基于 TVM 的算子融合调度策略研究

李 明[1] 邢 亮[1] 李振彦[1] 李中武[1] 王 蝶[1] 张文豪[1]
LI Ming XING Liang LI Zhenyan LI Zhongwu WANG Die ZHANG Wenhao

摘 要 针对自然语言处理领域中广泛存在的 Embedding Lookup 与 Sum 算子结构，文章通过研究其在 TVM 编译器框架下的高性能融合调度策略，深入分析该组合算子的计算特征与数据依赖关系，结合 CPU 与 GPU 不同的硬件体系结构特点，分别设计了面向缓存层次和向量化执行的 CPU 调度策略，以及基于线程层次映射和显存带宽优化的 GPU 调度策略，实现了算子深度融合。实验结果表明，与 TVM 默认调度及 PyTorch 2.5 的原生实现相比，文章设计的融合调度内核在 CPU 和 GPU 平台上平均取得了良好的加速效果，有效减少了中间数据读写开销，显著提升了嵌入层操作的执行效率，为 TVM 支持领域特定算子优化提供了有效的调度模板与实验依据，对提升 Transformer 类模型的推理性能具有实用意义。

关键词 算子融合；调度优化；自然语言处理；内核实现；推荐系统

doi：10.3969/j.issn.1672-9528.2025.12.014

0 引言

深度学习技术的快速演进推动了计算机视觉、自然语言处理等领域的广泛应用，而深度学习模型的执行效率很大程度上依赖于底层算子的优化。如在自然语言处理领域，嵌入查找 Embedding Lookup 与求和 Sum 组成的复合算子结构，是 Transformer、BERT、LSTM 等主流模型的核心组件。其中，Embedding Lookup 负责将离散的词索引映射为连续的高维向量，Sum 算子则对序列维度的向量进行聚合，生成批次级的特征表示。这两个操作在 NLP 模型的每一轮前向与反向传播中高频执行，其性能瓶颈会直接导致模型整体运行效率下降，尤其在大批次、长序列的工业级部署场景中，该问题更为突出。

随着深度学习编译器技术的发展，算子融合已成为解决算子性能瓶颈的关键技术之一。通过将多个连续执行的算子合并为单一计算内核，算子融合能够有效减少中间数据的内存分配、读写与传输开销，同时提升计算资源的利用率。TVM 作为当前主流的开源深度学习编译器，具备自动算子融合能力。然而，TVM 的算子融合机制大多适配通用深度学习算子，缺乏对领域特定复合算子的深度支持。在此背景下，针对特定领域模型的复合算子特性，设计高效的融合策略能够充分挖掘模型中的算子融合机会，结合融合后的调度优化，提升模型在 CPU、GPU、NPU 等硬件平台上的推理效率具有重要意义。

当前，针对领域特定复合算子的融合优化研究仍处于起步阶段，尤其在非常规融合方案设计与硬件算子特性匹配方面缺乏系统性方法。本文以 NLP 领域核心的 Embedding Lookup+Sum 复合算子为研究对象，开展基于 TVM 的算子融合调度研究，通过定制化的内存访问策略与计算调度，解决 TVM 通用融合策略的不足，为领域特定复合算子的优化提供技术参考。

1 相关工作

算子融合作为深度学习编译器的关键图优化技术之一，通过将多个存在数据依赖关系的算子合并为单一核函数，减少中间数据的存储与访问开销，提升模型执行效率与硬件资源利用率，已成为推动深度学习模型高效部署的关键手段。现有研究从多个角度对该问题展开了广泛探索。例如，DeepCuts[1] 采用基于分析成本模型的贪婪策略以指导图级融合决策；Shi 等 [2] 开发的 Welder 将算子切分为细粒度数据块，并通过公共数据块的重用实现内存融合；TensorFlow XLA[3] 支持对归约和元素级操作进行通用融合；AStitch[4] 则基于数据局部性的内存融合策略，有效实现访存密集型算子间的融合。这些工作采用了多样化的优化算法与策略，共同推动了模型性能与部署效率的提升。

算子融合调度作为融合实现的关键环节，负责在计算图划分阶段对同一融合组内的算子进行循环结构、存储布局与并行策略的统一调度，其效果高度依赖深度学习编译器的调度能力。Rammer[5] 借助固定模式融合进一步降低内核启动开销；Cortex[6] 面向动态递归模型提出了一组基于内核融合的优化方法；DNNFusion[7] 通过将算子及其组合分类到不同映

