中国科学: 信息科学 2024 年 第 54 卷 第 12 期: 2743–2760
SCIENTIA SINICA Informationis
《中国科学》杂志社
SCIENCE CHINA PRESS
论文

保证延迟敏感型任务服务质量的情况下利用流处理器内所有并行性以最大化系统吞吐

赵涵[1,4], 邓俊骁[1], 崔炜皞[1], 陈全[1*], 曾德泽[2], 杨静[3], 过敏意[1]

1. 上海交通大学电子信息与电气工程学院, 上海 200240
2. 中国地质大学计算机学院, 武汉 430074
3. 贵州大学机械工程学院, 贵阳 550025
4. 上海市金融信息技术研究重点实验室 (上海财经大学), 上海 200433
* 通信作者. E-mail: chen-quan@cs.sjtu.edu.cn

收稿日期: 2024–07–02; 修回日期: 2024–08–13; 接受日期: 2024–08–21; 网络出版日期: 2024–12–03

国家重点研发计划 (批准号: 2022YFB4501400) 和国家自然科学基金 (批准号: 62302302) 资助项目

摘要 为了应对越来越高的算力需求, GPU 在流处理器内集成了多种通用计算单元及专用计算单元 (FP32 Core, INT32 Core, FP64 Core, Tensor Core, RT Core). 任意一种 GPU 内可能包含以上计算单元中的部分单元. 尽管 GPU 的流处理器内存在着多种计算单元, 它们之间的计算并行性无法从硬件设计白皮书中获知. 与此同时, 现有调度接口无法支持使用不同计算单元的核函数并行利用这些计算资源, 更无法支持运行时的精细调度以最大化系统吞吐. 面对以上问题, 我们提出了硬件感知吞吐导向的核函数调度方法 Hato. Hato 首先设计了一个硬件并行性感知工具, 支持为任意 GPU 定位出所有的流处理器内并行性. 其次, Hato 提出了一个核函数混跑建模方法, 通过核函数混跑利用到流处理器内并行性, 并支持核函数在混跑情况下的执行时间精准预测. 最后, Hato 提出了一个吞吐导向的调度策略, 支持在保证延迟敏感型应用服务质量的同时, 利用到所有可能的流处理器内并行性, 以最大化整体系统吞吐. 实验结果表明, Hato 相比最新调度系统 Tacker 提升了平均 19.2%, 最高 54.1% 的系统吞吐.

关键词 GPU, 流处理器内并行性, 吞吐提升, 运行时系统

1 引言

为了应对大量应用对算力的高需求, GPU 集成了多种计算单元. 如图 1 所示, GPU 由多个流处理器组成 (streaming multiprocessor, SM). 它首先在 SM 内集成了多种负责不同功能的通用计算核心, 例如负责处理 32 位浮点数的 FP32 Core, 负责处理 64 位浮点数的 FP64 Core, 以及负责处理整数计

引用格式: 赵涵, 邓俊骁, 崔炜皞, 等. 保证延迟敏感型任务服务质量的情况下利用流处理器内所有并行性以最大化系统吞吐. 中国科学: 信息科学, 2024, 54: 2743–2760, doi: 10.1360/SSI-2024-0121
Zhao H, Deng J X, Cui W H, et al. Exploiting all intra-SM parallelism to maximize the throughput while ensuring QoS (in Chinese). Sci Sin Inform, 2024, 54: 2743–2760, doi: 10.1360/SSI-2024-0121

© 2024《中国科学》杂志社

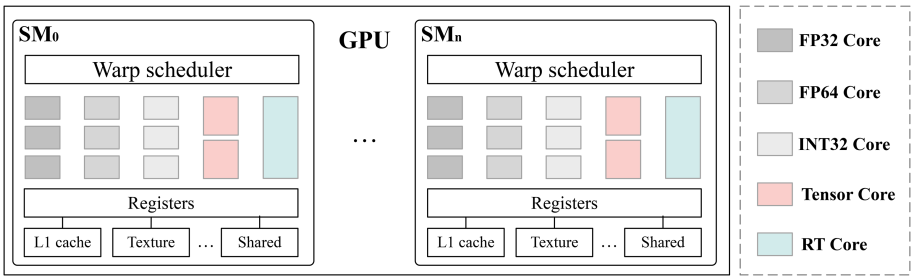

图 1 (网络版彩图) GPU 架构图

Figure 1 (Color online) GPU architecture

算的 INT32 Core. 此外, 为了满足深度学习应用和视频渲染应用的加速需求, 最新发布的 GPU 在 SM 内集成了 Tensor Core 和 RT Core. Tensor Core 专用于深度学习应用中的矩阵乘算法 [1], 而 RT Core 专用于视频渲染应用中的光线追踪算法 [2].

鉴于 GPU 的强大算力, 很多研究工作关注于将多个任务混合部署在同一个 GPU 上, 以有效地提升 GPU 利用率. 根据应用的服务质量需求 (Quality-of-Service), 可以将数据中心上的任务划分为两类: 延迟敏感型任务和尽力而为型任务. 延迟敏感型任务需要在特定时间目标内完成计算, 以提供给用户满意的服务, 例如实时物体识别任务 [3]. 尽力而为型任务则没有任何的时间目标, 尽可能快地完成计算即可, 例如离线数据分析任务 [4]. 在这种情况下, 调度尽力而为型任务来使用延迟敏感型任务没有利用到的 GPU 资源, 能够很好地提高 GPU 利用率.

混合部署的相关研究主要分为两类: GPU 层次的混合部署方法, 以及 SM 层次的混合部署方法. 对于 GPU 层次的混合部署. 具体而言, Baymax [5] 通过调整核函数 (kernel) 的发射顺序, 在保证延迟敏感型任务服务质量的条件下, 支持尽力而为型任务使用空闲的 SM 时间片. 对于 SM 层次的混合部署, Tacker [6] 在核函数重排序之外, 提出了核函数融合方法, 支持两种核函数同时利用到 SM 内的 Tensor Core 与 FP32 Core.

尽管核函数融合方法能够利用到 Tensor Core 与 FP32 Core 的并行性, 它仍然存在着几个问题. 首先, 现有核函数融合方法假设以上两种计算单元的理论并行性是已知的, 然而这个信息是无法轻易得到的. 虽然白皮书给出了相应的硬件设计信息, 但是并没有指示 SM 内可能的计算并行性. 举例而言, 2080Ti 的 RT Core 与 FP32 Core, V100 的 Tensor Core 与 FP64 Core 都不存在并行性.

其次, 核函数融合方法并不能适用于使用 RT Core 的核函数. 这是因为 RT Core 的使用需要使用编程语言 Optix [7], 而其他计算单元的使用需要使用编程语言 CUDA [8]. 使用不同编程语言的核函数无法进行核函数融合. 最后, 核函数融合可能会带来次优的并行性能. 举例而言, 核函数 1 每个线程需要 100 个寄存器, 核函数 2 每个线程需要 50 个寄存器. 在核函数融合之后, 融合核函数的每个线程都需要使用 100 个寄存器. 额外的资源使用最终带来了次优的并行性能.

面对第一个问题, 我们发现, 基于一套基准应用的混跑实验可以真实反映一个未知 GPU 在 SM 内的所有计算并行性. 通过仅仅运行一次这套基准应用, 我们可以低开销地定位出所有可能的 SM 内计算并行性. 面对后两个问题, 我们发现, CUDA stream 提供了解决问题的可能机会 [9]. 首先, Optix 语言和 CUDA 语言都支持 CUDA stream 的使用. 我们可以使用 CUDA stream 来混合部署使用 RT Core 的核函数与使用其他计算单元的核函数. 其次, CUDA stream 支持一个核函数使用另一个核函数没有使用完的 SM 资源. 这也为核函数在 SM 层级上进行混合部署提供了可能性.

然而, CUDA stream 也存在着自己的问题. 首先, CUDA stream 也不能感知一个未知 GPU 上可

能的 SM 内并行性. 其次, 直接使用 CUDA stream 来混跑两个使用不同计算单元的核函数无法带来吞吐提升, 这是因为核函数在 SM 上遭遇了严重的资源竞争. 第三, 延迟敏感型任务要求精准的预测每一个核函数的执行时间. 如何精准的预测核函数在混跑情况下的性能仍然是未知的. 最后, 由于一个 GPU 上可能有多种 SM 内并行可能性, 如何设计一套通用的调度策略来最大化整体系统吞吐也是亟需解决的难题.

为了解决以上难题, 我们提出了硬件感知吞吐导向的核函数调度方法 Hato. Hato 能够针对任意一个 GPU 定位出所有的 SM 内并行性, 并能够在保证延迟敏感型任务服务质量的同时, 利用到所有可能的 SM 内并行性, 以最大化整体系统吞吐. 本文贡献总结如下.

- 提出了一个硬件并行性感知工具. 对于一个未知 GPU, 该工具提供一套基准应用, 通过仅仅运行一遍这些基准程序, 定位出该 GPU 上所有可能的 SM 内并行性.
- 提出了一个核函数混跑建模方法. 对于一种特定的 SM 内并行性, 该方法首先使用持久线程块技术对核函数的 SM 资源使用进行管理, 然后对可能的核函数混跑对[1)]进行性能预测建模.
- 提出了一个吞吐导向的通用调度策略. 在拿到可能的核函数混跑对以及性能预测模型后, 吞吐导向的在线调度模块设计了一套通用的核函数管理方法, 以在保证延迟敏感型任务服务质量的同时, 综合考虑所有的 SM 内并行性, 以最大化整体的系统吞吐.

