大模型训练热点识别：跨线程依赖感知的关键路径分析

杨桐[1,2,3]，李宁[1,2,3]，黄波[1,2,3]，郭健美[1,2,3]
1. 华东师范大学数据科学与工程学院，上海 200062；
2. 上海市大数据管理系统工程研究中心，上海 200062；
3. 区块链数据管理教育部工程研究中心（华东师范大学），上海 200062

摘要

随着大语言模型的迅猛发展，模型规模不断扩大，其训练所需的计算资源和时间成本急剧增长，性能优化成为降低训练成本的关键手段，而准确的热点识别与性能瓶颈定位则是确保优化效果的关键前提。然而，大模型训练广泛使用的深度学习框架具有复杂的异构计算、异步执行与多线程协作模式，导致传统基于时间累加的分析方法易产生“假热点”。为解决上述问题，现有的工具通过分析和构建程序执行的关键路径，以期有效识别端到端性能瓶颈。然而，在大模型训练场景下，它们通常假设CPU线程间相互独立，因而无法准确刻画跨线程依赖关系，可能导致关键路径识别不完整、无法准确定位程序热点。为此，提出一种跨线程依赖感知的关键路径分析方法（Cross-Thread Dependency-Aware Critical Path Analysis，CTDA-CPA），其主要创新在于：引入跨线程事件关联机制，有效建模线程间依赖关系，从而构建完整的跨线程计算依赖图并准确识别出贯穿于CPU多线程与GPU多CUDA流之间的关键路径。实验结果表明，CTDA-CPA克服了现有方法在大模型训练场景下的分析局限，在不同规模的典型负载中均成功构建了完整的关键路径，并基于此准确定位了ResNet-18训练任务的数据加载瓶颈以及Llama 2分布式微调中的通信瓶颈。针对上述瓶颈进行的优化分别取得了47.06%和7.21%的训练任务整体性能提升，验证了该方法在实际优化工作中的有效性。

关键词

大模型训练；关键路径分析；跨线程依赖；热点识别；性能优化

中图分类号：TP302.7　　文献标志码：A

Hotspot Identification in Large Model Training: A Cross-Thread Dependency-Aware Critical Path Analysis Method

YANG Tong[1,2,3], LI Ning[1,2,3], GUO Jianmei[1,2,3], HUANG Bo[1,2,3]
1. School of Data Science and Engineering, East China Normal University, Shanghai 200062, China;
2. Shanghai Engineering Research Center on Big Data Management System, Shanghai 200062, China;
3. Engineering Research Center of Blockchain Data Management (East China Normal University), Ministry of Education, Shanghai 200062, China

Abstract

With the rapid evolution of Large Language Models (LLMs), the continuous expansion of model scale has led to a

dramatic increase in computational resources and training time costs. Consequently, performance optimization has become a key means for reducing training costs, where accurate hotspot identification and performance bottleneck localization serve as essential prerequisites for ensuring optimization effectiveness. However, modern deep learning frameworks widely employed in large model training are characterized by complex heterogeneous computing, asynchronous execution, and multi-threaded collaboration patterns. These complexities render traditional analysis methods based on time accumulation prone to identifying "false hotspot. " To address these issues, existing tools aim to effectively identify end-to-end performance bottlenecks by analyzing and constructing the critical path of program execution. However, in large model training scenarios, these tools typically assume independence among CPU threads, thus failing to accurately characterize cross-thread dependencies. This limitation leads to incomplete critical path identification, thereby hindering the accurate localization of program hotspot. To overcome these challenges, we proposed CTDA-CPA (Cross-Thread Dependency-Aware Critical Path Analysis). The main innovation lay in the introduction of a cross-thread event correlation mechanism to effectively model inter-thread dependencies. This mechanism enabled the construction of a complete cross-thread computation dependency graph and the accurate identification of the true critical path spanning across multiple CPU threads and GPU CUDA streams. Experimental results demonstrated that CTDA-CPA overcame the limitations of existing methods in large model training scenarios. It successfully constructed complete critical paths across typical workloads of varying scales, accurately locating the data loading bottleneck in ResNet-18 training tasks and the communication bottleneck in Llama 2 distributed fine-tuning. Optimizations targeting these identified bottlenecks yielded overall training task performance improvements of 47.06% and 7.21% respectively, which validates the effectiveness of the proposed method in actual optimization.

Key words
Large Model Training, Critical Path Analysis, Cross-Thread Dependency, Hotspot Identification, Performance Optimization

0 引言

随着大语言模型（Large Language Models，LLMs）和生成式人工智能（Generative Artificial Intelligence，GenAI）的迅猛发展[1]，模型规模不断扩大，训练所需的计算资源和时间成本急剧增长[2-3]，使得训练效率的优化成为学术界和工业界共同关注的焦点。尽管分布式训练[4]、算法优化[5]等技术不断演进，准确的性能分析与瓶颈识别始终是指导优化的关键前提——只有精确地定位到影响整体性能的关键瓶颈，识别出真正值得优化的热点（Hotspot），才能进行有效的代码、算法或系统层面的优化[6]，从而提升训练效率、降低计算成本。在本文中，“热点”是指直接影响程序端到端执行时间、并占用大量系统资源的关键事件——只有优化这些事件才能实际缩短程序的总运行时间。准确的热点识别对大模型训练具有重要意义：一方面，能够帮助研究团队快速定位瓶颈，缩短模型迭代周期；另一方面，考虑到大模型训练成本高昂（如GPT-4训练成本可能接近8 000万美元[7]），即使10%的性能提升也可能节省数百万美元开支；此外，准确的热点识别还能最大化GPU（Graphics Processing Unit，图形处理

器）集群利用率，避免因通信延迟或同步等待造成的算力浪费。

对于PyTorch[8]、TensorFlow[9]等现代深度学习框架，大模型训练是一个复杂的异构计算过程：CPU（Central Processing Unit，中央处理器）负责数据加载、预处理、算子调度等控制流任务，GPU则执行大规模并行计算密集型任务（如矩阵乘法、卷积等）[10]。为最大化硬件利用率，现有框架大量采用异步执行[11]、多线程协作[12]和多CUDA流（Compute Unified Device Architecture Stream，CUDA Stream）[11]等技术，使得CPU与GPU任务能够并行重叠执行。这种复杂的执行模式给传统的性能瓶颈分析带来了严峻挑战。现有的性能分析工具，例如Linux Perf[13]和NVIDIA Nsight[14]，往往侧重于CPU或GPU单侧的性能剖析，缺乏从异构协同执行的整体视角进行统一的自动化热点分析。PyTorch Profiler[15]、DLProf（Deep Learning Profiler）[16]等集成化工具虽能同时收集跨设备端到端数据，但大多基于对函数或算子执行时间的简单累加与排序来提供热点参考。这类方法很可能忽略了CPU-GPU计算重叠，从而导致“假热点”的问题，例如，某个GPU通信Kernel（在GPU上并行执行的计算函数）在单独分析中显示耗时200 ms，因执行时间大于其他Kernel被标识为热点，但实际上这200 ms与反向传播计算时间完全重叠，此时真正的瓶颈是反向传播阶段的计算（如CPU算子或GPU矩阵乘法Kernel），优化该通信Kernel并不会缩短总训练时间。

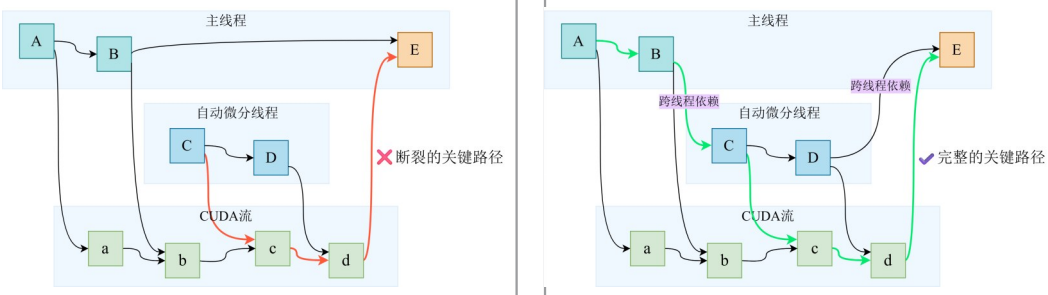
针对上述问题，关键路径分析（Critical Path Analysis，CPA）方法为异构系统的热点识别提供了新的思路。关键路径是指程序执行流中一系列前后依赖的事件，其总时长直接决定了整个程序的端到端执行时间，理论上只有缩短关键路径上事件的耗时才能有效提升程序性能。Kelley等人[17]最初将该方法用于项目调度，后续研究[18]验证了其在刻画CUDA程序CPU-GPU异构执行过程中的有效性。最新工作HTA（Holistic Trace Analysis）[19]已经将关键路径分析应用于深度学习场景，相比传统的简单累加排序方法，能够更准确地识别影响端到端性能的真正热点。

然而，我们在研究中发现，现有方法仍存在局限性：在构建程序的依赖关系图时假设了CPU线程间的独立性，将多个CPU线程视为独立并行的执行单元，忽略了Python GIL（Global Interpreter Lock，全局解释器锁）导致的线程间串行执行约束。在大模型训练场景中，这一问题尤为突出——以PyTorch为代表的深度学习框架广泛采用多线程机制，主线程负责前向计算和损失计算，而自动微分（Autograd）引擎常在子线程中执行反向传播，线程间受GIL约束存在严格的执行时序关系。现有方法忽视了这种由GIL约束产生的执行顺序依赖，无法正确关联具有时序约束的跨线程事件，导致关键路径构建不完整或错误：在从后向前将事件纳入关键路径的计算过程中，一旦子线程上的事件被纳入关键路径，由于缺乏与主线程的关联，关键路径会在子线程处断开。这将导致主线程上可能存在的数据加载或前向计算瓶颈被完全忽略，使得基于片面分析结果的优化决策难以取得预期效果。

为此，本文提出了一种跨线程依赖感知的关键路径分析方法（Cross-Thread Dependency-Aware Critical Path Analysis，CTDA-CPA）。该方法基于PyTorch Profiler[15]采集的运行时性能追

