有能够在架构特定约束下联合优化张量层与逻辑层的可重定向映射框架。

我们介绍了 Pudding，这是首个面向 PUD 架构、可重定向的端到端编译器。Pudding 的核心思想是将架构描述作为一等对象，使其贯穿每个编译阶段，例如用于参数化重写规则、原语成本、数据布局约束以及映射与调度决策；因此，重定向只需更新描述文件，而无需重写编译器遍。我们的具体贡献如下：

a) 参数化原语模型。我们为 PUD 系统定义了一种架构描述，用于记录目标平台支持的计算原语、这些原语的命令开销、计算行预算（按子阵列计），以及存储体和子阵列的结构约束。该描述是各编译阶段共用的接口。我们分别针对经过修改和未经修改的商用现成DRAM目标平台实例化了该描述，证明同一框架能够表达特征迥异的架构。

b) 一种架构感知逻辑优化器和代码生成器。我们开发了一种逻辑优化器，通过在 e图上进行等式饱和来重写并选择布尔网络实现，其重写规则由目标平台的原语模型自动推导而来。关键在于，该优化器探索子网络等价关系，而不是固守某种固定表示：e图会保留所有等价实现，并从中提取出在目标平台上命令数量最少的实现。

c) 一种映射与调度优化器。我们提出一种联合映射与调度方法，用于在多存储体DRAM架构上执行张量程序。我们的整数线性规划公式刻画了并行性与存储体间通信之间的权衡。与当前最先进的方法相比 [32],我们的方法不仅纳入了存储体级资源约束，还对操作特定延迟进行建模，从而捕捉算术运算与逻辑操作之间的差异。最终的映射器联合优化操作放置与数据布局，以尽可能缩短端到端执行时间，同时确保指令分派与执行正确无误。

d) 一个端到端的基于*MLIR*的框架。我们开发了一款编译器，可从机器学习框架接收高级算术程序，通过可扩展的 MLIR方言流程逐步降级为PUD原语，并针对指定目标平台生成优化后的命令序列。这种集成是本文的核心贡献：首次实现了从原语建模、逻辑优化到布局的架构感知贯通，无需针对每个目标平台进行人工干预。

在三种PUD架构上，Pudding在逻辑级优化中的成本效率最高可达1.92×，相较于张量级优化中的解耦映射，加速比最高可达3.25×；在所有内核大小和存储体数量下均能获得有效布局，映射加速比则比OptiPIM [32] 高1.06–1.38×。在SIFT1M [23],向量搜索任务中，从高于内核级别的PyTorch端到端编译，可将相较于CPU基线的加速比从 2.9× 提升至 5.8×。

2 背景

2.1 动态随机存取存储器的组织与操作

Fig. 1展示了现代动态随机存取存储器系统的层级组织。在系统层面（❶），CPU 通过一个或多个内存控制器发出内存请求，每个控制器管理一个独立通道，该通道连接一个或多个动态随机存取存储器模块。每个模块都包含若干动态随机存取存储器芯片组成的秩组，这些芯片以锁步方式运行，共同提供完整的数据总线宽度（通常为 64 位）。在每个芯片内部，存储器被划分为多个存储体，它们共用一条连接至芯片 I/O 电路的内部数据总线；每个存储体又进一步划分为多个子阵列，数据存储在由动态随机存取存储器存储单元构成的二维阵列中，每个存储单元由一个电容器和一个访问晶体管组成。每个子阵列中的外围电路，即本地字线（LWL）驱动器和感测放大器（SA），负责协调数据访问与恢复。

图 1 (❷) 展示了 DRAM读取的命令序列。首先，控制器向目标行地址发出激活命令（ACT），LWL解码器驱动相应的

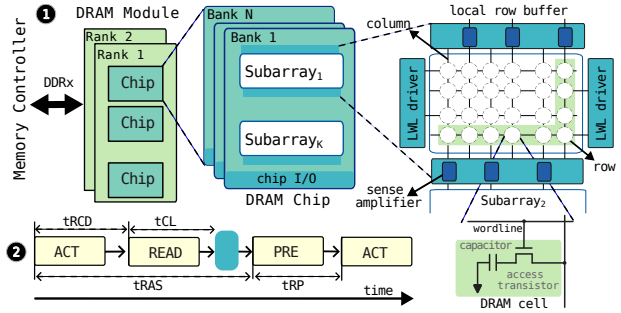


图 1：DRAM组织结构概览。

字线置高，将该行的单元格连接到其位线，感应放大器检测并放大微小的电荷差异，将该行锁存到行缓冲区。其次，控制器发出带有列地址的 READ命令；列译码器选择相应的感应放大器，数据通过内部总线传输到芯片的 I/O接口，再进入通道。最后，控制器发出预充电命令（PRE），将位线恢复到 $V_{DD}/2$，并使存储体返回空闲状态，为下一次激活做好准备。

2.2 使用DRAM进行处理（PUD）

PUD 利用标准 DRAM 命令直接在阵列中进行计算。RowClone [42]（图 2，❷）采用激活-激活-预充电（AAP）序列：对源行执行 ACT，将其锁存到感应放大器中；对目标行执行 ACT，触发电荷共享，用源数据覆盖目标行；随后执行 PRE，重置位线。在此基础上，多行激活（MRA）会同时激活子阵列中的多行，随后进行预充电（即激活-预充电，AP 序列），以执行批量位逻辑 [15, 19, 43]：这些 AAP/AP 序列称为原语，是 PUD 的基本构建模块。例如 MAJ3（图 2，❶），电荷在三行之间共享，产生多数编码电压偏差，感应放大器将其解析为数字结果，并以此破坏性地覆盖全部三行。MRA 可以通过定制行译码器同时断言多条字线来实现 [43],，也可以通过连续执行 ACT 命令违反 DRAM 时序约束，使多条字线同时保持激活状态 [15, 44, 52]。

为了实现功能完备性，非 可通过双触点单元（DCC）实现（图 2，❸），这类单元通过专用字线连接到位线或其反相信号，并通过RowCopy [43], 实现；或者在 PRADA 中 [44], 利用内部感应放大器行为（OCSA类型，不同于标准 BLSA）在无需额外电路的情况下反转数据。类似地，阵列内计算也存在不同的行分配策略：在设计时预留专用行，以换取更简单、更可靠的感测，但代价是存储容量减少且需要制造商支持 [43], ；或者通过违反时序让任意行参与计算 [15, 44],，从而恢复容量，但代价是电路复杂度或通用性降低。最后，未经修改的商用现成DRAM仍可通过激活共享感应放大器的相邻子阵列中的行来执行布尔逻辑（与/或）：激活-预充电-激活(APA)序列使用预先设置为分数电压的参考行 [16] 来偏置共享的感应放大器，而相邻的操作数行则执行实际计算 [15, 53]。

2.3 MLIR 编译器基础设施 re

MLIR 是一种编译器基础设施，用于在多个抽象层级和异构硬件目标之间表示和转换程序 [29]。MLIR 不采用固定的中间表示（IR），而是提供少量内置项，并通过方言实现其余部分的可扩展性；方言是自定义类型、操作和属性的逻辑分组。操作是核心构建单元，可使用和生成编译时间已知的带类型的值，并通过属性携带编译时元数据。方言通常按领域（linalg、tosa）、表示形式（affine、scf、memref）或目标平台（gpu、x86vector）组织。抽象会通过方言转换遍逐步降级，从高层领域专用方言转向特定于平台的方言；也可以升高 以恢复更高层次的结构 [4]。每种方言都会在中间表示中直接保留其转换所需的前置条件，从而降低分析成本和复杂性。模式重写是实现降级和优化的主要手段：将匹配给定中间表示模式的操作替换为语义等价、但处于不同抽象层级的操作，并通过可组合的贪心重写引擎实现。

2.4 布尔重写的等价图

等价图（e-图）是一种数据结构，可紧凑地表示大量等价项。e-图包含一组等价类 （e类），每个等价类都包含一组等价节点；每个等价节点由一个函数符号和一个有序的子等价类列表组成。e-图的关键操作是等式饱和 [49, 51]：从一个初始项开始，反复应用一组重写规则 $\ell \to r$。每当找到与 ℓ 匹配的项时，就将 r 加入与 ℓ 相同的等价类，而不是将其替换。此过程持续进行，直到没有新的重写可应用（达到饱和），或达到超时等停止条件。

这种非破坏性重写是 e-图相较于传统项重写 [8, 21], 的核心优势；传统项重写必须贪心地确定每次重写。例如，对 $\mathrm{MAJ}(a, b, \mathrm{MAJ}(a, c, c))$ 应用结合律，会得到 $\mathrm{MAJ}(a, c, \mathrm{MAJ}(a, b, c))$；之后，幂等模式 $\mathrm{MAJ}(x, x, y) = x$ 在顶层便不再可见，从而掩盖了最优结果 $\mathrm{MAJ}(a, b, c)$。相比之下，e-图会同时保留所有中间项和派生项，因此无论重写规则的应用顺序如何，都仍然可以得到最优形式。

3 动机

将 PUD 编译器移植到不同目标平台，并非简单替换指令集架构。PUD 设计的差异会同时影响多个编译器层。原语集决定布尔网络必须采用何种结构；行分配模型决定操作数放置和行预算；破坏性操作语义限制活跃性；数据布局方向则决定张量维度如何映射到子阵列行。任一属性发生变化，都会使其他层级已作出的决策失效。我们以两个目标平台为例：它们同属批量按位 DRAM 系统，却需要从根本上不同的编译策略。

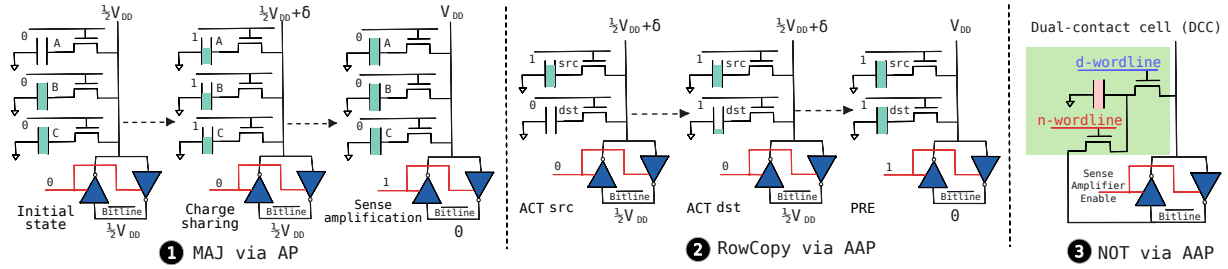


图 2：使用DRAM进行处理操作概览。

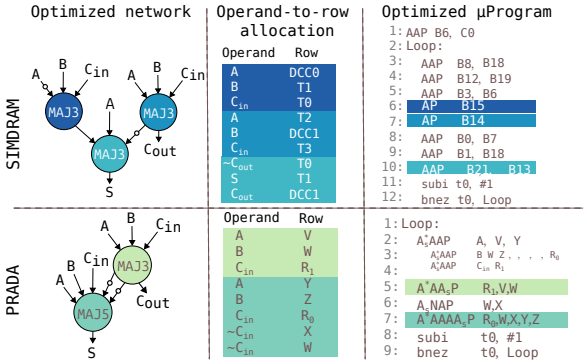


图 3：SIMDRAM 中的 1 位全加器 [19]和 PRADA 中的 [44]。

我们来看图 3 中的 1 位全加器示例。在 SIMDRAM 类目标平台上， [19],原语集（AAP/AP）可直接利用为 MRA 和 DCC 指定的行实现 RowCopy、非和 MAJ3。SIMDRAM 框架会将与/或/异或/非表示转换为经过优化的多数反相图（MIG），为 MAJ 节点操作数分配计算行，并生成一系列 AAP/AP 命令。整个优化流水线（即逻辑优化器、行分配器和代码生成器）都与这些假设密不可分。在 PRADA 上， [44], 原语（例如 AAAP 和 ANAP）是由命令序列构成的：每个被激活的行对应一条 ACT（A），还可选地包含一条非命令（N），最后以 PRE（P）结束。PRADA 放宽了行角色的限制，并允许非顺序激活，因此支持操作数多于三个的操作（例如五输入多数函数 MAJ5）、原位消耗操作数，以及从根本上不同的分配策略。同一个全加器编译后会形成结构不同的布尔网络、不同的行分配以及彼此不等价的命令序列；在任何层级上，这些结果都无法相互移植。

