计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(8): 1962−1978, 2025

面向昇腾处理器的高性能同步原语自动插入方法

李帅江[1,2,3] 张馨元[1,3] 赵家程[1,2,3] 田行辉[4] 石曦予[1,3] 徐晓忻[4] 崔慧敏[1,2,3]

[1](中国科学院计算技术研究所 北京 100190)

[2](中国科学院大学 北京 100190)

[3](处理器芯片全国重点实验室(中国科学院计算技术研究所) 北京 100190)

[4](华为技术有限公司 杭州 310051)

(lishuaijiang19b@ict.ac.cn)

Automatic Insertion Method of High-Performance Synchronization Primitives for Ascend Processors

Li Shuaijiang[1,2,3], Zhang Xinyuan[1,3], Zhao Jiacheng[1,2,3], Tian Xinghui[4], Shi Xiyu[1,3], Xu Xiaoxin[4], and Cui Huimin[1,2,3]

[1](*Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190)

[2](*University of Chinese Academy of Sciences*, *Beijing* 100190)

[3](*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)

[4](*Huawei Technologies Co.*, *Ltd.*, *Hangzhou* 310051)

Abstract Instruction-level parallelism (ILP) is a classic challenge in the field of processor architecture. Domain-specific architectures, such as the Ascend processor, expose more pipeline details to upper-layer software, and compilers/programmers explicitly control the synchronization between pipelines to optimize ILP. However, the physical synchronization resources between pipelines are limited, which limits the improvement of ILP. To address this issue, a high-performance automatic synchronization primitive insertion method for the Ascend processor is proposed. By introducing the abstraction of "virtual synchronization resources", this method decouples the insertion of synchronization primitives from the selection of physical synchronization resources. Firstly, a heuristic algorithm is proposed to insert virtual synchronization primitives in complex control flow graphs. Then, a significant number of virtual synchronization resources are mapped to an extremely limited number of physical synchronization resources through virtual synchronization primitive merging and other techniques. At the same time, redundant synchronization primitives in the program are removed based on the partial order relationship between instructions, while ensuring program correctness and stringent hardware resource constraints. Experiments on the Ascend 910A platform using instruction-level and operator-level benchmark programs show that the programs with automatically inserted synchronization primitives achieve performance comparable to or on par with those manually inserted by expert programmers, while ensuring correctness.

Key words Ascend processor; synchronization primitives; heterogeneous programming; domain specific architecture; automatic insertion

收稿日期: 2024-02-03; **修回日期**: 2025-01-20

基金项目: 国家重点研发计划项目(2022ZD0116316); 国家自然科学基金重点项目(62232015); 中国科学院计算技术研究所创新课题(E361010, E261110)

This work was supported by the National Key Research and Development Program of China (2022ZD0116316), the Key Program of the National Natural Science Foundation of China (62232015), and the Innovation Funding of CAS Institute of Computing Technology (E361010, E261110).

通信作者: 赵家程(zhaojiacheng@ict.ac.cn)

摘　要　指令级并行（instruction level parallism, ILP）是处理器体系结构研究的经典难题. 以昇腾为代表的领域定制架构将更多的流水线细节暴露给上层软件，由编译器/程序员显式控制流水线之间的同步来优化ILP，但是流水线之间的物理同步资源是有限的，限制了ILP的提升. 针对这一问题，提出一种面向昇腾处理器的高性能同步原语自动插入方法，通过引入“虚拟同步资源”的抽象将同步原语的插入和物理同步资源的选择进行解耦. 首先提出了一种启发式算法在复杂的控制流图上进行虚拟同步原语的插入，随后通过虚拟同步原语合并等技术，将虚拟同步资源映射到有限数量的物理同步资源上，并同时在满足程序正确性与严苛硬件资源限制的前提下，根据指令间的偏序关系删除程序中冗余的同步原语. 使用指令级与算子级基准测试程序在昇腾910A平台上的实验表明，该方法自动插入同步原语的程序在保证正确性的基础上，整体性能与专家程序员手动插入同步原语接近或持平.

关键词　昇腾处理器；同步原语；异构编程；领域定制架构；自动插入

中图法分类号　TP314

DOI: 10.7544/issn1000-1239.202440093　　**CSTR:** 32373.14.issn1000-1239.202440093

对指令级并行（instruction level parallism, ILP）的追求是体系结构软硬件研究的永恒问题. 通用CPU由微架构承担了挖掘ILP的核心职责. 具体来说，通用CPU的微架构设计了众多复杂的控制逻辑，通过乱序执行[1]、分支预测[2]、计分板动态调度[3]、寄存器重命名[4]等技术在充分挖掘ILP的基础上保证程序的执行不会违反指令间的数据/控制依赖，从而在提升性能的同时保证程序的正确性.

但是随着以深度学习为代表的人工智能（AI）技术在图像处理[5−6]、自然语言处理[7−8]、蛋白质结构预测[9−10]等技术领域的成功，AI应用开始爆炸式发展，同时驱动芯片进步的摩尔定律的终结，使得传统的CPU愈发难以满足各类AI应用不断增大的算力需求[11]，由此催生了各类领域定制架构用于加速AI计算，如通用GPU（general purpose GPU, GPGPU）（如基于SIMT架构的英伟达GPU[12]、AMD GPU[13]和海光深度计算处理器（deep computing unit, DCU）[14]）以及专用的神经网络处理器（neural network processing unit, NPU）（如寒武纪的MLU（machine learning unit）[15]和华为昇腾[16]）等.

与CPU不同，领域定制架构为了利用有限的芯片面积最大化算力，将指令级并行挖掘的职责留给软件来负责. 具体来说，和传统的通用处理器类似，领域定制架构的单核上通常集成了多种不同功能的流水线单元，如直接内存访问（direct memory access, DMA）流水线、张量计算流水线、向量计算流水线等，但是和传统的通用处理器不同，领域定制架构不同的流水线单元有相对独立的指令队列，队列之间或流水线之间微架构不提供执行序的保证，而在传统的通用处理器当中，微架构则会通过计分板动态调度等技术保证流水线之间的数据/控制依赖的正确性. 为此，在领域定制架构当中，微架构通常会暴露更多的流水线间同步接口，由上层软件（编译器或者最终开发者）直接对流水线间的执行顺序进行编程，解决必要的数据/控制依赖，从而通过软硬件协同的方式兼顾完备性和效率，但是也大幅提升了软件的负担.

以昇腾系列处理器为例，其内部有6个独立的执行单元：标量计算单元（scalar unit）、向量计算单元（vector unit）、矩阵计算单元（cube unit）以及3个数据搬运单元，分别完成其对应的计算、存储等功能，每个执行单元有一条独立的指令队列. 该6个执行单元之间是无序的，需要程序员或编译器手动插入显式同步原语来确保不同执行单元之间按照依赖关系来执行[17−18].

尽管当前的GPGPU如英伟达GPU和AMD GPU也面临相同的问题，但是微架构暴露的流水线同步接口具有巨大的差异，从而使得这些技术难以直接适用于昇腾架构. 具体来说，英伟达的GPGPU微架构提供了一种依赖寄存器形式的流水线同步方法，由PTXAS编译器在生成体系结构相关的汇编过程当中根据指令依赖关系为每条指令设置对应的依赖寄存器，来进行依赖关系的管理. 而AMD GPU则提供了一种计数器形式的同步原语. 而正如上文所言，昇腾架构的底层同步原语与通用GPGPU存在着巨大的差异，昇腾架构提供了一种基于set-wait（也称post-wait）抽象的流水线间同步方法，硬件提供set/wait指令，可以指定一对执行单元之间形成“锁”的关系，以此来进行不同执行单元间的同步.

当前，英伟达与AMD GPU已实现编译器自动插

入同步原语, 与处理器协同调度来维护指令间的数据依赖. 而昇腾处理器采用的 set-wait 同步原语在硬件上的自动插入问题仍未解决. 因此, 程序员在昇腾处理器上进行算子编程时需要手动插入同步原语[17−18], 更容易出现错误. 然而手动插入同步原语需要编程人员对底层硬件有足够深入的了解, 才能保证程序的正确性与性能, 大大增加了编程难度和工作量, 因为高性能算子库为了获得较好的性能, 往往需要数月的时间来对每个算子进行专门调优, 阻碍了新算法的发展[19]. 因此, 如何在昇腾处理器上编译器自动插入同步原语, 以实现和处理器的协同调度来保证指令间数据依赖, 成为昇腾处理器算子编程亟待解决的问题.

通过深入研究发现, 当前在昇腾处理器上实现同步原语的自动插入需要面临 3 个挑战:

挑战 1. 复杂的程序控制流图与执行路径. 同步语句 set-wait 需要根据指令间的数据依赖严格地一一对应, 但是复杂的控制流图与执行路径使得为依赖的指令插入严格对应的 set-wait 变得极为困难, 即既要保证依赖指令间的数据依赖得到满足, 又要求 set-wait 同步指令一一配对.

挑战 2. 严苛的同步资源限制. 在昇腾架构中, set-wait 同步原语需要置位/释放指定的硬件资源(信号量/标志位), 而这些硬件物理资源的数量是极为有限的. 这一问题对于昇腾架构异常突出, AMD GPU 采用的计数器同步原语不限制同步的资源数量, 英伟达 GPU 采用依赖寄存器同步原语, 每条指令都可以设置或等待一定数量的依赖寄存器, 限制较为宽松. 而在现实昇腾应用中, 同时活跃的需要编译器维护的依赖关系可高达数十个, 软件依赖与硬件同步资源二者间数量级的巨大差距给自动的同步方法带来了严苛的挑战.

挑战 3. 如何在满足上述限制的同时尽可能提升性能. 同步原语的自动插入不仅要保证正确性, 还应该尽可能提升硬件利用效率. 而冗余的同步原语会导致过多非必要的阻塞, 降低硬件单元的执行效率. 因此需要在满足硬件资源限制与程序正确性的同时, 尽可能去除冗余同步原语, 以提升性能.

为了解决上述挑战, 本文提出了一种面向昇腾处理器的高性能同步原语自动插入方法. 我们提出“虚拟同步资源”的概念, 将昇腾处理器上的同步原语插入问题解耦为 3 部分: 1)虚拟同步原语插入; 2)冗余虚拟同步原语删除; 3)物理同步资源分配. 我们基于 LLVM 实现了该方法, 并在昇腾 910A 处理器上进行了实验, 实验表明, 该自动插入方法在保证正确性的基础上, 整体性能与专家程序员手动插入同步原语接近或持平.

