软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(10):4768−4791 [doi: 10.13328/j.cnki.jos.007420] [CSTR: 32375.14.jos.007420]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

BIVM: 类脑计算编译框架及其原型研究*

杨 乐[1,2], 刘晓义[1,2], 李广力[3], 渠 鹏[1,2], 崔慧敏[3], 张悠慧[1,2,4]

[1](清华大学 计算机科学与技术系, 北京 100084)

[2](清华大学 北京信息科学与技术国家研究中心, 北京 100084)

[3](中国科学院 计算技术研究所, 北京 100190)

[4](中关村实验室, 北京 100094)

通信作者: 张悠慧, E-mail: zyh02@tsinghua.edu.cn

摘 要: 各类新型架构的类脑计算芯片正不断涌现, 类脑神经网络训练/学习算法和高效的生物神经网络仿真也是研究热点. 但如何在架构迥异的类脑计算芯片上优化运行计算/访存特征不同的类脑应用是关键难点, 也是建立类脑计算良好生态环境的重点, 而通用计算领域的繁荣生态已经表明, 一个灵活、可扩展、可复用的编译框架是解决这一问题的有效途径. 为此提出 BIVM, 一个类脑计算编译框架及其验证原型. BIVM 基于领域定制化体系结构 (domain specific architecture, DSA) 的多层中间表示 (multi-level intermediate representation, MLIR) 框架, 设计了为类脑神经网络定制的多层 IR, 包括脉冲神经网络方言 (高层 IR)、由 MLIR 内置方言为主组成的中间层 IR 和各类芯片的底层 IR. 针对不同类脑芯片的体系结构跨度很大且其提供的硬件功能粒度不一等问题, BIVM 充分利用 MLIR 的 progressivity 特性, 所设计的 IR 能够混合不同的抽象层次和概念 (比如混合细粒度指令与某些后端的以交叉开关结构为运算主体的粗粒度运算), 从而能够复用软件模块、简化开发; 在此基础上, 在多层 IR 的递降转换中灵活组合不同级别的编译优化方法, 包括被广泛采纳的 SNN 特定优化技术 (如计算稀疏性挖掘与时空并行度挖掘) 和适配目标硬件的底层优化技术, 以实现不同后端上的高性能. 目前, BIVM 原型支持的后端有通用处理器 (控制流架构)、具有控制流/数据流混合架构的脉冲神经网络加速芯片 (FPGA), 以及基于 ReRAM (resistive random-access memory, 阻变存储器) 的数据流架构类脑芯片 (软件仿真), 能够将智能应用与生物神经网络仿真应用优化编译为适配不同架构芯片的执行程序. 随后, 进行编译技术适配性分析与性能比较, 结果表明该类框架在编译高生产力、高可移植性、高性能方面具有良好潜力.

关键词: 类脑计算; 编译框架; 类脑计算芯片

中图法分类号: TP18

中文引用格式: 杨乐, 刘晓义, 李广力, 渠鹏, 崔慧敏, 张悠慧. BIVM: 类脑计算编译框架及其原型研究. 软件学报, 2025, 36(10): 4768–4791. http://www.jos.org.cn/1000-9825/7420.htm

英文引用格式: Yang L, Liu XY, Li GL, Qu P, Cui HM, Zhang YH. BIVM: Compilation Framework for Brain-inspired Computing and Prototype Research. Ruan Jian Xue Bao/Journal of Software, 2025, 36(10): 4768–4791 (in Chinese). http://www.jos.org.cn/1000-9825/7420.htm

BIVM: Compilation Framework for Brain-inspired Computing and Prototype Research

YANG Le[1,2], LIU Xiao-Yi[1,2], LI Guang-Li[3], QU Peng[1,2], CUI Hui-Min[3], ZHANG You-Hui[1,2,4]

[1](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

[2](Beijing National Research Center for Information Science and Technologu, Tsinghua University, Beijing 100084, China)

[3](Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China)

* 基金项目: 国家自然科学基金面上项目 (62072266); 国家自然科学基金原创探索计划 (62250006); 国家自然科学基金青年科学基金 (62202254); 北京信息科学与技术国家研究中心青年创新基金 (BNR2022RC01003)
收稿时间: 2023-05-22; 修改时间: 2023-08-01, 2024-10-24; 采用时间: 2025-02-25; jos 在线出版时间: 2025-07-23
CNKI 网络首发时间: 2025-07-23

[4](Zhongguancun Laboratory, Beijing 100094, China)

Abstract: Brain-inspired computing chips of various architectures are emerging, and the inference/training/learning algorithms of spiking neural network (SNN) and the efficient simulation of biological neural networks have become research hotspots. Meanwhile, efficiently executing applications with different computation/memory-access characteristics on various chips remains a significant challenge, which is crucial for establishing a robust brain-inspired computing ecosystem. The success of the general-purpose computing ecosystem indicates that a flexible, scalable, and reusable compiler infrastructure is an effective solution to this problem. This study proposes BIVM, a compilation framework for brain-inspired computing, along with its proof-of-concept implementation. Based on the multi-level intermediate representation (MLIR) framework of domain specific architecture (DSA), multi-layer IRs customized for SNNs are designed, including an SNN dialect, middle-layer IRs composed mainly of MLIR's inherent dialects, and the underlying IRs for various target chips. To address challenges such as the large architectural differences and varying granularity of hardware primitives in brain-inspired chips, BIVM leverages MLIR's progressivity feature. This allows for the mixing of different abstraction levels and concepts (e.g. combining fine-grained instructions with coarse-grained computation based on the crossbar structure specific to certain back-ends), enabling software module reuse and reducing compiler development costs, ultimately leading to high productivity. In addition, the framework provides flexibility to combine various levels of compilation optimizations, including widely-used SNN-specific optimizations (e.g. exploring computing sparsity and improving parallelism) and low-level optimizations tailored to different back-ends, ensuring performance portability. The current BIVM prototype supports back-ends such as general-purpose processors (control-flow architecture), SNN accelerator chips (FPGAs) with a hybrid control-/data-flow architecture, and data-flow chip designs based on ReRAM (resistive random-access memory, a widely-used neuromorphic device). It can optimize and compile deep SNN and biological neural network simulation applications into executables tailored for these chips. Comprehensive testing and performance comparisons demonstrate the potential of this compilation framework in achieving high productivity, portability, and performance.

Key words: brain-inspired computing; compilation framework; brain-inspired computing chip

类脑计算被认为是实现下一代人工智能的极具潜力的技术路线[1], 是后摩尔时代计算机体系结构重大发展方向之一[2], 对它的研究也促进了更通用的存算融合体系结构的发展[3]. 不少研究团队专注于各类新型类脑计算架构与芯片的研发, 但是如何在研制的架构/芯片上高效运行各类类脑计算应用 (包括偏向于完成 AI 任务的深度脉冲神经网络, 简称 D-SNN, 以及侧重于生物脉冲神经网络模拟的仿真计算) 是一个难点; 另一方面, 研发脑启发计算模型/算法的团队也试图在各类类脑芯片上完成高效计算——高效算力的提供及便捷使用对于算法的发展非常关键, 这已经在深度学习的发展历程中得到证实[4].

类脑计算编译软件需在此间发挥关键的"桥梁"作用, 即将各类类脑计算应用转变为能高效驱动架构迥异的类脑计算芯片的可执行形式. 这也利于将各领域研究力量联合起来[5], 形成良好的研发生态, 比如将神经学领域的新发现应用到 AI 领域, 或者将最新的 (类脑) 计算芯片广泛高效地应用起来, 帮助新算法快速进化与应用.

据我们所知, 目前面向各类类脑计算芯片的通用编译框架还是空白——类脑计算编译框架是一个不针对特定硬件的编译基础设施 (compiler infrastructure), 通过灵活的多层次内部架构、中间表示以及接口, 可以扩展支持多种多样的类脑计算应用以及类脑计算芯片.

这与现有的种种针对特定类脑芯片的编译软件是不同的, 后者能够在目标体系结构上获得良好性能, 但各软件模块与层次间接口是面向目标芯片的, 使得软件模块的可重用性 (reusability)、工具链间的互操作性 (interoperability) 与可组合性 (composability) 受到限制, 而且开发新型芯片编译器的成本较高.

编译框架的服务对象主要是类脑芯片和类脑应用开发工具研发人员. 目前, 已经出现了不少面向应用/模型研发人员的类脑计算应用开发工具, 如 BindsNET[6]、Norse[7]、SpykeTorch[8]、SpikingJelly[9]等. 理想情况下, 只要将类脑计算芯片的软硬件接口或功能库与编译框架的下层表示相适配, 就可以支持各类类脑应用; 而开发工具研发人员对该编译框架进行前端适配也将有利于扩大开发工具的硬件支持范围, 从而在各类类脑计算应用与类脑芯片间建起桥梁: 一个高层次的、硬件无关的神经网络应用程序只需编写一次, 就能被编译为不同芯片上的可执行程序 (可移植性, portability), 并且具有较高的执行效率 (性能, performance), 同时该编译框架可以灵活扩展或复用, 以支持多种类型的类脑芯片和编译优化技术 (生产力, productivity). 这一方法论的有效性已经在计算机发展历史上得到了证明: 通用计算领域, 基于处理器开源编译框架 gcc/LLVM 的应用生态蓬勃发展, 极大地降低了新型处理器

芯片的工具链开发门槛, 从而促进其应用. 近年来广泛流行的深度学习前端开发工具 (如 PyTorch、TensorFlow) 与编译后端 (如 TVM) 也是成功的佐证. 这类开放性编译框架的益处还在于, 利于汲取编译领域的技术积累, 利于吸引开源社区的研发力量.

类脑领域这一类实现的理论可行性已经被文献 [10] 所证明. 其研发必要性与技术可行性还在于以下两点: 第一, 类脑计算的主要计算范式——脉冲神经网络 (SNN) 展示出了与深度神经网络不同的计算、访存特征, 同时类脑计算芯片体系结构的跨度很大, 这就提供了较大的编译优化空间, 也凸显其设计难度; 第二, 旨在尽力解决 DSA (domain specific architecture, 特定领域架构) 引起的软硬件碎片化问题的编译基础设施 MLIR (multi-level intermediate representation, 多层中间表示) 正在兴起, 提供了可重复利用的编译技术资源与框架. 我们将在第 3 节对此进行详细分析.

总之, 研究界[10−13]已认识到, 类脑计算“通用”编译至关重要, 但缺乏将计算机系统领域编译基础设施的最新进展与类脑领域一般需求结合起来的工作. 本文在这方面做出尝试, 提出基于 MLIR 的类脑计算编译框架 BIVM 及其验证原型: 当前重点支持脉冲神经网络应用, 展示 BIVM 在不同芯片上支持各种典型 SNN 应用 (从 D-SNN 到生物神经网络仿真) 的编译和优化能力, 芯片类型包括通用处理器、控制流/数据流混合架构的 SNN FPGA 芯片[14]以及基于 ReRAM (resistive random-access memory, 阻变存储器) 的数据流类脑芯片[15]. 具体贡献如下.

(1) 设计提出类脑计算编译框架的多层 IR (在本文中，使用 MLIR 指代 MLIR 项目，多层 IR 表示本文提出的多层 IR 设计). 包括脉冲神经网络方言 (高层 IR)、由 MLIR 内置方言为主组成的中间层 IR 和针对典型类脑芯片架构的多种底层 IR; 所设计的 IR 能够混合不同的抽象层次和概念, 如混合细粒度指令与某些后端的以交叉开关结构为运算主体的粗粒度运算, 以便复用软件模块、简化开发.

(2) 设计实现多层 IR 间的优化递降转换. 针对 SNN 计算/访存特征, 递降过程支持 SNN 计算稀疏性挖掘、神经元组计算合并/向量化等高层优化技术, 以及适配目标硬件的底层优化技术, 实现不同后端上的高性能.

(3) 实现 BIVM 原型, 支持 3 类芯片后端并完成测试与比较. 测试表明上述优化技术能够有效提升编译性能, 实现比目前广泛采用的开发框架更快的性能 (针对处理器芯片), 或者在目标芯片上达到比其原有工具链更高 (或者相近) 的性能.

未来将扩展该原型, 支持更多的典型类脑计算应用以及更多种类的类脑芯片.

本文第 1 节是研究背景介绍. 第 2 节给出类脑计算编译框架所面临的挑战, 以及基于 MLIR 的总体方案, 包括各层 IR 的设计思路与编译优化技术的概览. 第 3 节则给出各层 IR 的具体设计思路与优化技术细节. 第 4 节是 BIVM 量化测试. 最后是下一步工作与总结全文.

1 类脑神经网络计算特征及其与深度神经网络的差异

1.1 SNN 数值计算形式介绍

类脑计算中应用最广泛的计算模型是脉冲神经网络 (SNN), 其主要计算过程可描述如下: 首先是神经元计算, 求解一个关于神经元膜电位的动力学方程来获取当前膜电位值, 如果膜电位达到阈值, 神经元会发出脉冲信号同时神经元自身进入不应期, 这期间内膜电位会停留在复位电压; 接着是脉冲传输, 即发出的脉冲将根据网络连接传播到它们的目标突触; 最后, 接收到脉冲信号的突触将根据突触模型更新其目标神经元的内部状态.

神经元模型定义了膜电位和其他内部状态的动态, 分为很多种类[16], 例如 LIF (leaky-integrate-and-fire)[17]、Izhikevich[18]、Hodgkin-Huxley[19]等. 不同的模型提供不同级别的生物真实性, 在计算复杂度上也差别很大.