这些差异并不止于逻辑层；架构差异会向上传播。**结果 1**：张量级映射无法脱离底层算术成本进行独立优化。在位串行执行中，加法的命令数量随操作数位宽线性增长，乘法的命令数量则呈二次增长，因此即使精度只降低一位，也能为布尔网络中的每个操作节省命令。混合精度分配和操作调度等张量级决策的影响也会因此放大；要使这些决策有意义，就必须依据目标平台的位级成本。OptiPIM [32] 表明，如果整数线性规划纳入底层命令开销，张量级映射的质量会显著提升。**结果 2**：算术策略决定数据布局。如果目标平台的原语集使位并行进位保存加法比位串行脉动进位 [27], 更便宜，编译器就必须采用水平数据布局，并在任何假定采用垂直布局的地方插入显式转置。这一决策不能仅在张量层级上做出：它取决于算术策略，而算术策略又取决于目标平台的原语成本。布局并非预处理步骤，而是逻辑级编译的结果，并且会因目标平台而异。**所需方案**。可重定向的 PUD编译器必须对不同设计共有的特性进行抽象，例如位切片布局、破坏性操作语义和行级操作语义，同时将可变因素参数化，包括原语集、行模型、位串行与位并行执行方式以及成本。目标平台无关的优化应实现一次；目标平台相关的优化则应由架构描述驱动。我们提出了首个满足这些要求的编译器。

4 Pudding 流程概述

Fig. 4 概述了我们的多级编译流程。Pudding 通过 torch-mlir [39]接收来自 PyTorch 等高级框架的程序 [50], 并将生成的线性代数程序划分到 PUD 和主机 CPU 上。如果算子满足以下条件，则会将其卸载到 PUD：(a) 它符合可降低为位方言的某种计算模式，即具有规则访问模式的数据并行计算；并且 (b) 其迭代空间超过可配置阈值。该阈值随目标内存配置的并行性而变化，默认值为其处理单元中的 SIMD 通道总数。所有其他算子均通过 MLIR 的 CPU 流水线进行降低。

对于卸载算子，我们首先将针对 张量 的计算转换为针对 切片的计算。切片是在我们的 位 方言（第 5 节）中定义的一种数据类型。一个 切片 是一个二维位平面，与存储阵列一一对应。每个 切片 由一维张量派生而来，而更高维度的计算则通过循环表达，并相应地插入必要的归约。针对切片 的算术运算和按位逻辑运算会进一步降级为针对行 的操作循环，且与后端无关，例如 行_加 与 行_与。主机生成并广播至所有 PUD SIMD 通道的行操作数，无需存储在 DRAM 行中。对于该操作数的每一位，Pudding 都会针对另一个操作数单独生成两个更简单的操作，供主机在运行时选择：一个对应零，另一个对应一。反之，在链式卸载算子之间交换的中间结果

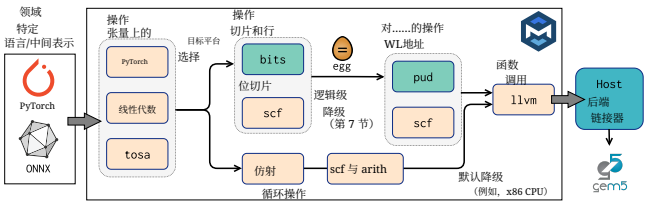


图 4：基于 MLIR 的 Pudding 流程概览。

可以共用预留行，将生产算子的结果与消费算子的操作数绑定，从而支持原位读取并避免与主机之间的往返传输。

对于逻辑级降级，我们提出了一种架构感知逻辑优化与代码生成器（第 7 节），用于根据这些行级操作构建布尔逻辑网络。逻辑级优化使用等式饱和，探索目标平台指令类型的等效实现，并采用贪心或基于整数线性规划的提取方法，确定适用于目标架构的最优指令序列，例如由AP 和AAP （第 2.2 节）构成的序列。

随后，我们将操作映射到动态随机存取存储器存储体，并将其视为处理单元（PU）。整数线性规划求解器（第 6 节）生成映射，在跨存储体并行性与同步开销之间取得平衡。在最后的代码生成步骤中，Pudding 将行级操作降级为pud方言的指令序列，为行类型的操作数标注全局字线地址，并展开循环。生成的代码构成一个数据流图，分派到各个动态随机存取存储器存储体上执行，从而确保指令按正确顺序排列和执行。

为了评估我们的工作，我们进一步将 pud IR 降级为 LLVM IR，并用编码自定义 PUD 操作码的内联汇编替换 pud 原语。这些指令会被汇编到编译后的二进制文件中，再由我们扩展的 gem5 [33, 37]ISA 解码器原生解码并执行，同时运行周围的 CPU 代码，以模拟完整的 CPU-PUD 系统。

5 数据表示与位切片

为衔接字级算术与位级执行模型，我们引入 bits 方言，作为线性代数与 PUD 后端之间的中间表示。该方言明确规定了位平面在动态随机存取存储器子阵列中的物理组织方式，因为不同的 PUD 执行模型对布局有不同要求：位并行方法 [30] 采用标准的字导向布局，而位串行方法则需要垂直布局。例如，在后一种情况下，张量<NxiW> 会转换为 bits.slice<WxN>，其中每一行代表一个位位置，每一列代表一个张量元素，从而在动态随机存取存储器子阵列内的各元素之间形成 SIMD 通道。张量与切片之间的转换通过两个结构性操作显式表示：bits.transpose 和 bits.assemble。具体实现方式取决于目标平台：它们可以降级为主机 CPU 上显式的数据移动例程，也可以由专用转置单元执行 [19, 44], ；如果数据已按所需形式存储且无需物理移动，也可以完全省略这些操作。这样便将布局语义与物理实现分离，使方言保持简洁，并避免不必要的数据移动。转换为切片后，计算按位平面组织，还需要进一步降级，以匹配基于行的 PUD 操作；为此，bits 方言定义了一种行类型，用于描述切片中的单行，并作为低级 PUD 原语操作数的抽象。

考虑一个 gemv 操作 tensor<MxKxi4>×tensor<Kxi2>，其目标底层平台是 SIMDRAM 这类位串行平台。图 5 展示了从 linalg 降级到 bits 方言，再从切片降级到行的过程。这要求编译器：(1) 通过沿一个维度循环，将二维矩阵映射为一维张量；(2) 采用图 5(b) 所示的映射-计算-归约方案，以实现高效调度。我们首先沿维度 M 广播向量操作数，使其形状与矩阵匹配，然后遍历 K：每次迭代都从两个操作数中提取一个长度为 M 的子张量，并执行位切片乘法，随后进行累加。乘法操作数（用于 bits.imul 的 bits.slice<4x64> 和 bits.slice<2x64> ）通过在 CPU 上转置提取出的张量 tensor<64xi4> 和 tensor<64xi2> 得到。累加作为沿 K 维度的归约，也可以延迟到乘法循环结束后执行。由于迭代之间不存在依赖关系，这样便能让所有迭代并行运行；不过，这会在存储体级并行性与归约所需的银行间数据移动之间形成权衡，我们将在第 6 节讨论这一选择。

从二维张量降级为位切片后，计算会进一步转换为在位行（即 DRAM 字线）上执行。我们实现了一个 MLIR pass，其中预定义的模式会将对 bits.slice<WxN> 的操作重写为一个循环，该循环在 bits.row<N>上迭代 W 次。为增强可扩展性，我们的框架还允许用户定义自定义重写规则，以组织位行上的计算，例如满足不同的计算需求。图 5(c) 展示了我们如何将 bits.imul 重写为 bits.row_and 和 bits.row_add 等行级操作，关键步骤见图 5(d)。在此示例中，切片级乘法通过遍历乘数位位置来降级。在每个位位置上，单个内层循环从当前移位偏移量开始，一直运行到最高有效结果行；它通过 bits.row_and 逐行生成部分积，并使用带进位链的 bits.row_add 将其累加到移位后的结果行中。当被乘数行用尽时，循环会自然转为仅对剩余的较高结果行进行进位传播。每次外层迭代结束时产生的进位会作为下一次迭代的初始进位传递过去，从而确保在不同乘数位位置之间正确衔接。通过这种方式，bits.slice<WxN> 乘法被分解为基于 bits.row<N> 原语逐行执行的提取、 → 与 → 加法、 → 插入，从而在降级过程中以位切片方式将整数乘法表示为位移加法，同时保留其算术语义。同样，针对 bits.slice<WxN> 的切片级 bits.iadd 会被降级为 W 次逐行提取、加法和插入迭代，顺序从最低有效位（LSB）到最高有效位（MSB），并在各次迭代之间正确传播进位。