踪数据（Trace），通过事件聚合策略与依赖关系建模，构建准确反映程序执行逻辑的计算依赖图（Computation Dependency Graph，CDG）——一种有向无环图（Directed Acyclic Graph，DAG）[20]结构，进而基于节点时间戳比较并结合实际依赖语义分析，准确地识别出贯穿于CPU多线程与GPU多CUDA流之间的真实关键路径。针对现有方法无法正确关联跨线程事件的问题，CTDA-CPA引入跨线程事件关联机制，通过分析Python GIL约束下多线程串行执行的特性，基于时间戳顺序建立跨线程事件间的执行依赖，从而准确建模训练流程中关键的跨线程依赖关系，确保关键路径的完整性。

图1展示了现有方法与CTDA-CPA的关键路径识别效果对比。在典型的多线程异构计算场景中，主线程负责前向计算（节点A，B）并最终触发参数更新（节点E），自动微分线程（子线程）执行反向传播（节点C，D），CUDA流则在GPU上并行执行具体的计算任务（节点a，b，c，d）。如图1（a）所示，现有方法由于缺乏有效的跨线程依赖建模能力，无法建立线程间的依赖关系（B到C，D到E），导致可能生成一条断裂且错误的“关键路径”（图中红色路径）。相比之下，如图1（b）所示，CTDA-CPA通过跨线程事件关联机制，成功捕捉到了主线程与自动微分线程间的关键依赖，从而构建出完整、准确的关键路径（图中绿色路径）。



（a） 现有方法关键路径识别效果　　（b） CTDA-CPA关键路径识别效果

图1 现有方法与CTDA-CPA的关键路径识别效果对比

本文的主要贡献如下：

（1）提出并实现了一种跨线程依赖感知的关键路径分析方法——CTDA-CPA。该方法通过对程序执行过程进行准确的CDG建模与关键路径分析，有效克服了传统时间累加方法因计算重叠而导致的热点误判问题，并解决了现有关键路径分析工具在多线程场景下关键路径构建不完整的局限性，能够准确定位真正影响端到端训练时长的性能瓶颈。

（2）设计了一种跨线程事件关联机制。通过分析Python GIL机制，突破了现有工具对CPU线程独立性的假设，实现了主线程与自动微分子线程间依赖关系的建模，确保了CDG构建的完整性与准确性。

（3）实现了一个无侵入性的自动化分

析工具。CTDA-CPA完全基于标准的PyTorch Profiler数据进行离线分析，无需修改用户代码，可直接应用于包括DeepSpeed[21]、Megatron-LM[22]等主流分布式框架下的模型训练场景。

（4）通过实验验证了CTDA-CPA的优越性与有效性。相比HTA，CTDA-CPA在处理包含数十万事件的大型Trace数据时能够显著降低CDG规模，在大多数场景下将分析时间减少37%-51%，并在所有场景下保持0.90以上的关键路径覆盖率，成功处理了HTA无法完整分析的多线程大模型训练场景。在实际应用中，CTDA-CPA成功识别出ResNet-18[23]训练中的数据加载瓶颈和Llama 2[24]分布式微调中的梯度通信瓶颈，通过针对性优化分别实现了47.06%和7.21%的性能提升，证明了该方法在实际优化工作中的有效性。

（5）提供了CTDA-CPA的开源实现与可复现的实践范例。我们开源了CTDA-CPA的代码①，并提供了使用DeepSpeed与Megatron-LM框架训练的代码和配置。此外，为验证方法的有效性，还开源了ResNet-18数据加载与Llama 2通信瓶颈的优化代码，为大模型训练性能优化提供了可复现的实践参考。

本文的组织结构如下：第1章介绍相关工作并分析其局限性；第2章概述CTDA-CPA的整体框架；第3章详细阐述基于跨线程依赖感知关键路径分析的热点识别方法，包括CDG构建、关键路径识别以及热点提取；第4章通过性能开销分析、关键路径完整性评估、准确性对比验证以及两个实际优化案例，系统评估CTDA-CPA的有效性，并讨论方法的适用范围与理论基础；第5章对总结全文并展望未来研究方向。

1 相关工作

性能分析与热点识别是优化大模型训练效率的关键技术。根据分析方法的不同，现有工作可分为以下三类：基于时间累加的分析方法、基于程序结构的分析方法以及基于关键路径的分析方法。

基于时间累加的分析方法通过统计函数或算子的执行时间总和进行热点排序，是目前应用最为广泛的性能分析技术。在单设备分析工具方面，Linux Perf[13]通过硬件性能计数器和软件事件采样定位CPU端的性能瓶颈，Intel VTune[25]则提供了更为丰富的CPU微架构级分析能力。在GPU侧，NVIDIA Nsight Systems[14]能够展示CUDA Kernel的执行时间、内存带宽利用率等关键指标，虽然该工具也会收集CPU端事件并提供全局时间线视图，但其分析功能仅对CUDA Kernel进行统计，无法自动进行跨设备瓶颈分析。NVIDIA Nsight Compute[26]则专注于单个Kernel的深度性能分析。这类单设备工具缺乏对CPU-GPU异构计算模式的整体认知，难以分析跨设备依赖关系和计算重叠现象。为应对异构计算挑战，跨设备分析工具可同时采集并分析CPU和GPU端性能数据。PyTorch Profiler[15]借助CPU端插桩机制与GPU端NVIDIA CUPTI[27]接口，能够同时采集CPU算子和GPU Kernel的执行信息。TensorFlow Profiler[28]为TensorFlow框架提供了类似的功能，DLProf[16]则为深度学习应用提供了专门的可视化分析界面。然而，这些工具大多基于对函数或算子执行时间的简单累加与排序来提供热点参考，忽略了流水线并发执

①https://github.com/solecnugit/CDTA-CPA

行所产生的计算重叠的影响：一个执行时间很长的GPU Kernel，如果其执行完全被CPU任务覆盖，优化它可能不会带来任何端到端性能提升。

基于程序结构的分析方法侧重于利用程序的层次结构或调用关系，将性能指标从平铺的底层事件映射回高层的代码逻辑或模型结构。在通用程序分析方面，Gprof[29]是经典的基于调用图的性能分析工具，通过构建函数调用关系将时间开销归因到具体的函数调用链中。HPCToolkit[30]利用二进制分析技术重建调用栈，支持对大规模并行程序进行性能分析，能够将性能指标归因到源代码层级。TAU（Tuning and Analysis Utilities）[31]则通过灵活的插桩和采样机制，支持对MPI、OpenMP等并行程序进行跨模块的性能剖析与归因。在深度学习专用分析方面，Hotline Profiler[32]利用PyTorch的钩子机制，根据深度神经网络（Deep Neural Networks，DNN）[33]的层次结构进行自顶向下的瓶颈定位。DeepContext[34]则从代码执行路径的角度出发，通过构建调用上下文树将性能开销关联到具体的程序调用栈，并自底向上聚合执行时间和资源指标完成性能归因。上述方法虽然在语义关联上有所提升，但本质仍依赖时间累加统计，未能解决异构并行场景下计算重叠导致的热点误判问题，且大多难以准确处理CPU-GPU异构执行中复杂的依赖关系。

基于关键路径的分析方法从程序执行的依赖关系出发，识别决定端到端执行时间的事件序列，从理论上提供了更准确的性能瓶颈识别思路。关键路径方法最初由Kelley等人[17]提出并应用于项目调度领域，Hollingsworth[35]将其引入并行程序性能调优，证明了其在识别真实瓶颈方面的有效性。后续研究[18]进一步探索了如何通过关键路径分析刻画CPU-GPU异构执行过程，尝试解决计算重叠带来的分析难题。Schmitt等人[36]针对混合MPI-CUDA应用，提出了一种基于依赖图的等待状态分析方法，为异构系统的依赖关系建模提供了重要参考。深度学习领域，HTA[19]是基于关键路径分析的代表性工作，通过构建程序执行的依赖关系图来识别关键路径，进而定位热点，能够有效避免计算重叠导致的热点误判问题。然而，HTA在处理大模型训练场景时存在明显不足：其假设CPU线程间相互独立，无法准确捕获主线程与自动微分子线程间的依赖关系，导致关键路径构建不完整，进而影响热点识别的准确性。

综上所述，现有的性能分析方法在处理大模型训练的复杂执行模式时存在不同程度的局限性：基于时间累加的方法忽略了计算重叠的影响，容易产生“假热点”问题；基于程序结构的方法侧重于改善性能指标与代码逻辑的关联性，但其本质仍依赖时间累加统计，未能解决计算重叠带来的误判问题；基于关键路径的方法理论上能够更准确地识别瓶颈，但现有工作在多线程场景下存在依赖建模不完整的问题。本文提出的CTDA-CPA方法属于基于关键路径的分析方法，其核心创新在于引入跨线程依赖感知机制，能够准确建模主线程与自动微分子线程间的依赖关系，从而解决了现有关键路径分析方法无法正确处理多线程场景的问题，实现完整、准确的关键路径构建与热点识别。

2 方法概述

基于上述分析，我们设计并实现了一

种跨线程依赖感知的关键路径分析方法——CTDA-CPA，其继承了关键路径分析的核心思想：程序执行时间由关键路径决定，只有优化关键路径上的事件才能缩短总体运行时间。CTDA-CPA通过分析PyTorch Profiler[15]采集的运行时Trace数据，准确识别和建模PyTorch中主线程与子线程间的依赖关系，自动构建包含CPU算子、CUDA Kernel等事件的完整CDG，并识别出跨CPU多线程与GPU多CUDA流的关键路径。如图2所示，CTDA-CPA的整体流程主要包括以下四个阶段：

1. 数据采集：基于PyTorch Profiler获取训练过程的Trace数据；

2. CDG构建：将Trace中的事件（包括CPU算子、Python调用栈函数、CUDA Kernel和GPU内存操作等）根据所处的训练阶段（如前向计算、反向传播等）自动聚合为节点，并根据执行顺序、调用关系和同步机制所决定的因果依赖关系构建节点间的有向边；