本文的主要贡献包括 4 个方面:

1)提出“虚拟同步资源”概念, 并基于该概念实现了为异构处理器上的程序进行虚拟同步原语插入, 解决了复杂的控制流图和执行路径所带来的一系列问题;

2)通过同步原语合并等技术, 解决了大量虚拟同步资源到极有限数量的物理同步资源的映射问题;

3)在满足程序正确性与严苛硬件资源限制的同时, 根据指令间的偏序关系对程序中冗余的同步原语进行删除, 进一步提升了硬件的利用效率;

4)基于 LLVM 实现了该方法, 并在昇腾 910A 平台上进行了实验, 实现了昇腾平台上同步原语的自动化插入.

1 背景及相关工作

1.1 昇腾处理器架构

昇腾处理器是华为公司推出的面向 AI 场景的领域定制架构处理器, 其由 AI CPU 与 AI Core 共同组成片上系统(system of chip, SOC), 其中 AI Core 负责神经网络运算, 提供绝大部分算力. 通过配置不同产品中 AI Core 的数量(1~32), 可以满足云—边—端不同场景对 AI 算力的需求. 例如, 昇腾 910 处理器包含 32 个 AI Core, FP16 算力达 256 TFLOPS, INT8 算力为 512 TOPS; 配备 32 GB HBM(high bandwidth memory), 带宽达 1 200 GBps. 其 SOC 架构如图 1 所示.

AI Core 内部结构如图 2 所示. 在存储方面, AI Core 内部由 L1(L1 buffer)、L0A(L0A buffer)、L0B(L0B buffer)、L0C(L0C buffer)、UB(unified buffer)等多个不同功能和层次的存储单元组成. L1 大小为 1 MB, 为一块较大的数据中转区, 可暂存一些反复使用的数据, 减少访问片外内存的次数, 要求 512 B 对齐; L0A 与 L0B 大小均为 64 KB, 分别存放矩阵计算单元的输入左矩阵与右矩阵, 要求 512 B 对齐; L0C 大小为 256 KB, 用于存放矩阵计算单元的输出数据, 进行累加乘计算时, 也作为输入数据的一部分, 要求 512 B 对齐; UB 大小为 256 KB, 为统一缓冲区, 存放向量与标量计算的输入与输出数据, 要求 32 B 对齐[18].

在计算方面, AI Core 包含矩阵计算单元、向量计算单元、标量计算单元 3 部分. 矩阵计算单元负责矩阵乘计算, 其每次执行完成 1 个 FP16 精度的 16×16

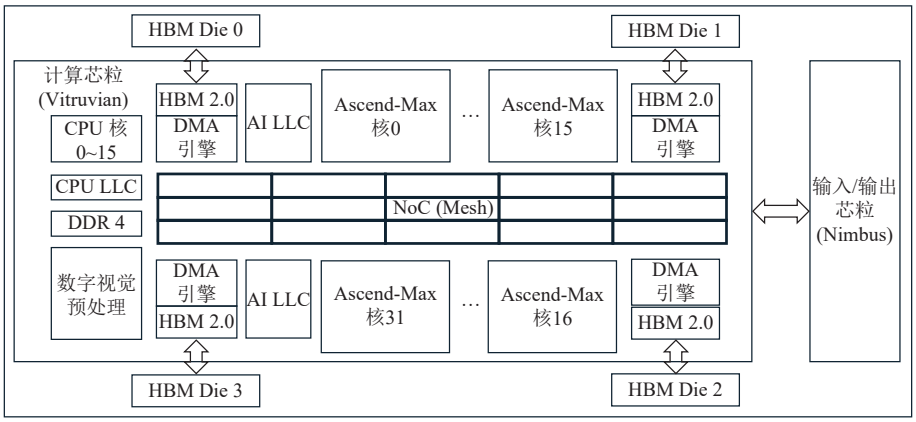


Fig. 1 Ascend 910 architecture[16]

图 1 昇腾 910 架构[16]

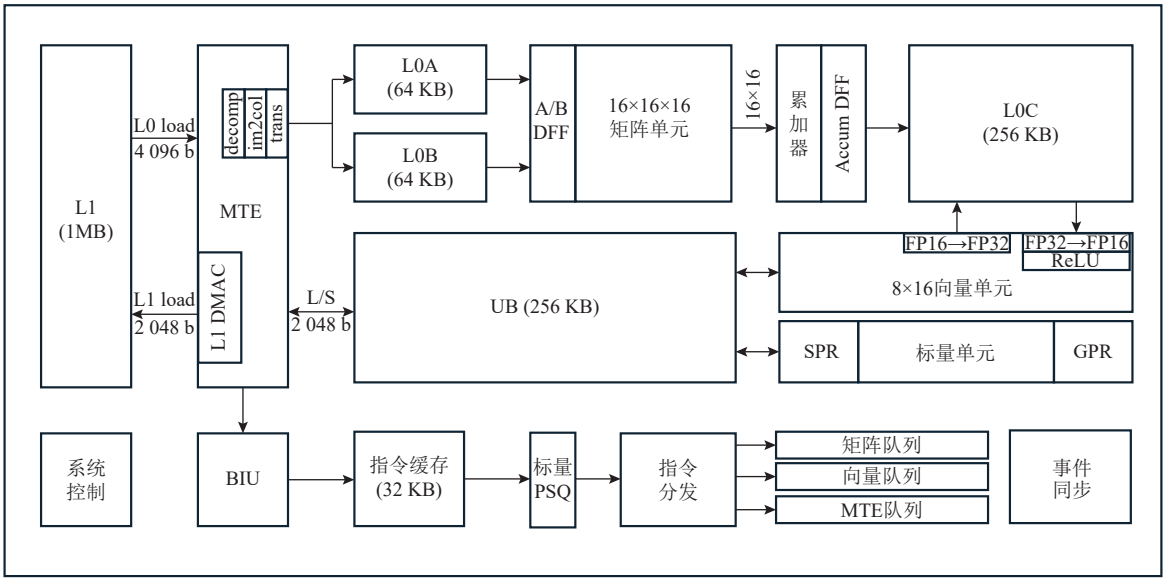


Fig. 2 AI Core architecture[16]

图 2 AI Core 架构[16]

与 16×16 的矩阵乘；向量计算单元负责激活函数等非矩阵乘计算. 标量计算单元负责循环控制、分支判断、地址计算以及基本的算术运算等功能. 其中，矩阵计算单元算力最强，提供了 AI Core 的绝大部分算力，但也最不灵活，只能进行矩阵乘计算. 标量计算单元算力最弱，但其计算最灵活. 向量计算单元则介于二者之间[18].

计算与存储单元的联系表现为矩阵计算单元的左操作数来自 L0A，右操作数来自 L0B，矩阵乘结果与中间结果放在 L0C；向量计算单元所有源数据与计算结果都要求位于 UB 中；标量计算单元的操作数可以存放在寄存器与 UB 中. 由于复杂的对应关系，AI Core 内部有 3 个专门的存储转换引擎（memory transfer engine，MTE）单元进行数据传输与搬运. MTE1 负责从 L1 传输数据到 L0A/L0B/UB，或者用 SPR 初始化 L0A/L0B；MTE2 负责从 L2/HBM/DDR 传输数据到 L1/L0A/L0B/UB；MTE3 负责从 UB 传输数据到 L2/HBM/DDR [18].

3 个计算单元与 3 个 MTE 单元一起，共 6 个独立的执行单元，每个单元对应一条指令队列，共 6 条独立的指令队列，如表 1 所示. 同一指令队列内部顺序执行，不同指令队列之间乱序执行. 硬件不保证不同队列间的同步与数据依赖关系，仅提供同步原语. 因此，需要程序员或编译器向程序中插入同步指令来确保不同队列间的数据依赖关系.

如图 3 所示，昇腾处理器采用 set-wait 形式的同步原语. 硬件提供 set/wait 指令，可以指定一对执行单元之间形成“锁”的关系，即当 set 指令之前的所有读写操作都完成之后，set 指令执行，将硬件中对应的标志位设置为 1；另一个执行单元执行到对应的 wait 指

Table 1 Six Instruction Queues[18]

表 1 6 个指令队列[18]

队列缩写	含义	队列内指令功能
S	标量指令队列	标量计算
V	向量指令队列	向量计算
M	矩阵指令队列	矩阵乘计算
MTE1	MTE1 指令队列	从 L1 到 L0A/L0B/UB，或者用 SPR 初始化 L0A/L0B
MTE2	MTE2 指令队列	从 L2/HBM/DDR 到 L1/L0A/L0B/UB
MTE3	MTE3 指令队列	从 UB 到 L2/HBM/DDR

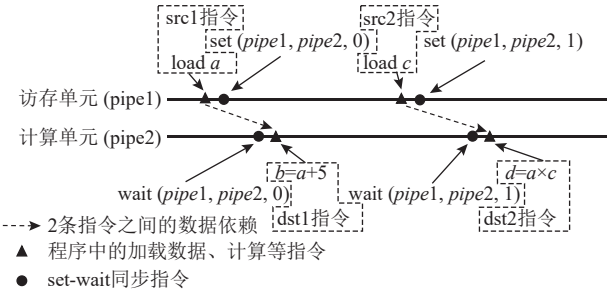


Fig. 3 An example of Ascend processor synchronization primitives

图 3 昇腾处理器同步原语示例

令时, 如果发现对应标志位为 0, 该执行单元的后续指令将一直被阻塞; 如果发现对应标志位为 1, 则将对应标志位设置为 0, 同时开始执行后续指令.

为便于陈述, 下文将标志位称为 ID, “物理 ID”即表示“物理标志位”.

如图 3 所示, src1 指令与 dst1 指令之间有依赖关系(虚线箭头所示), a, b, c, d 为指令中的变量, 必须等 load a 执行结束, $b = a + 5$ 才能开始执行. src1 指令由访存单元(假设为 pipe1)执行, dst1 指令由计算单元(假设为 pipe2)执行, 因此需要在 src1 指令后插入 set(pipe1, pipe2, 0)指令, 在 dst1 前插入 wait(pipe1, pipe2, 0)指令, 表示使用第 0 个标志位. 下一对 src2 与 dst2 指令就使用第 1 个标志位, 以此类推.