以类脑计算领域最为常用的 LIF 模型为例, 典型的 LIF 模型可以用以下方程表示. 其中, $V_i(t)$ 是神经元 i 在时间 t 的膜电位, $S_i(t)$ 是一个二值变量指示神经元 i 是否在时间 t 发放脉冲信号, w_{ji} 是连接源神经元 j 和目标神经元 i 的权重, λ 和 V_{leak} 是与漏电流相关的常数.

$$V_i(t) = V_i(t-1)+\lambda[V_i(t-1) - V_{\text{leak}}] + \sum_j w_{ji} S_j(t),$$

$$\text{if } V_i(t) > V_{\text{threshold}} \text{ then } S_i(t) = 1,\ V_i(t) = V_{\text{reset}} \text{ else } S_i(t) = 0.$$

针对这些模型, 多数类脑计算软硬件使用了前向欧拉方法[20]来提供数值解. 该方法相对简单, 精度也在可接受范围内. 一些软件模拟器则使用更复杂的方法, 例如后向欧拉和 Crank-Nicholson 方法[20]来提供更准确和稳定的数值解. 与深度神经网络 (DNN) 的神经元计算相比, 求解膜电位的动态方程 (包括计算输入电流) 计算过程相对复杂且耗时.

1.2 SNN 的计算特点

图 1 展示了 SNN 的计算形式. SNN 的神经元组代表类型与功能相似的一组神经元, 类似于 DNN 的层 (layer); 神经元的状态会根据膜电位方程进行不断迭代更新; SNN 的突触组代表功能类似的一组突触, 单向连接两个神经元组.

```
1  neuron_upd(neurons):              # 神经元更新过程
2    for neu in neurons:             # 访问所有神经元
3      neu.lif_upd()                 # 神经元根据动力学方程更新内部状态
4    return firer_state(neurons)     # 返回神经元的发放状态
5                                    #  (0 代表不发放 / 1 代表发放)
6
7  spike_prop(fire_state, synapses): # 脉冲传递过程
8    for neu in index(fire_state):   # 访问所有发放脉冲的神经元
9      for syn in synapses(neu):     # 获取与之相连的突触
10       dst_neu = dst(syn)          # 获取突触的目标神经元
11       dst_neu.spike_upd(syn)      # 根据突触模型计算具体的影响
12
13 simulation(network):               # 整体的模拟过程
14   neurons, synapses = network.state()  # 获取 SNN 的神经元组与突触组
15   for t in network.T:              # 按照时间步进行迭代
16     fire_state = neuron_udp(neurons)   # 神经元更新, 获取神经元的脉冲发放
17     spike_prop(fire_state, synapses)   # 根据神经元脉冲发放计算传播后的影响
```

图 1 SNN 计算过程伪代码

与 DNN 相比, SNN 的不同计算特征体现在以下两部分.

(1) 脉冲传播的稀疏性

对于目标神经元而言, 脉冲传播过程需要累加其每一个突触前发放神经元对其自身的状态影响 (见图 1 中 spike_prop 函数), 该计算过程中的两个输入, 发放状态和连接矩阵都具有稀疏特性. 具体而言, 发放状态由 0/1 构成, 指示神经元是否发放. 生物神经网络神经元的放电速率通常低于 100 Hz, 甚至只有几赫兹或更低[21], 而数值模拟方法内的一个时间步通常为 0.1 ms, 故平均一个时间步内产生脉冲的神经元不超过 1% (稀疏度大于 99%, 见表 1, 其中 SLAYER 与 ANTLR 是两种典型的基于误差反向传播的 SNN 训练方法; NMNIST 或 MNIST 是其对应的手写数字分类任务及 MLP 网络, 而 N-CARS 是对应的汽车分类任务及卷积网络, 均是 SNN 训练算法研究中经常被作为应用示例的图像分类任务). 发放状态的非零元数量和位置会根据输入和网络的情况随时间步而变化, 可以视作随机分布. D-SNN 的构造特性偏向于 DNN (不论是其网络拓扑还是训练方法), 因此整体发放率会高些, 一般在 10% 以下, 且由 ANN 转化而来的发放率要比经过训练直接得到的发放率更高.

连接矩阵代表着神经元组之间的突触连接, 生物网络的稀疏性会带来连接矩阵的稀疏性. 连接矩阵的非零元不会随计算过程发生变化, 故一般以稀疏格式进行存储; 稀疏格式会使计算过程的访存变得不规则. 计算神经学应用中生物突触连接密集程度一般在 20% 以下, 连接矩阵较稀疏, 而 D-SNN 应用往往使用全连接或卷积层, 密集程度较高或者连接较为规则.

表 1 不同应用的脉冲发放率

SNN学习算法	SNN应用	神经元发放率 (%)
SLAYER[22]	NMNIST	0.53–6.97
SLAYER	N-CARS	⩽5
ANTLR[23]	MNIST	0.04–0.11

(2) 以时间维度划分的计算流程

神经元膜电位的动力学方程难以直接求解, 在计算中一般使用前向欧拉方法提供数值解, 通过对连续的时间进行离散采样, 不断迭代时间而计算膜电位值. 实际计算时一般每隔一小段物理时间 (也称作时间步) Δt 进行采样, 神经元的膜电位跟随时间步不断更新.

神经元发出的脉冲会通过突触对后续的神经元产生影响, 而突触传播的延迟使得此影响可能要在若干时间步后才被施加. 同时, 生物突触可塑性的调节往往与时序信息相关. 比如, 被广泛接受的脉冲时序依赖可塑性规则 (简称为 STDP 规则) 就指出, 突触会随着突触前后神经元的发放相关性来调整自身权值, 两者产生脉冲发放的时间差影响着权值的调整幅度. 综上所述, SNN 中一般以时间维度来迭代动力学方程, 同时时间的绝对值也影响网络的内部活动, 故一般会按照时间维度划分计算流程.

在以时间维度划分的计算流程中, 神经元需要更新自身状态, 并计算其对后续突触的影响, 这种影响一般需要每个时间步进行同步以确保网络活动的正确性. 由于更新和交换数据量较大, 所以访存可能会占据整个计算过程的主体. 为优化访存, 可通过合并同类型神经元/突触进行并行计算提高空间并行度, 或多个时间步后进行集体同步以减少数据的整体交换.

DNN 中的层既作为功能主体, 又通过构造出的计算图设定了计算的先后顺序. 即使是 RNN、LSTM 这种引入时间维度的网络, 在处理中也是沿着时间维度展开网络, 依次计算隐藏状态, 因此整体上还是遵循层级划分的逻辑. 需要指出的是, D-SNN 的构造特性通常偏向于 DNN, 所以也可以按照层级划分的方式进行计算, 其突触间传播延迟为 0, 时间只作为一个循环维度. 此类网络可使用针对 DNN 的优化方法, 具体策略本文不作过多讨论.

1.3 不同架构的类脑芯片

类脑芯片的架构跨度很大, 从传统冯诺依曼架构到基于新兴神经形态器件的数据流架构, 并且两端之间有各种混合架构; 共同特征是高效支持 SNN 这一计算范式, 且通常在架构设计上体现了不同层次的存算融合特点.

属于冯诺依曼架构的类脑芯片是 SpiNNaker[24], 主要依赖于众多的 ARM 核进行软件形式的神经元计算, 但其脉冲信号传输是由定制硬件支持的. 此外, 类脑软件的开发与功能测试 (在部署前) 通常也是依赖于传统通用处理器 (包括 GPGPU) 进行的.

基于新兴神经形态器件的芯片以模拟电路形式完成突触处理 (如, 针对神经元间脉冲信号与突触权重的基于交叉开关结构的点积运算) 与/或神经元计算, 并因为具有存算合一特点, 可以将其抽象为一种数据流架构 CGRA (tile-based coarse-grained reconfigurable array). 此外, 采用类交叉开关结构, 但基于数字电路实现的也可归为这一类, 如 IBM 的 TrueNorth[25].

还有混合架构, 如 Tianjic 芯片[26]、基于 FPGA 的 GaBAN[14]、Intel 的 Loihi[27]等. 比如, GaBAN 是数据流+控制流架构, 即采用简单众核, 每核都支持其设计的类脑计算指令集以实现具体的神经基元计算, 核间任务调度与驱动则采用数据流机制.

上述众多的类脑芯片一般都附带有自身的工具链软件, 其核心部分就是编译器. 编译器是类脑计算基础软件的关键组成部分之一, 但目前主要是为目标芯片定制设计的.

1.4 现有类脑计算基础软件

综述文献 [28] 总结了这一领域的一些情况, 其将类脑计算基本软件主要分为 3 种: 类脑芯片的编译器、专用的类脑神经网络开发与模拟工具, 以及在现有 DNN 开发框架基础上扩展支持 SNN 的开发与模拟工具.

需要指出的是 Intel 于 2021 年底发布的 Lava (https://lava-nc.org/) 框架. Lava 将 SNN 应用抽象为基于 CSP

(通信顺序过程) 的计算模型, 希望不同类型的类脑芯片能够执行 CSP 模型; 目前 Lava 提供了支持 CPU/GPU 的 Magma 库, 以及支持 Loihi 芯片的 Magma and Process 库. 但是, 没有针对不同硬件进行优化编译的统一编译框架与层次结构, 而后者是本研究的重点, 因此两者在很大程度上是互补的.

此外, 有的工作注重于针对神经形态芯片的硬件约束条件 (如资源规模等), 研究如何将 SNN 高效地映射到硬件资源上, 比如文献 [29] 采用同步数据流图表示 SNN, 在考虑可用的计算和存储容量、缓冲区大小和通信带宽等的前提下, 优化地将 SNN 分区并部署到以交叉开关结构为主体的类脑计算硬件上. 此外, OCC[30]是一个面向存算一体的端到端编译框架, 通过在 MLIR 中定义存算一体的读写方言来支持可读写存算一体的编译并面向存算一体的特定优化步骤; 但这一工作不涉及脉冲神经网络的支持.

再者, 还有一些基础性软件工具的用途是探索针对 SNN 的芯片体系结构优化, 如 NeuroXplorer[31], 或者是自动生成神经网络加速器的可综合代码的研究, 如 SODA[32].

最近 SODA 的扩展工作 [33]设计了一种新的 MLIR 方言以表达 SNN, 并通过该方言支持脉冲神经元到加速芯片设计的映射, 而后者是通过 SODA 自动综合生成的. 除了应用目标不同, 本工作与之区别有 3 点: 第一, 所提出的 SNN 方言支持更多的类脑应用类型, 而后者主要支持 D-SNN; 第二, 本工作支持 SNN 编译优化, 而后者不涉及; 第三, 本工作支持“广谱”的类脑芯片架构, 而后者针对的是基于新型非易失性器件的数据流架构芯片.

2 挑战与解决思路

2.1 类脑计算编译框架面临的挑战

编译器是连接类脑计算应用程序和算力提供芯片的基础软件, 能够将跨学科领域的应用程序优化转换为可以高效驱动目标芯片的可执行形式. 目前, SNN (脉冲神经网络) 是该领域的主要计算范式, 其也显示出与 DNN 融合的发展趋势.

软硬件解耦的编译框架不仅可以降低特定编译器的开发成本, 复用编译技术资源, 还可以帮助消除软件碎片化现象、提高软件质量. 更重要的是, 它还可以促进跨域资源交流——这是类脑计算发展的一大推动力量. 现在, 出现了不少前端工具来支持广泛的类脑计算应用程序的开发, 但是对作为后端的编译框架的研究非常有限. 通过第 1.2 节的分析, 可以看到具体挑战包括两大方面.

第一, 应用计算特征跨度大. 从计算与数据访问特征来看, 其跨度大且复杂——从计算密集和数据访问 (相对) 规则的深度脉冲神经网络 (DSNN), 到以稀疏事件触发计算为特征的应用 (如, 源自神经科学的生物神经网络的仿真), 且后者丰富的时空神经动力学特性使得这一计算模型有更大的优化空间[34].

第二, 芯片体系结构跨度大. 由于类脑计算芯片目前还没有被广泛接受的技术路线, 因此微架构的跨度也很大, 从技术成熟的传统冯诺依曼架构到基于新兴神经形态器件的数据流架构, 并且两端之间有各种混合架构, 如天机芯片 [26]、Loihi[27]、GaBAN[14]等. 而且, 芯片对外提供的硬件原语的粒度迥异, 如基于传统冯诺依曼架构的 SpiNNaker[24]芯片以 ARM 软核为计算核心, 提供传统的通用指令集作为细粒度的软硬件接口, 而很多类脑芯片 (TrueNorth[25]、BrainScaleS[35]等) 将神经元模型粒度的硬件功能暴露出来作为粗粒度接口, 还有基于新兴非易失性器件的存算一体芯片以交叉开关结构 (crossbar) 为运算主体, 即直接将一定规模的点积计算作为硬件原语. 因此, 我们定义“细粒度”原语以传统的通用指令集中的指令 (或者类似粒度) 为功能原语, 比如 CPU 或者 ReRAM 类脑芯片中的控制核提供的指令; “粗粒度”则是某些类脑芯片提供的以神经基元模型为基本功能单元的硬件原语, 或者某些存算一体芯片的以交叉开关结构为突触运算主体的硬件原语. 这样就需要在这些不同架构的芯片上实现具有较高资源复用度的统一编译框架以提高开发效率 (productivity), 同时需要能够灵活组合不同级别的编译优化方法以针对不同应用实现不同后端上的高性能 (performance portability).

2.2 基于 MLIR 的设计思路

幸运的是, 新兴的面向 DSA 的 MLIR 编译基础设施有助于解决上述挑战和实现具有 P3 特性的编译, 即可移

植性 (portability)、(高) 生产力 (productivity) 和 (高) 性能 (performance).

MLIR 是著名的通用计算编译器基础设施 LLVM 的子项目, 被称为后摩尔时代的编译器基础设施[36], 提供一系列可复用、易扩展的基础组件, 包括中间表示 (IR) 的规范, 并提供一个可扩展的编译框架来进行 IR 的递降 (lowering).

