

申请上海交通大学博士学位论文

面向异构数据中心的新型硬件架构与调度算法

论文作者 王振宁  
学 号 0120339012  
导 师 过敏意教授  
专 业 计算机科学与技术  
答辩日期 2017 年 6 月 1 日



Submitted in total fulfillment of the requirements for the degree of Doctor  
in Computer Science and Technology

# The Architecture Designs and Scheduling Algorithms for Heterogeneous Datacenters

ZHENNING WANG

Advisor

Prof. MINYI GUO

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING,  
SCHOOL OF ELECTRONIC INFORMATION AND ELECTRICAL ENGINEERING  
SHANGHAI JIAO TONG UNIVERSITY  
SHANGHAI, P.R.CHINA

Jun. 1st, 2017



## 上海交通大学 学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：\_\_\_\_\_

日期：\_\_\_\_\_年\_\_\_\_\_月\_\_\_\_\_日



## 上海交通大学 学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于

保 密口，在\_\_\_\_\_年解密后适用本授权书。  
不保密口。

(请在以上方框内打√)

学位论文作者签名: \_\_\_\_\_

指导教师签名: \_\_\_\_\_

日 期: \_\_\_\_\_ 年 \_\_\_\_ 月 \_\_\_\_ 日

日 期: \_\_\_\_\_ 年 \_\_\_\_ 月 \_\_\_\_ 日



# 面向异构数据中心的新型硬件架构与调度算法

## 摘要

随着云计算技术的发展，作为云计算支撑平台的数据中心中的体系结构与调度问题已经成为了当前计算机科学研究的重要领域。与传统的高性能计算集群不同，数据中心需要同时为多个用户提供服务，而用户所运行的延迟敏感型（Latency-Sensitive, LS）应用对其服务质量（Quality of Service, QoS）有着特定的需求。传统数据中心使用多核 CPU 进行计算，并且其中的应用以负载稳定的单线程应用为主，这些应用调度方法已有很多研究。随着处理器技术和用户应用的发展，数据中心中大量采用了如通用图形处理单元（GPGPU）等加速器以提高数据中心的功耗比和性能，从而形成了同时使用多种不同架构处理器的数据中心，即异构数据中心。然而，异构数据中心里出现了四个新的问题：首先，通用图形处理单元最初是为高性能计算的独占式服务而设计的，在硬件架构上不能完全适应数据中心的多用户共享的需求。如何修改 GPU 的硬件架构以满足数据中心的这一需求成为了一个重要的问题。其次，GPU 并不具有硬件级的 QoS 支持。这也让数据中心在使用 GPU 运行 LS 应用时不能充分利用硬件资源，降低了系统性能。同时，异构处理器也带来了使用 GPU，或是同时使用 CPU 和 GPU 的异构任务。这些异构任务也对调度算法提出了新的挑战。最后，负载可变的多线程新型应用的使用逐渐增长，而传统的数据中心调度算法是为负载稳定的单线程应用而设计，不适用于这些新型应用。综上，异构数据中心需要新的硬件机制与调度算法来适应新型应用的需求。

针对数据中心中的异构处理器硬件架构和新型应用，本文完整并深入地研究了异构数据中心中的硬件机制和调度问题。本文的主要研究内容分为以下四个方面：

1. 高效多任务 GPU 的硬件机制：针对 GPU 的高效共享问题，本文提出了细粒度共享 GPU 的硬件机制，即同时多任务 GPU (*Simultaneous Multikernel GPU, SMK*)。这一硬件机制包括了低开销的部分上下文切换机制，公平静态资源分配算法和公平的 warp 调度算法。使用 SMK，GPU 上执行的多个任务可以根据其资源需求动态的调整静态资源（寄存器等）的分配和执行时钟周期的分配从而达到公平、高效的共享 GPU。实验表明，和顺序执行相比，SMK 可以提高 37% 的吞吐率，而和粗粒度共享机制相比，SMK 可以提高 12.7% 的吞吐率。
2. 支持 QoS 的多任务 GPU 的硬件机制：针对 GPU 没有硬件级 QoS 支持的问题，本文提出了 GPU 细粒度共享的 QoS 支持机制。这一 QoS 支持机制包括了对每

个任务的线程级并行度 (TLP) 的分配算法和 QoS 感知的指令调度算法，其可以在时钟周期级别上控制每个任务的执行进度。实验表明，对于给定的一系列 QoS 目标，这一机制相比之前的方法能达到的 QoS 目标多 43.8%，且吞吐率提高了 20.5%。

3. 异构任务工作负载分配调度算法：针对使用 CPU 和 GPU 的异构任务的工作负载分配问题，本文根据 CPU 和 GPU 的性能特性，提出了基于渐进分析的 *CPU+GPU 协同调度算法 (Co-Scheduling Based on Asymptotic Profiling, CAP)*。CAP 只需要很小的性能开销就可以在运行时精确的测出 GPU 和 CPU 之间的性能比，从而为 CPU 和 GPU 均衡的分配工作负载。实验表明，相比之前的工作负载分配算法，CAP 可以平均提高 42.7% 的性能。
4. 新型负载可变的多线程应用的 QoS 调度算法：针对新型负载可变的多线程应用的 QoS 调度问题，本文提出了 *EMU: QoS 感知的弹性资源竞争管理机制*。EMU 使用机器学习方法来预测应用在给定输入和资源分配下的性能，从而为应用分配足够的资源来支持其对 QoS 的要求。实验表明，相比之前的方法，EMU 在保证延迟敏感型应用的 99% 尾延迟的前提下，一起运行的尽力服务型 (Best Effort, BE) 程序的吞吐率提高了 36.02%。

**关键词：**数据中心 通用图形处理单元 细粒度共享 服务质量  
任务调度算法

# The Architecture Designs and Scheduling Algorithms for Heterogeneous Datacenters

## ABSTRACT

With the development of cloud computing, the system architectures and scheduling algorithms in datacenters, which serve as the foundation of cloud computing, have become important research areas in computer science. Different from traditional high performance computing clusters, datacenters need to serve multiple users simultaneously, and users have their demands for quality of service (QoS) for latency-sensitive applications. Traditionally, datacenters are equipped with multi-core CPUs to host single-threaded application with stable workload. Many algorithms have been proposed to schedule these tasks. Recently, heterogeneous processors like General Purpose Graphic Processing Units (GPGPU) have been widely adopted in datacenters to improve their performance and energy efficiency, making them heterogeneous datacenters. However, there are four new problems in heterogeneous datacenters. First, current GPUs are designed for high performance computing clusters, and can only be used exclusively, which does not meet the demands of sharing in datacenters. It is important to design new mechanisms for GPUs to support highly efficient sharing, which is required by datacenters. Second, GPU does not have hardware support for QoS, leading to resource under-utilization and lower system performance in datacenters when executing latency-sensitive applications. At the same time, the usage of GPUs brings heterogeneous tasks which use GPUs or both CPUs and GPUs. These heterogeneous tasks also bring challenges to scheduling algorithms. Last, new applications are multi-threaded with changing workloads. Previous scheduling algorithms designed for traditional applications do not fit this new type of applications. Hence, heterogeneous datacenters need new architectures and scheduling algorithms to meet the needs of new applications.

Targeting the heterogeneous processors and new applications in datacenters, we investigate the architectural designs and scheduling algorithms in heterogeneous datacenters. Our main research is in the following areas:

1. Highly efficient multitasking GPU architecture: For efficiently sharing GPU, we propose a fine-grain sharing GPU architecture: Simultaneous Multikernel GPU (SMK). It

consists of three components: Partial Context Switch with low overhead, Fair Static Resource Allocation algorithm, and Fair Warp Scheduling Algorithm. With these schemes, the GPU can dynamically allocates the resources for sharer kernels to execute them with high fairness and performance. Evaluation result shows that SMK improves the system throughput by 37% over non-shared execution and 12.7% over a state-of-the-art design.

2. QoS mechanism for multitasking GPUs: For the QoS support of GPU, we propose QoS mechanisms for fine-grain GPU sharing. Our QoS support mechanism can provide control over the progress of kernels on per cycle basis with the thread-level parallelism allocation and QoS-aware warp scheduling for each kernel. Evaluation result shows the our techniques achieves QoS goals 43.8% more often, and a throughput 20.5% higher than previous techniques.
3. Scheduling algorithm for heterogeneous tasks which use both CPU and GPU: To balance the workload among CPUs and GPUs, we propose Co-Scheduling Based on Asymptotic Profiling (CAP) to distribute the workload between CPUs and GPUs with optimizations for the performance characteristics of GPU. CAP accurately predicts the performance ratio between CPUs and GPUs at runtime with low performance overhead, and dynamically distributes the workload based on the performance ratio. Evaluation result shows that CAP produces up to 42.7% performance improvement on average compared with the state-of-the-art algorithms.
4. Scheduling algorithm for multi-thread applications with different workloads: For QoS-aware scheduling of multi-thread applications with different workloads, we propose EMU, a QoS-Aware Elastic Contention Management mechanism. EMU uses machine learning algorithms to predict the latency of each user query with different resource configurations, and allocates just enough resources to the application to meet its QoS target. Evaluation result shows that EMU improves the throughput of the co-located applications by 36.02% on average compared with the state-of-the-art technique while achieving the 99%-ile latency target for latency-sensitive applications.

**KEY WORDS:** Datacenters, GPGPU, Fine-grain Sharing, Quality of Service, Task Scheduling

# 目 录

插图索引	xi
表格索引	xiii
算法索引	xv
<b>第一章 引言</b>	<b>1</b>
1.1 数据中心架构 . . . . .	2
1.1.1 传统数据中心中的 CPU 架构 . . . . .	2
1.1.2 异构数据中心中的 GPU 架构 . . . . .	3
1.2 数据中心应用 . . . . .	6
1.2.1 传统应用 . . . . .	6
1.2.2 新型多线程应用 . . . . .	6
1.2.3 异构应用 . . . . .	7
1.3 研究框架概述 . . . . .	8
<b>第二章 同时多任务 GPU：细粒度多任务高吞吐处理器</b>	<b>11</b>
2.1 研究背景 . . . . .	11
2.2 相关工作 . . . . .	13
2.3 研究动机 . . . . .	15
2.3.1 共享的粒度 . . . . .	16
2.3.2 Kernel 异构性 . . . . .	16
2.4 同时多任务 GPU . . . . .	17
2.4.1 部分上下文切换 . . . . .	17
2.4.2 资源使用率 . . . . .	19
2.4.3 公平资源分配 . . . . .	20
2.4.4 通过 warp 调度算法实现的公平动态资源分配 . . . . .	25
2.4.5 增加并发 kernel 的数量 . . . . .	26
2.5 实验验证 . . . . .	28
2.5.1 实验方法 . . . . .	28

2.5.2	SMK 设计的结果 . . . . .	29
2.5.3	与空间划分方法的比较 . . . . .	30
2.5.4	阻塞时钟周期 . . . . .	33
2.5.5	抢占开销 . . . . .	33
2.5.6	3 个或 4 个 kernel . . . . .	35
2.5.7	硬件开销 . . . . .	35
2.6	本章小结 . . . . .	36
<b>第三章</b>	<b>细粒度共享 GPU 的 QoS 支持</b>	<b>37</b>
3.1	研究背景 . . . . .	37
3.2	相关工作 . . . . .	38
3.3	研究动机 . . . . .	40
3.4	细粒度共享的 QoS 设计 . . . . .	42
3.4.1	需要管理的资源 . . . . .	42
3.4.2	从 QoS 目标到体系结构指标 . . . . .	43
3.4.3	体系架构概览 . . . . .	44
3.4.4	QoS 算法 . . . . .	46
3.4.5	管理 Non-QoS Kernel . . . . .	49
3.4.6	静态资源分配和调整 . . . . .	50
3.5	实验验证 . . . . .	52
3.5.1	实验方法 . . . . .	52
3.5.2	$QoS_{reach}$ 的比较 . . . . .	53
3.5.3	Non-QoS Kernel 的吞吐率 . . . . .	55
3.5.4	QoS Kernel 的吞吐率 . . . . .	56
3.5.5	基于优先级的 QoS . . . . .	57
3.5.6	SM 数量的可扩展性 . . . . .	59
3.5.7	能效 . . . . .	60
3.5.8	抢占开销和其他结果 . . . . .	60
3.5.9	硬件开销 . . . . .	61
3.6	本章小结 . . . . .	61
<b>第四章</b>	<b>基于渐近分析的 CPU+GPU 协同调度算法</b>	<b>63</b>
4.1	研究背景 . . . . .	63
4.2	相关工作 . . . . .	64

4.3	研究动机 . . . . .	65
4.3.1	当前调度算法介绍与分析 . . . . .	67
4.4	基于渐近分析的协同调度算法 . . . . .	68
4.4.1	CAP 的设计 . . . . .	68
4.4.2	与其他调度算法的比较 . . . . .	70
4.5	具体实现 . . . . .	71
4.6	实验验证 . . . . .	72
4.7	本章小结 . . . . .	76
<b>第五章</b>	<b>EMU: QoS 感知的弹性资源竞争管理机制</b>	<b>79</b>
5.1	研究背景 . . . . .	79
5.2	相关工作 . . . . .	81
5.3	研究动机 . . . . .	82
5.3.1	真实系统配置 . . . . .	82
5.3.2	OS 调度的长尾延迟 . . . . .	83
5.3.3	现有技术的低硬件利用率 . . . . .	84
5.3.4	EMU 设计指导方针 . . . . .	85
5.4	EMU 的设计 . . . . .	86
5.4.1	性能预测器 . . . . .	87
5.4.2	运行前资源分配器 . . . . .	89
5.4.3	运行时资源分配器 . . . . .	91
5.5	实验验证 . . . . .	94
5.5.1	实验方法 . . . . .	94
5.5.2	EMU 的性能 . . . . .	95
5.5.3	运行时机制的效果 . . . . .	97
5.5.4	动态缓存分配的效果 . . . . .	97
5.5.5	调整间隔敏感性 . . . . .	98
5.5.6	建模算法的敏感性 . . . . .	99
5.5.7	应用到仓库级数据中心 . . . . .	100
5.6	本章小结 . . . . .	101
<b>第六章</b>	<b>总结和展望</b>	<b>103</b>
6.1	主要结论 . . . . .	103
6.2	研究展望 . . . . .	104

---

参考文献	105
致 谢	121
攻读学位期间发表的学术论文	123
攻读学位期间参与的项目	125

## 插图索引

1-1 多路多核 CPU 架构概览 . . . . .	2
1-2 GPU 架构概览 . . . . .	4
1-3 系统整体架构 . . . . .	7
1-4 应用处理算法选择流程图 . . . . .	8
2-1 多任务 GPU 的演化 . . . . .	14
2-2 支持 SMK 的 SM。增加的组件用阴影表示。SM 驱动器控制 SM 和抢占引擎。其可以通过给抢占引擎发射指令来换出 SM 中的 TB 或是把 TB 换入 SM . . . . .	18
2-3 按需资源分配算法 . . . . .	21
2-4 <i>cutcp+stencil</i> 使用按需资源分配时的 TB 分布 . . . . .	22
2-5 计算一个 SM 内的资源分区 . . . . .	23
2-6 资源分区下的 TB 发射算法 . . . . .	23
2-7 一个划分 2 个 kernel 的资源分区的例子 . . . . .	24
2-8 不同数量的 kernel 一起在 SM 上执行时的 L1 和 L2 缓存命中率和公平性 . . . . .	27
2-9 SMK 设计的比较 . . . . .	29
2-10 “C+M”组的系统吞吐率 . . . . .	30
2-11 SMK-(P+W) 和 Spart 的比较 . . . . .	31
2-12 SMK-(P+W) 和 Spart 的“C+M”组的系统吞吐率 . . . . .	31
2-13 相比 Spart 减少的阻塞时钟周期 . . . . .	32
2-14 不考虑抢占开销时的吞吐率 . . . . .	33
2-15 SMK-(P+W) 和 Spart 在 3 个和 4 个 kernel 时的比较 . . . . .	34
3-1 不同的共享 GPU 的方法 . . . . .	41
3-2 细粒度共享 QoS 体系架构扩展（黄色标注）概览 . . . . .	44
3-3 限额分配策略概览。 $K_0$ 是一个 QoS kernel, $K_1$ 是一个 non-QoS kernel, $C_{K_i}$ 是 kernel $K_i$ 的限额计数器 . . . . .	45
3-4 带有基于历史信息的限额调整的简单分配算法未满足 $IPC_{goal}$ 的情况的数量（总共 900 种情况）和其离 $IPC_{goal}$ 的差距 . . . . .	48

3–5 满足 QoS 目标的比例, $\text{QoS}_{reach}$ 。每个柱状图取 90 对或 60 组 (3 个 kernel) 的平均值 . . . . .	54
3–6 根据程序分组的满足 QoS 目标的比例, $\text{QoS}_{reach}$ . . . . .	55
3–7 non-QoS kernel 的归一化吞吐率。X 轴是 QoS 目标 . . . . .	56
3–8 QoS kernel 的归一化吞吐率。其归一化到 QoS 目标。 . . . . .	57
3–9 QoS kernel 的 $\text{QoS}_{reach}$ . . . . .	58
3–10 归一化吞吐率。 . . . . .	58
3–11 56 个 SM 的情况下满足 QoS 目标的比例, $\text{QoS}_{reach}$ 。每个柱状图取 90 对或 60 组 (3 个 kernel) 的平均值。 . . . . .	59
3–12 56 个 SM 的情况下 non-QoS kernel 的归一化吞吐率。X 轴是 QoS 目标。 . . . . .	60
3–13 相比 Spart 的能效提升 . . . . .	61
4–1 不同性能评测程序在不同工作负载大小下的性能曲线 . . . . .	66
4–2 调度算法总览 . . . . .	67
4–3 CAP 的加速比 . . . . .	73
4–4 CAP 中 CPU 和 GPU 执行时间的差距 (越小越好) . . . . .	74
4–5 CAP 的每一步的划分之间的方差 (越小越好) . . . . .	75
4–6 CAP 中不同初始工作负载大小的比较 . . . . .	76
5–1 单线程/多线程 LS 应用 ( $P_1$ ) 和 BE 应用 ( $P_2$ ) 放到一个多核服务器上的性能干扰 . . . . .	80
5–2 归一化到 QoS 目标的 LS 应用的 99% 尾延迟 . . . . .	83
5–3 BE 应用的归一化到单独运行时性能的吞吐率 . . . . .	83
5–4 共享缓存与内存带宽使用情况 . . . . .	84
5–5 EMU 概览 . . . . .	86
5–6 在测试集上预测查询时长的可决系数 $R^2$ 的结果 (越高越好) . . . . .	88
5–7 在测试集上预测查询 IPC 的可决系数 $R^2$ 的结果 (越高越好) . . . . .	89
5–8 运行时资源分配的例子 . . . . .	93
5–9 所有 LS 应用和 BE 应用在 12 个核上的加速比。这些性能评测程序有不同的可扩展性。 . . . . .	95
5–10 使用 Static-Opt 和 EMU 时 LS 应用的 99% 尾延迟和 BE 应用的吞吐率 . . . . .	96
5–11 使用 EMU 及其变体时 LS 应用的 99% 尾延迟 . . . . .	97
5–12 使用不同调节间隔时 LS 应用的 99% 尾延迟和 BE 应用的吞吐率 . . . . .	98
5–13 使用不同模型算法时 LS 应用的 99% 尾延迟和 BE 应用的吞吐率 . . . . .	99

5-14 使用 OS 调度, Static-Opt 和 EMU 时所有协同定位的违反 QoS 的不同程 度的比例 . . . . .	100
5-15 使用 OS 调度, Static-Opt 和 EMU 时所有协同定位的 BE 应用的不同吞吐 率的比例 . . . . .	100



## 表格索引

1–1 数据中心的三种应用类型的比较 . . . . .	6
2–1 GPU 资源随着架构的变化。这里列出的是每个 <b>SM</b> 的资源。 . . . . .	15
2–2 Nvidia GTX980 上的资源使用情况，受限的资源用 <b>粗体</b> 列出 . . . . .	15
2–3 使用公平资源分配时的资源和性能的公平性（越高越好） . . . . .	25
2–4 GPGPU-Sim 的模拟参数 . . . . .	28
2–5 硬件开销 . . . . .	35
3–1 细粒度 QoS 和之前方法的比较 . . . . .	39
3–2 模拟器参数 . . . . .	52
4–1 实验环境 . . . . .	71
4–2 实验中的性能评测程序 . . . . .	72
4–3 CAP 的性能提升 . . . . .	73
5–1 EMU 和之前方法的比较 . . . . .	81
5–2 软硬件环境和性能评测程序 . . . . .	94



## 算法索引

3-1 TB 分配调整算法，每个时间段开始时执行 . . . . .	51
4-2 基于渐进分析的协同调度算法 . . . . .	69
5-3 运行前资源分配算法 . . . . .	90
5-4 运行时资源分配算法 . . . . .	92



## 第一章 引言

传统的高性能计算集群使用任务队列<sup>[1]</sup>来管理集群，并使用例如 MPI<sup>[2]</sup>这样的编程模型来编写程序。使用高性能计算集群的用户可以独占式的使用集群中的资源。随着分布式计算技术的发展，云计算<sup>[3]</sup>作为一种灵活、高效的计算方式在产业界获得了广泛的应用。云计算使用虚拟化<sup>[4]</sup>、容器<sup>[5]</sup>等技术让用户共享式的使用集群中的资源。除了让用户直接使用集群中的资源（基础设施即服务，IaaS），云计算还把平台和应用作为服务提供给用户，从而实现了平台即服务（PaaS）<sup>[6]</sup>和软件即服务（SaaS）<sup>[7]</sup>。云计算中的一部分应用是延迟敏感型（Latency-Sensitive, LS）应用。延迟敏感型应用是一类要求用户的查询在给定的时间内返回的应用。云计算中的另一部分应用则是尽力服务型（Best Effort, BE）应用。尽力服务型应用要求在单位时间内处理尽量多的查询，但对查询的返回时间没有要求。

作为支撑云计算的硬件基础，数据中心为云计算应用提供了强大的计算能力和存储能力。数据中心中包含了了大量的服务器，而服务器之间通过高速网络进行连接。在传统的数据中心里，其计算能力由多路多核 CPU 处理器提供。多个 CPU 之间共享一个服务器中的内存和硬盘，并通过数据总线进行 CPU 之间的通信。当用户查询到来的时候，查询所需的计算任务会被分发到这些 CPU 上进行计算。

但是，传统数据中心的设计架构面临着新的挑战。当前的多路多核 CPU 处理器的性能增长并不能适应新出现的应用的计算需求。虽然增加服务器的数量可以增加数据中心的计算能力，但功耗也会随之增加<sup>[8]</sup>。增加的功耗不单会增加数据中心的维护开销，还对供电、散热等子系统施加了更大的压力。因此，单纯的对服务器和 CPU 的数量进行扩展并不能更有效的提高数据中心的计算能力。

为了解决这一问题，数据中心引入了例如 GPU 等和传统多路多核 CPU 相异的异构处理器来增加其性能功耗比。与 CPU 不同，GPU 使用了大量的简单核心来进行海量并行计算。这一设计使 GPU 可以提供比 CPU 高的多的计算性能和性能功耗比。使用 GPU，数据中心可以在不增加功耗的情况下增加其计算能力。但是，当前的 GPU 架构是为高性能计算场景而设计，并未考虑到数据中心环境下的共享式使用场景。受到这一限制，用户和云计算应用只能独占式的使用 GPU，降低了系统的整体利用率。

随着云计算应用的发展，当前新兴的云计算应用<sup>[9-11]</sup>不再局限于传统的负载稳定的单线程应用<sup>[12, 13]</sup>。在传统应用中，每个用户查询会被分发给一个线程，而这个线程的计算任务会由一个 CPU 核进行处理，并且计算任务的工作负载并不随着不同的用户查询

而变化。随着应用需求的变化，单个核心已经不能为用户的查询提供满意的用户体验。因此，在新型应用中，一个用户查询需要由多个线程（即多个 CPU 核）同时进行处理。同时，在这些应用中，不同的用户查询会导致计算任务的总工作负载不同。

除了新型的多线程应用，某些应用（例如深度神经网络）<sup>[9]</sup>需要使用 GPU 进行加速。这些应用需要大量的计算，即使是多路多核 CPU 也不能在短时间内完成计算。为了充分利用系统资源，这些应用的计算任务可以同时使用 CPU 和 GPU 进行计算。工作负载在 CPU 和 GPU 之间的均衡分配对性能起着重要的作用。

## 1.1 数据中心架构

数据中心主要分为由 CPU 提供计算能力的传统数据中心与由 CPU 和 GPU 共同提供计算能力的异构数据中心。本文主要是针对异构数据中心进行研究。下面，我们分别介绍支撑这两种数据中心架构的多路多核 CPU 处理器和 GPU 处理器的编程模型和硬件架构。

### 1.1.1 传统数据中心中的 CPU 架构

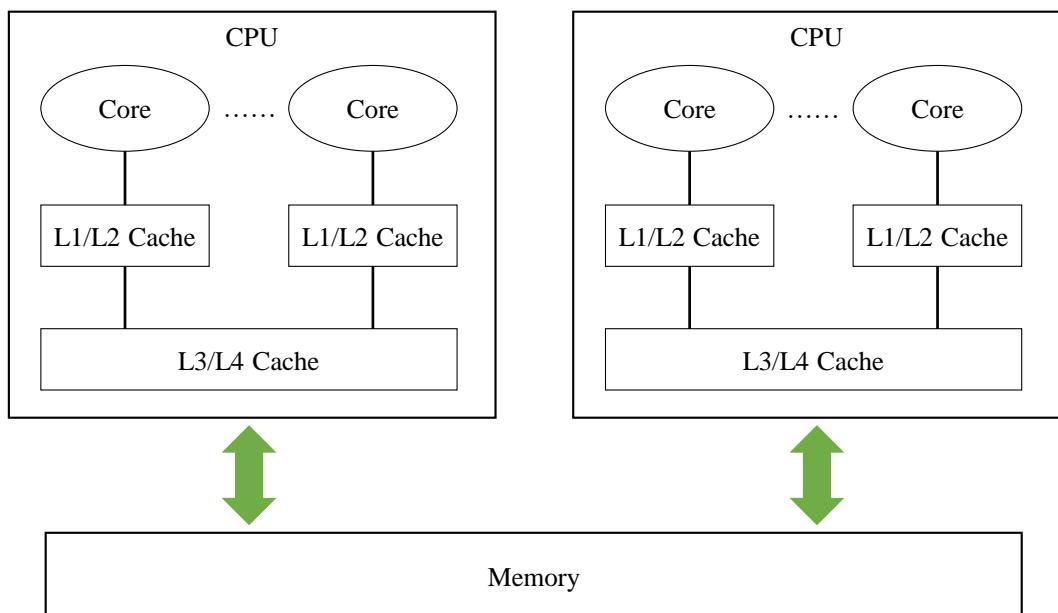


图 1-1 多路多核 CPU 架构概览

Figure 1-1 Overview of Multi-socket Multi-core CPU Architecture

随着 CPU 处理器技术的发展，多核 CPU 处理器已经成为一种主流的 CPU 架构。最初，每个 CPU 内部只有一个核心。由于单核性能随着技术的发展而渐渐不再增长，多个核心被集成到一个处理器内部以提升其并行性能。受到晶体管制造工艺和片上网络技术的限制，一个 CPU 处理器内的核心数是有限的（通常不超过 32 个）。为了满足数据中心对计算能力的需求，多个多核处理器通过主板上的数据总线连接起来，集成到一个服务器内一起工作。这样由多个多核处理器组成的架构就被称为多路多核处理器架构（其中一个处理器即为一路）。

