doi:10.11887/j.issn.1001-2486.25050003 http://journal.nudt.edu.cn

针对控制流计算图的内存优化方法

王向前[1],申彧昊[1],景 琨[1*],吕亚飞[2]
(1. 安徽大学 互联网学院,安徽 合肥 230039;2. 科大讯飞股份有限公司,安徽 合肥 230026)

摘 要:AI芯片在深度学习应用中受限于片上内存容量,当前主流内存优化方法针对静态计算图,对动态计算图的内存优化存在进一步的优化空间。针对该问题,提出一种控制流计算图模型的内存优化框架,在子图内部实现内存复用的基础上,结合控制流特性递归进行子图间的内存复用。针对片上与片外内存的内存墙问题,针对控制流计算图的权重数据提出一种有效的乒乓缓存实现策略,在子图内部实现访存和计算操作的重叠执行。基于国产LUNA AI芯片进行验证,结果表明,该内存优化框架实现了控制流计算图的片上内存优化使用,相比原有方法进一步提升5.9%。该策略有效解决了内存墙问题,减少了片上片外内存的数据传输时间,计算图的执行效率最高提升29%。

关键词:AI芯片;内存优化;内存重用;跨内存传输
中图分类号:TP18 **文献标志码**:A **文章编号**:1001-2486(2025)06-071-10

论文拓展

Memory optimization method for control flow computation graph

WANG Xiangqian[1], *SHEN Yuhao*[1], *JING Kun*[1*], *LYU Yafei*[2]
(1. School of Internet, Anhui University, Heifei 230039, China; 2. iFLYTEK Co., Ltd., Heifei 230026, China)

Abstract: AI chips face on-chip memory limits in deep learning. Current optimization methods focus on static computation graphs, leaving room to improve memory efficiency for dynamic graphs. To overcome this limitation, a memory optimization framework for control-flow computation graphs was developed. The framework realized operator-level memory reuse within subgraphs and further achieved recursive reuse across subgraphs by exploiting control-flow characteristics. In addition, a ping-pong buffering strategy for weight data was introduced to mitigate the memory wall between on-chip and off-chip memory, thereby allowing overlapping of memory access and computation operations within subgraphs. Validation on the domestic LUNA AI chip has demonstrated that the proposed framework improves on-chip memory utilization by 5.9% compared with existing methods. Moreover, the strategy effectively alleviates the memory wall problem by reducing data transfer time between on-chip and off-chip memory, resulting in execution efficiency improvements of up to 29%.

Keywords: AI chip; memory optimization; memory reuse; cross-memory transfer

随着人工智能和深度学习技术的发展,研究边缘端场景下深度学习模型快速部署技术以及相关优化方法成为研究的热点。神经网络模型的参数量和数据量的大规模上升,导致传统的通用系统级芯片(system on chip,SoC)平台架构已无法满足模型训练对速度以及内存的需求,使用人工智能(artificial intelligence,AI)芯片[1]加速神经网络计算成为一种流行选择。

考虑到性能以及成本等因素,AI芯片往往提供高速但容量较小的片上内存,低速但容量较大的片外内存。基于此,神经网络模型往往把算子参数比如权重数据放置在片外内存,计算图模型的中间结果张量放置在片上内存[2],以避免通信开销,而算子参数数据较大,只有在该算子计算时才会使用,一般放置在容量大但是读取速度慢的片外内存,导致了参数读取时的内存墙问题。

要实现在AI芯片上高效部署深度学习模型,一般需要AI编译器或者推理引擎进行计算图的优化,包括公共子表达式删除、算子融合、内存优

收稿日期:2025-05-06
基金项目:国家自然科学基金资助项目(62406003)
第一作者:王向前(1985—),男,河南南阳人,副教授,博士,硕士生导师,E-mail:wangxiangqian@ahu.edu.cn
***通信作者**:景琨(1996—),男,山东济南人,讲师,博士,E-mail:jingkun@ahu.edu.cn
引用格式:王向前,申彧昊,景琨,等. 针对控制流计算图的内存优化方法[J]. 国防科技大学学报,2025,47(6):71-80.
Citation: WANG X Q, SHEN Y H, JING K, et al. Memory optimization method for control flow computation graph[J]. Journal of National University of Defense Technology, 2025, 47(6): 71-80.

化等,最终转换为能够在目标硬件运行的高效代码。本文所研究的内容即为针对控制流计算图的内存复用以及优化使用方法,其所描述的框架既适用于 AI 编译器,同样也适用于推理引擎。

要实现高效的神经网络模型的部署,必须考虑模型计算过程的特点和硬件平台的体系结构特征。模型在推理的过程中需要产生大量的中间张量[3],例如,Inception v3[4] 的中间张量占用了总运行时内存的 37%,而 MobileNet v2[5] 的中间张量占用了 63%;模型算子参数较大,一般放置在容量大但是读取速度慢的片外内存,导致了参数读取时的内存墙问题。考虑到 AI 芯片提供的高速但容量有限的片上内存,中间结果张量需要进行高效的管理;模型参数引起的内存墙问题需要得到有效的缓解。

动态内存分配器可以用于管理运行时内存的分配和释放,PatrickStar[6] 将模型数据进行分块,以减少通信并提升带宽利用率,在基于中央处理器(central processing unit,CPU)和图形处理器(graphics processing unit,GPU)协同的异构内存中进行动态分配,并根据内存使用情况进行调整。然而频繁分配和释放内存不仅影响运行时的效率,还需要通过内存碎片整理和垃圾回收来处理碎片和防止内存不足,这都带来了更多的开销。为了避免这些问题,需要减少内存分配释放的次数,还可以采取静态分配的方式,编译时推断出每个中间张量的形状,在运行模型之前分配中间张量的内存缓冲区。

通过重计算(一些中间结果在需要时选择重计算而不是存放在内存中)可以有效减少中间缓存,使用动态规划可以探索重计算中间结果和缓存中间结果的平衡点[7],但是该种方法实现代价较高。MXNet[8] 探索了其他技术来减少内存消耗,包括判断节点的依赖并及时释放内存,以及中间张量内存共享,通过分析计算图中的中间结果和其生命周期,将不同时刻使用的中间张量分配到相同的内存块,该算法可以有效地调度内存,但其主要侧重于安全性和稳定性,尽管这些优化策略在内存减少和并行调度上提供了帮助,但是没有深入探讨更高级的内存管理策略,没有探索能够以最有效的方式解决这一问题的不同算法。

