



面向加速器的动态横向融合自动编译加速机制

崔炜峰¹, 陈煜康¹, 赵涵¹, 陈全^{1*}, 杨静², 过敏意¹

1. 上海交通大学计算机学院, 上海 200240

2. 贵州大学公共大数据国家重点实验室, 贵阳 550025

* 通信作者. E-mail: chen-quan@cs.sjtu.edu.cn

收稿日期: 2025-06-14; 修回日期: 2025-08-29; 接受日期: 2025-09-25; 网络出版日期: 2026-03-20

国家重点研发计划 (批准号: 2024YFB4505700)、国家自然科学基金 (批准号: 62302302)、国家自然科学基金重点项目 (批准号: 62232011) 和上海市自然科学基金面上项目 (批准号: 24ZR1430500) 资助

摘要 加速器如图形处理单元 (graph processing unit, GPU) 等凭借强大的并行计算能力, 被广泛用于加速深度学习等计算密集型任务. 为充分释放 GPU 的硬件潜力, 系统层优化技术尤为关键, 其中核函数融合是一种被主流框架广泛采用的优化手段. 横向融合可并行调度多个无依赖核函数, 提高资源利用率. 然而, 现有横向融合方法多基于静态计算图, 难以适配具备动态分支结构的任务, 如混合专家模型等, 其输入在运行时动态路由至不同子网络, 导致融合核函数难以预先生成. 为此, 本文提出 FLUXER, 一种面向动态分支的自动化横向融合技术. 该方法引入动态分支表达抽象, 辅助编译系统识别动态结构, 并结合激活特性筛选候选核函数, 生成一个支持运行时输入分发的融合核函数. 编译过程中进一步引入参数结构体化、选择性循环展开等优化技术, 提升融合函数执行效率. 实验结果表明, FLUXER 在无需人工干预的前提下实现了最高 1.76 倍的性能提升, 兼顾通用性与执行效率, 适用于深度学习等多类动态计算任务.

关键词 加速器, 核函数融合, 横向融合, 深度学习编译, 动态分支

1 引言

加速器如图形处理单元 (graph processing unit, GPU) 等被大范围用于加速各种计算密集任务, 例如科学计算、深度学习任务等. 在 GPU 中, 一般配备有数十乃至数百个流式多处理器 (streaming multiprocessor, SM), 这些 SM 可以并行运行. 同时, 单个 SM 内部还配备有支持多种数值类型的计算核心 (FP64, FP32, INT32, Tensor Cores), 这些计算核心亦可以并行运行. 这样的硬件架构设计使得在 GPU 上在处理各种计算密集型任务时, 可以同时存在成千上万个独立的线程并行处理不同的数据, 从而实现远超普通中央处理器 (central processing unit, CPU) 的计算吞吐量^[1~3]. GPU 通过上述硬件架构设计提供了极其可观的算力, 然而, 这只是硬件可达的上限. 为了能够让各种应用的性能趋近硬件上限, 尤其是在深度学习系统领域, 研究者们开发了诸多系统软件层面的编译优化技术^[4~10], 提升 GPU 算力的利用率, 加速应用的执行.

引用格式: 崔炜峰, 陈煜康, 赵涵, 等. 面向加速器的动态横向融合自动编译加速机制. 中国科学: 信息科学, 2026, 56: 776–790, doi: 10.1360/SSI-2025-0283

Cui W H, Chen Y K, Zhao H, et al. Automatic compilation mechanism for dynamic horizontal fusion on accelerators. Sci Sin Inform, 2026, 56: 776–790, doi: 10.1360/SSI-2025-0283

在这些技术中,核函数¹⁾融合技术是一种流行且行之有效的重要方法^[10~16],其几乎被所有主流深度学习训练和推理框架所采纳使用^[4,8,17,18].在GPU程序中,单个计算任务(例如深度神经网络)通常会被拆分成多个核函数,每个核函数对应一个可被GPU并行的子任务,如矩阵乘法等.按照这些核函数在计算任务中的数据依赖图,由CPU通过PCIe按拓扑顺序调度至GPU上执行.核函数融合技术的思想是将相邻的两个或更多个GPU核函数合并成一个更大但功能等效的核函数,以期更好地利用GPU资源,提高任务整体性能.目前深度学习框架中采用的核函数融合包含两种:纵向融合^[4]和横向融合^[10].其中,横向融合^[10]主要针对任务数据依赖图中可并发执行的核函数,即核函数之间在拓扑上不存在依赖关系.融合后的核函数同时包含原先多个核函数的所有线程块,其中同时执行的线程块各自负责不同核函数的计算任务.横向融合的性能提升主要源于单个核函数只能利用GPU的部分并行计算资源,当多个这样的核函数可以并发执行时,横向融合后的核函数可以更加充分地利用GPU的计算资源.

然而,现有横向核函数融合技术^[10,16]主要面向具有静态计算图的任务设计,对于包含动态行为的计算任务则难以适用.特别是在深度学习场景中,越来越多的模型在结构上引入了动态多分支机制(例如混合专家模型),使得不同输入数据在运行时根据数值条件被动态路由至不同分支执行.由于各分支的数据规模在运行时可能存在显著波动,核函数必须适配多样的输入形状,无法预定义单一静态核函数供传统融合技术使用.因此,在当前动态计算任务场景中,通常只能采用两种替代方案:其一是串行地调用支持动态输入的通用核函数库,但这往往难以充分发挥GPU的并行计算能力;其二是手工实现融合核函数以支持各分支的动态并行执行,虽然性能更优,但引入了显著的开发复杂度与维护成本.若对现有横向融合方案进行简单扩展,通常需预先生成覆盖各种输入形状组合的融合核函数集合,并在运行时按需调用.该方式需为所有潜在的分支组合提前编译融合核函数,编译开销巨大,导致编译产物体积膨胀,显著增加用于保存大量核函数二进制编译文件的存储成本.

为了解决上述困境,本文提出了一种面向动态分支的自动化横向融合技术——FLUXER. FLUXER提供了一套自动化的源对源横向融合编译机制,面向动态分支场景,通过合并同构分支并支持动态输入,仅需生成一个融合核函数,便可在运行时将各分支的输入自动分发至相应的候选子核函数进行执行.在编译过程中,本文分析了多种输入分发策略在不同分支规模下的性能权衡,并进一步提出了参数结构体化与选择性循环展开等编译优化技术,以确保融合核函数在各类动态分支场景中具备高效的调用与执行性能.为支撑动态横向融合编译,FLUXER还面向深度学习任务设计了一种新的动态分支表达抽象.该抽象可辅助编译框架在计算图中自动识别动态分支结构,并支持用户定义各分支的激活特性,从而实现融合编译的自动化支持.该表达机制设计简洁高效,具备良好的可扩展性,可适配其他类型的计算任务.在候选子核函数选择阶段,FLUXER充分结合上层的激活特性与底层的硬件特征,采用多源子核函数适配与面向硬件的调优策略等优化手段,自动筛选出适配不同动态输入形状的高性能子核函数,并完成融合编译.

本文的重要贡献总结如下.

- 构建源对源自动编译机制,支持在运行时将输入自动分发至对应的子核函数,并通过引入参数结构体化、动态访存常量优化等技术提升融合核函数的执行效率.
- 提出动态分支表达抽象,支持从计算图中自动识别动态分支结构,并结合用户定义信息或运行时采样结果提取分支激活特征,从而实现自动化的横向融合.
- 所提出的FLUXER编译框架可在无需人工介入的前提下,完成多动态分支的横向融合编译,在保障编译产物体积较小的同时,自动实现融合加速效果.

FLUXER通过引入源对源自动编译机制、动态分支表达抽象和多种编译实现优化等,实现了一种自动化且高效的动态横向融合技术,显著降低了人工优化成本,同时提升了GPU的并行计算资源利用率,加速了整体计算任务的执行.实验结果表明,与动态分支的串行执行方式相比,FLUXER实现了最高1.76倍的执行速度提升;与现有手工融合优化方案相比,也能达到相当的性能表现.

1) 在GPU执行的函数一般称为核函数(kernel function).

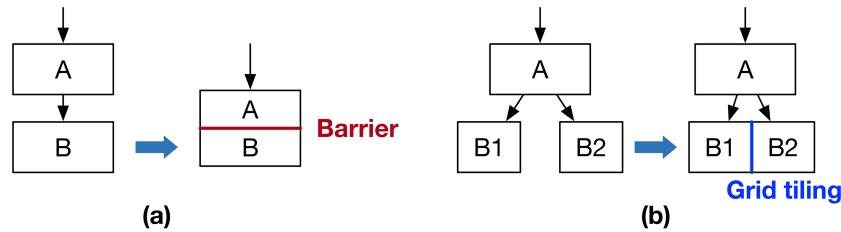


图 1 (网络版彩图) 两种主流核函数融合范式。(a) 纵向融合; (b) 横向融合。

Figure 1 (Color online) Paradigms of two mainstream kernel fusion. (a) Vertical fusion; (b) horizontal fusion.