多样的计算单元、复杂的多级内存层次和数据传输单元, 加剧了同步原语自动插入的困难性. 在算子编程方面, 昇腾提供了多种方式.

TBE(tensor boost engine)、TIK(tensor iterator kernel)和 TBE DSL(TBE domain-specific language)调用封装好的 API 接口编程, 以避免处理同步原语, 但其调用封装接口进行编程的形式层次相对较高, 限制了算子编程的灵活性, 不利于昇腾芯片极致性能的挖掘. CCE(cube-based computing engine)是华为公司推出的类 C 编程语言, 相比 TBE TIK 与 TBE DSL 更接近底层, 但其需要程序员手动插入同步原语[17]; Ascend C 是昇腾 2023 年新发布的算子编程语言, 将向量算子计算过程分为 CopyIn, Compute, CopyOut 这 3 个步骤, 矩阵算子计算过程分为 CopyIn, Split, Compute, Aggregate, CopyOut 这 5 个过程[18], 其本质上依然是通过程序员手动划分计算过程, 来控制数据传输、数据计算等执行单元间的同步, 算子编程的复杂度依然较大.

关于 2 个执行单元间物理 ID 的数量, 华为公司没有公开资料说明, 但通过分析华为 Ascend C 编程语言对各种算子生成的.cce 文件可知, .cce 文件中同步语句使用到的物理 ID 范围均在 0~3 之间, 因此可判断, 昇腾处理器硬件上任意 2 个执行单元间物理 ID 数量为 4(如果物理 ID 数量大于 4, Ascend C 生成的.cce 文件就会使用更多的物理 ID, 而不是只使用 ID0~ID3 这 4 个物理 ID).

因此, 由于复杂的控制流图以及执行路径、严苛的硬件资源限制对性能的要求, 昇腾处理器的同步原语自动插入问题目前仍未解决, 增大了算子编程的难度和工作量.

1.2 同步原语自动插入问题

CPU 上多线程程序间显式同步原语的插入是一个经典问题, 之前已经有较多研究. 但据我们所知, 目前尚无面向异构处理器的 post-wait 硬件同步原语自动插入研究.

Nicolau 等人[20]探索了 CPU 上多线程程序之间的 post-wait 同步原语放置, 通过代码移动等技术来优化 post 和 wait 同步原语的放置, 增强具有迭代依赖关系的循环的线程级并行性. Zhai 等人[21]使用基于 CFG 的分析来确定在哪里适当地放置 wait 和 signal 同步语句.

在 CPU 多线程间同步原语消除上, Nicolau 等人[22]提出根据迭代之间的依赖关系, 消除 CPU 上多线程间同步的冗余 wait 语句. Tseng[23]提出了消除屏障(barrier)同步的方法. Chen 等人[24]提出了消除 DOACROSS 循环中冗余同步的技术. 随后, Aldrich 等人[25]、Bogda 等人[26]提出了在 Java 中消除不必要同步的技术. Han 等人[27]也提出了一种新的解决方案, 使用 fork-join 模型来减少多线程系统中所需的同步总数. 而本文是根据指令间的偏序关系同时消除冗余的硬件 post 与 wait 指令.

作为当前的主流 AI 加速器之一, GPU 上同步语句的插入已经有较多研究. Li 等人[28]在 GPU 上构建了生产者-消费者(即 post-wait)同步原语, 用于线程

块中协作线程之间的细粒度数据同步；Liu 等人[29]通过数据依赖分析来为 GPU 程序插入同步语句，并消除冗余同步语句. Moses 等人[30]通过分析同步语句前后指令的内存行为，判断该同步语句是否必要，实现同步语句的移动与冗余消除；Sorensen 等人[31]实现了跨工作组(work-group)而不仅仅是在 1 个工作组内的 GPU 同步指令. 然而，文献 [28−31] 工作是针对 GPU 的，由于硬件架构差异较大、硬件同步模型不同，所以上述工作的方法并不适用于昇腾处理器.

综上所述，先前的研究大多关注于 CPU 上多线程之间，或 GPU 上不同线程之间的同步问题，昇腾处理器由于其硬件架构、并行模型等方面与传统 CPU 及 GPU 有巨大差异，面临 3 点挑战：1)苛刻的硬件资源限制；2)复杂的程序控制流图与执行路径；3)满足前述限制的同时尽可能提升性能，因此现有方法无法解决昇腾处理器中 post-wait 原语的自动插入问题.

2 面向昇腾处理器的同步原语自动插入方法

本文提出的同步原语高效自动插入方法探索并实现了昇腾处理器上硬件同步原语的自动插入，以降低编程人员开发负担，同时通过优化技术尽可能减少硬件单元阻塞，充分发挥硬件的算力. 本节首先介绍了方法的整体结构. 然后分别介绍方法步骤，包括虚拟同步原语插入、冗余同步原语删除、物理同步资源分配.

2.1 概 述

同步原语自动插入方法由 3 个阶段构成，如图 4 所示. 该方法的输入是未插入同步原语的中间表示程序. 同时，为了解耦问题的复杂度，我们在处理过程中引入了虚拟硬件资源的概念，在插入过程中可以使用无限多个虚拟硬件资源，对应地，使用“虚拟硬件资源”的同步原语称为“虚拟同步原语”. 在插入同步原语之前，首先会根据数据流方程，对输入程序进行依赖分析，得到程序中指令间存在的读后写(write after read, WAR)、写后读(read after write, RAW)以及写后写(write after write, WAW)三种数据的依赖关系.

得到数据依赖关系后，第 1 阶段需要解决插入的正确性问题. 首先对程序控制流图进行兄弟结点插入、循环规范化等处理，使程序结构满足后续处理过程的要求；然后向程序中初步插入虚拟同步原语；最后处理程序中的分支与循环结构，解决控制流图和

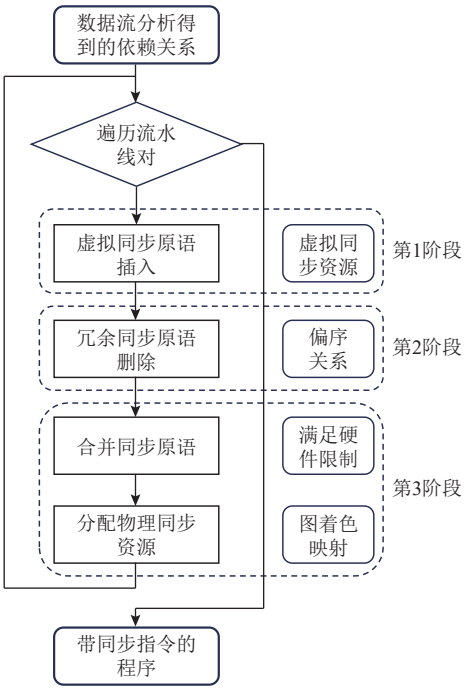


Fig. 4 Overview of automatic insertion process

图 4 自动插入过程总览

执行路径带来的复杂性，满足程序的正确性. 正确性包括：1)post 与 wait 原语两两配对；2)程序指令间的依赖关系得到保证.

插入虚拟同步原语后，为了在确保程序正确性的基础上尽可能优化性能，第 2 阶段根据指令间的偏序关系，去除第 1 阶段插入的冗余同步原语.

第 3 阶段负责将虚拟 ID 映射到硬件资源所支持的有限数量的物理 ID 上，以得到最终的结果. 与虚拟寄存器到物理寄存器的映射不同，在物理寄存器容纳不下虚拟寄存器时，可以选择将一些物理寄存器中的值溢出(spill)到内存(或缓存)中. 在本文中，虚拟 ID 映射到物理 ID 时，如果同时活跃的虚拟 ID 数大于物理 ID 数，无法像物理寄存器那样溢出. 所以第 3 阶段首先对程序中的虚拟 ID 进行合并处理，使得在任意程序点，同时活跃的虚拟 ID 数目都不超过硬件能提供的物理 ID 数量. 然后使用图着色算法将虚拟 ID 分配到物理 ID，完成整体流程.

由于数据流分析算法已经有许多比较成熟的研究，因此使用数据流分析得到依赖关系的过程不在本文讨论范围.

2.2 虚拟同步原语插入

为了将控制流图执行路径的复杂性与物理资源的限制这 2 个问题尽可能解耦，降低问题复杂度，我们借鉴了编译优化中寄存器分配过程的“虚拟寄存器”概念，引入了“虚拟 ID(虚拟硬件资源)”. 与物理

硬件资源的不同在于, 虚拟硬件资源有无限多个, 没有物理硬件资源那样严苛的限制, 所以在插入同步原语时不用考虑硬件资源限制问题.

2.2.1 规范化控制流图

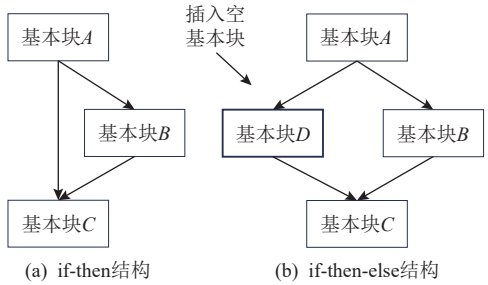
在插入同步原语之前, 首先需要处理程序的分支和循环等结构, 插入一些空的基本块(basic block, BB), 使程序控制流图变成“规范化”形式:

1)所有分支结构都有 then 和 else 这 2 个分支, 即为 if-then-else 的形式.

2)所有循环结构都满足规范形式, 即有 1 个前置头结点(preheader); 只有 1 个回边(backedge); 循环只有 1 个出口基本块(exit basic block), 且其被循环头结点(loop header)支配.

传统控制流图规范化着重于控制流图的特例——循环结构的处理, 如 LLVM 中的 loop-simplify pass 把不规整的循环规范化为标准的循环简化形式(loop simplify form)[32]; 而本文的方法关注更加通用的控制流: 一方面, 像 LLVM 一样, 对循环进行处理; 另一方面, 需要把 if-else 结构规范化为 if-else-then 结构, 以保证程序在任意执行路径上, set 与 wait 语句都能一一匹配.

如图 5 所示, 通过规范化处理 if-then 形式的分支结构变为 if-then-else 分支结构.