1. 中国航空工业集团公司西安航空计算技术研究所
陕西西安 710000

射类别，生成更为激进的融合计划。为充分挖掘算子之间的数据复用机会，满足多算子链式依赖的融合需求，Zheng 等[8]提出了 Chimera，通过优化融合算子底层循环的执行顺序以最大化数据重用。

TVM 作为主流编译器，其融合调度优化也引起了广泛关注。Chen 等[9]提出的 AutoTVM 框架，采用预定义调度模板并结合基于机器学习的代价模型评估性能，以生成融合调度配置；Zheng 等[10]开发的 Ansor 则通过对调度空间的分层采样，结合进化算法与学习型代价模型实现更优调度选择；王磊[11]针对 TVM 在计算密集型算子融合粒度与场景覆盖方面的不足展开了优化研究；郑光飞[12]则针对 TVM 在访存密集型算子融合调度能力上的局限，提出了相应的调度规则。

然而，TVM 的现有调度机制仍存在不足，比如在面对自然语言处理、推荐系统等特定领域的复杂算子时，其自动调度模块难以生成适配性强、性能优的融合方案，无法满足这类算子的高效融合需求。针对这一问题，本文通过深入分析 Embedding 与 Sum 目标算子的计算特征及数据依赖关系，结合不同硬件体系结构的存储层次与并行特性，为算子分配合适的调度原语，设计了面向不同硬件的融合策略，实现了 Embedding 与 Sum 操作的深度融合；同时，通过系统性实验验证了融合方案的有效性，旨在为 TVM 框架支持特定领域复杂算子融合提供技术补充，进一步完善深度学习编译器的算子融合能力。

2 相关理论及基础

TVM（tensor virtual machine）是一个端到端的开源深度学习编译器栈，其核心设计目标是高效地将来自多种前端框架（如 PyTorch、TensorFlow）的深度学习模型编译并优化到多种硬件后端，其编译流程如图 1 所示。

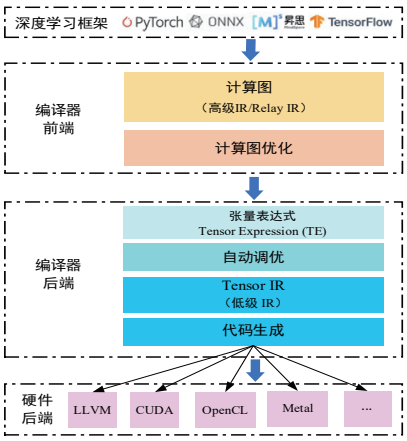

图 1 TVM 编译流程

前端部分负责模型导入与解析，支持接收不同框架的模型格式并转换为 TVM 的中间表示 Relay IR；Relay IR 负责进行图层的算子优化，之后被向下转换为张量级的中间表示 TE，TE 经过自动调优框架找到最佳实现策略进而降级为低级中间表示 Tensor IR；最终生成可在目标硬件上高效执行的机器码，从而实现在各个硬件平台上部署。TVM 不仅高度灵活，且有效实现了跨框架、跨硬件的深度学习模型高效部署。

TVM 借鉴了 Halide 语言中计算与调度分离的核心思想。计算仅描述要算什么，而调度则定义如何计算。这种解耦使得开发者可以在不改变计算正确性的前提下，通过灵活尝试不同的调度策略来最大化硬件性能。TVM 提供了两种调度方式，一种为手动调度，开发者可基于对计算模式、数据依赖及硬件架构的理解，通过手动组合调度原语构建优化方案，从而实现极致化、定制化的性能优化；一种是自动调度，以基于模板搜索的 AutoTVM 和基于自动生成搜索空间的 Ansor 为代表，借助自动化算法对调度空间进行探索。然而，针对本文研究的 Embedding+Sum 算子组合，自动调度在处理这类非常规或领域特定的复杂算子组合时，难以在有限时间内找到最优解或者无法识别特殊的融合机会。因此，本文采用手动调度方法探究其最优融合方案，并验证融合带来的性能增益，旨在为这类规则的自动化实现提供技术参考。