我们基于 2080Ti, V100, 3090 三种真实硬件进行了相应实验. 实验结果表明, Hato 不仅保证了延迟敏感型任务的服务质量, 并相比 Tacker 平均提升了 19.2%, 最高提升了 54.1% 的系统吞吐.

2 动机

2.1 核函数融合方法的问题

我们将使用 FP32 Core 的核函数命名为 FP32 kernel, 使用 FP64 Core 的核函数命名为 FP64 kernel, 使用 INT32 Core 的核函数命名为 INT32 kernel, 使用 Tensor Core 的核函数命名为 TC kernel, 使用 RT Core 的核函数命名为 RT kernel. 最新研究工作 Tacker [6] 将 TC kernel 和 FP32 kernel 融合成为一个新的核函数, 这个核函数能够利用到 Tensor Core 与 FP32 Core 的并行性.

举例而言, 来自英伟达的 $tgemm$ [10] 是一个 TC kernel, 来自基准测试集 Parboil [11] 的 fcp 是一个 FP32 kernel. 核函数 $tgemm$ 的一个线程块有 128 个线程, 核函数 fcp 的一个线程块也有 128 个线程. 当核函数 $tgemm$ 与核函数 fcp 被融合成为一个新的核函数, 这个核函数的一个线程块有 128+128=256 个线程. 由于使用不同计算单元的线程通过一个融合线程块发射到 SM 上, 所以核函数融合能够利用到 SM 内 Tensor Core 与 FP32 Core 的并行性.

然而, Tacker 提出的核函数融合方法有几个问题. 首先, 现有核函数融合方法假设以上两种计算单元的理论并行性是已知的, 然而这个信息是无法轻易得到的. 其次, 核函数融合方法带来了次优的并行性能. 很多核函数混跑对有很差的吞吐提升. 这是因为核函数融合带来了额外的资源使用. 最后, 核函数融合方法无法处理 RT kernel 与其他核函数的融合. 这是因为 RT kernel 需要使用编程语言 Optix, 而其他核函数需要编程语言 CUDA.

1) 核函数混跑对: 两个被混跑而同时执行的核函数.

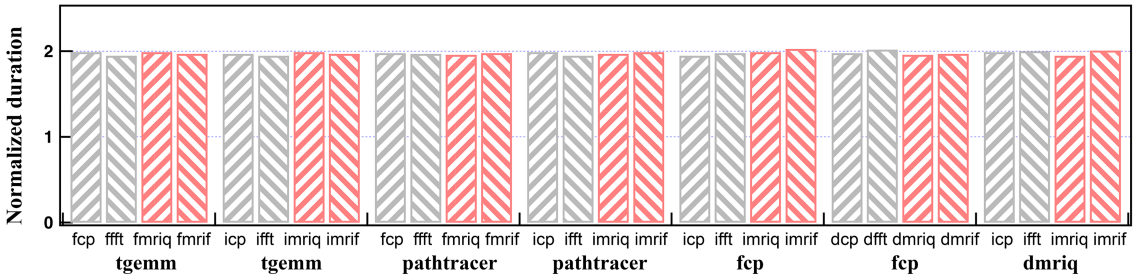

图 2 (网络版彩图) 直接使用 CUDA stream 进行混跑的执行时间

Figure 2 (Color online) Duration of different kernel pairs under CUDA streams

2.2 CUDA stream 的机会和问题

CUDA stream [9] 是英伟达官方提出的调度接口. 当一个 stream 中的核函数没有使用完 SM 上的资源时, 另一个 stream 中的核函数的线程块就有机会调度到 SM 上. 具体而言, 一个 SM 上有 1024 个线程槽, 64KB 共享内存, 65536 个寄存器. 假设核函数 1 使用了 512 个线程槽, 32KB 共享内存和 32768 个寄存器. 只要核函数 2 使用了小于等于 512 个线程槽, 小于等于 32KB 共享内存, 以及小于等于 32768 个寄存器, 它们就可以同时在 GPU 上进行计算.

在以上过程中, 由于 CUDA stream 不影响两个核函数的资源使用, 它首先不会遭遇核函数融合方法的资源浪费问题. 与此同时, 我们对于 CUDA stream 和 Optix 有一个发现. Optix 应用也可以使用 CUDA stream 进行核函数管理, 并且支持非 RT kernel 的发射管理. 换句话而言, CUDA stream 也可以用于解决核函数融合方法不能支持 RT kernel 与其他核函数统一调度管理的问题.

然而, CUDA stream 在利用 SM 内并行性方面, 也遇到了自己的问题. 图 2 展示了直接使用 CUDA stream 来混跑使用不同计算单元的核函数的执行时间. 我们使用来自英伟达的官方基准测试集 CUDA samples [12] 和 Optix toolkit [13], 以及广泛使用的 GPU 基准测试集 Parboil [11] 中的应用来作为测试程序. 在这个实验中, *pathtracer* 是 RT kernel, *tgemm* 是 TC kernel, fcp, fft, $fmriq$, $fmrif$ 是 FP32 kernel, icp, $ifft$, $imriq$, $imrif$ 是 INT32 kernel, dcp, $dfft$, $dmriq$, $dmrif$ 是 FP64 kernel. 在 Parboil 中, icp 和 fcp 采用同样的算法, 但是使用不同的数值精度进行计算.

对于实验配置而言, 包含 FP64 kernel 的混跑实验在 V100 上进行, 包含 RT kernel 的混跑实验在 3090 上进行, 而其他核函数的混跑实验在 2080Ti 上执行. 所有核函数的计算时间都被归一化为 1. 具体而言, 我们将所有核函数的执行时间都调整到同一个时间 T, 例如 5 ms. 在此基础上, 我们收集核函数混跑对的执行时间, 并基于时间 T 进行归一化计算. 从图 2 可知, 所有核函数混跑对的归一化执行时间都为 2, 也就是 10 ms. 这意味着, 使用 CUDA stream 的一个问题是直接混跑两个核函数无法带来吞吐提升.

此外, 保证延迟敏感型任务的服务质量需要精准的性能预测, 最大化系统吞吐需要清晰感知核函数混跑情况下的执行过程. 然而, 现有的性能预测方法无法直接适用于 CUDA stream 下的核函数性能预测. 与此同时, CUDA stream 下的执行状态不感知, 也无法支持精细的核函数调度. 因此, CUDA stream 的另一个问题是无法保证延迟敏感型任务的服务质量, 以及获得最大的系统吞吐. 最后, CUDA stream 也面临着无法感知 SM 内并行性的问题.

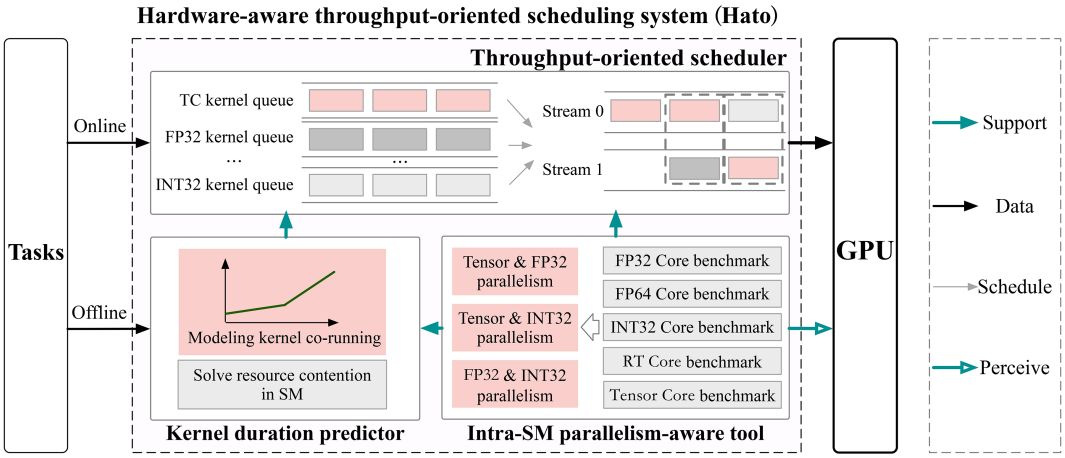

图 3 (网络版彩图) 系统架构图

Figure 3 (Color online) Architecture overview of Hato

3 系统架构

面对以上问题, 我们提出了硬件感知吞吐导向的核函数调度方法 Hato. Hato 可以支持为不同 GPU, 自适应的搜索 SM 内所有的并行可能性, 并为该 GPU 制定吞吐导向的调度策略.