6 操作与数据映射

为了执行多个存储体矩阵-必须乘海个操作定义存储体分配和调度。这些决策本质上相互耦合，即分布到不同存储体上可以提高并行性，但会导致存储体间数据传输。

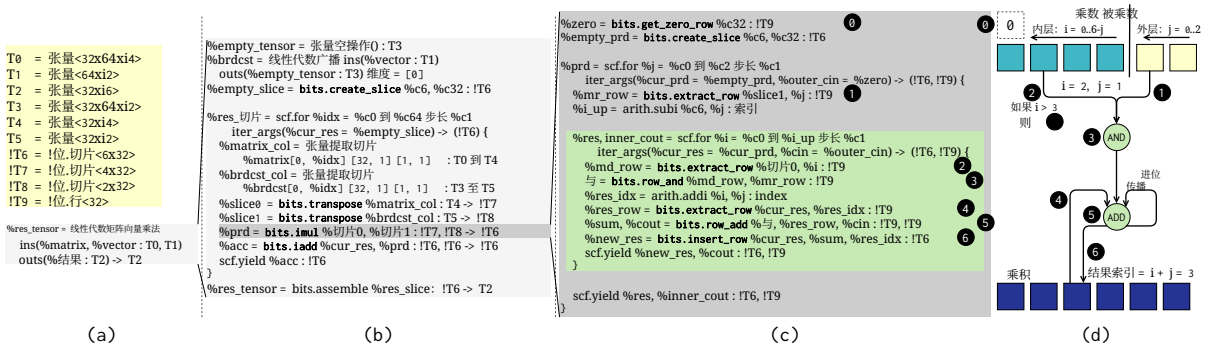


图 5：GEMV 从 linalg 经切片级位操作到行级位操作的渐进式降级。

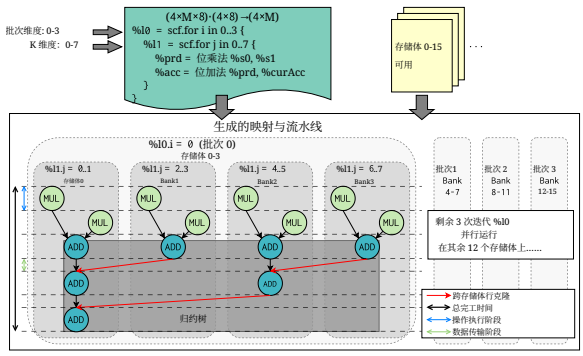


图 6：操作到存储库映射，以及批量矩阵-向量乘法的流水线处理。

我们将这一联合优化建模为一个整数线性规划，以最小化总完成时间。

我们映射 bits 方言中的算术操作，即那些被降级为固定内存命令序列的操作，但排除 bits.extract_行等结构性操作。我们考虑切片级操作而非行级操作，因为从单个切片级操作派生的所有行级操作都必须在同一个存储体上执行。每个待映射的操作都关联有一个延迟，该延迟根据逻辑优化器生成的命令数量（见 第 7 节）以及由位宽给出的迭代次数估算得出。例如，在 SIMDRAM 中，将两个 n位整数相加需要 $8n+1$ AP/AAP 条命令，而在 PRADA 中需要 [19] 条 $6n+1$命令 [44]（图 3）。该比率因操作而异，因此针对一个底层平台定价的调度并不适用于另一个底层平台。我们以一条命令作为时间单位。对于每个操作，整数线性规划决定存储体分配和开始时间，而结束时间则由操作延迟推导得出。约束确保每个操作恰好分配到一个存储体，并且满足数据依赖。

数据放置引入了第一个关键约束：值驻留在生成它们的存储体上。如果消费者被放置在同一存储体上，执行时无需承担存储库间传输开销；否则，该值会逐个位平面行进行复制，每行的成本取决于两个存储体是否共享同一通道。这通过每条边的二元变量建模，用于选择适用成本，该成本由两个操作的存储体和通道分配决定；如果二者共享同一存储体，则将延迟加为零。第二，存储体是一种顺序资源，但存在一个例外：由于一行存储体行包含许多元素，相同类型且位宽相同的独立操作可以共享一行，并作为一个行范围操作执行。排序变量将所有其他操作对串行化，但仅当两个操作共享同一存储体时才激活，从而让求解器在局部性与并行性之间进行权衡。第三，传输也会消耗存储库资源：一次传输会在其持续时间内占用两个端点存储体，因此我们要求它与共享任一端点的其他操作或传输不重叠。我们的整数线性规划公式详见 表 2。

循环体只有在既不包含归约也不包含嵌套循环时，才会以独立节点的形式进入整数线性规划；其他所有循环嵌套都会变成一个具有预计算延迟的单一节点，因为将其展开会使模型规模超出求解器的处理能力。对于归约循环，预计算延迟通过选择使计算加归约开销最小的并行度获得，并直接作为节点成本使用。该并行度并非孤立选择：归约未占用的存储体会被其外层并行循环中的并发实例占用，这些实例只计入其局部工作量，不计入其归约树；归约树的传输会争用相同的通道总线。因此，两个循环需要联合搜索，即搜索如何在二者之间划分存储体。

最后，我们采用两项简化。首先，整数线性规划根据传输两端存储体的位置为传输定价，但只强制执行银行级排斥；通道数据总线上的争用则在整数线性规划之外建模，由为归约树定价的闭式估计器处理。其次，以聚合方式定价的循环嵌套属于原子节点，这会限制全局最优性。求解时间受可配置的时间上限约束（默认为 300 秒）：达到上限时，使用当前最优解；只有在不存在当前最优解时，才回退到轮询分配。

图6 展示了一个批量矩阵-向量乘法（即矩阵形状为M×K、向量长度为K，其中K = 8，批大小为4），以及在16个具备计算能力的存储体可用时的一种映射和调度方案。这16个存储体被划分为四组，以实现批维度上的完全并行。每个通用矩阵-向量乘法分配到四个存储体，并以因子2展开内层循环。每个存储体依次执行两次乘法运算并累加其结果，随后通过一个倒置的二叉树执行存储体间归约

表 2：操作到存储体映射的整数线性规划公式。

符号	定义
集合与参数	
$\mathcal{N} = \{1, \ldots, N\}$	有向无环图节点（操作 / 循环）
$\mathcal{B} = \{0, \ldots, B-1\}$	存储体，划分为$\mathcal{K}$个通道
$E \subseteq N^2$; $C \subseteq N$	有向无环图边（依赖关系）；提升后的循环节点
$d_i \in \mathbb{Z}_{\geq 0}$	节点 i的执行延迟
w_i, v_i	结果的位平面行和元素
γ^{in}, γ^{x}	通道内/跨通道的每行克隆成本
W, M	存储体行容量；Big-M 常数
决策变量	
$x_{i,\ b} \in \{0,\ 1\}$	若节点 i分配至存储体 b，则为 1
s_i, $e_i \in \mathbb{Z}_{\geq 0}$	节点 i的开始时间、结束时间
$cl_{ij} \in \{0,1\}$	若边ij 需要跨存储体克隆，则为 1
$q_{ijb}, q_{kijk} \in \{0,1\}$	线性化辅助变量：$q_{ijb} = x_{i,\ b} \cdot x_{j,\ b}$，同样适用于每个通道 k
σ_{ij}; $\tau_{ij} \in \{0,\ 1\}$	若节点 i，j共享同一存储体，则为 1；共享同一通道（推导得出）
o_{ij}; $\pi_{ij} \in \{0,\ 1\}$	独立对的排序辅助变量；若打包在同一行中则为 1
约束	
C1; C2	$\sum_{b \in \mathcal{B}} x_{i,b} = 1; \quad e_i = s_i + d_i \quad \forall i \in N$
C3	$q_{ijb} \leq x_{i,\ b}, \quad q_{ijb} \geq x_{i,\ b} + x_{j,\ b} - 1 \quad \forall (i,j) \in E \cup P \quad \forall b \in B$
C4	$\sigma_{ij} = \sum_{b \in \mathcal{B}} q_{ijb}, \quad \tau_{ij} = \sum_{k \in \mathcal{K}} q_{kijk} \quad \forall (i,\ j) \in E \cup P$
C5; C6	$cl_{ij} + \sigma_{ij} = 1; \quad s_j \geq e_i + \gamma_{ij} \quad \forall (i,j) \in E$
C7; C8	$\gamma_{ij} = w_i \left[\gamma_{\text{在}}(\tau_{ij} - \sigma_{ij}) + \gamma_{\text{x}}(1 - \tau_{ij}) \right]$ $s_j \geq e_i - M(1 - \sigma_{ij} + \pi_{ij} + o_{ij})$ $s_i \geq e_j - M(1 - \sigma_{ij} + \pi_{ij} + 1 - o_{ij})$ $\pi_{ij} \leq \sigma_{ij}, \quad \lvert s_i - s_j \rvert \leq M(1 - \pi_{ij}), \quad v_i + \sum_j v_j \pi_{ij} \leq W \quad \forall (i,\ j) \in P$
C9	$T \geq e_i \quad \forall i \in N \setminus C$
目标函数	最小化（总完成时间）T

$\mathcal{P} = \{(i,\ j) \mid i < j,\ (i,\ j) \notin \mathcal{E}^*,\ (j,\ i) \notin \mathcal{E}^*,\ i \notin C\ j \notin C\}$：在两个方向上都不存在依赖关系的配对（$\mathcal{E}^*$ = $\mathcal{E}$的传递闭包）。π_{ij} 仅适用于种类和位宽相同的配对。根据 C7 的 big-M 模式，克隆会排除对两个端点存储体的其他操作 $[e_i, e_i + \gamma_{ij}]$，除非两个克隆共享两个端点，且其中一个同时服务于两者。

，共两层。每个通用矩阵-向量乘法安排了三次存储体间数据复制，其中第一层的两次复制并行进行。尽管在此示例中所有存储体都遵循同一条流水线，但映射过程支持为每个存储体分配独立的流水线。全局流水线确保内存命令在正确的时间发往正确的存储体。

归约树遵循存储层次结构，而非扁平的存储体列表：首先在每个通道内，将部分和减半，利用该通道自身的数据总线并在各通道间并行处理，然后再跨通道合并剩余项。任意配对存储体会为获得相同结果带来多得多的跨通道流量。映射过程不仅会构建这样的归约树以实现高效归约，还支持可选的位宽优化。默认情况下，累加会按输入程序要求的位宽进行回绕；也可以选择将 $n\,b$ 位值的和以其所需的 $b + \lceil \text{对数}_2\, n \rceil$ 位宽表示。只有当合并值的数量达到 2 的幂时，才会增加一位：这次加法会将进位输出作为新的 MSB 保留，并多花费一条命令，因此位宽增长与合并值的数量呈对数关系，而不是每次加法增加一位。在 图6中，八个 8 位乘积在第一级之后形成 9 行切片，并在根节点处得到 11 位结果；部分和的传输则按总和当前达到的位宽计价。

操作被分配到存储体后，便进行数据映射。一个动态随机存取存储器地址是由通道、秩、存储体、子阵列和行组成的元组。每个切片都会被分配到同一子阵列内连续的行。如果某个操作数 (1) 是通过对张量进行转置得到的，或 (2) 是由映射到不同存储体的操作生成的，则会为其分配这样的空间。每个结果始终会被分配一个这样的空间。

7 架构感知代码生成 n

将布尔函数映射到 PUD 指令（内存命令）时，一个核心挑战是如何在不同架构之间共享优化。以往的代码生成器 [6, 13, 17, 19, 35, 47] 每种只针对单一架构，并且在行分配之前、独立于行分配对网络的规模和深度进行优化（第 3 节）。我们提出了一种代码生成引擎（图 7），只需更新架构描述文件，无需重写优化遍，即可适配不同的 PUD 架构。在 ❶中，输入可以是经 MLIR 降级的网络，也可以是独立的逻辑描述，同时还需提供一份架构规范，其中编码了 PUD 指令约束和成本。该引擎先执行基于 e-graph 的重写与提取，再进行候选选择，从而生成带有隐式行分配的优化后命令序列。