3 融合调度策略设计

针对访存密集型 Injective 类与 Reduction 类算子的特征分析表明，当计算任务中存在 Injective+Reduction 结构时，可将多个算子融合为单一内核。尽管 TVM 的自动调度模块已支持该类算子的基础融合规则，但对于诸如 Embedding+-Sum 等复杂结构的融合仍存在一定局限。因此，本节基于张量表达式调度原语，手动设计调度规则，实现对 Embedding+Sum 算子结构的有效融合。

3.1 计算特征分析

Embedding+Sum 结构在推荐系统、自然语言处理等模型中广泛应用，其计算过程可分为两个步骤：首先通过 Embedding Lookup（或 Gather 操作）从嵌入表中检索得到一组向量，随后通过 Sum 操作沿序列维度进行归约求和。该结构具有如下显著特征：Embedding Lookup 操作通常伴随不规则的内存访问模式，当序列长度或嵌入维度较大时，中间结果将占用大量显存，并引入显著的内存读写开销。由于该操作的数据获取效率高度依赖内存带宽，因此属于典型的访存密集型操作。随后的 Sum 操作作为 Reduction 类计算，需对 Gather 的输出张量进行全局遍历，进一步加剧内存访问压力，成为系统性能瓶颈。

基于上述分析，对 Embedding + Sum 结构实施算子融合，可有效避免中间结果的全局内存读写操作，将 Gather 与 Sum 操作合并到单一内核中执行后，数据可直接在寄存器或高速

缓存中流转，大幅减轻内存带宽压力。由此可见，算子融合对于优化内存访问效率、提升 Embedding + Sum 结构整体执行性能具有重要作用。

3.2 融合算子定义

在深入研究张量表达式调度原语的基础上，本文通过张量表达式显式定义了 Embedding Lookup+Sum 结构的计算过程，并借助调度原语实现两个算子的融合。

首先，使用张量表达式分别定义 Injective 类型的 Embedding Lookup 算子和 Reduction 类型的 Sum 算子，并建立 Sum 对 Embedding Lookup 输出的依赖关系。随后，调用 te.create_schedule() 函数生成初始调度方案。以下为 Embedding Lookup + Sum 融合算子的张量表达式定义示例：

```
defembedding_lookup_sum_baseline(vocab_size, embedding_dim, batch_size, seq_length):
    data= te.placeholder((vocab_size, embedding_dim), name= "embedding_table", dtype= "float32")
    indices = te.placeholder((batch_size,seq_length), name = "indices", dtype= "int32")
    #1.(Gather)
      gathered=te.compute((batch_size,seq_length,embedding_dim), lambda i, j, k: data[indices[i, j], k], name= "gather")
    #2.(Sum)
      te.reduce_axisr=te.reduce_axis((0,seq_length), name="r")
      sum_reduce=te.compute((batch_size,embedding_dim),
      lambda i, k: te.sum(gathered[i,r,k],axis=r),name= "sum_reduce")
    return [data, indices], sum_reduce
```

3.3 CPU 的融合调度策略

基于此前对 Embedding+Sum 算子结构特征的分析，本节针对 CPU 硬件架构特性，设计算子融合调度策略，将 Embedding 与 Sum 算子整合为统一计算内核，以提升计算效率与内存利用率。CPU 作为通用硬件，其架构注重内存访问效率和指令级并行，其性能高度依赖于缓存利用率和单指令多数据流（SIMD）向量化执行能力。因此，本文以提高数据局部性、减少内存访问开销、充分利用多核并行与向量化计算为目标，对融合后的算子进行调度优化。

首先，使用 TVM 框架的自动调度接口（te.create_schedule）实现基准调度方案。TVM 自动调度会基于 CPU 架构信息，完成基础的多核任务分配与 SIMD 指令适配，但未针对 Embedding 算子的嵌入维度数据访存、Sum 算子的归约维度累加进行优化，存在中间数据读写主存、归约计算分支开销较大等问题。针对基准调度的不足，结合 CPU 硬件特性与算子结构特征，本文针对融合内核设计使用了 parallel（多核并行）、vectorize（向量化）、unroll（循环展开）三类调度原语，进行融合调度的优化，具体实现如下：

```
#CPU 基准调度
    i, k=sch[out].op.axis
#CPU 优化调度
    i, k=sch[out].op.axis
    r =sch[out].op.reduce_axis[0]# 获取归约轴
# 并行化外层循环
    sch[out].parallel(i)
# 向量化内层循环
    k_o, k_i=sch[out].split(k, factor=8)
    sch[out].vectorize(k_i)
# 循环展开减少分支开销
    r_o, r_i=sch[out].split(r, factor=4)
    sch[out].unroll(r_i)
```