Chen 等[9] 在 GPU 上实现提前计算来减少使用内存的算法,探索如何降低在训练更复杂的模型时中间特征和梯度的内存成本。Pisarchyk 等[10] 提出了启发式算法来预先规划共享对象的内存分配,以便在 GPU 硬件中有效使用,还提出了贪心算法的一种变体,优化了深度神经网络(deep neural network,DNN)在 GPU 中的内存使用,有向无环图(directed acyclic graph,DAG)被转换为辅助图,在辅助图上使用最小成本流算法,然后回到原始图来完成内存规划。王鑫等[11] 针对国产深度计算处理器(deep computing unit,DCU)研究了本地数据共享(local data share,LDS)的分配方法和高效访问,完成了片上内存的自动分配。尽管上述算法有效减少了内存使用,但是这些算法针对 GPU 等架构设计,不适合部署在边缘端设备。

Lee 等[12] 在深度学习框架 TFLite 中使用内存管理器管理缓冲区,提出了两种类似于寄存器分配的内存管理方法;由于内存管理的张量大小不等,需要更复杂的分配方案,这种内存管理关注缓冲区的有效利用。许鹏等[13] 提出 TFLite 内存复用的改进算法,对同尺寸的张量进行排序,减少了内存碎片的产生,同时利用内存的多层次架构分别存储在不同的内存空间,提高内存的利用率。Sekiyama 等[14] 将内存分配问题抽象为二维矩形装箱问题的特例来解决,以有效处理张量的分配和重用。曹博钧等[15] 提出了一种即时内存复用的算法,在效率不低于当前主流算法的基础上,在节点更密集的图中节省了内存。

上述提到的针对内存复用优化以及多层次存储管理的研究内容是对静态单个图的优化,未考虑对包括多个子图的控制流计算图进行内存复用优化。随着深度神经网络已经成功地应用于越来越多的领域,控制流概念被引入深度学习[16-19]。Niu 等[20] 通过秩与维度传播推断张量形状,将动态计算图划分为多个子图,在子图中进行静态内存规划,但是该静态内存规划未利用动态计算图的特性进一步优化。Zhang 等[21] 使用 uTask 作为程序的基本单元,用于表示计算图中的控制流和数据流。在分支 uTask 中,分支张量的中间结果共享内存空间。Ma 等[22] 提出张量的块传播机制,将控制流分析与内存依赖分析结合,实现跨分支的内存合并。但这些工作并未对控制流计算图内存复用优化的具体实现进行深入探讨。

针对以上问题,本文提出一个通用的含控制流计算图的内存复用优化算法,实现对含控制流计算图在执行过程中的内存占用。同时设计含控制流计算图的内存复用多层次存储优化算法,提升了高速存储的利用率,提高模型的执行效率。基于开源编译框架实现以上方法,并通过实验验证以上方法的有效性。

1　算法设计

1.1　基于内存访问冲突图的内存复用算法

在计算时,大部分中间张量在片上内存的存储时间非常短,并不需要长期占用内存,如果两个张量的生命周期互不重叠,则它们的内存复用是安全的。因此基于内存访问冲突图复用已分配的内存,可以显著提高内存空间的利用率。具体通过分析张量的生命周期和内存使用的时序特性[23],最大限度地减少了内存的空闲时间,通过智能分配实现多个张量的内存空间在时间上互不冲突的复用。神经网络模型可以用有向无环图表示,这种表示形式也称为计算图。计算图的节点表示算子,弧表示算子之间传递的张量。在分析计算图时,需要对图中的所有节点进行拓扑排序,为每个节点分配一个唯一的自然数,作为该节点的执行顺序。

$$L(\boldsymbol{T}) = [t(u), \max(t(v_i))] \quad i \in (1,k) \tag{1}$$

式(1)展示了张量 $\boldsymbol{T}$ 的生命周期的计算方法。假设节点 u 的执行顺序为 $t(u)$,张量 $\boldsymbol{T}$ 在该节点上产生,并在后续 v_i 节点中被使用,那么 $L(\boldsymbol{T})$ 即为该张量的生命周期范围。

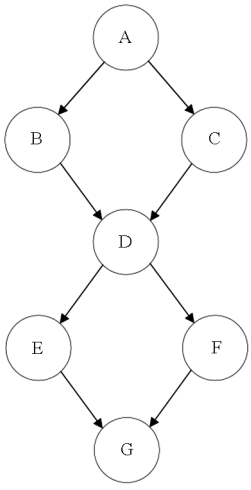
$$m(\boldsymbol{T}_i) \cap m(\boldsymbol{T}_j) = \begin{cases} \max[m(\boldsymbol{T}_i), m(\boldsymbol{T}_j)] & L(\boldsymbol{T}_i) \cap L(\boldsymbol{T}_j) = \varnothing \\ \min[m(\boldsymbol{T}_i) + d(\boldsymbol{T}_i), m(\boldsymbol{T}_j) + d(\boldsymbol{T}_j)] & L(\boldsymbol{T}_i) \cap L(\boldsymbol{T}_j) \neq \varnothing \end{cases} \tag{2}$$

式(2)展示了张量之间内存复用的计算方法。m 表示张量占用的内存空间,设 $L(\boldsymbol{T}_1)$ 和 $L(\boldsymbol{T}_2)$ 分别为两个张量 $\boldsymbol{T}_1$、$\boldsymbol{T}_2$ 的生命周期,$\boldsymbol{T}_1$ 的内存空间先于 $\boldsymbol{T}_2$ 建立,当 $L(\boldsymbol{T}_1) \cap L(\boldsymbol{T}_2) = \varnothing$ 时,它们的生命周期不重叠。因此在分配内存空间时可以复用对方的内存空间,从而实现两者的内存共享,否则需要基于 $\boldsymbol{T}_1$ 或 $\boldsymbol{T}_2$ 的内存基址进行偏移来保证内存空间不冲突,从而确保数据的安全。通过计算偏移量 d 避免了两个张量的生命周期冲突,同时比较两个张量的偏移量以获得最小的总内存占用。

考虑无控制流的深度学习模型的内存复用策略。在一个计算图中存在大量算子产生的张量时,这些张量占用的内存空间的复用情况较为复杂,按一定的方式对张量进行排序可以提高内存复用的效果,提前为一部分张量分配内存空间,后续张量在与已分配内存空间的张量不冲突的情况下复用后者的内存空间。对这些算子采用不同的复用策略会达到不同的优化效果,如顺序复用算法、大张量优先复用算法[14]以及短张量优先复用算法[24],如图 1 所示。本文对三种内存复用算法进行介绍并提出改进思路。