图 3 展示了 Hato 的系统架构图. Hato 由并行性感知子模块、混跑建模子模块、以及吞吐导向调度子模块组成. 并行性感知子模块通过一套基准应用定位出一个 GPU 上所有的 SM 内并行性. 在获知一个 GPU 上的 SM 内并行性后, 混跑建模子模块构建相应的核函数混跑对. 对于这些核函数混跑对, 混跑建模子模块首先筛选出有吞吐提升的核函数混跑对, 再为这些核函数混跑对构建性能预测模型. 在获得预测模型后, 吞吐导向调度子模块设计了一套通用的吞吐导向调度策略. 其在运行时进行合适的核函数混跑调度, 以在保证延迟敏感型任务服务质量的同时, 最大化整体系统吞吐.

在任意一个未知的 GPU 上, 并行性感知子模块通过运行一套基准应用, 首先定位出 GPU 内的主要计算单元以及所有 SM 内并行性. 例如, 2080Ti 上有 FP32 Core, INT32 Core, Tensor Core 与 RT Core 四种计算单元; 并拥有 FP32-Tensor, INT32-Tensor, FP32-INT32 三种 SM 内并行性.

在获知一个 SM 内并行性后 (例如 Tensor Core 与 FP32 Core), 混跑建模子模块首先基于该 GPU 上运行的应用, 构建可能的 TC-FP32 核函数混跑对. 对于任意一个核函数混跑对 (例如 *tgemm* 与 *cp*), 混跑建模子模块首先使用持久化线程块解决两个核函数在 SM 上的资源竞争, 其次为这个核函数混跑对构建性能预测模型. 这个性能预测模型支持在动态负载下进行精准的性能预测.

在获得所有带来吞吐提升的核函数混跑对后, 吞吐导向调度子模块提出了一套通用的混跑调度策略. 该策略能够支持在不同 GPU 上, 保证延迟敏感型任务服务质量的同时, 最大化整体系统吞吐. 具体而言, 该模块统一考量不同 SM 内并行性带来的吞吐提升, 根据延迟敏感型服务的目标时间余量, 决定调度延迟敏感型任务的核函数进行单独执行或者混跑执行.

需要注意的是, Hato 针对私有数据中心而设计. 在私有数据中心上, 延迟敏感型任务与尽力而为型任务都是长时间运行的, 而且数据中心管理员能够有它们代码的管理权限. 这个实验场景与之前很多工作一致 [5,14,15]. 在这个实验场景下, 离线分析应用的核函数并构建相应的核函数混跑对, 以获得长期的吞吐提升是可以接受的.

```
dim3 kernel_grid, kernel_block;
kernel_grid.x = SM_NUM;      // ① Utilize all SMs in the GPU
kernel_block.x = 512;        // ① Use half resources in SM, like thread slots

__global__ void fp32_bench(float* in, float* out, int iteration, unsigned int* sm_id) {
    float register[];
    load_memory_to_register(in, register);     // ② Data loading only happens once
    for (int i = 0; i < iteration; i++) {      // ④ Adjust kernel duration via iteration
       perform_computation(register);
       // ② Compute only with registers
       // ⑤ fp32_bench, int32_bench and fp64_bench use the same compute mode
       //    to measure the compute speed
    }
    store_data_to_memory(out, register);     // ② Data storing also only happens once

    asm("mov.u32 %0, %smid;" : "=r"(sm_id[blockIdx.x]) ); // ③ Get SM ID
}
```

图 4 (网络版彩图) 基准应用模式

Figure 4 (Color online) Design mode of benchmark applications

4 并行性感知子模块

4.1 现有问题

首先, 现有 GPU 白皮书中仅仅给出了 SM 内的所有计算单元, 却并没有告知它们之间的可能并行性. 例如, 2080Ti 内同时包含着 5 种计算单元, 而这些计算单元间的并行性是未知的. 其次, 这 5 种计算单元中可能包含着辅助型计算单元. 例如, 2080Ti 的每个 SM 上有 64 个 FP32 Core, 却仅仅有 2 个 FP64 Core. 由于 2080Ti 的设计目标不包含高精度的计算任务, FP64 Core 带来的峰值算力仅仅为其他计算单元的几十分之一, 并行利用其与其他单元带来的吞吐提升可以忽略不计. 最后, 同样的两种计算单元可能在一种 GPU 上不存在并行性, 而在另一种 GPU 上存在并行性. 例如, RT Core 与 FP32 Core 在 2080Ti 和 3090 上的并行性是不同的.

4.2 基准应用

面对以上问题, 我们仅仅能够通过实验手段确定一个未知 GPU 上的主要计算单元以及所有的 SM 内并行性. 也因此, 我们尝试通过提出一套基准应用, 以自动化低开销地定位出任意 GPU 在 SM 内的所有并行性. 这套基准应用需要实现 5 个功能. 第一, 需要在没有人为干预的情况下, 通过一次运行获知所有需要的信息. 第二, 应该分别表达每种计算单元的计算需求, 并获得尽可能高的计算效率. 第三, 能够从所有计算单元中识别出主要计算单元和辅助计算单元. 第四, 需要支持任意两种基准应用可以同时发射到 SM 上. 第五, 应该进行自适应的计算时间归一化, 以支持准确的并行性指标计算.

为了实现以上功能, 我们针对所有计算单元设计了计算效率优先的应用模式. 图 4 展示了具体的应用设计. 首先, 我们仅仅支持基准应用在每个 SM 上最多使用一半的 SM 资源 (线程槽、共享内存、寄存器). 通过这样的方式, 可以支持任意两种基准应用能够同时在一个 SM 上执行计算.

其次, 我们仅仅支持基准应用在应用开始时进行一次访存读入, 在应用结束时进行一次访存写入. 而在应用的中间部分, 我们添加了相应计算以表达对特定计算单元的使用需求. 需要注意的是, 核心计算部分仅仅利用寄存器进行相应计算, 这样可以避免访存竞争带来的结果不准确.

第三, 我们通过对核心计算部分添加循环, 以支持基准应用的执行时间归一化. 简单添加循环并不会对以上的应用特征带来任何影响, 这是因为应用有读入和写出的强依赖关系, 中间计算部分不会被编译器视为无效计算. 因此, 在核心计算区添加循环能够支持执行时间自适应归一化.

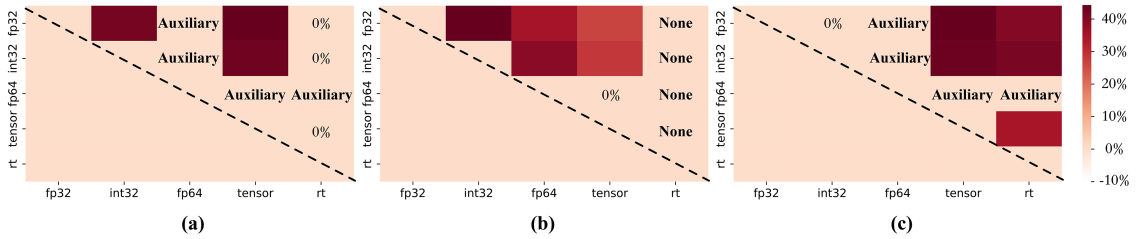

图 5 (网络版彩图) 2080Ti, V100 以及 3090 上的 SM 内计算并行性

Figure 5 (Color online) Intra-SM parallelism on 2080Ti, V100 and 3090. OverlapRate on (a) 2080Ti; (b) V100; (c) 3090

最后, 由于 Tensor Core 和 RT Core 相比通用计算单元有更高的峰值算力, 我们仅仅从 FP32 Core, FP64 Core, INT32 Core 中定位辅助型计算单元. 我们为这 3 种计算单元采用同样的核心计算指令, 例如不规则乘加计算. 通过对核心计算的运算次数进行提前统计, 我们可以运行时计算出每种计算单元的计算速度 (operations per second, 每秒操作数). 如果一种计算单元的计算速度为另一种计算单元的十分之一以下, 则我们可以确认其为辅助型计算单元.

基于以上应用模式, 我们为每一种计算单元准备了一种基准应用. 在面向一个未知 GPU 时, 我们首先运行一遍基准应用来确认 GPU 上 SM 内存在的计算单元, 并通过 $cudaDeviceGetAttribute()$ 获知 GPU 上的 SM 数量. 如果某些应用无法执行, 则该 GPU 上不存在相应的计算单元. 其次, 我们计算 3 种通用计算单元的计算速度, 以定位出辅助型计算单元. 由于辅助型计算单元有非常低的计算峰值, 其不被考虑作为 SM 内并行性. 第三, 我们针对主要计算单元, 调整其对应基准应用的线程块数量, 以支持其使用到每一个 SM; 并调整其核心计算部分的循环数量, 以支持所有基准应用有同样的计算时间. 最后, 我们针对任意 2 个主要计算单元进行混跑实验, 以检测这些计算单元之间的理论并行性.

4.3 并行性检测结果

我们使用混跑重叠率 $OverlapRate = (T_1 + T_2 - T_{colo})/(T_1 + T_2)$ 来表示任意两种硬件之间的并行性. 在这个等式中, T_1 和 T_2 分别表示两个基准应用单独执行的时间, T_{colo} 表示两个基准应用在 CUDA stream 混跑下的执行时间. 混跑重叠率的理论范围为 0 到 50%.