相较于 LLVM, MLIR 关键特色之一是 progressivity, 即能够通过多层 IR 将高层表示逐步递降 (progressive lowering) 和优化为针对各种架构的低级抽象, 而且支持在同一层 IR 中混合不同的抽象层次和概念. BIVM 充分利用 MLIR 上述特点开展设计, 以应对第 2.1 节提到的挑战. 具体来说, 设计了从高到低 3 个层级的 IR, 并融合到 MLIR 中, 具体如图 2 所示.

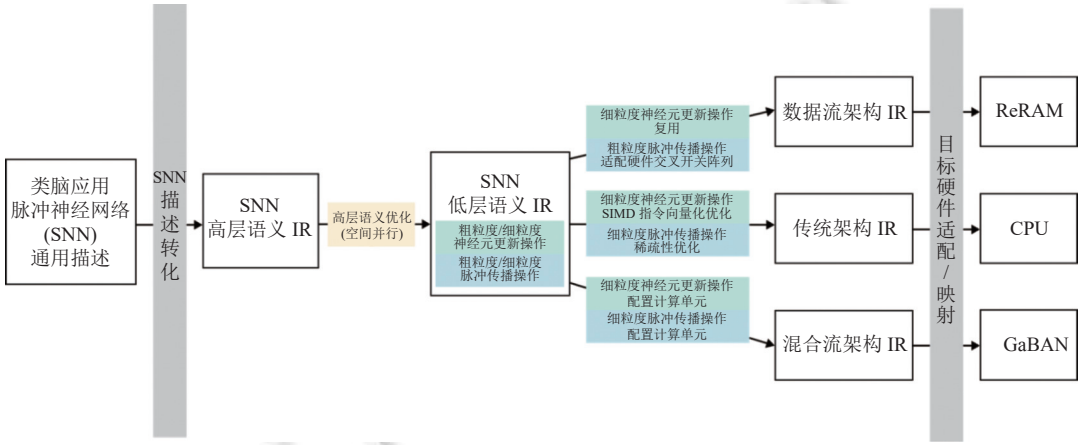

图 2 BIVM 整体示意图

第 1 层 IR 主要由我们提出的 SNN 方言组成, 包含 SNN 应用相关的数据类型 (data types)、属性 (attributes) 和操作 (operations). 方言 (dialect) 是 MLIR 的重要组成部分, 每类方言可以定义特定于编程语言、领域专用语言 (domain specific language, DSL) 或硬件目标的新操作和类型.

第 1 层 IR 强调 SNN 中神经元组 (neuron population)、突触组 (synapse projection) 的主体性以及对应的语义, 描述 SNN 神经元更新和脉冲传播计算流程及其时空并行特性. 这样在递降到下一层 IR 的过程中可针对这些性质引入典型的 SNN 计算、数据布局和调度优化技术, 并且不影响其他优化, 例如, 在时间维度上并行化 SNN 计算中的时间步, 或在空间维度上合并相似神经元或突触结构 (目前采用的面向 SNN 或者类脑后端的编译优化技术的归纳请见表 2).

此外, 由于脉冲张量稀疏性等特性需要反映到硬件相关的存储计算过程中, 所以在第 1 层 IR 中声明的这些属性需要在递降中被传播到下一层 IR (具体见第 3 节). 对应的, SNN 方言引入一类特殊的数据类型脉冲张量 (spike tensor), 被用于描述 SNN 计算过程中产生的神经元发放情况, 它们只包含 0/1 元素并具有稀疏性.

第 2 层 IR 将 SNN 的高层语义转化为低层语义, 对第 1 层 IR 中 SNN 的各个主体进行存储、计算以及控制流方面的递降, 特色在于可面向迥异的类脑处理器架构选择不同粒度的抽象.

比如, 针对提供细粒度软硬件接口的后端硬件, 可以在第 2 层 IR 中递降为细粒度描述, 即使用 MLIR 的 memref、scf、sparse tensor 和 linalg 等方言来描述神经元和脉冲传播操作的内部访存、计算过程, 从而更容易转变为底层抽象; 而针对粗粒度接口 (如神经元模型粒度的接口, 和以交叉开关结构为运算主体的接口), 则可以维持粗粒度描述.

这样, 针对不同后端架构的计算特性, 第 2 层 IR 可灵活地组合不同粒度的抽象. 如基于 ReRAM 的类脑计算架构因使用交叉开关阵列来加速脉冲传播的计算, 故可使用粗粒度的脉冲传播操作来描述, 而将神经元更新操作递降为细粒度描述. 这种设计方式符合 MLIR 中 progressivity 的设计思路, 使得不同级别的软件模块可轻松复用, 并可以针对不同后端架构灵活选择与组合编译优化方法 (如在我们设计的第 2 层 IR 向第 3 层递降过程中, 可以针

对不同硬件选择不同优化策略). 对于一个新的后端架构, 将适配过程尽量变为各类方言与编译优化方法的选取、定制与融合过程, 简化开发过程.

第 3 层 IR 是针对特定目标硬件的 IR, 如针对通用芯片的 LLVM IR 或者对应某类具体硬件的 SNN 底层 IR. 我们也针对部分后端提出专属方言, 承接第 2 层 IR 中的计算描述过程, 并将神经网络计算过程映射到目标硬件上 (支持调用目标硬件的运行时库接口或者其原生编译器等方式), 以展示在高生产力和高可移植性方面的潜力.

表 2　3 层 IR 优化示意

层级	存储优化	并行计算优化	其他优化
第1层IR	利用SNN发放状态全为0/1特性, 设计脉冲向量类型, 利用稀疏性节省空间	—	—
第1层IR → 第2层IR	利用SNN计算的时空并行特性, 在空间维度上合并相似神经元/突触结构, 提高数据局部性, 并利于后续提高计算并行度		—
第2层IR	面向不同的类脑处理器架构选择不同粒度的抽象, 以便在第3层递降过程中, 可以针对不同硬件选择不同优化策略		
第2层IR → 第3层IR	面向CPU后端, 仅保存发放状态的下标, 将连接矩阵保存为稀疏格式节省空间	利用SNN并行特性: (1) 面向CPU后端, 采用单指令流多数据流 (SIMD)方式并行化神经元更新与脉冲传播计算 (2) 面向GaBAN后端, 转化为对应的类脑计算指令集	(1) 面向CPU后端, 利用SNN稀疏特性优化脉冲传播计算, 使用稀疏格式索引发放状态与连接矩阵, 减少不必要运算 (2) 面向ReRAM后端, 将脉冲传播操作转化为交叉开关阵列对应的指令
第3层IR	(1) 面向CPU后端, 采用通用优化 (如常量传播、无用代码消除) (2) 面向ReRAM后端, 采用精度微调优化, 使用较低精度计算也能保持任务的高精度结果 (3) 面向GaBAN后端, 采用向量指令优化计算过程		

3　各层 IR 设计与示例

本节首先给出 BIVM 的各层 IR 设计以及各层所体现出的针对 SNN 计算特点的编译优化技术, 并以一个经典的生物神经网络为示例, 来说明每一层 IR 的设计和递降过程. 该示例网络为 Brunel[37], 用于表示一类随机连接的生物网络结构, 常被用作生物神经网络仿真的基准测试之一. 该网络包含 10 000 个 LIF 神经元, 被分割为 1:4 两部分, 之间以 10% 的概率使用静态突触相互连接.

特别的, 在充分复用 MLIR 框架及其已有方言的前提下, 针对 3 种不同类型的类脑计算后端硬件, 我们设计了不同的底层方言, 来体现这类基于编译框架的开发方法的便捷性.

3.1　应用描述层

应用描述层是类脑计算编译框架的输入, 为用户提供 Python 函数接口来声明 SNN 神经网络的结构和计算流程, 其设计可以是灵活多样的. 目前我们参照 PyNN[38]的 Python 接口设计该层——PyNN 是一种与模拟器无关的 SNN 通用编程与声明接口, 可以确保该层完备、易移植且为领域开发人员所熟悉.

声明包含以下几部分: (1) 神经元组的数量、类型、内部变量的初始值; (2) 突触组的数量、类型、内部变量的初始值, 以及神经元组的网络连接方式; (3) 神经元更新和脉冲传播的计算流程; (4) 对神经元组、突触组、神经元更新以及脉冲传播的属性描述, 这些附加属性会影响后续递降过程中的具体编译优化过程, 如连接矩阵或者发放状态的存储格式会影响脉冲传播操作的具体实现. Brunel 神经网络的结构和计算流程如后文图 3(a) 所示. 将应用描述层中的声明和 MLIR 操作进行匹配, 可以得到 MLIR 中第 1 层的 IR 表示.

3.2　第 1 层 IR

第 1 层 IR 主要由我们提出的 SNN 方言构成, 包含 SNN 应用相关的操作 (operations)、属性 (attributes) 和数据类型 (data types), 主要描述神经元组、突触组、神经元更新过程、脉冲传播流程 (见图 1 中的 neuron_upd 和 spike_prop 函数) 等 SNN 高层概念 (见表 3; 方言的形式化描述请见附录 A 中的表 A1). 第 1 层 IR 中的变量与操作具有 SNN 高层语义, 在描述以及递降为下一层 IR 的过程中可以利用语义进行编译优化.

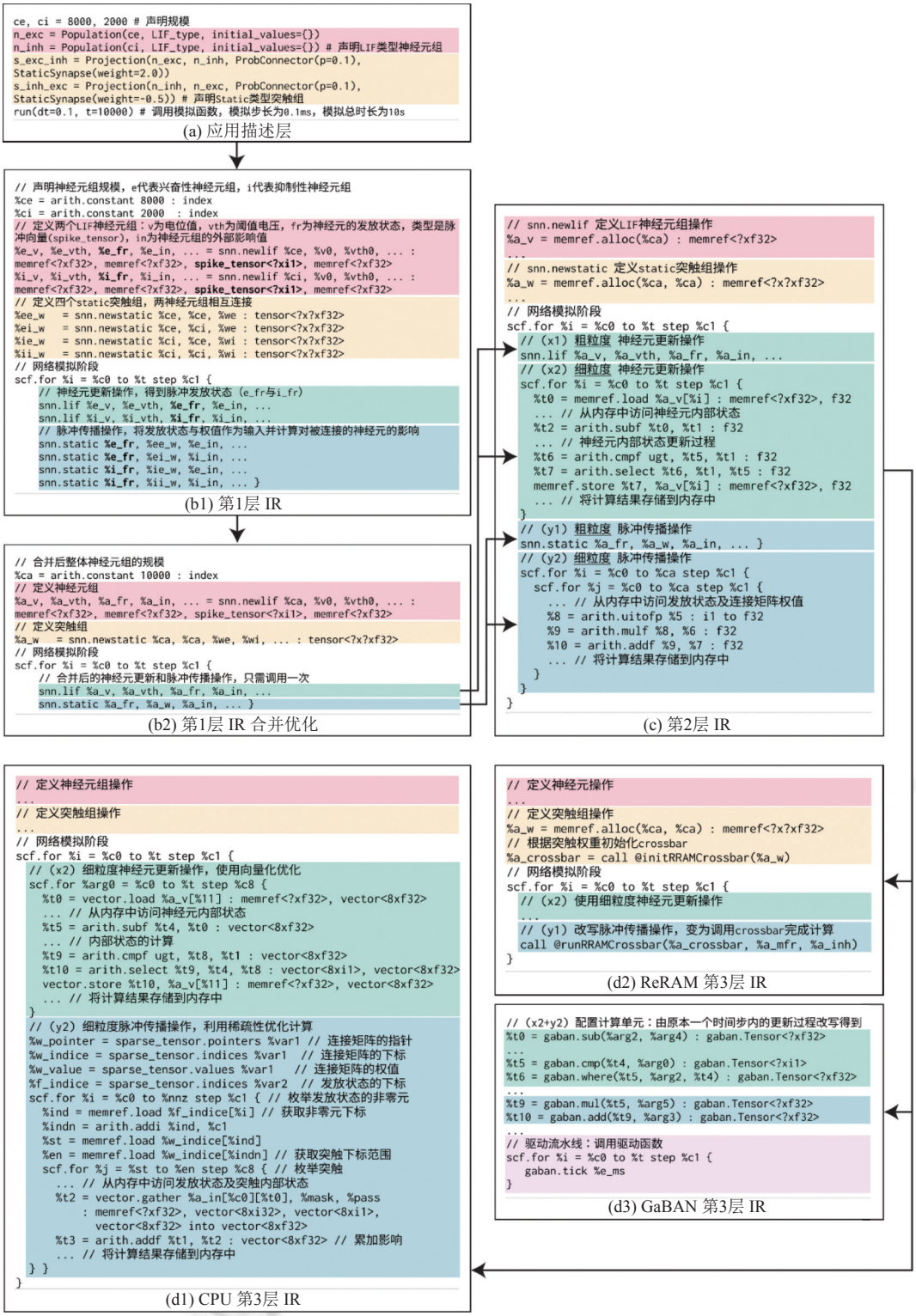

图 3 BIVM 的 3 层 IR 递降示意

表 3 SNN 方言中的操作、数据类型与属性

内容	方言命令	说明
神经元/突触定义操作	snn.newlif	定义LIF神经元组
	snn.newstatic	定义静态突触组
	snn.newizhikevich	定义Izhikevich神经元组
	snn.newpoisson	定义Poisson神经元组
神经元/突触计算操作	snn.lif	LIF神经元更新
	snn.static	静态突触脉冲传播
	snn.izhikevich	Izhikevich神经元更新
	snn.poisson	Poisson神经元更新
数据类型	spike_tensor	脉冲张量类型
属性	sparse_tensor encoding	稀疏矩阵格式

● SNN 操作

SNN 目前已经实现的操作主要包含神经元/突触定义操作和神经元/突触计算操作两类. 前者负责定义与初始化各类型的神经元组和突触组, 在网络定义阶段使用; 后者描述了各类型的神经元组的更新与突触组的脉冲传播过程, 在网络模拟的阶段被调用. 目前支持的神经元类型包括 LIF、Izhikevich 与 Poisson (泊松编码神经元), 突触类型包括静态突触; 未来可支持更多模型. Brunel 神经网络的第 1 层 IR 描述如图 3(b1) 所示.