3. 关键路径识别：基于时间戳比较并结合依赖关系分析，准确识别出关键路径；

4. 热点提取与排序：从关键路径中提取出未被完全重叠的热点事件，按总执行时长进行排序，生成最终建议优化的热点列表。

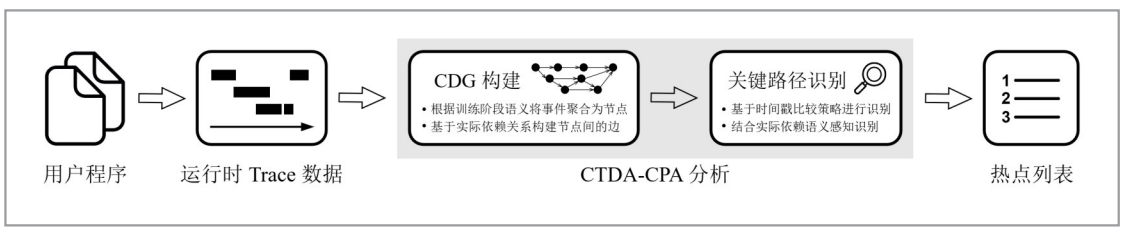


图2 CTDA-CPA的整体架构与工作流程

作为一个基于标准Profiler数据的离线分析工具，CTDA-CPA具有高度的无侵入性，无需用户对模型或训练代码进行任何修改即可完成热点识别。在可扩展性方面，CTDA-CPA不仅支持PyTorch原生训练，还可直接应用于DeepSpeed[21]、Megatron-LM[22]、FSDP[37]等主流分布式训练框架，覆盖了从单机到大规模分布式的各种训练场景。得益于统一的Trace数据建模方式与标准化的运行时事件格式，CTDA-CPA的设计理念同样适用于TensorFlow[9]、JAX[38]和MXNet[39]等其他深度学习框架，在这些系统中亦可扩展实现类似的事件依赖分析与关键路径识别。

3 基于跨线程依赖感知关键路径分析的热点识别

3.1 传统时间累加方法的局限性

传统性能分析工具通常基于事件执行时间的简单累加来识别热点。例如，Linux Perf[13]通过采样频率间接反映事件的时间占比，将其进行排序得到热点列表；PyTorch Profiler[15]和DLProf[16]则直接统计各事件的总执行时间并按降序排列来提供热点参考。然而，这种方法在CPU-GPU异构并行计算中存在显著局限性：由于大量的计算重叠和异步执行，对基于该

方法得到的热点结果进行优化可能只会减少资源使用量，而不会对程序的端到端执行时间产生实质性影响。

图3展示了一个典型的计算重叠导致热点误判的案例，图示上方为CPU线程的算子（Operator，Op）序列（Op A至Op E），下方为GPU端CUDA流上异步执行的Kernel序列。在该示例中，CUDA Kernel B是执行时间最长的事件，传统分析工具会将其识别为首要优化目标。但是，由于CPU和GPU的异步执行机制，Kernel B的整个执行过程都被CPU端的计算任务所覆盖，即使将其执行时间大幅缩短，程序总执行时间也不会改善，因为真正决定程序执行时间的是CPU端的计算链。这种“假热点”问题在大模型训练中尤为普遍，因为训练框架会通过异步Kernel启动、多CUDA流并行等技术最大化CPU和GPU的并行度。

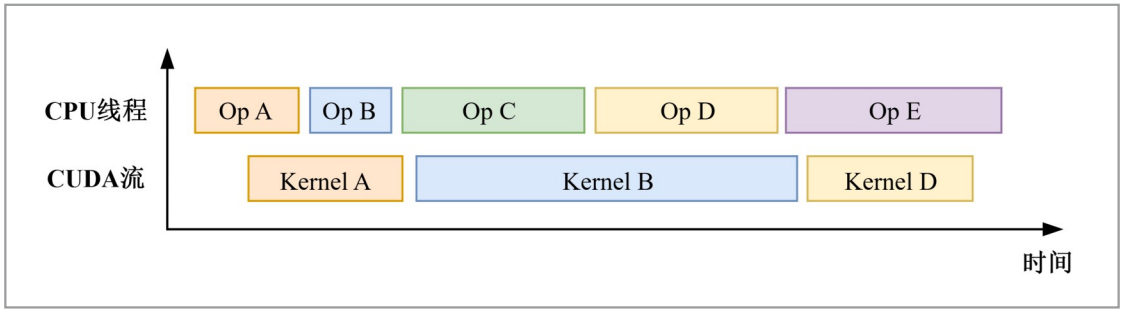


图3 计算重叠导致的热点误判示例

因此，准确的热点识别必须基于程序的关键路径，即决定程序端到端执行时间的事件序列。只有优化关键路径上的事件才能真正缩短程序运行时间。

3.2 计算依赖图构建

关键路径识别的基础是构建准确反映程序执行依赖关系的CDG。CTDA-CPA基于PyTorch Profiler采集的Trace数据进行建图，将CPU算子、Python调用栈函数、CUDA Kernel和GPU内存操作等事件组织成CDG结构。

3.2.1 事件聚合策略

为了降低CDG结构复杂度并提高后续关键路径识别效率，CTDA-CPA采用事件聚合策略，将CDG中的节点定义为一类事件聚合后的结果。这种聚合策略的核心原则是确保同一节点内的事件之间仅存在顺序执行依赖关系，从而保证CDG能够准确反映程序的并行结构。具体而言，对于CPU侧事件，按以下维度进行归并：

（1）训练阶段划分：根据事件在训练流程中的语义，将其归类为数据加载（DataLoad）、前向传播（Forward）、损失计算（Loss）、反向传播（Backward）、通信（Communication）、参数更新（Update）六个阶段。这种划分不仅减少了节点数量，还保留了训练流程的语义信息，便于后续性能分析。

（2）CUDA流边界识别：当CPU事件调用的GPU事件发生CUDA流切换时，需

要结束当前节点的聚合并开启新节点，以准确反映不同CUDA流上GPU事件的并行关系。

（3）同步操作识别：位于同步操作（如cudaDeviceSynchronize等）前后的事件会被划分到不同的节点。这些事件往往是CPU-GPU交互的关键点，需要在聚合时予以区分。

对于GPU事件，其聚合策略则依据对应的cudaLaunchKernel等CUDA运行时调用在CPU事件序列中的上下文位置进行映射与合并，即属于同一个CPU聚合节点发起的所有GPU事件会被聚合到一个GPU节点中。这种设计既保持了CPU-GPU的调用关系，又避免了过度细粒度的节点划分。

此外，CTDA-CPA在节点构建阶段同时保留并整合Python函数栈的调用信息，识别对训练流程产生实质阻塞效应的上层函数调用，使方法能够同时捕捉底层计算瓶颈与上层调度瓶颈。

3.2.2 依赖关系建模

构建好CDG节点后，下一步是建立节点间的依赖边。受Schmitt等人所提出的基于事件依赖图的等待状态分析方法[36]的启发，CTDA-CPA基于以下四类依赖关系进行边的构建：

（1）顺序执行依赖：同一CPU线程内的事件节点按照执行顺序形成依赖链，即后执行的事件依赖于先执行的事件。类似地，同一CUDA流上的GPU事件也遵循严格的顺序依赖。这种依赖反映了硬件执行的基本约束。

（2）调用依赖：CUDA Kernel或GPU内存操作事件依赖于发起调用的CPU事件。这种依赖体现了CPU对GPU的控制关系。由于GPU的异步执行特性，CPU发起调用后可以立即返回继续执行，因此这种依赖是单向的——GPU事件依赖CPU事件，但CPU后续事件不一定依赖GPU事件的完成。

（3）同步依赖：这是异构计算中较为复杂的依赖类型，可以进一步分为两类：

① CPU-GPU同步依赖：cudaDeviceSynchronize会阻塞CPU直到所有CUDA流上的任务完成，因此其后面执行的CPU事件同步依赖于调用时刻所有CUDA流的最后一个事件；cudaStreamSynchronize只等待指定的CUDA流完成，即其后的CPU事件依赖于该指定CUDA流的最后一个事件；cudaEventSynchronize只等待特定的CUDA事件完成，通过事件ID匹配建立精确的点对点同步依赖。

② GPU-GPU同步依赖：不同CUDA流之间可能因为数据依赖或显式同步产生依赖关系。这种跨CUDA流依赖主要通过cudaEventRecord、cudaStreamWaitEvent等CUDA事件同步机制实现，是GPU并行执行中的重要约束。

（4）跨线程依赖：这是CTDA-CPA引入的新型依赖关系，通过跨线程事件关联机制来准确建模线程间的执行约束。由于Python GIL的存在，Python解释器在任何时刻只允许一个线程执行Python字节码，多个Python线程需要竞争GIL并以时间片轮转方式串行执行，在全局时间轴上具有严格的先后顺序关系。这种GIL约束使得看似并行的多线程实际上存在隐式的时序依赖关系。在大模型训练中，主线程负责前向计算的调度，而自动微分引擎会创建专门的子线程执行反向传播，这两个线程受GIL约束存在执行顺序依赖。因此，必须进行跨线程的依赖关系分析，以准确

捕获线程间隐式的时序约束。

然而，HTA 将 CPU 线程视为独立并行的执行单元，无法识别这种由 GIL 产生的跨线程时序约束。图 4 通过示意图展示了这一局限性。图中自上而下分别展示了 CPU 主线程、自动微分子线程及 GPU CUDA 流的执行序列。其中，带有红色边框的节点及红色虚线箭头分别表示 HTA 识别出的关键路径事件与依赖关系，而无红色边框的节点则表示被判定为非关键路径上的事件（为突出核心观点，图中省略了无关的计算 Kernel 等事件）。可以看出，HTA 虽然能够识别出最后的参数更新事件、GPU 上的通信 Kernel 以及子线程中的反向传播事件，但其构建的关键路径在反向传播阶段起始处发生了断裂（如图中粉色标签所示）。这正是由于未能有效建模跨线程依赖，HTA 在识别关键路径时无法从子线程继续向主线程回溯，导致主线程上耗时较长的数据加载、前向计算和损失计算等关键环节被错误地排除在关键路径之外。这种路径断裂使得分析结果具有片面性，无法为性能优化提供完整有效的指导。