2 背景与动机

2.1 GPU 上的编程模型

NVIDIA GPU 提供 CUDA 编程语言以及运行时来编写和运行核函数, 其他的 GPU 如 AMD GPU 和各种国产 GPU 也都提供了类似的编程和运行时环境。CUDA 是一种类 C++ 语言的语法, 运行时可使用多种 GPU 资源, 包括寄存器、共享内存以及全局内存 (显存)。由于 GPU 采用单指令多线程 (single instruction multiple threads, SIMT) 架构, 每次启动核函数时, 会并行启动多个线程块 (blocks), 每个线程块中又包含多个线程 (threads)。CUDA 中, 网格维度 (grid dimension, 指线程块的数量) 和线程块维度 (block dimension, 指每个线程块中的线程数) 通常是三维可调参数, 一般和需要处理的数据规模等相关。每个线程在执行过程中可通过 `blockIdx.x`, `blockIdx.y`, `blockIdx.z` 和 `threadIdx.x`, `threadIdx.y`, `threadIdx.z` 等内建变量获取其所属线程块和在线程块内的编号, 实现数据的并行划分与协同处理。另外需要特别说明的一些特性如下: (1) 共享内存是 GPU 上同一个线程块内所有线程可以高速读写的一块片上内存, 用于线程间数据交换和加速访问; (2) `__syncthreads()` 是 CUDA 提供的线程块级的同步屏障, 确保该线程块内所有线程都执行到该点后再继续执行后续代码。

2.2 核函数融合编译技术概述

算子融合技术是一种优化 GPU 上的复杂计算任务中核函数执行效率的重要手段, 尤其在深度学习编译器中, 已被大范围应用, 是加速模型执行的重要编译优化手段^[10~16]。如图 1 所示, 现有的核函数融合范式主要分为两种: 纵向融合和横向融合。本小节介绍他们的融合编译方法。

2.2.1 纵向融合

标准的核函数融合优化通常是纵向融合。假设有两个核函数 `A()` 和 `B()`, 它们的网格维度和线程块维度均相同。纵向融合的核函数将直接顺序组合 `A()` 和 `B()` 的源代码, 因此融合后的核函数仍拥有相同的网格和线程块尺寸, 但每个线程需要顺序执行来自 `A()` 和 `B()` 的两段指令。图 1(a) 展示了纵向融合后一个融合核函数的执行流程。从图中可见, 为了保证正确性, 融合后的核函数需要在两者之间增加同步屏障。如图所示这样的纵向融合一般只能减少核函数的 CPU 调用开销。在深度学习编译器中, 由于算子一般通过相应的中间表达式撰写, 如果此时 `A()` 的输出是 `B()` 的输入, 编译器通过中间表达式的推导, 可以实现 `A()` 核函数无需写回直接继续开始 `B()` 中的计算, 同步屏障会在代码生成时按需使用。此时, 由于减少昂贵的读写操作, 增益进一步提升。

2.2.2 横向融合

与纵向核函数融合中同样的线程执行前后两个子核函数的代码不同, 横向融合技术创建不同的线程块给需要融合的核函数。如图 1(b) 所示, 可以独立并行执行的核函数 `B1()` 和 `B2()`, 可以将他们的线程块合并到一个融合核函数内同时执行, 新融合的核函数的线程块数量为原先两个核函数之和。此时, 由于原先两个核函数串行执行且无法占满所有的 SM 资源, 新融合的函数充分利用 SM 资源, 并行执行两个核函数的线程块, 因此可以加

```

1  __device__ void B1(char* shared_buffer, int& block_idx, int& thread_idx, ...) {
2  // ① name shadowing for built-in variables
3  const dim3 gridDim(B1_GRID_X, B1_GRID_Y, B1_GRID_Z); // grid size, built-in constants
4  const dim3 blockDim(B1_BLOCK_X, B1_BLOCK_Y, B1_BLOCK_Z); // block size, built-in constants
5  const dim3 threadIdx(thread_idx % blockDim.x, thread_idx / blockDim.x % blockDim.y, thread_idx / blockDim.x /
   ↳ / blockDim.y); // restore the threadIdx of B1 to its value before flattening.
6  const dim3 blockIdx(block_idx % gridDim.x, block_idx / gridDim.x % gridDim.y, block_idx / gridDim.x /
   ↳ gridDim.y); // restore the blockIdx of B1 to its value before flattening.
7  // ② mask the threads that are not required by B1
8  if (thread_idx >= B1_BLOCK_X * B1_BLOCK_Y * B1_BLOCK_Z) return;
9  // following is the transformed original code
10 // ③ assign shared memory with pre-allocated buffer instead of allocating a new one.
11 __shared__ float original_shared_buffer = (float*)shared_buffer;
12 ...
13 // ④ replace __syncthreads() with A PTX assembly to only sync the required threads
14 asm("bar.sync 1, B1_BLOCK_X * B1_BLOCK_Y * B1_BLOCK_Z");
15 ...
16 }
17 __device__ void B2(char* shared_buffer, int& block_idx, int& thread_idx, ...) {
18 ...
19 }
20 __global__ void fused_B1_B2(...){
21 // grid and block are both flattened into a one-dimensional vector.
22 auto blockidx_x = blockIdx.x;
23 // Pre-allocate shared memory with the maximum size required by K1 and K2
24 __shared__ char shared_buffer[max_of_K1_K2_shared_buffer_size]; // compile-time constants
25 if (blockidx_x <= B1_GRID_X * B1_GRID_Y * B1_GRID_Z) {
26   B1(shared_buffer, blockidx_x, threadIdx.x, ...);
27 } else {
28   blockidx_x = blockidx_x - B1_GRID_X * B1_GRID_Y * B1_GRID_Z;
29   B2(shared_buffer, blockidx_x, threadIdx.x, ...);
30 }
31 }

```

图 2 (网络版彩图) 静态横向融合的编译方法。

Figure 2 (Color online) Compilation method for static horizontal fusion.

速程序执行。

相较于纵向融合, 横向融合的实现较为复杂. 图 2 中展示了一个具体横向融合编译例子, 该例子中设计两个分支的核函数, 该例子可推广至更多分支. 原先的核函数 B1() 和 B2() 会被处理成为设备函数²⁾. 被处理后的设备函数会新增 3 个参数, 包括共享内存指针 (shared_buffer)、线程块索引 (block_idx)、线程索引 (thread_idx). 代码行 20~31 展示了新融合的核函数以及新增参数的作用. 针对共享内存, 我们需要提前在新融合的内核函数中申请共享内存, 传递给被融合的核函数使用. 这是因为在 GPU 中共享内存的使用是针对线程块的, 由于融合后的线程块事实上存在原始两个核函数的线程块代码, 直接融合会导致线程块使用的共享内存成为两者之和. 若提前申请则仅需两者中的最大值, 这可以减少不必要的共享内存分配. 融合后的核函数中, 网格和线程块的维度都被摊平成一维, 同时采用控制流来让前半部分线程块执行 B1(), 后半部分线程块执行 B2(), 被摊平的网格和线程块变量 blockidx_x 和 threadIdx.x 会被作为参数传递给设备函数.

对于原始核函数的处理, 本文以 B1() 为例 (代码行 1~16) 解释. ① 由于在外层融合核函数内网格和线程块维度都被摊平, 首先要恢复内置网格维度和线程块维度为原始核函数中使用的, 否则原始代码执行的正确性无法保证. 这里会利用变量名称遮蔽特性, 屏蔽外层融合函数中的内置参数, 重建新的内置参数. ② 重建完毕后, 由于被融合的核函数线程块的尺寸可能存在差异, 需要利用控制流屏蔽不需要的线程执行. 接下来对原始代码进行处理, 主要包括共享内存和线程块同步操作的替换. ③ 对于原始代码中的共享内存申请, 使用外层核函数申请的共享内存对其赋值, 如果存在多处共享内存申请, 可以根据偏移量对外层申请的共享内存进行分片, 在此过程中, 编译器会考虑内存对齐问题. ④ 由于部分线程被屏蔽, 直接进行线程块粒度 (即调用 __syncthreads()) 同步会导致死锁. 因此, 对于线程块同步操作, 需要使用并行线程执行中间表示 (parallel thread execution, PTX) 汇编代码 bar.sync 替代 __syncthreads(), 仅对所需要的线程进行选择同步. 具体来说, 如代码行 14 所示, bar.sync 汇编指令有两个参数: 第一个参数是屏障索引号, 同一个线程块内可以有最多 16 个屏障可用, 第二个参数指示了线程块内参与同步屏障的线程数目, 当线程块内执行该条语句的线程数量达到该值以后, 所有被屏障阻塞的线程会立即向后执行, 不会出现线程间死锁的情况. 当 bar.sync 的第二个参数被省略时, 效果等同于 __syncthreads().