使用这类粒度较大的描述的原因在于, 虽然神经元的内部状态可以拆分为若干独立的向量, 内部更新的计算过程也可以拆分成若干 element-wise 的向量计算, 但这种细粒度的描述方法存在如下问题: 一方面, 这种描述会丢失掉部分 SNN 的语义, 例如在某类型神经元当中, 某些状态的取值范围较为有限, 或是某些计算过程精度要求不高. 使用细粒度的描述会使得这种语义信息难以精确附加到操作上. 另一方面, 部分类脑硬件 (如 Tianjic[26]、基于 FPGA 的 SNN 芯片[39,40]等) 就提供了这类神经元或突触级别的硬件接口, 粗粒度的描述在后续递降过程中会较为方便.

● 脉冲张量数据类型

作为 SNN 计算模型中最重要的一类语义, 我们提出一种新的数据类型, 脉冲张量 (spike tensor) 以描述 SNN 的发放状态. 神经元的发放状态是 SNN 计算流程的重要中间计算结果 (如同 DNN 中的 feature maps), 其由神经元更新过程产生: 在某一时刻中, 如果神经元的膜电位大于阈值电压, 则会发放脉冲, 否则不发放, 一般使用 0/1 来指代神经元的发放状态; 在一段时间内一组神经元的发放状态可以表示为由 0/1 构成的张量, 记作脉冲张量. 特别地, 一个神经元在一段时间内的发放状态, 可以称作脉冲序列.

神经元的稀疏发放使得发放状态具有稀疏性. 稀疏性一方面影响脉冲传播阶段的计算量, 另一方面由于发放状态只由 0/1 构成, 使得其存储、传输和传统的 DNN 不同. 综上, 我们使用了脉冲张量数据类型格式来表示发放状态.

目前脉冲张量类型只参与 SNN 相关的计算过程, 未来将围绕该类型实现一系列操作以融入 MLIR 方言生态, 包括脉冲张量和稠密张量相互转换操作、脉冲张量的下标提取操作等.

● SNN 操作属性

SNN 方言中还包含了若干从应用描述层中传递得到的属性, 这些属性描述了 SNN 中神经元或突触的性质、或指定了在递降过程中对神经元更新及脉冲传播操作的编译优化行为. 例如, 可以通过附加稀疏张量属性来指定突触中权值采用某类稀疏格式进行存储, 并基于此优化脉冲传播操作计算过程 (见第 3.4.1 节); 或通过属性来指定第 2 层 IR 中各个操作的粒度, 从而符合后端的硬件计算特征 (见第 3.3 节).

● 第 1 层 IR 递降中的编译优化

第 1 层 IR 中可以加入部分针对 SNN 语义的编译优化, 例如合并同类型神经元组以便更好地挖掘其并行性[41]: 以时间进行划分的计算流程需要更新每个神经元组的状态, 并且这些更新是可以并行处理的, 只需要保证它们在

脉冲传播阶段前完成即可. 这样相同类型的神经元组更新可以并行处理, 而由下面的 IR 层级决定具体并行化策略; 同样可以对脉冲传播阶段做类似并行化处理. 此类优化与后续的优化措施是相互独立的.

具体实例请见图 3(b2): 将第 1 层 IR (图 3(b1)) 中的相同类型的神经元组 (2 个 snn.newlif 声明) 和突触组 (4 个 snn.newstatic 声明) 分别合并为规模更大的单一神经元组和单一突触组, 即图 3(b2) 中的 snn.newlif 与 snn.newstatic, 并且在计算阶段只需调用一次计算过程; 这给予了下面 IR 层级更大优化空间.

3.3 第 2 层 IR

针对不同后端架构的计算特性, 第 2 层 IR 可灵活地组合不同粒度的计算过程描述. 在第 1 层 IR 到第 2 层 IR 的递降过程中, 可以选择沿用第 1 层 IR 中的粗粒度描述方式, 或是使用更低级的 MLIR 方言对各个操作的存储、计算以及控制流进行详细描述. 粗粒度的 IR 描述了操作的类型与其输入输出变量, 这种表示适用于 TrueNorth、BrainScaleS 等类脑芯片, 它们将神经元模型粒度的硬件功能暴露出来作为粗粒度接口. 细粒度的 IR 则描述了操作内部的计算过程, 通过 MLIR 中内置的 memref、scf、arith 和 linalg 等方言对计算过程进行描述, 从而适合如 CPU、SpiNNaker 等采用通用指令集的架构. 组合不同粒度的 IR 描述可以发挥 MLIR 中 progressivity 的特性, 提高复用与开发效率, 并使得底层优化技术可自由选用, 实现不同后端上的高性能.

例如, 数据流 ReRAM 后端可使用交叉开关阵列对脉冲传播计算进行加速, 所以其脉冲传播操作在第 2 层 IR 中需要维持粗粒度的描述方式, 此方式可直接将突触权值、脉冲发放状态映射到交叉开关阵列上, 而无需将其递降为更细粒度的描述方式. 但同时, 神经元更新操作仍需递降为细粒度描述.

图 3(c) 展示了 Brunel 神经网络的第 2 层 IR. 在网络模拟阶段, 神经元更新操作可以维持第 1 层 IR 中的粗粒度表示 (x1); 也可以递降为细粒度表示 (x2), 使用循环操作描述每一个神经元根据方程更新内部状态的计算过程. 同理, 脉冲传播操作可以根据后端选用粗粒度表示 (y1) 或细粒度表示 (y2).

第 2 层 IR 的细粒度表示主要使用 MLIR 内置方言对各类操作的存储、计算及控制流进行表示, 具体如下.

- memref 方言用作 SNN 模型中的神经元内部状态存储, 通过声明 memory buffer 来指示变量的存储空间. 由于 SNN 中存储的变量会被反复改写, 因此不适合使用 MLIR 中的 tensor 方言来存储 (MLIR 中的 tensor 值无法被修改, 多用于存储 DNN 中的 feature maps).
- sparse tensor 方言用作稀疏张量存储. sparse tensor 方言提供了一套通用的张量稀疏存储格式, 例如可声明突触权值矩阵使用稀疏格式进行存储. 其优势还在于实现了一系列稀疏张量操作, 在操作过程中只需要附加稀疏属性, 而不会破坏张量运算描述的原有结构.
- scf 方言和 linalg 方言用作描述 SNN 计算流程中的控制流. linalg 方言是对高层控制流的描述, 可以抽象地表示一个张量计算的过程, 并且自动完成到低层的递降. scf 方言则通过 if 操作和 for 操作来描述低层控制流, 在计算规则较为固定的时候可以更好地实现优化.
- arith 方言用于描述 SNN 的核心计算. arith 可以实现对标量或向量的基础计算, 是一个 MLIR 核心方言, 贯穿整个递降过程.

3.4 第 3 层 IR

在 MLIR 框架下不同后端的第 3 层 IR 既有差异部分, 也有复用部分, 从而在确保定制优化的前提下, 提高开发效率. 不失一般性, 本文工作主要针对 3 类硬件后端, 即传统控制流后端 CPU、基于 ReRAM 交叉开关结构实现的数据流后端 (简称为 ReRAM 后端)、基于 FPGA 实现的混合架构后端芯片 GaBAN (简称为 GaBAN 后端). 第 2 层 IR 到第 3 层 IR 的递降过程需要结合硬件后端特性, 针对特定操作进行改写与递降以适配计算方式.

具体而言, 控制流架构在第 3 层 IR 中继续延续第 2 层 IR 中的细粒度描述, 利用 SNN 的底层语义对网络中神经元/突触的存储、更新传播过程中的内部状态计算, 以及模拟过程的控制流进行优化 (见第 3.4.1 节). 数据流 ReRAM 后端通过其完成计算的功能主体 (交叉开关结构), 对脉冲传播计算进行加速, 即通过对第 2 层 IR 中粗粒度的脉冲传播过程进行改写, 并额外处理其存储和计算过程, 使其与交叉开关结构的计算方式相匹配 (ReRAM 后端的详细说明请见附录 B); 而对神经元更新操作, 仍采用第 2 层 IR 的细粒度描述 (见第 3.4.2 节). GaBAN 后端的第 3 层

IR 由第 2 层 IR 的细粒度描述通过递降得到: 一方面, 需要将神经网络模拟过程中的具体计算操作转变为计算内核执行的 GaBAN 指令集中的指令序列实现, 另一方面, 需要生成控制核心执行的硬件配置、驱动过程 (见第 3.4.3 节).

3.4.1 CPU 后端

面向 CPU 后端的第 3 层 IR 大体延续了第 2 层 IR 的细粒度描述, 在存储、计算、控制流上描述 SNN 的计算操作过程. 递降过程中, 一方面针对神经元组的空间并行性, 对神经元更新过程进行向量化优化; 针对 SNN 的稀疏特性, 实现了脉冲张量类型的存储与索引方式, 并对脉冲传播操作进行编译优化. 图 3(d1) 展示了神经元更新过程向量化与脉冲传播过程稀疏性优化后的结果.

● 神经元更新操作的并行优化

同一个类脑计算应用内的神经元模型一般是一样的 (参数可能不同), 即多个神经元的同一状态的计算类型是一样的, 仅数据不同, 属于单指令流多数据流 (SIMD) 计算, 因此神经元更新过程中存在向量化优化空间: 第 3 层 IR 对神经元更新操作的计算过程进行拆解, 若将神经元组的内部状态看作几个一维张量, 则其计算过程可表示若干逐元素的张量操作 (element-wise operation), 包括张量加法、减法、乘法等操作. 这些张量操作具有并行性, 针对 CPU 后端可以采用 SIMD 指令集进行加速. 故利用 MLIR vector 方言 (一个用于描述向量化的数据结构方言) 声明更新过程中的并行计算. 图 3(d1) 的神经元更新操作中使用 vector.load 与 vector.store 对连续一段神经元的内部状态进行访存, 并通过 arith 方言对 vector 数据类型进行计算, 这些指令会在后续递降过程中转化为具体硬件的 SIMD 指令.

● 脉冲张量类型

脉冲张量类型描述了神经元的发放状态, 第 1、2 层 IR 主要负责描述与设计, 注重于数据类型的构建和方言生态的完善. 第 3 层 IR 需要考虑脉冲张量的具体实现和优化策略, 即存储格式、操作的实现与优化.

脉冲张量采用 sparse tensor 方言描述存储. sparse tensor 针对不同的稀疏存储格式抽象出一套通用的编码方法, 分别存储非零元素的下标和值. 对于脉冲张量, 可以直接存储非零值的下标而无需存储值 (因为值必定为 1).

● 脉冲传播操作的稀疏性优化

脉冲传播操作可以视作一类特殊的矩阵-向量乘法操作. 使用 linalg.generic 操作可以对此过程进行描述, 但其生成的计算内核无法充分利用发放状态与连接矩阵的稀疏性. 针对上述问题, 本文设计了利用两者稀疏性加速脉冲传播操作的方法, 避免形如 $x+0=x$ 以及 $x\times 0=0$ 等不必要的计算.

图 3(d1) 的脉冲传播操作展示了优化的一种具体实现. 首先使用 sparse tensor 方言将连接矩阵 (%var1) 与发放状态 (%var2) 转变为稀疏格式, 并通过 pointers、indices、values 操作获取内部状态, 方便下一步对非零元的索引过程. 计算时则通过仅遍历发放状态的非零元 (访问 f_indice 变量) 与连接矩阵中的非零元来减少计算 (访问 w_pointer、w_indice、w_value 变量). 内层循环中进一步使用 vector 方言对计算过程进行 SIMD 向量化优化, 此处需要使用 vector.gather 对结果向量中非零元对应下标的值进行不连续访存 (访问 a_in 变量).

需要指出的是, 挖掘稀疏性有多种方法, 而后续测试表明, 针对连接矩阵与发放状态的不同稀疏度, 不存在一个“统一”的最优方法. BIVM 实现了不同格式的计算与加速方法, 用户可以在描述中通过附加属性来起到类似编译指示 (directives) 的作用, 进而在第 3 层 IR 中会被转化为对应的实现.

● 硬件执行

面向传统架构处理器, 第 3 层 IR 后续会被递降为 LLVM IR. LLVM IR 是一个通用的描述方式, 便于递降为不同后端 (通常是通用处理器) 的可执行文件. 至此我们已经将原有的 SNN 应用递降为一个不再包含 SNN 语义而全部由基础操作构成的 IR. 通过后续的编译链接步骤, 可以得到最终的可执行文件.

3.4.2 ReRAM 后端

ReRAM 是一种被广泛应用的神经形态器件, 可利用其静态或者动态特性来高效仿真神经网络突触. 当前版本的 BIVM 主要支持基于 ReRAM 静态特性的、以存内计算 (processing-in-memory) 形式完成的高效、高密度矩阵向量乘运算. 具体而言, 对于 ReRAM 交叉开关结构中的每一行施加电压, 遵循欧姆定律和基尔霍夫定律, 该电压和交叉点的电导值相乘得到电流, 每一列电流累加得到结果, 从而完成了一次模拟的矩阵向量乘法 (详细说明请见

附录 B); 这可以视作一次脉冲传播计算, 也是 BIVM 要优化适配的主体.

ReRAM 交叉开关结构具有低功耗、高密度、计算速度快等特点, 避免了传统神经网络加速器在突触发放计算时的权重数据移动所带来的内存墙问题.