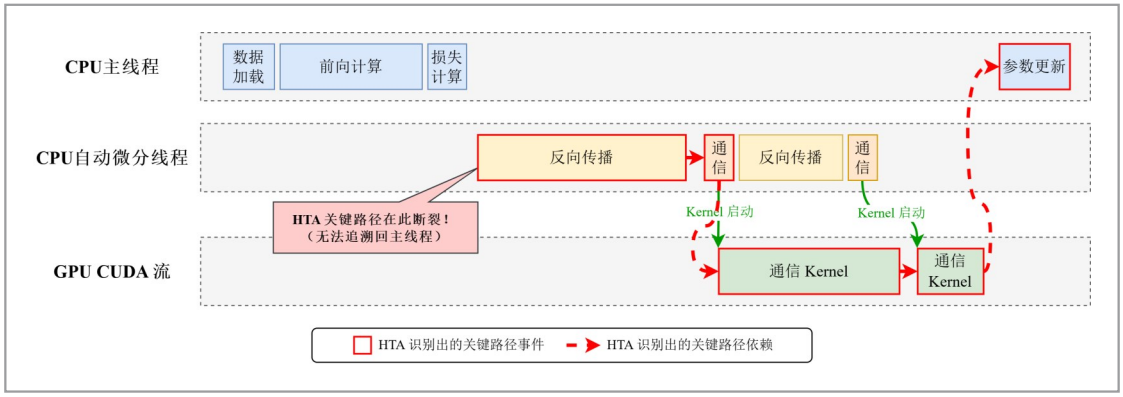


图 4 HTA 在多线程场景下的关键路径识别局限性示意图

基于对 GIL 机制的理解，CTDA-CPA 设计了跨线程事件关联机制，采用了简洁有效的处理方式：将主线程和子线程的事件视为在同一个逻辑线程中顺序执行，按照事件的实际时间戳顺序建立依赖关系。这种处理方式既反映了 GIL 约束下的实际执行模式，又避免了复杂的跨线程同步分析，确保了依赖图构建和后续关键路径识别的正确性。

综合上述节点聚合策略和四类依赖关系建模，CTDA-CPA 最终构建出准确反映程序执行逻辑的 CDG，其拓扑结构示意图如图 5（a）所示。图中蓝色节点表示聚合后的 CPU 事件，绿色节点表示聚合后的 GPU 事件（不同行代表不同的 CUDA 流）。节点间的连接线表示依赖关系：水平箭头为顺序依赖，垂直箭头为调用依赖，斜向箭头则表示跨执行单元的同步依赖。通过跨线程依赖建模，CTDA-CPA 能够将分散在不同线程中的事件正确关联，构建出完整的执行依赖图，为后续准确的关键路径识别奠定了基础。

3.3 关键路径识别算法

在传统的项目管理和电路分析中，关

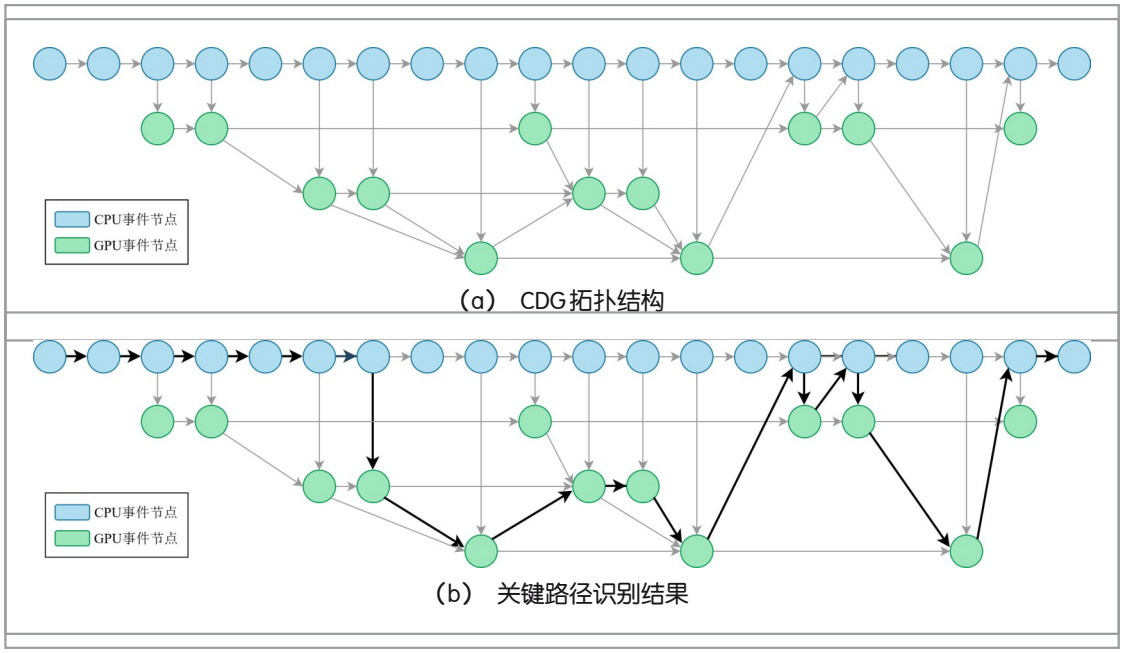


图5 CTDA–CPA计算依赖图建模与关键路径识别

键路径通常通过最长路径算法进行识别。该算法隐含“尽早调度”假设，即所有活动在其前驱完成后立即开始执行。在此假设下，当节点存在多个前驱时，执行时间最长的前驱必然最晚结束，因此可通过累计路径长度（即各节点执行时间之和）来识别关键路径。然而，在大模型训练的事件执行分析场景中，每个事件节点的开始时间由实际运行时调度决定，上述“执行时间最长即结束最晚”的等价性不再成立。此时，关键路径的识别必须直接比较各前驱节点的实际结束时间，选择结束时间最晚的前驱作为关键路径上的前驱节点（不考虑CPU同步语义），该节点直接决定了当前节点的实际开始时间，进而影响整个程序的端到端执行时间。

如图6所示，节点A、B、D位于CUDA流1上，节点C位于CUDA流2上与其他节点并行执行。节点A有两个后继节点B和C，它们都是节点D的前驱。尽管节点B的执行时间长于节点C，但由于节点C的开始时间较晚，其实际结束时间反而晚于节点B。传统最长路径算法基于尽早调度假设，会因节点B的执行时间更长而选择路径A→B→D作为关键路径。然而，从实际执行时序可以看出，节点D的开始时间实际上受节点C的结束时间，而非节点B。因此，真正的关键路径应该是A→C→D，这表明传统最长路径算法在固定时间戳场景下可能会产生错误的结果。

因此，CTDA–CPA提出了一种基于节点时间戳比较并结合实际依赖语义分析的关键路径识别算法。该算法从程序终点开始回溯，根据前驱节点的结束时间进行选择，并结合事件间的依赖类型动态决策：当前节点位于CPU端时，若该节点同步依赖于GPU节点，优先选择对应GPU端的前驱节点纳入，因为同步语义表明该CPU节点的开始时间受其同步依赖的GPU节点的完成时间约束；若无同步依赖，则继续沿CPU端依赖向前纳入。当前节点位于GPU端时，需要准确计算所有前驱节点的

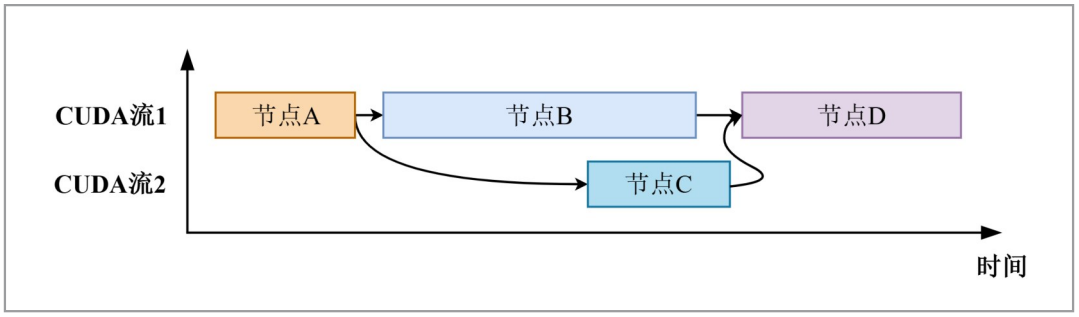


图6 传统最长路径算法在多CUDA流场景下的路径识别错误示例

有效结束时间：对于GPU前驱节点，直接使用其实际结束时间；对于CPU前驱节点，查找其中第一个cudaLaunchKernel事件的时间作为有效结束时间，然后选择有效结束时间最晚的前驱节点纳入关键路径。算法循环执行上述回溯过程，逐步将前驱节点纳入关键路径，直至到达程序起点，最终构建出完整的关键路径。算法1展示了CTDA-CPA关键路径识别的详细过程。

算法1 CTDA-CPA关键路径识别算法

输入：CPU事件节点集合N_{CPU}，GPU事件节点集合N_{GPU}，依赖关系集合R，CUDA运行时信息I_{cuda}
输出：关键路径$P_{critical}$

```
// 步骤1：初始化与构建反向依赖映射
dependency_map ← ∅
for each (source, target) ∈ R do
    dependency_map[target].append(source)
end for
// 步骤2：确定算法迭代的起始节点
last_cpu ← N_CPU中结束时间最晚的节点
last_gpu ← N_GPU中结束时间最晚的节点
if last_gpu.end_time > last_cpu.end_time then
    current ← last_gpu
else
    current ← last_cpu
end if
P_critical ← [current]
// 步骤3：回溯构建关键路径
while dependency_map[current] ≠ ∅ do
    predecessors ← dependency_map[current]
    if current ∈ N_CPU then
        sync_events ← FindSyncEvents(current, I_cuda)
        if sync_events ≠ ∅ and predecessors中存在GPU节点 then
            // 有同步事件，切换到GPU端
            current ← predecessors中结束时间最晚的GPU节点
        else
            // 无同步事件，或无GPU前驱，继续在CPU端
            current ← predecessors中结束时间最晚的CPU节点
        end if
    else // current ∈ N_GPU
        // 计算所有前驱的有效结束时间
        for each pred in predecessors do
            if pred ∈ N_GPU then
                pred.effective_end_time ← pred.end_time
            else // pred ∈ N_CPU，处理计算重叠
                t_launch ← GetKernelLaunchTime(pred, I_cuda)
                pred.effective_end_time ← t_launch
            end if
        end for
        // 选择结束时间最晚的前驱作为下一个节点
        current ← argmax(predecessors, effective_end)
    end if
    P_critical.append(current)
end while
return reverse(P_critical) // 返回正序的关键路径
```