图1-1展示了多路多核 CPU 架构的概览。在多路多核 CPU 架构中，每个处理器包括了多个核心。每个核心各自独立，并有自己的私有高速缓存（通常为 L1 和 L2 缓存）。一路处理器内部的核心共享一个缓存（通常为 L3 和 L4 缓存）。这些核可以通过共享缓存高速的交换数据。不同的处理器之间可以通过数据总线（如 Intel QPI）进行通信，以实现跨处理器的缓存一致性。同时，这些处理器还共享同一块内存，这也可以被用来进行线程间或进程间通信。

多路多核 CPU 处理器是一种并行架构。人们提出了许多并行编程模型以利用处理器的并行计算能力。这些编程模型可以分为两类，一类需要程序员手动为计算任务分配核心，而另一类则由一个运行时系统来调度计算任务。需要手动分配任务的编程模型的典型代表有 MPI (Message Passing Interface)<sup>[2]</sup> 和 Pthreads (POSIX threads)<sup>[14]</sup>。MPI 定义了一套消息传递的接口。不同的进程和线程可以通过这些消息传递接口进行通信，并完成并行计算任务。MPI 支持跨节点的通信。Pthreads 是由 IEEE POSIX 1003.1c 标准所定义的标准化的 C 语言线程接口。使用这个接口，程序员可以在单个节点上生成多个线程来完成并行计算任务。线程之间的通信可以通过共享内存完成。在这两个模型中，程序员都需要手动指定每个进程或线程所需要完成的计算任务。

由运行时系统调度任务的编程模型的典型代表有 Cilk<sup>[15]</sup> 和 OpenMP<sup>[16]</sup>。在这类编程模型中，程序员需要定义一系列可并行的计算任务。运行时系统会根据计算任务之间的依赖关系和执行情况自动的进行通信和任务调度。Cilk 通过扩展 C 语言，实现了一套基于任务窃取的调度系统。在这个系统中，空闲的核会从忙碌的核中“窃取”任务，实现负载均衡。OpenMP 定义了一系列编译器指令，让程序员可以在程序中指定可并行的计算部分（例如没有依赖关系的循环）。编译器会自动分析并生成对应的并行代码完成通信和计算。这样的自动调度系统大大简化了并行编程的难度。

### 1.1.2 异构数据中心中的 GPU 架构

多路多核处理器架构成功的扩展了 CPU 的计算能力。但是随着云计算应用对计算能力的需求不断的增长，其需要更多的核才能在用户需要的时间内完成计算任务。单纯

的扩展处理器的数量会增加数据中心的功耗，并依赖更先进的处理器之间的互联技术。为了解决这一问题，GPU 被引入数据中心来提供更强大的计算能力和更高的性能功耗比。这样同时包括了多种处理器架构的数据中心被称为异构数据中心。

GPU 最初只被用于进行图形图像处理。由于图形图像处理算法天然具有很高的并行度，GPU 的架构被设计成可以高效的进行并行计算。最初的 GPU 使用例如 OpenGL<sup>[17]</sup> 和 DirectX<sup>[18]</sup> 这样的图形编程接口进行编程，其不具备通用计算的能力。随着并行编程逐渐成为主流，GPU 被扩展为可以提供通用计算的能力，即通用计算图形处理单元 (General Purpose Graphic Processing Unit, GPGPU)。为了实现在 GPU 上的通用计算，人们提出了例如 CUDA<sup>[19]</sup>，Brooks+<sup>[20]</sup> 和 OpenCL<sup>[21]</sup> 等编程框架。这些编程框架扩展了 C/C++，让程序员可以相对容易的为 GPU 进行通用编程。下面，我们使用 Nvidia/CUDA 的术语和架构来描述 GPU，这些描述也适用于其他厂商生产的 GPU。我们在这里没有介绍集成 GPU 的情况。集成 GPU 和独立 GPU 的主要区别只有和 CPU 之间的连接方式。

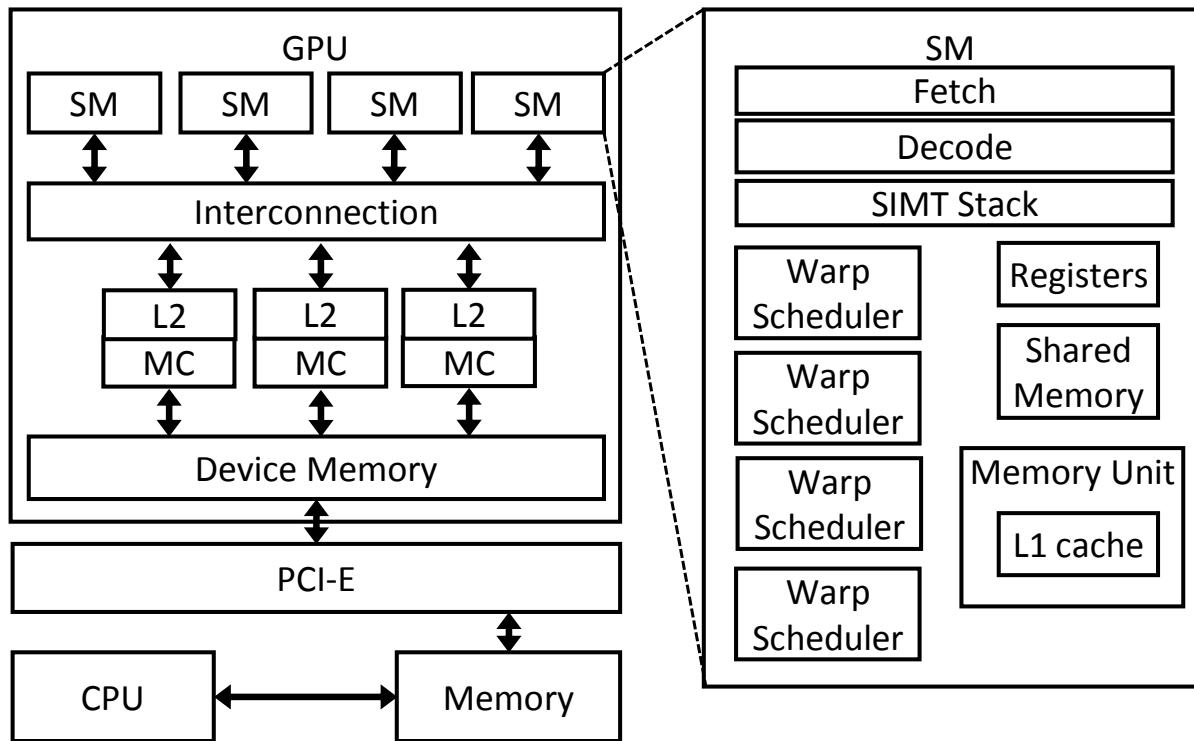


图 1-2 GPU 架构概览  
Figure 1-2 Overview of GPU Architecture

图 1-2展示了 GPU 架构的概览。独立 GPU 通过 PCI-E 总线连接到 CPU 上。一个 GPU 有多个流式多处理器 (Streaming Multiprocessors, SM)。每个 SM 包含许多计算核

和例如寄存器，共享内存和 L1 缓存的资源。**SM** 通过片上互联网络共享一块 **GDDR5** 内存。**SM** 里发出的内存请求根据请求地址分配到不同的内存控制器上。每个内存控制器上还有一块 **SM** 之间共享的 L2 缓存。OS 控制的驱动和运行时系统负责通过 PCI-E 总线给 GPU 发送初始化，数据传输，发射 kernel 和同步的指令。

一个 GPU 程序分为两部分：主机端代码和设备端代码（GPU kernel）。一个 GPU 程序可以包含多个 kernel，每个 kernel 负责一个特定的任务。kernel 之间可能有依赖关系。kernel 是 GPU 上的一段单指令多线程程序（Single Instruction Multiple Threads，SIMT）。程序员为一个线程编写代码，GPU 上会生成多个线程执行这一段代码。GPU 上的线程被组织成线程块（Thread Block，TB），这也是 GPU 上每次线程发射的最小单元。线程的数量和 TB 的大小由程序员指定，而每个线程的资源需求由编译器决定。因此，GPU 可以简单的计算出一个 TB 的资源需求和一个 SM 能容纳多少 TB。

一个 GPU 上最大的并发 TB 的数量受到 GPU 的资源（寄存器，暂存空间大小，线程数和 TB 数）大小的限制。一个 SM 可以接受整数个 TB，直到其中一个所需的资源，如寄存器，共享内存或最大的线程数或 TB 数达到了 SM 的限制。这时剩下的资源就无法被利用，如果 GPU 上的资源不足以让一个 kernel 里所有的 TB 发射，那么余下的 TB 会等待正在执行的 TB 完成后再发射。例如，如果 GPU 上的资源可以容纳 384 个线程，但是 TB 的大小是 256 个线程，那么只有一个 TB 可以被发射，剩下的资源就无法被这个 kernel 利用。

在 TB 被发射到 SM 上之后，其线程被硬件上被组织成 warp。每个 warp 有 32 个线程，即 GPU 一条指令的执行宽度。SM 里的寄存器，共享内存和缓存被所有 warp 共享。SM 有多个 warp 调度器来取指，解码和发射指令。每个 warp 调度器管理 32 个 ALU，即 warp 的大小。warp 被均等的分配到 warp 调度器上。warp 调度器根据 warp 调度算法选择合适的指令来发射，以提高性能。SM 在不同 warp 间切换的时候没有开销。这是因为 SM 不需要切换 warp 的上下文。当一个 warp 被内存请求或其他操作阻塞的时候，warp 调度器会选择其他 warp 可以发射的指令来发射以保持流水线繁忙。因此，阻塞的 warp 的延迟可以被其他 warp 掩盖，从而对吞吐率不会产生影响。

虽然 GPU 具有强大的并行计算能力和高性能功耗比，但是现在的 GPU 不能让动态到来的程序共享一块 GPU，也不能抢占当前正在运行的程序。这种独占式的使用导致 GPU 的计算能力没有得到充分的利用。同时，GPU 也没有在硬件上支持 QoS 的机制。即便多个程序可以通过软件方法静态的共享 GPU，其 QoS 也无法得到保证。

表 1-1 数据中心的三种应用类型的比较

Table 1-1 Comparison of three kinds of applications in datacenters.

	传统应用	新型多线程应用	异构应用
处理查询的线程数	单个线程	多个线程	海量 GPU 线程, 多个 CPU 线程
工作负载	稳定	可变	可变
硬件资源使用	单个 CPU 核	多个 CPU 核	GPU 或 CPU+GPU

## 1.2 数据中心应用

数据中心上运行的延迟敏感型应用主要分为三种：传统负载稳定的单线程应用，新型负载可变的多线程应用，以及随着异构数据中心的发展而引入的异构应用。这些应用需要为用户提供稳定、快速的响应。同时，为了提高数据中心的资源利用率，这些应用常常会和其他尽力服务型应用一起运行。表1-1总结了这三种应用的特点和区别。下面，我们具体介绍这三种应用。

### 1.2.1 传统应用

传统的数据中心应用包括了例如网页服务<sup>[12]</sup> 和网页搜索<sup>[13]</sup> 等应用。这些应用在接收到用户的查询之后，会为每个查询分配一个线程进行处理。使用这种方法，每个用户查询会使用一个 CPU 核进行处理，并在处理完成之后把结果返回给用户。在这类应用中，不同的用户查询的工作负载基本保持恒定。

之前的研究提出了许多调度算法来限制应用的延迟并提高系统利用率。有些算法根据之前用户查询的延迟来为以后的查询分配资源和调节处理器频率。由于这类应用的不同查询的工作负载是稳定的，这些调度算法可以有效的保证查询的 QoS。同时，由于每个查询都用一个单独的核进行处理，不同应用之间的干扰也更容易被量化。这样，在多个应用一起运行的时候，调度算法可以根据应用之间的干扰情况调节资源分配和处理器频率。

### 1.2.2 新型多线程应用

随着云计算应用的发展，传统的单线程应用已经不能提供足够的计算能力。例如智能个人助理<sup>[9]</sup> 和金融服务<sup>[11]</sup> 等应用需要更复杂的计算来支撑。这时再使用单个线程处理每个查询就不能为用户提供快速的响应。因此，这类应用使用多个 CPU 核上的多个线程来处理一个用户查询。在这类应用中，不同的用户查询的工作负载也不同。因此，不同的查询在同样的资源分配下会出现不同的延迟。

新型多线程应用的这些特性让之前的调度算法不再适用。首先，可变的工作负载让我们无法直接根据之前的查询的延迟来调整后面查询的资源分配。其次，在这种情况下我们需要为每个查询分配一定数量的线程和 CPU 核来进行处理，而传统应用中不需要考虑这一点（每个查询固定由一个 CPU 核上的一个线程处理）。由于不同的应用的扩展性和性能各不相同，这导致很难设计出一个调度算法来进行多个多线程应用之间的资源分配。因此，我们需要开发新的调度算法来调度这类新型多线程应用。

### 1.2.3 异构应用

对于某些运算量巨大的应用（例如深度神经网络<sup>[9, 22–25]</sup>），即使是多路多核 CPU 处理器也无法在短时间内返回结果。因此，这些应用会利用 GPU 强大的并行计算能力来提供良好的用户体验。有些异构应用<sup>[9]</sup> 只使用 GPU 进行处理，而有些<sup>[26]</sup> 同时利用 CPU 和 GPU 进行处理。

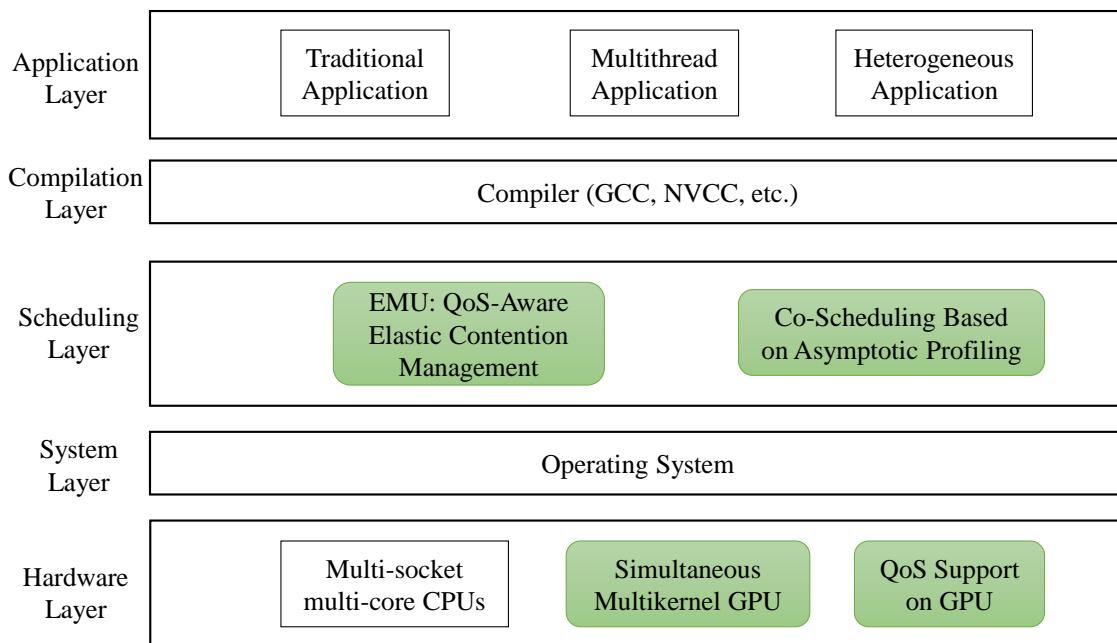


图 1-3 系统整体架构  
Figure 1-3 Overview of System Architecture

这类异构应用的调度受到了 GPU 硬件架构的限制。当前的 GPU 架构不支持抢占，这让许多 CPU 上的调度算法不适用于 GPU。传统调度算法中利用共享来提升系统利用率的方法也由于 GPU 没有共享的支持而不能使用。这需要新的 GPU 硬件架构来为这些

数据中心所需的功能提供支持。同时，GPU 的性能不仅和应用本身有关，还和其每次处理的数据量的大小有关。这让 CPU 和 GPU 之间的负载均衡变得更加困难。虽然之前有研究提出了各种 GPU 的性能模型，但其都太过复杂，不适合在运行时系统中使用。因此，我们需要开发新的 CPU 和 GPU 间的负载均衡算法来提高异构应用的性能。

### 1.3 研究框架概述

本文的研究工作提出了面向异构数据中心和新型应用的硬件架构和调度算法。本文提出的新型硬件架构解决了当前 GPU 架构的缺陷，使其支持共享和 QoS，而调度算法则解决了负载可变的多线程应用和异构应用的 QoS 和负载均衡问题。这些硬件架构和调度算法分别在模拟器和真机测试中进行了验证。

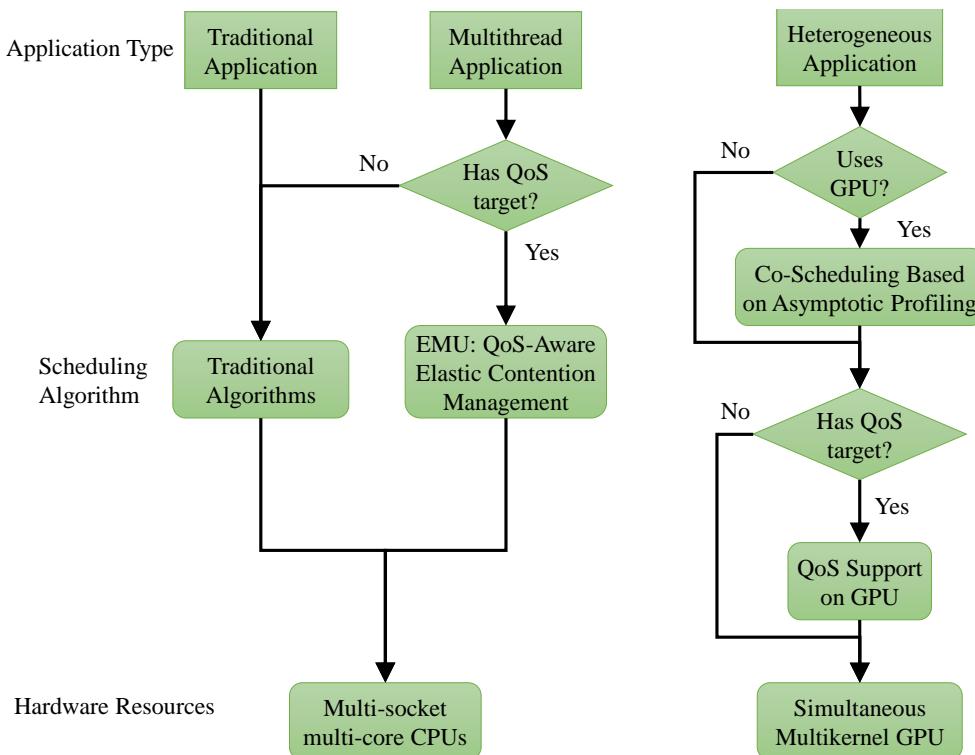


图 1-4 应用处理算法选择流程图

Figure 1-4 The Processing Flow of Different Types of Applications

图 1-3展示了本文研究工作在整个系统中的位置。如图所示，本文的研究工作在调度层和硬件层。硬件层上的研究包括了同时多任务 GPU (*Simultaneous Multikernel GPU, SMK*) 和 GPU 细粒度共享的 QoS 支持机制。这两个新型硬件架构分别提供了高效的共享 GPU 的机制，和在硬件上支持应用 QoS 的机制。调度层上的研究包括了基于渐进

分析的 *CPU+GPU* 协同调度算法 (*Co-Scheduling Based on Asymptotic Profiling, CAP*) 和 *EMU: QoS* 感知的弹性资源竞争管理机制。这两个调度算法分别针对异构应用和负载可变的多线程应用进行调度，以达到更好的性能。

图 1-4 给出了数据中心在处理不同种类的应用时所使用的方法。如图所示，针对不同类型的应用，数据中心使用不同的方法来进行调度。我们分别在第二章到第五章介绍了这些调度方法，其中第二章和第三章介绍了调度 GPU 应用的硬件架构，第四章和第五章介绍了调度异构应用和负载可变的多线程应用的方法。

针对使用 GPU 的尽力服务型应用 (即没有 QoS 要求的应用)，我们在第二章中提出的同时多任务 GPU 架构可以高效的支撑多个应用共享一块 GPU。我们发现当前的 GPU 架构在运行单个应用和进行粗粒度共享时硬件资源没有得到充分的利用。根据这一观察，我们研究了 GPU 细粒度共享的硬件机制。通过资源的细粒度公平分配和硬件调度算法的改进，同时多任务 GPU 架构可以在保证应用公平性的前提下显著的提高系统吞吐率。

针对使用 GPU 的延迟敏感型应用 (即有 QoS 要求的应用)，我们在第三章中提出的 GPU 细粒度共享的 QoS 支持机制可以为这类应用保障其 QoS。当前的 GPU 架构并不提供 QoS 的硬件支持，并且一起运行的应用之间会互相干扰，影响应用的 QoS。通过研究影响 GPU 上应用的性能的关键因素，我们提出了保障应用 QoS 的硬件机制。通过为不同应用合理的分配线程数，计算和内存资源，GPU 细粒度共享的 QoS 支持机制在保障了延迟敏感型应用的 QoS 的同时显著提高了尽力服务型应用的吞吐率。

针对同时使用 CPU 和 GPU 的异构应用，我们在第四章中提出的基于渐进分析的 *CPU+GPU* 协同调度算法可以为 CPU 和 GPU 均衡的分配工作负载。当前的负载分配算法要么需要承受高昂的运行时开销，要么无法准确的预测 CPU 和 GPU 之间的性能比。基于 GPU 的性能特征，我们提出了通过少量性能采样即可准确预测 CPU 和 GPU 之间性能比的方法。通过为 CPU 和 GPU 均衡的分配工作负载，应用的整体性能有了大幅度的提升。

针对负载可变的多线程应用，我们在第五章中提出的 *EMU: QoS* 感知的弹性资源竞争管理机制可以高效的保障应用的 QoS 并提高系统整体性能。传统的调度算法假设应用的工作负载是恒定的，因此其利用之前的执行情况对后面的查询进行调度，而在调度负载可变的多线程的应用时这一假设不再成立。通过基于机器学习的性能模型，我们可以准确的预测应用在不同资源配置下的性能，并根据预测结果为其分配刚刚好够用的资源。通过对核，缓存和内存带宽资源分配的精确控制，*EMU* 既能适应延迟敏感型应用多变的需求来保障其 QoS，还有效的提高了尽力服务型应用的吞吐率。

在第六章中，我们总结了全文的主要研究成果，并对下一步研究进行了展望。



## 第二章 同时多任务 GPU：细粒度多任务高吞吐处理器

### 2.1 研究背景

GPU 已在云计算，数据中心和移动/嵌入式计算机中得到了广泛应用。GPU 通常包含许多流式多处理器（Streaming Multiprocessors，SM）。每个 SM 包含许多简单的执行核心<sup>[19]</sup>。GPU 具有海量计算核心，并利用线程级并行来掩盖内存访问延迟。许多应用都被移植到 GPU 上以利用其强大的计算能力来提供显著的加速比<sup>[27-29]</sup>。

但是，这些程序有时并未对 GPU 进行充分的优化，导致了片上资源利用率低。一个 kernel<sup>1</sup> 可能只使用很少的片上高速暂存空间，而大量使用 L1 缓存。另一个 kernel 则可能是相反的资源使用情况。这一情况可能来源于用户没有足够的经验来写出可以利用 GPU 所有资源的代码。之前的工作也报告了类似的发现<sup>[30]</sup>。另外，尽管 GPU 有很多个线程，但由于现在的 GPU 性能受到内存带宽的限制，仍然有一些计算核会在一些时钟周期处于空闲状态。不能及时返回的内存请求可能导致多线程不能掩盖的内存延迟。这一情况在内存密集型应用上尤为常见，导致了核的动态计算时钟周期没有得到充分利用。我们在 10 个来自 Parboil 性能评测程序集的程序上观测到了 52.5%–98.1% 的核空闲时间。

不仅如此，数据中心和云计算通常作为服务展现给用户。这些场景下通常使用共享和虚拟化技术来让多个应用共享硬件资源，从而提高费效和能效。但是，即使是最先进的 GPU 也几乎没有提供多个应用共享执行的支持。一旦一个 kernel 发射到了 GPU 上，它就不能被抢占。其他应用的 kernel 必须等到当前执行的 kernel 结束之后才能开始执行。

之前有工作试图通过软件方法共享 GPU。这些技术主要依靠用户或代码转换的方法静态的将可一起执行的 kernel 合并成一个 kernel<sup>[30, 31]</sup>。这些方法对嵌入式系统比较适合，因为在嵌入式场景下需要执行的 kernel 基本是固定的。但是，这些方法不适用于通用系统。在通用系统中 kernel 到来的顺序会动态的变化。因此，静态方法不能解决 GPU 动态共享的问题。并且，使用这些方法无法解决 kernel 不能抢占的问题。

最近有工作提出了硬件上的 kernel 的抢占机制<sup>[32, 33]</sup>。主要的方法是切换 SM 上 kernel 的上下文，从而让新的 kernel 可以执行。上下文切换通过把一个 kernel 的上下文从 SM 中保存到内存中来实现。这一方法产生大量的内存访问，并有很高的性能开销。还

<sup>1</sup>一个 GPU 程序包括多个 kernel，每个 kernel 会生成许多组织成线程块（Thread Block, TB）的线程。

有一种方法是，一个 SM 可以不再接收当前 kernel 的 TB，并等到当前执行的所有 TB 结束。由于当前执行的 TB 结束所需的时间不定，这种方法可能会导致很长的抢占延迟。Chimera<sup>[32]</sup> 提出了另一种方法。他们提出满足幂等性 (Idempotent) 的 kernel 的 TB 可以直接丢弃，而不影响最后的结果。他们提出的系统结合了上面三种上下文切换的机制来降低上下文切换的开销。这些方法让来自不同应用的 kernel 执行在不同的 SM 上，实现了空间划分的多任务 GPU (Spatially-Partitioned Multitasking GPU, Spart)<sup>[33, 34]</sup>。但是，资源利用率低下的问题主要发生在一个 SM 内部。因此，这些上下文切换的技术不能提高 GPU 的资源利用率。