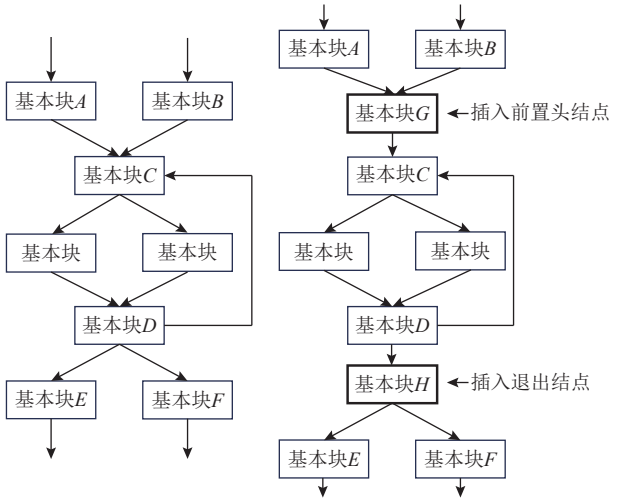
(a) if-then结构 (b) if-then-else结构

Fig. 5 An example of normalization of branch structure

图 5 分支结构规范化示例

图 6(a)所示为处理前的循环结构, 通过规范化处理之后, 变为图 6(b)所示的处理后的循环结构, 经过处理后加入了前置头结点(基本块 G), 插入了 1 个退出结点(基本块 H), 使得循环只有 1 个退出基本块.

规范化循环结构主要包括为循环结构添加的前置结点和出口基本块. 规范化分支结构是指为分支结构中的一个基本块添加兄弟结点. 兄弟结点是指没有前驱和后继关系的多个基本块, 并且在控制流图中具有相同支配结点和后支配结点的基本块. 对控制流图进行规范化的过程如图 7 所示.



(a) 处理前的循环结构 (b) 处理后的循环结构

Fig. 6 An example of normalization of loop structure

图 6 循环结构规范化示例

2.2.2 初步插入同步原语

根据数据流分析得到的结果, 可得到程序中所有的依赖指令对〈src, dst〉, 其中 dst 指令依赖于 src 指令, 即必须先执行完 src 指令, dst 指令才能执行.

遍历〈src, dst〉指令对, 依次对每对指令初步插入虚拟同步指令. 该步骤插入的同步指令使用虚拟 ID, 按照遍历的顺序, 对应流水线的虚拟 ID 从 0 开始递增使用.

1)若 src 指令和 dst 指令不在循环内, 则直接在 src 指令之后插入 set 指令, 在 dst 指令之前插入 wait 指令. 如图 8 所示, set 和 wait 为初步插入的同步原语.

2)若 src 指令或 dst 指令至少有 1 个在循环内, 则需找到要插入 set 和 wait 的最外层循环, 根据二者最外层循环之间的关系, 来判断 set 与 wait 的插入位置.

如图 9 所示为一个控制流图示例. src 指令在循环 L0 内, 且 dst 指令不在 src 指令所在的最外层循环 L 中(如: dst 指令不在循环内, 或 dst 指令在循环 L1 内, 但循环 L1 不在循环 L 内), 可以在最外层循环 L 的出口基本块中插入 src 指令对应的 set 指令.

外提 set 的原因在于: 如果紧跟在 src 指令后插入 set 指令, 则程序在每次执行循环 L0 或者最外层循环 L 时, 都会执行 1 次该处的 set 指令, 将该指令中 ID 标志位进行置位, 这种连续多次置位同一个标志位的行为会导致硬件出错, 因此需要将该 set 指令放到最外层循环 L 外, 例如在最外层循环 L 的出口基本块中插入该 set 指令.

同样地, 在与上述情况对应的情况下, wait 需要外提到外层循环的前置头结点(即 preheader)中, 以

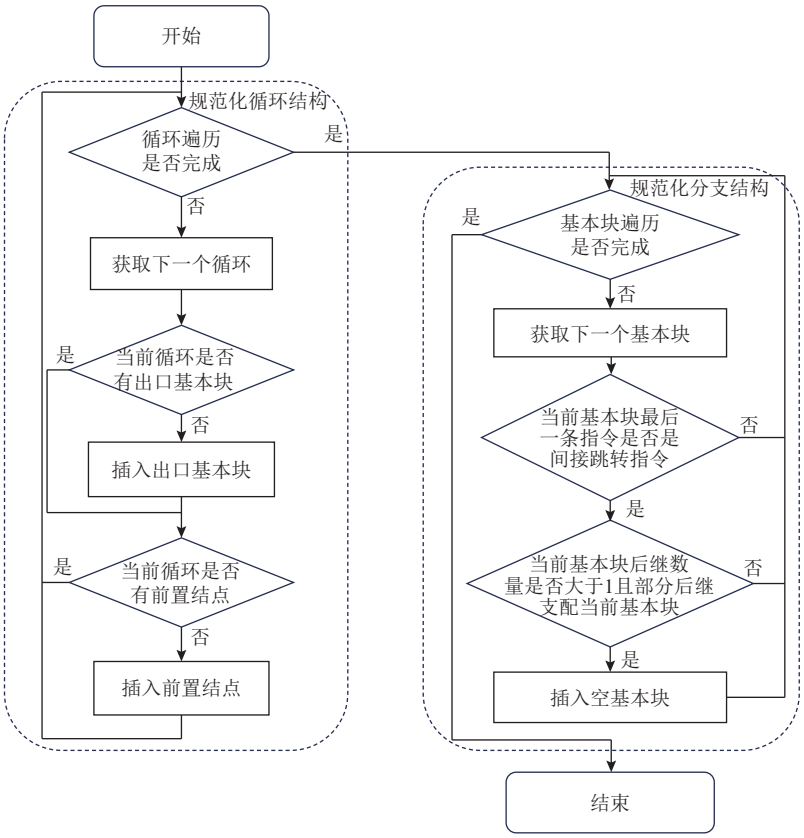


Fig. 7 The process of normalizing the control flow graph

图 7 规范化控制流图过程

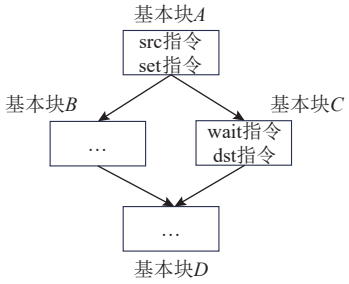


Fig. 8 An example of preliminary insertion of synchronization primitives

图 8 初步插入同步原语示例

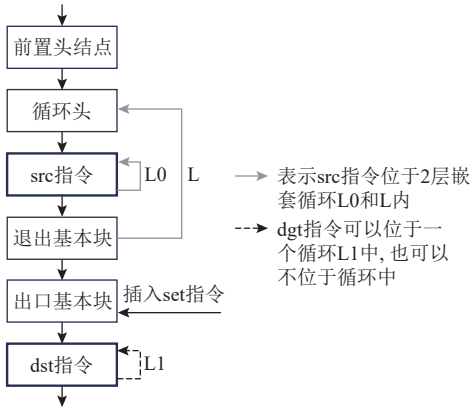


Fig. 9 An example of src/dst instructions in a loop

图 9 src/dst 指令在循环中示例

避免硬件报错.

2.2.3 插入额外同步原语

在根据上述方法为依赖指令对初步插入虚拟同步原语之后，还需要对部分指令对插入额外的虚拟同步原语，以确保在程序执行的任何路径上，set 指令与 wait 指令都可以两两对应，从而保证程序正常执行. 而由于控制流图和执行路径的复杂性，额外虚拟同步原语的插入是自动插入同步原语过程中面临的最大挑战之一.

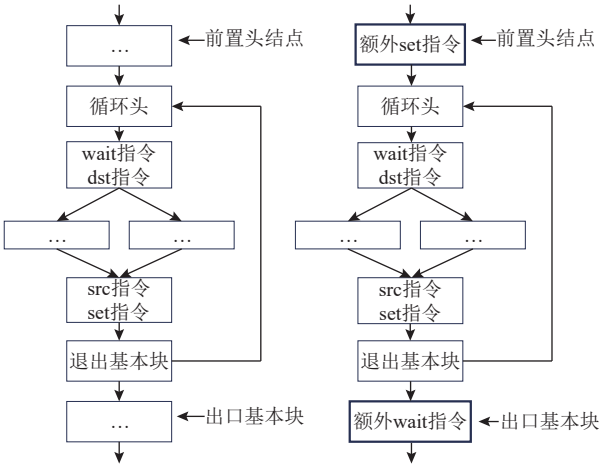
如图 8 所示，在初步插入同步原语后，若程序执行路径为 $A\rightarrow C\rightarrow D$，则路径上的 set 指令与 wait 指令相互配对；若执行路径为 $A\rightarrow B\rightarrow D$，则该路径上就只有 set 指令，没有与之配对的 wait 指令，导致硬件错误. 因此，应该在基本块 B 中插入一个额外的 wait 指令，以保证配对.

值得注意的是，额外虚拟同步原语与前述为〈src, dst〉指令对所初步插入的虚拟同步原语所使用的虚拟 ID 相同. 具体地，可以分 3 种情况为指令对插入额外的虚拟同步原语.

1）src 指令和 dst 指令之间具有跨迭代依赖关系

跨迭代依赖关系是指, 在执行第 $i+1$ 次循环迭代的过程中, dst 指令的执行依赖于之前迭代(如第 i 次、第 $i-1$ 次等)的 src 指令的结果. 其中, i 为大于或等于 1 的正整数.

如图 10(a)所示, 该情况下若不插入额外虚拟同步原语, 在第 1 次执行该循环的过程中执行到 wait 指令时, 由于在之前未执行 set 指令, wait 指令中虚拟 ID 对应的标志位没有被置位, 会导致程序一直等待而无法继续执行. 因此需要在该循环的前置头结点中插入一条 set 指令.



(a) 插额外指令前 (b) 插额外指令后

Fig. 10 Cross-iteration dependency relationship exists between src and dst instructions

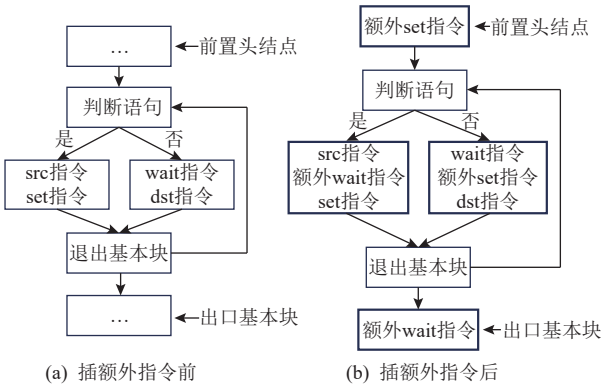
图 10 src 指令与 dst 指令之间存在跨迭代依赖关系

如果仅在循环体中插入上述 set 指令, 则在执行最后一次循环的过程中, 执行完循环内的 set 指令之后, 会将对应的标志位置位, 此时程序跳出循环, 该标志位没有被释放, 会一直处于置位状态, 导致程序执行出错. 因此需要在该循环的出口基本块再插入 1 条 wait 指令, 如图 10(b)所示.