2) 在 CUDA 语言中, __global__ 用来标识一个核函数, 可以直接从 CPU 端发射至 GPU 上执行, __device__ 用来标识一个设备函数, 它需要由核函数调用才能执行.

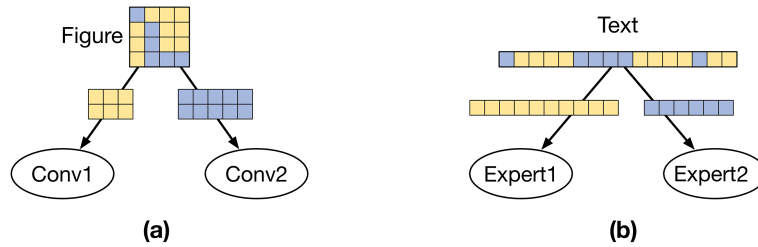


图3 (网络版彩图) 常见的多分支深度学习网络。(a) 多分支卷积网络; (b) 混合专家语言模型。

Figure 3 (Color online) Common deep learning networks with multiple dynamic branches. (a) Multi-branch convolution network; (b) mixture of expert language model.

2.3 现有融合编译方法的问题

计算任务通常需要动态适配不同的输入大小,例如在深度学习网络中需要应对不同批次大小任务的计算。为了处理这种动态性,现有的深度学习编译工作^[19]采用基于采样动态核函数编译的方法,即离线选取若干批次大小的形状,针对性地自动调优若干核函数进行编译,以供运行时根据输入形状选择合适的核函数调用。此方法可以和纵向融合完美兼容,因为编译器可以首先利用中间表达式推理出纵向融合版本的中间表达,再根据采样到的形状进行核函数的自动调优。

横向融合编译同样需要应对动态输入的问题,且其动态性更为复杂。如图3所示,典型的多分支深度学习模型结构通常包含动态输入。在这些结构中,完整的图像或文本被划分为更小的单元(如子图或词元),并在运行时被分发至不同的分支进行处理。此类设计有助于发挥各分支模型的优势,从而在深度学习任务中获得更优的处理效果。

当待融合的两个分支各自呈现独立的动态性时,基于静态输入假设的横向融合方法将难以适用。如图2所示,现有横向融合方法依赖静态已知的网格与线程块参数进行预处理与融合,这种静态依赖使其难以支持运行时具有动态行为的多分支结构。为支持这类动态分支的融合,可效仿纵向融合引入基于采样的动态横向核函数编译方法^[20]。该方法需首先为每个分支分别调优多个候选核函数,并在此基础上进行组合与融合,生成新的横向融合核函数。若每个分支采样得到 N 个候选实现,且共有 M 个分支,则组合空间将达到 N^M ,形成指数级增长。如此将导致生成体积庞大的二进制文件,带来显著的编译产物膨胀问题,并消耗大量额外内存用于存储,影响系统部署效率与运行开销。

现有工作通常只能采取两种折中方式:一是串行执行各个分支,每个分支使用现有算子库提供的算子进行计算,结束各分支计算后再聚合。由于每个分支仅处理输入的一部分,其并行度较低,导致GPU的并行计算资源利用效率不高;二是手工编写可适配多个动态分支并行计算的核函数。这类方法不仅高度依赖人工优化,开发成本较高,而且通常仅适用于具有高度同构性的分支。例如在混合专家模型中,各分支均执行相同尺寸的矩阵乘法,因此研究者们开发了专用于混合专家模型的分组矩阵乘法核函数。然而对于存在尺寸差异的矩阵计算,或涉及其他类型算子(如卷积)时,这种方法便难以适用。

综上,面向存在动态分支的GPU程序,现有的解决方案都存在其各自的限制性,亟需一种面向加速器的动态横向融合自动编译优化机制。该机制应该遵守以下几个原则。

- 可以适配各种核函数,无需人工干预,自动化编译需要进行横向融合的分支。
- 需要编译的融合函数少,二进制文件小,无需占用指数级增加的存储空间。
- 可以适配分支的动态性,运行时高效分发至合适核函数,充分利用并行资源加速计算。

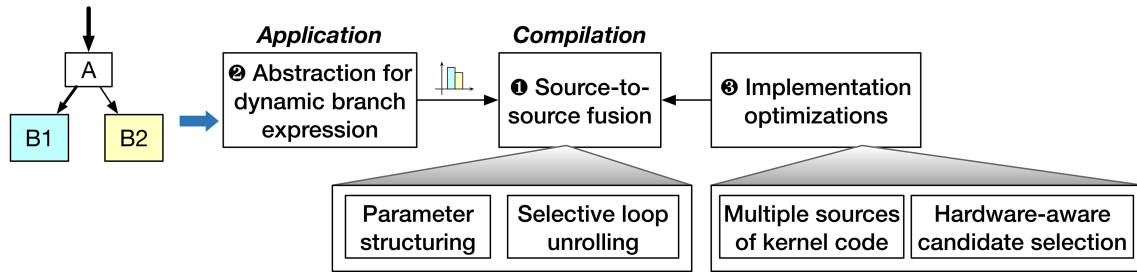


图4 (网络版彩图) FLUXER 的整体架构.

Figure 4 (Color online) Overview of FLUXER.

3 动态横向融合编译

3.1 Fluxer 架构

为应对 2.3 小节中提出的挑战, 本文提出了 FLUXER, 一个面向动态分支的横向融合编译框架. 图 4 展示了该框架的整体架构, 其主要由三部分组成: 源对源编译模块、动态分支表达抽象以及编译实现优化. **源对源融合模块**是 FLUXER 的核心组件, 构建了一套从候选核函数自动融合生成支持动态分支的融合核函数的完整编译流程. 在此过程中, 我们深入研究了动态融合函数的运行特性, 并提出了如参数结构体化、动态访存常量化等关键优化技术, 以提升融合核函数在动态场景下的执行效率. **动态分支表达抽象**是 FLUXER 面向深度学习任务设计的一套统一表达机制, 为源对源编译模块提供关键的辅助信息, 如分支激活特性等. 尽管该抽象现阶段针对深度学习场景, 但其设计简洁通用, 具备良好的扩展性, 为扩展适用于高性能计算等其他类型的计算任务打下良好基础. **编译实现优化**聚焦于候选核函数融合前的关键优化策略, 主要包括: 结合底层硬件特性进行适配性核函数选择, 以及支持来自多种源码形态的候选实现, 包括 CUDA C++ 和 CUDA PTX 汇编代码等, 从而提升框架的通用性与可移植性.

接下来, 本文将从以上三部分组成展开阐述 FLUXER 的具体设计细节.

3.2 动态横向融合的源对源编译方法

本节介绍 FLUXER 的源对源动态融合编译核心组件. 具体来说, 首先介绍本文设计的一种支持动态横向融合编译的初步方案, 并挖掘其低效性; 根据发现的低效原因通过两种优化手段针对性解决, 最终形成 FLUXER 的动态横向编译方法.

3.2.1 动态横向编译的初步方案

首先, 本文分析动态横向融合所期望达到的编译结果. 图 5 展示了该融合过程的目标编译流程. 在初始阶段, 仍然为各个分支采样常见的输入形状, 并据此选择合适的核函数, 以支持不同分支在动态输入下的高效执行. 对于所有分支选出的候选子核函数, 如图 5(a) 所示, 为了减少编译生成的二进制文件体积, 本文尝试将所有子核函数融合为一个统一的核函数. 该融合核函数在运行时能够根据输入形状, 将来自不同分支的数据分发至对应的子核函数中执行, 从而兼顾动态性与效率.

为了进一步压缩编译输出大小, 需要将同构分支进行合并. 在诸如混合专家等深度学习模型结构中, 常涉及形如 $A \times B^3$ 的矩阵乘法操作, 其中不同分支之间通常仅存在 A 矩阵在 M 维度的动态变化, 其余维度保持一致. 因此, 这些分支的核函数具有同构性, 在融合后具备高度的可复用性. 如图 5(b) 所示, 最终融合所得的新核函数仅包含一种计算逻辑, 对应不同批处理大小的子核函数可复用执行, 融合核函数可根据输入动态分发各分支至相应子核函数. 此时, 可以看到由于不同分支的输入大小和被融合的子核函数的顺序不一致, 运行时可能需要把分支的输入按照输入大小进行排序, 这样才能按需分发到正确的子核函数中进行计算. 特别地, 当多个分支具有

3) 矩阵乘法中, A 与 B 矩阵的维度通常为 (M, K) 和 (K, N) , 其中 M 常表示批处理大小, 也是动态输入变化的主要维度.