在这一章中，我们提出了一种新的多任务 GPU，其可以：(1) 显著提升静态和动态资源的利用率，并提高整体系统吞吐率；(2) 为并发 kernel 提供公平的共享机制；(3) 改善并发 kernel 的返回时间。我们提出了同时多任务 (Simultaneous Multikernel, SMK) GPU 来提高 GPU 的线程级并行度。这一概念借鉴自 CPU 上的同时多线程技术。SMK 通过挖掘 kernel 的异构性来让多个 kernel 在一个 SM 内部进行细粒度共享。SMK 通过以 TB 为单位的细粒度上下文切换机制来实现。同时，我们还提出了新的 TB 发射策略和新的 warp 调度策略来保证 kernel 之间的公平性。

在同一个 SM 里执行多个 kernel 的基本原则是让这些 kernel 的资源互补，这样才能提高资源利用率和执行效率。例如，一个内存密集型 kernel 可以和一个计算密集型 kernel 在一起执行。前者不使用片上高速暂存空间，还会产生许多内存访问延迟。而后者则会大量使用片上高速暂存空间，并且主要的执行时间都在进行计算。将这两个 kernel 放在一个 SM 里可以有效的提高片上高速暂存空间的利用率，并提高计算核的利用率，从而提高整体资源利用率和系统吞吐率。我们在这章中主要有如下技术贡献：

- 支持 SMK 的低抢占开销的细粒度上下文切换机制。我们提出了以 TB 为单位的上下文切换机制。在一个新 kernel 到来的时候，我们只换出让新 kernel 足够开始执行的正在执行的 kernel 的 TB。这一设计显著降低了上下文切换的开销和新 kernel 开始执行的延迟，从而改善了响应时间。
- 一个考虑了多个 kernel 间静态资源使用率和公平性的 TB 发射策略。当向一个 SM 上发射一个 TB 的时候，我们考虑了公平性来确保静态资源的分配是公平的。通过资源划分的方法，我们保证了资源的公平。因此，我们的策略是 Spart 的一个超集，并且上下文切换开销更小。
- 一个管理 kernel 间计算时钟周期的 warp 调度算法。我们设计了一个在 SM 内部为不同 kernel 分配计算时钟周期的 warp 调度算法来减少计算核的空闲时间。这个调度算法根据 kernel 单独运行时的计算时钟周期利用率（通过在线性能分析获得）来分配 kernel 的计算时钟周期。

通过对 45 对来自 Parboil 性能评测程序集的程序对的实验证，SMK 相比 Spart 最高可提升 61.2% 的系统吞吐率，在 kernel 具有互补资源的时候可以平均提高 17% 的吞吐率，而对所有的 45 个程序对可以平均提高 12.7% 的吞吐率。相比 Spart，SMK 的公平性改善了 5.7%。SMK 还大大降低了平均返回时间，对于互补的 kernel 平均降低了 19.0%，而对所有的 45 个程序对则平均降低了 10.6%。随着并发 kernel 数量的增多，SMK 可以达到更高的吞吐率。在 3 个和 4 个 kernel 一起执行的情况下，相比 Spart，SMK 平均提高了 14.8% 和 14.3% 的系统吞吐率。

## 2.2 相关工作

kernel 并发的初期支持依赖于程序员来定义可并行的 kernel 流（Streams），每个流包括了一组互相之间具有依赖关系的 kernel。Hyper-Q<sup>[35]</sup> 在硬件上为多个流提供了多个硬件队列，这样不同队列中的流中的 kernel 就可以同时在一块 GPU 上运行。这种用户定义的静态并行方法适合具有多个 kernel 的单一程序，但是对于多个程序就不适用了。MPS<sup>[36]</sup> 提供了一个运行时系统，使来自不同进程的 kernel 可以同时被送入 Hyper-Q 中。但是，在 kernel 进入了硬件队列之后，MPS 无法控制这些 kernel 的 TB 是如何被调度的。正如 Wu 等人的研究所指出的<sup>[37]</sup>，因为不能控制 TB 的发射，一般情况下硬件会将两个可并行的 kernel 串行执行。并且 MPS 没有资源管理机制。一起运行的 kernel 会以不受控的方式争抢资源，损害了公平性。另外，这些方法也不能处理动态到来的 GPU 任务，一旦 kernel 被发射就不能被其他 kernel 抢占。

一些复杂的软件方法被提出来以绕过硬件的限制。他们通过编译器技术把两个 kernel 合并成一个 kernel<sup>[30, 31, 37]</sup>，而执行路径则通过条件语句来进行控制<sup>[38, 39]</sup>。图 2-1(a)展示了这种方法。尽管这个方法可以并行化多个程序，这种方法是一种静态方法，不能处理动态到来的程序。而且在这种情况下，因为来自不同程序的 kernel 都被合并成一个 kernel 了，硬件不能分辨来自不同程序的执行路径。因此，这种方法可能会导致不同程序的 kernel 的调度不公平。同时，这个方法要求必须在执行之前提供 kernel 的源代码，这有时也不可行。因此，这个方法只适用于提前知道会运行什么 kernel 的场景（如嵌入式系统）。

也有一些工作在操作系统和虚拟机层面试图让 GPU 支持多任务<sup>[40-42]</sup>。一个典型的方法是根据需要在操作系统内核层面上截获 kernel 的发射请求，只允许一部分 kernel 的发射请求通过<sup>[43]</sup>。这个方法的主要问题是一个已经被发射的 kernel 不能被抢占，所以新的 kernel 必须等待正在执行的 kernel 结束之后才能发射，就如图 2-1(b)所示的那样。因此，这种方法不支持共享，GPU 的利用率也没有得到提升。

最近的研究提出了在 GPU 硬件架构上进行扩展来支持 kernel 的抢占和共享 GPU。

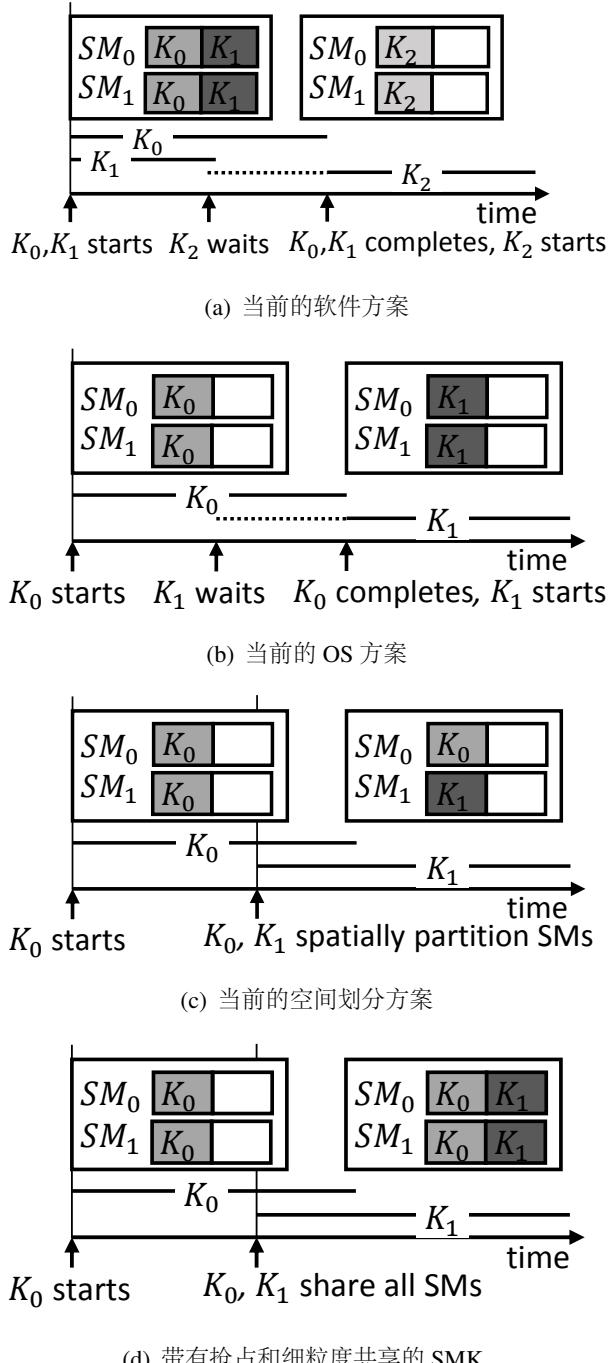


图 2-1 多任务 GPU 的演化

Figure 2-1 Evolution of multitasking GPU

GPU 上的上下文切换可以通过把一个 SM 上的 kernel 的上下文保存到内存里，或是等待当前 kernel 的所有正在这个 SM 上运行的 TB 结束来实现<sup>[33]</sup>。如果是通过内存进行切换，就会产生大量的内存访问，对性能产生影响。如果等待 TB 结束，则会导致不可预期的抢占延迟。Chimera<sup>[32]</sup> 研究了某些具有“幂等性”(idempotent)<sup>[44]</sup> 的 kernel。如果把这种 kernel 正在运行的 TB 直接强行结束掉，并在之后重新启动，kernel 最后的结果也不会受到影响。如果一个 kernel 具有这种性质，在上下文切换的时候 Chimera 就直接把它的 TB 强行中止，从而减少了上下文切换的开销。如图 2-1(c)所示，这些技术通过把不同的 SM 划分给不同的 kernel，实现了多个 kernel 通过空间划分(Spart)共享一个 GPU。但是，每个 SM 里仍然只执行一个 kernel。而每个 SM 里的资源利用率仍然低下。而我们提出的 SMK(图 2-1(d))则让多个 kernel 可以共享一个 SM，从而提高了资源利用率和 GPU 吞吐率。

表 2-1 GPU 资源随着架构的变化。这里列出的是每个 SM 的资源。

Table 2-1 The resources of the GPU of ascending generations. The resources are for each SM.

架构	SM 的数量	ALU 的数量	寄存器	共享内存	线程数限制	TB 数限制
Tesla(GTX280)	30	8	64KB	16KB	1024	8
Fermi(GTX580)	16	32	128KB	48KB	1536	16
Kepler(GTX780 Ti)	15	192	256KB	48KB	2048	16
Maxwell(GTX980)	16	128	256KB	96KB	2048	32

表 2-2 Nvidia GTX980 上的资源使用情况，受限的资源用粗体列出

Table 2-2 Resource usage on Nvidia GTX980. The limiting resource is in **bold**

Kernel	寄存器	共享内存	线程数	类型	阻塞时钟周期
lattice6overlap	87.5%	67.1%	<b>100%</b>	计算密集型	54.5%
StreamCollide	<b>92.3%</b>	0%	52.7%	内存密集型	89.9%
mysgemmNT	<b>94.5%</b>	5.7%	68.8%	计算密集型	52.5%
spmvjds	46.9%	0%	<b>93.8%</b>	内存密集型	91.2%
block2Dregtiling	75.0%	0%	<b>100%</b>	内存密集型	91.2%
genhists	76.6%	<b>94.8%</b>	87.5%	计算密集型	60.2%

## 2.3 研究动机

在这一节中，我们首先简要介绍了空间划分方法，并讨论了其局限性。然后，我们讨论了我们这个工作的研究动机：kernel 的异构性。

### 2.3.1 共享的粒度

如上文所讨论的，当前的 kernel 抢占机制以 SM 为粒度进行上下文切换<sup>[32][33]</sup>。这种抢占机制有两个缺点。首先，这个机制的抢占开销很高。每次上下文切换都要传输几百 KB 的寄存器，共享内存和其他执行状态。这导致了上下文切换时会产生很高的内存访问流量。由于占用了内存带宽，这些流量不仅阻碍了当前 kernel 的切换，还影响了其他没有被抢占的 kernel 的执行。

其次，以 SM 为粒度进行上下文切换导致了 kernel 之间以 SM 为单位的空间划分式共享方法。这种共享方法受到一个 GPU 内部的 SM 的数量的限制，并且只在 SM 数量很多的时候效果好。但是，SM 的数量在最近几代 GPU 中并不是一直增长的。表 2-1 展示了最近几代 Nvidia GPU 架构中的 SM 数量和 SM 里的资源大小。SM 的数量在 Tesla 架构里有 30，但在 Fermi 中就降到了 16，并在接下来的几代产品中都保持在这一量级。虽然 SM 的数量没有增加，每个 SM 变得越来越大。例如，GTX580 里每个 SM 可以每个时钟周期以 16 个线程的宽度发射 2 条 warp 指令，而 GTX980 中每个 SM 可以每个时钟周期以 32 个线程的宽度发射 4 条 warp 指令。因此，GTX980 里的一个 SM 拥有 GTX580 里的四个 SM 的计算能力。SM 数量没有增长的一个可能的原因是片上互联网络还不能服务大量的 SM。当前，SM 通过交叉开关连接，这一连接方式有扩展性的问题。因此，Spart 受到 SM 数量的限制。另外，Spart 也不能解决资源利用率低下的问题。这一问题在 SM 变大之后变得更为严重，我们将在下一小节中讨论这个问题。

### 2.3.2 Kernel 异构性

资源的使用和运行时的行为随着 kernel 的不同而不同。我们考察了 Parboil 和 Rodinia 两个性能评测程序集里的所有程序。我们观察到几乎里面所有的程序在最近的 GPU 架构，Nvidia GTX980，上利用率都很低。表 2-2 展示了几个具有代表性的程序的资源使用和运行时行为。

如表所示，不同的 kernel 的资源使用情况截然不同。*StreamCollide* 和 *mysgemmNT* 充分的使用了寄存器，但是共享内存几乎没有使用。相反的，*block2Dregtiling* 和 *lattice6overlap* 则被线程数所限制，这导致了寄存器和共享内存的利用率很低。由于在空间划分 SM 的方法里一个 SM 只能分配一个 kernel，每个 SM 内部的资源利用率还是很低，因此这个问题不能用空间划分 SM 来解决。

Kernel 的运行时行为也截然不同。*lattice6overlap*, *mysgemmNT* 和 *genhists* 都是具有高 IPC 的计算密集型 kernel，而其他 kernel 则是内存密集型的。具有高 IPC 的 kernel 利用共享内存来缓存经常访问的数据，降低了阻塞时钟周期的数量。内存密集型 kernel 利用线程并行来掩盖长内存访问延迟，其阻塞时钟周期要比计算密集型 kernel 大得多。计

算时钟周期和内存阻塞时钟周期之间的掩盖只发生在 **SM** 内部的线程之间。

如果多个 kernel 可以被分到一个 **SM** 上，具有不同资源需求和运行时行为的 kernel 就可以放在同一个 **SM** 上。这个策略可以提高资源利用率和性能。假设我们有两个 kernel ( $K_1$  和  $K_2$ ) 放在一个有 3K 个寄存器和 3KB 共享内存的 **SM** 上。 $K_1$  每个 TB 需要 1K 个寄存器和 2KB 共享内存，而  $K_2$  每个 TB 需要 2K 个寄存器和 0B 的共享内存。如果我们使用空间划分，每个 **SM** 只能容纳一个 TB。但是，如果我们可以把两个 kernel 放到同一个 **SM** 上，每个 **SM** 可以容纳一个来自  $K_1$  的 TB 和一个来自  $K_2$  的 TB。这样资源利用率和线程并行度都提高了。更重要的是，如果  $K_1$  和  $K_2$  之间可以互相掩盖对方的阻塞时钟周期，那 GPU 的吞吐率可以进一步提升。例如，假设  $K_1$  是计算密集型 kernel， $K_2$  是内存密集型 kernel。那么在  $K_2$  被内存访问阻塞的时候， $K_1$  就可以执行，这样就降低了整体的阻塞时钟周期。

综上，当前空间划分共享机制被 GPU 架构所限制。Kernel 内在的异构性让我们可以设计出一套可以为现代 GPU 架构提高资源利用率和整体吞吐率的全新的共享机制。

## 2.4 同时多任务 GPU

为了允许共享，**SMK** 考虑一个通用的动态场景，即 GPU 正在执行可以用尽了至少一种资源的 kernel  $K$ 。为了让新来的 kernel  $newK$  和  $K$  在每个 **SM** 上一起执行，抢占机制必须支持换下  $K$  在每个 **SM** 上的一部分的上下文，并换上  $newK$  的新上下文，从而使两个 kernel 可以一起在 **SM** 上执行。而空间划分的抢占方法则会换出  $K$  在一个 **SM** 上的所有上下文，让那个 **SM** 只运行  $newK$ 。因此，我们要考虑的第一个问题是设计一个部分上下文切换机制。然后，我们需要考虑多少  $K$  的上下文应该被换出来以容纳  $newK$  和怎么在  $K$  和  $newK$  之间分配 **SM** 的资源。这一设计应在保证公平的前提下尽可能的提高 GPU 的吞吐率。另外，由于 **SM** 内的 warp 调度器不能感知到多个 kernel，一个 **SM** 内部 warp 的执行可能对一起运行的 kernel 不公平。因此，我们讨论了部分上下文切换的设计，公平资源分配的方法和多个 kernel 的 warp 调度的方法。

### 2.4.1 部分上下文切换

在基础的 GPU 架构中，一个中央化的 **SM** 驱动器（**SM Driver**）负责从 CPU 端接受指令，如 kernel 发射和内存传输操作；为 kernel 执行初始化 **SM**；和把一个 kernel 的 TB 发射到不同的 **SM** 上。在每个时钟周期，**SM** 驱动器都会在包含了一组 TB 的激活队列（Active Queue）中寻找可以发射的 TB 来发射。

我们假设一个通用的处理场景，即应用不一定是同时到达 GPU 的。应用的 kernel 通过先到先服务的方式发射到 GPU 上。当  $newK$  到达的时候， $K$ （来自不同的应用）会

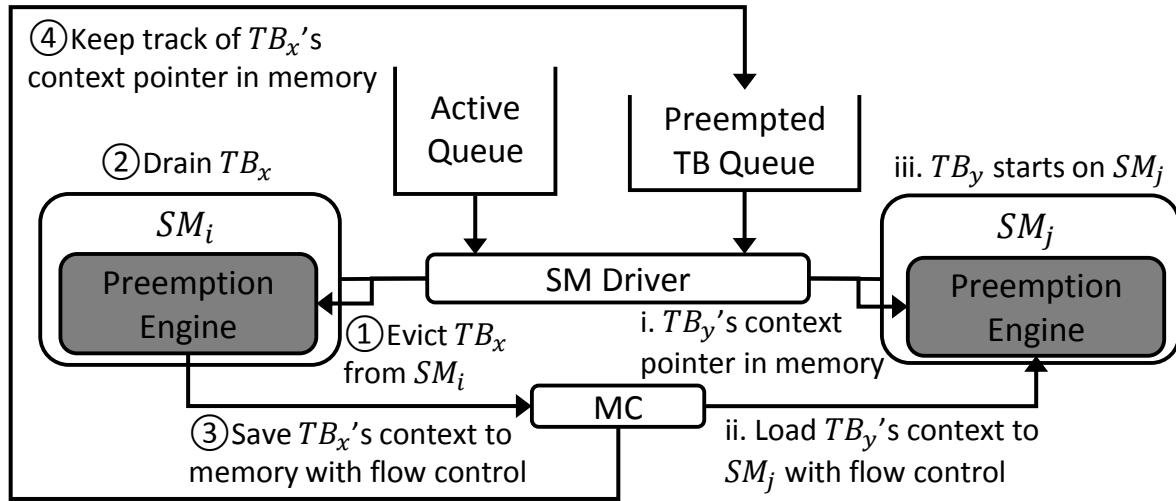


图 2-2 支持 SMK 的 SM。增加的组件用阴影表示。SM 驱动器控制 SM 和抢占引擎。其可以通过给抢占引擎发射指令来换出 SM 中的 TB 或是把 TB 换入 SM

Figure 2-2 SMK-supported SM. Added components are shaded. SM Driver controls the SMs and the Preemption Engine. It can issue commands to Preemption Engine to swap TBs into a SM or swap TBs out of a SM.

被部分的抢占，以让  $newK$  可以和  $K$  一起运行。这取决于发射  $newK$  的 TB 需要多少资源。我们选择以 TB 为单位切换 kernel 的上下文，这与 SM 驱动器发射的单位一致。因此，切换时会一次把  $K$  的一个 TB 的上下文保存到内存中，直到有足够的资源来容纳  $newK$  的一个 TB。例如，如果 SM 上只有 10% 的静态资源是空闲的，而  $newK$  的一个 TB 需要 15% 的资源，那么 SMK 会切换出  $K$  的多个 TB，直到有多余 5% 的资源被释放出来。因此我们将我们的抢占机制取名为部分上下文切换（*Partial Context Switching, PCS*）。

PCS 的主要特性是抢占并不会阻塞 SM，即 SM 上剩下的  $K$  的 TB 在进行上下文切换的时候仍然继续执行。在 Spart 中，上下文切换时整个 SM 都会阻塞住。PCS 不仅可以让 kernel 可以继续执行，还能降低切换开销。我们会在下一段介绍 PCS 的具体步骤。

我们使用和之前工作相同的基础 GPU 执行引擎<sup>[32, 33]</sup>，其涵盖了基本的抢占和多任务支持。我们只讨论和我们设计相关的组件。如图 2-2 所示，我们给每个 SM 添加了一个抢占引擎（Preemption Engine, PreEng）来支持 PCS。当 SM 驱动器决定从  $SM_i$  中换出  $TB_x$  时，它会把这个信息发送给  $SM_i$  中的 PreEng (①)。接着，PreEng 会阻止从  $TB_x$  中再取新的指令，并让流水线中所有来自  $TB_x$  的指令执行完，包括内存请求指令 (②)。一旦这些指令结束，PreEng 会发动 PCS，通过直接向内存控制器发送内存写入请求的

方式把  $TB_x$  的上下文，即寄存器，共享内存，同步屏障信息和 SIMT 栈信息保存到内存中 (③)。在 PCS 完成的时候，PreEng 会把指向  $TB_x$  的上下文的内存地址的指针发送给被抢占 TB 队列 (Preempted TB Queue) (④)。被抢占 TB 队列维护了被抢占的 TB，SM 驱动器在以后可以从这里选择和发射 TB。

换入一个 TB 的上下文和换出其上下文是对称的。图 2-2 展示了把  $TB_y$  换入  $SM_j$  的过程。 $SM$  驱动器首先从被抢占 TB 队列中读取  $TB_y$  上下文的内存地址，然后将其发送给  $SM_j$  的 PreEng (i)。然后 PreEng 向内存控制器发射内存读取请求 (ii)。当上下文被载入之后， $TB_y$  就可以在  $SM_j$  上执行了 (iii)。由于所有那个 TB 正在的指令在上下文切换之前都完成了，并且 TB 之间的资源都是静态隔离的，上下文切换中不会有数据冲突。

综上，PCS 的目的是尽可能的减少上下文切换的开销并最大化 SM 的资源利用率，同时保证被抢占 kernel 的执行。

## 2.4.2 资源使用率

在能把多个 kernel 放到一个 SM 上之后，下一个问题是决定要怎么发射 kernel 的 TB 到 SM 上才能提高资源利用率。这个过程通过在 SM 驱动器里检查 kernel 和 SM 的资源使用率来完成。 $SM$  驱动器在保证公平的前提下最大化整体的吞吐率。这两个目的可能导致相反的决定。例如，为了提高 GPU 的吞吐， $SM$  驱动器可以不停的发射计算密集型 kernel 的 TB (IPC 高) 而让内存密集型 kernel 饿死。因此，保证 SMK 中 TB 发射的公平性十分重要。

在 Spart 中，每个 kernel 会获得整数个 SM。公平性通过保证每个 kernel 获得相同数量的 SM 来保证。但在 SMK 中，其需要分配多种静态资源，而不同的 kernel 对不同资源的需求也不同。有些 kernel 大量使用寄存器而非共享内存，而另一些 kernel 的资源需求则相反。因此，“公平”的划分资源并不简单。

我们使用了主要资源公平 (Dominant Resource Fairness, DRF)<sup>[45]</sup>，一个针对多资源的指标来衡量 kernel 和 SM 的资源使用率。DRF 原本是为了在集群中调度任务而设计的。其思想是多资源的分配应该由 kernel 主要的资源占用来决定，即这个 kernel 所占用的资源的最大使用率。在 GPU 中，有 4 种资源可以分配给一个 kernel 的 TB：寄存器、共享内存、线程数和 TB 数<sup>[19]</sup>。这些资源限制了一个 SM 最大能容纳多少 TB。如果 kernel A 主要使用寄存器而 kernel B 主要使用共享内存，那么 DRF 试图平衡 A 的寄存器使用和 B 的共享内存使用。Kernel ( $rK$ ) 和 SM ( $rSMs$ ) 的主要资源使用率用下列公式计算得出：

$$rK(rSM) = \max\{r(Register), r(Threads), \\ r(SharedMemory), r(TB)\},$$

where  $r(x) = \frac{x_{taken}}{x_{limit}}$

我们用一个例子来说明。假设 kernel A 的一个 TB 使用一个 SM 10% 的寄存器，20% 的共享内存，30% 的线程数和 3% 的 TB 数。那么线程数就是 A 的主要资源。每次 kernel A 要发射一个 TB 的时候，其都会占据一个 SM 30% 的线程数资源。SM 的资源使用率也用类似的方法计算。如果当前 SM 有 30% 的寄存器，15% 的共享内存，20% 的线程数和 10% 的 TB 数被占用，那么那个 SM 的资源使用率就是 30%。在计算一个 kernel 的资源使用率的时候，我们统计其在 GPU 上所占用的资源。统计 kernel 和 SM 的资源使用量十分简单，因为 kernel 每个 TB 的资源使用都是在编译时决定的。SM 驱动器知道每个 SM 里的 TB 的信息。因此 SM 驱动器可以利用资源信息来决定公平的资源分配方法。

### 2.4.3 公平资源分配

在 kernel 和 SM 的资源使用率被定义了之后，下一个问题就是如何公平的分配资源。公平资源分配的目标是尽量平衡每个 kernel 的资源分配（即  $rK$ ）。这可以用最小化所有 kernel 里最大和最小的  $rK$  的差来量化，即  $rK$  的区间。这个指标被 SM 驱动器用于发射 TB 到 SM 上。

我们首先开发了一个简单的资源分配算法，即按需资源分配。其使用一个简单的背包问题启发式策略。基于这个算法，我们开发了第二个算法，即资源分区算法，对资源进行切分后再使用启发式策略。

#### 2.4.3.1 按需资源分配算法

按需资源分配算法在每次 TB 发射的时候分配其资源。这个算法每次会在 SM 上寻找一个受害者 TB，并检查将其切换出去是否可以降低  $rK$  的区间。算法流程图如图 2-3 所示。SM 驱动器首先选择一个  $rK$  最小的 kernel 作为候选 kernel，即  $K_c$ 。然后，SM 驱动器选择一个  $rSM$  最小的 SM 作为候选 SM，即  $SM_c$ 。接着，SM 驱动器检查  $SM_c$  里是否有足够的资源来容纳一个来自  $K_c$  的 TB。如果是的话，SM 驱动器就从被抢占 TB 队列里换入一个之前被抢占的  $K_c$  的 TB（如果有的话），或是发射一个  $K_c$  的新 TB。SM 驱动器优先恢复之前被抢占的 TB 来降低保存被抢占 TB 的内存开销。