● ReRAM 方言

针对 ReRAM 后端设计的第 3 层 IR, 我们提出了 ReRAM 方言, 以描述 ReRAM 交叉开关阵列加速脉冲传播操作的过程. 第 2 层 IR 中脉冲传播操作使用粗粒度描述表示, 递降到第 3 层 IR 后会转变为如下 ReRAM 方言以符合交叉开关阵列的计算方式: rram.Crossbar 类型代表初始化完成的 ReRAM 交叉开关阵列, 在 IR 中这个类型的实例通过 rram.init_crossbar(weights) 操作基于给定的权值生成, 对应硬件的初始化; 之后, 可以使用 rram.run_crossbar 操作使用交叉开关阵列实例进行矩阵-向量乘法计算 (图 3(d2)). ReRAM 方言的形式化描述请见附录 A 中的表 A2.

● 基于权重微调的精度优化

由于 ReRAM 本身的物理局限, 进行计算时从权值矩阵获取的实际权重数值是围绕特定值 (初始化值) 的一个分布. 因此在基于精确的突触权重 (如经过软件训练得到的网络模型权重) 进行 ReRAM crossbar 初始化之前, 需要进行网络权重微调 (fine-tuning), 以保证在使用较低精度的计算时也能保持其所执行的应用任务结果的高精度 (如后续测试中的 MNIS 神经网络分类任务). BIVM 框架集成了这一功能——当基于 BIVM 的编译器为 ReRAM 类脑芯片进行编译时, 如果使用者输入了目标 ReRAM 的权重分布情况, 就可自动进行权重微调优化过程, 在精度受限的情况下提升计算结果的准确度. 需指出的是, 这类微调方法适用于面向 AI 任务的深度脉冲神经网络 (即 D-SNN): 这类任务有明确的数据集, 可以利用神经网络强大的泛化能力, 基于已有网络拓扑与初始参数进行受限条件下的权重调整[15]; 而对于侧重生物脉冲神经网络模拟的仿真计算应用是否适用, 还需要进一步研究.

● 硬件执行

需要指出的是, 从 ReRAM 方言进一步递降为可以在目标硬件上运行的程序可以采用两种方式, 一是直接调用目标硬件的运行时库, 本工作示例就是采用这一形式: ReRAM 方言中在运行时通过链接外部函数来完成与硬件相关的操作, 这类形式在 DNN 编译框架 (如 TVM[42]) 中也广泛使用, 比如直接链接英伟达 GPGPU 的深度学习库而不是直接生成底层 CUDA 代码. 外部函数@initRRAMCrossbar 和@runRRAMCrossbar 分别对应方言中初始化和使用 ReRAM 交叉开关阵列进行计算的过程. 目前我们使用了 ReRAM 模拟器来模拟交叉开关阵列的运行. 对于其他的 ReRAM 的硬件实现, 动态链接库的设计保证了 ReRAM 后端的灵活性, 实现时只需要改写外部函数, 而不用更改 IR 以及编译器的具体实现; 这一方式也可使得交叉开关阵列的某些物理约束 (如尺寸等) 对编译框架透明. 图 3(d2) 展示了 Brunel 神经网络递降后的 IR.

另一种则是调用目标硬件的底层原生编译器方式, 如 OCC[30]就是一个基于 MLIR 的存算一体编译器, 支持基于交叉开关阵列的矩阵乘法算子的优化调度; 与此类工作的融合将作为下一步工作之一.

3.4.3 GaBAN 后端

GaBAN 是一个针对类脑计算应用所设计的基于 FPGA 的向量处理器, 包含一个 RISC-V 控制核心和数个计算内核, 计算核心中采用自定义指令集以及基于 Buffets[43]的异步访存机制.

● GaBAN 方言

GaBAN 后端第 3 层 IR 由第 2 层 IR 的细粒度描述递降得到. 一方面, 需要将神经网络模拟过程中的具体计算操作转变为计算核心上执行的 GaBAN 指令序列, 另一方面, 需要生成控制核心上的硬件配置、驱动过程. 其中, 由于 GaBAN 指令集采用硬件驱动的隐式循环, 因此为计算核心生成的代码对应一次循环内的计算逻辑. 本研究针对 GaBAN 后端第 3 层 IR 提出了专属的 GaBAN 方言, 包含一种数据类型与两种类型操作: gaban.tensor 对应了 GaBAN 后端中声明的向量数据类型, 它是存储与计算的基本单元; 其中一类操作是向量运算操作, 对应了 GaBAN 中的向量指令, 例如浮点加法操作 gaban.addf 以及减法操作 gaban.subf; 另一类则是控制计算核心相关的功能性操作, 例如写入代码段基地址和循环上下界等配置参数的 gaban.configure 操作以及与计算核心同步的 gaban.wait 操作. GaBAN 方言的形式化描述请见附录 A 中的表 A3.

从第 2 层 IR 到第 3 层 IR 的递降过程首先需要将各类操作中循环体内部的计算过程提取到循环外的代码块中. 第 2 层 IR 中神经元更新、脉冲传播操作的内部调用依赖关系可视作对应静态单赋值 (static single-assignment, SSA) 代码的输入、输出关系, 通过对 SSA 代码的分析, 可以将一个时间步内的模拟计算过程转换为 GaBAN 指令集中对应的向量指令和相应的循环上下界, 最后增加额外的配置写入和同步逻辑. 图 3(d3) 展示了 Brunel 神经网络递降到 GaBAN 后端的第 3 层 IR.

● 硬件执行

在进一步的递降过程中, GaBAN 方言的计算操作会被转化为 LLVM IR 中的专用指令 (通过 Intrinsic 形式), 并最终转换为 GaBAN 指令序列. 控制核心将首先运行计算核心的配置过程, 再触发计算核心开始执行第 3 层 IR 转化得到的指令序列以完成网络模拟中的计算过程. 最终等待全部计算完成后, 控制核心收集计算核心的输出, 得到模拟结果.

4 对于不同硬件的支持与优化效果评测

在本节中, 通过若干示例评估 BIVM 系统的可用性及其编译优化效果. 对于 CPU 后端, 我们评估了脉冲传播操作的稀疏性优化、神经元组合并与向量化等多种优化方法的效果; 对 Brunel 和 MNIST 推理任务这两个分别代表计算神经学领域与 AI 应用领域的测例进行了端到端测试, 并与广泛应用于这两个领域的 CPU 端 SNN 模拟框架进行了性能对比. 针对 ReRAM, 本文通过 MNIST 推理任务验证了其运行的正确性. 在 GaBAN 后端上测试了 Brunel 与 MNIST 分类任务, 在验证正确性的同时, 比较了 BIVM 与其原生工具链的效果.

具体的测试环境如下: CPU 后端的运行环境为 AMD EPYC 7742 64-Core Pro; ReRAM 后端通过软件模拟实现[15]; GaBAN 基于 FPGA 实现, 具体设备为 Xilinx Virtex UltraScale+ HBM VCU128 评估板, 使用的综合工具为 Xilinx Vivado (2020.2 版). 实验执行的操作系统为 Ubuntu 20.04, 编译器为 GCC 9.4.0.

作为对比的 SNN 模拟框架分别为 NEST (使用具有 Python 接口的 PyNEST 2.20 版, 这是一个常用的生物神经网络模拟框架, 侧重于神经系统动力学以及结构的模拟) 与 SpikingJelly (0.6 版, 这是一个基于 PyTorch 的支持 SNN 的开发与模拟工具, 可以训练与运行 SNN).

最后需要指出的是, 针对不同优化选项与此 3 类后端, 这两类 Benchmark 的 BIVM 编译时间范围是 0.17–0.28 s, 远小于目标程序的运行时间.

4.1 脉冲传播操作性能优化评测

我们针对 SNN 的稀疏性设计了脉冲张量这一数据类型, 并且在第 2 层 IR 递降过程中针对脉冲传播过程进行优化. 这里就针对该类稀疏性进行不同编译优化策略的效果对比, 具体如下: (1) 均视作稠密张量; (2) 利用脉冲张量的稀疏性; (3) 利用脉冲张量与连接矩阵的稀疏性. 同时, 我们也测试了方法 (2) 与 (3) 上使用 SIMD 向量化的性能对比. 通过调整发放状态和连接矩阵的非零元比例, 可得到不同稀疏度下各方法的性能表现.

我们设定发放状态的大小为 5000, 其发放率被设定为 1%、5%、10%, 代表神经网络不同的发放情况, 运行时随机选取对应比例的神经元并设置其为发放状态. 连接矩阵的基础规模为 5000×5000, 突触均使用静态类型, 矩阵的非零元比例设定为 10% 与 100%, 代表计算神经学与 D-SNN 两类应用的连接情况: 前者较为稀疏, 随机地将 10% 的突触权重设定为非零; 后者则使用全连接层进行连接. 我们重复运行脉冲传播过程 5000 次, 最终得到表 4 测试结果. 对于均视作稠密张量处理的方法, 由于计算量较为一致, 运行时间不随发放率与连接密度明显变化.

利用发放状态稀疏性的方法运行速度取决于脉冲张量的稀疏度, 在 1%、5%、10% 发放率的情况下其加速比可分别达到 98.02 倍、21.77 倍、10.57 倍, 接近理论加速比. 进一步地, 在此方法上使用 SIMD 指令向量化可实现约 3.96 倍的性能提升. 同时利用连接矩阵与发放状态稀疏性的方法取决于两者的稀疏度. 当连接密度为 10% 时, 在 1%、5%、10% 发放率的情况下, 相对于只利用发放状态稀疏性的方法, 其加速比可分别达到 1.49 倍、3.67 倍、5.20 倍; 但当连接密度为 100%, 即矩阵不再具有稀疏性时, 此方法会发生退化, 执行时间反而增加. 进一步地, 在此

方法上使用 SIMD 指令向量化无法达到加速效果, 原因在于计算向量化带来的性能提升无法覆盖由 vector.gather 操作引入的不连续访存开销.

表 4 脉冲传播操作运行时间优化评测 (s)

发放状态稀疏性	连接矩阵稀疏性	SIMD向量化	神经元发放率(%)/连接密度(%)					
			1/10	5/10	10/10	1/100	5/100	10/100
稠密	稠密	×	87.710	83.314	83.558	84.175	83.884	84.846
稀疏	稠密	×	0.854	3.826	7.898	0.853	3.859	7.824
稀疏	稠密	√	0.257	0.876	1.880	0.255	0.853	2.011
稀疏	稀疏	×	0.571	1.042	1.519	5.476	11.740	20.521
稀疏	稀疏	√	0.562	1.107	1.836	5.185	11.693	19.965

需要指出的是, 测试表明, 针对连接矩阵与发放状态的不同稀疏度, 不存在一个一致的最优方法 (见表 4); 可以通过应用描述层的相关附加属性由使用者来给出编译指示, 如连接矩阵与发放状态的存储方式 (这一属性将递降为 SNN 方言中的属性“sparse_tensor encoding”, 见表 3. 比如在发放状态属性为“compressed” (表示稀疏, 即压缩后仅存非零元) 的前提下, 连接矩阵属性若为“dense”, 则采用“SIMD 向量化”优化, 取值“compressed”则不采用, 但是进行稀疏计算优化等).

4.2 合并与向量化优化评测

我们评测了神经元组的合并以及计算向量化所带来的性能变化. 计算神经学应用的网络往往包含多个相同类型神经元组, 可以将其合并成一个更大的神经元组. 首先构造了一个包含 8 个大小为 10 000 的 LIF 神经元组的网络, 并将其合并为 4 个大小为 20 000、2 个大小为 40 000、1 个大小为 80 000 的神经元组, 对比不同合并粒度的性能提升. 同时, 也对比了使用 SIMD 向量计算对神经元计算 (浮点计算) 的性能提升. 具体的, 对比了在 vector 方言中设定不同向量长度带来的性能变化 (如 vector<8xf32> 将 8 个 32 位浮点数视作一个单位, 参与访存以及运算). vector 方言最终被转换为 AVX256 指令进行运算.

图 4 为重复执行神经元更新过程 10^5 次后的测试结果. 由结果可知, 神经元合并与向量化都可以带来性能提升. 当向量长度为 1 时 (以串行方式执行), 整体运行时间最慢. 随着向量长度不断增加 (2、4、8、16), 神经元更新的并行性逐渐被挖掘, 运行时间逐渐减少; 当向量长度等于 16 时, 整体性能达到最优, 与向量长度为 1 时 (不使用 SIMD 指令加速) 相比可达到 6.7 倍的性能加速. 但随着向量长度的进一步增大 (32、64、128), 运行时间则略微增加. 经过分析不同向量长度所对应的汇编指令, 发现性能降低的原因是由于向量寄存器数目有限, 当向量长度增加时, 不足以同时存储所有内部状态, 对于超出部分会发生溢出 (spilling, 即将部分寄存器的值转移到栈上, 并直到需要参与计算时再将这些值从栈上重新转移回寄存器中), 从而导致部分不必要的内存移动, 在执行时增加了额外开销.

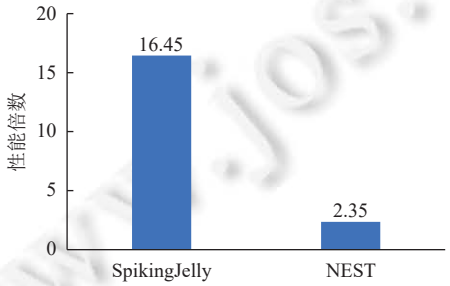

图 4 CPU 后端上相比于其他实现的性能优势

4.3 CPU 后端的端到端的测试

Brunel 和 MNIST 推理任务分别代表计算神经学和 D-SNN 方向的典型类脑应用. 我们测试了由 BIVM 编译

生成的这两个应用的性能, 并与 NEST 与 SpikingJelly 框架进行对比 (性能对比见图 4).