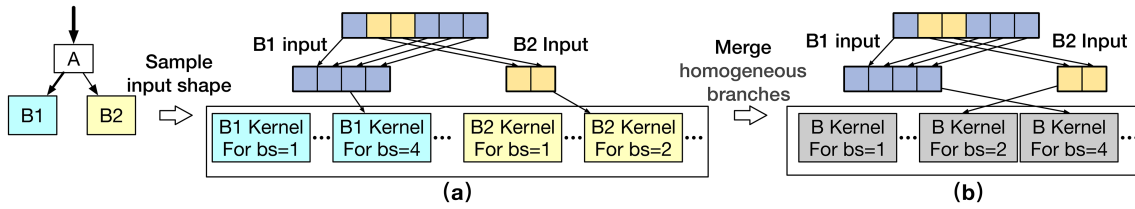


图 5 (网络版彩图) 横向融合的目标编译流程. (a) 初始融合后的新核函数; (b) 合并同构分支的融合函数.

Figure 5 (Color online) Expected compilation workflow of horizontal fusion. (a) New kernel with initial fusion; (b) fusion kernel with homogeneous branches merged.

```

1  __device__ void B1(char* shared_buffer, int& block_idx, int& thread_idx, float* ptr_1, float* ptr_2) {
2  ...
3  }
4  ...
5  // ①add extra parameters and merge device pointers into pointer array for launching
6  __global__ void dynamic_fused_B1_B2(int B1_active_grids, int B2_active_grids, float* ptr_array_1[], float*
   → ptr_array_2[]){
7
8  int blockidx_x, arg_idx;
9  __shared__ char shared_buffer[max_of_K1_K2_shared_buffer_size];
10 // grid size for first fused kernel B1
11 auto B1_GRID_SIZE = B1_GRID_X * B1_GRID_Y * B1_GRID_Z;
12 // grid size for first fused kernel B2
13 auto B2_GRID_SIZE = B2_GRID_X * B2_GRID_Y * B2_GRID_Z;
14 if (blockIdx.x < B1_GRID_SIZE * B1_active_grids) {
15     blockidx_x = blockIdx.x;
16     // ②calculate the input handled by this block.
17     arg_idx = blockIdx_x / B1_GRID_SIZE;
18     // ③calculate the actual flattened blockIdx.x used within device kernel
19     blockidx_x = blockIdx_x % B1_GRID_SIZE;
20     B1(shared_buffer, blockidx_x, threadIdx.x, ptr_array_1[arg_idx], ptr_array_2[arg_idx]);
21 } else if (blockIdx.x <= B1_GRID_SIZE * B1_active_grids + B2_GRID_SIZE * B2_active_grids){
22     blockidx_x = blockIdx_x - B1_GRID_SIZE * B1_active_grids;
23     // calculate the input handled by this block.
24     arg_idx = blockidx_x / B2_GRID_SIZE + B1_active_grids;
25     // calculate the actual flattened blockIdx.x used within device kernel
26     blockidx_x = blockidx_x % B2_GRID_SIZE;
27     B2(shared_buffer, blockidx_x, threadIdx.x, ptr_array_1[arg_idx], ptr_array_2[arg_idx]);
28 }
29 ...
30 }
31
32 // invoking the fused kernel for dynamic branches
33 // ④assigning the host-side ptr array with the required order
34 host_ptr_array_1[0] = branch_2_input_1;
35 host_ptr_array_1[1] = branch_1_input_1;
36 host_ptr_array_2[0] = branch_2_input_2;
37 host_ptr_array_2[1] = branch_1_input_2;
38 // ⑤copying them to device-side ptr array
39 cudaMemcpy(device_ptr_array_1, host_ptr_array_1, ...);
40 cudaMemcpy(device_ptr_array_2, host_ptr_array_2, ...);
41 dynamic_fused_B1_B2<<...>>(1, 1, device_ptr_array_1, device_ptr_array_2);

```

图 6 (网络版彩图) 横向融合编译方法的初步方案.

Figure 6 (Color online) Naive compilation method for horizontal fusion.

相同的输入时, 对应的子核函数需要进行相应的重复调用, 以满足多个分支的计算需求.

基于以上分析, 本文提出如图 6 所示的动态横向融合初步方案. 在这个初步方案中, 被融合的子核函数的前处理和图 2 所示静态横向融合方法中一致, 因此本示例中略过这部分的介绍, 本例子中主要介绍两个同构分支的横向融合, 所涉及的待融合子核函数的输入分别是批处理大小为 1 和 2. 异构分支更为简单, 同构与异构并存时, 可以按本例扩展. 为了使融合后的核函数支持动态分支, 融合核函数的模版相比静态版本发生了重大变化. 这主要体现在融合核函数的调用参数 (代码行 6) 和子核函数的动态分发机制 (代码行 14~29).

① 在调用参数方面, 首先为每一个被融合的子核函数 $B_x()$ 新增了 $B_x_active_grids$ 参数, 该参数用来运行时决定每个被融合的子核函数需要被激活执行的次数. 其次, 为了满足同构分支的子核函数合并, 每一个动态激活的子核函数需要运行时获取对应的输入的指针. 因此, 需要合并所有分支的相同位置的参数进指针数组, 以供运行时分发至合适的子核函数进行计算.

在子核函数的动态分发机制方面, 每个子核函数对应的分支需要根据从融合函数传入的激活次数参数决定是否是否需要被激活计算. 主要涉及两个变量的计算, ② 首先是当前子核函数对应的指针参数在指针数组中的索

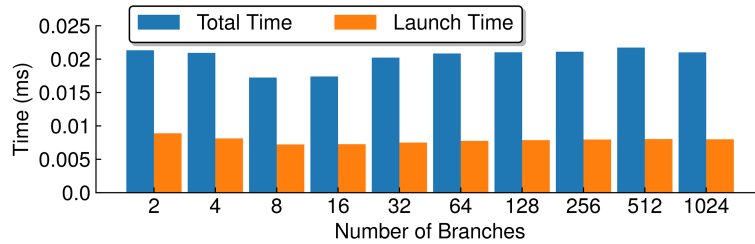


图 7 (网络版彩图) 不同分支数量下完整执行时间与仅融合内核函数的执行时间对比。

Figure 7 (Color online) Latency comparison of total execution and only the execution of the naively fused kernel with different branches.

引, ③ 其次是将要执行的子核函数真实线程块的索引, 当获取到这两个变量后, 当前子核函数所需的所有调用参数得到确认, 可以调用子核函数进行对应的计算。

融合模版的变化也对融合核函数的调用有重大影响 (代码行 32~41), 所有分支的指针参数需要以正确的顺序放到对应的指针数组中。由于在 GPU 上只能访问 GPU 侧的指针数组, ④ 需要在 CPU 侧首先按照预计的分支与子核函数分发关系, 对分支的指针排序, 并将其存入一个 CPU 侧指针数组, ⑤ 再调用 CPU 与 GPU 侧的内存拷贝函数, 拷贝至 GPU 侧的指针数组以供融合核函数使用。

3.2.2 融合函数的参数结构体化与选择性循环展开

为了验证初步方案的可行性, 首先在不同分支数量下进行了实验。为排除融合子核函数内部计算的影响, 将所有被融合的子核函数简化为仅执行赋值后立即退出的形式。图 7 展示了初步方案下, 融合核函数在不同分支数量下的完整执行时间 (对应于图 6 中的 Lines 32~41), 以及核函数调用本身的时间开销 (对应于图 6 中的 Line 41)。从图中可以观察到, 初步方案中由于需调用内存拷贝函数对指针数组进行复制, 平均引入了约 0.015 ms 的调用开销, 该开销已超过核函数本身的启动开销。当融合核函数在运行中被频繁调用时, 这一额外开销将显著拉高端到端时延, 进而削弱横向融合带来的性能收益。

为优化初步方案中的调用开销, 本文提出了第一个编译优化策略: 参数结构体化。在初始方案中, 之所以需要通过显式内存拷贝传递各个未融合分支的指针数组, 是因为 CUDA 核函数中对数组的动态索引访问统一被视为显存访问。受限于 CUDA 的内存语义, 核函数无法直接接受来自 CPU 端的数组作为参数并执行动态索引操作。如图 8① 所示若将 CPU 端的指针数组封装于结构体中并以结构体形式传入核函数, 由于结构体采用值传递, 其内部的指针数组可被内联展开, 并自动驻留于 GPU 侧的全局内存、局部内存或寄存器等存储介质中, 从而绕过了动态索引所带来的访问限制。通过参数结构体化, 融合后的核函数在调用时不再依赖显式的内存拷贝操作, 从而有效消除了初始方案中由拷贝操作引入的额外调用开销。