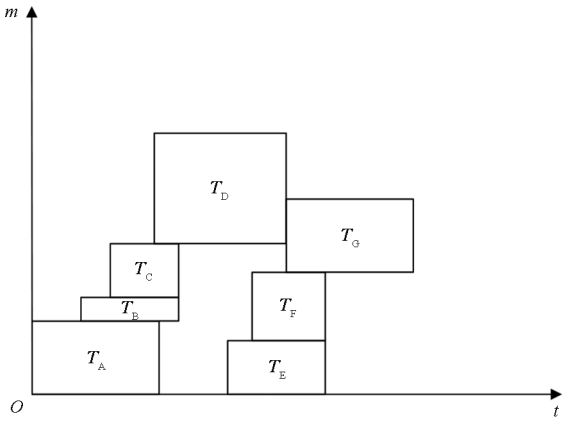
顺序复用算法最早提出,创新性地分析张量的生命周期,让非相交张量复用内存空间。该算法首先按计算图中每个节点的拓扑排序确定算子的执行顺序,根据执行顺序确定每个张量的生命周期。接着按算子的执行顺序依次为算子产生的张量分配缓冲区。当待分配的缓冲区与已分配的缓冲区生命周期不重叠时,可以复用之前已分配缓冲区中的内存,并记录当前分配缓冲区的偏移量,在分配的过程中不断根据缓冲区的大小和偏移量计算并更新使用的峰值内存。在对图 1(a)中的基本计算图执行该算法后,张量分布如图 1(b)所示。

在执行上述流程之后,得到使用的峰值内存大小以及每个算子使用的缓冲区的偏移量和内存长度,再分配一块峰值内存大小的统一缓冲区。算子在执行的过程中,根据缓冲区的起始地址和



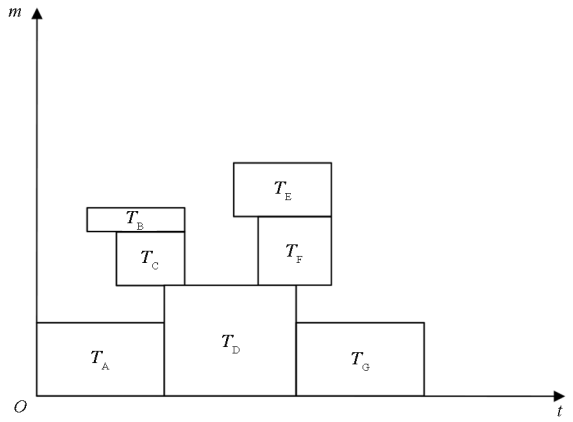
(a) 计算图

(a) Computation graph



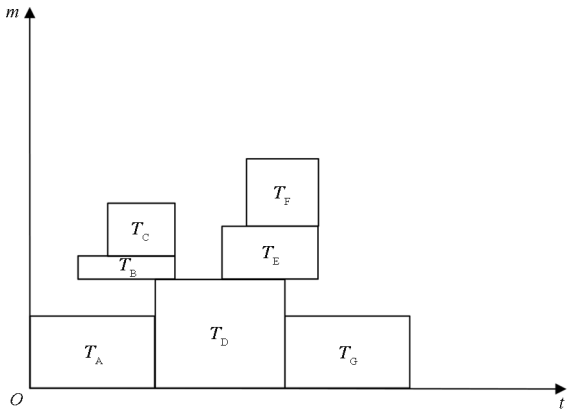
(b) 顺序复用算法

(b) Sequential reuse algorithm



(c) 大张量优先复用算法

(c) Large tensor priority reuse algorithm



(d) 短张量优先复用算法

(d) Short tensor priority reuse algorithm

图1 计算图不同策略复用结果

Fig. 1 Reuse results of different strategies in the computation graph

偏移量得到对应的片上内存并使用,实现了计算图在执行过程中所需片上内存的统一分配,在提高片上内存利用率的同时减少了内存碎片的产生。

顺序复用算法虽然复用张量的内存以提高内存利用率,但是对内存布局进行改进后,有机会得到占用内存更小的最优内存布局。当计算图较大时,通过暴力计算所有布局来实现最优布局是不可行的,当前使用较多的优化方案为大张量优先复用算法。大张量优先复用算法属于启发式算法,通过局部贪心结合生命周期分析以实现近似内存最优分配,在牺牲部分全局最优性的条件下,换来较高的运行效率与可部署性。该算法在根据算子的执行顺序获取张量的生命周期后,对张量按大小进行降序排列,优先为更大的张量分配内存空间,通过优先复用大张量的内存,可以让大张量尽量使用空闲内存块,最大化利用已释放的大块内存。避免小张量分割大块内存后,大张量需要额外申请新的内存,减少碎片化,提高整体复用率,优化后该算法的张量分布如图1(c)所示。

短张量优先复用算法的核心思想是优先回收生命周期短的张量,使释放的内存尽快被后续的张量使用。相较于大张量优先复用算法,该算法针对使用内存池机制管理的内存空间进行优化,由于内存池机制对内存空间进行了分块,当较长生命周期的张量在内存中保留时间过长时,后续的短生命周期算子只能在当前内存中剩余的空间内进行分配,若当前内存中剩余空间不满足分配条件,需要重新开辟一个内存空间,无法实现内存空间的连续分配,而且频繁的分配和释放操作会导致计算的执行效率降低。该算法对张量按生命周期的长度进行升序排列,优先为更短的张量分配内存空间,使得生命周期更短的张量在低地址进行内存空间的连续分配,减少内存碎片化,执行该算法后的张量分布如图1(d)所示。

将图1进行对比,发现张量复用优化的策略中,大张量优先复用算法得到了更优的效果,但是其他策略在不同的特定情况下均可能达到最优布局。因此将复用策略抽象为接口,通过接口可以灵活选择策略,实现了一个可以使用多种复用策略的内存分配方法。该复用方法也可以在子图中使用,实现子图内部张量复用,为后续优化打下基础。