7.1 输入表示

逻辑优化器的输入是一个布尔网络：一种有向无环图，其节点为逻辑门（与、或、多数、异或），边上携带单比特信号，且信号可以选择是否取反。优化器接受来自两个入口的网络：通过 mockturtle C++ library [46] 导入的现有 BLIF、Verilog 或 k-LUT 格式逻辑描述（提供与反相器图（AIG）和多数反相图（MIG）等标准类型）；或 bits 方言，其中的操作（例如 bits.row_add、bits.row_mux 等，每个操作都会产生一个单比特 SSA 值）通过 MLIR 流水线转换为标准网络类型（将 bits.slice 降级为 bits.row）。例如，R = torch.where(s, A, B) 会降级为 %R = bits.row_mux %B, %A, %s，对应的逻辑网络为（$A \wedge s$） $\vee$ （$B \wedge \neg s$），并在初始 e-graph 中表示为图 7（❺）所示。无论采用哪种方式，网络都会传递给 e-graph 重写器，由其探索目标平台指令类型下的等价实现。

7.2 架构约束与成本

我们通过结构化的架构规范来描述每个 PUD 目标平台，该规范涵盖共同决定编译成本的三个属性：可用单元类型、操作数放置规则和指令语义。 图 7 (❷) 展示了映射到一组 结构体 和枚举的规范语法： 单元格 包含名称和可选容量，操作数 将单元格组合在一起，并带有可选的反相标志；而 操作数模式规定在给定指令输入或输出位置上哪些单元格和极性是合法的。最后，指令 包含作为枚举变体的原语（多数等）、输入操作数模式集合列表、输出操作数模式集合列表，以及一组破坏性输入位置。

架构约束首先来自操作数模式系统：每种指令类型都会静态限制每个操作数槽位可容纳的单元类型，以及其极性（是否反相）。例如，图 7 (❸) 给出了 SIMDRAM 的具体规范 [19], , 其中高亮显示的 操作数模式 规定了哪些行组合可用于 MRA。第二种约束来源是 复制图 (图 7，❹)，它是一个有向图，节点表示单元类型模式，边则编码它们之间代价最低的

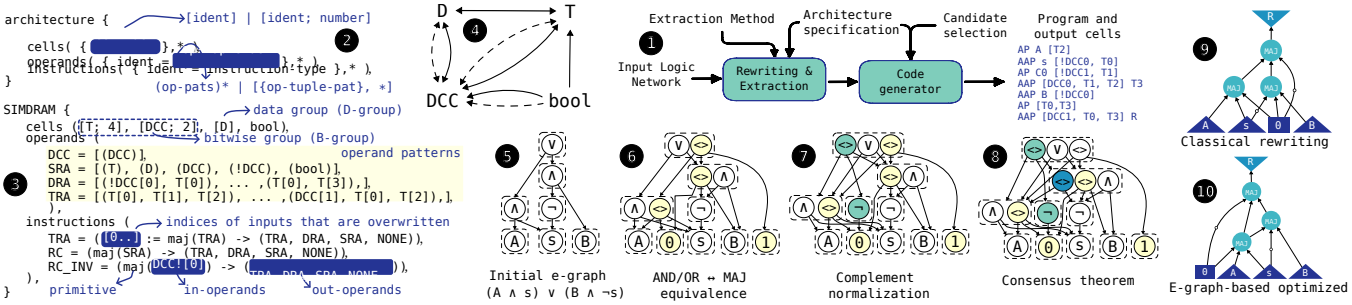


图 7：代码生成器概览。输入是以逻辑网络表示的电路；输出是隐式进行行分配的优化后命令序列。

表 3：相对于 PIMsynth 的重定向工作量。

组件	PIMsynth [17]	Pudding
前端、位切片、行级降低	复用；每个内核对应的寄存器传输级	复用
张量级映射与调度	未提供	复用
重写规则	无（ABC 映射）	已选择子集
命令编码	每条命令、每种模式	若共享则重复使用
架构描述、成本权重	标准单元库；单位成本	<20 行
复制图、提取成本	手工编写的遍	派生
命令序列及每次操作的成本	每个门手写	派生
保留行地址映射	手写	派生

非破坏性复制操作，并标注其代价以及复制是否会反相（虚线边）。对于 SIMDRAM，单元类型包括 DCC、数据（D）、按位（B）和常量 0/1 行；复制图 ❹ 编码了信号可在这些类型之间如何移动。

复制图在启动时通过符号求值，根据指令类型自动派生，因此用户无需手动指定。在代码生成期间，每当必须移动信号以满足操作数约束时，就会在复制图上进行最短路径搜索（Dijkstra 算法），以确定成本最低且合法的复制指令序列。这种复制开销正是具有灵活寻址功能的架构（如 PRADA [44] ）与具有严格行分配限制的架构（如 SIMDRAM [19] ），或具有限制只能使用相邻子阵列的架构（如 FCDRAM [53]）之间的区别所在。复制开销会纳入成本函数，该函数将每种指令类型映射为非负整数；程序总成本则是所有指令成本（包括复制指令成本）之和。

描述范围。 如果目标平台满足两个条件，则称其可表达：(i) 每个原语都必须以整行作为操作对象，以计算 与、或、多数$_k$或 异或门，并可选择反转输入和输出，或写入常量；(ii) 目标平台的约束必须能够用每个操作数可用的单元格和极性，以及每个原语会覆盖的输入来表达（第 7.2 节）。这涵盖了采用专用计算行的设计（SIMDRAM、Am- bit [43]）、行角色灵活的设计（PRADA），以及未经修改的商用现成 DDR4（FCDRAM），每种设计的描述都不到 20 行。 表 3列出了在 Pudding 中支持新目标平台所需的内容，并与 PIM-synth 进行了比较。

7.3 等式饱和步骤

在 Pudding 中，我们使用 egg 库 [51], , 因为当时没有可与 MLIR 集成的开源 e图 实现。

在本研究中。将等式饱和（第 2.4 节）应用于我们的场景时，给定布尔函数的初始逻辑网络会被置入初始 e图（图 7，❺）：每个门和信号都会成为一个 e类（虚线矩形）。应用重写规则（其中一组列于 [2, 9]）会扩展等价类，纳入若干推导出的替代方案（步骤❻、❼和❽）。重要的是，规则集会根据架构进行筛选：如果某种架构只有多数原语，则只启用与多数相关的规则；如果架构同时具有多数 与XOR，如 [48, 55]，则也会纳入“多数分配于XOR”等交叉规则 [9]。

7.4 提取：选出最佳候选方案

等式饱和将重写与选择解耦：所有等价网络都累积在一个 e图中；重写完成后，提取过程会全局应用架构特定的代价和布局约束，选择使目标平台指令数量最少的实现，而不是孤立地最小化门数量。重写之后，我们必须从 e图中为代码生成选取逻辑网络的一种具体表示。提取问题正是应用架构特定代价和约束的环节。我们设计了三种提取方法：

代码生成引导的提取。 最简单的方法是将实际的代码生成器用作成本预言机。从根等价节点开始，我们遍历该等价节点中的代表项，并为由其得到的部分逻辑网络生成代码。我们贪心地选择实际程序成本最低的项，然后递归处理其子等价节点。由于成本来自真实的生成器输出，因此每个局部选择都依据实际分配开销进行评估；但按锥逐一贪心决策仍可能导致最终组装的程序在全局上并非最优。这种方法可能很慢，因为为每个候选项运行代码生成的开销很大。

局部成本函数提取。这种方法会重写 e- 图，使架构成本成为每个节点的局部成本。PUD 特有的难点在于非对称极性成本：某些指令天然会产生反相结果，因此要获得相反的极性就需要额外复制。我们通过将每个等价类拆分为两个访问等价类 （反相和非反相）来对此进行建模：访问指令原生生成的极性无需成本，访问另一种极性则需支付一项根据架构复制图估算的复制成本。这样，每个等价节点都带有局部成本（逻辑门的指令成本，访问节点的复制成本），从而可以使用 egg 的整数线性规划 [51] 进行标准提取，在此模型下找到成本最优的代表；

也可以使用成本更低的贪心变体，但它可能会忽略共享的公共子表达式。

关闭编译提取。局部成本提取仅对估算成本而言是最优的：它计入的复制开销是根据复制图得出的近似值，因此无法保证它返回的单个网络能够编译成成本最低的程序。我们不再依赖单次提取，而是以较低成本生成多个候选网络，再让实际代码生成器从成本扰动贪心提取（为每个节点加入确定性抖动，以使生成的结构多样化）或整数线性规划解中挑选完整程序。每个候选网络都会完整编译，我们保留实际程序成本最低的那个。与代码生成引导的提取不同，后者按锥逐一贪心决策；关闭编译提取会比较完整组装后的网络，从而纳入跨输出共享以及行分配在全局范围内引入的复制。

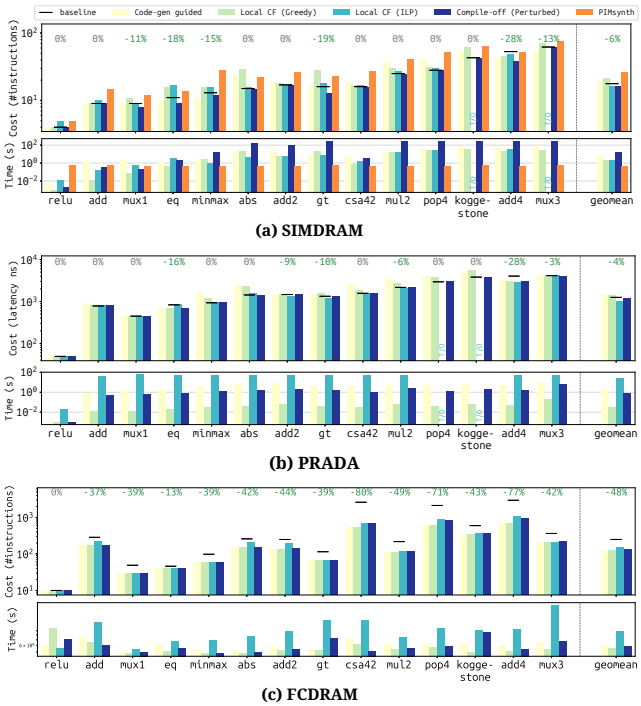
最终得到的是一个逻辑网络（图 7，⓫），专为目标架构量身打造：它仅使用可用原语，其形式可减少行分配规则所引起的复制开销，并按照目标平台的成本函数将指令成本降至最低。图 7 (⓬) 展示了最先进的经典重写所得到的解 [46],与基于 e图 的结果 ⓫相比，指令数从 8 个减少到 7 个 AP/AAP。随后将生成的网络输入代码生成器，以生成最终的 pud 方言指令序列。

8 实验设置与评估

模拟基础设施。我们在基于 gem5-CIMD 构建的主机-PUD 系统上评估 Pudding [28], , 该系统已通过 [19, 37],验证，可模拟三种 PUD 架构（第 2.2 节）下的 HBM3-4Gb-x64 和 DDR4-2400-x64：SIMDRAM [19], PRADA [44], 和 FCDRAM [53]。每个目标平台都作为架构描述添加（第 7.2 节，表3）；我们没有修改任何优化过程。我们采用两种执行模式，以捕捉与各项实验相关的影响。对于 表4，由于主机阶段会影响运行时间，我们使用 gem5-CIMD 的全系统模式：x86 核心将每条行命令作为一条指令发出。对于 图10–图12，我们则直接在内存控制器上执行 Pudding 和 OptiPIM 编译后的调度。这样可以排除代码布局等主机端影响，并适配 OptiPIM 的输出；其输出由命令流组成，而非主机程序。