该算法的核心设计体现在两个方面：

第一，对CPU前驱节点有效结束时间的准确计算（算法第28–36行）。在CDG中，GPU聚合节点N_{GPU}包含一系列连续执行的CUDA Kernel，其起始时间由内部第一个Kernel（记为k_1）的开始时间决定。由于CUDA的异步执行机制，k_1的开始时间仅取决于CPU端第一个cudaLaunchKernel的调用时刻。任何晚于该cudaLaunchKernel的CPU操作（包括启动后续Kernel k_2、k_3等）都与N_{GPU}的执行存在重叠，不构成N_{GPU}开始执行的前提条件。因此，将CPU前驱节点的有效结束时间定义为其内部第一个cudaLaunchKernel的执行时间，从而准确刻画CPU操作对GPU后继节点的真实时间约束。若直接使用完整CPU节点的结束时间，虽然不会影响关键路径的拓扑结构，但会导致与GPU执行重叠的CPU事件被错误计入关键路径，造成后续热点提取时的时间统计偏差。第二，结合节点时间戳比较与同步依赖语义分析的前驱选择策略（算法第19–26行和第37–38行），确保能够在多个前驱中准确选择关键路径上的正确节点。关于算法整体正确性的形式化证明，详见第4.5.2节的分析。图5（b）展示了在CDG上所识别出的关键路径示意图，黑色路径标识了最终识别出的关键路径。

3.4 基于关键路径的热点提取

识别出关键路径后，需从中提取真正的热点。当CPU节点及其调用的GPU节点同时位于关键路径上时，需要仔细分析其时间贡献以避免重复计算。如3.3节所述，CPU节点的有效结束时间由其第一个cudaLaunchKernel的执行时间决定，这一原理同样适用于热点提取阶段的时间统计。图7展示了这一典型场景：上方红色虚线框为关键路径上的CPU节点，下方绿色虚线框为对应的GPU节点。CPU节点从T_{start}开始执行，包含多个算子（如Op A、Op B）以及cudaLaunchKernel调用，该调用触发GPU端Kernel执行。虽然整个CPU节点都在关键路径上，但只有从T_{start}到第一个cudaLaunchKernel结束时刻T_{launch}之间的部分真正贡献了执行时间（图中蓝色区域），称这段时间为“有效执行时间”。T_{launch}到T_{end}期间，CPU继续执行后续事件（Op C、Op D等），但这部分与GPU Kernel执行存在时间重叠（图中灰色斜线区域）。CTDA-CPA的处理策略是：识别CPU节点中第一个cudaLaunchKernel事件的结束时间T_{launch}作为有效结束时间，在热点统计时只计入有效执行时间内的事件（如图中的Op A和Op B），而将重叠区域内的操作排除，从而避免重复计算。

经过节点间的重叠分析后，CTDA-CPA对所有保留在关键路径上的事件进行聚合统计：按事件名称进行分组，计算每组的总执行时间及其在程序端到端执行时间中的占比，最后按总时间降序排列，生成最终的热点列表。图8以DeepSpeed框架下Llama 2 13B的分布式微调任务为例，展示了CTDA-CPA的实际分析输出。从图中可以看出，热点列表结果按各事件耗时占比降序排列，其中通信开销占比最大（22.8%），其次是矩阵乘法和框架调度开销。该列表清晰地展示了真正影响训练性能的瓶颈，基于这些信息，开发者可以有针对性地进行优化，例如使用梯度压缩或更高效的通信拓扑来减少AllReduce[40]开销、使用更优化的BLAS库来加速矩阵运算、或者通过算子融合来减少框架调度开销等。

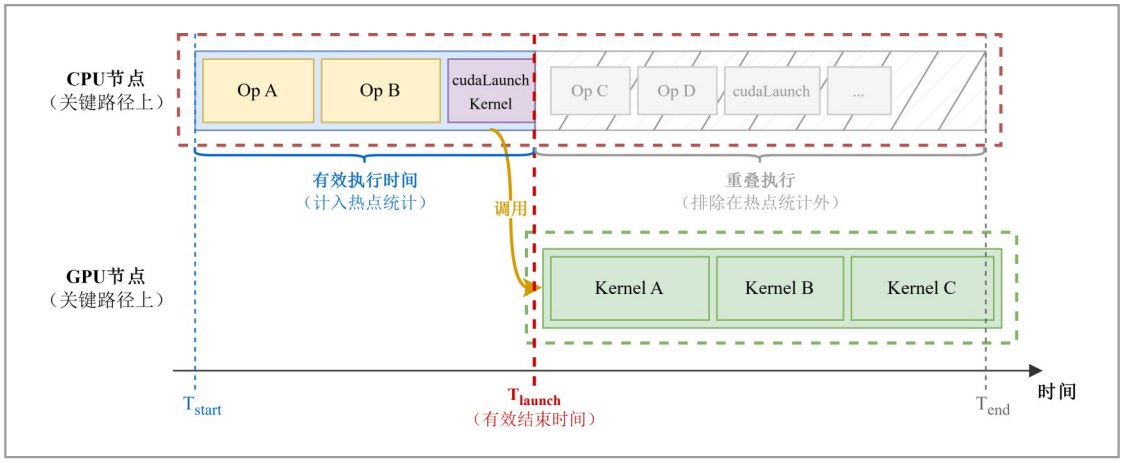


图7 基于有效执行时间的关键路径节点重叠处理示意图

```
(llama2_torch2.4.0) → ctda-cpa python hotspot_analysis.py --input llama2.parquet --top_functions 10
程序总执行时长为 4091.97 ms。
以下是在程序执行关键路径上的前 10 个热点事件：
name                                                                                                | category        | duration  | proportion | prop_in_total
----------------------------------------------------------------------------------------------------------------------------------------------------------
ncclDevKernel_AllReduce_Sum_bf16_RING_LL(ncclDevComm*, unsigned long, ncclWork*)                    | kernel          | 933.136ms | 53.48%     | 22.8%
aten::mm                                                                                            | cpu_op          | 203.438ms | 11.66%     | 4.97%
<built-in method run_backward of torch._C._EngineBase object at 0x75f1c7b29790>                    | python_function | 120.846ms | 6.93%      | 2.95%
ncclDevKernel_AllGather_RING_LL(ncclDevComm*, unsigned long, ncclWork*)                             | kernel          | 99.268ms  | 5.69%      | 2.43%
aten::mul                                                                                           | cpu_op          | 98.241ms  | 5.63%      | 2.4%
peft/tuners/lora/bnb.py(439): forward                                                               | python_function | 76.099ms  | 4.36%      | 1.86%
aten::transpose                                                                                     | cpu_op          | 55.129ms  | 3.16%      | 1.35%
<built-in method to of Tensor object at 0x75f10ab9ea70>                                             | python_function | 54.239ms  | 3.11%      | 1.33%
aten::empty                                                                                         | cpu_op          | 53.727ms  | 3.08%      | 1.31%
aten::empty_strided                                                                                 | cpu_op          | 50.777ms  | 2.91%      | 1.24%

热点函数的详细信息见 top_cpu_events.json 和 top_gpu_events.json 文件。
```

图8 CTDA-CPA对Llama 2 13B模型的实际热点分析结果

4 实验评估

本章通过系统的实验评估验证CTDA-CPA的优越性、完整性和有效性。实验从四个维度展开：首先分析CTDA-CPA的性能开销，验证其在处理大规模训练数据时的效率；其次通过引入关键路径覆盖率（CPCR）指标，定量评估关键路径识别的完整性；然后通过与HTA的对比实验验证热点识别的准确性；最后通过两个优化实验展示CTDA-CPA在指导性能优化方面的实际价值。实验设计覆盖了六个典型场景，涵盖从传统CNN模型到最新大语言模型的多种架构、从千万级到百亿级的参数规模、以及从单卡到16卡的分布式配置。实验硬件环境由两个计算节点组成，每个节点配备2×Intel Xeon Gold 6326 CPU（2.90 GHz，16核）和8×NVIDIA GeForce RTX 3090 GPU（24 GB显存），共计16张GPU。软件环境包括PyTorch 2.4.0、CUDA 12.6、DeepSpeed 0.14.4和Megatron-LM 3.0.0等。

4.1 性能开销分析

4.1.1 数据采集与存储优化

CTDA-CPA的数据采集完全依赖PyTorch Profiler，因此运行时开销取决

于Profiler的配置。为了降低存储和分析开销，CTDA-CPA实现了可选的数据压缩功能，将JSON格式的Trace数据转换为Parquet格式。如表1所示，压缩后的文件大小减少了90%以上，同时数据加载速度提升了3-7倍。这对于需要反复分析的场景尤其重要。

表1 Trace数据压缩前后性能对比

模型	原始Trace数据		压缩后Trace数据		存储空间节约率	数据加载加速比
	文件大小	加载时间	文件大小	加载时间		
GPT-2 Large	34.64 MB	0.45 s	3.22 MB	0.12 s	90.70%	3.75
Llama 2 13B	179.47 MB	2.91 s	14.15 MB	0.41 s	92.12%	7.10
Llama 2 13B该Trace数据额外记录了Python调用栈信息。	3.11 GB	21.19s	214.85 MB	5.45 s	93.25%	3.89

4.1.2 CDG构建与分析效率

表2和表3展示了CTDA-CPA与HTA在处理不同规模训练负载时的性能对比。如表2所示，实验设计了6个典型训练场景（标记为S1至S6），所有场景的Trace数据均采集自1个训练步，以确保不同场景间的对比具有一致的粒度基准。实验设计覆盖了三个关键维度：（1）模型架构：包括传统CNN模型（ResNet-18）和多种主流大语言模型（Llama 2、GPT-2、Qwen3），涵盖了从经典架构到最新发布模型的广泛范围；（2）参数规模：从ResNet-18（约11 M参数）、GPT-2 Large（约774 M参数）到Llama 2 7B、Qwen3 8B，再到Llama 2 13B，覆盖了从轻量级模型到十亿级参数模型的典型规模区间；（3）分布式规模：从单卡场景（S1、S2）到8卡分布式（S3、S4、S5），进一步扩展至16卡（S6）。