在计算图中,需要遍历节点以收集张量信息。算法1实现了在计算图上的内存复用,首先初始化张量映射队列,该数据结构内收集了张量的大小,首次使用位置和最后一次使用的位置,通过这些可以获取生命周期以及后续分配时的空间大小。在第1行获得图内算子的定义-使用链,通过该结构获得每个算子的输出被使用的位置,以此获取张量的生命信息。在第4行收集每个张量的具体信息并存放在张量映射表内,之后将每个映射表入队。随后,在第7行对收集到的张量按输入的优化策略进行排序。排序完成后,为队列内首个元素,即映射表内的张量分配初始偏移量(一般为0)作为缓冲区空间的起始地址,分配结束后出队,并加到已分配张量的集合内,视为该张量在片上内存的分配缓冲区。后续将遍历未分配的张量队列,计算偏移量,即获得缓冲区的位置。依次检查每个张量的生命周期是否与已分配张量发生冲突。如果没有冲突,则按当前的起始地址为该张量分配内存空间,否则起始地址从低到高偏移直到生命周期不重叠。获取偏移量的同时计算内存占用峰值大小并更新。后续将根据峰值大小建立一整块连续的内存空间供所有张量使用。

算法 1　内存复用

Alg. 1　Memory reuse

输入:计算图,优化策略
输出:优化后的计算图

1. 获得计算图中每个节点的 Userange 信息
2. **for** (∀b∈region) **do**
3.　**if** (节点 b 表示张量) **then**
4.　　收集 b 的 Userange 到张量映射队列 AllocInfo
5.　**end if**
6. **end for**
7. 对张量映射队列按不同的策略进行排序
8. 为队列首个元素分配初始偏移量并出队
9. 将出队的首个元素和偏移量加入已分配表 Alloced
10. **for** (∀I∈AllocInfo) **do**
11.　根据当前张量信息和已分配的张量计算偏移量
12.　将张量和偏移量加入已分配表 Alloced
13.　更新内存峰值
14. **end for**
15. 根据内存峰值分配统一张量

1.2　支持控制流的内存复用优化算法

在控制流的子图内进行内存复用仍采用上述的方法,基于该方法,添加对控制流的支持。

当节点中存在分支控制流时,对于分支控制流算子,张量可能有多条执行路径,导致难以静态推断张量的使用情况。例如图 1(a)中的 B 和 C、E 和 F 分别属于分支控制流算子控制下的不同分支,每个分支可以视为一个子图,程序在运行时会根据条件只选择其中一个子图执行[25-27],为其他子图分配的内存空间处于空闲状态,这种特性为内存优化提供了潜在的机会。在这种情况下,可以对子图的内存空间进行进一步复用,如图 2 所示。这种优化策略的核心在于对子图内存的合理复用和统一管理,具体来说,对于同一个控制流算子控制下的不同子图,可以在每个子图内部进行一次内存复用,计算出该子图的峰值内存占用作为该子图的内存大小。如图 2 所示,基于图 1(c)大张量优先复用算法进行改进,比较子图之间的内存并复用,保留 $\boldsymbol{T}_{\mathrm{C}}$ 的张量空间供两个分支内的张量 $\boldsymbol{T}_{\mathrm{B}}$ 和 $\boldsymbol{T}_{\mathrm{C}}$ 使用,保留 $\boldsymbol{T}_{\mathrm{F}}$ 的张量空间供两个分支内的张量 $\boldsymbol{T}_{\mathrm{E}}$ 和 $\boldsymbol{T}_{\mathrm{F}}$ 使用。将其分别作为该控制流算子的张量空间,当张量从子图提取到外层后,可以与外层的张量进行复用。从图 2 中虚线的对比可以看出,经过该优化后内存占用峰值减小。由于控制流之间可能存在嵌套,需要从最内层开始进行子图的复用和外提,对于每个子图和张量的深度,同一深度的张量之间才能进行复用操作,从而实现多层嵌套的情况下分支控制流计算图的内存复用优化。算法 2 为控制流优化内存复用算法,在算法 2 的第 4 行通过检查控制流中的条件语句执行分支控制流算子的内存复用优化策略。考虑到控制流中可能存在多层嵌套的复杂结构,在第 5 行递归地从最内层的控制流算子子图开始优化。这一递归过程确保了对每个嵌套控制流中的张量缓冲区的复用能够逐层推进,从而避免遗漏深层次的优化机会。在第 6 行,当子图内部的张量缓冲区复用完成后,对每个子图的缓冲区进行整合,通过比较统一控制流算子的不同子图张量空间需求,选择最大内存作为子图之间共享缓冲区的基准。基于这一基准,算法在这些子图的外层为该控制流算子建立新的缓冲区。新建立的缓冲区可以基于与这一层的张量缓冲区进行复用来提升内存空间的利用率。

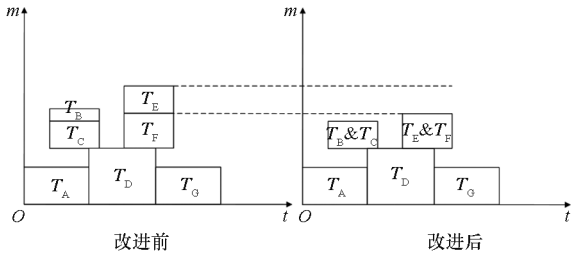


图 2　分支控制流节点复用优化结果对比

Fig. 2　Comparison of node reuse optimization results in branch control flow

算法 2　控制流优化内存复用算法

Alg. 2　Control flow optimization memory reuse algorithm

输入:控制流节点,计算图
输出:优化后的计算图

1. 收集输入控制流节点产生的子图 Sub_blocks
2. **for** (∀b∈Sub_blocks) **do**
3.　**for** (∀v∈b) **do**
4.　　**if** (节点 v 表示控制流节点) **then**
5.　　对该节点递归执行 CFOpt
6.　　比对子图内存峰值并在子图外层建立统一内存
7.　　**end if**
8.　　在外层执行 MemoryReuse 进行内存复用
9.　**end for**
10. **end for**

1.3　基于乒乓缓存方式的存储优化算法

直接内存访问(direct memory access, DMA)

是一种高效的数据传输机制,用于在外设与存储器之间以及各存储器之间提供高速数据传输。相较于将计算图的数据交换到外部的主机内存[28-30],DMA 可以在无须 CPU 介入的情况下进行数据传输,从而释放 CPU 资源,使其可以执行其他计算任务,提高系统的整体吞吐量。