然而, 我们进一步的研究发现, 指针数组参数的结构体化并非完全无代价。实验结果显示, 当融合核函数涉及特定数量分支时, 结构体化会带来显著的性能开销。通过分析相关核函数生成的 PTX 汇编代码, 我们发现该开销源于编译器对结构体内指针数组的动态索引访问所触发的启发式优化。具体而言, 当指针数组长度小于等于 16 时 (64 bit 指针, 整体大小超过 128 Byte), 编译器倾向于使用局部内存为每个线程缓冲结构体。但 GPU 的局部内存本质上是由全局内存模拟, 导致其动态访问延迟较高。而当数组长度超过 16 时, 结构体体积过大, 编译器会避免使用局部内存缓冲, 选择直接访问结构体, 此时访问性能得以恢复。

基于上述分析, 本文提出了第二个编译优化策略: 选择性循环展开。如图 8② 所示当分支数量小于 16 时, 系统会将对结构体中指针数组的访问循环进行显式展开, 从而规避动态索引操作, 避免触发局部内存缓冲优化。FLUXER 将子核函数所需指针的获取过程提取为编译期已知的循环体并强制展开, 避免了编译器使用局部内存。当分支数量超过 16 时, FLUXER 不会展开该访问循环, 这是因为循环展开在此场景下将显著增加无效代码路径的长度, 带来额外的执行开销。此时, 保持原始的直接访问路径仍是更优的编译策略。综上, FLUXER 会在编译期检查 PTX 汇编中间结果, 自适应地选择最优的编译策略, 保证动态横向融合后的核函数在运行时可以保持高效

```

1  // ① Structuring parameters as compound data types
2  struct Params {
3      float* a[NUM_BRANCH];
4  }
5
6  __global__ void dynamic_fused_B1_B2(int B1_active_grids, int B2_active_grids, Params P1, Params P2) {
7      ...
8      if (blockIdx.x < B1_GRID_SIZE * B1_active_grids) {
9          blockIdx_x = blockIdx.x;
10         arg_idx = blockIdx_x / (B1_GRID_X * B1_GRID_Y * B1_GRID_Z);
11         blockIdx_x = blockIdx_x % (B1_GRID_X * B1_GRID_Y * B1_GRID_Z);
12     } else if (blockIdx.x <= B1_GRID_SIZE * B1_active_grids + B2_GRID_SIZE * B2_active_grids){
13         blockIdx_x = blockIdx_x - B1_GRID_SIZE * B1_active_grids;
14         arg_idx = blockIdx_x / B2_GRID_SIZE + B1_active_grids;
15         blockIdx_x = blockIdx_x % B2_GRID_SIZE;
16     }
17     // ② unroll the access of pointer array
18     float* ptr_1, ptr_2;
19     #pragma unroll
20     for (int i = 0; i < NUM_BRANCH; i++) {
21         if (i == arg_idx) {
22             ptr_1 = P1.a[arg_idx];
23             ptr_2 = P2.a[arg_idx];
24         }
25     }
26     if (blockIdx.x < B1_GRID_SIZE * B1_active_grids) {
27         B1(shared_buffer, blockIdx_x, threadIdx.x, ptr_1, ptr_2);
28     } else if (blockIdx.x <= B1_GRID_SIZE * B1_active_grids + B2_GRID_SIZE * B2_active_grids){
29         B2(shared_buffer, blockIdx_x, threadIdx.x, ptr_1, ptr_2);
30     }
31     ...
32 }

```

图 8 (网络版彩图) 动态横向融合的参数结构体化和选择性循环展开。

Figure 8 (Color online) Structuring parameters as compound data types and selective loop unroll for dynamic horizontal fusion.

```

1  import torch.nn as nn
2  import fluxer
3  class BranchLayer(nn.Module):
4      def __init__():
5          self.branch_a = nn.Linear(...)
6          self.branch_b = nn.Linear(...)
7          self.scatter = fluxer.Scatter(branch_num=2, prefer_capacities=[[1, 2, 4], [1, 4, 8]])
8          self.gather = fluxer.Gather(branch_num=2)
9      # ① before fusion
10     def forward(input: Tensor, branch_indices) -> Tuple[Tensor]:
11         a_x, b_x, branch_indices = self.scatter(input, branch_indices)
12         a_y = self.branch_a(a_x)
13         b_y = self.branch_b(b_x)
14         output = self.gather(a_y, b_y, branch_indices)
15     # ② after compilation of dynamic horizontal fusion
16     def fused_forward(input: Tensor, branch_indices) -> Tuple[Tensor]:
17         x, branch_indices = self.scatter(input, branch_indices)
18         y = self.fused_branches(x)
19         output = self.gather(y, branch_indices)

```

图 9 (网络版彩图) 支撑动态横向融合的动态分支表达例子。

Figure 9 (Color online) Example of defining dynamic branches with the support for dynamic horizontal fusion.

分支的分发和执行。

3.3 支撑动态横向融合的动态分支表达

如 3.2 小节所示, 动态横向融合方法依赖于识别计算任务中的动态分支信息, 从而实现源对源编译。为支撑上述融合编译模块, 本文进一步提出了一种动态分支表达抽象, 用于自动化提供所需的分支信息。总体而言, 该抽象需满足两个核心要求: 其一, 能够辅助从原始计算任务中提取出待横向融合的动态分支; 其二, 能够提供分支激活特性的获取机制。

围绕上述目标, FLUXER 设计了一种面向深度学习任务的动态分支表达抽象。该抽象主要包括两个特殊算子: Scatter 与 Gather。其中, Scatter 用于在运行时根据计算结果按需将成批数据分发至各个分支, 它支持用户指定分支激活特性; 而 Gather 则作为 Scatter 的逆操作, 在多个分支完成计算后负责将子张量恢复至原始输入中的相对位置, 便于后续计算处理。

图 9 展示了一个示例, 我们使用 FLUXER 提出的抽象定义具有两个分支的动态模型结构。该示例中, 两个

分支均为线性层; **Scatter** 在定义时明确指定了分支数量, 并附带每个分支倾向被激活时所处理的批量大小; 而 **Gather** 仅需指定所接收的分支数量. 值得一提的是, **FLUXER** 默认以张量的首维 (即批处理维度) 进行动态数据分发. ① 在执行阶段, **Scatter** 与 **Gather** 均接收一个特殊张量 `branch_indices`, 用于指示原始输入中每个子张量应被分发至的分支. **Scatter** 据此对输入数据进行重排, 而 **Gather** 则基于该张量在分支执行结束后恢复原有顺序. ② 当 **FLUXER** 从深度学习模型中识别出通过动态分支表达定义的动态分支时, **FLUXER** 将把所有分支融合进单个核函数中, 此时 **Scatter** 和 **Gather** 内部进行数据重排的方法也会相应改变, 多个分支的张量被融合成单个.

上述面向深度学习任务设计的动态分支表达抽象, 仅通过在现有深度学习框架中引入两个特殊算子即可实现. 对于其他类型的计算任务, 该抽象亦具备良好的可扩展性, 可通过类似方式进行适配与实现. 此外, 若其他计算任务借助深度学习框架进行建模与表达, 也可直接复用本抽象, 从而自动获得对动态横向融合编译的支持.

3.4 编译实现优化

通过编译层的源对源融合模块与应用层的动态表达抽象, **FLUXER** 实现了一种自动化的动态横向融合编译技术. 该技术依托 PyTorch^[18] 的 `torch.fx` 机制, 将应用层的抽象表达与编译层的融合流程有效桥接, 实现了自动化支持. 需要指出的是, **FLUXER** 并不直接负责子核函数的性能优化; 其高性能实现依赖于现有的算子自动调优编译框架^[4, 21]. 此外, 在子核函数的选择过程中, **FLUXER** 也设计了相应的优化策略.

面向硬件的子核函数调优. 除依据激活特征选择适配的子核函数外, **FLUXER** 还需结合加速器的硬件特性进行优化选择. 主要考虑两个方面: 首先, 针对底层硬件的计算单元特性 (如 GPU 的 Tensor Core), 这些单元通常对输入形状有特定要求 (如批处理维度需为 4 的倍数). 因此, **FLUXER** 仅融合满足硬件约束的子核函数, 并在运行时通过动态表达抽象自动填充不满足条件的输入. 其次, 为避免资源浪费, **FLUXER** 限制融合子核函数的规模. 例如, 若性能测试表明批处理大小为 128 的融合子核函数已能充分利用 GPU 资源, 则对于批处理大小为 256 的输入, **FLUXER** 会将其划分为两个子任务并分别分配至两个 128 规模的子核函数执行.