基线。对于张量级优化，我们将其与 OptiPIM 进行比较。[1]OptiPIM 的代码生成器和模拟器采用了若干近似，因此直接与我们的 gem5 模型比较并不公平。我们通过以下方式进行修正：(i) 将 OptiPIM 的代码生成从 Ramulator 移植到 gem5，并通过相同的 DRAM 控制器驱动；(ii) 将 SIMDRAM 的 n 位 imul 成本建模为 $\approx 13n^2$ 次操作（而 OptiPIM 采用 $7n^2+1$ 次操作，未计入 RowClone 操作）；(iii) 使用浮点数计算其整数线性规划成本界，因为 64 位整数在超过 2^{32} 个乘加运算单元时会溢出。默认情况下，OptiPIM 仅模拟一个代表性存储体，并假设其他 $N-1$ 个存储体完全同步执行；我们的 gem5 模型则通过在 t_{FAW} 窗口内进行 t_{RRD} 流水线，捕捉真实的存储体并行性。在逻辑级优化方面，我们将其与 PIMsynth [17],进行比较。PIMsynth 在我们的目标平台中仅支持 SIMDRAM；此外，我们还与基于 mockturtle [46] 构建的基线进行比较。mockturtle 是一个开源逻辑综合库，实现了基于切割的重写

[1]https://github.com/ETHZ-DYNAMO/OptiPIM/ – 提交 9be8f9a。



(a) SIMDRAM

(b) PRADA

(c) FCDRAM

图 8：不同提取方法的程序成本（指令数或延迟）和优化时间。

以及代数MIG优化 [2] ，这些优化用于标准ASIC/FPGA流程，也是 SIMDRAM 所依赖的同类优化器。两种基线都会最小化网络规模和深度，但不了解目标平台的行分配和复制成本。

忠实还原 SIMDRAM。我们根据其描述 [19], 复现了 SIMDRAM 的行分配和 μ程序生成。照此流程直接执行，无法生成任何正确的 μ程序。因此，我们将其与 SIMDRAM 的最优重构进行比较，即在其命令集下进行穷举分配搜索。这与 SIMDRAM 已公布的加法、减法、比较、if-else 和最大值/最小值操作的计数相符，但与 for、equal 和 ReLU 的计数不符。由于该重构针对给定的 MIG 是最优的，因此成本由 MIG 本身决定，而 SIMDRAM 在选择 MIG 时并未考虑分配。

8.1 逻辑级优化

本节表明，在 e图上进行候选选择时，将分配成本反馈纳入其中，能够使 PUD 综合生成成本低于现有先进方法的程序。我们将同一综合流程重新定向至 SIMDRAM [19], PRADA [44], 和 FCDRAM [53],，并在 14 个内核上，按提取策略报告程序质量（指令数或延迟）及端到端优化时间（即逻辑优化器生成指令序列所需的时间），见图 8。

基线代表了先前的 PUD 综合方法 [19,37] ，这些方法将网络优化与门到行分配视为两个独立的顺序阶段。基线不执行 e-图重写，但网络仍会先经过 mockturtle 预优化，然后交由与其他所有策略完全相同的列表调度器处理。

```
C3 {
  单元格 ([T; 4], [U; 5], [S0], [S1], [D], 布尔值),
  操作数 (
    SRC0 = [(T), (S0), (D), (布尔值)],  DST0 = [(T), (S0)],
    SRC1 = [(U), (S1), (D), (布尔值)],  DST1 = [(U), (S1)],
    INV = [(T[0]), (T[3])],              NINV = [(!U[1]), (!U[4])],
    TRA = [(T[1], T[2], T[3]), (U[1], U[2], U[3]), (U[0], U[1], U[4]), (U[2], U[3], U[4])],
  ),
  指令 (
    TRA = ([0..] := 多数(TRA) -> (无)),
    RC0     = (多数(SRC0) -> (DST0)), RC1     = (多数(SRC1) -> (DST1)),
    RC_INV = (多数(INV) -> (NINV)),
  ),
}
```

图 9：C³ [24]的原语集作为 Pudding 描述。

在 SIMDRAM 上，PIMsynth 的几何平均成本比基线高 1。47× 图 8将三种提取方法与第 7.4 节中的基线进行了比较；每个内核采用最佳策略所获得的成本降低幅度标注在柱形图上方。

与不进行重写相比，贪心提取器和代码生成引导提取器的表现较差，见 图 8a 和 图 8b。在所有架构中，基于 ILP 的局部 CF 和经过扰动的关闭编译平均而言是最佳策略。基于 ILP 的局部 CF 是一种强大的方法：它是唯一一种在 PRADA 上优于关闭编译的方法，但无法扩展，在较大的内核上会超时（300 秒）。关闭编译的质量达到或超过 ILP，且不会超时，也从未比基线更差。

与解耦方法相比，关闭编译最多可降低 25% 的成本；相较于不进行重写的方法，FCDRAM 的性能提升了 48%，SIMDRAM 和 PRADA 则提升了 4–6%。其机制在所有情况下都相同：只有关闭编译的方法会评估整个程序的成本，因此能够计入破坏性存内计算在子表达式被多个输出共享时插入的复制操作所带来的开销，而代码生成引导的提取器和局部 CF 提取器都无法察觉这一点。关闭编译提取会编译一小组候选网络，并保留成本最低的网络；每个内核的编译时间从几秒到几分钟不等，且随候选网络数量增加而延长。基于整数线性规划的提取器可作为质量参考，但处理较大的内核时会超时。

效果取决于架构。PRADA 最能体现这一机制：由于一次复制的成本约为一次计算操作的两倍，忽略复制成本的优化器若复制共享逻辑，就要付出双倍代价，而耦合方法带来的优势会直接体现在延迟上。我们使用延迟作为 PRADA 的质量指标，因为其指令的延迟差异很大；这也检验了成本函数的可重定向性。FCDRAM（图 8c）则处于相反的情形。其未重写网络规模较大，因此所有策略都能带来显著改善（0.52×–0.60×），而区分各策略的分配成本则不那么重要。

重定向工作量。我们将 Pudding 重定向到 C³ [24],，这是近期提出的一种 PUD 基板，并已在商用 DDR4 上得到验证。它只保留经过验证可靠的原语，因此所有电路都必须通过 `MAJ3` 以及一种通过复制到相邻子阵列来实现的取反。 图 9 展示了它在 Pudding 中的架构描述。仅凭这一描述，Pudding 就能自动推导出一个包含 16 条命令的全加器程序：10 次复制、2 次反相复制、4 次 `MAJ3`。这一命令序列与 C³ [24], 中给出的序列一致，后者由人工编写并在硅芯片上验证。进行重定向时，Pudding 的前端、张量级映射和重写规则都可以复用，而复制图、命令序列和成本则根据所提供的描述生成。与以往工作相比，像 SIMDRAM 这样的特定目标平台流程会围绕一套原语集构建其逻辑分解、行分配、成本模型和 μ程序，因此每个变体都意味着要重写这些内容。PIMsynth 支持不同的门库，但它仅用数量描述计算行，并假设每行都支持取反，因此 C³固定的行三元组和跨子阵列取反无法纳入其描述。在 Pudding 中，更改行选择只需修改操作数模式（图 7，❸），之后无需其他输入即可重新编译每个电路。

标准逻辑综合针对 PUD 的优化不足。 以往的 PUD 流程沿用 ASIC/FPGA 综合中的逻辑优化：SIMDRAM 使用代数MIG重写 [2],，而 PIMsynth 则依赖 ABC [5, 17]。这些工具依据一种成本模型来最小化门数量和深度；在该模型中，扇出不产生成本，且门输入在计算后仍然保留。然而，这些最小化假设在 PUD 中并不总是成立。 图 8 显示，我们的基线方案和 PIMSynth 虽然采用了逻辑门数量最优的网络，但可能因此编译成 更多 命令（例如 图 7中的 ❾ 与 ❿相比），因此逻辑优化必须考虑目标平台的资源分配和复制成本。

8.2 张量级优化

本节将展示，我们基于整数线性规划的映射与调度速度提升了 6–38%，并且不同于 OptiPIM [32],，能够为每种形状和存储体数量生成有效布局。我们通过改变存储体数量（从 8 到 128）来测试不同规模的 通用矩阵-向量乘法 和 矩阵乘法 ，输入采用 `int8` 格式，进行 `int32` 累加。 图 10 报告了在高带宽内存 HBM3 底层平台上，采用公平的顺序执行模型测得的总完成时间。

我们的调度之所以效率更高，有两个原因。首先，我们通过阵列内复制来执行归约，而不是经由 HBM 输入输出流式传输部分和（因此省去了 OptiPIM 命令序列中用于读出的 37%）。这是存储体数量较少时的主要开销，也是 8 个存储体时差距最大（最高达 1.38×）的原因。随着存储体数量增加，两种调度都会从受读出限制转为受 ACT 限制，而阵列内归约也越来越多地被并行工作掩盖，因此在 128 个存储体时，加速比降至 1.06–1.34×。这一比较对 OptiPIM 有利，因为其成本模型只计算整数乘法的开销，并将加法视为免费操作，尽管加法占 GEMV 命令序列的 13.4–15.2%。若将加法计入成本，无论是在内存中执行，还是通过内存之外的额外逻辑执行，都只会进一步扩大对我们有利的差距。

其次，Pudding 暴露了细粒度、相互独立的行操作，在 t_{RRD}/t_{FAW} 限制下仍能让每存储体激活流水线保持满载；而 OptiPIM 粗粒度的逐乘法广播会将归约步骤串行化：位切片整数乘法和加法成为独立的宏操作，整数线性规划可在各存储体间对其进行流水线调度，无需设置屏障；而 OptiPIM 发出操作主序广播，并在每个时间步设置屏障，直到当前归约步骤在所有 N 个存储体上完成后才执行下一步，导致流水线未能充分利用。

另外，OptiPIM 将数据布局建模为一个 ILP，一次性将整个 $M \times K \times N$迭代空间映射到 HBM 上，并让每个归约始终驻留其中，不留溢出或流式传输部分和的路径。因此，可映射容量受每列、每个存储体和存储体总容量的乘积限制（对于此高带宽内存配置，该乘积为 $M \cdot K \cdot N \le 2^{29}B$，其中 B 为存储体数量），并随存储体数量