在该调度策略中，针对批量维度的并行特性，采用 parallel 原语充分发挥 CPU 的多核性能；采用 vectorize 原语将嵌入维度上的标量操作转换为向量操作，极大提高了计算密度和数据加载效率。利用归约维度的计算依赖特征设计 unroll 调度，解决分支开销大与缓存重用率低的问题。算子融合本身消除了中间数组，避免了中间数据在主存与缓存间的频繁交互，从根本上降低了缓存污染，进一步适配了 CPU 的层次化内存结构。

3.4 GPU 的融合调度策略

GPU 作为高性能并行计算硬件，其架构设计与 CPU 存在显著差异，其专为大规模数据并行计算而设计，核心特征包含大量轻量级计算核心、高带宽显存以及层次化的线程组织模型。因此结合 Embedding 与 Sum 算子的结构特性，本文在 GPU 后端针对 Embedding 与 Sum 算子的融合内核设计了显式的线程映射与执行配置策略，以充分释放其并行计算潜力，具体的优化调度实现如下：

```
#GPU 基准调度策略
# 对 Gather 操作调度
    i, j, k = sch[gathered].op.axis
    k_o, k_i = sch[gathered].split(k, factor=1024)
    sch[gathered].bind(i, te.thread_axis("blockIdx.x"))
    sch[gathered].bind(j, te.thread_axis("blockIdx.y"))
    sch[gathered].bind(k_o, te.thread_axis("blockIdx.z"))
    sch[gathered].bind(k_i, te.thread_axis("threadIdx.x"))
# 对 Sum 操作调度
    i, k = sch[out].op.axis
    r, = sch[out].op.reduce_axis
```

```
    k_o, k_i = sch[out].split(k, factor=1024)
    sch[out].bind(i, te.thread_axis("blockIdx.x"))
    sch[out].bind(k_o, te.thread_axis("blockIdx.z"))
    sch[out].bind(k_i, te.thread_axis("threadIdx.x"))
#GPU 优化调度策略
    i,k=sch[out].op.axis
    r=sch[out].op.reduce_axis[0]# 获取归约轴
# 绑定线程
    sch[out].bind(i,te.thread_axis("blockIdx.x"))
    sch[out].bind(k,te.thread_axis("threadIdx.x"))
# 对 r 轴进行切分以更好地利用并行性
    r_o,r_i=sch[out].split(r,factor=4)
    sch[out].unroll(r_i)
```

以上调度策略通过线程绑定、循环拆分与循环展开三类操作实现优化，旨在将计算任务高效地映射到 GPU 的线程层次结构上。具体地，通过 bind 原语显式绑定计算轴到 GPU 线程网格，确保了大规模并行执行的硬件支持；通过 split 和 unroll 原语对规约轴的拆分与展开，有效优化了规约过程的执行效率，降低了原子操作或同步带来的开销。同时，算子融合显著减少了对全局显存的访问次数，避免了中间结果的读写操作，从而降低了内存延迟并提高显存带宽的利用效率。

4 实验结果分析

本小节对前述调度策略进行实验验证。实验环境采用的主要工具与版本包括：TVM 0.18.0、LLVM 15.0、CUDA 12.6 及 Python 3.12。GPU 平台选用 NVIDIA GeForce GTX 2080 Ti，其配备 4 352 个 CUDA 核心，11 GB GDDR6 显存，显存带宽为 616 GB/s。CPU 平台为 x86 架构，单片具 8 个物理核心，基准运行频率可达 3.0 GHz。

在许多现代深度学习模型中，Embedding 查找后接 Sum 求和的模式较为常见。例如，BERT 等基于 Transformer 的预训练语言模型在其嵌入层中广泛使用该结构，将 token、位置等嵌入向量相加；DeepFM、DCN 等深度推荐模型也常利用 Embedding + Sum 结构融合多种特征嵌入。为全面评估融合优化效果，本文分别从典型模型中选取多处不同规模的 Embedding + Sum 结构作为测试用例，涵盖不同词汇表大小、嵌入维度、批次大小与序列长度，具体配置如表 1 所示。