2)src 指令和 dst 指令之间互不可达

src 指令和 dst 指令之间互不可达是指不考虑循环多次迭代的情况下, src 指令到达不了 dst 指令, 且 dst 指令到达不了 src 指令. 例如 src 指令和 dst 指令位于判断语句之后的不同分支中, 在执行一次循环的过程中只会执行 src 指令或 dst 指令中的 1 个.

如图 11(a)所示, 如果不插入额外的虚拟同步原语, 当相邻 2 次循环都执行同一分支时, 就会执行连续的 set 指令或连续的 wait 指令. 因此, 如果 src 指令和 dst 指令互不可达, 则在紧邻着 set 指令之前再插入 1 条 wait 指令并在紧邻着 wait 指令之后再插入 1 条 set 指令. 如图 11(b)所示.



(a) 插额外指令前 (b) 插额外指令后

Fig. 11 An example of mutually unreachable src and dst instructions

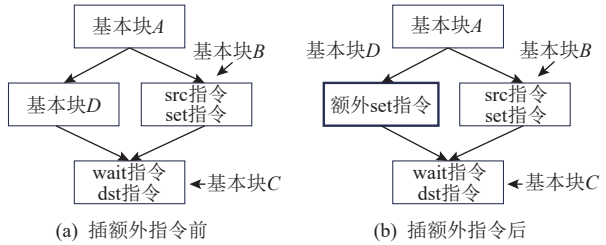
图 11 src 与 dst 指令互不可达示例

另外, src 指令和 dst 指令也是跨迭代依赖关系, 会在其所在的循环的前置头结点中插入 1 条 set 指令, 在循环的出口基本块中插入 1 条 wait 指令. 如果没有此步骤, 在进入循环时, 如果第 1 次迭代执行左侧分支时就会出现连续执行 set 指令的情况以及在最后 1 次循环退出前执行右侧的分支时就会出现连续执行 wait 指令的情况.

3)其他情况处理

在执行完前述 2 种情况后, 该阶段会处理其他的由控制流结构引入的同步原语不配对.

如图 12(a)所示, 该步骤遍历各个基本块, 对于访问的当前基本块 C, 需要根据 C 的各个前驱中的同步原语序列长度和当前基本块的入口状态来确定是否需要在各个前驱中插入额外的同步原语, 以及插入 set 指令还是 wait 指令.



(a) 插额外指令前 (b) 插额外指令后

Fig. 12 An example of unpaired synchronization primitives introduced by control flow structures

图 12 由控制流结构引入的同步原语不配对示例

其中, 某个前驱基本块的序列长度是指, 从程序入口处到该前驱基本块的路径上, 所有的 set 原语和 wait 等待指令的数量; 当前基本块的入口状态是指, 当前基本块的所有前驱中序列长度最大的值.

对于当前基本块 C, 共有 2 个前驱: B 和 D, 首先获取其各个前驱(即 B 和 D)的序列长度, 并据此得

到 C 的入口状态. 然后对于前驱 B, 其序列长度对 2 取余的余数 mod2(B)与 C 的入口状态对 2 取余的余数 mod2(C)相等, 则确定不需要在 B 中插入额外的同步原语.

对于前驱 D, mod2(D)与 mod2(C)不相等, 确定需要在 D 中插入 set 指令或 wait 指令. 在该情况下, 若 mod2(C)≠0, 则需要在 D 中插入一条额外 set 指令; 若 mod2(C)=0, 则在 D 中插入一条额外 wait 指令. 具体地, 在图 12 的情况下, 需要在 D 中插入一条额外 set 指令.

之所以选择对 2 取余, 是因为只有 2 种同步语句 set 与 wait, 二者是相互匹配的(可以理解为 set 与 wait 两两抵消), 所以这里选择对 2 取余进行比较, 而不是比较整个同步原语序列的长度.

2.3 冗余同步原语删除

虽然上述同步原语插入方法相比朴素的插入方法已经尽可能减少了执行单元的阻塞, 但依然可能存在过多的或冗余的同步原语, 导致过多的阻塞, 降低硬件的利用率, 导致程序性能下降. 因此, 在保证正确性的基础上, 本阶段根据指令间的偏序关系删除冗余的 post-wait 同步原语, 以进一步提升处理器利用率.

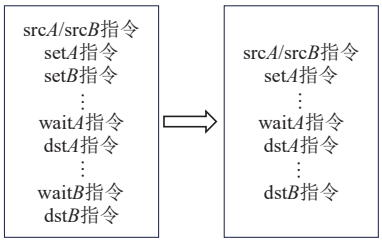
对某一流水线对中插入的虚拟同步原语进行去冗余时, 需要遍历所有的虚拟 ID 组合, 根据不同虚拟 ID 对应的虚拟同步原语之间的偏序关系进行去冗余. 其中, 如果该流水线对中插入了 n 个虚拟同步原语, 即, 虚拟 ID 范围是0～(n−1), 则虚拟 ID 的组合的数量是 $n(n-1)$ 个. 例如, 如果该流水线对中插入了 3 个虚拟 ID, 分别是 0, 1, 2, 则虚拟 ID 的所有(ID_A, ID_B)组合数量为 3×(3−1)=6 个, 分别为(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1). 即, 需要遍历这些组合来判断某对 set-wait 原语是否冗余.

对于任意一组虚拟 ID 的组合(ID_A, ID_B), 其对应的同步指令分别为 set(pipe1, pipe2, A), wait(pipe1, pipe2, A); set(pipe1, pipe2, B), wait(pipe1, pipe2, B)(以下简称 set_A, $wait_A$, set_B, $wait_B$); 对应的依赖指令对分别为: $\langle src_A, dst_A\rangle$, $\langle src_B, dst_B\rangle$.

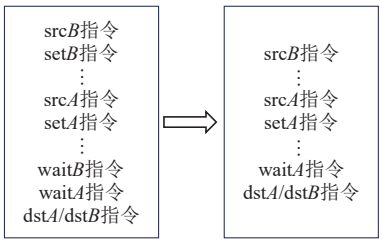
有如下 3 种情况进行去冗余, 分别如图 13(a)～(c)所示(偏序关系在这里指指令之间的支配与后支配关系, 如下文所述).

1) src_A 与 src_B 相同

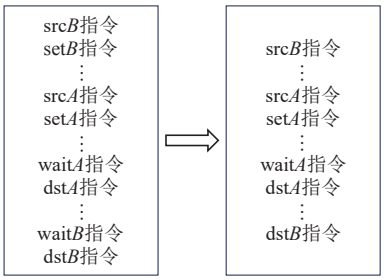
此时, 判断 $wait_A$ 与 $wait_B$ 之间是否存在支配关系. 所谓支配关系是指, 从 src 指令出发, 到 ID_B 对应的所有 wait 指令(包括初步插入的与额外插入的), 路径上都存在 1 个 $wait_A$ 指令, 则认为 $wait_A$ 支配 $wait_B$. 如图 13(a)所示.



(a) src$A$与src$B$指令相同



(b) dst$A$与dst$B$指令相同



(c) src与dst指令都不相同

Fig. 13 Three cases of redundant synchronization primitives

图 13 冗余同步原语的 3 种情况

如果 $wait_A$ 支配 $wait_B$, 则 ID_B 即为冗余的, 可以被消除, 其对应的 set 与 wait 指令可以被删除. 其背后的逻辑是如果 $wait_A$ 支配 $wait_B$, 意味着从 src 到 $wait_B$ 的所有路径上, 都必然会经过 1 个 $wait_A$, 也即 $wait_B$ 执行时, $wait_A$ 必然已经执行过了, 即 src 指令必然执行结束了, 所以 dst_B 必然可以执行了, 因此 ID_B 对应的虚拟 set/wait 指令可以删除, 以减少对物理资源分配阶段的压力, 从而避免引入更多阻塞, 进而提升性能.

2) dst_A 与 dst_B 相同

此时, 判断 set_A 与 set_B 之间是否存在后支配关系. 类似地, 所谓后支配关系是指从所有 set_B 出发到 dst 指令的路径上都存在 1 个 set_A, 则认为 set_A 后支配 set_B. 如图 13(b)所示.

如果 set_A 后支配 set_B, 则 ID_B 即为冗余的, 可以被消除, 其对应的 set 与 wait 指令可以被删除. 其原因同 1)所述.

3)如果 src 指令与 dst 指令都不相同

此时判断 set_A 是否后支配 set_B, 同时 $wait_A$ 支配 $wait_B$, 若是, 则 ID_B 是冗余的, 可以被消除, 其对应的 set 与 wait 指令可以被删除. 如图 13(c)所示. 原因同 1)所述.

2.4 物理 ID 分配

由于昇腾处理器硬件上每对流水线之间可用的 ID 只有 4 个, 而在每对流水线对之间用到的虚拟 ID 至多可达到数十个, 这中间存在数量级的差距是同步原语自动插入面对的最大挑战之一.

该问题与虚拟寄存器到物理寄存器的分配问题有较大差距: 1)寄存器分配问题中, 如果同时活跃的虚拟寄存器数量大于物理寄存器数量, 可以选择将某些物理寄存器中的值溢出到内存中, 但在本问题中, 硬件同步资源不能被溢出, 是一直被占用的; 2)寄存器分配场景中, 可供分配的物理寄存器数量一般有十几个甚至数十个, 但在本问题中, 硬件上每对流水线之间可用的物理 ID 只有 4 个, 大大增加了虚拟 ID 到物理 ID 的分配难度.

因此, 本文分 2 步解决该问题: 1)合并虚拟 ID, 使得在任意程序点, 同时活跃的虚拟 ID 数目不超过 4; 2)基于图着色算法将虚拟 ID 分配到物理 ID 上.