4.3.1 Brunel 神经网络模拟

Brunel 神经网络是计算神经学领域中一类典型的网络形式, 包含兴奋性神经元组和抑制性神经元组, 组间按照一定概率相连. 我们将 BIVM 与 NEST 框架进行性能对比.

如表 5 所示, 测试的 Brunel 神经网络包含大小分别为 8000 和 2000 的 Izhikevich 两个神经元组, 并以 10% 的概率相互连接, 连接权重固定. Izhikevich 神经元组参数固定为 NEST 框架中的默认值, 但参数恒定电流 (i_e) 调整至 50 pA. 在此系列参数下, 神经元的平均发放率约为 100 Hz. 测试中时间步设定为 1 ms, 并运行 10 s (10000 个时间步).

表 5 Brunel 神经网络参数说明

网络规模与参数	说明
神经元规模	兴奋性神经元组: 8000, 抑制性神经元组: 2000
突触规模	10^7
神经元类型	Izhikevich
神经元参数	a: 0.02, b: 0.2, c: −65.0, d: 8.0, i_e: 50.0 pA, v_{th}: 30.0 mV
突触类型	静态突触
突触参数	兴奋性突触权重w: 0.005 pA, 抑制性突触权重w: −0.001 pA

本研究开展了消融测试, 对比了不同优化选项对 BIVM 执行时间的影响. 测试采用了如下的优化方法: 对神经元组与突触组进行合并、对神经元更新操作使用 SIMD 向量化、对脉冲传播操作使用不同的稀疏格式运算. 表 6 展示了消融测试结果. 其中, 对脉冲传播操作进行优化的加速效果最优, 连接矩阵转换为稀疏格式方法可以达到 4.04 倍加速比 (72.406→17.896). 神经元组与突触组合并带来了 36.94% 的性能提升 (17.896→13.068), 而神经元更新操作使用 SIMD 向量化带来了 4.98% 的性能提升 (13.068→12.448).

表 6 Brunel 神经网络消融测试

神经元更新操作方法	元组合并	脉冲传播操作方法	执行时间 (s)
标量计算	×	发放状态稀疏	72.406
标量计算	×	发放状态、连接矩阵均稀疏	17.896
标量计算	√	发放状态、连接矩阵均稀疏	13.068
SIMD向量化	√	发放状态、连接矩阵均稀疏	12.448

最终, 我们对比了 BIVM 与 NEST 的网络模拟时间 (除去模型加载与编译部分). NEST 配置为 8 线程并行运行, 其模拟运行时间为 29.369 s; BIVM 的模拟运行时间为 12.448 s, 性能是前者的 2.35 倍. 性能的差异主要来自我们针对脉冲传播过程的稀疏性进行优化, 而 NEST 并没有对此做额外支持.

4.3.2 MNIST 分类任务

MNIST 是一个经典的数字分类任务 (参数设置见表 7), 我们对比了 BIVM 与 SpikingJelly 在 MNIST 上的性能. MNIST 网络是一个双层 SNN 网络, 分为输入层与输出层; 输入层接受大小为 28×28 的图片转换得到的脉冲编码作为输入. 脉冲编码使用泊松分布生成, 其能产生与输入值 x 对应的并满足泊松分布的脉冲序列 (平均间隔 $1/x$ 个时间步进行一次发放). 输出层包含 10 个 LIF 神经元, 分别对应 MNIST 任务中 10 种数字的分类, 最终会使用 Softmax 函数选择一段时间内输出层中发放次数最多的神经元而得到分类结果. 输入层与输出层之间使用静态突触以全连接方式相连, 权重通过 SpikingJelly 框架训练得到, 分类精度为 92.4%.

针对 CPU 后端, 我们对比了 BIVM 与 SpikingJelly 在 MNIST 推理任务上的执行时间. 进行推理任务时, 首先需要根据权重加载网络模型, 再以 1 ms 为时间步运行 0.1 s (100 时间步), 并统计输出层的累计发放次数得到结果并确认准确性.

除去模型的加载以及预处理时间, SpikingJelly 处理一张图片的平均时间为 10.104 ms; BIVM 处理一张图片的平均时间为 0.614 ms, 性能为前者的 16.45 倍. 产生此差距的原因一方面在于 SpikingJelly 基于 PyTorch 实现, 会在运行时引入额外开销, 另一方面 SpikingJelly 并没有针对脉冲张量的稀疏性进行优化.

表 7 MNIST 神经网络参数说明

网络规模与参数	说明
神经元规模	输入层: 784, 输出层: 10
突触规模	7840
神经元类型	输入层: Poisson, 输出层: LIF
神经元参数	v_{th}: 1 mV, τ: 0.5 s, v_{reset}: 0 mV
突触类型	静态突触
突触参数	权重w由SpikingJelly训练得到

我们进一步开展了消融测试, 对比了不同优化选项对执行时间的影响. 实验采用了如下优化方法: 对神经元更新操作使用 SIMD 向量化、对脉冲传播操作使用不同的稀疏格式与向量化方法运算. 表 8 展示了消融测试结果. 其中发放状态稀疏性, 达到了 1.97 倍加速比 (1.262→0.640). 由神经元更新使用 SIMD 向量化以及脉冲传播操作中进行向量化带来的性能提升较小, 主要原因为该网络的网络规模较小, 向量化带来的并行性提升不明显.

表 8 MNIST 神经网络消融测试

神经元更新操作方法	脉冲传播操作方法	执行时间 (ms)
标量计算	均为稠密	1.262
标量计算	发放状态稀疏	0.640
标量计算	发放状态稀疏 + SIMD向量化	0.617
SIMD向量化	发放状态稀疏 + SIMD向量化	0.614

4.4 ReRAM 后端的端到端测试

我们采用了 XB-SIM*[15]框架作为这一测试中的 ReRAM 行为仿真后端. XB-SIM*是一个可灵活配置的、基于 ReRAM 交叉开关结构的神经网络加速器开源仿真框架, 以时钟驱动方式模拟加速器结构与流水线行为.

XB-SIM*在 ReRAM 行为模型以外, 实际上还包括了面向 ReRAM 特性的神经网络训练等功能, 但在这一测试中仅使用其 ReRAM 神经网络加速器 (ReRAM-based NN accelerator, RNA) 仿真功能, 即以时钟驱动方式仿真配备了控制逻辑和数据缓冲区的由多个处理单元构成的神经网络推断流水线, 每个处理单元的主体是若干个 ReRAM 交叉开关结构, 进而运行 SNN 推断过程.

仿真所涉及的主要配置参数如表 9 所示; 每个交叉点上存储的权重被限制到 [−1, 1] 区间中的 32 个离散数值上, 并在计算中模拟了随机误差. 交叉开关的输出被限制在 [−128, 128] 区间中, 并被舍入为最接近的整数. XB-SIM*的结构、运行过程以及参数的具体解释请见附录 B.

表 9 ReRAM 仿真参数

ReRAM参数	说明
权重范围	[−1, 1]区间
权重精度	32种离散可能取值
ADC输出位宽	8 bits
ADC输出精度	1
ADC输出范围	[−128, 128]
DAC输出电压	0.15 V
RNA流水线延迟	100 ns

基于此后端, 我们首先测试了编译生成的 MNIST 分类任务准确率, 如表 10 所示.

其中, 未经过微调的转换只是将原矩阵中权值直接替代为最接近的离散权值, 由于 ReRAM 硬件带来的数据表示与计算精度限制, 相比原始准确率有显著下降. 经过微调的转换则是在替代权值基础上, 使用上述的 RNA 仿真作为前向过程进行了训练, 使得权值能够适应 ReRAM 低精度计算, 进而使得分类准确率接近原始精度.

其次, 我们还测试了不同硬件配置情况下, 分类任务的计算耗时与功耗. 对于不同的交叉开关结构规模, 需要

将神经网络连接矩阵以不同的方式映射到硬件上. 在已知矩阵规模和硬件规模的情况下, BIVM 对连接矩阵进行不同的切分, 并给出 ReRAM 后端上的计算耗时和功耗. 在 784×10, 392×20, 1568×5 这 3 种交叉开关结构规模下, XB-SIM*的计算耗时和功耗如表 11 所示.

表 10 MNIST 推理任务在 ReRAM 后端的预测准确率 (%)

微调策略	预测准确率
CPU后端	92.46
未经权重微调	90.78
经过权重微调	92.41

表 11 MNIST 推理任务在 ReRAM 后端的计算时间与功耗

Crossbar规模	每帧耗时 (ns)	功率 (mW)
784×10	100	352.247
392×20	100	587.724
1568×5	200	271.294

计算耗时和功耗不同的原因在于, MNIST 连接矩阵大小是 784×10, 因此需要两个 392×20 规模的交叉开关结构才能容纳所有权重, 而发放过程会被拆分成两次同时进行的矩阵-向量乘法计算, 再将结果求和. 对于 1568×5 这一规模, 则会通过时分复用的方法执行两次计算, 即第 1 次采用开关结构的上半部分 (784×5), 第 2 次用下半部分, 这样权重存储不会冲突, 进而各得到 5 个元素的输出. 矩阵在交叉开关结构上以不同的方式映射会影响信号传播延迟、扇出等属性, 进而反映到计算耗时 (考虑到流水线处理, 这儿的耗时指的是连续两帧的结果间隔) 和功耗上. 具体映射方式请见附录 B.

4.5 GaBAN 后端的端到端测试

本研究测试了 Brunel 神经网络在 GaBAN 后端的性能表现. GaBAN 后端采用 8 通道的 GaBAN 内核 (支持浮点计算), 时钟频率为 250 MHz. 本测试使用 BIVM 将 Brunel 神经网络递降到 GaBAN 后端并执行模拟过程, 最终模拟时间为 20.31 s. 另外, 在合并同类型神经元/突触组后, BIVM 递降到 GaBAN 后端的模拟时间降低到 16.30 s, 与未合并相比提升了 24%.

GaBAN 为用户提供了原生工具链, 包括配套的编译器. 该编译器支持将网络描述转化为专属 IR, 并执行乘加融合、公共表达式消除和冗余指令消除等通用编译优化. 直接使用 GaBAN 工具链对 Brunel 神经网络进行递降, 测试得到的模拟时间为 20.49 s, 与未采用合并优化的 BIVM 相比性能相近. 两种方法得到的测试结果一致, 说明 BIVM 可以保证网络模拟的正确性.

本测试使用 BIVM 将 MNIST 推理网络递降到 GaBAN 后端并执行模拟过程, 得到处理一张图片的平均时间为 1.06 ms; 直接使用 GaBAN 工具链对此网络进行递降, 处理一张图片的平均时间为 1.04 ms, 与使用 BIVM 相比性能相近.

5 进一步需解决的问题

本文对基于 MLIR 的类脑计算编译框架进行了研究尝试, 结果初步表明该类框架在编译高生产力、高可移植性、高性能方面具有良好潜力. 但是, 还有几类关键问题需要深入研究.

第一, 类脑计算处理器的抽象模型. 建立目标硬件的抽象模型 (一类或者多类) 及其约束条件 (如资源规模、计算精度等), 包括其计算、存储层次、控制等方面的细化抽象是进行编译底层 IR 设计的关键, 良好的硬件抽象有助于对不同硬件后端的相似/相同的编译优化技术进行统一表述与适配, 从而提升编译器构建时的 productivity. 类脑计算处理器的抽象模型已有初步研究[10]涉及; 同时, BIVM 第 2 层 IR 所采用的针对粗粒度/细粒度后端的不同 IR 表示, 可以视作对类脑计算处理器抽象模型的初步分类. 而类脑应用可能需要的高精度计算 (比如生物神经网络模拟所需的计算精度) 与类脑芯片能够提供的精度 (如基于 ReRAM 的低精度计算, 或者一些类脑芯片仅提供定点计算) 之间存在差距, 在用户开发层面解决这个问题, 还是通过编译透明的解决[44]也是需要研究的问题.

第二, 与自动优化技术的融合. BIVM 以手工优化为主, 包括计算稀疏性挖掘与神经元组合并计算, 以及各个后端 IR 的设计等; 而有的深度学习运行或编译框架是以自动优化为主, 可以带来更多的自动效果且降低工程开销. 目前也逐渐出现了两者融合的技术趋势, 比如引入张量计算优化空间自动探索技术的 TensorIR[45]. 如何针对 SNN 计算特点, 引入领域知识结合自动优化的编译技术是一个重点.

6 总 结

本文将计算机系统领域编译基础设施的新进展 MLIR 与类脑计算领域对于可扩展、可复用编译工具的需求结合起来, 设计实现了一个类脑计算编译框架及其验证原型——BIVM. BIVM 充分利用 MLIR 的 progressivity 特性, 所设计的中间表示能够混合不同级别的抽象层次和概念, 而在高层到低层抽象的渐进递降过程中可灵活组合不同层次的编译优化方法, 从而较好地解决了编译框架所面临的类脑计算应用特征跨度大、芯片体系结构跨度大等问题. 测试表明, BIVM 编译生成程序的运行性能比目前广泛采用的开发框架更高 (针对处理器芯片), 或者比其原有工具链生成的程序更高或相近 (针对所支持的两种类脑芯片).