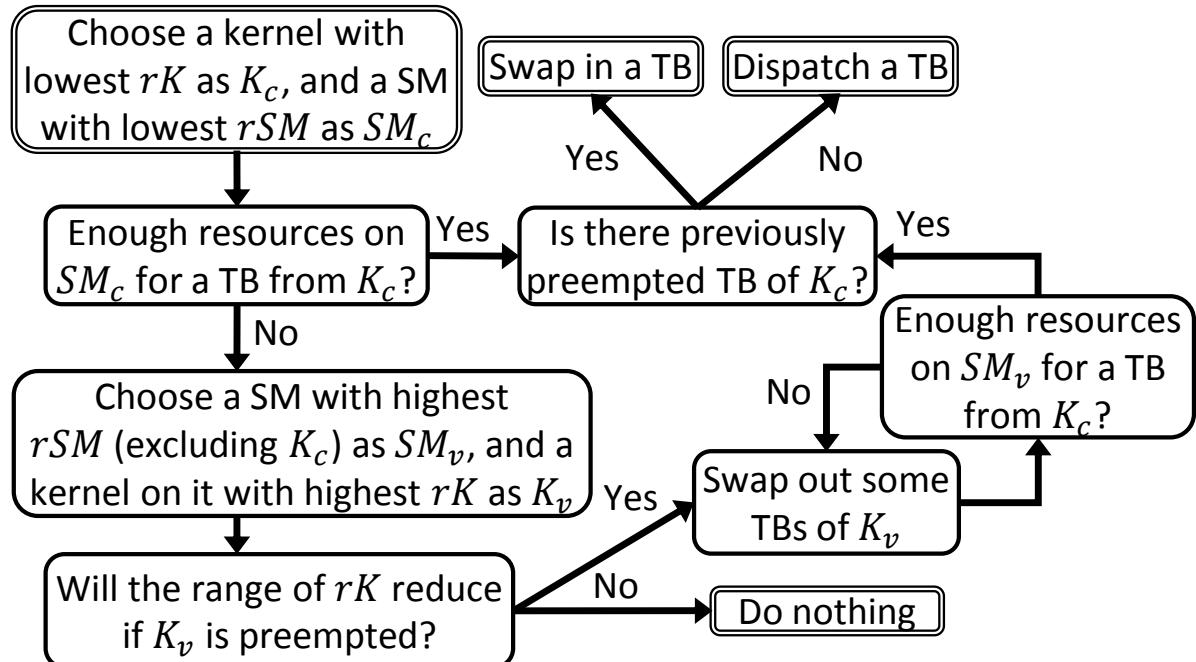


图 2-3 按需资源分配算法

Figure 2-3 On-demand Resource Allocation

如果  $SM_c$  上没有足够的资源，那么 SM 驱动器会寻找一个受害者 SM， $SM_v$ ，即  $rSM$  最大的 SM。在这里计算  $rSM$  的时候， $K_c$  的资源占用是不计算在内的。这是为了寻找可以被换出的其他 kernel。然后，在  $SM_v$  上的资源占用最多，即  $rK$  最大的 kernel  $K_v$  会被作为受害者 kernel。然后，SM 驱动器会检查用  $K_c$  的一个 TB 抢占  $K_v$  的 TB 是否可以降低  $rK$  的区间。如果是的话（即抢占可以提高公平性），那么 SM 驱动器就会进行抢占。 $K_v$  的一个或多个 TB 被换出，一个  $K_c$  的 TB 被换入。否则，抢占不会发生。

使用  $rK$  的区间来决定什么时候抢占对算法的效果很重要。如果这个区间很小，那么不同 kernel 之间的资源分配就相对平衡，那么就不需要抢占。我们的测试中观察到这个指标能帮助算法稳定下来，并减少过度频繁的抢占。

#### 2.4.3.2 资源分区算法

按需资源分配试图保证 kernel 的全局资源平衡。如果一个 kernel 的主要资源是寄存器，那么这个 kernel 的  $rK$  是通过计算所有 SM 上的寄存器的使用除以所有 SM 寄存器的总容量得出的。我们观察到这个算法无法控制每个 SM 内部的资源分配。SMK 的目的是增加互补的 kernel 在一个 SM 内部的资源利用率。但是，按需资源分配算法无法

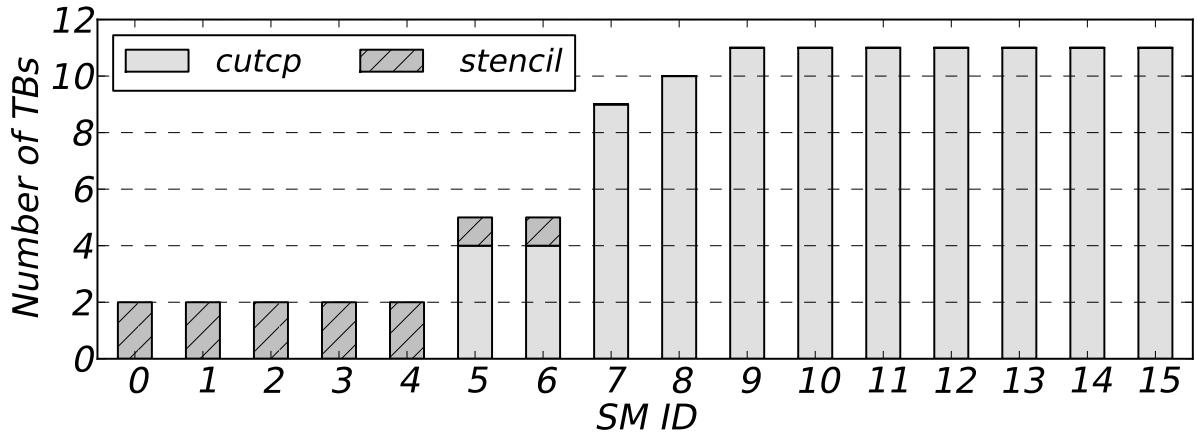
图 2-4 *cutcp+stencil* 使用按需资源分配时的 TB 分布

Figure 2-4 The TB distribution of *cutcp+stencil* with On-Demand Allocation.

保证这一点。因此，这个算法经常会导致许多 SM 上只运行一个 kernel，这样就和空间划分差不多了。图 2-4 展示了使用按需资源分配算法运行 *cutcp+stencil* 时的 TB 的分布。如图所示，SM 0-4 只在运行 *stencil*，而 SM 7-11 则只运行了 *cutcp*。只有 2 个 SM 在一起运行两个 kernel。这限制了 SMK 的效果。值得一提的是实现公平的资源分配不止一种方法。为了充分发挥 SMK 的效果，我们提出在 TB 发射之前为每个 kernel 对资源进行划分。我们将这个算法称为资源分区算法，一个 kernel 所分配的资源被称为一个资源分区。

SM 驱动器使用 DRF 策略<sup>[45]</sup> 在 SM 里创建资源分区。资源分区的目的是在一个 SM 内平衡其中 kernel 的  $rK$ 。如图 2-5 所示，分区的划分以一个空的 SM 开始，此时每个 kernel 的  $rK$  都是 0。然后 SM 驱动器会选择一个 kernel，比方说  $K_1$ ，并假设发射了一个来自  $K_1$  的 TB 来更新  $rK_1$ 。接下来，SM 驱动器会选择  $rK$  最低的 kernel，比方说  $K_2$ ，SM 驱动器会假设发射了一个来自  $K_2$  的 TB 来更新  $rK_2$ 。这一过程会一直迭代，直到再也没有 TB 可以被发射到这个 SM 上。也就是说，SM 上至少有一种资源被用尽了。这样，每个 kernel 在这个 SM 上的资源分区也就自然的被创建了出来。每个 kernel 在这个迭代过程中所占据的资源也就是这些 kernel 的资源分区。这个过程总是试图降低  $rK$  的区间：每次迭代都是选择一个  $rK$  最小的 kernel 进行发射，所以其  $rK$  就会更接近最高的  $rK$ ，从而降低了  $rK$  的区间。

使用资源分区时 TB 发射算法如图 2-6 所示。SM 驱动器首先像按需资源分配算法一样寻找  $K_c$  和  $SM_c$ 。然后，SM 驱动器检查  $SM_c$  上是否已经有了一个  $K_c$  的资源分区。如果有，并且那个分区里还有空闲的位置，SM 驱动器就会发射（或换入）一个 TB。如

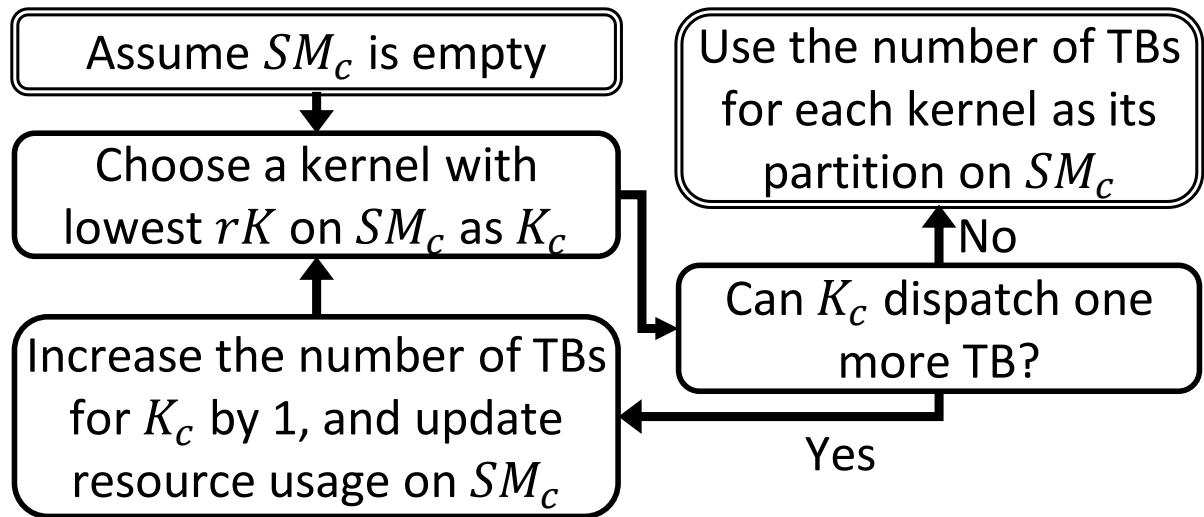


图 2-5 计算一个 SM 内的资源分区

Figure 2-5 Calculating resource partitions on one SM.

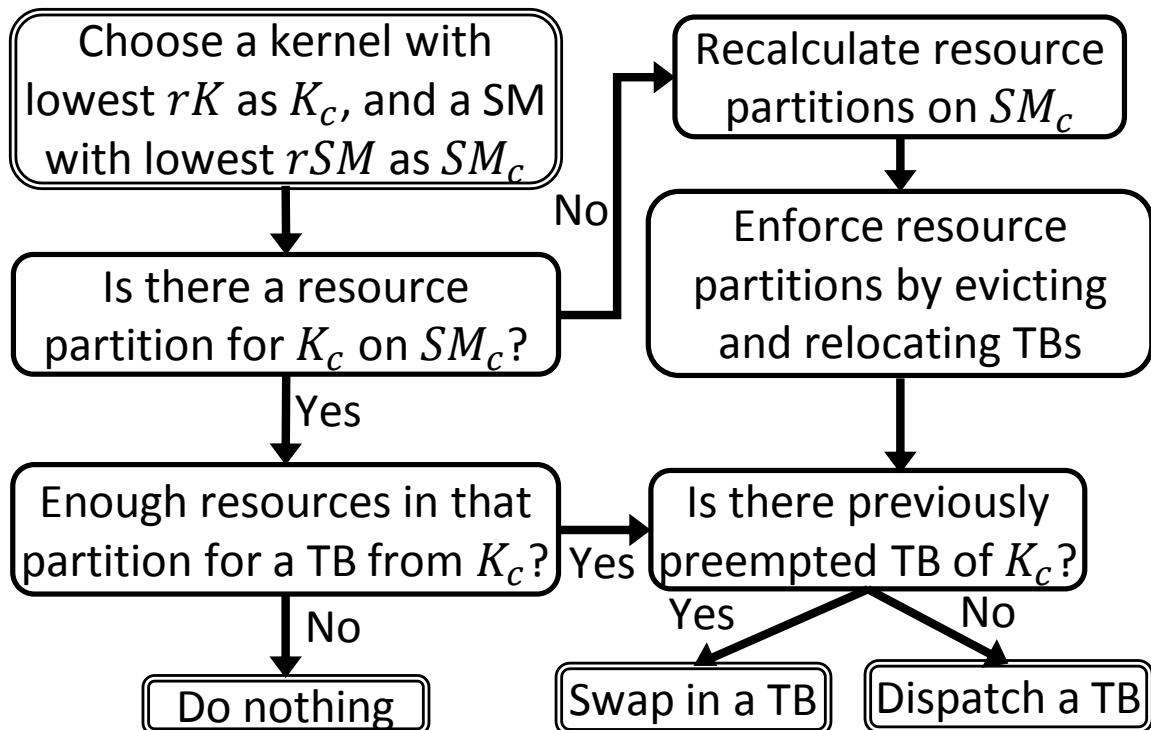


图 2-6 资源分区下的 TB 发射算法

Figure 2-6 TB dispatch with resource partitioning.

果有分区，但分区里没有位置，那么 SM 驱动器就放弃这次操作。

如果  $SM_c$  上还没有  $K_c$  的资源分区，那么 SM 驱动器就会开始在  $SM_c$  上启动创建资源分区的过程。新的资源分区会通过切出或重定位已经在  $SM_c$  上的 TB 来实现。这样每个资源分区内的 TB 都是对齐的。重定位通过换出和换入 TB 实现。即使这样引入了额外的开销，相比 Spart，我们还是可以观察到很好的加速比。这些抢占开销也可以通过 SM 内部的额外存储空间来回避。

TB of  $K_1$ : 10% Register, 0% Shared memory, 6.66% Thread Number  
TB of  $K_2$ : 3% Register, 6% Shared memory, 5% Thread Number

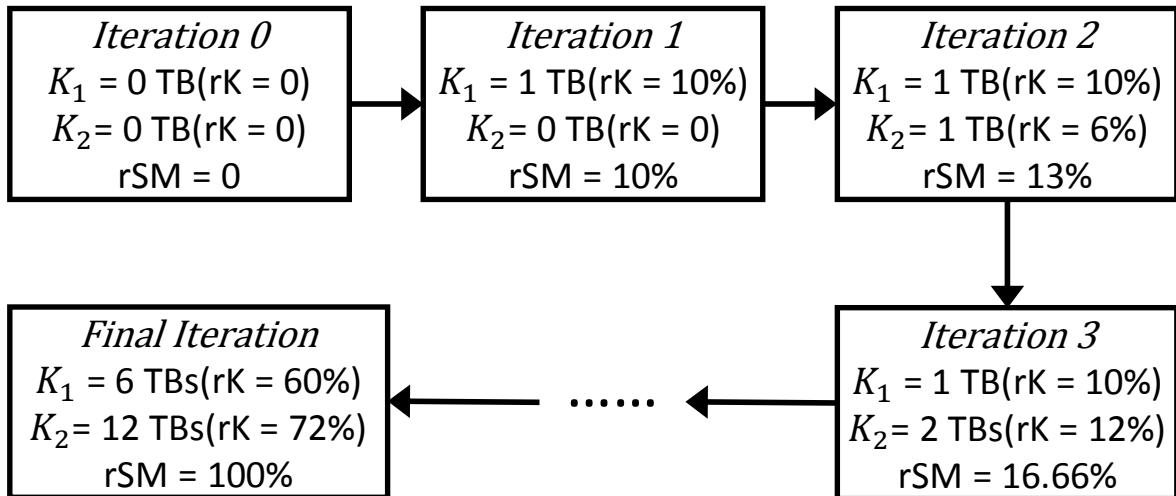


图 2-7 一个划分 2 个 kernel 的资源分区的例子

Figure 2-7 An example of resource partitioning for 2 kernels.

图 2-7 展示了一个资源分区的例子。假设 SM 驱动器需要在一个 SM 上为两个 kernel,  $K_1$  和  $K_2$ , 创建资源分区。假设一个  $K_1$  的 TB 带来的  $rK$  是 10%，而一个  $K_2$  的 TB 带来的  $rK$  是 6%。那么在迭代 0，即初始状态下，SM 是空的。在迭代 1 时，假设发射了一个来自  $K_1$  的 TB，把  $rK_{K_1}$  提高到了 10%。在这时， $K_2$  的  $rK$  更小，因此在迭代 2 时就会选择  $K_2$ ，将  $rK_{K_2}$  提高到 6%。这时 SM 的主要资源是寄存器，所以  $rSM$  就提高到了 13%（寄存器使用的和）。现在  $K_2$  的  $rK$  依然是最低的，所以迭代 3 还是选择发射一个来自  $K_2$  的 TB，从而  $rK_{K_2}$  增长到了 12%。在这次迭代之后，SM 的主要资源就变成了线程数，而  $rSM$  变成了 16.66%。迭代 4 没有显示在图上，但这次迭代是选择了  $K_1$ 。这个过程会一直迭代，直到  $rSM$  到达 100%，即当 SM 的线程数已经完全用尽。在那时， $K_1$  和  $K_2$  的资源分区也就形成了， $K_1$  最多可以发射 6 个 TB，而  $K_2$  最多可以

表 2-3 使用公平资源分配时的资源和性能的公平性（越高越好）

Table 2-3 Resource and performance fairness with fair resource allocation (higher is better).

kernel 对类型	资源公平	性能公平
计算 + 内存	0.94	0.49
计算 + 计算	0.94	0.64
内存 + 内存	0.88	0.55
所有	0.92	0.54

发射 12 个 TB。

利用资源分区算法，每个 SM 都为其 kernel 创建了资源分区，保证了多个 kernel 可以在同一个 SM 上一起执行。我们的实验显示这一资源分配算法显著提升了 GPU 的吞吐率，展现了 SMK 的优势。

#### 2.4.4 通过 warp 调度算法实现的公平动态资源分配

按需资源分配算法和资源分区算法都只管理了静态资源，即寄存器等资源的分配。在运行时，来自不同 kernel 的 TB 的 warp 由 warp 调度器调度来使用计算时钟周期。现在有许多 warp 调度算法<sup>[46, 47]</sup>，但这些算法都是为了提高一个 kernel 的性能而设计的。当多个 kernel 一起执行的时候，我们观察到即使 kernel 之间的静态资源分配是公平的，kernel 之间仍然会对计算时钟周期产生严重的竞争。例如，一个计算密集型 kernel 可能完全独占了计算时钟周期，完全剥夺了其他 kernel 的执行机会。一个内存密集型 kernel 则可能填满了未命中状态寄存器 (Miss Status Holding Register, MSHR)，导致其他不是内存密集型的 kernel 也因为内存访问而阻塞。正在阻塞的 kernel 的 warp 不能用来掩盖其他 warp 的延迟。因此，仅凭静态资源的公平分配不能保证每个 kernel 的性能也是公平的。

为了量化公平性，我们采用了在 Eyerman 等人工作<sup>[48]</sup> 中定义的指标，即所有 kernel 中最小的归一化 IPC (归一化到单独运行时的 IPC) 除以最大的归一化 IPC。我们把这个指标扩展到静态资源分配里，也就是最小的  $rK$  除以最大的  $rK$ 。表 2-3 展示了使用资源分区算法时，针对不同类型的 kernel 对（比如内存密集型加计算密集型）的平均资源公平性和性能公平性。1 代表完美的公平，而 0 则代表有一方被饿死了。我们观察到资源分区算法可以保证很好的静态资源分配的公平性，但性能公平则比较低，和静态资源分配的公平性也没有直接联系。因此，我们需要提出可以分配动态资源，即计算时钟周期的算法。

为了实现性能公平，我们定义计算时钟周期的公平分配为一个 kernel 按比例占据其在一个 SM 上单独执行时所使用的时钟周期。这一比例由 SMK 所分配的资源（即 TB

的数量) 相对于其单独在一个 SM 上执行时的资源来决定。直观上说, 如果一个 kernel 在单独执行时每个 SM 上有  $T$  个 TB, 并且  $x\%$  的周期被用于发射指令, 那么在 SMK 里这个 kernel 被分配的时钟周期, 设为  $C_k$ , 就是  $x\% \times \frac{S}{T}$ , 其中  $S$  是在 SMK 下所分配的 TB 的数量。所有 kernel 的  $C_k$  的和不应超过 100%。因此, 每个 kernel 的  $C_K$  都按比例缩小, 定义为  $Quota_K$ :

$$Quota_k = \frac{C_k}{\sum_{\text{for all } k} C_k}$$

在实现这个分配的时候, 我们需要获得一个 kernel 单独执行时的  $x$  和  $T$ 。这通过为每个 kernel 单独分配一个 SM 来采集。我们的实验表明, 这一分析的好处超过了使用更少的 SM 来一起运行 kernel 的损失。直接跟踪每个 kernel 的计算时钟周期并不现实, 这是由于 kernel 之间的执行会有重叠的部分。因此, 我们使用一段时间内每个 warp 调度器的每时钟周期 warp 指令数 (Warp Instructions per Cycle, WIPC) 来近似估计计算时钟周期。在每个时间段里, 每个 warp 调度器都为 kernel  $k$  分配  $Quota_k \times Epoch\_length$  条指令。这些指令平均的分配到所有的 warp 调度器上, 这是因为 TB 在发射到 SM 上的时候, 其中的 warp 也是平均分配到 warp 调度器上的。当一个 warp 调度器从一个 kernel 发射一条指令的时候, 其指令配额减 1。如果一个 kernel 的指令配额减到了 0, 那么 warp 调度器就不会从那个 kernel 发射新的指令。如果所有的 kernel 的指令配额都减到了 0, 那么新的指令配额会根据最近一次的 WIPC 来计算和重新分配。

例如, 假设有两个 kernel,  $K_1$  和  $K_2$ 。每个 kernel 在单独执行时都可以运行 8 个 TB。在 SMK 里, 假设在一个 SM 上  $K_1$  有 2 个 TB,  $K_2$  有 4 个 TB。每个 warp 调度器的  $K_1$  和  $K_2$  的 WIPC 分别为 0.4 和 0.5。那么,  $C_1 = 0.1$ , and  $C_2 = 0.25$ . 并且  $Quota_1 = \frac{0.1}{0.35}$  and  $Quota_2 = \frac{0.25}{0.35}$ .

对时钟周期的分配保证了一个 kernel 发射的指令数和其一个 SM 里的 TB 数量之间的关系。这样, kernel 的 warp 就可以被 warp 调度器以相对公平的方式进行调度。为了简化设计, 我们这里使用了一个对执行时间的线性估计。尽管这样的估计可能高估了 kernel 的性能, 但是这对分配的影响不大, 因为所有的 kernel 的性能都被高估了。我们的目标是在 kernel 之间平衡计算时钟周期, 而非精确的估计其性能。

#### 2.4.5 增加并发 kernel 的数量

在第 2.4.3 节和第 2.4.4 节中描述的算法可以支持每个 SM 上有任意数量的并发 kernel。但是, 一个 SM 里容纳更多的 kernel 可能导致潜在的问题。第一个问题就是 L1 和 L2 缓存的竞争可能会更加激烈。我们观察到有更多的 kernel 并不意味着 L1 和 L2 缓存

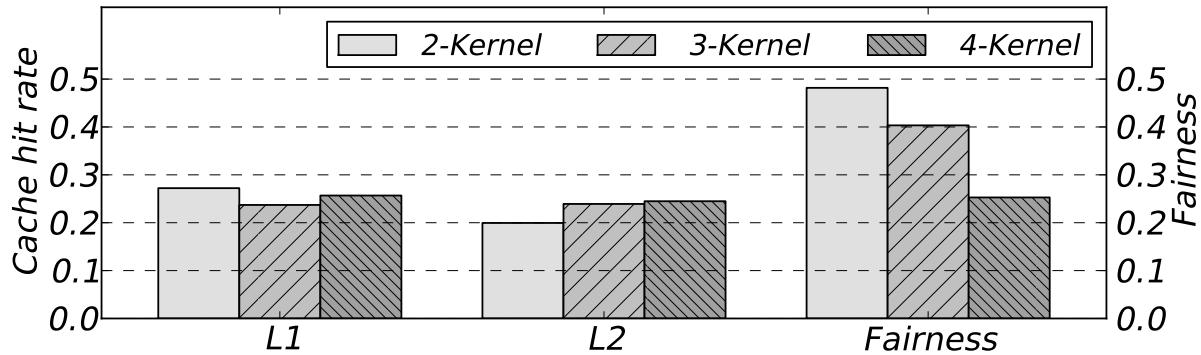


图 2-8 不同数量的 kernel 一起在 SM 上执行时的 L1 和 L2 缓存命中率和公平性

Figure 2-8 L1/L2 cache hit rate and fairness for different number of kernels coexisting within one SM. The 4 kernels used are *lmb*, *sgemm*, *stencil*, and *tpacf*.

的命中率一定会下降。图 2-8 中展示了一个 kernel 组合的例子。这是因为 warp 调度器可以捕捉到 kernel 内的局部性，并把对应的工作集保留在 L1 和 L2 缓存中。因此，L1 缓存命中率并不一定会减少。第二个问题是寄存器和共享内存可能会因为 kernel 资源需求的不匹配而导致资源碎片。资源碎片会降低资源的利用率。我们的资源分区算法通过重定位 TB 保证了每个 kernel 的资源分区内部是没有碎片的。碎片只会在 kernel 的资源分区之间产生。因此，在我们的实验中，我们没有观察到足以影响性能的显著的资源碎片的情况。

但是，我们观察到了当每个 SM 的 kernel 数量增加的时候，公平性有所下降。这是由于我们使用单独的 SM 来线性的估计每个 kernel 单独执行时的性能。这一估计有一定的误差。有些 kernel 的性能被高估，而有些则被低估。如果 warp 调度器原本的算法对 warp 并不公平（如 greedy-then-oldest, GTO 算法），那么调度器可能倾向于执行某些 kernel，导致了性能好的 kernel 和性能不好的 kernel。随着每个 SM 上 kernel 数量的增加，其他 kernel 会占据性能最不好的 kernel 的执行时间，导致其执行时间会越来越少。最终导致了公平性下降。更好的性能估计（如离线估计）或公平的 warp 调度算法（如轮询算法）可以用于解决这个问题。我们的研究表明每个 SM 上有 2 个 kernel 可以给出最好的公平性和返回时间。

如果我们限制了每个 SM 上至多有 2 个 kernel， $rK$  可以被扩展成包括了一个 kernel 所占据的 SM 的数量。由于我们使用了公平的资源分配算法，这自然会让所有 kernel 占据相同数量的 SM。例如有三个 kernel,  $K_1$ ,  $K_2$  和  $K_3$ ，运行在 3 个 SM 上，那么产生的分配就是  $(K_1, K_2)$  在  $SM_1$  上， $(K_2, K_3)$  在  $SM_2$  上， $(K_1, K_3)$  在  $SM_3$ 。

表 2-4 GPGPU-Sim 的模拟参数

Table 2-4 Simulation parameters for GPGPU-Sim.