2.4.1 合并虚拟 ID

该部分以逆后序(reverse post order, RPO)遍历程序中的基本块及其中的指令, 遍历过程中记录当前活跃的虚拟 ID 数量. 发现当前活跃的 ID 数大于 4 时, 就会开始对当前活跃的 ID 进行合并.

优先从当前活跃的跨迭代的 ID 中选择合并, 如对于 ID_A 与 ID_B: ID_A 与 ID_B 对应的额外 set 指令与 wait 指令需要位于同一个循环的前置头结点与出口基本块. 判断 ID_A 与 ID_B 能否合并(以下依然简称 ID_A 与 ID_B 对应的同步指令分别为 set_A, $wait_A$, set_B, $wait_B$; 对应的依赖指令对分别为$\langle src_A, dst_A\rangle$, $\langle src_B, dst_B\rangle$):

1)判断 $wait_A$ 能否支配 $wait_B$, 如果可以, 被支配的一方的 wait 指令可以删掉.

2)在此基础上, 判断 set_A 能否后支配 set_B. 如果可以, 被后支配的一方的 set 指令可以删掉. 如判断的结果是删除 $wait_A$ 与 set_B, 则对应的处理是从程序中删除 $wait_A$ 与 set_B, 删除 ID_B 位于前置头和出口基本块中的额外 set 指令与 wait 指令.

3)将 $wait_B$ 指令中的 ID_B 替换成 ID_A. 至此, 从跨迭代的 ID 中选择 ID_A 与 ID_B, 完成了合并, 合并之后的指令使用 ID_A, 即 ID_B 被删除了. 如图 14 所示, ID_A 与 ID_B 合并.

若跨迭代的 ID 中没有能合并的, 则从当前活跃的非跨迭代的 ID 中选择合并. 任意一对同时活跃的非跨迭代 ID 都可以进行合并.

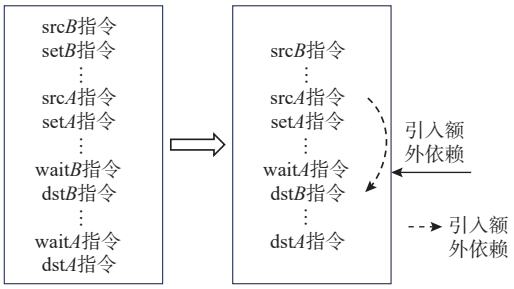


Fig. 14 An example of merging virtual ID

图 14 合并虚拟 ID 示例

需要注意的是, 该步骤需要对同时活跃的虚拟 ID 进行合并, 相当于在原本没有关系的 2 对指令间引入了额外的数据依赖, 可能导致硬件执行单元出现额外的阻塞, 降低程序性能. 如图 14 所示, 原本 dst_B 指令只需等待 src_B 指令执行结束就可开始执行, 但合并 ID_A 与 ID_B 之后, dst_B 指令需要等待之后的 src_A 指令执行结束才能开始执行, 相当于在 src_A 与 dst_B 之间引入了额外的依赖, 使 dst_B 指令提前阻塞了从 src_B 到 dst_A 指令间的这段时间.

因此, 2.3 节中的算法可以删除冗余的虚拟同步原语, 从而减小对本阶段的压力, 避免引入更多额外数据依赖, 进而提升性能.

该步骤执行结束后, 在任意程序点, 同时活跃的虚拟 ID 数都不大于 4, 也即一定可以分到 4 个物理 ID 上.

2.4.2 图着色算法分配物理 ID

本节以逆后序遍历程序中的基本块及其中的指令. 遇到 1 个 set 或 1 个 wait 指令, 首先查看其对应的虚拟 ID_A 是否已经被分配了物理 ID. 如果没有, 则对该虚拟 ID_A 进行物理 ID 分配:

1)检查 4 个物理 ID 是否已经全部被分配出去了. 如果没有, 则从还未被分配出去的物理 ID 中选择 1 个进行分配.

2)如果都已经分配出去, 则通过着色算法, 遍历当前同时活跃的虚拟 ID, 得到当前虚拟 ID_A 可以使用的物理 ID 有哪些, 未被它们使用的物理 ID 就是当前虚拟 ID_A 可以使用的物理 ID. 得到候选的可以使用的物理 ID 后, 从这些候选物理 ID 中选择 1 个分配给当前虚拟 ID_A:

① 判断候选者中是否存在这样的物理 ID, 上一个使用它的虚拟 ID_B, 从它对应的任意一个 $wait_B$ 指令到 ID_A 的任意一个 set_A 指令, 都有 happen-before 关系

(即“发生在前”或“先行发生”关系，A happen-before B 保证了 A 操作的结果对 B 操作可见，且 A 的执行顺序排在 B 之前)或不可达. 这里 wait_B happen-before set_A，保证了 ID_B 的所有 wait_B 指令都会在 ID_A 的 set_A 指令之前执行，也即不会出现连续的对同一物理 ID 的 set (这将导致处理器报错)，因为在 set_B 与 set_A 之间一定有 1 个 wait_B 指令. 因此，如果存在这样的物理 ID，直接分配该物理 ID.

② 如果不存在，则挑选候选物理 ID 中的第 1 个，找到上一次使用该物理 ID 的虚拟 ID_B，并找到 ID_B 所有的 wait_B，紧邻其后插入一对反向的 set 和 wait，以此来保证 ID_B 与 ID_A 之间的 happen-before 关系. 然后选择该物理 ID 进行分配.

至此，为当前处理的虚拟 ID_A 分配了一个物理 ID. 循环遍历完所有的虚拟 ID 后，可得到虚拟 ID 到物理 ID 的映射. 然后根据该映射，将同步原语中的虚拟 ID 替换为物理 ID 即可. 即完成了昇腾处理器上算子程序的同步原语插入.

3 实验结果

本节评估了本文提出的同步原语自动插入方法的正确性与性能.

3.1 实验设置

本文基于 LLVM 8.0 实现了昇腾处理器上的同步原语自动插入方法，测试环境为华为 Atlas 800 服务器(型号：9000)，其 CPU 为 4×鲲鹏 920，操作系统为 EulerOS 2.8，带有 8 张昇腾 910A 加速卡. 昇腾 910A 包含 32 个 AI Core，FP16 算力达 256 TFLOPS，INT8 算力为 512 TFLOPS；配备 32GB HBM，带宽达 1 200 GBps. 实验在单张昇腾 910A 加速卡上进行. 在正式运行测试前均进行预热(warm up)，并运行多次取数据平均值.

3.2 基准测试程序

本节使用微基准测试、单算子(包含矩阵类、向量类以及融合算子)基准程序进行测试. 对微基准测试，分别采用本文提出的方法自动插入同步原语，与使用 CCE 语言手动插入同步原语的方式进行比较. CCE 语言手动插入的同步原语是由华为公司经验丰富的算子开发工程师完成的，并进行了针对性的同步插入的调优(每个算子需要 1~2 人工作 1 周)，用来保证比较的公平性. 对单算子测试，由于 CCE 语言需要手动处理的同步原语过于复杂，故采用 TBE 进行编程. 由于 TBE 是调用封装好的 API 接口进行神经网络算子编程的，故相比 CCE 编程少了很多灵活性.

微基准测试覆盖了昇腾处理器上常见的向量类指令、矩阵类指令等. 单算子基准测试程序覆盖了深度学习中大部分常见算子，如向量类算子 LeakyRelu，LayerNorm，Softmax，Tanh 等，矩阵类算子 MatMul 等，以及融合算子 AttnScore.

出于性能考虑，通常编译器在处理 device 代码调用时都会内联，如 AMD ROCm[33] 等，CCE 也采用相同的内联策略，算子程序中的函数调用会被内联，再进行后续处理. 因此只需要处理内联后的程序，即程序内已无函数调用的情况，不涉及不同函数间的依赖关系. 故测试程序中不包含算子中有函数调用的情况，不影响实验完整性.

在昇腾处理器中，由于标量单元的运算能力很弱，被设计用来负责程序的流程控制(如循环控制、分支判断等)，以及矩阵/向量等指令的地址和参数计算. 因此出于性能考虑，一般使用向量与矩阵单元而不用标量单元来编写算子. 故实验中未对标量算子进行性能测试，不影响实验的完整性.

3.3 实验结果与分析

3.3.1 正确性分析

经验证，微基准测试与单算子基准测试中，所有自动插入方法的执行结果均与基线程序(手动插入与 TBE 算子)相同，验证了自动插入方法在不同场景下的正确性.

测试程序中矩阵乘(MatMul)等矩阵类算子最为复杂，其控制流图、执行路径等的规模与复杂度在常见算子中较为复杂. 在虚拟同步资源映射方面，矩阵乘算子 2 个执行单元间需要的虚拟 ID 最多达到了 24 个. 即对于矩阵乘算子，自动插入方法将 24 个虚拟 ID 映射到了 4 个物理 ID 上，由此证明本文提出的自动插入方法解决了大量虚拟同步资源到极有限数量的物理同步资源的映射问题，可以有效处理复杂程序的同步语句自动插入问题.

3.3.2 性能分析

本文中的性能，可视为算子执行时间的倒数，性能比可视为算子执行时间比例的倒数. 例如对于某算子，自动插入方法的执行时间为 T_1，手动插入方法的时间为 T_2，则自动插入相比手动插入的时间比为 T_1/T_2，性能比为 T_2/T_1. 具体来说，实验中所描述的性能，指的是通过统计算子在昇腾处理器上执行的时间所反映出来的性能. 如：自动插入方法相比手动插入的执行时间更短，则认为自动插入方法的性能更好. 由于对于同一算子，2 种方法的测试数据是相同的，因

此执行时间差距的背后, 就是硬件利用效率的差距, 即同步语句的插入质量更好, 硬件中不同执行单元间的阻塞就会越少, 硬件利用效率就会更高, 最终执行时间就会更短.

微基准测试中各算子的执行时间如图 15 所示. 在微基准测试中, 本文自动插入方法的平均性能为手动插入方法的 1.01 倍.

在单算子基准测试中, 由于不同算子的执行时间相差较大, 为了展示清晰, 我们对数据做了归一化处理, 以 TBE 的执行时间为基准, TBE 与自动插入方法算子的执行时间如图 16 和图 17 所示. 单算子基准测试的具体执行时间如表 2 所示.