表2 实验配置与不同规模训练负载说明

场景编号	模型	数据集	任务类型	框架	运行环境
S1	ResNet-18	CIFAR-10	训练	PyTorch	1 GPU
S2	Llama 2 7B	Belle 0.5M	微调	PyTorch	1 GPU
S3	Qwen3 8B	Alpaca	微调	DeepSpeed	8 GPUs
S4	Llama 2 13B	Belle 0.5M	微调	DeepSpeed	8 GPUs
S5	GPT-2 Large	WikiText-2	训练	Megatron-LM	8 GPUs
S6	Llama 2 13B	Belle 0.5M	微调	DeepSpeed	16 GPUs

表3 CTDA-CPA与HTA的CDG构建与分析效率对比

场景编号	Trace事件数	HTA			CTDA-CPA		
		节点数	边数	分析时间	节点数	边数	分析时间
S1	3 802	7 586	7 937	0.31 s	13	19	0.17 s
S2	453 293	906 564	960 743	35.99 s	76	132	17.72 s
S3	531 542	1 062 988	1 087 331	45.45 s	134	236	25.28 s
S4	515 405	1 030 680	1 070 893	42.90 s	137	242	24.16 s
S5	95 942	179 942	186 462	6.48 s	3 012	6 010	13.84 s
S6	516 656	1 033 182	1 082 461	43.45 s	165	296	27.37 s

从表3的CDG构建规模来看，HTA采用直接映射策略，将Trace中每个事件的起始时间点和终止时间点作为独立的CDG节点，导致构建的图结构规模极其庞大。以Llama 2 7B单卡微调（S2）为例，45万个原始事件产生了超过90万个CDG节点和近100万条边，严重影响了关键路径分析效率。相比之下，CTDA-CPA通过智能的事件聚合策略，将一系列语义相关且执行特征相似的事件归并为同一个逻辑节点，显著降低了CDG规模。具体而言，CTDA-CPA将ResNet-18（S1）的节点数缩减了292倍（从3 802降至13）；在大规模模型场景（S2-S6）中效果更为显著，其中Llama 2 7B单卡微调（S2）的节点数缩减高达5 964倍（从453 293降至76）。这种大幅度的规模缩减为高效的关键路径分析奠定了基础。

在分析时间方面，CTDA-CPA在大多数场景下展现出明显的效率优势。对于大模型训练或微调场景，HTA的分析时间随事件规模急剧增长。例如，对于Llama 2 13B分布式微调（S4）过程中1个训练步所采集的数据，需要42.90 s才能完成分析。而CTDA-CPA通过精简的CDG结构，将分析时间缩短至24.16 s，降幅约44%。在其余多数场景下，CTDA-CPA相比HTA的分析时间降幅均保持在37%-51%的范围内。需要注意的是，在GPT-2 Large的Megatron-LM训练场景（S5）中，CTDA-CPA的分析时间（13.84 s）长于HTA（6.48 s），且CDG节点数（3 012）显著高于其他大模型场景。这主要是由于Megatron-LM框架实现的张量并行机制引入了大量细粒度的通信操作，这些频繁的小规模通信事件在聚合阶段需要更复杂的处理逻辑，导致聚合效果受限、节点数量增加，从而增加了分析时间。尽管如此，CTDA-CPA在该场景中能够构建相对完整的关键路径，而HTA由于缺乏跨线程依赖建模能力，仅能识别出部分路径片段（详见4.2节）。

需要说明的是，表2和表3中的结果都是基于未开启Python调用栈的轻量级Profiler配置获得的。若需分析Python层的性能瓶颈，开启Python调用栈采集后，由于原始事件数量的显著增加CDG复杂程度和关键路径识别时间会有一定幅度的上升。

4.2 关键路径完整性评估

为定量评估关键路径识别结果的完整性，本文引入关键路径覆盖率（Critical Path Coverage Ratio，CPCR）指标，将其定义为关键路径上所有事件的累计执行时间与程序端到端执行时间的比值。该指标基于项目管理理论中“关键路径决定项

目总时长”的性质：在理想情况下，关键路径总耗时应等同于程序运行总时长（即CPCR=1）。但在实际训练场景中，受算子调度间隙、Profiler插桩开销及GPU Kernel启动延迟等因素影响，CPCR通常略小于1。即便如此，该指标仍能反映关键路径识别的完整程度——CPCR越接近1，意味着识别出的事件序列在时间覆盖上越接近真实的端到端关键路径，从而证明分析结果越完整。

表4展示了CTDA-CPA与HTA在各实验场景下的CPCR对比结果，其中“程序执行时间”为训练或微调1步的端到端运行时长，“关键路径时间”为各方法识别出的关键路径上所有事件的累计执行时间。在S1场景下，HTA与CTDA-CPA的CPCR分别为0.73和0.70，两者非常接近。这与4.3节中两种方法在该场景下热点识别结果高度一致的结论相吻合——该场景的关键路径恰好完全位于主线程内，不涉及跨线程依赖，故HTA能够正确处理。CTDA-CPA的CPCR比HTA略低，这是因为CTDA-CPA采用的事件聚合策略在识别关键路径时引入了一些精度牺牲，但如4.3节所述，这种牺牲通常不会影响最终的热点识别结果。

表4 CTDA-CPA与HTA的关键路径完整性对比

场景编号	程序执行时间	HTA		CTDA-CPA	
		关键路径时间	CPCR	关键路径时间	CPCR
S1	56.39 ms	41.12 ms	0.73	39.20 ms	0.70
S2	2 962.37 ms	1 870.39 ms	0.63	2 664.20 ms	0.90
S3	3 168.39 ms	2 165.07 ms	0.68	3 063.98 ms	0.97
S4	3 467.62 ms	2 205.46 ms	0.64	3 263.46 ms	0.94
S5	6 377.38 ms	4 568.39 ms	0.72	6 056.68 ms	0.95
S6	4 650.80 ms	2 981.06 ms	0.64	4 547.17 ms	0.98

在涉及跨线程依赖的S2-S6场景中，两种方法的差异十分显著。CTDA-CPA在所有场景下的CPCR均达到0.90以上，接近理论上限，表明其识别的关键路径具有较高的完整性；这些数值显著高于S1场景，是因为大模型训练中算子执行更为密集，事件间的调度间隙占比更低。相比之下，HTA的CPCR仅为0.63至0.72，存在明显的时间缺失。以S6场景为例，程序端到端执行时间为4 650.80 ms，CTDA-CPA的关键路径时间为4 547.17 ms（CPCR=0.98），几乎完全覆盖，而HTA仅为2 981.06 ms（CPCR=0.64）。这是因为HTA无法建立跨线程的依赖关系，导致关键路径在子线程处断裂，严重影响了关键路径的完整性。

4.3 准确性验证

热点识别准确性的评估本身就是一个挑战性问题。传统性能分析工具（如Linux Perf）能够识别出资源占用较高的函数或事件，若想判定其结果的准确性，往往需要工程师进行手工验证和实际优化测试才能确定。正如引言中所述，本文的“热点”是指真正影响程序端到端执行时间

的关键事件——只有优化这些事件才能实际缩短程序的总运行时间。然而，这种热点识别结果的准确性难以直接验证。在本文中，我们采用的评估方法是与现有工具HTA的结果进行对比。虽然HTA在多种场景下存在关键路径构建问题，但在其能够正确处理的场景中，其结果仍具有一定的参考价值。此外，为了进一步验证我们工具在实际场景中的有效性，我们在4.4节中进行了实验案例有效性验证，通过实际的性能优化效果来证明热点识别的准确性。

我们在ResNet-18训练场景（HTA唯一能正确处理的场景）中对比了两种方法的热点识别结果。实验结果表明，CTDA-CPA与HTA在热点识别上达到了高度一致：两种方法识别的前20个热点完全相同，且在所有关键路径事件的排序上序列相似度高达94.37%，显示出极强的一致性。

其中存在的细微差异主要源于CTDA-CPA中CDG节点对事件的聚合处理带来的精度牺牲。然而，这种偏差对实际应用的影响微乎其微。首先，在性能优化实践中，开发者通常只关注Top-3或Top-5的主要热点，这些热点由于执行时间占比巨大，其排序位置不会因微小的统计偏差而改变。其次，对于排名靠后的事件，即使存在一定的顺序变化，也不会影响优化决策，因为这些事件本身对整体性能的影响就很有限。

4.4 实验案例研究

4.4.1 ResNet-18数据加载优化

在ResNet-18单卡训练场景中，我们使用CTDA-CPA定位到最大热点位于数据加载（Data Loading）阶段中Pillow库的图像resize方法（<built-in method resize...>），该函数在程序端到端执行时间中的占比高达21.75%，远超其他任何事件。经过深入分析发现，该瓶颈源于DataLoader在每个训练步中都会动态地对原始图像进行尺寸调整，将不同大小的输入图像统一缩放到224×224的标准尺寸，造成了大量的时间浪费。

基于分析结果，我们设计了针对性的优化方案：将图像尺寸调整从在线处理改为离线预处理，在训练开始前对整个数据集进行一次性处理，将所有图像预先调整到目标尺寸并保存。预处理阶段采用多进程并行处理，充分利用多核CPU资源，有效提升了处理效率。同时在预处理阶段完成图像到张量的转换，并以PyTorch原生的张量格式存储，减少了训练时的数据转换开销。

优化后的性能提升显著：单个epoch的训练时间从75 s降至51 s，速度提升47.06%。此时数据加载不再是瓶颈，resize操作在程序端到端执行时间中的占比降至接近0。值得说明的是，由于CTDA-CPA在CDG构建阶段保留并整合了Python函数栈的调用信息，才能够将该resize操作识别为首要热点。相比之下，HTA的分析结果不包含Python调用栈信息，其识别的首要热点为aten::copy_算子，该算子在CTDA-CPA的分析结果中占比仅为5.04%，远低于resize操作的21.75%。更重要的是，aten::copy_算子在训练的各个阶段均有分布，仅凭此结果难以准确定位真正的性能瓶颈。