GPU 参数	值	SM 参数	值
核心频率	1216MHz	寄存器	256KB
内存频率	7GHz	共享内存	96KB
SM 数	16	线程数	2048
内存控制器	4	TB 数	32
调度策略	GTO	Warp 调度器	4

## 2.5 实验验证

这一节中，我们展示和讨论了我们实验的结果。首先，我们描述了我们实验验证所使用的实验方法，包括使用的性能评测程序和模拟参数。然后，我们展示了不同方法在处理 2 个 kernel 并发执行时的系统吞吐率，平均归一返回时间和公平性。在这之后，我们检查了执行中的阻塞时钟周期，抢占开销和 3 个与 4 个 kernel 并发执行时的性能。最后，我们讨论了我们方案的硬件开销。

### 2.5.1 实验方法

我们使用最新版本的 GPGPU-Sim<sup>[49]</sup>，一个被广泛使用的 GPU 模拟器来验证我们的设计。表 2-4 里的模拟参数来自于 Nvidia Geforce GTX980<sup>[50]</sup>，来反映最近的架构中每个 SM 都变得更大的现象。我们使用 Parboil<sup>[51]</sup> 性能评测程序集中的 10 个性能评测程序。在模拟多程序运行环境时，我们列举了所有的配对，共 45 个。*bfs* 只使用了很少的资源，其可以与其他 kernel 共存而不影响性能，因此我们将这个程序排除在外。除了评测抢占开销的部分，我们使用 Parboil 中最大的数据集运行程序。在评测抢占开销时，我们使用最小的数据集来模拟程序频繁抢占的场景。我们修改了 GPGPU-Sim，令其支持多个 kernel 运行在同一个 SM 上。我们假设寄存器和共享内存如之前的工作<sup>[52]</sup> 中所述一般按照线性寻址。

我们首先测试了三个 SMK 的变种：按需资源分配 (**SMK**)，资源分区算法 (**SMK-P**) 和把通过 warp 调度实现的公平动态资源分配应用在 SMK-P 上 (**SMK-(P+W)**)。SMK-(P+W) 的时间段长度为 10k 时钟周期。然后，我们比较了我们最好的方法，SMK-(P+W) 和空间划分方法 (**Spart**)。我们按照 Tanasic 等人工作<sup>[33]</sup> 中的描述实现了 Spart。在实验中，我们使用了三个 Eyerman 等人工作<sup>[48]</sup> 里提出的指标：系统吞吐率 (System Throughput, STP)，公平性和平均归一化返回时间 (Average Normalized Turnaround Time, ANTT)。系统吞吐率是所有测试程序的归一化 IPC (归一化到单独运行时的 IPC) 的和。类似的，平

均归一化返回时间是归一化返回时间的算术平均值。公平性（在 0 到 1 之间）是最小的归一化 IPC 除以最大的归一化 IPC。

我们每组测试运行两百万个时钟周期。根据之前的研究<sup>[34]</sup>，只要模拟时间大于一百万个时钟周期，结果就是准确的。如果程序在两百万个周期前结束，程序会被重新执行。如果一个程序包含多个 kernel 或是一个 kernel 被执行了多次，我们会把指令和时钟周期加起来计算 IPC。

作为比较，45 对程序被分成了三组。我们按组汇报平均值：“C+C”组是两个程序都是计算密集型的（比如 *cutcp*, *mri-q*, *sgemm* 和 *tpacf*），“M+M”组是两个程序都是内存密集型的（比如 *histo*, *lmb*, *mri-g*, *sad*, *spmv* 和 *stencil*），，“C+M”组则是一个程序是内存密集型的，另一个程序是计算密集型的。我们只汇报“C+M”组的详细结果，这是由于这部分结果是最有趣的。

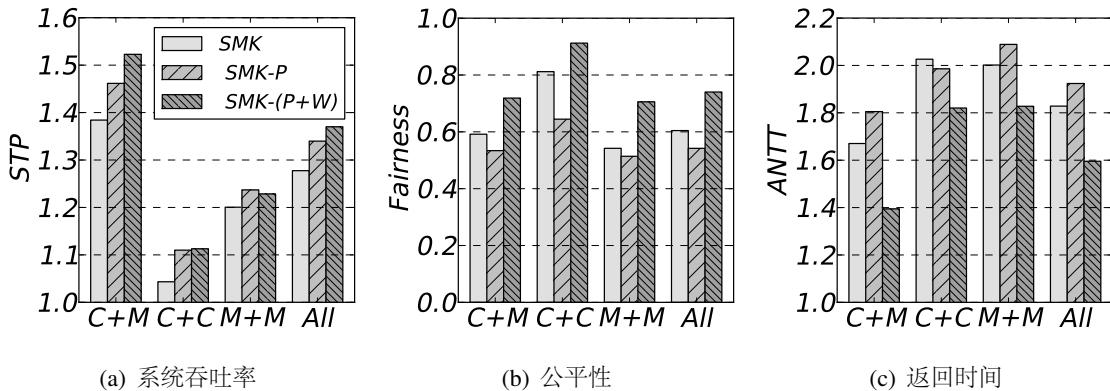


图 2-9 SMK 设计的比较

Figure 2-9 Comparison of the SMK designs.

## 2.5.2 SMK 设计的结果

图 2-9 和图 2-10 展示了 SMK 不同版本的结果。由于 SMK 是为了一起运行资源互补的 kernel 而设计的，其效果在“C+M”组是最好的。平均来看，在“C+M”组里 SMK, SMK-P 和 SMK-(P+W) 相比单独运行 kernel，系统吞吐率分别提高了 38%, 46% 和 52%。整体的平均提升则分别是 28%, 34% 和 37%。SMK-P 通过资源分区保证了 kernel 可以共享一个 SM，从而吞吐率相比 SMK 有了提高。在大多数情况下，SMK-(P+W) 可以通过平衡 kernel 的执行进度来创造更多相互掩盖的执行，从而进一步提高吞吐率。但是，有些时候 SMK-(P+W) 为了公平性也会限制一些 kernel，导致互相掩盖的机会变少。

从公平性的角度来看（图 2-9(b)），SMK-(P+W) 的公平性是最好的。这是由于它保证了每个 kernel 都有公平的执行机会。图中越接近 1，代表不同 kernel 的性能变化

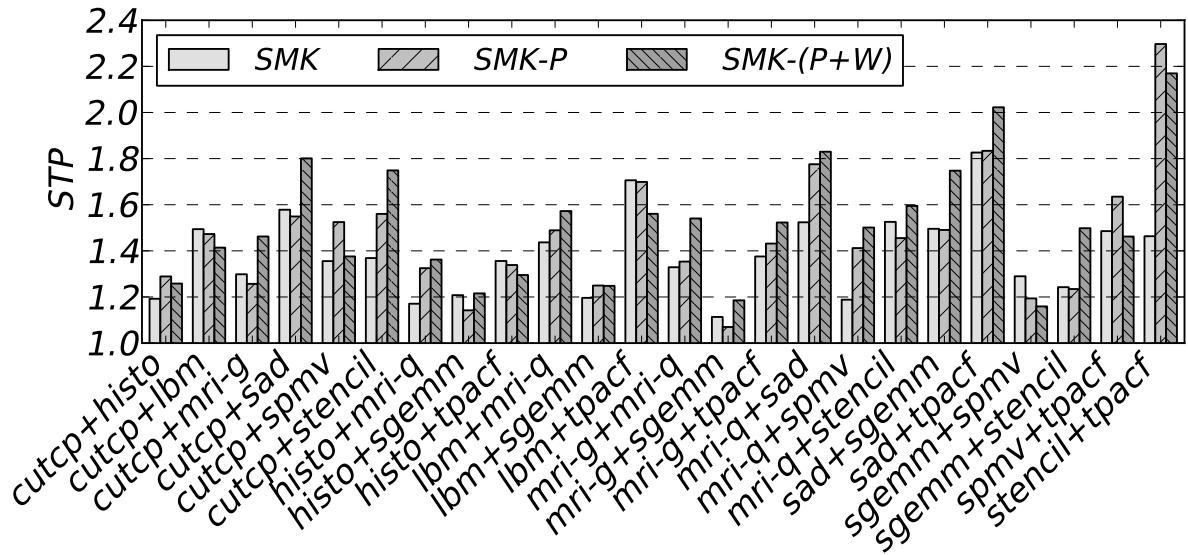


图 2-10 “C+M” 组的系统吞吐率

Figure 2-10 STP for kernel pairs in the “C+M” group.

越一致，即公平性越好。平均来看，SMK，SMK-P 和 SMK-(P+W) 的公平性分别为 0.6, 0.54 和 0.74。SMK-(P+W) 有效的提升了一起执行的 kernel 的公平性。例如，我们观察到 SMK-P 在 *mri-q+sgemm* 和 *sad+sgemm* (图中没有显示) 两组中的公平性特别的差，分别为 0.121 和 0.124。但是在 SMK-(P+W) 里它们的公平性提升到了 0.81 和 0.56。

对于返回时间来说，越接近 1 代表其返回时间越接近单独运行时的返回时间，也就是越好。如果吞吐率和公平性都很好，返回时间 (图 2-9(c)) 也会很好。如果其中一项不好，那么返回时间也会变长。由于 SMK-(P+W) 的吞吐率和公平性都是最好的，其返回时间也是最低的。SMK-P 的公平性不好，因此返回时间也比较长。

### 2.5.3 与空间划分方法的比较

图 2-11 和图 2-12 比较了 SMK-(P+W) 和 Spart<sup>[33]</sup>。总的来说，SMK-(P+W) 在大多数情况下超过了 Spart，最高的提升有 61.3%。在 Spart 比 SMK-(P+W) 好的情况下，其吞吐率的差距不超过 5%。

SMK-(P+W) 在 “C+M” 组中表现最好，这是因为计算密集型 kernel 和内存密集型 kernel 在计算时钟周期上天然互补。例如在 *lbm+tpacf* 中，SMK-(P+W) 的系统吞吐率比 Spart 高了 20% (见图 2-12)。这一提升来自两个方面。首先，SMK-(P+W) 的内存利用率更高，这是由于内存密集型 kernel 的 TB 都均匀的分布在 SM 里，从而充分利用了 SM

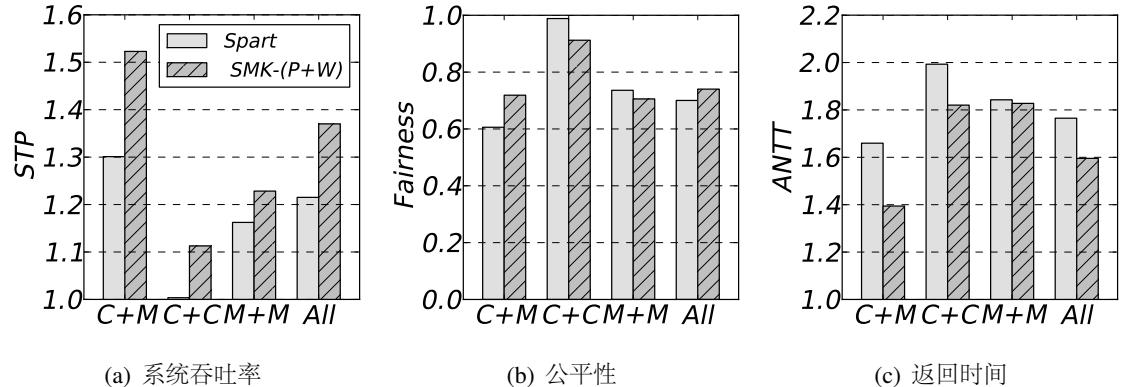


图 2-11 SMK-(P+W) 和 Spart 的比较  
Figure 2-11 Comparison of SMK-(P+W) and Spart.

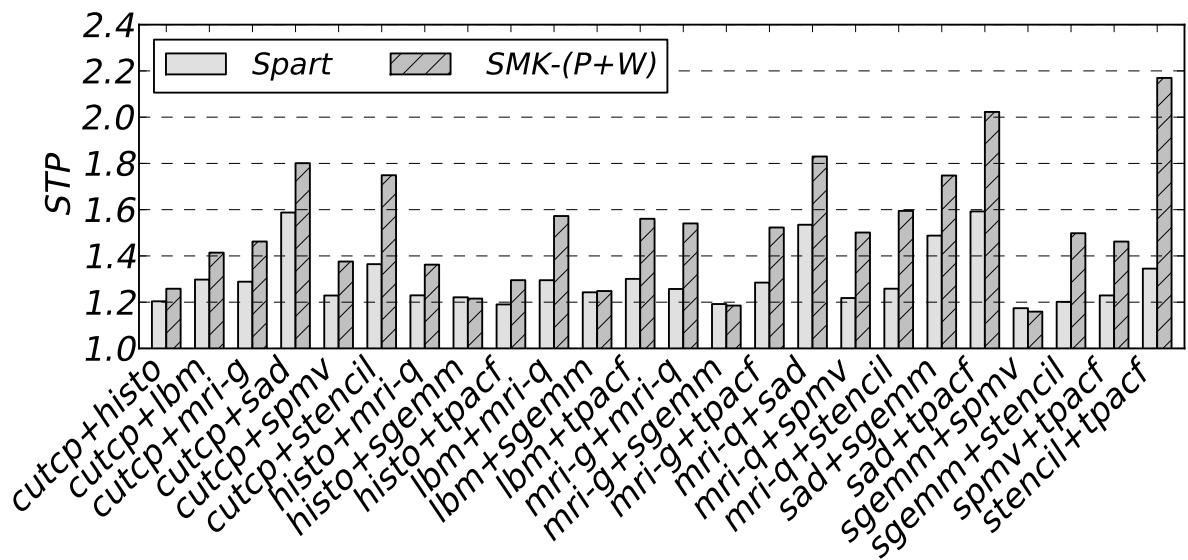


图 2-12 SMK-(P+W) 和 Spart 的“C+M”组的系统吞吐率  
Figure 2-12 STP of SMK-(P+W) vs. Spart for the “C+M” group.

里内存访问相关的组件（如 MSHR）。其次，对于计算密集型 kernel *tpacf* 来说，其有许多短延迟的指令可以掩盖内存密集型 kernel 的长延迟内存访问指令。平均来说，“C+M”组的吞吐率相比单独执行提升了 52%，相比 Spart 提升了 30%。

如果有 kernel 是缓存敏感型的，那么 SMK-(P+W) 相比 Spart 的提升就更大了。具体来说，在 Spart 中，每个 SM 都被分配了一个 kernel，并用该 kernel 的 TB 填满。但是，这可能会导致这些 TB 竞争缓存，导致缓存性能下降。对于一个计算密集型 kernel 来说，其使用片上高速暂存空间来存储数据，SM 的 L1/L2 缓存资源并没有得到充分的利用。有了 SMK，缓存敏感型 kernel（如 *sad* 和 *stencil*）和计算密集型 kernel 可以有效的组合到一起。这样缓存敏感型 kernel 可以占据缓存（实现高命中率），而不会影响另外一个 kernel。最显著的提升来自于互补的组合，例如 *stencil* 是缓存敏感型 kernel，其在和另一个计算密集型 kernel 搭配的时候性能很好。

“C+C”组里的 kernel 也获益于 SMK。线程级并行度的提升的让这些 kernel 可以更加充分的利用 GPU 的资源，也有了更多的机会掩盖内存访问的延迟。例如 *cutcp+mri-q* 相比 Spart 就有 15% 的提升。

但是对于“M+M”组的 kernel，共享一个 SM 并不能增加线程级并行，内存带宽也仍然是瓶颈。SMK 在这种情况下有可能降低性能。但是如图 2-11(a)所示，其平均吞吐率仍然比 Spart 高 5.1%

图 2-11(b)和图 2-11(c)展示了我们的设计相比空间划分方法还提升了公平性和返回时间。SMK-(PW+) 通过对 warp 执行的细粒度控制改善了公平性，并大幅降低了返回时间。

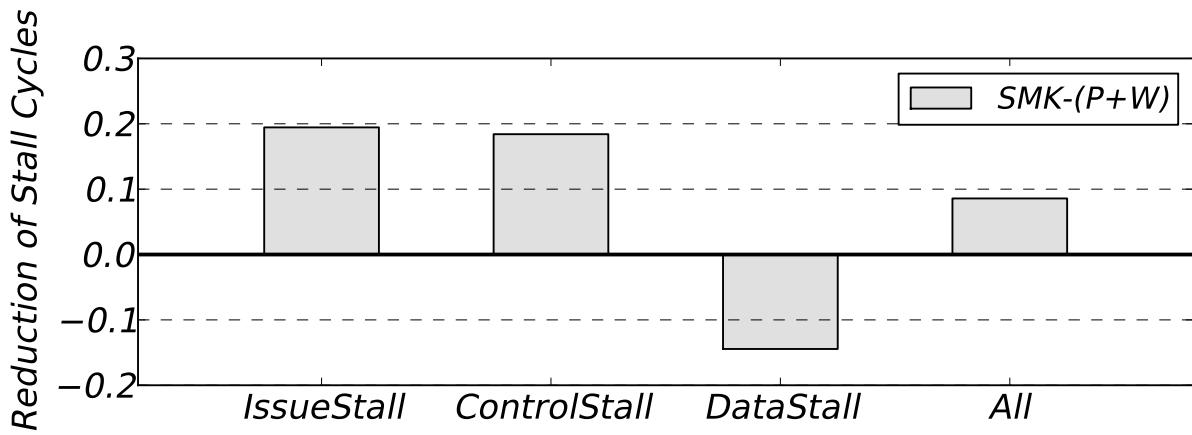


图 2-13 相比 Spart 减少的阻塞时钟周期

Figure 2-13 Reduction of stall cycles over Spart.

### 2.5.4 阻塞时钟周期

图 2-13 展示了不同类型的阻塞时钟周期在“C+M”组里所减少的百分比。阻塞有如下三种形式：指令流水线满，控制冲突和数据冲突。如图所示，SMK 显著减少了来自指令流水线的阻塞。这是因为 SMK 利用了来自不同 kernel 的指令，其有更多的机会发射不同类型的指令。例如，如果 SM 正在执行一个计算密集型 kernel，那么指令流水线经常会是满的，而存取单元则是空闲的。在 SMK 的情况下，来自另一个 kernel 的内存访问指令可以被发射到存取单元上，减少流水线阻塞的比例。SMK 还通过取指和执行的掩盖减少了控制冲突导致的阻塞。不过，SMK 的数据冲突比 Spart 多。这是因其在流水线中有更多正在执行的指令。更多的正在执行的指令导致了 SM 中记分牌里的项变多，也就更有可能导致数据冲突。总体来说，SMK-(P+W) 减少了 19.5% 的流水线阻塞，18.4% 的控制冲突阻塞，并增加了 14.5% 的数据冲突阻塞，总体减少了 8.6% 的阻塞时钟周期。

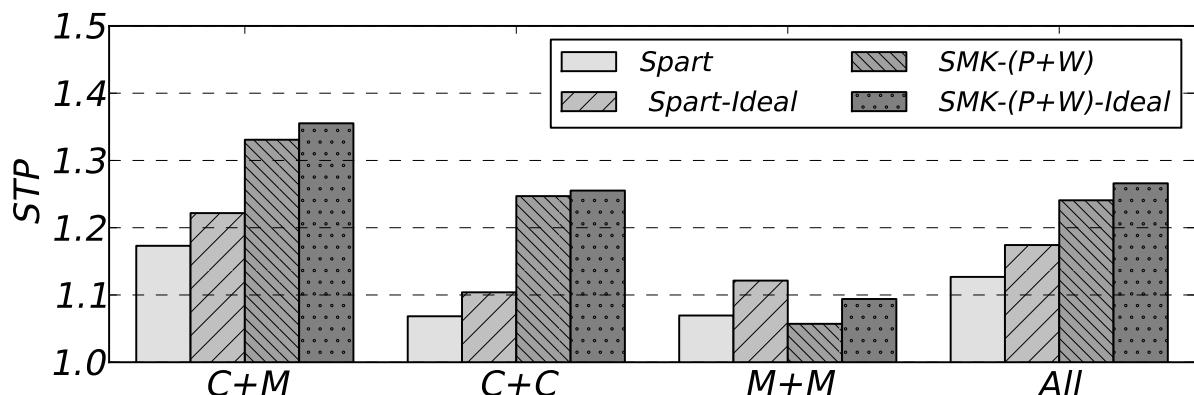


图 2-14 不考虑抢占开销时的吞吐率

Figure 2-14 Throughput without preemption overhead.

### 2.5.5 抢占开销

我们还测试了 SMK-(P+W) 相比 Spart 的抢占开销。抢占不仅需要花时间把上下文存到内存里，还会对内存子系统造成额外的压力。我们通过运行一个理想环境的模拟来测试这一效果，即在这种情况下保存和读取上下文的数据传输代价是 0。我们把这种理想环境设为 Spart-Ideal 和 SMK-(P+W)-Ideal。我们用小数据集作为输入来模拟那些对抢占代价更敏感的长度较短的 kernel 的执行情况。

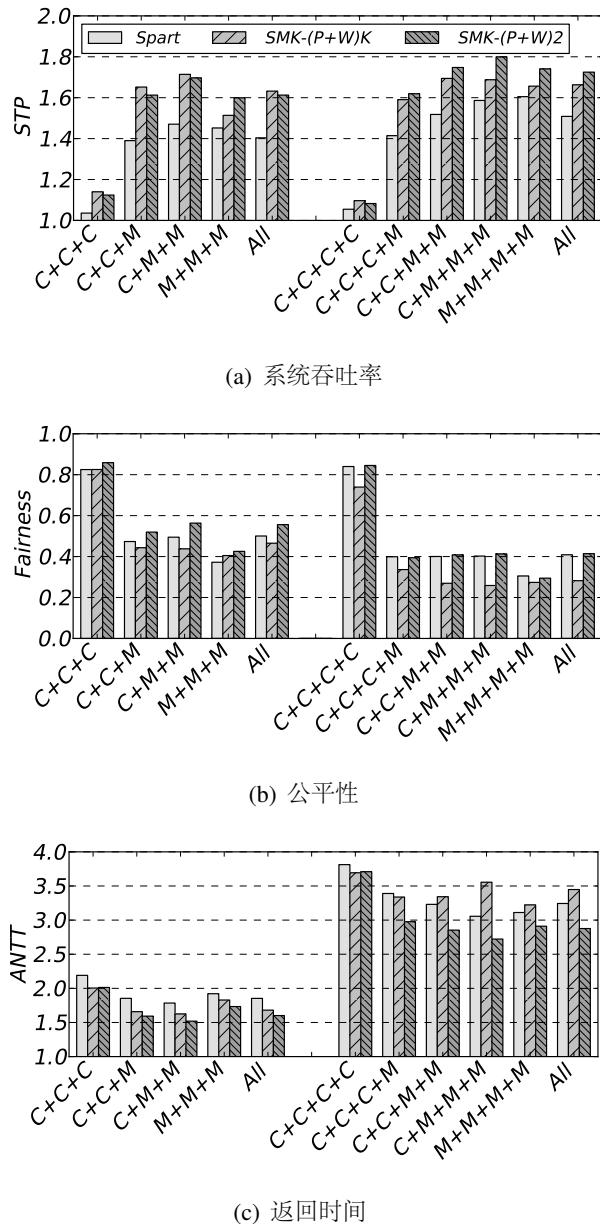


图 2-15 SMK-(P+W) 和 Spart 在 3 个和 4 个 kernel 时的比较  
Figure 2-15 SMK-(P+W) vs. Spart for three and four kernels.

如图 2-14 所示，抢占开销对 SMK 的影响较小。Spart 和 Spart-Ideal 的差距是 5%，而 SMK-(P+W) 和 SMK-(P+W)-Ideal 的差距是 2%。SMK-(P+W) 的抢占开销更小是因为其让抢占和正常的执行互相掩盖。抢占 TB 就像是执行一个内存密集型 kernel 一样。另外，结果还显示平均情况下，有抢占开销的 SMK-(P+W) 的性能比 Spart-Ideal 还好，这展示了我们提出的设计的有效性。

### 2.5.6 3个或4个 kernel

为了研究我们方案的可扩展性，我们测试了运行 3 个或 4 个 kernel 时的性能。我们随机选择了 3 个 kernel 的 56 个组合和 4 个 kernel 的 70 个组合。被选择的组合覆盖了所有可能的计算密集和内存密集 kernel 的组合种类。我们测试了两个设计，**SMK-(P+W)K**，其在一个 SM 里尽可能多的运行 kernel，和 **SMK-(P+W)2**，其限制了一个 SM 上最多有 2 个 kernel。如图 2-15(a)所示，吞吐率方面，SMK 对 3 个和 4 个 kernel 的扩展性很好。其整体相比 Spart 分别有 14.8% 和 14.3% 的提升。SMK-(P+W)2 和 SMK-(P+W)K 在 3 个 kernel 的情况下吞吐率类似，但是 4 个 kernel 的情况下 SMK-(P+W)2 的吞吐率更好。综上，多运行 kernel 可以提高系统吞吐率。

如图 2-15(b)所示，在有更多的 kernel 的情况下，控制不同 kernel 的进度更加的困难，因此当每个 SM 有多于 2 个 kernel 的情况下，公平性就低了一些。我们在第2.4.5节讨论了这个问题。**SMK-(P+W)K** 的公平性是最差的。

图 2-15(c)展示了返回时间的结果。随着 kernel 数量的增加，返回时间也随之增加，但并非线性。**SMK-(P+W)2** 的返回时间是最低的。

综上，SMK 在 2 个、3 个和 4 个 kernel 的情况下相比 Spart 都有很好的吞吐率的提升和很好的公平性。因此 SMK 是一个扩展性良好的设计。

表 2-5 硬件开销

Table 2-5 Hardware overhead.