基于 DMA 传输时不占用 CPU 资源的特性,使用乒乓缓冲时间进行数据传输和计算。乒乓操作是一种数据缓冲优化设计技术,可以看成是另一种形式的流水线技术,其核心思想是在两块缓存区(buffer)之间交替进行数据传输和计算,从而最大化数据流处理效率,减少等待时间,具有节约缓冲空间、对数据流无缝处理等优点。具体工作流程如下:在 DMA 启动时,由于两块缓存区均为空,在缓存区 A 执行数据流写入(从片外存储加载数据)。缓存区 A 数据写入完成后,立即启动缓存区 A 数据读取进行计算,同时缓存区 B 开始执行数据写入。计算(缓存区 A 读取)和数据搬移(缓存区 B 写入)并行执行,减少等待时间。当计算以及下一个算子需要的数据写入结束后,两者交换角色,缓存区 A 开始写入新数据,缓存区 B 读取数据进行计算,如此一直循环下去。

为了优化 DMA 数据传输和计算并行执行,本文设计了一种基于乒乓缓存方式的存储优化算法,并将其应用于神经网络模型的计算任务调度。该算法通过在片上高速共享内存(SharedMemory)中选择两块全局 DMA 内存(alloc0 和 alloc1)进行数据交替传输,从而实现高效的数据流管理,如图 3所示。

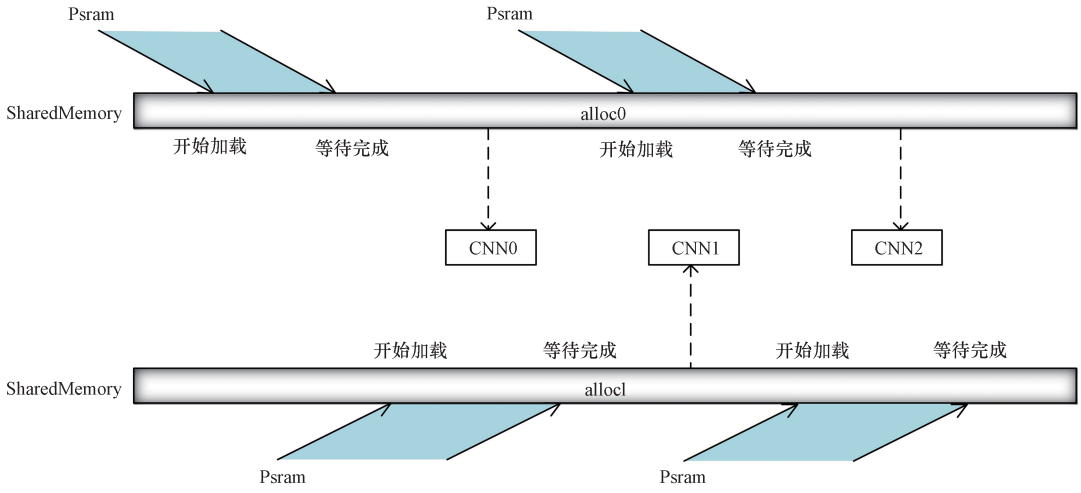


图 3 DMA 异步插入

Fig. 3 DMA asynchronous insertion

在无控制流的模型中,算子通过 DAG 的拓扑排序顺序执行,在节点计算前,从大容量低速外部内存(Psram)中使用 DMA 传输数据到 DMA 内存 alloc0,等待数据传输结束后,在节点 CNN0 开始读取 alloc0 中的数据进行计算前,启动 DMA 从片外内存将下一个节点 CNN1 所需的数据传输到另一个 DMA 内存 alloc1,后续交错执行该步骤。因此在 DAG 中顺序执行节点的过程中,序号为偶数的节点使用 DMA 内存 alloc0,序号为奇数的节点使用 DMA 内存 alloc1。

在 DMA 的数据传输过程中,DMA 的启动和结束由特定的 DMA 控制指令来完成,确保数据能够正确加载,并且计算过程不会因数据未准备就绪而发生错误。①DMA 启动(DMAStart):向 DMA 内存中加载数据。②DMA 等待(DMAWait):等待当前 DMA 传输完成后,执行后续的语句。因此每个算子需要在自己和上一个 DMAWait 之间插入下一个算子的 DMAStart,在算子执行结束后插入下一个算子的 DMAWait。

算法 3 实现了 DMA 异步插入。收集模型内每个节点需要的存储在 Psram 中的权重等信息,getglobalOp 作为指针指向 Psram 中 globalOp 数据的内存地址,以卷积算子为例,卷积计算的权重(Weight)和偏置(Bias)存放在 Psram 并通过 globalOp 表示为全局的数据。在通过指针获得 globalOp 中的数据后,使用 viewOp 根据偏移量和数据长度对获取的权重数据进行拆分,提取 Weight 和 Bias。并使用 expendshapeOp 维护数据的维度信息,确保计算节点可以正确访问和使用权重数据。该算法首先顺序遍历计算节点,根据节点的定义-使用链,获得 getglobalOp,权重数据的地址。在第 4 行中将权重数据的地址收集到

算法 3　DMA 异步插入

Alg. 3　DMA asynchronous insertion

输入:计算图,张量集合
输出:计算图

```
1. 初始化算子 Map
2. for ∀v ∈ moduleOp do
3.    for ∀u ∈ use_def_chain_of v do
4.       收集算子的形状信息和权重数据地址到 Map 内
5.    end for
6. end for
7. for (∀v ∈ funcOps) do
8.    收集张量所在 Region 内的奇偶序列最大内存
9.    建立 DMA 内存
10.   for (∀v ∈ Region) do
11.      if (存在控制流算子 CFOp) then
12.         for (∀Region ∈ CFOp.SubRegion) do
13.            递归执行
14.         end for
15.      end if
16.   end if
17.   插入 DMA 指令
18. end for
```

Map 中,Map 中节点的顺序即为算子执行的顺序,按执行顺序加载 globalOp(权重数据)到 DMA 内存块。在奇序列和偶序列中,分别找出权重数据的最大大小,作为该序列所使用的 DMA 内存块的大小,这确保了最大权重数据也能成功加载,避免数据溢出。在得到两个 DMA 内存的信息后,为每个计算节点插入 DMA 指令实现在无控制流计算图下的权重从 Psram 加载到片上内存的过程。