我们在 2080Ti, V100, 3090 三种硬件上进行了实验测试. 图 5 中分别展示了相应的实验结果. 从图 5(a) 可知, 在 2080Ti 上, FP64 Core 是辅助计算单元. Tensor Core, FP32 Core, INT32 Core 三种计算单元之间两两存在并行性. 从图 5(b) 可知, 在 V100 上, SM 内不存在 RT Core. Tensor Core, FP32 Core, INT32 Core 三种计算单元之间两两存在并行性. FP64 Core 与 FP32 Core, FP64 Core 与 INT32 Core 存在并行性. 从图 5(c) 可知, 在 3090 上, FP64 Core 是辅助计算单元. RT Core 与 Tensor Core, FP32 Core, INT32 Core 都存在理论并行性. Tensor Core 与 FP32 Core, INT32 Core 都存在理论并行性.

4.4 隐式假设及通用性

Hato 依赖三个隐式假设. 首先, 现有 GPU 都支持类 CUDA stream 的核函数管理接口, 例如 AMD ROCm 的 HIP stream, Intel OneAPI 的 kernel queue. 这些管理接口都支持, 在一个核函数没有使用完 SM(AMD GPU 的 CU, Intel GPU 的 Xe core) 上的计算资源时, 将另一个核函数被调度到 SM 上进行计算. 其次, 现有 GPU 都支持类 CUDA 的 SIMT 编程模式, 例如 AMD 的 ROCm, Intel 的 OneAPI. 而且, 编程人员可以通过使用这些编程语言, 而不需完全依赖黑盒第三方库来使用 GPU 内的具体计

算单元. 第三, Hato 能够提前感知该类 GPU 上可能存在的计算单元. 例如, Nvidia GPU 上可能包含着 Tensor Core, RT Core, FP32 Core, INT32 Core, FP64 Core 五种计算单元, AMD GPU 上可能包含着 Shader Core 与 Matrix Core 两种计算单元, Intel Xe Core 内同时包含着矢量引擎和矩阵引擎.

基于这三个隐式假设, 我们初步在 Nvidia 的 GPU 上进行了充分的实验验证. 而对于 AMD GPU 与 Intel GPU 而言, 我们受限于无法获得相应的实验环境, 并没有进行相应的实验验证. 然而, 从原理上进行分析, 由于 AMD ROCm 支持 HIP stream, 并且 AMD GPU 的基本计算单元 CU 内同时包含着 Shader core 与 Matrix Core, Hato 相应的设计思想可以用于定位 AMD GPU 的 CU 内并行性. 与此同时, 由于 Intel OneAPI 支持 kernel queue, 并且 Intel Xe Core 内同时包含着矢量引擎和矩阵引擎, Hato 相应的设计思想也可以用于定位 Intel Xe Core 内的计算并行性.

如果 Nvidia GPU 内加入了新的计算单元, 我们可以通过以下几步进行扩展. 第一, 我们检查新计算单元是否支持 CUDA stream 的使用. 如果新计算单元不支持 CUDA stream 的使用, 则它无法与其他计算单元进行并行执行. 第二, 我们检查新计算单元的使用模式, 确认其是否可以通过 CUDA 编程进行使用. 如果新计算单元仅仅支持黑盒第三方库进行使用, 则它也无法与其他计算单元进行并行执行. 如果新计算单元支持 CUDA 编程的方式进行使用, 我们准备新计算单元的基准应用. 第三, 将新计算单元的基准应用添加到硬件并行性感知子模块, 并行感知子模块在检测流处理器内并行性时, 考虑新计算单元与已有计算单元的并行性.

5 混跑建模子模块

在获知 GPU 上的计算并行性后, 本节中首先解决真实核函数在混跑时遭遇的资源竞争问题. 其次, 建模核函数混跑时的执行过程, 并构建核函数混跑情况下的性能预测模型.

5.1 解决资源竞争

众所周知, 当 SM 上存在足够的线程槽、共享内存以及寄存器时, 核函数可以将自己的线程块发射到 SM 上. 我们统计了在本篇文章用到的 17 个核函数的资源使用情况. 在这 17 个核函数中, 8 个使用了 100% 的线程槽, 8 个使用了超过 65% 的寄存器, 5 个使用了超过 85% 的共享内存. 对于一个核函数而言, 如果任意一种 SM 资源没有办法得到满足, 它就无法被调度到 SM 上. 也因此, 使用 CUDA stream 直接混跑两个核函数无法带来吞吐提升.

面对资源竞争问题, 我们使用持久线程块方法 (persistent thread block, PTB) 来管理核函数的资源使用 [16]. 对于原始编程模式, 核函数尝试发射足够多的线程块来掩盖 SM 上可能的计算间隔. 相反, 持久线程块模式仅仅发射有限个持久线程块. 每个持久线程块被视为一个线程块工作者. 它一直驻留在 SM, 循环完成多个原始线程块的计算.

通过调整持久线程块的数量, Hato 可以调整一个核函数的 SM 资源使用. 通过为每个核函数挑选合适的持久线程块数量, PTB 版本核函数可以达到原始性能的 85%, 甚至 95% 以上. 表 1 展示了所有核函数的持久化线程块数量, 资源使用情况, 以及对应的性能. 如表所示, 所有核函数在使用平均 43.3% 的线程槽, 平均 31.9% 的寄存器, 以及平均 9.8% 的共享内存情况下, 获得了平均 95.9% 的性能. 需要注意的是, PTB 版本核函数可能会带来相对于原始核函数版本的性能下降, 这是因为原始编程模式能够一直保持更高的并行度, 以及能够更好地利用硬件的自动负载均衡机制.

然而, 轻微的性能下降能够带来大幅度的资源释放, 这给探索 SM 内并行性带来了机会. 为此, 我们为每个核函数找到性能下降小于 10% 的持久线程块配置, 也即它们的 PTB 版本核函数. 在此基础

表 1 核函数使用持久化线程块技术后的资源使用及性能

Table 1 Resource usage and normalized performance of benchmark applications using the persistent thread block technique

	fcp	ffft	fmriq	fmrif	icp	ifft	dmriq	dmrif	tgemm	tracer	cutouts	whitted	blur
PTB block num.	6	3	4	3	6	3	4	3	2	1	1	1	1
Thread slots (%)	75	37.5	100	75	75	37.5	50	37.5	25	12.5	12.5	12.5	12.5
Registers (%)	45.7	19.3	48.3	37.5	46.8	21.1	78.1	63.3	41.8	3.13	3.13	3.13	3.13
Shared mem. (%)	0	9.6	0	0	0	9.6	0	0	57.6	12.5	12.5	12.5	12.5
Normalized perf. (%)	81.7	98.5	85.4	96.5	98.3	98.7	95.6	100	97.5	99.1	98.7	99.2	98.5

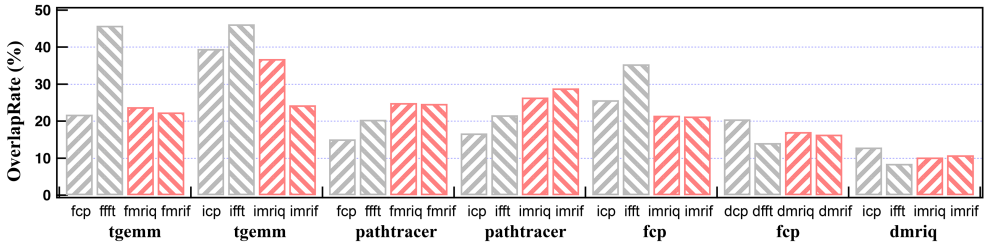

图 6 (网络版彩图) 解决掉资源竞争之后混跑重叠率

Figure 6 (Color online) Overlap rate after solving the resource contention using the PTB technique

上, 我们重新运行第 2.2 节的混跑实验. 在该实验中, 包含 FP64 kernel 的混跑实验在 V100 上进行, 包含 RT kernel 的混跑实验在 3090 上进行, 而其他核函数的混跑实验在 2080Ti 上执行. 图 6 展示了相应的实验结果. 如图所示, 所有核函数混跑对获得了平均 22.8% 的吞吐提升. 这是因为, 当解决完核函数在 SM 上的资源竞争之后, 使用不同计算单元的核函数能够在 SM 上同时计算, 而这带来了整体的系统吞吐提升.

需要注意的是, 理论并行性并不意味着实际并行性. 这是因为, 那儿很可能有大量的隐式资源竞争, 例如缓存和总线带宽. 举例而言, 即使我们解决了相应的资源竞争, 我们在 3090 上没有获得 RT Core 与 Tensor Core 的实际并行性. 除了 3090 上的 RT Core 与 Tensor Core 之外, 其他 SM 内的理论并行性都获得了实际并行性.

5.2 性能预测建模

由上一小节可知, 两个核函数在混跑执行时需要比单跑执行时花费更长的执行时间. 为了保证延迟敏感型任务的服务质量, Hato 需要对核函数混跑对的执行时间进行精准预测. 与此同时, 为了支持最大化吞吐的调度策略设计, Hato 仍然需要对核函数混跑对的混跑执行过程进行精准建模.