设计	开销
PCS	与之前的工作 <sup>[33]</sup> 类似
按需资源分配	主要是逻辑电路
资源分区算法	每个 kernel 和 SM 7 个计数器
Warp 调度	每个 kernel 和 SM 4 个计数器

### 2.5.7 硬件开销

要实现 SMK，SM 驱动器需要使用下列控制逻辑进行扩展：(1) TB 发射算法；(2) 控制抢占引擎的信号。同时，warp 调度器也需要实现对共享 kernel 的限额。

表2-5展示了我们设计的硬件开销。部分上下文切换的硬件开销和完全上下文切换类似<sup>[33]</sup>，也就是需要类似被抢占 TB 队列等硬件组件。对于按需资源分配算法，其需要对应的逻辑电路。算法所需的性能信息在当前的 GPU 架构中已经可以直接获得。对于资源分区和 warp 调度算法，其需要一些计数器来存储资源划分和限额的信息。

具体的来说，资源分区算法每个 SM 上的每个 kernel 需要 7 个计数器来记录寄存器，共享内存，线程数和 TB 数的分区范围。对于一个有 16 个 SM，每个 SM 最多 4 个 kernel 的 GPU 来说，其总开销为 448 ( $7 \times 4 \times 16$ ) 个计数器。类似的，warp 调度算法需要 4 个计数器 (1 个采集信息，1 个跟踪时间段，2 个存储限额信息) 来采集信息和跟踪限额信息。因此其总开销为 256 个计数器。总的来说，主要的开销是逻辑电路，如果计数器使用 32 位，那么存储的开销大概为 2.8KB。

## 2.6 本章小结

当前的 GPU 不支持抢占式调度，而抢占式调度对现代系统十分重要。之前的工作提出了抢占机制和共享 GPU 的策略，但没有充分利用 GPU 的资源。我们提出的新型设计使 GPU 可以支持多个 kernel 以细粒度的方式共享 GPU。另外，我们提出了几个策略来充分发挥这一机制的潜力。我们不仅维持了 kernel 之间的资源公平性，还保证了 kernel 的执行也是公平的。实验结果显示我们的设计在保证良好公平性的前提下显著的提升了系统吞吐率。这一工作发表在 IEEE CAL 和 HPCA 2016 上。

## 第三章 细粒度共享 GPU 的 QoS 支持

### 3.1 研究背景

GPU 已经被广泛用于数据中心来加速计算密集型应用，例如 MapReduce<sup>[29]</sup>，图计算<sup>[53]</sup> 和深度学习<sup>[54, 55]</sup>。GPU 有大量被组织成流式多处理器（Streaming Multiprocessors, SM）<sup>[19]</sup> 的简单计算核。GPU 利用这些核的线程级并行来掩盖内存访问的延迟。

在现在的 GPU 系统中，多个应用可能需要同时共享同一块 GPU。例如在桌面计算中，高清视频播放器需要和在线视频通话一起运行。虽然多个程序共享 GPU 可以用各种方法实现<sup>[33, 38, 41]</sup>，但现在的 GPU 硬件架构并不能管理共享 GPU 的程序的 QoS。图像处理这类任务需要及时返回以保障良好的用户体验。不能满足 QoS 的话就会导致不好的用户体验，例如游戏卡顿或掉帧。非图形程序也可能有性能需求。例如，数据中心中和云计算场景下的用户可能需要他们的应用以特定的速度进行处理。

GPU 提供 QoS 的一种方法是使用修改过的硬件驱动，系统调用或 API 来调度 GPU 指令（如设备初始化，数据传输和发射 kernel 等）并控制多个 kernel 的执行顺序<sup>[40-42]</sup>。在这种方法下，程序以时分复用的方式共享 GPU，并且调度的粒度是一个 kernel。由于当前的 GPU 硬件不支持抢占，一旦一个 kernel 被发射到 GPU 上，那么它就不能被中断。其他应用程序必须等待正在运行的 kernel 结束。因此，这些方法只适用于短 kernel，而运行时间长的 kernel 可能会长时间的阻止其他 kernel 运行。

最近，工业界开始研究 GPU 的抢占机制。异构系统架构（The Heterogeneous System Architecture, HSA）标准<sup>[56]</sup> 定义了三种类型的抢占：(1) 软抢占，即硬件可以推迟抢占请求；(2) 硬抢占，即硬件需要把上下文保存到内存中；(3) 上下文重置，即硬件会丢掉当前正在执行的 kernel 的上下文。

除了工业界的进展，学术界也在探索硬抢占<sup>[33, 57, 58]</sup> 和上下文重置<sup>[32]</sup> 的机制。同时，利用抢占机制，除了可以时分复用 GPU，还可以让多个程序通过切换正在执行的 kernel 和新的 kernel 来一起同时使用一块 GPU，从而提高系统运行效率<sup>[33, 57, 58]</sup>。在 Tanasic 等人的研究中<sup>[33]</sup>，上下文切换以一个 SM 的粒度来进行，即 kernel 的上下文按照整数个 SM 来切换。因此，不同的应用的 kernel 可以同时执行在不同的 SM 上。公平性和 QoS 则通过调整每个 kernel 的 SM 的数量来实现<sup>[33, 59, 60]</sup>。但是其他研究表明<sup>[30, 33, 58]</sup>，许多领域的应用都有资源利用率不高的问题，并且这一问题发生在 SM 内部。由于每个 SM 仍然只运行一个 kernel，在 kernel 之间划分 SM 不能解决这个问题。因此，通过调整 SM 的数量来控制公平性或 QoS 粒度太粗，性能也不能达到最优。

一种改进的共享方法是在每个 SM 内部运行多个 kernel<sup>[30, 57, 58, 61, 62]</sup>。这些方法给多个 kernel 分配 SM 内部的资源，而不是在 kernel 之间划分 SM。这种细粒度的共享可以由部分上下文抢占实现<sup>[57, 58]</sup>。这个技术以 TB 为单位切换 kernel 的上下文。这种粒度更细的抢占比切换一整个 SM 代价更小。相比粗粒度共享，这种细粒度共享的资源利用率更高，吞吐率也更高<sup>[57, 58]</sup>。公平性则通过为 kernel 分配 SM 内部的资源来实现。由于粒度更细，其效果比粗粒度的共享更好<sup>[58]</sup>。

在这一章中，我们提出了一个细粒度共享的多任务 GPU 的 QoS 机制。公平性和 QoS 的区别是前者只要求让 kernel 之间的性能保持一致，而后者则需要保证一部分 kernel 的性能。因此，其资源分配选择截然不同。我们的研究表明细粒度共享下的 QoS 比粗粒度共享下的 QoS 效果更好，可扩展性也更好。有 QoS 目标的 kernel 只会接受足以让他们完成目标的资源，而剩下的 kernel 则可以用剩下的资源来最大化他们的吞吐率。我们方案相比粗粒度共享的主要优势是我们通过调整 warp 调度来控制每个 kernel 的进度。这一方法相比调整 SM 数量的粗粒度方法能更准确的达到 QoS 目标。由于更精确的控制，我们的方法对更多个 kernel 的扩展性也更好。这一章主要的技术贡献如下：

- 细粒度共享 GPU 的轻量时钟周期级 QoS 管理技术。我们开发了一个简单的指令限额分配策略来控制每个 kernel 的进度。限额根据 QoS 目标设置，并与各种 warp 调度算法兼容。这一方法为 kernel 分配了刚刚好可以达到 QoS 目标的资源。而其他的资源则可以被用于提高 GPU 的整体吞吐率。
- kernel 间调整静态资源分配的算法。我们开发了一个为 kernel 分配 TB 的静态资源分配算法。这个算法为 kernel 提供了足够的线程级并行度来达到其 QoS 目标。这一算法也考虑到了上下文切换的开销，从而提高了整体性能。

在我们的实验验证中，我们使用了来自 Parboil 性能评测程序集的 90 个 kernel 对和 60 个 3-kernel 组对 10 个不同的 QoS 目标进行了测试。我们的方法不仅相比粗粒度共享方法可以达到更多的 QoS 目标，其对 kernel 数量和 QoS 目标的可扩展性也更好。平均来说，相比粗粒度方法，我们的方法在 3-kernel 组里达到的 QoS 目标要多 43.8%，在 kernel 对里达到的 QoS 目标要多 12.2%。总的 GPU 吞吐率在 3-kernel 组里平均提高了 20.5%，在 kernel 对里平均提高了 15.9%。

## 3.2 相关工作

虽然人们对 CPU 的 QoS 调度已经有了很多的研究<sup>[64–68]</sup>，但是这些研究都不能直接应用在 GPU 上。这是因为 GPU 并不支持 CPU 的抢占式时分复用。如果直接把 CPU 上的方法应用到 GPU 上，GPU 的性能会有很大损失。我们的实验也展示了这一点。现在的研究表明让多个 kernel 共享 GPU 可以提高资源利用率和整体吞吐率<sup>[30, 33]</sup>，也有研究

表 3-1 细粒度 QoS 和之前方法的比较

Table 3-1 Comparison between fine-grained QoS and prior work.

	CPU QoS	Kernel Fusion <sup>[38]</sup>	SMK <sup>[58]</sup>	Spatial QoS <sup>[59]</sup>	Warped-Slicer <sup>[61]</sup>	Baymax <sup>[63]</sup>	Fine-grained QoS
软件/硬件	软	软	硬	硬	硬	软	硬
QoS 感知	✓			✓		✓	✓
适用于 GPUs		✓	✓	✓	✓	✓	✓
抢占	✓		✓	✓			✓
主动共享 GPU		✓	✓	✓	✓		✓
SM 内的共享		✓	✓		✓		✓
细粒度性能控制	✓						✓
自适应的 TLP					✓		✓

探索了如何通过共享 GPU 提高整体性能<sup>[34, 61]</sup>，但由于 GPU 缺乏硬件级的共享支持，对 GPU QoS 的研究仍然很有限。

在系统层面，一些运行时系统试图控制 GPU 上程序的 QoS。TimeGraph<sup>[42]</sup> 通过调度 GPU 的命令队列来控制 QoS。也有方法通过在操作系统内核拦截和调度 GPU 的命令来控制 QoS<sup>[41]</sup>。这类方法要么需要开源驱动，要么需要对私有驱动进行逆向工程<sup>[43]</sup>，因此在实践中难以被采用。Baymax<sup>[63]</sup> 通过预测一个 kernel 的执行时间，并对队列里的 kernel 进行排序，从而为所有 kernel 留出足够的执行时间来保证 QoS。Mystic<sup>[69]</sup> 则使用了机器学习算法来预测几个 kernel 是否可以高效的共享一个 GPU，并根据预测的结果在集群中分发 kernel。这些方法都只能在 kernel 这个级别进行调度，由于 GPU 不支持抢占，长时间运行的 kernel 会阻碍其他 kernel 的执行。因此，这些方法不能很好的控制长时间运行的 kernel。

这几个工作都依赖于对 GPU 程序性能的估计。因为他们都先要知道每条指令所需的时间，才能合理的对指令进行调度。在这一方面，3D 渲染类程序的不同操作可以被映射到不同的渲染 API，而这些 API 一次执行的代价是稳定的。因此这个代价可以通过历史信息来进行估计，GERM<sup>[70]</sup>, TimeGraph<sup>[42]</sup> 和 VGRIS<sup>[71]</sup> 这三个系统都使用了这个方法来估计指令的执行时间。但是，对于通用程序来说，不同的程序根据输入不同执行时间会有很大变化。因此这个方法不适用于通用程序。Pai 等人的研究<sup>[72]</sup> 发现同一个 kernel 里，每个 TB 的指令数是类似的，因此 TB 完成的数量和总的需要执行的 TB 数量可以用来估计一个 kernel 的进度。与之前的工作不同，我们把 kernel 执行速率转化为 IPC，并通过硬件机制保证 IPC。

一种绕过这个限制的方法是用软件方法允许多个 kernel 在 GPU 上运行。Kernel fusion<sup>[38]</sup> 和 KernelMerge<sup>[39]</sup> 静态的把来自两个 kernel 的代码合并成一份，并通过条件语句控制执行路径。Elastic kernel<sup>[30]</sup> 也使用了这种方法，但是专注于探索 kernel 共享一个 SM 时的好处。Lee 等人<sup>[31]</sup> 也有类似的研究。也有研究探索如何在运行时改变资源分配<sup>[37, 62]</sup>。在这些方法中，kernel 是在编译时决定的，因此新的 kernel 不能在运行时被发

射上去，也不能对 kernel 进行热切换。并且，这些方法需要 kernel 的源代码来重编译，这在很多情况下是不能实现的。

为了激活真正的运行时共享，之前的研究提出了硬件级抢占支持<sup>[32, 33, 57, 58]</sup>。但就如第 3.3 节所述，这些方法没有提供 QoS 控制的机制。

Aguilera 等人<sup>[59]</sup> 基于硬件级抢占机制提出了通过划分 SM 来实现 QoS 的策略。这个工作使用一个线性模型来分析 SM 的数量和 kernel 性能的关系。但是这个模型没有分析 kernel 之间对内存带宽的竞争这个因素，因此其预测性能时不够准确。同时，这个方法资源分配的粒度太粗，一次至少要分配一个 SM。表 3-1 总结了我们的方法相比之前 QoS 技术的主要创新点。

### 3.3 研究动机

传统的并发运行多个 kernel 的方法是使用多块 GPU，每块 GPU 上运行一个 kernel。这可以保证每个 kernel 的性能。但过去的许多研究表明在只运行一个 kernel 的时候 GPU 的资源利用率并不高，而用多个 kernel 共享一块 GPU 可以提高资源利用率，GPU 吞吐率和能效<sup>[30, 32, 33, 38, 57, 58, 61]</sup>。

当前，有 4 种共享 GPU 的机制。第一种是利用当前的 GPU 架构允许不同的应用并发的发射 kernel 的机制，如 Hyper-Q<sup>[35]</sup> 和 MPS<sup>[36]</sup>。但是，硬件并不能控制 kernel 的 TB 是怎么发射到 SM 上的，大多数情况下，这些 kernel 仍然是顺序执行的<sup>[37]</sup>。

第二种机制是通过软件方法把多个 kernel 通过代码转换的方法合并成一个 kernel<sup>[30, 38]</sup>。这种情况下，多个 kernel 实际上变成了 1 个 kernel，也就可以在一个 SM 里共存，实现细粒度共享。Xu 等人<sup>[61]</sup> 设计了一个基于在线分析的 TB 分配算法来提高性能和公平性。这一方法的限制是硬件只能看见一个 kernel，因此也就不能控制每个 kernel 的进度。因此，kernel 的性能和 QoS 就无法得到保证。另外，这一方法是静态的，不能服务于动态到来的高优先级 kernel。

第三种方法是通过在 CPU 端调度任务，来时分复用的共享 GPU<sup>[64-68]</sup>。这可以通过拦截不同应用的 kernel 的发射指令来实现。一个消耗了很多 GPU 时间的应用不能再发射 kernel，而需要把 GPU 让给其他更需要 GPU 时间的应用<sup>[63]</sup>。这一方法使用了 MPS<sup>[36]</sup> 来允许并发 kernel 执行。而这也受到 MPS 缺陷的限制，即大多数情况下 kernel 仍然是串行执行的。因此，这一方法不能提高资源利用率，GPU 吞吐率和能效。QoS 可以在多个应用之间平衡，但同时运行多个 kernel 的时候其仍然无法保证 QoS。图 3-1(a) 展示了一个例子。kernel  $K_1$  和  $K_2$  尽管在时间上有重复，但仍然只能顺序执行。

第四种方法是通过硬件抢占机制来实现共享，其每次保存一个 SM<sup>[32, 33]</sup> 或是一个 TB 的上下文<sup>[57, 58]</sup>。这样，新的 kernel 就可以在同一块 GPU 上执行，与之前的 kernel 共

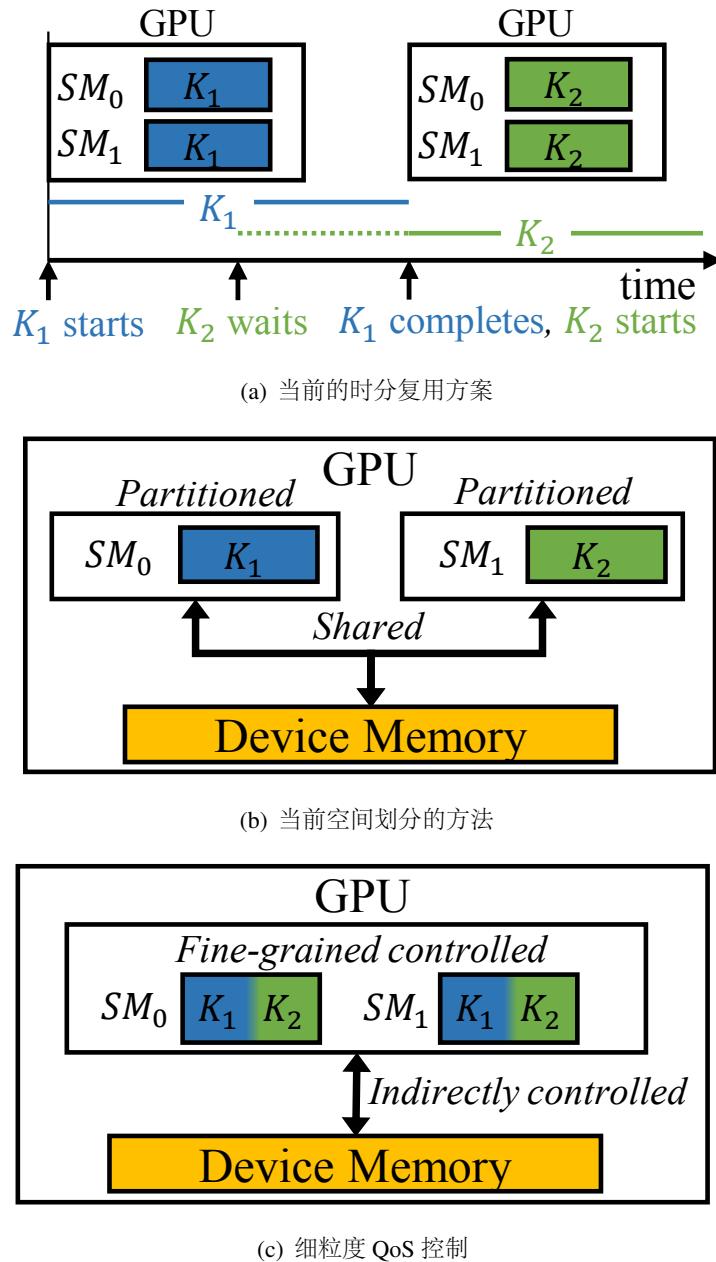


图 3-1 不同的共享 GPU 的方法

Figure 3-1 Different GPU sharing strategies.

享资源。这些 kernel 可以空间划分 SM<sup>[32, 33]</sup>, 如图 3-1(b)所示, 或是共享一个 SM 内部的资源<sup>[57, 58]</sup>, 如图 3-1(c)所示。基于抢占的共享也支持其他三种共享所能达到的效果。这个机制支持动态的切换 kernel, 还可以提高 GPU 的吞吐率。更重要的是, kernel 的 QoS 可以通过动态的划分资源来保证。例如, 空间划分方案里的 QoS 可以通过调整每个 kernel 的 SM 的数量来实现<sup>[59]</sup>。在细粒度共享方法, 如 SMK 中<sup>[57, 58]</sup>, 公平性可以通

过调节 kernel 的资源分配来实现，从而让每个 kernel 的性能损失相比单独运行时保持一致。但是，如果是要保证 QoS，那么就不应该使用这样为公平性设计的策略。

在这一章中，我们开发了一个细粒度共享 GPU 的 QoS 方法。我们的研究表明这一方法的 QoS 控制能力相比之前的方法都要好。

## 3.4 细粒度共享的 QoS 设计

我们假设在一起执行的 kernel 中，有一个或多个 QoS kernel。这种 kernel 具有 QoS 目标。而其他的 kernel 则是 non-QoS kernel。对于 QoS kernel 来说，其目标是满足每个 kernel 的 QoS 目标。对于 non-QoS kernel 来说，其目标是最大化其总吞吐率。QoS 管理机制会动态的分配资源，使每个 QoS kernel 都能满足其目标，同时让 non-QoS kernel 尽可能的提高吞吐率。

### 3.4.1 需要管理的资源

第一种资源是静态资源，例如寄存器，共享内存，线程数和 TB 数。更多的 TB 意味着更多的线程和更高的 TLP。有了足够的 TLP 才能保证 kernel 一直处于忙碌状态。但是 TLP 太高也会导致计算密集型程序出现缓存竞争<sup>[31]</sup>，或是内存密集型程序出现太多的内存访问请求<sup>[59, 73]</sup>。但是光靠分配静态资源是不能满足 QoS 的。这是因为 warp 调度器有可能会不管 TLP 而倾向某个 kernel。尽管如此，一个好的静态分配可以保证每个 kernel 的 TLP，从而允许动态资源管理机制快速找到可以满足 QoS 的资源分配方案。

第二种资源是动态资源，包括内存带宽和每个 SM 里的核计算时钟周期。我们的经验表明通过管理内存带宽来控制 QoS 是十分困难的，这是因为性能和内存带宽的关系被两层缓存，内存访问模式和 TLP 所影响。因此，要在运行时捕捉带宽和性能的关系十分困难。除此之外，一个 kernel 的带宽需求会在运行时发生变化<sup>[74]</sup>，这让基于历史信息的方法也不能奏效。同时，共享内存带宽相比划分内存带宽可以提供更好的性能<sup>[73]</sup>。因此，QoS 不能通过只管理内存带宽来实现。

核计算时钟周期由 warp 调度器来管理，其决定了每个时钟周期执行哪个 warp 的指令。这是一个更直接的控制 SM 里 kernel 的性能的方式。但是，warp 调度算法已经为一个 kernel 的缓存局部性，内存行为和同步操作做了很多优化<sup>[49, 75, 76]</sup>。细粒度共享的多 kernel 的 warp 调度应该保留原来的 warp 调度算法的性质以提高性能。

因此，QoS 控制不能通过只管理一种资源来实现。我们提出的方案整合了静态和动态资源的管理来实现 kernel 的 QoS 目标。

### 3.4.2 从 QoS 目标到体系结构指标

QoS 目标通常在应用层面指定，与硬件无关。因此这个目标不能直接用在硬件架构上进行 QoS 控制。因此，我们解决的第一个问题就是把高层的 QoS 目标转化为底层的，硬件可以衡量和控制的体系结构指标。QoS 目标可以有不同的形式，如帧率或数据处理率。在许多如视频处理类型的 GPU 应用中，一个常见的编程方法是用一个 kernel 处理一帧数据。这时帧率就等价于 kernel 完成速率，而目标帧率可以通过规定每个 kernel 的完成时间来达到。这一方法也被之前 GPU 的 QoS 工作所使用<sup>[63]</sup>。因此，我们的 QoS 管理方法保证了 kernel 的运行时间，或是平均 IPC (即 IPC 目标)，而不是 kernel 在执行过程中的执行速度。

kernel 的执行时间包括数据传输时间，这随着架构的变化而变化。在集成式 GPU 中，数据传输代价可以忽略。操作系统可以直接把对应的物理内存区域映射到 GPU 的虚拟内存空间中。在独立式 GPU 中，数据需要通过 PCI-E 总线从 CPU 复制到 GPU 上。为了简化模型，我们的设计不考虑通过异步操作对数据传输进行掩盖的优化。因此，数据传输时间和数据传输量成线性关系，并可以通过 PCI-E 总线的带宽和延迟计算得出。要从 kernel 完成时间计算出 IPC，我们需要一个 kernel 的总指令数，这可以通过机器学习算法来在运行时预测<sup>[63]</sup>。有了这些信息，IPC 可以用下列公式计算得出：

$$IPC = \frac{Instructions\_of\_Kernel}{Frequency \times (Kernel\_Time - Data\_Transfer\_Time)}$$

在我们的实验测试中，我们假设一个应用的 IPC 目标 ( $IPC_{goal}$ ，从 QoS 目标转化得来) 至多是其单独运行时的性能  $IPC_{isolated}$  (即  $IPC_{isolated} \geq IPC_{goal}$ )。在具体程序中，每个应用的 IPC 目标都会随着用户的需求而变化。为了测试我们方法的通用性，我们遍历测试了从低 (50% 的  $IPC_{isolated}$ ) 到高 (95% 的  $IPC_{isolated}$ ) 的各种 IPC 目标。结果展示了我们方法的有效性和通用性。

#### 对 OS kernel 调度器的好处。

我们提出的机制可以增强 OS 级 kernel QoS 调度器的功能<sup>[41, 42, 63, 69]</sup>。OS 级调度器要么假设 kernel 单独执行<sup>[41, 42]</sup>，要么假设 kernel 以不受控的方式并发执行<sup>[63, 69]</sup>。一旦 kernel 被发射到 GPU，其执行进度就不受到调度器控制。因此这些调度器依赖于控制 kernel 发射到 GPU 上的时间来控制 QoS。我们的设计填补了控制 kernel 该分配多少资源以达到 QoS 的空白，这样即使 kernel 开始得晚，其也有机会达到 QoS 目标。这样，我们的机制可以让 OS 级调度器放松对一个 kernel 发射到 GPU 上的时间的要求。

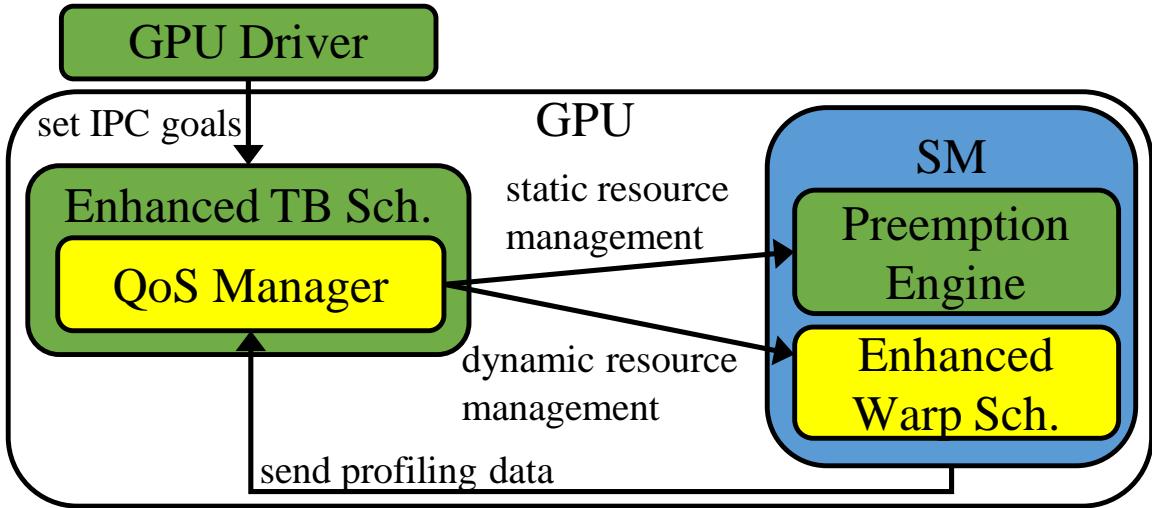


图 3-2 细粒度共享 QoS 体系架构扩展（黄色标注）概览

Figure 3-2 Overview of architecture extensions (in yellow) for QoS in fine-grained sharing of GPUs.