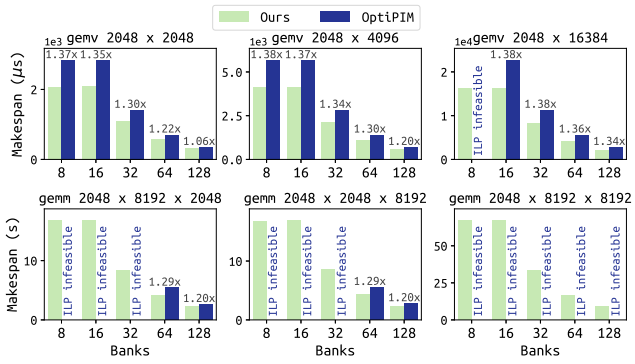


图 10：在 HBM3 上使用 8–128 个存储体时，我们的方法与 OptiPIM 在通用矩阵-向量乘法（上）和矩阵乘法（下）上的总完成时间。

线性增长 B：存储体数量较少时，大型矩阵乘法会超出这一上限，导致 ILP 无可行解；而 Pudding 会在时间维度上对归约进行分块，并映射所有形状。

位宽感知归约。我们的编译器率先在 PUD 架构的自动化流程中引入位宽分析。通过确定计算过程中每个位置表示数据所需的最少位数，我们在保持准确性和正确性的同时，将内存命令数量减少了 11–87%。图11 展示了输入精度从`int1`到`int8` 的变化通用矩阵-向量乘法(2048×8192，报告命令数量（左）以及使用 128 个存储体时的总完成时间（右）。由于操作数采用位串行表示，朴素的降级方式会将每个部分和累加到`int32` 累加器中，并为每次加法付出固定开销。我们的归约方式则在加法器树的每个层级将加法器位宽增加一位，因此每次加法的开销恰好与其操作数所需的位宽相符。精度较低时收益最大，随着 b 增大，收益逐渐减小。高精度下，乘法项（$\Theta(b^2)$）占主导地位，固定宽度累加器已接近实际运算所需的位宽，因此可精简的部分很少；而在使 PUD 具有吸引力的量化区间（`int1–int2`）中，归约占据了绝大部分开销，我们的编译器则消除了其中的大部分。

定宽基线将累加器限制在 $2b + \log_2 K$ 位的最终归约位宽（定宽累加器），弥补了部分差距，但在操作数较窄的众多早期树层级上仍需承担最宽位宽，因此，与定宽基线相比，逐层扩宽可进一步减少 9–74% 的命令。OptiPIM 在该图中的优势是成本模型不完整造成的假象，而非调度效率更高。由于 OptiPIM 将归约视为免费，其柱状条表示仅乘法下限。绿色和青绿色柱状条上的斜线填充区域表示 Pudding 的乘法下限，证实二者具有相同的乘法成本。深蓝色和青绿色柱状条之间的差距（斜线填充区域与非斜线填充区域之差）正是 OptiPIM 所忽略的归约，而这种遗漏恰恰在 PUD 最有用的低位宽情况下影响最大。

编译时间与可扩展性。单个int8 矩阵-向量乘法和矩阵乘法操作符在 2^{22}–2^{32} 乘加运算单元和 8–128 个存储体上端到端编译耗时 0.77–0.92 秒。逻辑优化器（图 8）占据其中 80–92% 的时间，其耗时仅取决于内核所使用的行操作。

无论张量大小和存储体数量如何，映射耗时均为 12 ms，且仅随算子数量增加而增长。对于 128 个存储体上的 64 个矩阵-向量乘法层，

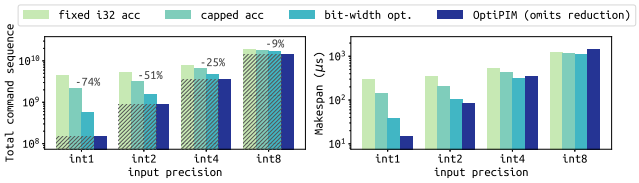


图 11：gemv 2048×8192 在 int1–int8 输入操作数下的命令数量（左）和 128 银行完成时间（右）。

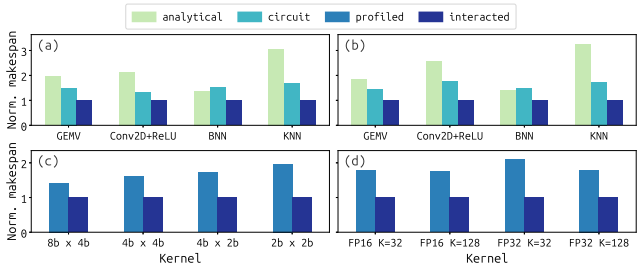


图 12：128 银行 DDR4 上的总完成时间，按每个内核归一化，以体现与逻辑级交互的映射。（a）SIM- DRAM；（b）PRADA；（c）SIMDRAM：包含四个近似乘法器的一个 int8 通用矩阵-向量乘法内核；（d）SIMDRAM：包含一个近似乘法器的四个浮点通用矩阵-向量乘法内核。

整数线性规划包含 78 万个约束，并在 11.4 秒内求得最优解。在 8 个存储体上，层数达到 32 层时，求解器触及 300 秒的时限，编译则使用当前最优解继续进行。从未需要使用轮询回退方案。

8.3 优化级别之间的交互

以往的 PUD 编译器要么省略张量级优化 [12, 17, 40] [32] ，要么利用手工推导的成本模型来执行张量级优化。Pudding 首次使用由逻辑级自动提供的逐原语成本来进行张量级映射和调度决策，并生成包含位宽分析等张量级信息的逻辑级程序。本节将量化这两个相互耦合的优化级别之间交互的价值。

图 12 （a）和（b）将两种解耦成本模型生成的映射与一种会与逻辑层交互的模型生成的映射进行了比较。该分析 模型使用张量层级上可获取的有关操作的所有信息，例如位宽和复杂度；而电路 模型则是架构特定的，并根据每个操作的逻辑网络中的逻辑门数量为其定价。两种模型都知道 DDR4 上存储体间数据传输的准确成本。在 SIMDRAM 和 PRADA 上的四个内核中，分析模型的速度比交互式模型慢 1.37–3.25× 倍。 电路 模型在四个内核中的三个上表现更接近，但仍在每个内核上落后，慢 1.34–1.78× 倍。原因在于，优化张量级循环时，数据局部性与并行性之间的权衡只有在已知精确的计算延迟后才能正确做出；而这需要依据编译后的逻辑级程序，为每个操作准确估算成本。已知传输延迟后，高估计算成本会导致循环过度展开，使

表 4：SIFT1M 上近似最近邻搜索的每次查询延迟（毫秒）和动态随机存取存储器能耗（毫焦）；每一行将更多部分编译至 PUD。

	投影	广播	扫描	选择	重新排序	总计	加速比	能耗节省
仅 CPU	0.01	–	5.07	1.05	0.26	6.39	–	39.0 –
内核卸载	0.05	0.07	0.59	1.18	0.27	2.18	2.93×	15.1 2.59×
+广播专用化	0.05		– 0.46	1.17	0.27	1.98	3.23×	13.4 2.90×
+内核链式调用	0.04		– 0.46	0.35	0.25	1.10	5.84×	8.1 4.79×

执行受传输限制；而低估计算成本则会使存储体闲置，导致执行受计算限制。

解耦映射器也可以使用测得的成本，例如目标平台上每种操作的实测延迟。对于从一个程序到一个底层平台的固定降级，基于实测数据的成本模型会得出与 Pudding 相同的调度。不过，每种降级都需要进行一次这样的性能分析。由于 Pudding 允许用户定义自己的算术算法（第 5 节），图 12（c）展示了如何使用断裂阵列乘法器的四种配置，改变 `bits.imul` 在一个 int8 矩阵-向量乘法 （2048×128）中的行级实现 [34], ，该乘法器保留高位 $A{\times}B$ 并丢弃低位的部分积；而（d）则以 近似方式实现四个 内核的 bits.fmul，方法见 [18]。与 Pudding 相比，使用精确 `int8`、`FP16` 和`FP32` 乘数的实测成本的解耦映射器，生成的映射会慢1.42– 2.1× 倍。在两个子图中，实测成本对于张量层级所见的程序都是准确的。性能损失仅源于逻辑层级的变化未传递给映射器，导致映射器将较便宜的计算拆分到过多的存储体中，得不偿失。要实现最优映射，解耦流程必须在每次此类变化后，针对每种操作、位宽和底层平台重新进行性能分析。在 Pudding 中，借助两个优化级别之间的交互，这一过程会在每次编译时自动完成。

8.4 端到端编译：向量搜索

本节将一个完整应用——针对 SIFT1M 的近似最近邻搜索（ANNS） [23], ——从 PyTorch 端到端编译到采用 128 个存储体的 DDR4 上的 SIMDRAM。查询通过浮点随机投影映射为 128 位 SimHash [7] 编码，并通过汉明相似度进行扫描，与 2^{20} 编码比较；随后选出排名靠前的 C=1000 个结果，并根据 L2 距离重新排序。这种先筛选再重新排序的结构是 C^3 手动映射到商用 DRAM 的。Pudding 对 torch-mlir 输入进行分区：投影的 128×128迭代空间远低于卸载阈值，而重新排序会动态遍历候选项（第4节），因此这些操作留在主机上执行。主机是一颗 2GHz 的乱序执行 x86 核心，配有 32KB L1 缓存和 1MB L2 缓存；它还负责为 PUD 发出行操作。CPU 单独运行时，也使用这颗核心和相同的存储系统。它使用硬件 popcount 指令，并以原生紧凑布局扫描编码，每个编码由两个 64 位字组成；而 PUD 配置则将编码以垂直布局存储在 DRAM 中，并在构建数据库时仅存储一次。DRAM 能耗依据 Micron/DRAMPower 模型计算，并将每次行操作激活都计作一次三行激活 [43]。

表4 逐一启用 Pudding 的程序级编译功能。 内核卸载 会编译卸载的算子，此处为扫描，使其执行时间从 5.07 毫秒降至 0.59 毫秒。然而，内核编译仍无法揭示两项开销。首先，每个广播查询都会存储在所有存储体的 128 行中，并执行同或非操作，尽管对于所有编码而言，查询的每一位都相同。其次，选择的每一步都是一个独立的小型内核，它通过主机传回并转置操作数。 广播特化 消除了第一项开销。如 第4节所述，每个查询不再存储在 DRAM 行中，且与查询位进行的每次 同或非 操作都会变成对编码行执行 复制 或非 操作，这由 C^3 手动推导得出。每个查询的行操作减少了 25%，广播和扫描合计耗时减少了 0.2 毫秒。内核链式调用 通过将链式内核共享的中间值保留在 DRAM 中，消除了第二项开销。数据与主机之间的往返传输也随之消除。小型内核通过共享每个存储体中的 20 行，在原位交换结果和操作数。主机只需读回它们的位图，选择操作的耗时便从 1.17 毫秒降至 0.35 毫秒。

9 相关工作

尽管 PUD 架构日益多样化，可编程性支持仍然高度碎片化（表1）。我们区分了硬件框架，它们将 PUD 底层平台与指令集架构及逐操作的 μ程序库相结合；而编译器则会将任意程序转换为原语序列。SIMDRAM 等硬件框架 [19]和 Proteus [12] 提供固定的一组操作，这些操作采用固定或运行时选择的位宽；库中未包含的任何操 μ作都需要手动推导新的程序，且程序员必须用这些操作来表达应用。相比之下，PUD 编译器会执行高层次综合，直接从程序中推导任意操作的布尔实现、行分配和调度。Chopper [40] 和 PIMsynth [17] 是适用于多种位串行 PUD 基底的逻辑级编译器。二者都不会降低来自机器学习框架的张量程序，也都不会针对目标平台的行分配和复制成本优化布尔网络。