假设有一个核函数 K_1 和一个核函数 K_2, 我们使用 T_1 表示核函数 K_1 的单跑执行时间, 使用 T_2 表示核函数 K_2 的单跑执行时间. 两个核函数的混跑执行时间 T_{colo} 完全取决于两个核函数的计算特征以及计算量. 一个核函数的计算特征包括计算访存比例, 控制流等因素. 与此同时, 一个核函数的单跑执行时间同样取决于该核函数的计算特征以及计算量. 由于一个核函数的单跑执行时间已经比较好的表达了其计算特征以及计算量, 因此我们可以得出, 两个核函数的混跑执行时间 T_{colo} 与两个核函数的单跑执行时间 T_1 和 T_2 有关.

为了简化两个变量带来的复杂影响, 我们提出了一个新的指标: 计算量比例 $ComputeRatio = T_2/T_1$, 来表示两个变量之间的关系. 基于此, 分析实验可以分为两个子实验: 在核函数 K_1 计算量

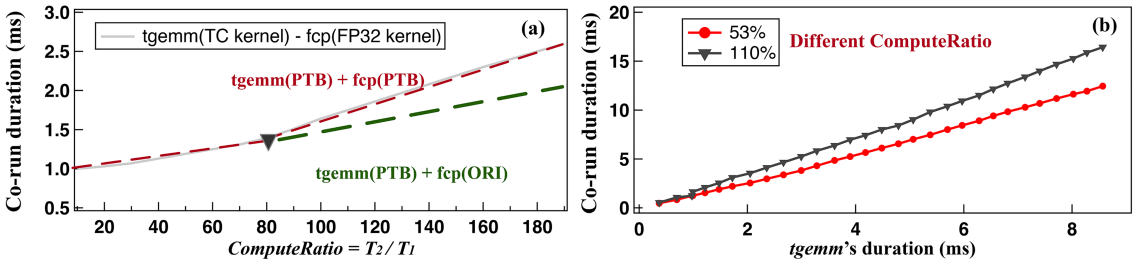

图 7 (网络版彩图) 核函数混跑对 *tgemm-cp* 的性能预测模型

Figure 7 (Color online) Duration prediction for the kernel pair *tgemm-cp*. (a) The first experiment with *tgemm-cp*; (b) the second experiment with *tgemm-cp*

不变的情况下, 探索两个核函数混跑时间 T_{colo} 与核函数 K_2 计算量 (也即单跑计算时间 T_2) 的关系; 在计算量比例 T_2/T_1 不变的情况下, 探索两个核函数混跑时间 T_{colo} 与核函数 K_2 计算量 (即单跑计算时间 T_2) 的关系.

对于这两个子实验, 我们选择了 *tgemm* 作为核函数 K_1, *cp* 作为核函数 K_2. 图 7(a) 展示了 *tgemm-cp* 在第一个子实验下的执行时间. X 轴表示两个核函数的计算量比例. 由于核函数 K_1 的计算量不变, 我们通过逐渐增加核函数 K_2 的计算量来增加计算量比例, Y 轴表示两个核函数在混跑情况下的完成时间, 而这个时间基于核函数 K_1 的计算时间进行归一化. 从图中可知, 这个性能曲线展现了一个两段线性关系.

图 7(b) 展示了 *tgemm-cp* 在第二个子实验下的执行时间. 不同的曲线表示了不同的计算量比例, X 轴表示变化的核函数 K_2 计算时间, Y 轴仍然表示两个核函数在混跑情况下的完成时间. 从图可知, 当计算量比例确定的情况下, 核函数混跑对的混跑时间与核函数 K_2 的单跑时间呈线性关系.

面对以上计算结果, 我们进一步分析了图 7(a) 的曲线变化过程. 随着计算量比例的增加, 混跑执行时间先以一个缓慢的斜率增加. 经过一个拐点之后, 两个核函数的执行时间再以一个陡峭的斜率增加. 我们发现, 陡峭的斜率等于核函数 K_2 单跑的斜率. 这意味着, 在拐点之后, 两个核函数的混跑可以划分为两个阶段: 两个核函数的并行执行, 以及一个核函数的单独执行.

基于以上分析, 我们可以得出两个结论. (1) 两个核函数在混跑情况下的执行时间能够通过一个两段线性模型进行预测. (2) 两个核函数在混跑情况下的执行过程分为两个阶段: 一个是两个核函数的并行执行, 一个是一种核函数的单独执行. 进一步的实验表明, 以上结论适用于 3 种 GPU 上的所有 SM 内并行性.

为了支持运行时的性能预测, Hato 需要在离线为每个可能核函数混跑对构建其性能预测模型. 具体而言, Hato 首先基于线性回归模型, 为每个核函数构建其单跑情况下的性能预测模型. 其次, Hato 为每个核函数混跑对收集 20 个计算量比例的数据点, 为其构建相应的两段线性回归模型. 而在运行时, Hato 定期收集每个核函数混跑对的执行时间, 来微调其模型参数, 以保证高预测精度.

6 吞吐导向调度子模块

6.1 调度低效问题

混跑建模子模块首先为所有的核函数配置合适的 PTB 版本, 然后基于这些核函数的 PTB 版本, 为所有可能的核函数混跑对构建相应的性能预测模型. 然而, 基于以上的核函数混跑调度带来了非常

有限的吞吐提升. 面对这个问题, 我们进一步研究了核函数混跑对的混跑重叠率. 对于图 7(a) 的实验, 我们发现在拐点之后, 随着计算量比例的增大, 混跑重叠率可能会下降到负数.

进一步分析发现, 由于两个核函数都使用了 PTB 版本的核函数, 它们都可能遭遇了轻微的性能下降. 而当一个核函数的计算量比较大时, 其单独执行带来的性能下降也在逐渐变大. 因此, 当单独执行阶段的性能下降超过了并行执行阶段的性能提升后, 混跑重叠率出现了负数.

面对以上问题, 进一步通过实验发现, 仅仅调整第一个核函数的资源使用就可以获得使用 SM 内并行性的机会. 这意味着, 仅仅需要调整第一个核函数为 PTB 版本, 而第二个核函数仍然可以使用 ORI 版本 (原始版本). 通过这样的调整, 图 7(a) 中的绿色线展示了 *tgemm-cp* 核函数混跑对在不同计算量比例下的执行时间. 从图中可以看出, 核函数混跑对获得了相当的性能提升. 实验结果表明, 即使核函数混跑对的混跑重叠率仍然有同样的变化趋势, 它一直处于理论范围内. 如果两个核函数以 PTB+ORI 模式进行混跑, 确实有可能出现 ORI 版本核函数先发射导致无法利用到 SM 内并行性的问题. 实验发现, 如果将 PTB 版本核函数放置在 stream 0 中, 将 ORI 版本核函数放置在 stream 1 中, 就可以实现 PTB 版本核函数的优先发射.

6.2 调度选择难题

除了调度低效问题, 现有调度子模块还面临着调度选择难题. 由于 SM 内拥有很多计算单元, 一个 GPU 可能拥有多种 SM 内并行性. 举例而言, 2080Ti 上同时拥有 Tensor-FP32 并行性、Tensor-INT32 并行性和 FP32-INT32 并行性. 由于不同的计算单元有不同的峰值计算速度, 而且同一个核函数混跑对在不同负载下也有不同的性能提升, 如何制定一个通用的调度策略来选择合适的核函数混跑对进行混跑, 仍然是一件非常困难的事情.

面对这个难题, 我们发现, 即使不同的计算单元有不同的算力, 但是一个核函数无法更改它要使用的计算单元. 这意味着, 多个使用不同计算单元的核函数混跑对于 GPU 而言, 在时间维度上是等价的. 例如, 一个 FP32 kernel 和一个 INT32 kernel 分别需要 5 ms 完成计算. 如果没有混跑执行带来的并行可能, 这两个核函数都需要 GPU 花费 5 ms 完成计算. 因此, 可以使用混跑减少时间 $MakespanReduction = T_1 + T_2 - T_{colo}$ 来表示任意一个核函数混跑对在任意负载下带来的吞吐收益. 在这个等式中, T_1 和 T_2 表示任意两个核函数的执行时间. 当多个核函数混跑对都可能带来吞吐提升时, Hato 可以根据当前情况下每个核函数混跑对带来的混跑减少时间来判断最优的调度选择.

6.3 QoS 感知的任务调度

延迟敏感型任务的执行时间划分. 一个延迟敏感型任务的执行时间是它的第一个核函数发射的时间点与最后一个核函数结束的时间点之间的时间间隔. 如图 8 所示, 一个延迟敏感型任务的端到端时间 T_e2e 由 4 个部分组成: (1) 已经被发射的核函数的执行时间 T_{queue}; (2) 当前延迟敏感型任务的执行时间 T_{lc}, 该时间也是它的 TC kernel(T_{L-TC_1}, ...), FP32 kernel(T_{L-FP32_1}, ...), INT32 kernel($T_{L-INT32_1}$, ...) 等核函数的整体完成时间; (3) 混跑情况下的执行时间 T_{colo}; (4) 尽力而为型任务中核函数的计算时间 T_{be}, 这个时间具体由它的核函数执行时间构成 (T_{B-FP32_i}, $T_{B-INT32_j}$).