子核函数多源适配. 一般情况下, 被融合的子核函数以 CUDA C++ 代码形式存在, 前述编译流程也以此为主要示例. 然而, 一些单算子调优框架 (如文献 [21]) 并不生成 CUDA C++ 代码, 而是直接生成 PTX 汇编代码. 为此, **FLUXER** 实现了一套将 PTX 汇编代码嵌入至 CUDA C++ 中的机制, 从而支持同时融合 CUDA C++ 与 PTX 汇编形式的子核函数, 拓展了其高性能实现的来源范围.

具体而言, 对于 PTX 形式核函数代码, **FLUXER** 首先使用 `asm()` 在 CUDA C++ 内实现内嵌. 内嵌完成以后, 我们将核函数包裹成为一个设备函数用于横向融合. 内嵌同样需要处理共享内存的申请与线程块内部的同步屏障, 这个过程与图 2 中类似, 只是处理对象变为 PTX 汇编代码. 共享内存的申请抽取依赖对 `.shared` 变量的识别, 同步屏障的处理则直接修改 `bar.sync` 指令即可. 与之前核函数直接转化成设备函数不同的是对函数参数的处理. 核函数的参数一般是主机端的变量, 在 PTX 代码中通常使用 `ld.param.u64` 指令来从主机端载入至核函数内部进行后续计算. 在函数融合中, 原始在主机端的变量由融合后的核函数载入, 设备函数在调用时使用的已经是 GPU 端变量. 因此, 在转化成设备函数时, 我们需要将 PTX 代码中的 `ld.param.u64` 参数载入指令替换成为 `mov.u64` 指令, 这样即可将已被载入的参数变量直接传递给内嵌的 PTX 代码使用.

4 实验与结果

本节通过对多个典型深度学习算子及多个完整深度学习模型应用动态横向融合, 全面评估了 **FLUXER** 的性能表现. 对比对象包括默认的串行执行方式、手工实现的动态横向融合方案、通过填充适配动态的静态横向融合方案以及离线编译多种融合组合适配动态的动态横向融合方案. 总体结果表明, **FLUXER** 相较于串行执行最高可实现 1.76× 的性能加速, 并在无需人工干预的情况下, 达到了与手工融合相当的优化效果.

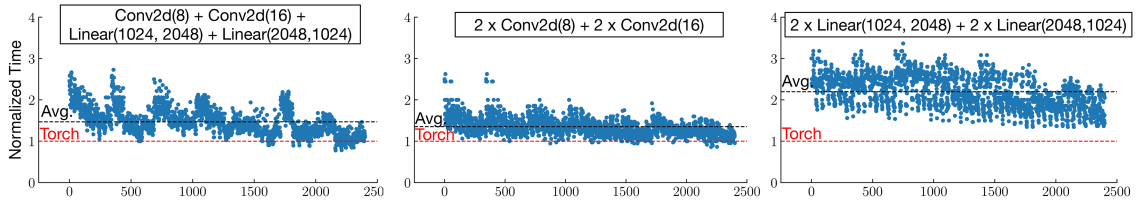


图 10 (网络版彩图) 相同分支数量不同分支核函数下, FLUXER 与 PyTorch 串行执行时间对比。

Figure 10 (Color online) Latency comparison of FLUXER and PyTorch's serial execution with the different kernels and the same number of branches.

4.1 实验环境

软硬件环境. 所有性能评估均在一台配置为 AMD EPYC 9654 处理器与 4 块 NVIDIA H100 (80GB) GPU 的服务器上进行. 该服务器运行环境包括 CUDA 12.8, PyTorch 2.7^[18], CUTLASS 3.5.0^[22] 和 TVM 0.19^[4]. 在实验中, 我们采用 TVM 生成用于横向融合编译的原始核函数代码.

比较对象. 本节的所有实验均主要将 FLUXER 与原生 PyTorch 以及 CUTLASS 进行对比. PyTorch 作为当前最先进的深度学习框架之一, 凭借 Python 的强大表达能力, 能够灵活支持动态神经网络结构. 本节中评估的所有模型均基于其官方 PyTorch 实现, 因此所有评估均以 PyTorch 为主要对比基线. 另一方面, CUTLASS 提供了针对动态分支场景手工优化的横向融合矩阵乘法核函数 (Grouped Gemm), 因此在本节中被用于与 FLUXER 进行针对手工优化融合方案的性能对比. 在端到端性能测试中, 除了原生 PyTorch 串行以外, 我们进一步和静态横向融合以及现有动态横向融合方法 Brainstorm 进行对比. 为了使得静态横向融合能够适配分支动态性, 静态融合下每个分支被填充至最大可能的大小. Brainstorm 使用的候选融合核函数与 FLUXER 一致, 它通过为不同分支组合的不同输入形状生成大量核函数, 并在离线阶段进行批量编译, 以支持运行时动态选择合适的核函数组合. 它以大量的离线开销, 获得了近似 FLUXER 的融合效果.

4.2 动态核函数融合性能

本小节主要在算子级别验证 FLUXER 的性能表现, 包括与 PyTorch 原生的串行执行相比, 以及与人工优化的横向融合算子对比.

串行执行对比. 本实验对比了 PyTorch 版本的串行执行与 FLUXER 所实现的动态横向融合在性能上的差异. 首先从 4.3 小节中的模型中抽取了具有代表性的分支算子, 包括两个卷积算子和两个矩阵乘法算子. 其中, 两组卷积算子的通道数分别为 8 和 16, 卷积核大小均为 $[3 \times 3]$, 输入特征图尺寸为 $[32 \times 32]$, 该类算子常见于视觉模型. 两组矩阵乘法算子的权重矩阵维度分别为 $[1024 \times 2048]$ 与 $[2048 \times 1024]$, 典型用于大语言模型中的专家网络, 其对应的输入词元维度分别为 $[1024]$ 与 $[2048]$. 我们将这四类典型算子组合构建为一个包含 4 个分支的动态分支结构, 每个分支的动态激活范围为 0~64. 实验过程中对所有分支进行随机激活, 比较 PyTorch 串行执行与 FLUXER 融合版本的性能表现. 图 10 展示了所有实验点的性能数据. 为便于对比, 我们将 FLUXER 的性能以串行版本为基准进行了归一化处理, 并按照各实验点的总分支负载对数据点排序, 使最右侧的点具有最高的总体负载. 图中的红色虚线代表 PyTorch 版本的串行执行, 黑色虚线代表了所有配置下 FLUXER 的平均加速比.

从图中可见, 在 3 种不同的分支结构下, FLUXER 分别获得了 1.47, 1.35 和 2.20 倍的平均性能提升. 由于卷积与矩阵乘法算子在资源占用上的差异, 卷积算子对寄存器的需求更高, 导致其横向融合的加速效果略逊于矩阵乘法算子. 此外, 当总体负载较高时, FLUXER 的性能提升趋于下降. 这一趋势是可以预期的: 在负载较大时, 各分支在串行执行中已能充分利用 GPU 资源, 融合执行与串行执行的差距因此缩小. 与此同时, 由于 FLUXER 在融合过程中存在线程块间的资源竞争, 在部分负载配置下甚至可能略低于串行性能. 由于不同的子核函数之间使用同一块共享内存, 因此融合并不会增加对共享内存的需求. 这里的资源竞争主要来自寄存器资源, 融合后的核函数的寄存器资源使用量一般会大于单个子核函数的使用量. 当单个子核函数已经使用较多寄存器资源时, 融合后会存在出现寄存器溢出、单个 SM 上活跃的 Warp 数量下降等问题, 从而导致性能下降. 因此, 横向融合

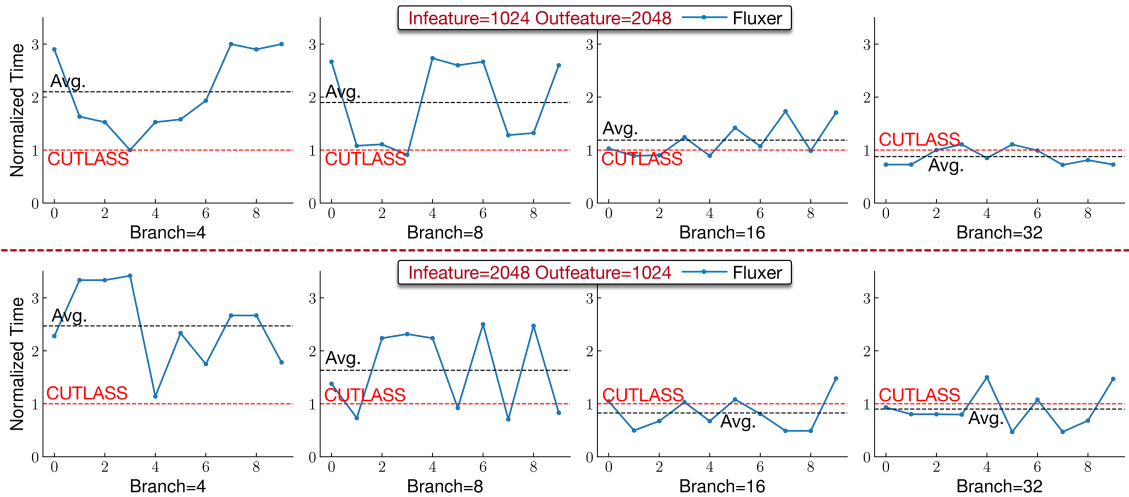


图 11 (网络版彩图) 不同分支数量下 FLUXER 与人工优化 CUTLASS 的执行时间对比。

Figure 11 (Color online) Latency comparison of FLUXER and manual optimization, CUTLASS, with the different numbers of branches.