OptiPIM [32] 表明，当整数线性规划调度器获知较低层级的命令成本时，张量级映射质量会显著提升——这一关键洞见促使我们采用联合优化方法——但它并未提供可重定向的逻辑后端。Meta-PIM [54] 为可配置的缓存内互连结构编译C程序，利用等式饱和为每条语句选择循环分块以及位并行或位串行模式。CINM [25] 是最接近的多目标平台基础设施，它通过设备无关的MLIR抽象将线性代数程序降级至UPMEM、内容寻址存储器和交叉开关目标，但明确将各技术专用后端留给设备供应商实现。Pudding与CINM是互补关系，而非竞争关系，因为我们为PUD提供了这样的后端。由于两者都基于MLIR构建，并从线性代数层切入，我们的抽象与CINM兼容。CIM-MLC [41] 和OCC [45] 则提供了用于忆阻器交叉开关中模拟矩阵向量乘法的MLIR技术栈。

基于 E 图的等式饱和已应用于 FPGA 查找表重映射 [21] 和 ASIC 逻辑综合 [8], ，其中由成本驱动的提取优于贪心重写。然而，它们的成本模型统计查找表、逻辑门或深度，而 PUD 成本主要取决于破坏性操作数和行分配；我们的架构描述将这些因素暴露给提取过程。目前没有工具能在可重定向架构描述下，同时优化张量层级和逻辑层级。

10 结论

Pudding 是首个面向 PUD 架构的可重定向端到端编译框架，它将架构多样性视为可灵活替换的一等参数，而非固定的编译器假设。我们证明，只需替换成本函数，同一套 e-图重写与提取流程便能提升三种根本不同的 PUD 底层平台（SIMDRAM、PRADA、FCDRAM）上的程序质量，无需架构专用启发式方法。根据目标平台不同，该方法相较于解耦综合方法可将延迟最多降低 25%，相较于不进行重写可降低 4–48%。在张量层级，我们基于整数线性规划的映射、调度和位宽感知归约将内存命令减少了 11–87%，并能为每种形状和存储体数量生成有效布局；相比之下，OptiPIM 的整数线性规划在大多数 GEMM 配置和低存储体数量的 GEMV 配置下均不可行，而我们的运行速度快 1.06–1.38× 倍。将逻辑层级的逐操作成本输入张量级映射器后，生成的调度速度比解耦映射快 1。34–3.25× 倍。将编译范围从单个内核扩展到整个 PyTorch 程序后，SIFT1M 上向量搜索相较于 CPU基线的加速比翻倍，从 2.9× 提升至 5.8×。总而言之，这些结果表明，将架构描述作为各编译阶段之间的共享接口，就足以达到或超越手工优化的架构特定编译器，同时不牺牲可重定向性。

参考文献

[1] Hameeza Ahmed、Paulo C. Santos、João P. C. Lima、Rafael F. Moura、Marco A. Z. Alves、Antônio C. S. Beck 和 Luigi Carro。2019年。用于自动选择合适存内处理指令的编译器。载于《2019年欧洲设计、自动化与测试会议及展览会（DATE）》。564–569。doi:10.23919/DATE.2019.8714956[2] Luca Amarú、Pierre-Emmanuel Gaillardon 和 Giovanni De Micheli。2014年。多数反相图：一种用于高效逻辑优化的新型数据结构与算法。载于《第51届年度设计自动化会议论文集》。1–6。[3] Shaahin Angizi 和 Deliang Fan。2019年。Redram：一种用于加速大规模逐位操作的可重构存内动态随机存取存储器平台。载于《2019年 IEEE/ACM 国际计算机辅助设计会议（ICCAD）》。IEEE，1–8。[4] Alexander Brauckmann、Elizabeth Polgreen、Tobias Grosser 和 Michael FP O'Boyle。2023年。mlirsynth：利用程序综合在多级中间表示中实现自动、可重定向的程序提升。载于《2023年第32届并行架构与编译技术国际会议（PACT）》。IEEE，39–50。[5] Robert Brayton 和 Alan Mishchenko。2010年。ABC：一种学术界与工业界通用的验证工具。载于《计算机辅助验证国际会议（CAV）》。Springer，24–40。[6] Sayak Chakraborti、Paturi Varun Chowdhary、Kamalika Datta 和 Indranil Sengupta。2014年。基于二叉决策图的忆阻器布尔函数综合。载于 IDT。136–141。[7] Moses S. Charikar。2002年。基于舍入算法的相似度估计技术。载于《第三十四届 ACM 年度计算理论研讨会论文集》（加拿大魁北克省蒙特利尔）（STOC '02）。计算机协会，纽约州纽约，美国，380–388。doi:10.1145/509907.509965[8] Chen Chen、Guangyu Hu、Dongsheng Zuo、Cunxi Yu、Yuzhe Ma 和 Hongce Zhang。2024年。E-syn：采用技术感知成本函数进行逻辑综合的 e图重写。载于《第61届 ACM/IEEE 设计自动化会议论文集》。1–6。[9] Zhufei Chu、MathiasSoeken、Yinshui Xia、Lunyao Wang 和 Giovanni DeMicheli。2019年。异或多数图中的结构重写。载于《第24届亚洲及南太平洋设计自动化会议论文集》。663–668。[10] Joao Paulo C de Lima、Ben Morris、Asif Ali Khan、Jeronimo Castrillon 和 Alex K Jones。2026年。Count2multiply：可靠的存内高基数计数。载于《2026年 IEEE 高性能计算机体系结构国际研讨会（HPCA）》。IEEE，1–15。[11] João Paulo C. de Lima、Ben Morris、Asif Ali Khan、Jeronimo Castrillon 和 Alex K. Jones。2026年。Count2Multiply：可靠的存内高基数计数。载于《HPCA 论文集》。1–15。doi:10.1109/HPCA68181.2026.11408436

[12] Geraldo Francisco De Oliveira Junior、Mayank Kabra、Yuxin Guo、Kangqi Chen、Abdullah Giray Yaglikci、Melina Soysal、Mohammad Sadrosadati、Joaquin Olivares Bueno、Saugata Ghose、Juan Gómez-Luna 和 Onur Mutlu。2025年。Proteus：通过动态位精度、自适应数据表示和灵活算术实现高性能使用DRAM进行处理。载于第39届ACM国际超级计算大会论文集。473–494。[13] Arash Fayyazi、Amirhossein Esmaili 和 Massoud Pedram。2020年。HIPE-MAGIC：一种面向忆阻器辅助逻辑高度并行执行的技术感知综合与映射流程。载于ACM/IEEE国际低功耗电子与设计研讨会论文集。235–240。[14] J. A. Fisher。1981年。轨迹调度：一种全局微代码压缩技术。《IEEE计算机汇刊》（1981年）。[15] Fei Gao、Georgios Tziantzioulis 和 David Wentzlaff。2019年。ComputeDRAM：使用商用现成DRAM进行存内计算。载于第52届IEEE/ACM国际微体系结构研讨会论文集。100–113。[16] Fei Gao、Georgios Tziantzioulis 和 David Wentzlaff。2022年。FracDRAM：商用现成DRAM中的分数值。载于2022年第55届IEEE/ACM国际微体系结构研讨会（MICRO）。IEEE，885–899。[17] Deyuan Guo、Mohammadhosein Gholamrezaei、Matthew Hofmann、Ashish Venkat、Zhiru Zhang 和 Kevin Skadron。2025年。PIMsynth：面向位串行存内处理架构的统一编译器框架。《IEEE计算机体系结构快报》（2025年）。[18] Lidong Guo、Zhenhua Zhu、Tengxuan Liu、Xuefei Ning、Shiyao Li、Guohao Dai、Huazhong Yang、Wangyang Fu 和 Yu Wang。2025年。迈向基于浮点数的无注意力大语言模型：采用非均匀数据格式并减少乘法运算的混合型存内处理。载于第43届IEEE/ACM国际计算机辅助设计大会（ICCAD '24）论文集。[19] Nastaran Hajinazar、Geraldo F Oliveira、Sven Gregorio、João Dinis Ferreira、Nika Mansouri Ghiasi、Minesh Patel、Mohammed Alser、Saugata Ghose、Juan Gómez-Luna 和 Onur Mutlu。2021年。SIMDRAM：一种使用DRAM进行位串行SIMD处理的框架。载于第26届ACM编程语言与操作系统架构支持国际会议论文集。329–345。[20] Yifan He、Hyung Joon Byun、Jimin Lee、Do-Yoon Lim、Houxiang Ji、Jungmin Choi、Hoshik Kim 和 Jae-sun Seo。2026年。PAGE：协同优化使用DRAM进行处理的架构与电路，以高效加速通用矩阵-向量乘法。载于第63届ACM/IEEE设计自动化大会论文集。1–7。[21] Matthew Hofmann、Berk Gokmen 和 Zhiru Zhang。2025年。EqMap：使用e图进行FPGA查找表重新映射。载于2025年IEEE/ACM国际计算机辅助设计大会（ICCAD）。IEEE，1–9。[22] Ali JavadiAbhari、Shruti Patil、Daniel Kudrow、Jeff Heckey、Alexey Lvov、Frederic T. Chong 和 Margaret Martonosi。2014年。ScaffCC：量子计算程序编译与分析框架。载于第11届ACM计算前沿会议论文集。[23] Hervé Jégou、Matthijs Douze 和 Cordelia Schmid。2011年。用于最近邻搜索的乘积量化。《IEEE模式分析与机器智能汇刊》33, 1 2011 117 128 10 1109 2010 57（1月），–。[24] doi：。/TPAMI。。Houxiang Ji、E. Ezgi Yücel、Yaochen Li、Prajwal Ramakrishna、Derrick Quinn、Jungmin Choi、Hoshik Kim、Mohammad Alian 和 José F. Martínez。2026年。C3：协同式CPU与商用DRAM计算，实现可靠的大语言模型加速。载于第59届IEEE/ACM国际微体系结构研讨会（MICRO）（MICRO '26）论文集。[25] Asif Ali Khan、Hamid Farzaneh、Karl Friedrich Alexander Friebel、Clément Fournier、Lorenzo Chelini 和 Jeronimo Castrillon。2024年。CINM（cinnamon）：面向异构存内计算和近存计算范式的编译基础设施。载于第29届ACM编程语言与操作系统架构支持国际会议论文集，第4卷。31–46。[26] Asif Ali Khan、Fazal Hameed、Robin Bläsing、Stuart S. P. Parkin 和 Jeronimo Castrillon。2019年。ShiftsReduce：减少赛道存储器4.0中的移位操作。《ACM体系结构与代码优化汇刊》（2019年）。[27] Tatsuya Kubo、Daichi Tokuda、Tomoya Nagatani、Masayuki Usui、Lei Qu、Ting Cao 和 Shinya Takamaeda-Yamazaki。2025年。Mvdram：在未经修改的DRAM中实现通用矩阵-向量乘法执行，以加速低比特大语言模型。arXiv预印本 arXiv:2503.23817（2025年）。[28] Alexander Kusnezoff、João Paulo C de Lima、Jeronimo Castrillon 和 Asif Ali Khan。2026年。基于gem5的DRAM内计算仿真框架。arXiv预印本 arXiv:2610.08186（2026年）。[29] Chris Lattner、Mehdi Amini、Uday Bondhugula、Albert Cohen、Andy Davis、Jacques Pienaar、River Riddle、Tatiana Shpeisman、Nicolas Vasilache 和 Oleksandr Zinenko。2021年。MLIR：扩展面向领域专用计算的编译器基础设施。载于2021年IEEE/ACM代码生成与优化国际研讨会（CGO）。IEEE，2–14。[30] Orian Leitersdorf、Dean Leitersdorf、Jonathan Gal、Mor Dahan、Ronny Ronen 和 Shahar Kvatinsky。2023年。AritPIM：高吞吐量存内算术。《IEEE计算新兴主题汇刊》11卷3期（2023年），720–735。