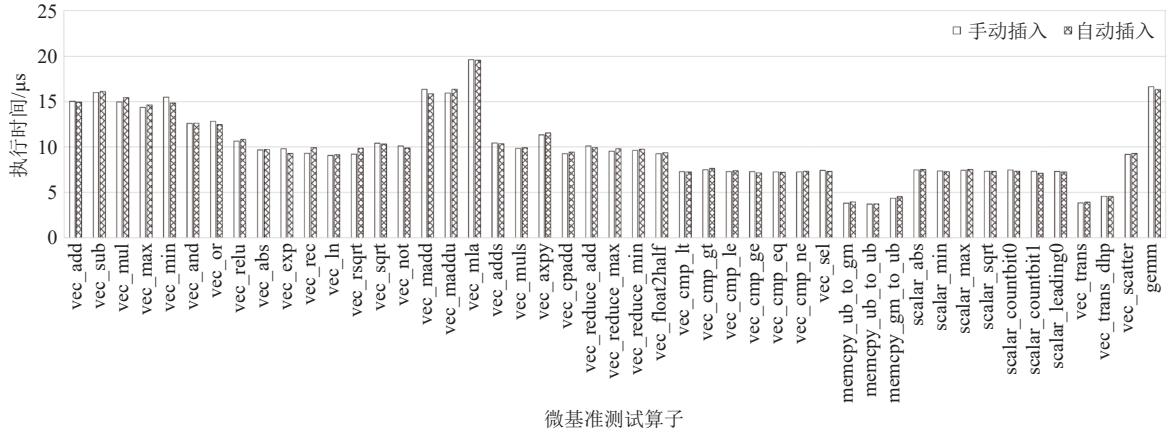


Fig. 15 Execution time of micro-benchmark test

图 15 微基准测试的执行时间

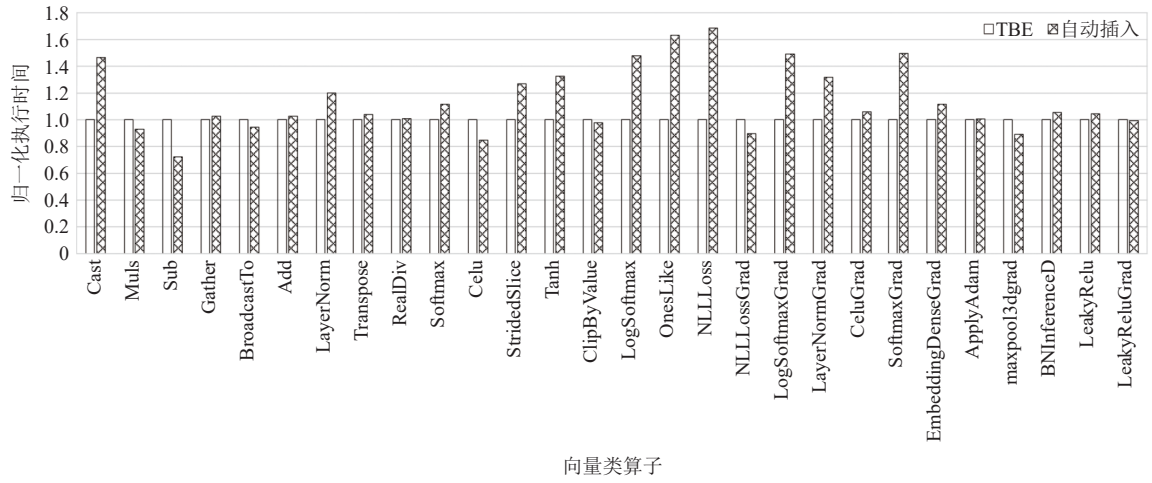


Fig. 16 Normalized execution time of vector operators

图 16 向量算子的归一化执行时间

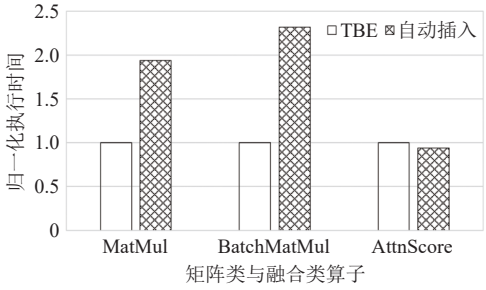


Fig. 17 Normalized execution time of matrix and fusion operators

图 17 矩阵与融合类算子的归一化执行时间

在单算子基准测试中, 本文的自动插入方法性能平均可达 TBE 算子的 0.89 倍, 同时在 BroadcastTo, Gelu, maxpool3dgrad 等算子上, 自动插入方法可达到比 TBE 算子更好的性能.

这是因为 TBE 算子调用的是经过专家高度优化的算子的 API 实现, 因此其平均性能比本文自动插入方法略高; 但也同样因为调用算子 API, 阻碍了一些可能的常见优化如算子融合等的实施, 因此本文提出的自动插入方法在一些算子上性能要好于 TBE 算子程序, 最高可达到 TBE 的 1.38 倍.

Table 2 Execution Time of Single Operator Benchmark Test

表 2 单算子基准测试执行时间 μs

算子类型	算子名称	TBE	自动插入
向量	Cast	2.81	4.12
	Muls	3.6	3.35
	Sub	4.09	2.96
	GatherV2	326.45	335.5
	BroadcastTo	4.11	3.88
	Add	53.83	55.29
	LayerNorm	243.02	291.48
	Transpose	30.49	31.65
	RealDiv	306.41	308.87
	Softmax	1 674.98	1 867.39
	Gelu	804.64	683.12
	StridedSlice	4.57	5.8
	Tanh	4.44	5.88
	ClipByValue	1.92	1.88
	LogSoftmax	4	5.91
	OnesLike	1.44	2.35
	NLLLoss	2.44	4.11
	NLLLossGrad	5.04	4.52
	LogSoftmaxGrad	3.74	5.58
	LayerNormGrad	71.11	93.77
	GeluGrad	1 377.39	1 458.49
	SoftmaxGrad	931.54	1 394.93
	EmbeddingDenseGrad	427.18	476.86
	ApplyAdam	10 989.29	11 052.46
	maxpool3dgrad	28.8	25.7
	BNInferenceD	109	115
	LeakyRelu	111	116
	LeakyReluGrad	4 466	4 437
矩阵	MatMul	321.79	623.69
	BatchMatMul	252.27	584.46
融合	AttnScore	570	536

同时我们注意到，矩阵类算子 MatMul 与 BatchMatMul 的性能不佳，这是因为矩阵类算子的复杂度极高，且厂商往往投入极大的资源专门优化 MatMul 类算子库，集成了大量专家经验与领域知识，因此本文的自动插入方法相对性能较差.

在融合算子 AttnScore 上，我们的自动插入方法相比于 TBE 调用封装算子的方法实现了 1.07 倍的加速，证明了相比于 TBE，本文方法可结合更多优化手段，以实现更好的性能目标.

3.3.3 算法分析

这里假设程序控制流图中的结点数为 M，每个流水线对的依赖指令对（即虚拟 ID）数量为 N.

本文提出的自动插入方法由虚拟同步原语插入、冗余同步原语删除、物理 ID 分配 3 部分组成.

1）虚拟同步原语插入

① 规范化控制流图. 遍历程序中所有的循环与 if-else 结构，按照规则进行处理即可. 故只需扫描 2 遍控制流图，该步骤的时间复杂度为 $O(M)$.

② 初步插入同步原语. 对于每个流水线对，遍历所有的依赖指令对〈src, dst〉，对每个指令对，按照规则处理即可，该步骤的时间复杂度为 $O(N)$.

③ 插入额外同步原语. 对每个指令对，根据规则判断并找到需要额外插入同步原语的位置并插入即可，时间复杂度为 $O(N)$；遍历基本块处理其他情况，时间复杂度为 $O(M)$；该步骤的时间复杂度为 $O(N+M)$.

2）冗余同步原语删除

对于每个流水线对，遍历所有虚拟 ID 的组合，组合数量为 $N(N-1)$，对每对虚拟 ID 组合，根据规则进行判断和去冗余即可，时间复杂度为 $O(N^2)$.

3）物理 ID 分配

① 合并虚拟 ID. 遍历程序控制流图，同时对活跃 ID 进行合并，最多进行 $N-4$ 次合并；每次合并的时间复杂度为 $O(1)$. 合并时，处理当前活跃的 5 个 ID，根据规则进行处理即可. 故该步骤时间复杂度为 $O(M+N)$.

② 图着色算法分配物理 ID. 由于合并虚拟 ID 已经确保任意程序点同时活跃的虚拟 ID 数不大于 4，一定可以分配到物理 ID 上，该步骤面临的问题复杂度相比传统图着色问题大为简化；故本文采用启发式算法，遍历程序控制流图处理所有虚拟 ID，对于遇到的虚拟 ID，按照简化后的启发式规则进行处理即可，因此该步骤时间复杂度为 $O(M+N)$.

综上所述，整个算法的时间复杂度为 $O(N+M)+O(N^2)$. 从实验中的基准测试程序可知，每个流水线对中依赖指令对的数量 N 至多为数十；同时遍历控制流图是编译优化中非常频繁的操作，算子程序的规模也属于正常程序的大小.

传统编译器中常见的算法，如线性扫描寄存器分配算法，如果使用平衡二叉树来搜索插入点，则时间复杂度为 $O(V\times\log R)$；若线性搜索插入点时，最坏情况下会导致 $O(V\times R)$ 的时间复杂度，其中 V 为待分配的变量数，R 为可用于分配的寄存器数[34]. 考虑到实际场景的算子程序中 M, N 的数量要比传统程序中

的 V, R 小很多,因此实际执行时,本文方法的时间开销与编译器中常见算法的时间开销相持平.综上所述,本文的自动插入方法不会引入过多的额外时间开销.

在生成的代码量方面,本文方法会为每对依赖指令初步插入 2 条同步原语,并视情况插入一定数量的额外同步原语(由 2.2.3 节可知,额外同步原语最多为 5 条).同时冗余同步原语删除、虚拟 ID 合并都会删除一定数量的同步原语,而手动插入同步原语时,对每个依赖指令对也需要插入至少 2 条同步指令(因为要优化性能,手动插入也需要一定数量的额外同步原语,只不过因为可以利用程序语义知识,插入的额外同步原语数量会比本文方法的少);因此鉴于依赖指令对的数量,相比手动插入方法,本文方法并不会明显增加生成的代码量.