### 3.4.3 体系架构概览

图 3-2展示了我们的 QoS 方案的硬件架构扩展。当前的 GPU 可以让来自不同 kernel 的 TB 在一个 SM 里运行，例如通过 MPS<sup>[36]</sup>。虽然这一过程不能被控制，但其提供了支持这一特性的硬件架构如 TLB 和内存管理。这些组件被之前的工作<sup>[32, 33, 58]</sup> 和这一章的方案继承了下来。因此，我们只关心和我们工作相关的设计。原来的 TB 调度器的功能通过细粒度共享的方案<sup>[58]</sup> 和 QoS 管理器进行了增强。增强后的 TB 调度器 (Enhanced TB Scheduler) 可以与每个 SM 进行通信，来管理静态和动态资源。静态资源管理机制决定了每个 kernel 应该在 SM 里有多少个 TB。这一资源分配根据 QoS 目标在运行时通过抢占机制实现。动态资源管理机制通过限额算法决定了每个 kernel 的执行进度。QoS 管理器把限额传给具有 QoS 感知能力的增强 warp 调度器 (Enhanced Warp Scheduler, EWS)。静态资源管理和动态资源管理都是必须的，前者保证了 kernel 和动态资源管理机制有足够的线程级并行度来保证 kernel 的进度。而 QoS 管理器则通过收集运行时数据来做出限额分配的决定。

**基于限额的管理方案。** EWS 通过 QoS 管理器指定的限额来分配核计算时钟周期。SM 里的每个 kernel 都有一个限额来表明其在一个时间段内的执行进度。这一策略与上一章所提出的细粒度 kernel 公平管理策略兼容，这让 QoS 管理可以和公平管理共存。GPU 固件可以根据运行时的需求切换不同的策略。

就如我们之前提到的，QoS 控制机制只需要保证 QoS kernel 的进度。一个简单的方

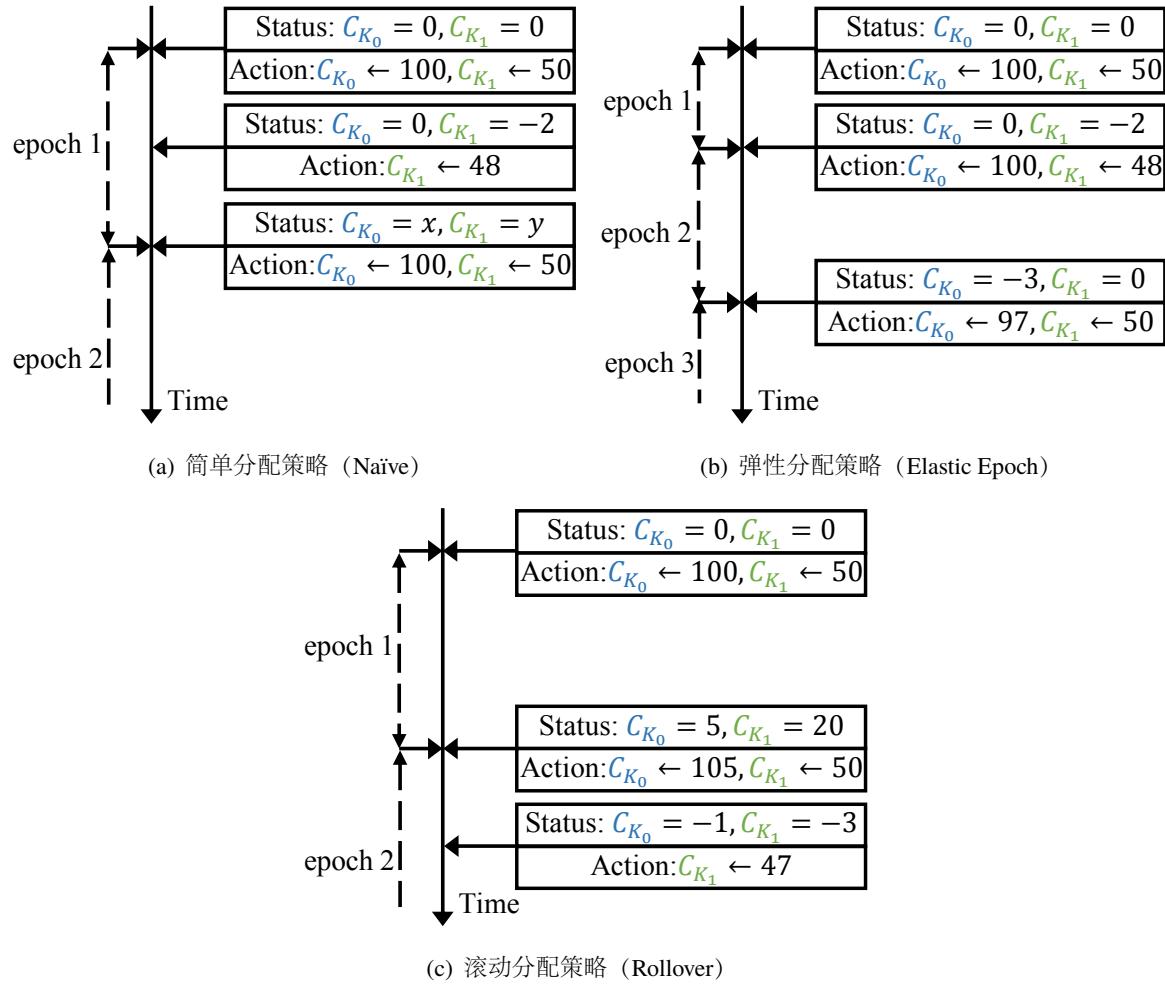


图 3-3 限额分配策略概览。 $K_0$  是一个 QoS kernel,  $K_1$  是一个 non-QoS kernel,  $C_{K_i}$  是 kernel  $K_i$  的限额计数器

Figure 3-3 Overview of quota allocation schemes.  $K_0$  is a QoS kernel,  $K_1$  is an non-QoS kernel, and  $C_{K_i}$  is the quota counter for kernel  $K_i$ .

法是先给 QoS kernel 分配限额，并只允许这些 kernel 运行。当他们在这个时间段的目标达到了之后，再让剩下的 kernel 在余下的时间里执行。这一方法让执行退化到了顺序执行。这个设计与 CPU 里的 QoS 管理类似，每个线程都被分配了一个时间片来分时复用 CPU。每个线程的性能就与它们总共分到的时间片线性相关。但在 GPU 里，kernel 的执行是互相重叠的。分配给一个 kernel 的时间片也有可能被其他并行执行的 kernel 所消耗。其次，如果 kernel 顺序执行，GPU 的并行执行能力就没有被利用起来，这会导致低下的资源利用率和 GPU 吞吐率。我们在实验验证环节里报告了这种类似 CPU 的 QoS 管理方案的结果。

利用限额机制，EWS 就像正常的 warp 调度器一样调度，但是在每个时钟周期它都检查是否有 kernel 达到了其限额。如果 kernel 已经花费掉了其所有的限额，EWS 就不会发射任何来自那个 kernel 的指令。由于这一设计只会在 kernel 的限额用完时限制 warp 的选择，而在其他时间并不影响原来的 warp 调度算法，因此其对 kernel 的性能几乎没有影响。使用限额同时也间接的控制了内存带宽的消耗。这是因为一个不能发射指令的 QoS kernel 不会产生任何内存访问的请求。

### 3.4.4 QoS 算法

QoS 管理的目标是在满足 QoS kernel 的 QoS 目标的前提下最大化 non-QoS kernel 的吞吐率。管理限额的分配是我们设计中的一个关键部分。正如上文所讨论的，一个应用的 QoS 需求可以转化为  $IPC_{goal}$ 。因此限额也就可以表达成每个时间段应该执行的指令数。由于  $IPC_{goal}$  是一个 kernel 的平均 IPC，因此每个时间段的实际 IPC 可能会随着 kernel 的运行时行为的变化而变化。因此，我们需要一个有效的 QoS 算法来管理 IPC。我们开发了四种限额分配算法。

#### 3.4.4.1 简单 (Naïve) 分配算法。

我们开发的第一种算法是简单分配算法。其根据  $IPC_{goal}$  和时间段的长度 ( $T_{epoch}$ ) 来计算出限额。给定  $IPC_{goal}$  和  $T_{epoch}$ ，我们可以算出一个 kernel  $k$  在一个时间段内应该完成的指令数：

$$Quota_k = IPC_{goal} \times T_{epoch} \quad (3-1)$$

这一限额  $Quota_k$  代表了所有 SM 在一个时间段内应该完成的指令数。QoS 管理器为每个 kernel 计算出限额并将其按每个 SM 所容纳的 TB 的数量的比例分发至每个 SM。例如，如果  $k$  一共有  $T$  个 TB，而  $SM_i$  有  $T_i$  的 TB，那么  $SM_i$  收到的限额 ( $quota_k$ ) 就

是  $Quota_k \times \frac{T_i}{T}$ 。因此， $Quota_k$  以一种平衡的方式分发到每个 SM 上，以充分利用每个 SM 的 TLP。

$Quota_k$  在每个时间段的开始分配。假设有一个本地计数器  $C_k$  来存储  $quota_k$ 。当一个  $k$  的 warp 指令完成的时候， $C_k$  就会减去那个 warp 指令实际完成的指令数（由于分支，可能会小于 32）。如果在时间段结束的时候还有剩余的限额，那么简单分配算法会丢弃这部分限额，并在下个时间段重置  $C_k$  到  $quota_k$ 。

如果  $C_k$  在时间段结束之前就变为 0 或是负数，这意味着这个 SM 还可以执行更多的指令，但是 QoS kernel  $k$  已经达到了其目标。因此其可以把剩下的时钟周期分配给 non-QoS kernel。我们也会为 non-QoS kernel 规定限额以防止其占用 QoS kernel 所需的资源。Non-QoS kernel 的限额分配策略会在第 3.4.5 节讨论。简单分配算法会检查是否一个 SM 上所有的  $C_k$  都是 0 或是负数，如果是的话，non-QoS kernel 的  $C_k$  会加上  $quota_k$  以保证其继续运行。对于 QoS kernel 来说，一旦其 IPC 目标被完成，这个时间段内就不会再给它新的限额。简单分配算法会在每个时间段的开始把所有 kernel 的  $C_k$  重置到  $quota_k$ 。

图 3-3(a) 展示了一个简单分配算法的例子。 $K_0$  是一个 QoS kernel，其限额为 100。 $K_1$  是一个 non-QoS kernel，其限额为 50。它们的限额在时间段 1 开始被分配。在时间段结束之前，两个 kernel 的限额都用完了，这时  $C_{k_0}$  是 0，而  $C_{k_1}$  在减去执行完毕的指令数之后已经减到了负数。这样， $C_{k_1}$  就会加上 50，而  $C_{k_0}$  保持不变。在时间段 1 的结束（即时间段 2 的开始）时，两个 kernel 的限额都会被重置，没用完的限额会被丢弃。

#### 3.4.4.2 基于历史信息的限额调整

简单分配算法的一个明显的缺点是其没有考虑 kernel 运行时行为的变化。有些时候 QoS kernel 的运行速度并不能达到其平均 IPC。也就是说， $Quota_k$  是根据 kernel 的整个执行时间的平均  $IPC_{goal}$  计算得出的。而某些时间段里 kernel 并不能达到这个 IPC，并且其他时间段里的 IPC 也不会高于  $IPC_{goal}$ 。结果就是每个时间段的 IPC 都不大于  $IPC_{goal}$ ，导致这个 kernel 根本达不到  $IPC_{goal}$ ，也无法满足 QoS。

为了解决这个问题，我们可以稍微增加一些  $Quota_k$ ，这样让 kernel 最终的 IPC（很可能比  $Quota_k$  所指定的小）能够达到  $IPC_{goal}$ 。 $Quota_k$  的增加量由所有过去的时间段所达到的 IPC 决定 ( $IPC_{history}$ )。

$$Quota_k = \alpha_k \times IPC_{goal} \times T_{epoch}$$

$$\alpha_k = \max\left\{\frac{IPC_{goal} \text{ of } k}{IPC_{history} \text{ of } k}, 1\right\}$$

$\alpha_k$  在每个时间段的开始计算得出。因此，如果  $IPC_{history}$  比  $IPC_{goal}$  小， $Quota_k$  就会增加。例如，如果一个 kernel 的  $IPC_{goal}$  是 125，而过去的时间段的平均 IPC 是 100，那么  $\alpha_k$  就是 1.25，并且分配给  $k$  的限额就会乘上 1.25。给  $k$  更多的限额让其有更多的动态资源可以达到  $IPC_{goal}$ 。

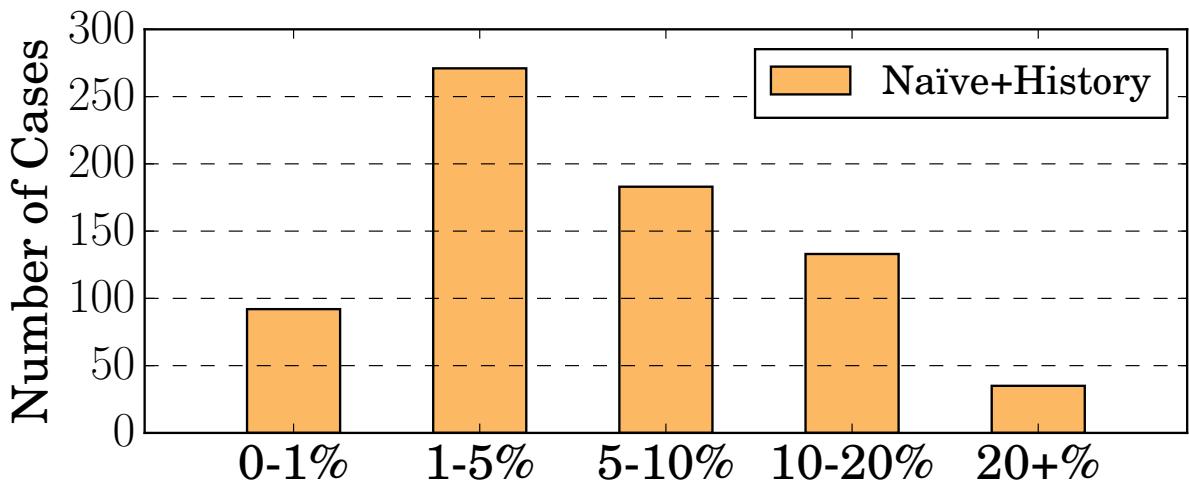


图 3-4 带有基于历史信息的限额调整的简单分配算法未满足  $IPC_{goal}$  的情况的数量（总共 900 种情况）和其离  $IPC_{goal}$  的差距

Figure 3-4 The number of cases (out of 900) that Naïve Allocation with History-based Adjustment misses the  $IPC_{goal}$  vs. how much it misses the goal.

图 3-4 展示了基于历史信息的限额调整的策略错过 QoS 目标的数量（总共 900 种情况）。我们根据其离 QoS 目标的距离对这些失败的例子进行了分组。我们可以看到即使有带有历史信息的限额调整，还是总共有超过 700 个情况都没能满足 QoS 目标。不过其中大部分都离目标不到 5%。在成功的情况下（186 个），其平均超过了 QoS 目标 1.3%。我们观察到根据当前限额的消耗速度（快或慢）可以在分配新限额的时候做出更好的决定。因此我们开发了另外两种方法，弹性分配算法和滚动分配算法来解决这两个场景下的问题。

#### 3.4.4.3 弹性分配算法.

图 3-4 展示了许多情况下 QoS kernel 虽然没达到 QoS 目标，但距离都不远。但是，我们测试了一些更激进的  $\alpha$  的调整策略，其会提高 QoS kernel 的性能，但是会降低整体的吞吐率。

因此，我们开发了一个弹性分配算法。弹性分配算法不再固定时间段的长度。一旦所有的 kernel 都在时间段结束之前消耗完了所有的限额，新的时间段会立刻开始。如图 3-3(b)所示， $K_0$  和  $K_1$  的限额在执行开始时分配，当他们的限额在时间段 1 结束之前被用完（即  $C_k$  为 0 或负数）时， $C_k$  的值就会立刻被加上限额的值，就好像时间段 2 开始了一样。

#### 3.4.4.4 滚动分配算法.

如果限额消耗的太慢，这些没有使用的限额意味着上一个时间段里 QoS kernel 没有达到规定的性能。因此，他们应该在以后的时间段里被给予更多的资源来弥补这些缺失。一个简单的方案就是保留 QoS kernel 没有使用的限额，而不是丢弃它们。

我们开发了滚动分配算法来保留 QoS kernel 没有使用的限额。当下一个时间段分配限额的时候，QoS kernel 上一个时间段没有使用的限额会被加到这个时间段的限额里。Non-QoS kernel 的限额还是会被丢弃。用这种方法，QoS kernel 在这个时间段就有了更多的动态资源来弥补上一个时间段的性能损失。以图 3-3(c)所示为例， $K_0$  和  $K_1$  的计数器都在时间段 2 开始的时候进行调整。那个时候， $K_0$  还有没用完的限额，由于  $K_0$  是 QoS kernel，这些限额会被保留下来。由于  $K_1$  是 non-QoS kernel， $K_1$  没用完的限额就会被直接丢弃。

弹性分配算法使用了可变的时间段长度。其有效性取决于是否限额经常很快就消耗完。滚动分配算法使用定长的时间段。其有效性取决于滚动到下一个时间段的限额能否有效的提高 QoS kernel 的性能。我们的实验结果表明两种方法都比之前提出的简单分配算法能满足更多的 QoS 目标。

#### 3.4.5 管理 Non-QoS Kernel

non-QoS kernel 的限额不能像 QoS kernel 那样分配，这是因为 non-QoS kernel 没有 QoS 需求。不为 non-QoS kernel 分配限额，或是只分配很少的限额就会让共享退化成时分复用的执行方式。因此，我们需要找到一个 non-QoS kernel 的合适的限额，来在保证 QoS kernel 正常执行的同时最大化 non-QoS kernel 的吞吐率。

我们开发了一个简单的方法来寻找 non-QoS kernel 的合适的限额。这个搜索过程依赖于 QoS kernel 完成其性能目标的情况。如果 QoS kernel 之前时间段的 IPC( $IPC_{epoch}$ ) 比其目标高，那么 non-QoS kernel 就可以有更高的限额。否则，non-QoS kernel 的限额就会减少。每个时间段开始的时候 non-QoS kernel 的限额都会利用之前时间段的信息进行更新。non-QoS kernel 的人工给定的  $IPC_{goal}$  如下式所示：

$$IPC_{goal} = IPC_{epoch} \times \prod_{\forall k \in QoS \text{ kernels}} \frac{IPC_{epoch \text{ of } k}}{\alpha_k \times IPC_{goal \text{ of } k}}$$

一旦 non-QoS kernel 的  $IPC_{goal}$  经计算得出，其限额也就可以通过公式 (3-1) 计算得出。non-QoS kernel 的  $IPC_{epoch}$  开始设成一个非常小的值来保证初始设置不会影响 QoS kernel 的性能。QoS kernel 开始的时候可以达到其 QoS 目标，接着，为了提高利用率，non-QoS kernel 的  $IPC_{goal}$  就会提高，但是并不会高到影响 QoS kernel。在计算 non-QoS kernel 的限额时，基于历史的调整里所使用的参数  $\alpha_k$  也用在这里，以保证 QoS kernel 的性能。

值得一提的是，non-QoS kernel 的  $IPC_{goal}$  会随着运行动态变化。如之前的研究所示<sup>[74]</sup>，一个 kernel 在运行时的行为可能会发生变化。所以 non-QoS kernel 的同一个  $IPC_{goal}$  也可能对 QoS kernel 产生不同的影响。因此，我们的设计动态的根据 QoS kernel 的需求来限制 non-QoS kernel 的性能。如果 QoS kernel 需要的多，non-QoS kernel 的资源就会减少。反之，non-QoS kernel 的资源就会增加。

### 3.4.6 静态资源分配和调整

如寄存器和共享内存等静态资源是通过 TB 的发射来控制的。这一资源也被管理起来。这是因为如果没有足够的 TLP，QoS kernel 也难以达到其 QoS 目标。而增加 TB 数就可以增加 TLP。

**2 个或更多个 kernel 的对称式 TB 分配。**一开始，QoS kernel 的 TB 均匀分配到每个 SM 上，这样每个 SM 开始的线程数都一样，TLP 也都一样。对于 non-QoS kernel 来说，之前的研究表明<sup>[57, 58]</sup> 在一个 SM 里放置太多 kernel 并不是好事。因此，我们把 non-QoS kernel 均等的划分到不同的 SM 里。每个 kernel 都对称的向自己的 SM 里发射 TB。例如，假设有一个 QoS kernel 和 2 个 non-QoS kernel，GPU 上有个 16 个 SM。那么 QoS kernel 会在 16 个 SM 上运行，而 non-QoS kernel 则分别会运行在 8 个 SM 上。在每个 SM 内部，每个 kernel 的线程数都相同。这一分配是通过部分上下文切换<sup>[58]</sup> 来进行运行时静态资源调整的基础。

**运行时调整。**在运行时，我们的设计会监视 kernel 的执行情况来判断是否要改变 TB 的分配。在每个时间段，我们都采集所有 kernel 的空闲 warp 数 (Idle Warp, IW)。空闲 warp 是有指令，但由于流水线满不能执行的 warp。这表明一个 kernel 的 TLP 可能是太多了<sup>[74]</sup>。空闲 warp 占据静态资源，但不能为 kernel 的运行提供贡献。因此，即使用更低的 TLP 也能达到一样的性能。但是如果一个 kernel 的空闲 warp 太少，那么增加一些 TLP 有可能提升其性能。基于这一观察，我们设计了改变 TB 分配的算法 3-1。

---

**算法 3-1** TB 分配调整算法，每个时间段开始时执行

---

```

1: for all kernel  $k$  do
2:   if  $k$  是一个 QoS kernel，并且  $IPC_{history}$  of  $k < IPC_{goal}$  of  $k$ ，并且  $k$  闲置 TB 数
   小于每 SM 2 个 then
3:     for all  $SM_i$  do
4:       if 有足够的资源在  $SM_i$  上发射一个  $k$  的 TB then
5:         在  $SM_i$  上发射一个  $k$  的 TB
6:       else
7:         for all  $SM_i$  上除了  $k$  的 kernel，设为  $v$  do
8:            $n \leftarrow$  接受一个  $k$  的新 TB 所需换出的  $v$  的 TB 数
9:            $N_v \leftarrow v$  当前的 TB 数
10:          if  $v$  是一个 non-QoS kernel，或者  $v$  每 SM 有  $n + 1$  个空闲 TB，或
     $IPC_{history}$  of  $v - IPC_{history}$  of  $v \times \frac{n}{N_v} > IPC_{goal}$  of  $v$ . then
11:            在  $SM_i$  上换出  $v$  的  $n$  个 TB
12:          end if
13:        end for
14:      end if
15:    end for
16:  end if
17: end for

```

---

在一个时间段开始时，每个 kernel 在每个 SM 里的空闲 warp 的数量都被收集起来。如果空闲 warp 的数量等于一个 TB 里 warp 的数量，那么换出一个 TB 就相当于换出了这些空闲 warp。我们就把这些 TB 称为空闲 TB。如果一个 QoS kernel 的空闲 TB 数不超过 1，并且其 IPC 没有达到其 IPC 目标，那么我们就会多为其分配一个 TB 以增加 TLP。如果 TB 不能直接发射，那么受害者 kernel 的 TB 就会被换出。受害者 kernel 是满足下列其中一个条件的 kernel：

- 是一个 non-QoS kernel
- 如果需要换出  $n$  个 kernel，那么其至少有  $n + 1$  个空闲 TB
- kernel 的  $IPC_{history}$  很高，达到  $IPC_{history} \times (1 - n/N) > IPC_{goal}$ ，其中  $N$  是这个 kernel 的总 TB 数。

因此，non-QoS kernel，或一个有多余 TLP 或 IPC 很高的 kernel 会被选为受害者 kernel。最后，为了限制抢占开销，同一时间只能进行一个抢占操作。

## 3.5 实验验证

### 3.5.1 实验方法

表 3-2 模拟器参数

Table 3-2 Simulation parameters.

GPU 参数	值	SM 参数	值
核心频率	1216MHz	寄存器	256KB
内存频率	7GHz	共享内存	96KB
SM 数	16	线程数	2048
内存控制器	4	TB 数	32
调度策略	GTO	Warp 调度器	4

**模拟器.** 为了验证我们的设计，我们使用最新版本的 GPGPU-Sim<sup>[49]</sup>。模拟器的参数如表 3-2 所示。这些参数和之前工作<sup>[57, 58]</sup> 里所用的参数类似。我们根据之前的工作<sup>[33, 57, 59]</sup> 里所报告的假设和实现方法修改了 GPGPU-Sim 以支持空间划分和细粒度共享（如 SMK）。

**性能评测程序；可扩展性.** 我们使用了 10 个来自 Parboil 性能评测程序集<sup>[51]</sup> 里的程序。因为 *bfs* 太小，并不与其他 kernel 的性能产生影响，所以实验中我们没有使用这个程序。我们所有的 kernel 都使用最大的数据集。我们使用 2 个或 3 个 kernel 一起共享 GPU 来测试我们方法的可扩展性。在 2 个 kernel 的情况里，我们使用了  $10 \times 9 = 90$  对 kernel：其中一个是 QoS kernel，而另一个是 non-QoS kernel。在 3 个 kernel 的情况里，由于所有的组合太多，我们使用了其中 60 个组合。这些组合中有 1-2 个 QoS kernel，而剩下的则是 non-QoS kernel。

我们每组测试运行两百万个时钟周期。根据之前的研究<sup>[34]</sup>，只要模拟时间大于一百万个时钟周期，结果就是准确的。如果程序在两百万个周期前结束，程序会被重新执行。如果一个程序包含多个 kernel 或是一个 kernel 被执行了多次，我们会把指令和时钟周期加起来计算 IPC。时间段的长度是一万个时钟周期。这是一个和之前工作<sup>[77]</sup> 所使用的值相同的经验值。根据之前工作的结果，这个值是一个足够好的值。我们每个时间段采集 100 次空闲 warp 的情况，并根据其平均值来进行 TB 调整的决策。

**评测指标.** 为了验证满足 QoS 的能力，我们使用达到 QoS 目标的百分比 ( $QoS_{reach}$ ) 作为我们比较不同方法的指标。 $QoS_{reach}$  定义为： $\frac{\# \text{ of Success Cases}}{\# \text{ of Total Cases}}$  我们比较了如下方法：使用爬山算法的空间划分<sup>[59]</sup> (Spart)，简单分配算法 (Naïve)，弹性分配算法 (Elastic) 和滚动分配算法 (Rollover)。如第 3.4.2 所解释的，QoS 目标被设置为 kernel 单独运行时的  $IPC_{isolated}$  的百分比，从 50% 到 95%，步长为 5%。这样对于 2 个 kernel 和 3 个 kernel

的情况，90 对和 60 组分别对应 10 个 QoS 目标，也就分别产生了 900 个和 600 个测试的情况。对于有 2 个 QoS kernel 的情况，我们的 QoS 目标从 (25%, 25%) 到 (70%, 70%)，步长为 (5%, 5%)。在计算吞吐率的时候，只有满足了 QoS 目标的情况才会被计算在内。更高的  $QoS_{reach}$  代表一个设计能满足更多的 QoS 目标。这个指标和之前工作<sup>[59]</sup>里的指标类似，但是覆盖的范围更广，QoS 目标也更难达到。

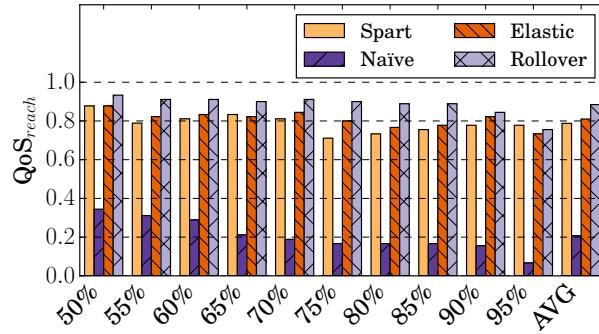
### 3.5.2 $QoS_{reach}$ 的比较

图 3-5 展示了不同 QoS 方法在运行 2 个 kernel (图 3-5(a)) 和 3 个 kernel(图 3-5(b) 和图 3-5(c)) 的情况下的  $QoS_{reach}$ 。在运行 2 个 kernel 的时候，*Naïve* 的  $QoS_{reach}$  最低，只有 20.6%。这是因为其不能有效的使用限额机制。而 *Rollover* 则有最好的结果 (88.4%)。*Spart* 满足了 900 个情况中的 78.8%。*Rollover* 在绝大多数情况下都比 *Spart* 好，平均提升了 12.2% 的  $QoS_{reach}$