[31] Shuangchen Li、Dimin Niu、Krishna T Malladi、Hongzhong Zheng、Bob Brennan 和 Yuan Xie。2017年。Drisa：一种基于动态随机存取存储器的可重构原位加速器。载于第50届IEEE/ACM国际微体系结构研讨会论文集。288–301。[32] Jiantao Liu、Minxuan Zhou、Yue Pan、Chien-Yi Yang、Lana Josipović 和 Tajana Rosing。2025年。OptiPIM：使用整数线性规划优化存内处理加速。载于第52届国际计算机体系结构研讨会论文集。867–883。[33] Jason Lowe-Power、Abdul Mutaal Ahmad、Ayaz Akram、Mohammad Alian、Rico Amslinger、Matteo Andreozzi、Adrià Armejach、Nils Asmussen、Brad Beckmann、Srikant Bharadwaj、Gabe Black、Gedare Bloom、Bobby R. Bruce、Daniel Rodrigues Carvalho、Jeronimo Castrillon、Lizhong Chen、Nicolas Derumigny、Stephan Diestelhorst、Wendy Elsasser、Carlos Escuin、Marjan Fariborz、Amin Farmahini-Farahani、Pouya Fotouhi、Ryan Gambord、Jayneel Gandhi、Dibakar Gope、Thomas Grass、Anthony Gutierrez、Bagus Hanindhito、Andreas Hansson、Swapnil Haria、Austin Harris、Timothy Hayes、Adrian Herrera、Matthew Horsnell、Syed Ali Raza Jafri、Radhika Jagtap、Hanhwi Jang、Reiley Jeyapaul、Timothy M. Jones、Matthias Jung、Subash Kannoth、Hamidreza Khaleghzadeh、Yuetsu Kodama、Tushar Krishna、Tommaso Marinelli、Christian Menard、Andrea Mondelli、Miquel Moreto、Tiago Mück、Omar Naji、Krishnendra Nathella、Hoa Nguyen、Nikos Nikoleris、Lena E. Olson、Marc Orr、Binh Pham、Pablo Prieto、Trivikram Reddy、Alec Roelke、Mahyar Samani、Andreas Sandberg、Javier Setoain、Boris Shingarov、Matthew D. Sinclair、Tuan Ta、Rahul Thakur、Giacomo Travaglini、Michael Upton、Nilay Vaish、Ilias Vougioukas、William Wang、Zhengrong Wang、Norbert Wehn、Christian Weis、David A. Wood、Hongil Yoon 和 Éder F. Zulian。2020年。gem5模拟器：20版。0+。arXiv:2007.03152[34] H. R. Mahdiani、A. Ahmadi、S. M. Fakhraie 和 C. Lucas。2010年。用于软计算应用高效VLSI实现的仿生不精确计算模块。《电路与系统汇刊第一部分》（2010年）。[35] Felipe S Marranghello、Vinicius Callegaro、Andre I Reis 和 Renato P Ribas。2015年。基于积之和的忆阻器IMPLY状态逻辑综合。载于2015年第33届IEEE国际计算机设计大会（ICCD）。IEEE，228–235。[36] Abdul Rafae Noor、Farzana Ahmed Siddique、Akash Kothari、Deyuan Guo、Kevin Skadron 和 Vikram Adve。2026年。PIM-AutoDSE：面向存内处理的自动化指令集架构与设计空间探索。载于第59届IEEE/ACM国际微体系结构研讨会（MICRO）（MICRO '26）论文集。[37] Geraldo F Oliveira、Ataberk Olgun、Abdullah Giray Yağlıkçı、F Nisa Bostancı、Juan Gómez-Luna、Saugata Ghose 和 Onur Mutlu。2024年。MIMDRAM：端到端使用DRAM进行处理的系统，实现高吞吐量、高能效且对程序员透明的多指令多数据计算。载于2024年IEEE高性能计算机体系结构国际研讨会（HPCA）。IEEE，186–203。[38] Geraldo F. Oliveira、Ataberk Olgun、Ismail Emir Yüksel、F. Nisa Bostancı、Pedro H. E. Becker、Mohammad Sadrosadati、Saugata Ghose、Juan Gómez-Luna 和 Onur Mutlu。2026年。PipeDRAM：一种无需数据转置并采用软硬件流水线的使用DRAM进行处理架构。载于第59届IEEE/ACM国际微体系结构研讨会（MICRO）（MICRO '26）论文集。[39] Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gimelshein、Luca Antiga、Alban Desmaison、Andreas Köpf、Edward Yang、Zach DeVito、Martin Raison、Alykhan Tejani、Sasank Chilamkurthy、Benoit Steiner、Lu Fang、Junjie Bai 和 Soumith Chintala。2019年。PyTorch：一种命令式风格的高性能深度学习库。Curran Associates Inc.，美国纽约州雷德胡克。[40] Xiangjun Peng、Yaohua Wang 和 Ming-Chang Yang。2023年。Chopper：一种使用DRAM存储器进行可编程位串行SIMD处理的编译器基础设施。载于2023年IEEE高性能计算机体系结构国际研讨会（HPCA）。IEEE，1275–1288。[41] Songyun Qu、Shixin Zhao、Bing Li、Yintao He、Xuyi Cai、Lei Zhang 和 Ying Wang。2024年。CIM-MLC：面向存内计算加速器的多级编译栈。载于第29届ACM编程语言与操作系统架构支持国际会议论文集，第2卷。185–200。[42] Vivek Seshadri、Yoongu Kim、Chris Fallin、Donghyuk Lee、Rachata Ausavarungnirun、Gennady Pekhimenko、Yixin Luo、Onur Mutlu、Phillip B Gibbons、Michael A Kozuch 和 Todd C Mowry。2013年。RowClone：快速、节能的DRAM内批量数据复制与初始化。载于第46届IEEE/ACM国际微体系结构研讨会论文集。185–197。[43] Vivek Seshadri、Donghyuk Lee、Thomas Mullins、Hasan Hassan、Amirali Boroumand、Jeremie Kim、Michael A Kozuch、Onur Mutlu、Phillip B Gibbons 和 Todd C Mowry。2017年。Ambit：使用商用DRAM技术执行批量按位运算的存内加速器。载于第50届IEEE/ACM国际微体系结构研讨会论文集。273–287。[44] Hoon Shin、Rihae Park 和 Jae W Lee。2024年。一种兼容性和可靠性增强的商用DRAM器件存内处理架构。载于第43届IEEE/ACM国际计算机辅助设计大会论文集。1–10。

[45] Adam Siemieniuk、Lorenzo Chelini、Asif Ali Khan、Jeronimo Castrillon、Andi Drebes、Henk Corporaal、Tobias Grosser 和 Martin Kong。2022年。OCC：一种用于存内计算的自动化端到端机器学习优化编译器。IEEE集成电路与系统计算机辅助设计汇刊 41, 6 (2022年), 1674–1686。doi:10.1109/TCAD.2021.3101464[46] Mathias Soeken、Heinz Riener、Winston Haaswijk、Eleonora Testa、Bruno Schmitt、Giulia Meuli、Fereshte Mozafari、Siang-Yun Lee、Alessandro Tempia Calvino、Dewmini Sudara Marakkalage 和 Giovanni De Micheli。2022年。EPFL逻辑综合库。arXiv:1805.05121v3。[47] Mathias Soeken、Saeideh Shirinzadeh、Pierre-Emmanuel Gaillardon、Luca Gaetano Amarú、Rolf Drechsler 和 Giovanni De Micheli。2016年。一种面向可编程存内逻辑架构的基于MIG的编译器。载于第53届年度设计自动化会议论文集。1–6。[48] Mohammad Arman Soleimani、Nezam Rohbani、Adrian Cristal Kestelman、Osman Unsal 和 Hamid Sarbazi-Azad。2025 年。WISEDRAM：一种可靠的 DRAM 内按位加速器。载于2025年第62届ACM/IEEE设计自动化会议（DAC）。IEEE，1–7。[49] Ross Tate、Michael Stepp、Zachary Tatlock 和 Sorin Lerner。2009年。等式饱和：一种新的优化方法。载于第36届ACM SIGPLAN-SIGACT编程语言原理研讨会论文集。264–276。[50] torch-mlir贡献者。2022年。torch-mlir。https://github.com/llvm/torch-mlir。[51] Max Willsey、Chandrakana Nandi、Yisu Remy Wang、Oliver Flatt、Zachary Tatlock 和 Pavel Panchekha。2021年。Egg：快速且可扩展的等式饱和。ACM程序设计语言汇刊 5, POPL (2021年), 1–29。[52] Ismail Emir Yuksel、Yahya Can Tugrul、F. Nisa Bostanci、Abdullah Giray Yaglikci、Ataberk Olgun、Geraldo F Oliveira、Melina Soysal、Haocong Luo、Juan Gomez Luna、Mohammad Sadrosadati 和 Onur Mutlu。2023年。PULSAR：在商用DRAM芯片上同时激活多行，以实现可靠的高性能计算。arXiv预印本 arXiv:2312.02880 (2023年)。[53] İsmail Emir Yüksel、Yahya Can Tuğrul、Ataberk Olgun、F Nisa Bostancı、A Giray Yağlıkçı、Geraldo F Oliveira、Haocong Luo、Juan Gómez-Luna、Mohammad Sadrosadati 和 Onur Mutlu。2024年。真实DRAM芯片上的函数完备布尔逻辑：实验表征与分析。载于2024年IEEE高性能计算机体系结构国际研讨会（HPCA）。IEEE，280–296。[54] Jingyao Zhang、Haoran Wang 和 Elaheh Sadredini。2026年。Meta-PIM：面向可配置存内处理的端到端编译器。IEEE计算机体系结构快报 (2026年)。[55] Ranyang Zhou、Arman Roohi、Durga Misra 和 Shaahin Angizi。2022年。Flexidram：一种支持并行通用计算的灵活DRAM内框架。载于ACM/IEEE国际低功耗电子与设计研讨会论文集。1–6。