4.4.2 Llama 2 13B通信优化

在将Llama 2 13B的QLoRA[41]微调从单卡扩展到8卡时，扩展效率不理想——8卡的加速比仅为6.83，远低于理想的线性

扩展。我们使用CTDA-CPA进行分析，结果清晰指出NCCL通信Kernel为最大热点，其在程序端到端执行时间中的占比达22.8%。进一步分析发现，DeepSpeed Zero-2在梯度同步时为了适配Zero-2的切分策略，在每次梯度通信时都使用AllReduce操作，导致了大量不必要的通信开销。

基于上述分析，我们设计了两层优化策略：在通信原语层面，对于只需要部分参数梯度通信的场景，对剩余其他参数的梯度用0进行填充，从而实现在通信时将AllReduce操作替换为通信量仅为其一半的ReduceScatter[42]操作，显著减少通信量和通信时间；在算法层面，我们实现了一个运行时决策机制，根据梯度桶大小、不同的通信阶段等因素在每次梯度通信时自动地选择通信开销最低的Kernel，从而最大程度上提升梯度通信速度。

使用上述方法进行优化后，单步梯度通信时间（通信Kernel在GPU上的执行时间）从933.14 ms降至524.25 ms，减少43.8%。由于在训练过程中部分通信时间与反向传播计算并行执行，因此通信Kernel执行时间的优化并不会完全转化为端到端性能提升。最终，端到端训练速度提升7.21%，8卡扩展比从6.83提升至7.33。图9展示了优化前后的扩展效率对比，其中绿线表示理想的线性扩展，蓝线为原始DeepSpeed Zero-2实现的扩展效率，红线为优化后的扩展效率。可以看到优化后的扩展曲线更接近理想的线性扩展，特别是在4卡和8卡场景下改善较为明显。

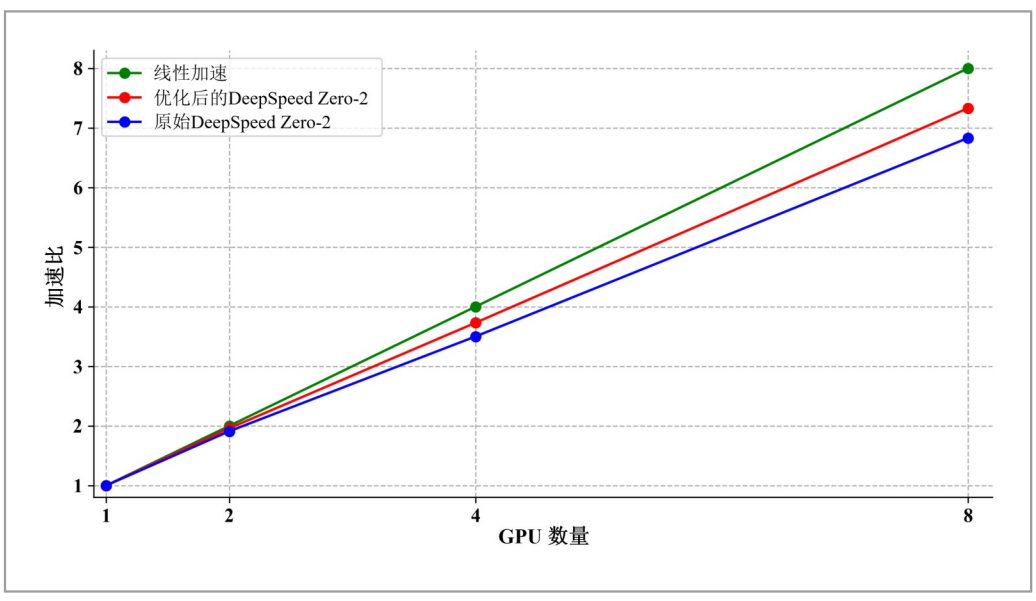


图9 Llama 2 13B微调任务的扩展效率优化效果对比

4.5 讨论

4.5.1 方法的适用范围

CTDA-CPA的跨线程依赖建模机制专门针对Python GIL的串行化特性设计，通过事件时间戳的全局顺序建立跨线程依赖关系，从而实现多线程训练场景下完整关键路径的构建。然而，这一方法在非Python环境下将不再适用。在纯C++实现的深度学习训练系统中，多线程可以真正并行执行，不受GIL约束。此外，Python 3.13已经提供了可选的无GIL执行模式（free-threading mode）[43]，尽管其目前仍处于实验阶段。在这些场景下，“基于时间戳的跨线程串行执行假设”将不再成立，可能导致虚假依赖关系的引入，影响关键路径识别的准确性。

针对上述非GIL约束场景，CTDA-CPA需要调整依赖建模策略，从依赖时间戳的隐式顺序关系转向基于显式同步原语的依赖分析。具体而言，需要在依赖建模模块中引入对互斥锁、条件变量、信号量等同步机制的识别能力，通过捕获这些同步事件来准确识别线程间的依赖关系。值得强调的是，完成上述调整后，CTDA-CPA的整体分析框架仍然适用，无需根本性重构。

4.5.2 关键路径准确性分析

CTDA-CPA的依赖建模遵循Happens-Before形式化框架[44]。在该框架下，CDG中的有向边当且仅当事件间存在Happens-Before关系时建立，确保依赖图的拓扑结构与程序的实际因果关系保持一致。

CTDA-CPA建模的四类依赖关系覆盖了PyTorch训练中所有可观测的因果约束。顺序执行依赖对应Happens-Before关系中同一进程内的程序顺序；调用依赖和同步依赖对应CUDA编程模型规范[11]中定义的主机-设备交互语义；跨线程依赖的准确性由Python GIL机制的串行化语义保证——由于GIL的存在，Python解释器在任意时刻最多允许一个线程执行字节码，CTDA-CPA通过比较时间戳建立依赖边，这与GIL约束下的实际因果关系完全一致。

关键路径识别算法的正确性可通过归纳法证明。首先，算法从程序终点（结束时间最晚的节点）开始回溯，该节点必然位于关键路径上。其次，设当前节点v在关键路径上，需分两种情况讨论：（1）当v为CPU节点且存在对GPU节点的同步依赖时，同步语义表明v的执行被阻塞直至所依赖的GPU操作完成，即v的开始时间直接受该GPU节点完成时间的约束，因此算法优先选择同步依赖所指向的GPU节点作为关键路径上的下一个节点是正确的。（2）当v无同步依赖时，其开始时间受所有前驱节点约束，即$s(v) \geq \max_{u \in \mathrm{pred}(v)} \mathrm{effective_end_time}(u)$。其中，$s(v)$表示节点$v$的实际开始时间，$\mathrm{pred}(v)$表示节点$v$在CDG中的所有直接前驱节点集合，$\mathrm{effective_end_time}(u)$表示前驱节点$u$的有效结束时间。算法选择有效结束时间最晚的前驱$u^*$作为关键路径上的下一个节点是正确的，因为只有$u^*$的结束时间直接约束了$v$的实际开始时间。最后，由于CDG是有向无环图，算法必然在有限步内回溯到程序起点，从而构建出完整的关键路径。

尽管CTDA-CPA在理论层面具备上述准确性保证，但当前模型仍存在一定的局限性。首先，CTDA-CPA所采用的事件聚合策略（见3.2.1节）在提升分析效率的同时，也可能在特定场景下引入偏差。节点聚合的前提假设是同一节点内的事件紧密连续执行，然而对于GPU节点，当其

中的CUDA Kernel执行较为稀疏时，当前Kernel的真实前驱可能是结束时间较晚的CPU端的启动调用事件而非前一个Kernel，此时聚合策略所建立的依赖关系与实际执行时序存在偏差，可能影响关键路径识别的准确性。在大模型训练这一本文主要关注的应用场景下，由于CUDA Kernel执行通常非常密集，上述问题的影响较为有限。本文采用该策略主要基于两方面考量：其一，节点聚合使CDG规模大幅降低（如表3所示，节点数缩减可达数千倍），显著降低了关键路径分析的计算复杂度，带来了可观的效率收益；其二，如4.3节实验所述，这种准确性牺牲通常不会影响最终的热点识别结果，排名靠前的主要热点由于执行时间占比显著，其排序位置不会因聚合引入的微小统计偏差而改变。

其次，当前依赖模型仅捕获CUDA API层面的显式同步语义，未建模流式多处理器（Streaming Multiprocessor，SM）数量、显存带宽等GPU硬件资源约束，也未分析Kernel间的显存访问模式，因此可能遗漏因资源争用或隐式数据读写而产生的实际依赖关系。此外，CTDA-CPA依赖PyTorch Profiler提供的时间戳数据，若存在测量误差或时钟漂移，也可能影响跨线程依赖关系的判定精度。上述局限性构成了未来工作的改进方向。

5 总结与展望

本文提出并实现了一种专门针对大模型训练热点识别设计的跨线程依赖感知关键路径分析方法——CTDA-CPA。在大模型已成为人工智能发展核心驱动力的今天，准确识别热点、定位性能瓶颈对于提升模型训练效率、降低计算成本具有重要意义。针对传统时间累加方法因忽略计算重叠而产生的热点误判问题，以及现有关键路径分析工具在多线程协作场景下的局限性，CTDA-CPA通过跨线程事件关联机制准确建模了主线程与自动微分子线程间的依赖关系，实现了完整关键路径的构建，能够从复杂的异构执行环境中准确识别出真正影响训练效率的性能瓶颈。

尽管CTDA-CPA已展现出强大能力，但仍有可探索的未来方向。例如，进一步优化张量并行等通信密集型场景下的CDG构建效率，减少事件聚合的时间开销；支持CUDA Graph等优化技术的性能分析；探索自适应的聚合粒度控制，在分析效率和精度之间实现更好的平衡。随着大模型规模的持续增长和训练场景的日益复杂，准确高效的性能分析工具将变得愈发重要。CTDA-CPA作为这一方向的探索，不仅解决了当前的技术挑战，也为未来的研究奠定了基础。我们相信，通过持续的技术创新和工程实践，性能分析工具将在推动大模型训练效率提升、降低AI发展成本方面发挥越来越重要的作用。

参考文献：

[1] CHANG Y P, WANG X, WANG J D, et al. A survey on evaluation of large language models[J]. ACM Transactions on Intelligent Systems and Technology, 2024, 15(3): 1-45.