表 1 测试集规模

配置编号	词汇表大小	嵌入维度	批次大小	序列长度
1	1 000	128	32	20
2	5 000	128	32	64
3	30 522	1 024	32	256
4	50 000	1 024	128	512

为充分评估本文所设计融合策略的有效性，实验设置以 Embedding 算子和 Sum 算子分别独立计算并显式写入和读取中间张量作为对照组；实验组为融合执行（Fused），即通过调度优化将两算子合并为单一内核，避免中间结果的产生与存取。

表 2 与表 3 分别给出了在 CPU 与 GPU 平台上，4 组典型 Embedding-Lookup+Sum 结构的实测性能。为便于对比，表中同时列出了 PyTorch 2.5 的默认实现（未融合）作为基准。所有数据均取 100 次热启动后的平均耗时，单位 ms。

表 2 CPU 平台性能测试结果

编号	PyTorch 基准	TVM 基准	TVM 融合调度	纵向加速比	PyTorch 的加速比
1	0.021 4	0.018 7	0.001 6	11.51	13.17
2	0.042 2	0.154 9	0.004 4	35.11	9.57
3	7.746 1	17.034 6	2.288 1	7.44	3.39
4	61.686 5	138.583 1	20.265 6	6.84	3.04

表 3 GPU 平台性能测试结果

编号	PyTorch 基准	TVM 基准	TVM 融合调度	纵向加速比	PyTorch 的加速比
1	0.020 0	0.010 7	0.003 0	3.56	6.65
2	0.019 7	0.024 8	0.004 8	5.18	4.12
3	0.211 8	0.198 1	0.061 5	3.22	3.45
4	1.587 8	1.513 3	0.449 7	3.36	3.53

由表 2 结果可知，TVM 融合调度相对于未融合时的加速比呈先上升后下降的趋势，这主要是因为小规模时内存占用低，数据量完全可控在 CPU 的 Cache 容量范围内，同时融合消除的中间张量存取开销收益占主导；当数据规模变大时，计算量占比提升，融合减少的内存带宽绝对值不变，相对收益被计算时间稀释。融合后的 TVM 内核在 CPU 上平均比 PyTorch 快 7.29。这是由于 PyTorch 的实现需先实例化中间张量，再对 Embedding 算子与 Sum 算子进行串行执行，该过程不仅会产生额外的内存动态分配与数据写回开销，还因中间张量需经历两次 Cache 遍历，极易触发 Cache 容量失效问题，大幅增加数据访问延迟；同时其依赖的 OpenMP 并行策略仅针对外层 reduce 操作进行并行化设计，未能充分适配算子计算特性，导致向量指令的利用宽度不足，难以发挥 CPU 的向量计算能力。而 TVM 通过 parallel、vectorize、split 等一系列调度原语的优化，不仅能在单个 CPU 核心内充分利用 AVX2 向量指令带宽，实现数据的高效并行处理，还能稳定利用多核扩展效率，最大程度发挥 CPU 的多核计算潜能，从而有效规避了 PyTorch 实现中的性能瓶颈，最终实现对 TVM 未融合调度及 PyTorch 默认实现的显著性能超越。

从表 3 的结果可以看出，在 GPU 平台上，Embedding+Sum 算子融合的效果相对稳定，纵向加速维持在 3.2×~5.18×，低于 CPU 平台的加速效果。这是因为 GPU 的显存带宽远高于 CPU，使得省略一次全局写回的边际收益减小；另一方面，融合算子实现导致 GPU 内核的寄存器用量增加，每个流处理器（SM）可同时调度的线程块数量减少，SM 理论占用率略有下降。但 TVM 的融合调度相对于 PyTorch 默认实现仍保持稳定优势。这是因为 PyTorch 在 CUDA 上的性能损耗主要集中于内核启动与数据传输环节。PyTorch 在 CUDA 上需分别为 Embedding 与 Sum 算子执行一次内核启动，同时还需在设备内存中完成中间张量的传输操作。而 TVM 融合后仅需一次内核启动操作，且将 threadIDx.x 直接映射到 Embedding_dim 维度，实现全局内存访问流量减半，进一步减小数据传输开销，因此在所有测试配置中均持续优于 PyTorch 原生实现。