总的来说,本文提出的自动插入方法在保证正确性的基础上,整体性能与专家程序员手动插入同步原语接近或持平,在编程灵活性、提高开发效率以及发挥硬件极端性能之间得到了一种平衡,大大方便了程序员进行算子编程,为进一步的深度优化打开了空间.

4 结 论

本文提出了面向昇腾处理器的高性能同步原语自动插入方法.通过“虚拟同步资源”的抽象解耦了同步原语插入和物理同步资源分配 2 个难题,通过自动插入虚拟同步原语,基于偏序关系消除冗余同步原语以及合并虚拟同步原语、基于图着色算法分配物理同步资源分配等技术,实现了昇腾处理器上同步原语的自动插入.实验表明,本文提出的自动插入方法在保证正确性的基础上,整体性能可接近或持平专家程序员手动插入同步原语的水平.

同时,本文的方法也存在可进一步完善之处,如对矩阵类复杂算子的优化尚有不足,未来将尝试更多的优化策略,进一步提升性能.另外,虽然本文是在昇腾处理器平台上进行的实验,但我们相信本文提出的方法同样适用于其他采用 post-wait 硬件同步原语的异构处理器,因此未来可以拓展到其他平台进行实验.

作者贡献声明:李帅江设计并实现算法、撰写论文;张馨元、石曦予协助设计实现了部分算法、设计实验方案,并进行实验对比分析;田行辉和徐晓忻协助进行实验对比分析;赵家程和崔慧敏对算法设计提出了关键性意见,并审阅与修订论文.

参 考 文 献

[1] Tomasulo R M. An efficient algorithm for exploiting multiple arithmetic units[J]. IBM Journal of research and Development, 1967, 11(1): 25−33

[2] Smith J E. A study of branch prediction strategies[C]//Proc of the 8th Annual Symp on Computer Architecture. New York: ACM, 1981: 135−148

[3] Thornton J E. Parallel operation in the control data 6600[C]//Proc of the Fall Joint Computer Conf, Part II: Very High Speed Computer Systems. New York: ACM, 1964: 33−40

[4] Moudgill M, Pingali K, Vassiliadis S. Register renaming and dynamic speculation: An alternative approach[C]//Proc of the 26th Annual Int Symp on Microarchitecture. Los Alamitos, CA: IEEE Computer Society, 1993: 202−213

[5] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks[C]//Proc of the 25th Int Conf on Neural Information Processing Systems-Volume 1. New York: Curran Associates Inc, 2012: 1097−1105

[6] Zhang Rui, Li Jintao. A survey on algorithm research of scene parsing based on deep learning[J]. Journal of Computer Research and Development, 2020, 57(4): 859−875 (in Chinese)
(张蕊, 李锦涛. 基于深度学习的场景分割算法研究综述[J]. 计算机研究与发展, 2020, 57(4): 859−875)

[7] Brown T B, Mann B, Ryder N, et al. Language models are few-shot learners[C]//Proc of the 34th Int Conf on Neural Information Processing Systems. New York: Curran Associates Inc, 2020: 1877−1901

[8] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[C]//Proc of the 31st Int Conf on Neural Information Processing Systems. New York: Curran Associates Inc, 2017: 6000−6010

[9] Jumper J, Evans R, Pritzel A, et al. Highly accurate protein structure prediction with AlphaFold[J]. Nature, 2021, 596(7873): 583−589

[10] Li Hongshun, Yu Hua, Gong Xiujun. A deep learning model for predicting RNA-binding proteins only from primary sequences[J]. Journal of Computer Research and Development, 2018, 55(1): 93−101 (in Chinese)
(李洪顺, 于华, 宫秀军. 一种只利用序列信息预测 RNA 结合蛋白的深度学习模型[J]. 计算机研究与发展, 2018, 55(1): 93−101)

[11] OpenAI. AI and compute[EB/OL]. [2024-02-01]. https://openai.com/research/ai-and-compute

[12] Nvidia. NVIDIA tensor cores[EB/OL]. [2024-02-01]. https://www.nvidia.com/en-us/data-center/tensor-cores/

[13] AMD. AMD Instinct™ MI300 series accelerators[EB/OL]. [2024-02-01]. https://www.amd.com/en/products/accelerators/instinct/mi300.html

[14] Hygon. Hygon deep computing unit[EB/OL]. [2024-02-01]. https://www.hygon.cn/product/accelerator (in Chinese)
(海光. 海光深度计算处理器[EB/OL]. [2024-02-01]. https://www.hygon.cn/product/accelerator)

[15] Chen Tianshi, Du Zidong, Sun Ninghui, et al. DianNao: A small-

footprint high-throughput accelerator for ubiquitous machine-learning[C]//Proc of the 19th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2014: 269–284

[16] Liao Heng, Tu Jiajin, Xia Jing, et al. Ascend: A scalable and unified architecture for ubiquitous deep neural network computing[C]//Proc of the 27th IEEE Int Symp on High-Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2021: 789−801

[17] Liao Heng, Tu Jiajin, Xia Jing, et al. DaVinci: A scalable architecture for neural network computing[C/OL]//Proc of the 31st Hot Chips Symp. Piscataway, NJ: IEEE, 2019[2024-02-01]. https://www.old.hotchips.org/hc31/HC31_1.11_Huawei.Davinci.HengLiao_v4.0.pdf

[18] Huawei. CANN 8.0. RC1. alpha001[EB/OL]. [2024-02-01]. https://www.hiascend.com/document/detail/zh/CANNCommunityEdition/80RC1alpha001/devguide/devguide/devguide_0001.html

[19] Barham P, Isard M. Machine learning systems are stuck in a rut[C]//Proc of the 17th Workshop on Hot Topics in Operating Systems. Berkeley, CA: USENIX Association, 2019: 177−183

[20] Nicolau A, Li Guangqiang, Kejariwal A. Techniques for efficient placement of synchronization primitives[C]//Proc of the 14th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2009: 199−208

[21] Zhai A, Steffan J G, Colohan C B, et al. Compiler and hardware support for reducing the synchronization of speculative threads[J]. ACM Transactions on Architecture and Code Optimization, 2008, 5(1): 1−33

[22] Nicolau A, Li Guangqiang, Veidenbaum A V, et al. Synchronization optimizations for efficient execution on multi-cores[C]//Proc of the 23rd Int Conf on Supercomputing. New York: ACM, 2009: 169−180

[23] Tseng C W. Compiler optimizations for eliminating barrier synchronization[J]. ACM SIGPLAN Notices, 1995, 30(8): 144−155

[24] Chen Dingkai, Yew P C. Redundant synchronization elimination for DOACROSS loops[J]. IEEE Transactions on Parallel and Distributed Systems, 1999, 10(5): 459−470

[25] Aldrich J, Chambers C, Sirer E G, et al. Static analyses for eliminating unnecessary synchronization from Java programs[C]//Proc of the 6th Int Symp on Static Analysis. Berlin: Springer, 1999: 19–38

[26] Bogda J, Hölzle U. Removing unnecessary synchronization in Java[C]//Proc of the 14th ACM SIGPLAN Conf on Object-Oriented Programming, Systems, Languages, and Applications. New York: ACM, 1999: 35−46

[27] Han H, Tseng C W. Compile-time synchronization optimizations for software DSMs[C]//Proc of the 1st Merged Int Parallel Processing Symp and Symp on Parallel and Distributed Processing. Piscataway, NJ: IEEE, 1998: 662−669

[28] Li Ang, van den Braak G J, Corporaal H, et al. Fine-grained synchronizations and dataflow programming on GPUs[C]//Proc of the 29th ACM on Int Conf on Supercomputing. New York: ACM, 2015: 109−118

[29] Liu Lifeng, Liu Meilin, Wang Chongjun, et al. Compile-time automatic synchronization insertion and redundant synchronization elimination for GPU kernels[C]//Proc of the 22nd Int Conf on Parallel and Distributed Systems. Piscataway, NJ: IEEE, 2016: 826−834

[30] Moses W S, Ivanov I R, Domke J, et al. High-performance GPU-to-CPU transpilation and optimization via high-level parallel constructs[C]//Proc of the 28th ACM SIGPLAN Annual Symp on Principles and Practice of Parallel Programming. New York: ACM, 2023: 119−134

[31] Sorensen T, Donaldson A F, Batty M, et al. Portable inter-workgroup barrier synchronisation for GPUs[C]//Proc of the 31st ACM SIGPLAN Int Conf on Object-Oriented Programming, Systems, Languages, and Applications. New York: ACM, 2016: 39−58

[32] LLVM. Loop-simplify: Canonicalize natural loops[EB/OL]. [2024-03-20]. https://llvm.org/docs/Passes.html#loop-simplify-canonicalize-natural-loops

[33] AMD. Register pressure in AMD CDNA2™ GPUs[EB/OL]. [2024-03-20]. https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-register-pressure-readme

[34] Poletto M, Sarkar V. Linear scan register allocation[J]. ACM Transactions on Programming Languages and Systems, 1999, 21(5): 895−913



Li Shuaijiang, born in 1998. PhD candidate. Student member of CCF. His main research interest includes heterogeneous programming.

李帅江, 1998 年生. 博士研究生. CCF 学生会员. 主要研究方向为异构编程.



Zhang Xinyuan, born in 1994. Master, engineer. Her main research interest includes heterogeneous compile.

张馨元, 1994 年生. 硕士, 工程师. 主要研究方向为异构编译.



Zhao Jiacheng, born in 1989. PhD, associate professor. His main research interests include heterogeneous programming, machine learning systems, and compiler infrastructure for domain specific architectures.

赵家程, 1989 年生. 博士, 副研究员. 主要研究方向为异构编程、机器学习系统和领域定制架构编译基础设施.



Tian Xinghui, born in 1989. Master. His main research interest includes AI compiler.

田行辉, 1989 年生. 硕士. 主要研究方向为 AI 编译器.



Shi Xiyu, born in 1996. Master, engineer. Her main research interest includes heterogeneous programming.

石曦予, 1996 年生. 硕士, 工程师. 主要研究方向为异构编程.



Xu Xiaoxin, born in 1986. Master. His main research interest includes AI system optimization.

徐晓忻, 1986 年生. 硕士. 主要研究方向为 AI 系统优化.



Cui Huimin, born in 1979. PhD, professor. Her main research interests include compiler construction, compiler optimization, and heterogeneous computing.

崔慧敏, 1979 年生. 博士, 研究员. 主要研究方向为编译器构建、编译优化、异构计算.