[2] ACHIAM J, ADLER S, AGARWAL S, et al. Gpt-4 technical report[EB]. arXiv preprint, 2023, arXiv:2303.08774.

[3] SEVILLA J, HEIM L, HO A, et al. Compute trends across three eras of machine learning[C]//2022 International Joint Conference on Neural Networks (IJCNN), July 18-23, 2022, Padua,

Italy. New York: IEEE, 2022: 1-8.

[4] 郑纬民. 分布式技术在大模型训练和推理中的应用[J]. 大数据,2024,10(5):1-10.
ZHENG W M. Application of distributed techniques in large language model training and inference[J]. Big Data Research, 2024, 10(5): 1-10.

[5] 陶伟,王健宗,张旭龙,等. 大语言模型长文本推断优化技术综述[J]. 大数据,2025,11(06):72-94.
TAO W, WANG J Z, ZHANG X L, et al. Long-context inference optimization for large language models: a survey[J]. Big Data Research, 2025, 11(06): 72-94.

[6] AMDAHL G M. Validity of the single processor approach to achieving large scale computing capabilities[C]//Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, NJ, USA. New York: ACM, 1967: 483-485.

[7] MASLEJ N, FATTORINI L, PERRAULT R, et al. Artificial intelligence index report 2025[J]. arXiv preprint arXiv: 2504.07139, 2025.

[8] PASZKE A, GROSS S, MASSA F, et al. PyTorch: An imperative style, high-performance deep learning library[C]// Advances in Neural Information Processing Systems 32. Red Hook, NY: Curran Associates, Inc., 2019: 8024-8035.

[9] ABADI M, BARHAM P, CHEN J, et al. TensorFlow: A system for large-scale machine learning[C]//12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16), November 2016, Savannah, GA, USA. Berkeley, CA: USENIX Association, 2016: 265-283.

[10] MITTAL S, VETTER J S. A survey of CPU-GPU heterogeneous computing techniques[J]. ACM Computing Surveys, 2015, 47(4): 1-35.

[11] NVIDIA Corporation. CUDA C++ Programming Guide: Version 13.1.0[R/OL]. (2025-12) [2025-12-10]. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[12] TULLSEN D M, EGGERS S J, LEVY H M. Simultaneous multithreading: Maximizing on-chip parallelism[C]//Proceedings of the 22nd Annual International Symposium on Computer Architecture, June 22-24, 1995, Santa Margherita Ligure, Italy. New York: ACM, 1995: 392-403.

[13] Linux Foundation. perf: Linux performance analysis tool[EB/OL]. (2024-12) [2025-12-10]. https://perf.wiki.kernel.org/.

[14] NVIDIA Corporation. Nsight Systems User Guide: Version 2025.6[R/OL]. (2025-11) [2025-12-10]. https://docs.nvidia.com/nsight-systems/UserGuide/index.html.

[15] PyTorch Team. PyTorch Profiler — PyTorch Tutorials 2.7.0+cu126[EB/OL]. (2025-04-23) [2025-12-10]. https://docs.pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

[16] NVIDIA Corporation. DLProf User Guide [R/OL]. (2024-7) [2025-12-10]. https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/index.html.

[17] KELLEY J E, WALKER M R. Critical-path planning and scheduling[C]//IRE-AIEE-ACM '59 (Eastern). New York: Association for Computing Machinery, 1959: 160-173.

[18] 林一松,杨学军,唐滔,等. 一种基于关键路径分析的CPU-GPU异构系统综合能耗优化方法[J]. 计算机学报,2012,35(1):123-133.
LIN Y S, YANG X J, TANG T, et al. A comprehensive energy consumption optimization method for CPU-GPU heterogeneous systems based on critical

path analysis[J]. Chinese Journal of Computers, 2012, 35(1): 123-133.

[19] META. Holistic trace analysis[EB/OL]. (2025-11) [2025-12-10]. https://hta.readthedocs.io/en/stable/.

[20] CORMEN T H, LEISERSON C E, RIVEST R L, et al. Introduction to algorithms[M]. 3rd ed. Cambridge: MIT Press, 2009.

[21] RAJBHANDARI S, RASLEY J, RUWASE O, et al. Zero: Memory optimizations toward training trillion parameter models [C]//SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, November 9-19, 2020. New York: IEEE, 2020: 1-16.

[22] SHOEYBI M, PATWARY M, PURI R, et al. Megatron-lm: Training multi-billion parameter language models using model parallelism[EB]. arXiv preprint, 2019, arXiv:1909.08053.

[23] HE K, ZHANG X, REN S, et al. Deep residual learning for image recognition [C]//IEEE Conference on Computer Vision and Pattern Recognition, June 26-July 1, 2016, Las Vegas, NV, USA. New York: IEEE, 2016: 770-778.

[24] TOUVRON H, MARTIN L, STONE K, et al. Llama 2: open foundation and fine-tuned chat models[EB]. arXiv preprint, 2023, arXiv:2307.09288.

[25] Intel Corporation. Intel VTune Profiler User Guide[R/OL]. (2025) [2026-02-12]. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/current/overview.html.

[26] NVIDIA Corporation. Nsight Compute Documentation[R/OL]. (2025) [2026-02-12]. https://docs. nvidia. com/nsight-compute/.

[27] NVIDIA Corporation. CUDA Profiling Tools Interface (CUPTI) User Guide[R/OL]. (2025) [2026-02-12]. https://docs.nvidia.com/cupti/.

[28] TensorFlow Team. TensorFlow Profiler: Profile model performance[EB/OL]. (2025) [2026-02-12]. https://www.tensorflow.org/guide/profiler.

[29] GRAHAM S L, KESSLER P B, MCKUSICK M K. Gprof: a call graph execution profiler[C]//Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, June 23-25, 1982, Boston, MA, USA. New York: ACM, 1982: 120-126.

[30] ADHIANTO L, BANERJEE S, FAGAN M, et al. HPCToolkit: tools for performance analysis of optimized parallel programs[J]. Concurrency and Computation: Practice and Experience, 2010, 22(6): 685-701.

[31] SHENDE S S, MALONY A D. The TAU parallel performance system[J]. The International Journal of High Performance Computing Applications, 2006, 20(2): 287-311.

[32] SNIDER D, CHEVALIER F, PEKHIMENKO G. Hotline profiler: automatic annotation and a multi-scale timeline for visualizing time-use in DNN training[C]//Proceedings of Machine Learning and Systems 5. Santa Clara, CA: MLSys, 2023: 104-126.

[33] COVINGTON P, ADAMS J, SARGIN E. Deep neural networks for YouTube recommendations[C]//Proceedings of the 10th ACM Conference on Recommender Systems, September 15-19, 2016, Boston, MA, USA. New York: ACM, 2016: 191-198.

[34] ZHAO Q, WU H, HAO Y, et al. DeepContext: a context-aware, cross-platform, and cross-framework tool for performance profiling and analysis of deep learning workloads[C]//Proceed-

ings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3. New York: ACM, 2025: 48-63.

[35] HOLLINGSWORTH J K. An online computation of critical path profiling[C]// Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, May 23-24, 1996, Philadelphia, PA, USA. New York: ACM, 1996: 11-20.

[36] SCHMITT F, DIETRICH R, JUCKELAND G. Scalable critical-path analysis and optimization guidance for hybrid MPI-CUDA applications[J]. The International Journal of High Performance Computing Applications, 2017, 31(6): 485-498.

[37] ZHAO Y, GU A, VARMA R, et al. PyTorch FSDP: experiences on scaling fully sharded data parallel[EB]. arXiv preprint, 2023, arXiv:2304.11277.

[38] FROSTIG R, JOHNSON M J, LEARY C. Compiling machine learning programs via high-level tracing[C]//1st Conference on Systems and Machine Learning (SysML 2018), February 15-16, 2018, Stanford, CA, USA. Stanford, CA: SysML, 2018.

[39] CHEN T, LI M, LI Y, et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems[EB]. arXiv preprint, 2015, arXiv:1512.01274.

[40] PATARASUK P, YUAN X. Bandwidth optimal all-reduce algorithms for clusters of workstations[J]. Journal of Parallel and Distributed Computing, 2009, 69(2): 117-124.

[41] DETTMERS T, PAGNONI A, HOLTZMAN A, et al. QLoRA: Efficient finetuning of quantized LLMs[C]//Advances in Neural Information Processing Systems 36. Red Hook, NY: Curran Associates, Inc., 2023: 10088-10115.

[42] THAKUR R, RABENSEIFNER R, GROPP W. Optimization of collective communication operations in MPICH[J]. The International Journal of High Performance Computing Applications, 2005, 19(1): 49-66.

[43] Python Software Foundation. PEP 703 - Making the Global Interpreter Lock Optional in CPython[S]. 2023.

[44] LAMPORT L. Time, clocks, and the ordering of events in a distributed system [J]. Communications of the ACM, 1978, 21(7): 558-565.

作者简介

杨桐（2001- ），男，硕士研究生，华东师范大学数据科学与工程学院，CCF学生会员，主要研究方向为大模型训练性能分析、系统优化。

李宁（1999– ），男，博士研究生，华东师范大学数据科学与工程学院，CCF 学生会员，主要研究方向为操作系统同步性能分析与优化。

黄波（1973– ），男，博士，华东师范大学数据科学与工程学院，特聘教授，博士生导师，CCF 专业会员，主要研究方向为数据驱动的软件/服务优化、编译技术。

郭健美（1981– ），男，博士，华东师范大学数据科学与工程学院，教授，博士生导师，CCF 高级会员，主要研究方向为软件系统优化与质量保障。

收稿日期: 2025–12–17
通信作者: 郭健美,jmguo@dase.ecnu.edu.cn
基金项目: 国家自然科学基金项目(No.62272167)
Foundation Items: The National Natural Science Foundation of China(No.62272167)