在包含分支控制流的计算图中,每个子图内部算子数量不同并且子图的计算过程是相互独立的。由于模型在执行过程中,仅会选择一个分支执行,因此无法像无控制流的计算图那样依据全局拓扑排序的奇偶序列来确定 DMA 传输的顺序,节点的执行顺序在运行时是不确定的。针对上述问题,算法基于控制流分析,将计算图进行划分,在编译框架内使用 Region 表示计算图中的可执行子图,Region 内可以包含多个 BasicBlock,同时一个 Region 可以被继续划分为多个 Region。划分规则如下:①从当前图的起始节点到第一个控制流节点之间为一块 Region;②控制流节点产生的每个子图单独划分为一个 Region;③控制流子图结束位置到下一个控制流节点之间为一个 Region;④当 Region 内部存在控制流节点时,递归执行①~③,直到所有子图划分完毕;⑤从最后一个最外层的控制流子图结束位置到计算图的结束作为一块 Region。每个 Region 内执行上述异步内存传输,如图 4 所示。

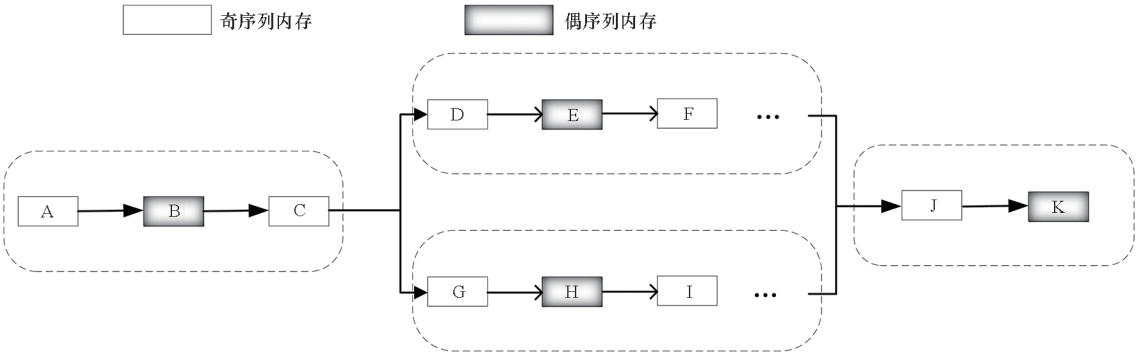


图 4　控制流计算图内执行 DMA 插入

Fig. 4　DMA insertion in control flow graph

在算法 3 中,第 11 行判断是否存在分支控制流节点,若不存在则为无控制流图,直接执行 DMA 插入操作。若存在控制流节点,递归执行计算图的 Region 拆分,直到拆分完成,在最终拆分的每个 Region 内执行 DMA 插入操作。

2　实验数据对比及分析

2.1　实验平台介绍

LUNA[31]是一款国产 AI 芯片,在 LUNA 芯片的内存架构中,内存管理策略采用多层次架构,结合了 SharedMemory 和 Psram,以兼顾性能与存储容量需求。模型节点所需的输入、输出张量保存在 SharedMemory 上,这样中间结果可以快速存储并传递给下一个节点。而模型的输入、输出节点信息以及权重等参数不会随着模型计算的过程被释放,同时数据量较大,存放在 SharedMemory 上不仅超过了内存的限制,还造成了资源的浪费,需要存放在 Psram 上,在需要时进行加载。

本节基于 LUNA AI 芯片开发板进行实验评估,实验平台采用 CentOS Linux release 7(x86_64)平台,基于 LLVM16.0 对模型进行编译和生成可执行文件,并使用 Xtensa 工具链进行执行和调试。根据模型执行的效果对提出的算法进行实验验证和效果测试。

2.2 模型的内存复用效果

本节通过模型编译过程中最终使用的峰值内存评估了管理中间张量的三种算法——大张量优先复用算法、短张量优先复用算法、顺序复用算法,并对比了其控制流优化算法的效果,对比结果见表1、表2。Naive 分配所有必需的内存,并且仅作为比较的基线。FaceAlgin、FaceDetect 是不包含控制流的神经网络模型。FusionDetect 是将 BodyDetect 和 FaceDetect 裁剪后组合的模型,用于测试算法优化效果。FusionDetect、唤醒和降噪模型为包含控制流的神经网络模型。

表1 不同模型优化前的效果

Tab. 1 Effect of different models using the algorithm before optimization

单位:B

模型	原始内存	大张量优先复用算法	短张量优先复用算法	顺序复用算法
FaceDetect	934 000	409 600	409 600	440 320
FaceAlgin	202 429	46 080	46 668	46 668
唤醒	312 664	24 152	24 152	24 152
FusionDetect	26 502 400	489 600	489 600	604 800
降噪	794 128	99 840	117 760	142 848

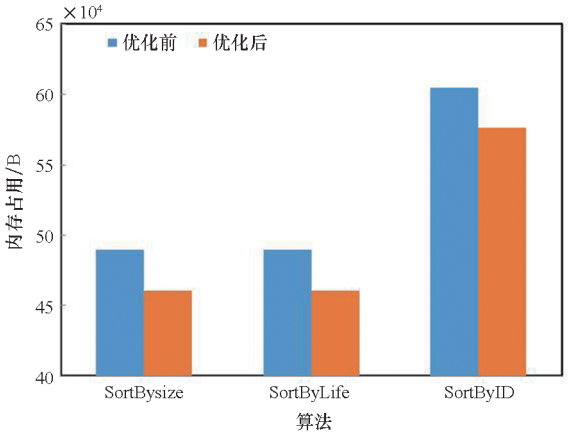
表2 不同模型使用控制流优化算法的优化效果

Tab. 2 Optimization effect of different models using control flow optimization algorithm

单位:B

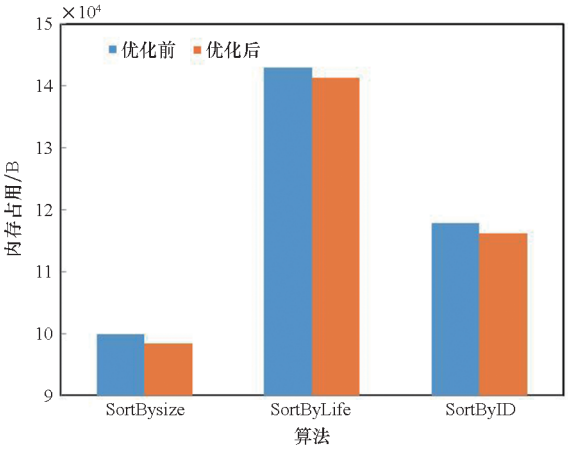
模型	大张量优先复用算法	短张量优先复用算法	顺序复用算法
唤醒	24 152	24 152	24 152
FusionDetect	460 800	460 800	576 000
降噪	98 340	116 224	141 312