References:

[1] Goertzel B. Artificial general intelligence: Concept, state of the art, and future prospects. Journal of Artificial General Intelligence, 2014, 5(1): 1–48. [doi: 10.2478/jagi-2014-0001]

[2] Hennessy JL, Patterson DA. A new golden age for computer architecture. Communications of the ACM, 2019, 62(2): 48–60. [doi: 10.1145/3282307]

[3] Roy K, Jaiswal A, Panda P. Towards spike-based machine intelligence with neuromorphic computing. Nature, 2019, 575(7784): 607–617. [doi: 10.1038/s41586-019-1677-2]

[4] Qu P, Yan J, Zhang YH, Gao GR. Parallel turing machine, a proposal. Journal of Computer Science and Technology, 2017, 32(2): 269–285. [doi: 10.1007/s11390-017-1721-3]

[5] Rhodes O. Brain-inspired computing boosted by new concept of completeness. Nature, 2020, 586(7829): 364–366. [doi: 10.1038/d41586-020-02829-w]

[6] Hazan H, Saunders DJ, Khan H, Patel D, Sanghavi DT, Siegelmann HT, Kozma R. BindsNET: A machine learning-oriented spiking neural networks library in Python. Frontiers in Neuroinformatics, 2018, 12: 89. [doi: 10.3389/fninf.2018.00089]

[7] Pehle CG, Pedersen JE. Norse—A deep learning library for spiking neural networks. 2021. https://zenodo.org/records/4422025

[8] Mozafari M, Ganjtabesh M, Nowzari-Dalini A, Masquelier T. SpykeTorch: Efficient simulation of convolutional spiking neural networks with at most one spike per neuron. Frontiers in Neuroscience, 2019, 13: 625. [doi: 10.3389/fnins.2019.00625]

[9] Fang W, Chen YQ, Ding JH, Yu ZF, Masquelier T, Chen D, Huang LW, Zhou HH, Li GQ, Tian YH. SpikingJelly: An open-source machine learning infrastructure platform for spike-based intelligence. Science Advances, 2023, 9(40): eadi1480. [doi: 10.1126/sciadv.adi1480]

[10] Zhang YH, Qu P, Ji Y, Zhang WH, Gao GR, Wang GR, Song S, Li GQ, Chen WG, Zheng WM, Chen F, Pei J, Zhao R, Zhao MG, Shi LP. A system hierarchy for brain-inspired computing. Nature, 2020, 586(7829): 378–384. [doi: 10.1038/s41586-020-2782-y]

[11] Richmond P, Cope A, Gurney K, Allerton DJ. From model specification to simulation of biologically constrained networks of spiking neurons. Neuroinformatics, 2014, 12(2): 307–323. [doi: 10.1007/s12021-013-9208-z]

[12] Aimone JB, Severa W, Vineyard CM. Composing neural algorithms with Fugu. In: Proc. of the 2019 Int'l Conf. on Neuromorphic Systems. Knoxville: ACM, 2019. 3. [doi: 10.1145/3354265.3354268]

[13] Lagorce X, Benosman R. STICK: Spike time interval computational kernel, a framework for general purpose computation using neurons, precise timing, delays, and synchrony. Neural Computation, 2015, 27(11): 2261–2317. [doi: 10.1162/NECO_a_00783]

[14] Chen JJ, Yang L, Zhang YH. GaBAN: A generic and flexibly programmable vector neuro-processor on FPGA. In: Proc. of the 59th ACM/IEEE Design Automation Conf. San Francisco: ACM, 2022. 931–936. [doi: 10.1145/3489517.3530561]

[15] Fei X, Zhang YH, Zheng WM. XB-SIM*: A simulation framework for modeling and exploration of ReRAM-based CNN acceleration design. Tsinghua Science and Technology, 2021, 26(3): 322–334. [doi: 10.26599/TST.2019.9010070]

[16] Brette R, Rudolph M, Carnevale T, *et al*. Simulation of networks of spiking neurons: A review of tools and strategies. Journal of Computational Neuroscience, 2007, 23(3): 349–398. [doi: 10.1007/s10827-007-0038-6]

[17] Burkitt AN. A review of the integrate-and-fire neuron model: I. Homogeneous synaptic input. Biological Cybernetics, 2006, 95(1): 1–19. [doi: 10.1007/s00422-006-0068-6]

[18] Izhikevich EM. Simple model of spiking neurons. IEEE Trans. on Neural Networks, 2003, 14(6): 1569–1572. [doi: 10.1109/TNN.2003.820440]

[19] Hodgkin AL, Huxley AF. A quantitative description of membrane current and its application to conduction and excitation in nerve. Bulletin of Mathematical Biology, 1990, 52(1–2): 25–71. [doi: 10.1016/S0092-8240(05)80004-7]

[20] Carnevale NT, Hines ML. The NEURON Book. Cambridge: Cambridge University Press, 2006.

[21] Olshausen BA, Field DJ. How close are we to understanding V1? Neural Computation, 2005, 17(8): 1665–1699. [doi: 10.1162/0899766054026639]

[22] Shrestha SB, Orchard G. SLAYER: Spike layer error reassignment in time. In: Proc. of the 32nd Int'l Conf. on Neural Information Processing Systems. Montréal: Curran Associates Inc., 2018. 1419–1428.

[23] Kim J, Kim K, Kim JJ. Unifying activation- and timing-based learning rules for spiking neural networks. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 1639.

[24] Furber SB, Galluppi F, Temple S, Plana LA. The SpiNNaker project. Proc. of the IEEE, 2014, 102(5): 652–665. [doi: 10.1109/JPROC.2014.2304638]

[25] Akopyan F, Sawada J, Cassidy A, Alvarez-Icaza R, Arthur J, Merolla P. TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip. IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, 2015, 34(10): 1537–1557. [doi: 10.1109/TCAD.2015.2474396]

[26] Pei J, Deng L, Song S, Zhao MG, Zhang YH, Wu S, Wang GR, Zou Z, Wu ZZ, He W, Chen F, Deng N, Wu S, Wang Y, Wu YJ, Yang ZY, Ma C, Li GQ, Han WT, Li HL, Wu HQ, Zhao R, Xie Y, Shi LP. Towards artificial general intelligence with hybrid Tianjic chip architecture. Nature, 2019, 572(7767): 106–111. [doi: 10.1038/s41586-019-1424-8]

[27] Davies M, Srinivasa N, Lin TH, Chinya G, Cao YQ, Choday SH, Dimou G, Joshi P, Imam N, Jain S, Liao YY, Lin CK, Lines A, Liu RK, Mathaikutty D, McCoy S, Paul A, Tse J, Venkataramanan G, Weng YH, Wild A, Yang Y, Wang H. Loihi: A neuromorphic manycore processor with on-chip learning. IEEE Micro, 2018, 38(1): 82–99. [doi: 10.1109/MM.2018.112130359]

[28] Qu P, Yang L, Zheng WM, Zhang YH. A review of basic software for brain-inspired computing. CCF Trans. on High Performance Computing, 2022, 4(1): 34–42. [doi: 10.1007/s42514-022-00092-1]

[29] Song SH, Balaji A, Das A, Kandasamy N, Shackleford J. Compiling spiking neural networks to neuromorphic hardware. In: Proc. of the 21st ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems. London: ACM, 2020. 38–50. [doi: 10.1145/3372799.3394364]

[30] Siemieniuk A, Chelini L, Khan AA, Castrillon J, Drebes A, Corporaal H, Grosser T, Kong M. OCC: An automated end-to-end machine learning optimizing compiler for computing-in-memory. IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, 2022, 41(6): 1674–1686. [doi: 10.1109/TCAD.2021.3101464]

[31] Balaji A, Song SH, Titirsha T, Das A, Krichmar J, Dutt N, Shackleford J, Kandasamy N, Catthoor F. NeuroXplorer 1.0: An extensible framework for architectural exploration with spiking neural networks. In: Proc. of the 2021 Int'l Conf. on Neuromorphic Systems. Knoxville: ACM, 2021. 10. [doi: 10.1145/3477145.3477156]

[32] Minutoli M, Castellana VG, Tan C, Manzano J, Amatya V, Tumeo A. SODA: A new synthesis infrastructure for agile hardware design of machine learning accelerators. In: Proc. of the 2020 IEEE/ACM Int'l Conf. on Computer Aided Design. San Diego: IEEE, 2020. 1–7. [doi: 10.1145/3400302.3415781]

[33] Curzel S, Agostini NB, Song SH, Dagli I, Limaye A, Tan C. Automated generation of integrated digital and spiking neuromorphic machine learning accelerators. In: Proc. of the 2021 IEEE/ACM Int'l Conf. on Computer Aided Design. Munich: IEEE, 2021. 1–7. [doi: 10.1109/ICCAD51958.2021.9643474]

[34] Schuman CD, Kulkarni SR, Parsa M, Parker Mitchell J, Date P, Kay B. Opportunities for neuromorphic computing algorithms and applications. Nature Computational Science, 2022, 2(1): 10–19. [doi: 10.1038/s43588-021-00184-y]

[35] Schemmel J, Grübl A, Hartmann S, Kononov A, Mayr C, Meier K, Millner S, Partzsch J, Schiefer S, Scholze S, Schüffny R, Schwartz MO. Live demonstration: A scaled-down version of the BrainScaleS wafer-scale neuromorphic system. In: Proc. of the 2012 IEEE Int'l Symp. on Circuits and Systems. Seoul: IEEE, 2012. 702. [doi: 10.1109/ISCAS.2012.6272131]

[36] Lattner C, Amini M, Bondhugula U, Cohen A, Davis A, Pienaar J, Riddle R, Shpeisman T, Vasilache N, Zinenko O. MLIR: Scaling compiler infrastructure for domain specific computation. In: Proc. of the 2021 IEEE/ACM Int'l Symp. on Code Generation and Optimization. Seoul: IEEE, 2021. 2–14. [doi: 10.1109/CGO51591.2021.9370308]

[37] Brunel N. Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. Journal of Computational Neuroscience, 2000, 8(3): 183–208. [doi: 10.1023/A:1008925309027]

[38] Davison AP, Brüderle D, Eppler J, Kremkow J, Muller E, Pecevski D, Perrinet L, Yger P. PyNN: A common interface for neuronal network simulators. Frontiers in Neuroinformatics, 2009, 2: 11. [doi: 10.3389/neuro.11.011.2008]

[39] Asgari H, Maybodi BMN, Payvand M, Azghadi MR. Low-energy and fast spiking neural network for context-dependent learning on FPGA. IEEE Trans. on Circuits and Systems II: Express Briefs, 2020, 67(11): 2697–2701. [doi: 10.1109/TCSII.2020.2968588]

[40] Fang HW, Shrestha A, Ma D, Qiu QR. Scalable NoC-based neuromorphic hardware learning and inference. In: Proc. of the 2018 Int'l Joint Conf. on Neural Networks. Rio de Janeiro: IEEE, 2018. 1–8. [doi: 10.1109/IJCNN.2018.8489619]

[41] Qu P, Zhang YH, Fei X, Zheng WM. High performance simulation of spiking neural network on GPGPUs. IEEE Trans. on Parallel and Distributed Systems, 2020, 31(11): 2510–2523. [doi: 10.1109/TPDS.2020.2994123]

[42] Chen TQ, Moreau T, Jiang ZH, Zheng LM, Yan E, Cowan M, Shen HC, Wang LY, Hu YW, Ceze L, Guestrin C, Krishnamurthy A. TVM: An automated end-to-end optimizing compiler for deep learning. In: Proc. of the 13th USENIX Conf. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2018. 579–594.

[43] Pellauer M, Shao YS, Clemons J, Crago N, Hegde K, Venkatesan R, Keckler SW, Fletcher CW, Emer J. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In: Proc. of the 24th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Providence: ACM, 2019. 137–151. [doi: 10.1145/3297858.3304025]

[44] Ji Y, Zhang YH, Chen WG, Xie Y. Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler. In: Proc. of the 23rd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Williamsburg: ACM, 2018. 448–460. [doi: 10.1145/3173162.3173205]

[45] Feng SY, Hou BH, Jin HY, Lin WW, Shao JR, Lai RH, Ye ZH, Zheng LM, Yu CH, Yu Y, Chen TQ. TensorIR: An abstraction for automatic tensorized program optimization. In: Proc. of the 28th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Vancouver: ACM, 2023. 804–817. [doi: 10.1145/3575693.3576933]

附录 A. SNN、ReRAM 与 GaBAN 方言

BIVM 所设计的 SNN、ReRAM 与 GaBAN 方言中, 各类原语的形式化描述请见表 A1–表 A3; 表中给出了各类操作/数据类型/属性的名称、语义层面的形式化描述、文字解释, 以及运行时数据格式层面的形式化描述.