具体调度流程. 图 8 展示了一个延迟敏感型任务和多个尽力而为型任务的调度过程. 假设 T_{qos} 是延迟敏感型任务的服务质量目标, T_{e2e} 是一个延迟敏感型任务的端到端执行时间. 仅仅当不等式 $T_{e2e} = T_{queue} + T_{lc} + T_{colo} + T_{be} \leqslant T_{qos}$ 满足时, 延迟敏感型任务的服务质量才能够得到满足.

在运行时调度时, 调度子模块首先需要计算当前延迟敏感型任务的计算时间余量 $T_{headroom}$, 也即完成延迟敏感型任务计算后剩余给尽力而为型任务计算的时间. 具体而言, 计算时间余量可以基于等

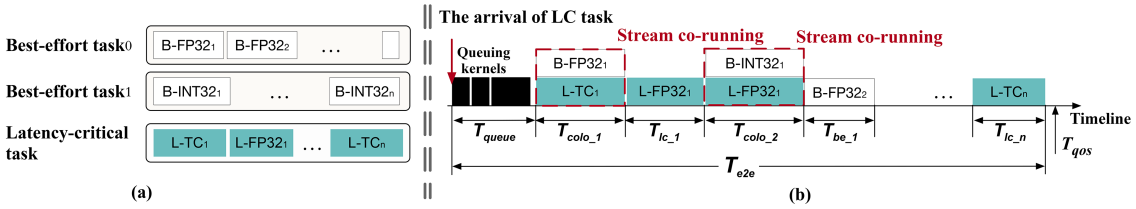

图 8 (网络版彩图) 使用 Hato 的调度流程

Figure 8 (Color online) Runtime scheduling with Hato. (a) Kernels in tasks; (b) scheduling process

式 $T_{headroom} = T_{qos} - T_{lc} - T_{queue}$ 计算得出.

每个核函数在发射之前, Hato 都会基于相应的性能预测模型进行核函数的执行时间预测, 因此 T_{queue} 和 T_{lc} 是已知的. 在这个等式中, 我们通过减去排队核函数的执行时间以及延迟敏感型任务的单跑执行时间, 计算出延迟敏感型任务的最初计算时间余量 $T_{headroom}$. 基于这个计算时间余量, 每次延迟敏感型任务的一个核函数被发射, Hato 遍历所有尽力而为型任务的已经准备好的核函数, 来检查是否有混跑提升系统吞吐的机会.

假设当前延迟敏感型任务的核函数是一个 TC kernel, 它的预测单跑执行时间是 T_{tc}, 一个 FP32 kernel 和一个 INT32 kernel 处于准备好执行的状态. Hato 首先预测 TC kernel 分别与 FP32 kernel 和 INT32 kernel 混跑时的执行时间 $T_{tc-fp32}$ 与 $T_{tc-int32}$. 其次, Hato 判断这两个时间是否大于计算时间余量 $T_{headroom}$. 如果大于 $T_{headroom}$, 则可能导致延迟敏感型任务违反它的服务质量目标.

如果两个混跑选择都不会带来服务质量违反, Hato 进而计算出两种混跑选择下的混跑减少时间. 基于混跑减少时间的比较, Hato 选择那个有最大吞吐提升的混跑选择.

如果当前时刻所有可能的核函数混跑对都无法带来吞吐收益, Hato 也会选择核函数重排序技术. 如果当前尽力而为型任务的核函数执行不会带来延迟敏感型任务的服务质量, Hato 则将尽力而为型任务的核函数发射到 GPU 上.

7 实验

7.1 实验配置及实现

实验配置. 我们选用 4 种常用的 DNN 推理服务 ($Bert$, $Resnet50$, $Vgg16$, $Inception3$) 和 4 种云游戏服务 ($SeveredSteel$ 简称为 SS, $DUTM$, $Quake2$, $Ascent$) 作为延迟敏感型任务. 我们从广泛使用的 GPU 基准测试集 Parboil [11] 中随机选用 4 种使用 FP32 Core 的应用 (fcp, fft, $fmriq$, $fmrif$), 4 种使用 FP64 Core 的应用 (dcp, $dfft$, $dmriq$, $dmrif$), 以及 4 种使用 INT32 Core 的应用 (icp, $ifft$, $imriq$, $imrif$) 作为尽力而为型任务. 每种 DNN 推理服务都有 2 种实现, 这 2 种实现分别使用 Tensor Core 和 FP32 Core 作为主要计算单元. 而云游戏服务主要使用 RT Core 和 FP32 Core. 我们使用 50 ms 作为 DNN 推理服务的服务质量目标, 使用 120FPS 作为云游戏服务的服务质量要求, 也即服务质量目标为 8.3 ms. 所有延迟敏感型任务以泊松分布到来. 所有的实验在 2080Ti, V100, 3090 三种 GPU 上执行. Hato 不依赖于特定的硬件特征, 可以适配于任意 GPU.

实现. 我们从零构建了 Hato 系统, 并能够拿到所有任务的 CUDA 源码. 为此, 我们使用 DNN 编译器 Rammer [17] 生成了 DNN 推理服务的源码, 使用 Nvidia 官方的光线追踪代码来模拟了云游戏服务, 并使用 Parboil 获取了尽力而为型任务的源码. 如果 DNN 推理服务使用 Pytorch 进行服务, 我们

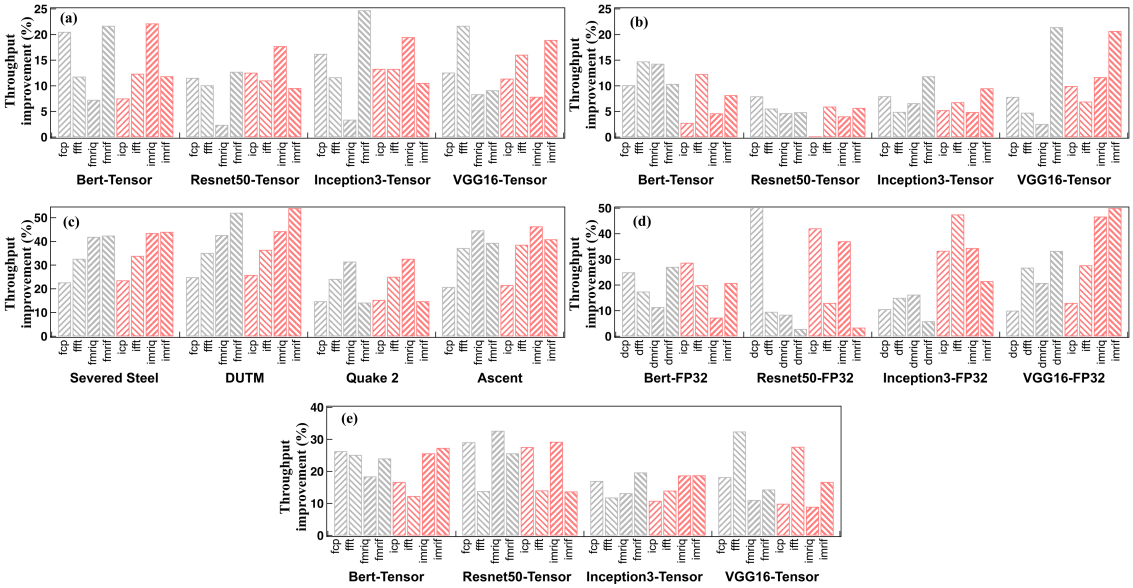

图 9 (网络版彩图) Hato 相比于 Tacker 的吞吐提升

Figure 9 (Color online) Throughput improvement of Hato compared with Tacker. (a) Co-locating DNN inference and best-effort tasks on 2080Ti; (b) co-locating DNN inference using tensor core and best-effort tasks on V100; (c) co-locating cloud gaming and best-effort tasks on 3090; (d) co-locating DNN inference using FP32 Core and best-effort tasks on V100; (e) co-locating DNN inference and best-effort tasks on 3090

可以基于 API 拦截技术进行支持. 具体而言, 我们可以拦截到 Pytorch 向 GPU 发射的具体核函数. 通过对拦截到的核函数进行分类以及识别, 我们可以支持使用不同计算单元核函数的混跑, 以利用到 GPU 上的 SM 内并行性.

7.2 吞吐提升

在本节的 5 个子实验中, 我们都使用 Tacker 作为对比系统. Tacker 首先尝试融合两个核函数以利用到 SM 内的并行性, 如果不能支持两种核函数的融合, Tacker 仅仅利用延迟敏感型任务的计算时间余量来运行尽力而为型任务. 虽然 Tacker 仅仅支持了 TC kernel 与 FP32 kernel 的融合, 我们仍然扩展它支持了 TC kernel 与 INT32 kernel, FP32 kernel 与 INT32 kernel 的融合. 基于此, 我们使用吞吐提升 $ThroughputImprovement = (T_{hato} - T_{tacker})/T_{tacker}$ 来表示 Hato 相比于 Tacker 的吞吐提升. T_{hato} 和 T_{tacker} 表示两个系统执行尽力而为型任务的计算时间.

实验 1. 在 2080Ti 上混跑 DNN 推理服务与尽力而为型任务. 在这个子实验中, DNN 推理服务主要使用 Tensor Core, 而尽力而为型任务会使用 FP32 Core 或者 INT32 Core. 如图 9(a) 所示, Hato 相比 Tacker 获得了平均 13.2%, 最高 24.7% 的吞吐提升.