图5展示了三个控制流模型在执行支持控制流的复用优化策略后的内存峰值占用情况与传统的算法对比。图5(a)为 FusionDetect 模型优化效果,图5(b)为降噪模型优化效果,从图中可以看到,该策略在这两个模型内均进一步降低了峰值内存占用量,最高优化效率达到5.9%。由于唤醒网络中不同分支之间内存差距过大,即使与另一个分支进行复用,复用的内存区域未能有效地减少原有分支的内存需求,因此未能实现优化。



(a) FusionDetect 模型优化效果

(a) FusionDetect model optimization effect



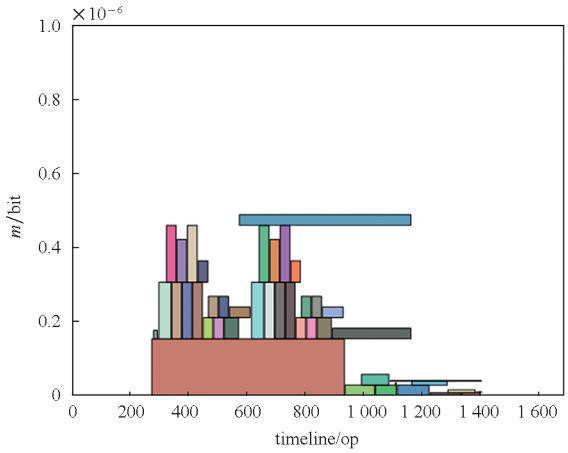
(b) 降噪模型优化效果

(b) Denoising model optimization effect

图5 优化效果对比

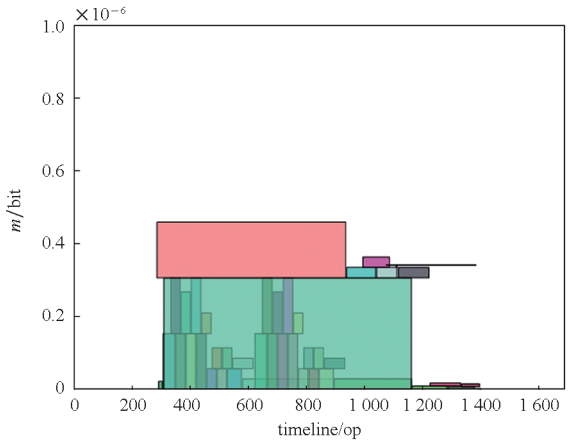
Fig. 5 Optimization effect comparison

图6展示了 FusionDetect 模型在优化前后的内存分配,展示了该模型所有张量。图6(a)为使用大张量优先复用策略后内存空间的分布图,两个分支内的张量之间的生命周期重叠后无法进行内存复用,内存分配较为分散。支持控制流的优化策略将两个分支的内存复用后分配一块大内存供两个分支内的节点使用,将两个分支的张量合并后,使用了绿色的内存块表示两个分支张量使用的内存空间,且峰值内存有所下降,如图6(b)所示。



(a) FusionDetect 模型控制流优化前内存分配

(a) Memory allocation before FusionDetect model control flow optimization



(b) FusionDetect 模型控制流优化后内存分配

(b) Memory allocation after FusionDetect model control flow optimization

图 6　FusionDetect 内存分配

Fig. 6　FusionDetect memory allocation

2.3　模型异步跨内存传输效果

本节基于不同的模型在执行时的计算效率,对 DMA 异步传输优化进行评估,该方法实现了在含控制流的模型中的 DMA 异步传输,将使用该方法后模型的计算效率与同步进行数据传输和计算的模型计算效率进行比较,结果如表 3 所示。使用该方法后,模型的执行效率均得到了不同程度的提升,尤其在 FaceAlign 和唤醒模型中优化效果较为显著。其主要原因在于这两个模型中算子尺寸较大,在插入 DMA 传输后显著节省了内存传输开销,最高可节省约 29% 的时间。

表 3　模型异步插入 DMA 优化结果

Tab. 3　Model asynchronous insertion DMA optimization results

模型	原始时间/ms	异步优化后的时间/ms	优化效率/%
FaceAlign	2 411.414	1 712.770	28.97
FaceDetect	4 978.807	4 342.279	12.78
BodyDetect	1 712.77	1 702.147	0.62
FusionDetect	6 006.455	5 129.129	14.61
唤醒	2 229.174	1 695.734	23.93

3　结论

神经网络编译器在支持复杂的含控制流神经网络模型时会遇到性能问题。控制流的多个分支会造成额外的内存开销。本文实现了一种通用的策略,可以有效对控制流模型内的子图进行内存复用,减少模型计算过程中的内存峰值占用。结果表明,在控制流模型内可以达到很好的复用优化结果。本文通过实验有效地证明了控制流模型提高内存优化的可行性,基于原有的多种复用算法最高优化效率达到 5.9%。对于循环控制流算子,张量可能在循环中被使用,并且有可能在循环外部继续被使用,在包含循环控制流算子的 DAG 内计算张量生命周期时,分析在图中会产生循环,在单次执行结束后,张量可能向下继续传递或回到循环的起始位置,基于该特性,当张量在循环内部使用时,生命周期计算将会包含循环的生命周期,后续将基于这些进一步开展研究。

本文还针对 AI 芯片片上内存和片外内存的内存墙问题提出一种有效的乒乓缓存实现策略,支持对控制流模型的多层次内存管理,在每个子图的内部单独进行乒乓缓存,在子图内部实现访存和计算操作的重叠执行,提高了高速内存的利用率。实验结果表明,该算法对模型计算效率提升明显,最高可达到 29%。但是当图内中间张量较少时,多层次内存传输的启动时间可能会导致模型的计算效率变低。

参考文献(References)

[1] LI B. Embedded AI accelerator chips[M]//LI B. Embedded artificial intelligence. Singapore: Springer Nature Singapore Pte Ltd., 2024: 121 - 180.

[2] 马玮良, 彭轩, 熊倩, 等. 深度学习中的内存管理问题研究综述[J]. 大数据, 2020, 6(4): 56 - 68.
MA W L, PENG X, XIONG Q, et al. Memory management in deep learning: a survey[J]. Big Data Research, 2020, 6(4): 56 - 68. (in Chinese)

[3] XIA C W, ZHAO J C, SUN Q Q, et al. Optimizing deep learning inference via global analysis and tensor expressions[C]//Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2024: 286 - 301.