表 A1 SNN 方言中的操作、数据类型与属性

名称	语义层面的形式描述	含义	数据格式层面的形式化描述
snn.newlif	Lif -> Lif[N]	根据输入的Lif 初始状态, 生成并初始化一组 LIF神经元	(i32, f32, f32, f32, f32, f32, i1) -> (memref<?xf32>, memref<?xf32>, memref<?xf32>, memref<?xf32>, spike_tensor<?xi1>, memref<?xf32>)
snn.newstatic	Weight[N][N] -> StaticSynapse	根据权重矩阵生成一组静态突触	(i32, i32, memref<?xf32>) -> tensor<?x?xf32>
snn.newizhikevich	Izhikevich -> Izhikevich[N]	根据输入的Izhikevich 初始状态, 生成并初始化一组 Izhikevich神经元	(i32, f32, f32, i1) -> (memref<?xf32>, memref<?xf32>, spike_tensor<?xi1>, memref<?xf32>)
snn.newpoisson	() -> Poisson[N]	生成并初始化一组Poisson神经元	() -> memref<?xi32>
snn.lif	(Weight[N], Lif[N]) -> Spike[N]	根据输入脉冲的权重对一组LIF神经元进行更新	(memref<?xf32>, memref<?xf32>, memref<?xf32>, memref<?xf32>, spike_tensor<?xi1>, memref<?xf32>)
snn.static	(Spike[N], StaticSynapse) -> Weight[N]	根据前序神经元产生的脉冲, 执行静态突触脉冲发放, 计算得到后继神经元的脉冲权重输入	(spike_tensor<?xi1>, tensor<?x?xf32>, memref<?xf32>)
snn.izhikevich	(Weight[N], Izhikevich[N]) -> Spike[N]	根据输入脉冲的权重对一组Izhikevich神经元进行更新	(memref<?xf32>, memref<?xf32>, spike_tensor<?xi1>, memref<?xf32>, f32, f32, f32, f32, f32, f32)
snn.poisson	(Probability[N], Poisson[N]) -> Spike[N]	根据输入的概率参数, 由 Poisson 神经元产生脉冲	(memref<?xi32>, spike_tensor<?xi1>, memref<?xi32>)
spike_tensor	—	脉冲张量类型, 表示一列神经元发放的脉冲	{ tensor<type, #SparseEncoding> }
sparse_tensor encoding	—	稀疏矩阵格式, 表示稀疏矩阵的逻辑下标和存储位置之间的映射关系	这一属性复用了MLIR中sparse_tensor方言的encoding属性

表 A2 ReRAM 方言中的操作、数据类型与属性

名称	语义层面的形式描述	含义	数据格式层面的形式化描述
rram.init_crossbar	Weight[N][N] -> Crossbar	使用权值初始化硬件交叉开关, 并得到一个 Crossbar 实例	memref<?xf32> -> Crossbar
rram.run_crossbar	(Spike[N], Crossbar) -> Weight[N]	根据前序神经元产生的脉冲, 使用 Crossbar 实例计算得到后继神经元的脉冲权重输入	(spike_tensor<?xi1>, Crossbar, memref<?xf32>)
Crossbar	—	交叉开关类型, 表示一个已经初始化完成, 可以使用的ReRAM交叉开关	无运行时存储

表 A3 GaBAN 方言中的操作、数据类型与属性

名称	语义层面的形式描述	含义	数据格式层面的形式化描述
gaban.sub gaban.add gaban.mul	(Float[N], Float[N]) -> Float[N]	执行二元计算	(Tensor<?xf32>, Tensor<?xf32>) -> Tensor<?xf32>
gaban.cmp	(Float[N], Float[N]) -> Bool[N]	执行逐个元素比较	(Tensor<?xf32>, Tensor<?xf32>) -> Tensor<?xi1>
gaban.where	(Bool[N], Float[N], Float[N]) -> Float[N]	根据布尔值, 选择两个输入中的一个	(Tensor<?xi1>, Tensor<?xf32>, Tensor<?xf32>) -> Tensor<?xf32>
gaban.load gaban.store	MemoryConfig -> Float[N] (MemoryConfig, Float[N]) -> ()	使用控制核预先配置好的访存配置, 加载或存储数据	(i1, i32, i32) -> Tensor<?xf32> (i1, i32, i32, Tensor<?xf32>)
gaban.configure	GaBANParams -> ()	配置 GaBAN 核心的运行时参数, 例如循环周期等	i32 -> ()
gaban.wait	() -> ()	等待 GaBAN 核心完成执行	() -> ()
Tensor	—	GaBAN Tensor 类型, 表示运行时每一个循环中某一步的数据	无运行时存储

SNN IR 的优势主要体现在以下 3 个方面.

(1) 完备描述 SNN: SNN IR 提供了神经元/突触定义操作与神经元/突触计算操作, 并且还支持扩展新的模型, 因此能够完备地描述 SNN 应用.

(2) 易于移植: 设计参照了 PyNN——PyNN 是一种与模拟器无关的 SNN 通用编程与声明接口, 可以确保该层易移植且为领域开发人员所熟悉.

(3) 便于后续优化: 首先, SNN IR 的设计利于后续优化转化的粗细粒度混合下降——SNN 方言使用粗粒度设计, 这样能够保证对于不同粒度的硬件后端, SNN 方言的算子均能够下降到该后端. 其次, SNN IR 的设计还利于进行神经元合并优化, 参照正文图 3(b1) 与 (b2) 可以看到如下的神经元组合并规则.

● 对于神经元定义算子 (snn.newlif, snn.newizhikevich, snn.newpoisson), 合并时会更改神经元数量为合并后的总神经元数量, 并返回合并后的神经元状态变量 (正文图 3(b1) 中的 %e_v, %i_v 合并后变为图 3(b2) 中的 %a_v);

● 对于突触定义算子 (snn.newstatic), 合并时会更新源神经元与目的神经元的数量, 并返回一个合并后的突触变量 (图 3(b1) 中的 %ee_w, %ei_w, %ie_w, %ii_w 合并后变为图 3(b2) 中 %a_w);

● 对于神经元更新过程 (snn.lif, snn.izhikevich, snn.poisson), 合并时需要将原先的神经元状态变量替换为经过合并后的神经元状态变量 (图 3(b1) 中的 %e_v, %i_v 被替换为图 3(b2) 中 %a_v);

● 对于脉冲传播过程 (snn.static), 合并时也需要将原先的突触变量替换为合并后的突触变量 (图 3(b1) 中的 %ee_w, %ei_w, %ie_w, %ii_w 被替换为图 3(b2) 中 %a_w).

ReRAM IR 的优势除了与硬件后端指令集对应之外, 还利于进行权重微调优化. 由于 ReRAM 方言采用的是粗粒度设计, 因此较为容易进行权重微调优化 (只需要对优化后的权重使用 snn.init_crossbar 将新的权重初始化到 ReRAM 上即可). 此外, 以 crossbar 为单位进行权重初始化还有利于进行权重的结构化剪枝, 减少 crossbar 上的计算, 这将是本工作后续扩展的方向之一.

GaBAN IR 的优势也是类似, 除了与硬件后端指令集对应之外, 其利于高效地进行向量计算. GaBAN 硬件的一个重要特点是支持向量计算, 而 GaBAN IR 很好地使用了硬件向量化的特点, 如表 A3 中的 gaban.sub, gaban.mul, gaban.add 操作的操作数都是 tensor 类型, 保证了生成的代码能够有效地利用 GaBAN 硬件向量化资源. 同时 GaBAN IR 是由第 2 层 IR 转化生成的, 而 GaBAN IR 的计算、访存设计同时与第 2 层 IR 形式类似, 使得这一转换能够高效地进行 (gaban.add 对应 arith.add, gaban.load 对应 vector.load 等).

附录 B. ReRAM 与模拟器简介

阻变存储器 (resistive random-access memory, ReRAM) 是一种新型非易失性存储器件, ReRAM 单元的阻值状

态受过去一段时间内通过的电流影响, 即可以通过不断地向其输入正负向电流来改变该点阻值, 即改变存储内容; 同时又不需要额外电压来维持其存储内容不变, 因此可以实现高密度存储.

利用这一特性, 可以在基于 ReRAM 的交叉开关结构上为神经元突触存储权重并实现脉冲传播计算. 具体的如图 B1 所示: 使用 ReRAM 组成交叉开关结构, 可以实现高效的矩阵-向量乘法运算.

根据矩阵的权值调整交叉开关结构中每个 ReRAM 的阻值, 使得其电导和权值成正比. 脉冲向量由左侧输入, 经过 DAC (数模转换器件) 转换为电压. 根据欧姆定律和基尔霍夫电流定律, 每列输出满足:

$$I_j = \sum G_{ij} V_i.$$

这和矩阵-向量乘法形式一致, 因此将输出的电流经过 ADC (模数转换器件) 转换为数字信号, 即可得到矩阵-向量乘法运算结果. 这种计算形式可以达到很高的并行度、速度和计算密度, 并且功耗较低. 交叉开关结构所支持的最大输入元素个数和输出元素个数与它的高度和宽度相对应.

ReRAM 的主要局限是计算精度较低, 体现在两个方面: 第一, 在设置电导时难以达到精确的电导值, 即由于器件本身的不稳定性以及设置电压可调范围有限, 只能设置到数个离散的电导值附近; 第二, 在计算时环境温度、电磁场以及输入电压本身会对电导值产生影响, 这使得每次计算时 ReRAM 的有效电导值可看作为一个围绕所设置离散电导值的正态分布.

为了缓解这一局限带来的影响, 通常采取以下手段.

● 将输入拆分为逐位输入, 每次计算只计算一位, 保证经过 DAC 之后的电压是二值的, 并将结果移位累加. 由于 SNN 所传播的脉冲信号本身就仅为 0/1, 因此天然就满足这一条件, 且不需要输出时的移位累加.

● 在训练模型时, 考虑到 ReRAM 本身的精度限制, 可以采用的方法是首先使用高精度方法训练模型, 在训练完成后针对特定 ReRAM 器件再进行权重微调, 使权值适应 ReRAM 的计算特性. BIVM 采用这种方法进行精度优化.

ReRAM 的另一个局限是无法计算负权值. 这一局限可以通过略微修改交叉开关的设计, 采用 2T2R (2-transistors 2-resistors) 的结构解决. 在 2T2R 结构的交叉开关中, 每个交叉点上由两个晶体管和两个 ReRAM 组成, 如图 B2 所示. 输出线上的电流 I_{out}, 为流经两个 ReRAM 的差值, 因此在 R_{neg} 上写入权值的绝对值, 在计算中就会表现为负权值.

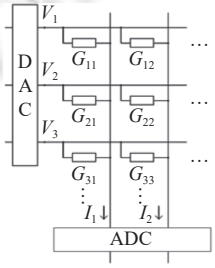

图 B1 交叉开关示意

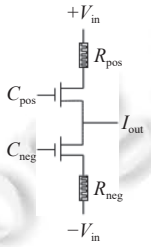

图 B2 2T2R 结构示意

当一个模型需用到多次交叉开关结构进行矩阵-向量乘法时, 需要考虑不同的连接矩阵如何映射到交叉开关结构上. 由于 ReRAM 写入过程十分耗时, 在资源充足的前提下, 可以初始化写入多个矩阵, 并在整个模型的计算过程中以时分复用的方式使用. 以图 B3 所示情况为例, A_1、A_3 共享输入, 对于相同输入可以同时计算. A_1、A_2 共享输出, 必须分两次计算. A_1、A_4 不共享输入和输出, 就可以同时独立计算.

XB-SIM*以时钟驱动方式仿真了上述过程, 即以时钟驱动方式仿真配备了控制逻辑和数据缓冲区的多个处理单元构成的流水线, 每个处理单元的主体是若干个 ReRAM 交叉开关结构. 每个处理单元的大致结构如图 B4 所示, 交叉开关采取上述的 2T2R 结构.

图 B4 中, DAC 负责将数字输入信号转换为交叉开关所需的模拟电压输入, 交叉开关进行计算后, ADC 负责将模拟电流输出采样转换为数字输出. 当数字输入大于一位, 但采取逐位输入时, 需要移位累加模块将部分乘积累加为完整结果. 激活函数和池化层主要用于 DNN 计算任务的加速.

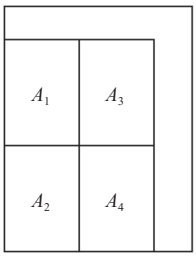

图 B3 矩阵映射示例

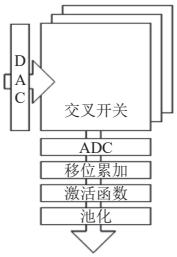

图 B4 XB-SIM*处理单元

在使用 ReRAM 计算 SNN 的发放过程时, 主要需要利用到的处理单元中的组件包括 DAC、ADC 以及交叉开关. 由于脉冲信号本身仅为 0/1, 因此无需使用移位累加单元. 激活函数单元和池化单元在 SNN 中没有对应的结构, 因此也无需使用这些单元. 每次计算将会独占 DAC、ADC 和交叉开关, 考虑 3 个器件各自的可能的延迟相加, XB-SIM*将处理一次 SNN 发放过程计算的总时延设为 100 ns. XB-SIM*中 DAC、ADC 和交叉开关的重要参数包括:

交叉开关规模 w、h: 交叉开关所包含的行数、列数. 这一参数会影响矩阵向交叉开关上的映射方式.

权值存储范围 $W_{\max}$, $W_{\min}$: 交叉开关中每个交叉点存储权值的范围, 取决于器件材料物理特性所决定的最大可容许电导范围. 在采取 2T2R 结构的交叉开关中, 这个范围可以同时包含正数和负数.

权值存储精度 W: 交叉开关中每个交叉点的可能离散电导值个数, 取决于器件材料物理特性所决定的有效电导稳定性. 在采取 2T2R 结构的交叉开关中, 可能的离散电导值个数是单个 ReRAM 的两倍.

以上两个参数会影响交叉开关所能有效表现的权值范围.

ADC 输出精度 ϵ_{out}: ADC 将交叉开关一列的电流转换为读数时, 所能区分的最小电流变化.

ADC 输出位宽 b_{out}: ADC 电流读数的位宽. 超出这一范围的电流读数将会被截断到最大或最小值上.

以上两个参数决定了交叉开关所能有效表现的输出范围. ADC 所能有效表达的电流范围是 $\pm\epsilon_{\text{out}}^{2^{b_{\text{out}}}-1}$, 在电流超出这一范围时会被截断到最大或最小值上.

DAC 输出电压: 当输入为 1 时, DAC 所输出的电压.

杨乐(1997－), 男, 硕士, 主要研究领域为类脑计算, 计算机体系结构.

渠鹏(1991－), 男, 博士, 助理研究员, CCF 专业会员, 主要研究领域为计算机体系结构, 类脑计算.

刘晓义(1999－), 男, 硕士生, CCF 学生会员, 主要研究领域为类脑计算, 计算机体系结构.

崔慧敏(1979－), 女, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为编译优化, 并行编译, 异构编程, 异构编译优化.

李广力(1992－), 男, 博士, 助理研究员, CCF 专业会员, 主要研究领域为异构编程, 编译优化.

张悠慧(1974－), 男, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为高性能处理器微体系结构, 类脑计算芯片, 基础软件.