实验结果表明，本文提出的 Embedding Lookup 与 Sum 算子融合方案在不同硬件平台和不同规模配置下均能带来显著的性能提升。与业界主流框架 PyTorch 相比，TVM 融合方案也展现出明显的性能优势，为深度学习模型的高效推理提供了有力支持。

5 结语

本文围绕 TVM 深度学习编译器中 Embedding Lookup 与 Sum 这一典型算子组合的融合调度问题展开研究，针对该结构访存密集、计算规约的特点，分别设计了面向 CPU 和 GPU 硬件后端的定制化融合调度策略。在 CPU 平台上，通过并行化、向量化和循环展开等调度原语，显著提升了数据局部性和指令级并行度；在 GPU 平台上，通过线程块绑定和规约轴优化，有效利用了显存带宽和大规模并行计算资源。实验结果表明，所提出的融合内核策略在不同规模数据集及不同硬件平台上均取得了显著性能提升，验证了融合策略在减少中间数据读写、最大化硬件计算效率方面的有效性。本研究不仅为 TVM 框架中特定领域算子的优化提供了可复用的调度模板和实验依据，也为进一步探索编译器导向的算子融合自动化优化提供了重要基础。未来工作可集中于调度策略的自动化生成、更复杂嵌入模式的扩展支持以及多硬件平台的自适应优化等方面。

参考文献：

[1]JUNG W, DAO T T, LEE J.DeepCuts:a deep learning optimization framework for versatile GPU workloads[C]//PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. NewYork: ACM, 2021:190-205.

[2]SHI Y N, YANG Z, XUE J L,et a1.Welder:scheduling deep learning memory access via tile-graph[C]//17th USENIX Symposium on Operating Systems Design and Implementation(OSDI23).USA:USENIX Association,2023:701-718.

[3]TensorFlow[EB/OL].(2015-11-09)[2025-11-22].https://www.tensorflow.org/xla.

[4]ZHENG Z, YANG X D, ZHAO P Z,et a1.AStitch:enabling anew multi-dimensional optimization space for memory-intensive ML training and inference on modem SIMT architectures[C]//Proceedings of the 27th ACM International Conference on Architectural Suppon for Programming Languagesand Operating Systems.NewYork:ACM,2022:359-373.

[5]MA L X, XIE Z Q, YANG Z,et al. Rammer: enabling holistic deep learning compiler optimizations with rTasks[C]//15th USENIX Symposium on Operating Systems Design and Implementation.USA: USENIX Association,2020:881–897.

[6]FEGADE P, CHEN T Q, GIBBONS P B, et al. Cortex:a compiler for recursive deep learning models[EB/OL].(2021-03-05)[2025-11-22].https://doi.org/10.48550/arXiv.2011.01383.

[7]NIU W, GUAN J X, WANG Y Z,et al. DNNFusion: accelerating deep neural network sexecution with advanced operator fusion[C]//PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.NewYork:ACM,2021:883-898.

[8]ZHENG S Z, CHEN S Y, SONG P D, et a1. Chimera: an analytical optimizing framework for effective compute-intensive operators fusion[C]//2023 IEEE International Symposium on High-Performance Computer Architecture(HPCA). Piscataway:IEEE, 2023: 1113-1126.

[9]CHEN T Q, ZHENG L M, YAN E,et a1.Learning to optimize tensor programs[EB/OL].(2019-01-08)[2025-11-22]. https://doi.org/10.48550/arXiv.1805.08166.

[10]ZHENG L M, JIA C F, SUN M M,et a1.Ansor:generating high-performance tensor programs for deep learning[EB/OL].(2023-10-15)[2025-11-22].https://doi.org/10.48550/arXiv.2006.06762.

[11] 王磊 . 面向深度学习编译器 TVM 的算子融合编译优化 [D]. 郑州 : 郑州大学 , 2024.

[12] 郑光飞 . 面向 TVM 深度学习编译器的算子融合技术研究 [D]. 郑州 : 郑州大学 , 2023.

【作者简介】

李明（1994—），女，河南驻马店人，博士研究生，工程师，研究方向：人工智能、编译优化以及异构调度等。

（收稿日期：2025-10-14 修回日期：2025-12-10）