实验 2. 在 V100 上混跑 DNN 推理服务与尽力而为型任务. 在这个子实验中, DNN 推理服务主要使用 Tensor Core, 而尽力而为型任务会使用 FP32 Core 或者 INT32 Core. 如图 9(b) 所示, Hato 相比 Tacker 获得了平均 8.2%, 最高 21.4% 的吞吐提升.

实验 3. 在 3090 上混跑云游戏服务与尽力而为型任务. 在这个子实验中, 云游戏服务主要使用 RT Core, 而尽力而为型任务会使用 FP32 Core 或者 INT32 Core. 如图 9(c) 所示, Hato 相比 Tacker 获得了平均 32.2%, 最高 54.1% 的吞吐提升.

实验 4. 在 V100 上混跑 DNN 推理服务与尽力而为型任务. 在这个子实验中, DNN 推理服务主

要使用 FP32 Core, 而尽力而为型任务会使用 FP64 Core 或者 INT32 Core. 如图 9(d) 所示, Hato 相比 Tacker 获得了平均 23.1%, 最高 48.7% 的吞吐提升.

实验 5. 在 3090 上混跑 DNN 推理服务与尽力而为型任务. 在这个子实验中, DNN 推理服务主要使用 FP32 Core, 而尽力而为型任务会使用 FP64 Core 或者 INT32 Core. 如图 9(e) 所示, Hato 相比 Tacker 获得了平均 19.5%, 最高 32.1% 的吞吐提升.

通过以上实验配置, 我们在实验中仅仅没有涵盖到 2080Ti 上的 FP32-INT32 并行性与 V100 上的 FP64-INT32 并行性. 虽然 FP32-INT32 并行性在 2080Ti 上没有测试, 但是该并行性在 V100 上进行了相应验证. 此外, 对于 FP64-INT32 并行性, 目前没有合适的混合部署场景进行相应验证.

从以上结果可知, Hato 在所有硬件上的所有混跑情况下都相较于 Tacker 获得了更好的性能. 在 2080Ti 和 V100 上的混跑配置 (子实验 1, 2, 4), Hato 更好的性能可以归结于更好的资源使用以及更好的运行时管理. 由于核函数融合会带来相应的资源浪费, 而 Hato 的核函数混跑方法不会带来资源浪费, Hato 能够在混跑情况下在 SM 上发射更多的线程块. 因此, Hato 能够获得更好的吞吐提升. 由于 Hato 能够确保所有情况下都能获得混跑的正收益, 而核函数融合方法不能保证这一点, 所以 Hato 能够获得更好的吞吐. 此外, 在 3090 上的混跑实验, Hato 更好的性能来源于 Hato 能够处理 RT kernel 与其他 kernel 的混跑情况, 而 Tacker 的核函数融合不能支持两种编程语言的融合. 因此, Hato 在 3090 上也能获得更好的吞吐性能.

由于核函数混跑对 *tgemm-fmrif* 相比于核函数混跑对 *tgemm-icp* 有更高的混跑重叠率, *tgemm-fmrif* 在 2080Ti 和 V100 上都获得了相比于 *tgemm-icp* 更高的吞吐提升. *fmrif*, *ffft*, 相比于 *fcp* 与 *fmriq* 一直能够带来更高的混跑重叠率, 更高的系统吞吐. 与此同时, 运行时的系统提升仍然取决于任务的到达模式. 所有的任务都以泊松分布到来, 这可能带来一些微小的性能波动.

对于云游戏服务而言, 我们使用 4 种不同的 RT kernel 来模拟 4 种云游戏的负载. 因此, 云游戏服务场景下 Hato 相对于 Tacker 的性能提升, 完全取决于 4 种 RT kernel 与混跑核函数的混跑重叠率. 此外, 由于 Tacker 无法支持云游戏服务场景下的核函数融合, 其无法利用到该场景下的 SM 内并行性. 因此, Hato 相比于 Tacker 有最高的系统吞吐提升.

7.3 保证服务质量

在以上的 5 个子实验中, 我们进一步统计了所有延迟敏感型服务的 99% 完成时间. 图 10(a) 展示了在 $8 \times 8 = 64$ 个混跑配置下的延迟敏感型任务的 99% 完成时间 (一个延迟敏感型任务的多次计算时间中, 从小到大排序第 99% 分位的计算时间). 如图所示, Hato 在所有混跑配置下都保证了延迟敏感型任务的服务质量. 这是因为, Hato 运行时根据延迟敏感型任务的服务质量余量来决定是否进行核函数混跑. 如果延迟敏感型任务可能遭遇服务质量违反, Hato 则退化成单独执行.

7.4 性能预测精度

我们评估 Hato 中两段线性模型来预测核函数混跑执行时间的有效性. 由于页面限制, 图 10(b) 仅仅展示了部分的实验结果. 不过没有展示出的实验结果也和图 10(b) 中的结果有类似的结果. 如图所示, 混跑时间的预测值与实际值最多相差 6.5%, 平均预测误差小于 1.4%. 因此, Hato 能够使用两段线性回归模型来预测两个核函数的混跑执行时间.

7.5 混跑性能提升

我们评估了 Hato 混跑方法相对于 Tacker 核函数融合方法的性能提升. 我们使用 TC kernel *tgemm*

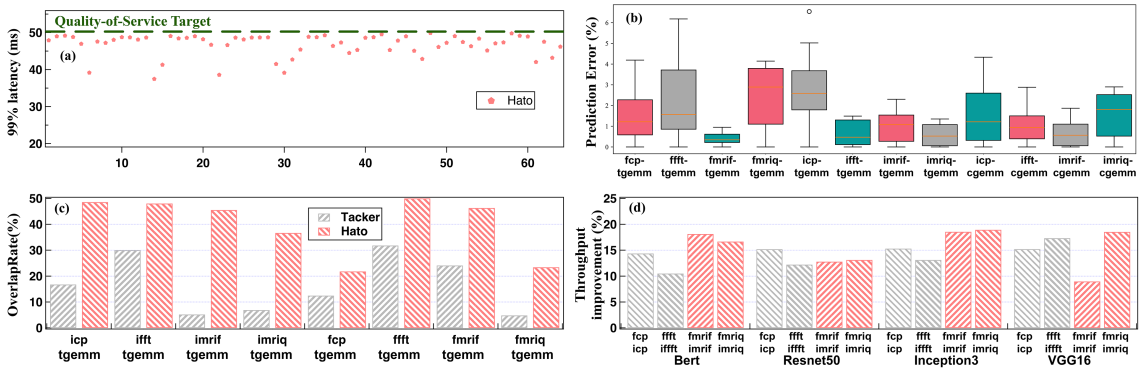

图 10 (网络版彩图) Hato 各个子模块的有效性

Figure 10 (Color online) Effectiveness of different modules in Hato. (a) Quality-of-service guarantee; (b) prediction accuracy of two-stage linear regression models; (c) performance improvement of Hato compared with kernel fusion; (d) results for the co-location with multiple applications

与 4 种 FP32 kernel 和 4 种 INT32 kernel 进行混跑实验. 图 10(c) 展示了相应的实验结果. 如图所示, Hato 混跑方法相对 Tacker 融合方法, 平均提升 23.7%, 最高提升 38.5% 的系统吞吐. 这是因为, 核函数融合方法带来了相当的资源浪费. 额外的资源使用导致核函数融合方法不能发射足够的线程块到 SM 上, 这也因此带来了核函数融合方法的低性能.

7.6 多个任务混跑

我们通过一个延迟敏感型任务与多个尽力而为型任务的混跑来进一步验证 Hato 的有效性. 具体而言, 我们同时使用 FP32 Core 的应用, 与使用 INT32 Core 的应用来与一个 DNN 推理服务进行混跑. 图 10(d) 展示了相应的实验结果. 如图所示, Hato 相对 Tacker 平均提升 14.9%, 最高提升 18.9% 的吞吐提升. 这是因为, Hato 一直能够选择有最优吞吐提升的核函数进行混跑, 而 Tacker 仅仅能够选择最先到来的核函数进行核函数融合.

7.7 实验开销

Hato 的实验开销来自于两方面: 并行性感知子模块以及性能建模子模块. 对于并行性感知子模块, 每个基准应用仅仅需要运行 10 ms, 5 种基准应用仅仅需要 50 ms, 而 5 种基准应用的两两混跑需要 $C_5^2 \times 20 = 200$ ms. 并行性感知子模块最多需要 250 ms 的计算开销. 对于性能建模子模块, 假设有 50 个尽力而为型任务, 10 个延迟敏感型任务, 每个核函数混跑对混跑需要 10 ms, 则总共需要 $50 \times 10 \times 10 = 5000$ ms. 因此, Hato 的实验开销是可接受的.

8 相关工作

在数据中心上进行任务的混合部署以提升系统吞吐是研究人员一直非常关注的研究方向. 混合部署领域主要包括两个核心的研究子方向: 服务质量管理和纯粹吞吐提升.