[4] SZEGEDY C, VANHOUCKE V, IOFFE S, et al. Rethinking the inception architecture for computer vision [C]// Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016: 2818 – 2826.

[5] SANDLER M, HOWARD A, ZHU M L, et al. MobileNetV2: inverted residuals and linear bottlenecks[C]// Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018: 4510 – 4520.

[6] FANG J R, ZHU Z L, LI S G, et al. Parallel training of pre-trained models via chunk-based dynamic memory management[J]. IEEE Transactions on Parallel and Distributed Systems, 2023, 34(1): 304 – 315.

[7] GRUSLYS A, MUNOS R, DANIHELKA I, et al. Memory-efficient backpropagation through time [J]. Advances in Neural Information Processing Systems, 2016, 29: 4125 – 4133.

[8] CHEN T Q, LI M, LI Y T, et al. MXNet: a flexible and efficient machine learning library for heterogeneous distributed systems[EB/OL]. (2015 – 12 – 03) [2025 – 04 – 20]. https://arxiv.org/abs/1512.01274? context = cs.MS.

[9] CHEN T Q, XU B, ZHANG C Y, et al. Training deep nets with sublinear memory cost [EB/OL]. (2016 – 04 – 22) [2025 – 04 – 20]. https://arxiv.org/abs/1604.06174.

[10] PISARCHYK Y, LEE J. Efficient memory management for deep neural net inference [EB/OL]. (2020 – 02 – 16) [2025 – 04 – 20]. https://arxiv.org/abs/2001.03288.

[11] 王鑫, 李嘉楠, 韩林, 等. 面向国产异构平台的 OpenMP Offload 共享内存访存优化[J]. 计算机工程与应用, 2023, 59(10): 75 – 85.
WANG X, LI J N, HAN L, et al. Optimization of OpenMP Offload shared memory access for domestic heterogeneous platforms [J]. Computer Engineering and Applications, 2023, 59(10): 75 – 85. (in Chinese)

[12] LEE J Y, CHIRKOV N, IGNASHEVA E, et al. On-device neural net inference with mobile GPUs[EB/OL]. (2019 – 07 – 03) [2025 – 04 – 20]. https://arxiv.org/abs/1907.01989.

[13] 许鹏, 宋岩. TFLite-micro 内存管理与分配策略的优化[J]. 单片机与嵌入式系统应用, 2022, 22(10): 11 – 15.
XU P, SONG Y. Optimizations of TFLite-micro memory management and allocation policy [J]. Microcontrollers & Embedded Systems, 2022, 22(10): 11 – 15. (in Chinese)

[14] SEKIYAMA T, IMAMICHI T, IMAI H, et al. Profile-guided memory optimization for deep neural networks [EB/OL]. (2018 – 04 – 26) [2025 – 04 – 20]. https://arxiv.org/abs/1804.10001.

[15] 曹博钧, 钱入意, 徐远超. 一种面向计算图的及时内存重用算法[J]. 计算机工程与科学, 2024, 46(9): 1539 – 1546.
CAO B J, QIAN R Y, XU Y C. An urgent memory reuse algorithm for computational graphs[J]. Computer Engineering & Science, 2024, 46(9): 1539 – 1546. (in Chinese)

[16] HOCHREITER S, SCHMIDHUBER J. Long short-term memory[J]. Neural Computation, 1997, 9(8): 1735 – 1780.

[17] SUTSKEVER I, VINYALS O, LE Q V. Sequence to sequence learning with neural networks[C]//Proceedings of the 28th International Conference on Neural Information Processing Systems, 2014: 3104 – 3112.

[18] TAI K S, SOCHER R, MANNING C D. Improved semantic representations from tree-structured long short-term memory networks[EB/OL]. (2015 – 05 – 30) [2025 – 04 – 20]. https://arxiv.org/pdf/1503.00075.

[19] WANG X, YU F, DOU Z Y, et al. SkipNet: learning dynamic routing in convolutional networks[C]//Proceedings of Computer Vision—ECCV 2018, 2018: 420 – 436.

[20] NIU W, AGRAWAL G, REN B. SoD^2: statically optimizing dynamic deep neural network execution[C]//Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2024: 386 – 400.

[21] ZHANG C, MA L X, XUE J L, et al. Cocktailer: analyzing and optimizing dynamic control flow in deep learning[C]// Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation , 2023: 681 – 699.

[22] MA J M, LI X H, WANG Z H, et al. A holistic functionalization approach to optimizing imperative tensor programs in deep learning [C]//Proceedings of the 61st ACM/IEEE Design Automation Conference, 2024: 1 – 6.

[23] LAMPROU I, ZHANG Z, DE JUAN J, et al. Safe optimized static memory allocation for parallel deep learning [C]// Proceedings of the 6th Machine Learning and Systems, 2023: 305 – 324.

[24] WANG Q P, XU M W, JIN C, et al. Melon: breaking the memory wall for resource-efficient on-device machine learning[C]//Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services, 2022: 450 – 463.

[25] CHENG A C, LIN C H, JUAN D C, et al. InstaNAS: instance-aware neural architecture search[J]. Proceedings of the AAAI Conference on Artificial Intelligence, 2020, 34(4): 3577 – 3584.

[26] LI Y W, SONG L, CHEN Y K, et al. Learning dynamic routing for semantic segmentation [C]//Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020: 8550 – 8559.

[27] GUO J F, PHILIP CHEN C L, LIU Z L, et al. Dynamic neural network structure: a review for its theories and applications[J]. IEEE Transactions on Neural Networks and Learning Systems, 2025, 36(3): 4246 – 4266.

[28] LE T D, IMAI H, NEGISHI Y, et al. TFLMS: large model sup-port in tensorflow by graph rewriting[EB/OL]. (2019 – 10 – 02) [2025 – 04 – 20]. https://arxiv.org/abs/1807.02037.

[29] MENG C, SUN M, YANG J, et al. Training deeper models by GPU memory optimization on TensorFlow [C]// Proceedings of the 31st Conference on Neural Information Processing Systems, 2017.

[30] YANG H M, ZHOU J, FU Y, et al. ProTrain: efficient LLM training via memory-aware techniques [EB/OL]. (2024 – 06 – 12) [2025 – 04 – 20]. https://arxiv.org/abs/2406.08334.

[31] 宋鹤鸣. 智能语音系统加速器设计[D]. 上海: 上海交通大学, 2019.
SONG H M. Design of accelerator for voice intelligent system[D]. Shanghai: Shanghai Jiao Tong University, 2019. (in Chinese)