策略应结合具体运行时负载进行动态选择. 本文聚焦于自动化横向融合能力的实现, 融合策略的动态选择问题超出本文讨论范围, 将留待后续研究进一步探讨.

人工优化对比. 接下来, 将 FLUXER 与人工横向融合算子, 即 CUTLASS 中的 Grouped GEMM 算子的性能进行对比. 由于 CUTLASS 的人工优化方案仅适用于矩阵乘法, 本实验聚焦于上一小节中两种矩阵乘法算子的融合性能, 并扩展了更多分支数量的对比情形. 图 11 展示了在不同分支数量下, CUTLASS 与 FLUXER 在两类矩阵算子上的性能对比, 所有性能数据相对 CUTLASS 进行了归一化处理. 图中每个点所对应的分支大小均为随机动态采样, 红色虚线代表基线 CUTLASS 版本的执行, 黑色虚线代表了所有配置下 FLUXER 的平均加速比.

从图中可以看出, 在分支数量较少时, FLUXER 的性能显著优于 CUTLASS, 最高可实现平均 $2.47\times$ 的加速. 当分支数量增多时, FLUXER 与 CUTLASS 的性能逐渐接近, 在某些情况下可达到 CUTLASS 性能的 83%. 通过分析 CUTLASS 的底层实现, 我们发现其在调用核函数前需显式拷贝所有分支的设备指针, 这一过程与 FLUXER 的初始实现类似, 会引入额外调用开销, 进而影响整体性能. 然而, 随着分支数量增加, 该调用开销趋于稳定, 且 CUTLASS 中手工优化的算子在计算效率上优于 FLUXER 所依赖的 TVM 自动生成算子, 因此其性能优势逐步体现. 总体而言, FLUXER 所提出的动态横向编译融合机制, 不仅具备高度的自动化能力, 在动态核函数调度方面展现出更低的调用开销, 同时在算子执行性能上也能达到与手工优化算子相当的水平.

4.3 端到端性能测试

本小节端到端验证了 FLUXER 的性能表现, 选取了 3 个典型的动态深度学习模型, 覆盖视觉与自然语言处理任务: Switch^[23] 为基于编码器-解码器架构的混合专家大语言模型, Llama-MoE^[24] 为基于仅解码器架构的混合专家大语言模型, Llava-MoE^[25] 则是一个混合专家多模态大模型. 这些模型均包含动态多分支的混合专家层, 在主流实现中常以串行方式执行, 具备应用 FLUXER 自动横向融合优化的实际需求. 由于本工作聚焦单机场景下的横向核函数融合优化, 3 种模型均可在单个 GPU 上直接运行, 对于更大规模、需分布式部署的模型, 其部分子模块亦可从 FLUXER 中受益.

图 12 中的实验结果展示了 FLUXER 相较于其他 3 种方法的端到端性能对比, 实验数据相比 PyTorch 串行做了归一化处理, 实际的性能数据在图中也做了标注, 单位为秒. 语言模型采用 MNLI^[26] 数据集进行评估, 多模态模型则基于 image-text^[25] 数据集进行推理, 每个模型在数据集中采样 100 条输入, 测量平均推理速度. 其中, 多模态模型分别在批处理大小为 1 与 8 的场景下进行测试. 实验表明, FLUXER 相比串行执行最高带来 $1.76\times$ 、平均 $1.51\times$ 的加速收益, 这主要是因为, FLUXER 能够在运行时自动融合多个专家分支, 进而更加高效地利用 GPU 的并行资源. FLUXER 相比静态横向融合执行最高带来 $1.75\times$ 、平均 $1.47\times$ 的加速收益, 主要归因于其在

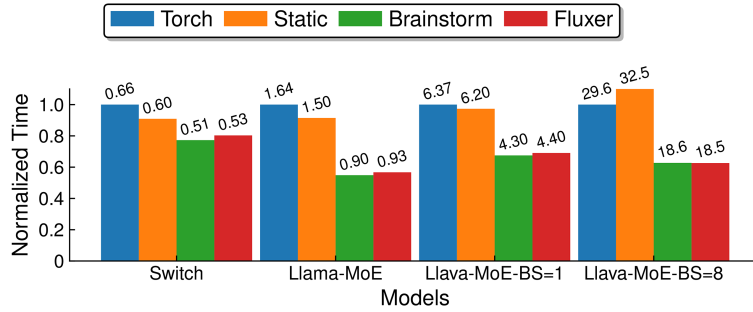


图 12 (网络版彩图) 真实模型下, FLUXER 和其他基线的执行时间对比.

Figure 12 (Color online) Latency comparison of FLUXER and other baselines with the real-world models.

表 1 编译产物大小比较 (MB).

Table 1 Size (MB) comparison of compilation artifact.

	2 branches	3 branches	4 branches
FLUXER	0.46	1.0	1.6
Brainstorm	4.12	81.7	1139.5

运行时去除了无效填充导致的冗余计算. 静态横向融合虽然有无效填充, 但是受益于其可以更好地利用并行资源, 其相比串行之行仍然可以获得一定加速. FLUXER 与 Brainstorm 的性能相近, 两者使用相同的核函数执行计算, 不同之处主要在于核函数的选择, FLUXER 在 GPU 上进行, Brainstorm 则在主机端进行. 同时, FLUXER 相比 Brainstorm 仅需要很少的编译产物, 具体在 4.4 小节中进行验证.

在 Llava-MoE 模型中, 当批处理大小为 8 时, 相较批处理大小为 1 时, FLUXER 相比串行和静态横向融合都取得了更高的加速比. 我们进一步分析其分支激活情况发现, Llava-MoE 包含 16 个专家分支, 批处理增大时激活的专家数量显著提升, 导致串行执行开销增大, 从而放大了 FLUXER 的动态融合优势. 同时, 静态横向融合中无效填充也增多, 甚至比串行执行表现更差.

4.4 动态融合扩展性测试

本节主要对 FLUXER 的各种编译优化机制进行验证, 验证内容包括编译产物大小以及融合核函数的调用开销.

编译产物测试. 首先, 对比了不同方案在编译产物大小方面的表现, 主要比较对象为 FLUXER 与 Brainstorm. Brainstorm 是一种同样支持动态融合编译的技术, 其可实现与 FLUXER 相近的运行时性能. 然而, Brainstorm 通过为不同分支组合的不同输入形状生成大量核函数, 并在离线阶段进行批量编译, 以支持运行时动态选择合适的核函数组合. 这一策略虽能提供灵活性, 但也带来了大量冗余的编译产物, 导致系统二进制文件体积显著膨胀.

表 1 展示了在使用 4.2 小节中的四类算子进行 2 分支、3 分支和 4 分支融合 (各分支结构均不同) 时, FLUXER 与 Brainstorm 在编译产物体积上的对比结果. 从表中可以看出, FLUXER 生成的编译产物显著小于 Brainstorm: 在 4 分支场景下, Brainstorm 生成的二进制文件超过 1 GB, 而 FLUXER 仅为 1.6 MB.