很多相关工作都关注于在保证延迟敏感型任务的服务质量的同时提升整体系统吞吐. Baymax [5] 基于英伟达官方调度接口 MPS, 通过重新排序核函数的发射, 以支持尽力而为型任务使用延迟敏感型任务没有使用完的 GPU 时间片. Rollover [18] 在硬件层支持延迟敏感型任务抢占尽力而为型任务占据的 GPU 资源, 以保证延迟敏感型任务的服务质量. 这些工作都仅仅关注于 GPU 层次, 也就是 SM 粒

度上的时间片利用效率. 它们并不关注于 SM 内的计算单元使用效率, 因此无法用于利用 SM 内并行性提升整体系统吞吐. 此外, Tacker [6] 通过核函数融合的方法, 将 TC kernel 与 FP32 kernel 融合成一个新的核函数. 由于这个核函数能够同时将使用 Tensor Core 与 FP32 Core 的计算部分发射到一个 SM 上, 它能够支持两种计算单元的同时使用. ISPA [19] 和 Plasticine [20] 尝试调整核函数在流处理器上使用的线程槽、共享内存、寄存器数量, 以最大化一个核函数混跑对的执行性能. 然而, Tacker, ISPA 和和 Plasticine 都假设两种计算单元的并行性是已知的, 而这个信息是无法轻易得到的. 与此同时, 以上 3 个工作都无法处理存在多种 SM 内并行性的情况. 因此, 现有工作都无法带来最优的系统吞吐.

还有很多相关工作关注于通过混合部署多个尽力而为型任务来提升系统吞吐. Mastro [21] 通过考虑动态任务信息, 运行时自适应地调整多任务的执行模式. SMK [22] 通过为 GPU 添加线程块粒度的切换机制, 支持 GPU 进行更细粒度的核函数调度. 在这些关注于核函数粒度的调度优化工作之外, 另外一些工作则关注于核函数的资源管理. 在文献 [23] 中, 相关研究人员通过内存带宽的资源管理, 实现整体吞吐的调度优化. 而在文献 [24] 中, 相关研究人员通过内存使用的管理, 提升整体的系统吞吐. 除了以上工作之外, 很多研究工作基于 GPU 模拟器进行了相应的研究 [25, 26]. 它们通过从硬件层获取硬件计数器信息来进行相应的性能预测与资源管理. 由于它们都依赖于 GPU 模拟器, 在真实 GPU 上都是不可用的, 同时它们也都没有考虑到现有 SM 内的多种计算单元.

9 总结

目前 GPU 内可能包含着多种计算单元, 例如 FP32 Core, INT32 Core, FP64 Core, Tensor Core, 以及 RT Core. 任意一种 GPU 可能包含着以上单元的部分计算单元. 虽然 GPU 内流处理器内包含着多种计算单元, 这些信息是无法直接获知的. 而且, 现有调度接口无法直接利用到这些计算资源的并行性. 为此, 我们提出了硬件感知吞吐导向的核函数调度方法 Hato. Hato 能够通过一套并行性感知工作, 为任意 GPU 定位出所有的 SM 内并行性. 其次, Hato 解决了核函数在 SM 上的资源竞争, 并为核函数混跑对构建了混跑性能预测模型. 最后, Hato 通过一个吞吐导向的调度策略, 支持在保证延迟敏感型任务服务质量的同时, 利用到所有的 SM 内并行性, 提升整体系统吞吐. 实验结果表明, Hato 相比最新调度系统 Tacker 提升了平均 19.2%, 最高 54.1% 的系统吞吐.

参考文献

1 Nvidia. Volta architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

2 Nvidia. Turing architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. Accessed: April 20, 2024

3 Zou Z X, Chen K Y, Shi Z W, et al. Object detection in 20 years: A survey. In: Proceedings of the IEEE, 2023. 257–276

4 Li P L, Luo Y, Zhang N, et al. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. In: International Conference on Networking, Architecture and Storage, 2015. 347–348

5 Chen Q, Yang H L, Mars J, et al. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. ACM SIGPLAN Notices 51, 2016: 681–696

6 Zhao H, Cui W H, Chen Q, et al. Tacker: Tensor-cuda core kernel fusion for improving the gpu utilization while ensuring QoS. In: International Symposium on High-Performance Computer Architecture, 2022. 800–813

7 Nvidia. Optix Ray Tracing Engine. https://developer.nvidia.com/rtx/ray-tracing/optix

8 Nvidia. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit

9 Nvidia. CUDA stream. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

10 Nvidia. Tensor Core example code. https://github.com/NVIDIA/cuda-samples/tree/master/Samples/cudaTensorCoreGemm

11 Stratton J, Rodrigues, Sung I, et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Center for Reliable and High-Performance Computing, 2012, 72: 127

12 Nvidia. CUDA samples. https://github.com/NVIDIA/cuda-samples

13 Nvidia. Optix toolkit. https://github.com/NVIDIA/optix-toolkit

14 Wang Z N, Yang J, Melhem R, et al. Quality of service support for fine-grained sharing on GPUs. In: International Symposium on Computer Architecture, 2017. 269–281

15 Chen Q, Yang H L, Guo M Y, et al. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In: International Conference on Architectural Support for Programming Languages and Operating Systems, 2017. 17–32

16 Wu B, Chen G Y, Li D, et al. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In: International Conference on Supercomputing, 2015. 119–130

17 Ma L X, Xie Z Q, Yang Z, et al. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In: 14th USENIX Symposium on Operating Systems Design and Implementation, 2020. 881–897

18 Wang Z N, Yang J, Melhem R, et al. Quality of service support for fine-grained sharing on GPUs. In: Proceedings of the 44th Annual International Symposium on Compute Architecture, 2017. 269–281

19 Zhao H, Cui W H, Chen Q, et al. ISPA: Exploiting intra-SM parallelism in GPUs via fine-grained resource management. IEEE Trans Comput, 2022, 72: 1473–1487

20 Zhao H, Cui W H, Chen Q, et al. Exploiting intra-SM parallelism in GPUs via persistent and elastic blocks. In: IEEE 39th International Conference on Computer Design, 2021. 290–298

21 Park J K, Park Y J, Mahlke S. Dynamic resource management for efficient utilization of multitasking GPUs. In: International Conference on Architectural Support for Programming Languages and Operating Systems, 2017. 527–540

22 Wang Z N, Yang J, Melhem R, et al. Quality of service support for fine-grained sharing on GPUs. In: 2016 IEEE International Symposium on High Performance Computer Architecture, 2016. 358–369

23 Wang H N, Luo F, Ibrahim M, et al. Efficient and fair multi-programming in GPUs via effective bandwidth management. In: International Symposium on High-Performance Computer Architecture, 2018. 247–258

24 Jog A, Kayiran O, Kesten T, et al. Anatomy of gpu memory system for multi-application execution. In: International Symposium on Memory Systems, 2015. 223–234

25 Hu Q D, Shu J W, Fan J, et al. Run-time performance estimation and fairness-oriented scheduling policy for concurrent GPGPU applications. In: International Conference on Parallel Processing, 2016. 57–66

26 Zhao W Y, Chen Q, Lin H, et al. Themis: Predicting and reining in application-level slowdown on spatial multitasking GPUs. In: International Parallel and Distributed Processing Symposium, 2019. 653–663

Exploiting all intra-SM parallelism to maximize the throughput while ensuring QoS

Han ZHAO[1,4], Junxiao DENG[1], Weihao CUI[1], Quan CHEN[1*], Deze ZENG[2], Jing YANG[3] & Minyi GUO[1]

1. *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai* 200240*, China*;
2. *School of Computer Science, China University of Geosciences, Wuhan* 430074*, China*;
3. *School of Mechanical Engineering, Guizhou University, Guiyang* 550025*, China*;
4. *Shanghai Key Laboratory of Financial Information Technology (Shanghai University of Finance and Economics), Shanghai* 200433*, China*

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

Abstract To address the burgeoning demand for computational capacity, GPUs incorporate an array of both general-purpose and specialized computing units, including FP32 Core, INT32 Core, FP64 Core, Tensor Core, and RT Core, within their streaming multiprocessors (SM). Various types of GPUs may encompass a subset of these computing units. Despite the presence of multiple computing units within the SM, the parallelism among them is not elucidated in the hardware design documentation. Concurrently, official scheduling interfaces lack the capability to facilitate the parallel utilization of these computing resources by co-running the kernels using different computing units. Also, they could not support the runtime scheduling to optimize overall system throughput. Faced with the above problems, we propose a hardware-aware throughput-oriented kernel scheduling method Hato. Hato first designs a parallelism-aware tool that supports finding all intra-SM parallelism for any GPU. Secondly, Hato proposes a kernel co-running modeling method, which supports the existing scheduling interfaces to utilize the intra-SM parallelism, and the accurate duration prediction for the co-running kernels. Finally, Hato proposes a throughput-oriented scheduling strategy that supports utilizing all possible intra-SM parallelism to maximize overall system throughput while ensuring service quality for latency-sensitive applications. Experimental results show that compared with the state-of-the-art scheduling system Tacker, Hato improves the system throughput by an average of 19.2% and by as much as 54.1%.

Keywords GPU, intra-SM parallelism, throughput improvement, runtime system