动态融合编译优化分析. 其次, 比较了 FLUXER 中采用的编译优化技术. 图 13 展示了在不同分支数量下融合后函数的调用时间. 为便于直观比较, 该实验中每个线程仅读取指针所指向的数据并执行加 1 写回操作. 由图可见, 初步实现 (Naive) 在所有分支数下均表现出较为稳定的调用开销, 这是由于其始终依赖于对融合函数参数进行显式内存拷贝. 采用参数结构体优化后 (Strcutted), 在分支数小于等于 16 时调用时间逐步上升至约 0.4 ms, 超过 16 时则出现明显下降. 此处, 在分支数量不大于 16 时, 因为编译器触发了内置启发式优化策略, 利用局部内存缓冲参数加载, 进而带来了相应开销. 当分支数量超过 16 时, 编译器的内置优化策略取消了对局部内存的使用, 这一开销消失. 相比之下, 在结构封装优化的基础上使能参数循环展开优化 (Unrolled), 降低了分

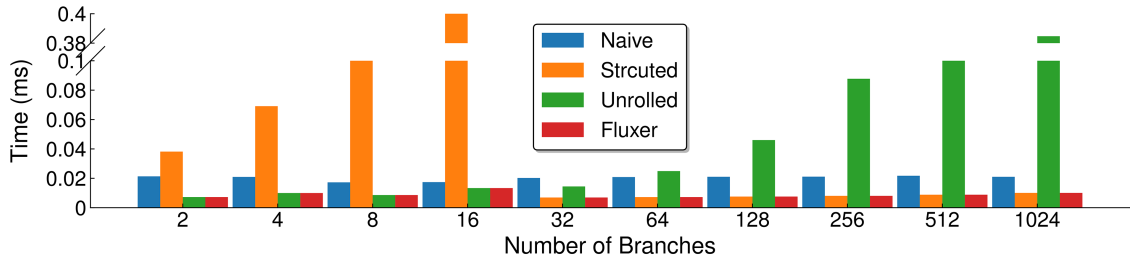


图 13 (网络版彩图) 不同分支数量下多种动态横向融合编译方法与 FLUXER 的执行时间对比。

Figure 13 (Color online) Latency comparison of different dynamic fusion methods and FLUXER with different numbers of branches.

支数量小于 16 的开销。但是, 在分支数超过 16 后调用时间逐步升高至 0.4 ms, 并在分支数继续增加时呈增长趋势。这主要是因为, 在分支数量不超过 16 时, 参数展开可以避免编译器使用本地内存, 降低仅使用结构体封装优化的开销。当分支数量超过 16 时, 继续展开不再能避免本地内存使用, 强制展开反而增加了无效代码的执行, 这部分无效执行与分支数量相关, 所以开销逐步增长。综合来看, FLUXER 融合了参数结构体与参数展开两种方式的优势, 首先使能参数结构体化, 其次自动在不同分支规模下选择性使能参数循环展开, 在各类分支规模下均能保持最优的函数调用性能。

5 结论

GPU 凭借其强大的并行计算能力, 已成为深度学习等计算密集型任务的核心加速硬件。为提升 GPU 算力利用率, 核函数融合技术在主流系统中广泛应用, 尤其是横向融合能将无执行依赖的多个核函数并行执行, 从而显著提高硬件资源利用率。然而, 传统横向融合方法主要面向静态计算图, 难以适配混合专家等具备动态分支结构的任务, 造成性能浪费或开发负担。为此, 本文提出 FLUXER, 一种面向动态分支的自动化横向融合技术。其通过引入动态分支表达抽象, 辅助识别计算图中的动态结构, 并结合激活特性筛选融合候选核函数, 仅生成一个融合核函数即可在运行时完成输入自动分发。结合参数结构体化、选择性循环展开等优化策略, FLUXER 在仅生成有限编译产物的同时, 实现高效融合加速。实验表明, 该方法在动态任务场景下实现了显著的性能提升, 有效降低人工优化成本。

参考文献

- 1 NVIDIA Corp. NVIDIA Volta GPU architecture. 2025. <https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/>
- 2 NVIDIA Corp. NVIDIA Ampere architecture. 2025. <https://www.nvidia.com/en-us/data-center/ampere-architecture/>
- 3 NVIDIA Corp. Hopper architecture overview. 2025. <https://www.nvidia.com/en-sg/data-center/technologies/hopper-architecture/>
- 4 Chen T, Moreau T, Jiang Z, et al. TVM: an automated end-to-end optimizing compiler for deep learning. In: Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, 2018
- 5 Jia Z, Padon O, Thomas J, et al. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019
- 6 Rotem N, Fix J, Abdurassool S, et al. Glow: graph lowering compiler techniques for neural networks. 2018. ArXiv:1805.00907
- 7 Boemer F, Lao Y, Cammarota R, et al. nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data. 2018. ArXiv:1810.10121
- 8 Abadi M, Agarwal A, Barham P, et al. TensorFlow: large-scale machine learning on heterogeneous distributed systems. 2016. ArXiv:1603.04467
- 9 Wei R, Schwartz L, Adve V. DLVM: a modern compiler infrastructure for deep learning systems. 2017. ArXiv:1711.03016
- 10 Ma L, Xie Z, Yang Z, et al. Rammer: enabling holistic deep learning compiler optimizations with rTasks. In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, 2020
- 11 Appleyard J, Kocisky T, Blunsom P. Optimizing performance of recurrent neural networks on GPUs. 2016. ArXiv:1604.01946

- 12 Diamos G, Sengupta S, Catanzaro B, et al. Persistent RNNs: stashing recurrent weights on-chip. In: Proceedings of the 33rd International Conference on Machine Learning, 2016
- 13 Wang G, Lin Y, Yi W. Kernel fusion: an effective method for better power efficiency on multithreaded GPU. In: Proceedings of the 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing, 2010
- 14 Wahib M, Maruyama N. Scalable kernel fusion for memory-bound GPU applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014
- 15 Filipović J, Madzin M, Fousek J, et al. Optimizing CUDA code by kernel fusion—application on BLAS. 2013. ArXiv:1305.1183
- 16 Li A, Zheng B, Pekhimenko G, et al. Automatic horizontal fusion for GPU kernels. In: Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization, 2022
- 17 Chen T, Li M, Li Y, et al. MXNet: a flexible and efficient machine learning library for heterogeneous distributed systems. 2015. ArXiv:1512.01274
- 18 Ansel J, Yang E, He H, et al. PyTorch 2: faster machine learning through dynamic Python bytecode transformation and graph compilation. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2024
- 19 Zheng B, Jiang Z, Yu C H, et al. DietCode: automatic optimization for dynamic tensor programs. In: Proceedings of Machine Learning and Systems, 2022
- 20 Cui W, Han Z, Ouyang L, et al. Optimizing dynamic neural networks with Brainstorm. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation, 2023
- 21 Tillet P, Aiken A, Kozyrakis C. Triton: an intermediate language and compiler for tiled neural network computations. In: Proceedings of the 5th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2021
- 22 NVIDIA Corporation. CUTLASS: CUDA templates for linear algebra subroutines. 2025. <https://github.com/NVIDIA/cutlass>
- 23 Fedus W, Zoph B, Shazeer N. Switch transformers: scaling to trillion parameter models with simple and efficient sparsity. *J Mach Learn Res*, 2023, 23: 5232–5270
- 24 Zhu T, Qu X Y, Dong D Z, et al. LLaMA-MoE: building mixture-of-experts from LLaMA with continual pre-training. 2024. ArXiv:2406.16554
- 25 Lin B, Tang Z, Ye Y, et al. MoE-LLaVA: mixture of experts for large vision-language models. 2024. ArXiv:2401.15947
- 26 Wang A, Singh A, Michael J. GLUE: a multi-task benchmark and analysis platform for natural language understanding. In: Proceedings of International Conference on Learning Representations, 2019

Automatic compilation mechanism for dynamic horizontal fusion on accelerators

Weihao CUI¹, Yukang CHEN¹, Han ZHAO¹, Quan CHEN^{1*}, Jing YANG² & Minyi GUO¹

1. School of Computer Science, Shanghai Jiao Tong University, Shanghai 200240, China

2. State Key Laboratory of Public Big Data, Guizhou University, Guiyang 550025, China

* Corresponding author. E-mail: chen-quan@cs.sjtu.edu.cn

Abstract Accelerators such as graphics processing units (GPUs) have been widely adopted to accelerate compute-intensive tasks like deep learning, owing to their powerful parallel processing capabilities. To fully exploit the hardware potential of GPUs, system-level optimization techniques are crucial, among which kernel fusion has become a widely used approach in mainstream frameworks. Horizontal fusion enables parallel scheduling of multiple independent kernels, thereby improving resource utilization. However, existing horizontal fusion methods are typically designed for static computation graphs and struggle to support tasks with dynamic branching structures—such as mixture-of-experts (MoE) models—where inputs are dynamically routed to different sub-networks at runtime, making it difficult to predefine fused kernels. To address this challenge, we propose FLUXER, an automated horizontal fusion technique tailored for dynamic branches. FLUXER introduces a dynamic branch expression abstraction to help the compiler identify dynamic structures and selects candidate kernels based on activation characteristics to generate a single fused kernel that supports runtime input dispatching. The compilation process further incorporates optimizations such as parameter structuring and selective loop unrolling to improve execution efficiency. Experimental results show that FLUXER achieves up to 1.76× performance improvement without manual intervention, balancing generality and efficiency, and is applicable to a wide range of dynamic computation tasks such as deep learning.

Keywords accelerator, kernel fusion, horizontal fusion, compilation of deep learning, dynamic branches