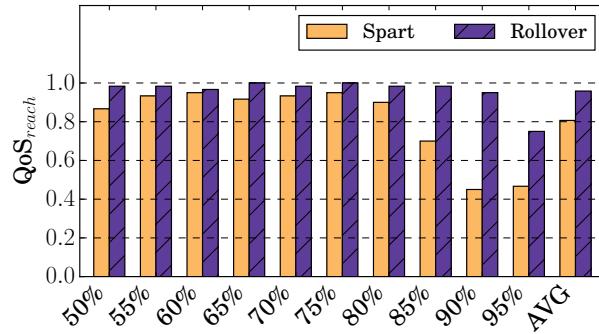
*Elastic* 和 *Rollover* 都比 *Naïve* 要好。这是由于它们改进了 *Naïve* 方法的主要缺陷。*Rollover* 比 *Elastic* 的结果好的原因是 *Rollover* 可以在 QoS kernel 没有达到性能目标的时候（也就是在时间段结束有剩余的限额）直接的改善它们的性能。弹性分配算法则是让 QoS kernel 在限额消耗快的时候性能变得更好。

图 3-5(b)和图 3-5(c)展示了在同时运行 3 个 kernel 时，分别有 1 个或 2 个 QoS kernel 时的  $QoS_{reach}$ 。我们在这里只比较了 *Spart* 和我们提出的最好的方法 *Rollover*。两张图都显示 *Rollover* 比 *Spart* 达到了更多的 QoS 目标。在 1 个和 2 个 QoS kernel 的情况下分别提高了 18.8% 和 43.8%。*Rollover* 在有更多的 kernel 和更多的 QoS kernel 时表现的都比 *Spart* 更好。在 QoS 目标变得更高的时候，*Spart* 不能充分利用 GPU 资源的缺点就展现了出来。例如，*Spart* 在超过 (60%, 60%) 的情况下表现的很差 (图 3-5(c))，而完全无法处理 (70%, 70%) 的 QoS 目标。细粒度 QoS 可以更好控制多种资源 (如计算时钟周期)，而粗粒度 QoS 只能控制 SM 的数量。因此，*Rollover* 比 *Spart* 的扩展性更好。

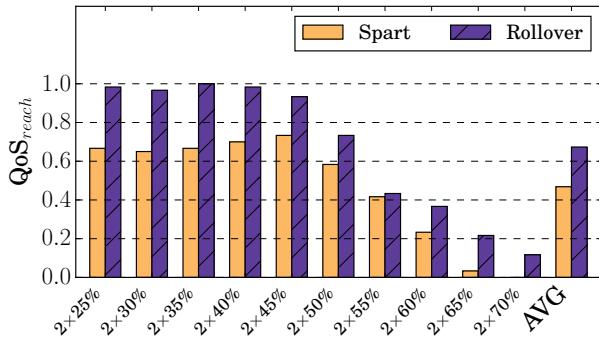
我们在图 Figure 3-6 中报告了 *Rollover* 和 *Spart* 根据 QoS kernel 分组的达到 QoS 目标的情况。每个柱状图取的是 90 个情况的平均值 (9 个配对和 10 个 QoS 目标)。我们还报告了配对的类型的汇总结果 (计算密集型 + 计算密集型 (“C+C”), 计算密集型 + 内存密集型 (“C+M”) 和内存密集型 + 内存密集型 (“M+M”))。我们发现对于 C+C 的 kernel 来说，*Spart* 都能 *Rollover* 满足所有的 QoS 目标。但是，对于 M+M 型的分组，由于 *Spart* 没有控制内存带宽的方法，其结果要比 *Rollover* 要差。尽管 *Rollover* 不直接控制内存带宽，其通过限额机制来限制指令的发射。这一方法减少了在限额用完时的内存通信，也就减少了 kernel 之间对于内存带宽的竞争。同样地，C+M 里 *Rollover*



(a) 两个 kernel 的情况



(b) 三个 kernel 里其中一个是 QoS kernel

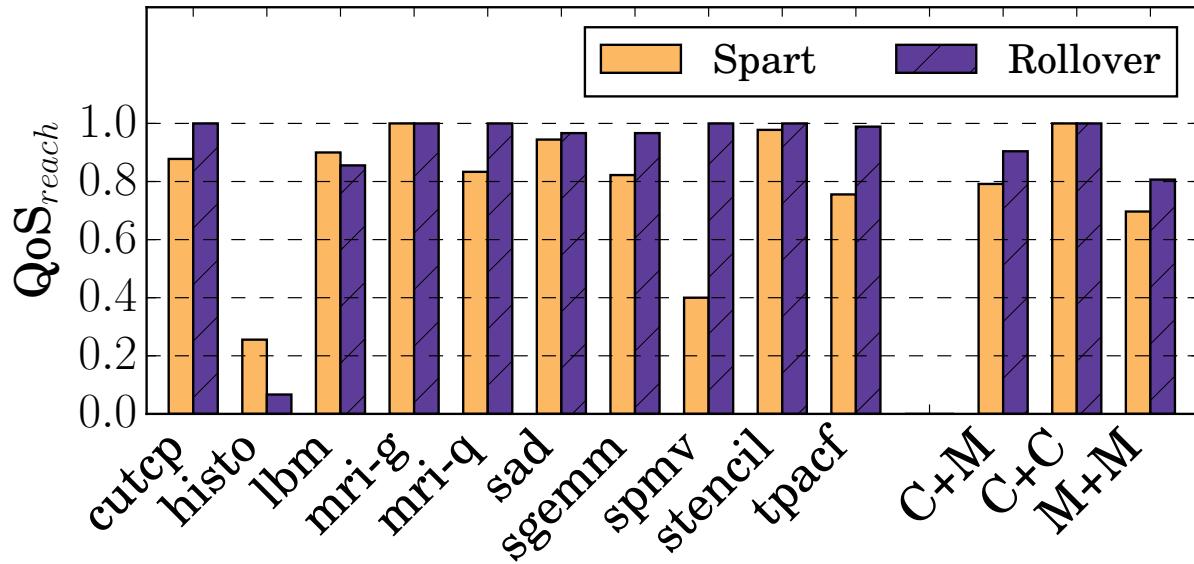


(c) 三个 kernel 里其中两个是 QoS kernel

图 3-5 满足 QoS 目标的比例,  $QoS_{reach}$ 。每个柱状图取 90 对或 60 组 (3 个 kernel) 的平均值  
 Figure 3-5  $QoS_{reach}$  vs. QoS goals. Each bar is averaged over all 90 pairs (or 60 trios) of benchmarks.

也比 Spart 的结果要好

Rollover 在 6 个性能评测程序中都满足了所有的 QoS 目标。对于 histo, 这两个方法的表现都不好, 这是由于这个程序里的 kernel 都很短, 而 Rollover 有在线分析的开销。对于这种情况, 一个 OS 级的 kernel 调度器可能表现的会更好。

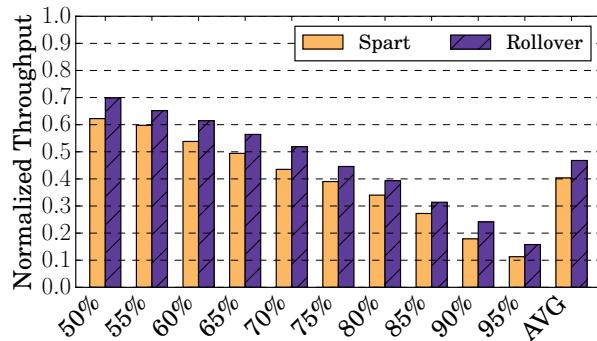
图 3-6 根据程序分组的满足 QoS 目标的比例,  $QoS_{reach}$ 。Figure 3-6  $QoS_{reach}$  vs. QoS kernel in two-kernel sharing.

### 3.5.3 Non-QoS Kernel 的吞吐率

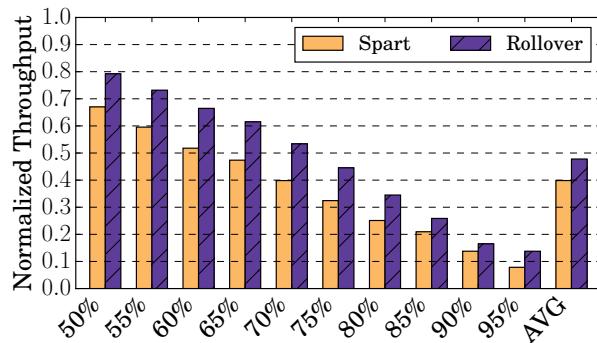
图 3-7展示了 non-QoS kernel 在 2 个 kernel 和 3 个 kernel 的情况下的归一化吞吐率。我们这里只包含了那些满足 QoS 目标的情况的结果。从结果中我们可以看到：

1. non-QoS kernel 的性能随着 QoS 目标的提高而降低
2. Rollover 在所有的情况下吞吐率都比 Spart 高。而这也随着 QoS 目标的提高而提高。在 2 个 kernel 的情况下，平均提升是 15.9%。而在 3 个 kernel 的情况下则提高到了 19.9% 和 20.5%。
3. Rollover 相对于 Spart 的提升也随着 QoS 目标的提高而提高。例如，图 3-7(b)中最高的提升来自于 95% 组里 (75.5%)。而在图 3-7(c)中，最后三组的提升都超过了 10 倍。

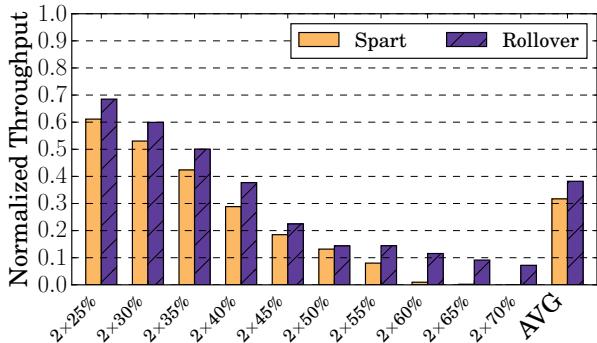
这一结果表明细粒度资源分配比粗粒度资源分配效果更好。Spart 在 QoS kernel 需要的资源粒度小于一个 SM 的时候无法将多余的资源分配到 non-QoS kernel 上。也就是说，在粗粒度方法中，SM 的资源不能分割。而 Rollover 则可以把资源精细的在 QoS kernel 和 non-QoS kernel 之间进行分配。我们预计随着 kernel 数的增多，Rollover 相对于 Spart 的提升会更为显著。



(a) 2个 kernel 的情况



(b) 3个 kernel 里有 1 个 QoS kernel 的情况



(c) 3个 kernel 里有 2 个 QoS kernel 的情况

图 3-7 non-QoS kernel 的归一化吞吐率。X 轴是 QoS 目标

Figure 3-7 Throughput normalized to isolated execution of non-QoS kernels. The x-axis is QoS goals.

### 3.5.4 QoS Kernel 的吞吐率

对于 QoS kernel 来说，其目标是达到 QoS 目标，但不超过太多。这样 GPU 可以就把剩下的资源分配 non-QoS kernel 以提高其吞吐率。图 3-8展示了 QoS kernel 的归一化到其 QoS 目标的吞吐率。如图所示，Spart 平均超过了 QoS 目标 11.6%，这就降低了

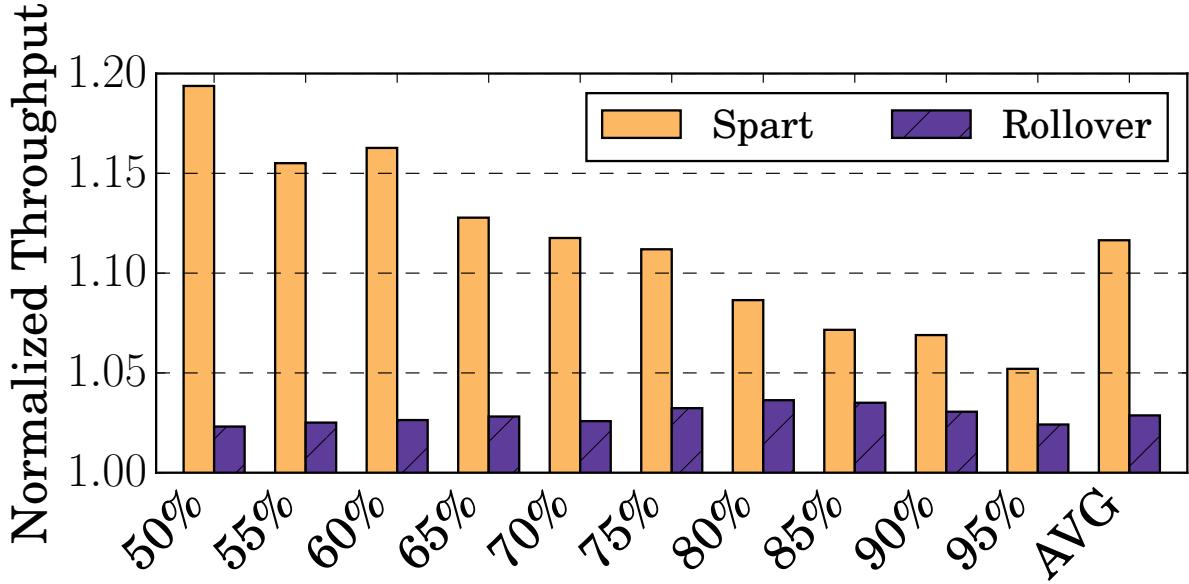


图 3-8 QoS kernel 的归一化吞吐率。其归一化到 QoS 目标。

Figure 3-8 Actual throughput of QoS kernels, normalized to their QoS goals.

non-QoS kernel 的性能。而 Rollover 只超过了 2.8%。这一结果来自于细粒度 QoS 中对资源的精细控制，例如其可以以时钟周期为精度控制 kernel 的指令发射。相比之下，Spart 只能以 SM 为单位分配资源。这就导致了即使 QoS kernel 不需要一个 SM 的资源，Spart 依然无法做出分割。

### 3.5.5 基于优先级的 QoS

如第 3.4.3 节所讨论的，传统 CPU 中利用优先级来进行 QoS 控制的方法不适用于 GPU。图 3-9 和图 3-10 比较了 Rollover 和一个时分复用的 warp 调度策略 (Rollover-Time) 的  $QoS_{reach}$  和 non-QoS kernel 的吞吐率。这个时分复用的策略会先执行 QoS kernel，直到其达到了 QoS 目标，再执行 non-QoS kernel。如图所示，这两个方法的  $QoS_{reach}$  非常接近，平均差距只有 3%。这表明这两个方法达到 QoS 目标的能力基本相同。但是，Rollover-Time 大大降低了 non-QoS kernel 的性能，其降低了 1.47 倍。因此，允许 kernel 重叠的执行对 kernel 的性能提升有很大帮助。

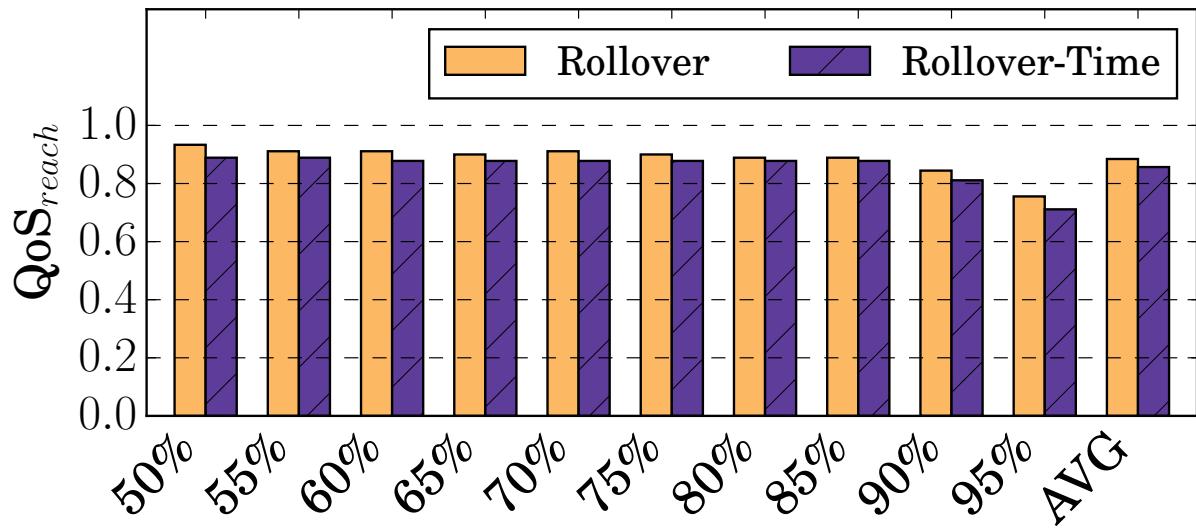


图 3-9 QoS kernel 的  $QoS_{reach}$ 。  
Figure 3-9  $QoS_{reach}$  of QoS kernels.

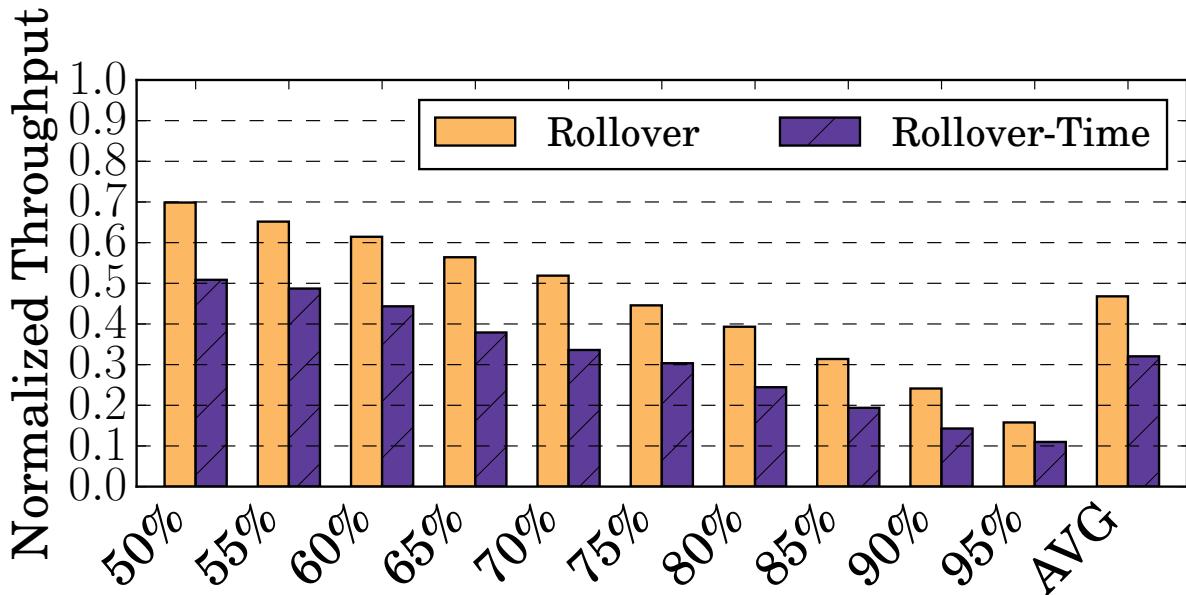


图 3-10 归一化吞吐率。  
Figure 3-10 Throughput, normalized to the isolated execution, for non-QoS kernels.

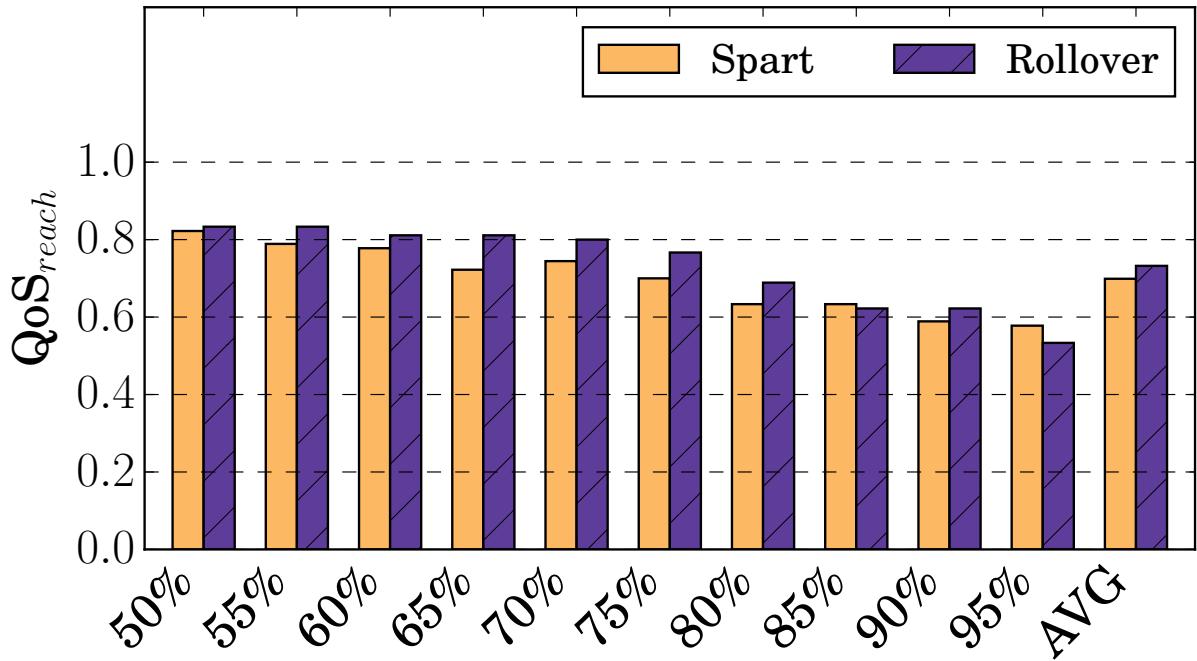


图 3-11 56 个 SM 的情况下满足 QoS 目标的比例,  $\text{QoS}_{\text{reach}}$ 。每个柱状图取 90 对或 60 组 (3 个 kernel) 的平均值。

Figure 3-11  $\text{QoS}_{\text{reach}}$  vs. QoS goals for 56 SMs. Each bar is averaged over all 90 pairs of benchmarks.

### 3.5.6 SM 数量的可扩展性

最近的 GPU 架构在一个 GPU 里集成了更多的 SM<sup>[78]</sup>。为了测试我们设计对于 SM 数量的可扩展性, 我们模拟了一个有 56 个 SM 的 GPU, 每个 SM 有 2 个 warp 调度器。其他的模拟参数仍与表 3-2 中的相同。由于这些参数的变化, 每个 kernel 单独运行时的性能也不同了, 这导致的 QoS 目标的变化。因此, 这一小节的结果不能直接与之前的结果进行比较。

图 3-11 和图 3-12 展示了 Spart 和 Rollover 的  $\text{QoS}_{\text{reach}}$  和 non-QoS kernel 的归一化吞吐率。如图所示, 更多的 SM 让 Spart 的  $\text{QoS}_{\text{reach}}$  有了一定的提升。这是因为更多的 SM 让 Spart 的资源分配粒度更细, 但是这一结果仍比 Rollover 低 4.76%。在吞吐率方面, Rollover 则远远超过了 Spart, 平均提升达到了 30.65%。这表明了 Rollover 在 SM 数量增加的时候依然能保证良好的  $\text{QoS}_{\text{reach}}$  和吞吐率。

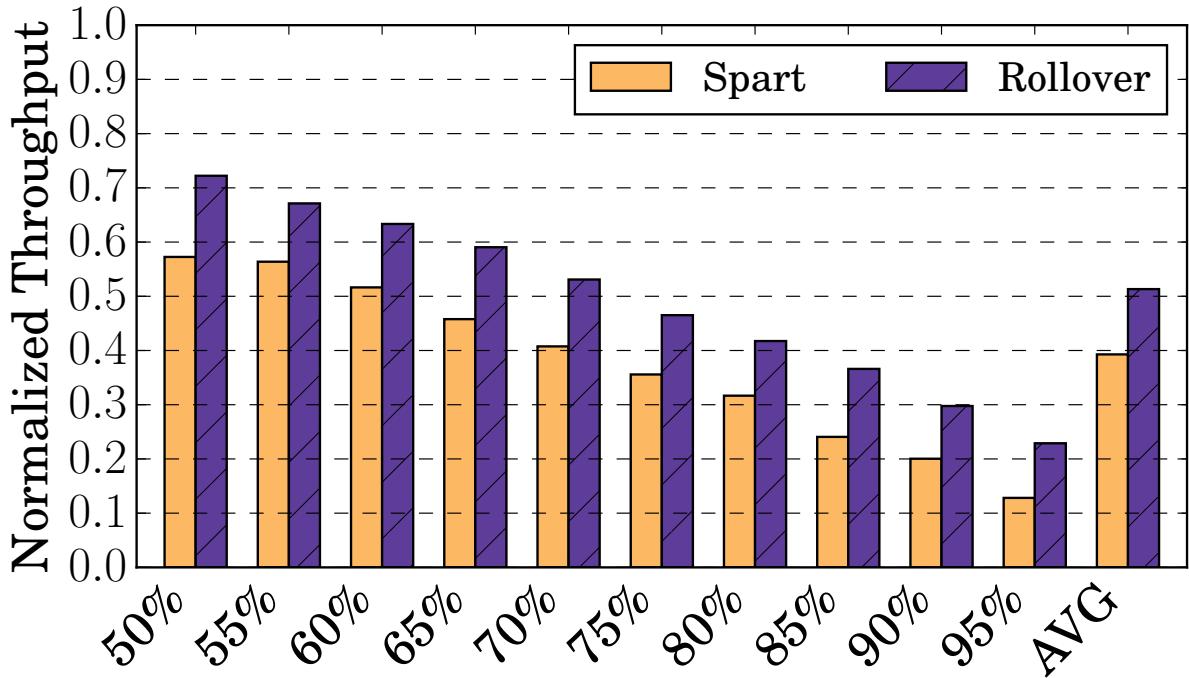


图 3-12 56 个 SM 的情况下 non-QoS kernel 的归一化吞吐率。X 轴是 QoS 目标。

Figure 3-12 Throughput normalized to isolated execution of non-QoS kernels for 56 SMs. The x-axis is QoS goals.

### 3.5.7 能效

最后，我们预计细粒度 QoS 管理机制会提高 GPU 的能效。我们使用 GPUWattch<sup>[79]</sup>，一个 GPGPU-Sim 里的 GPU 功耗模型来测试不同 QoS 方案的功耗。图 3-13展示了在 2 个 kernel 的情况下 Rollover 相比 Spart 每瓦特指令数的提升。如图所示，由于更好的资源利用率，Rollover 相比 Spart 平均提升了 9.3% 的能效。由于我们的设计在同样 SM 数量的情况下有更好的性能和能效。我们的设计有可能在更少的 SM 上达到与粗粒度 QoS 同样的性能。这样我们就提供了关掉一些 SM 来节能的机会。

### 3.5.8 抢占开销和其他结果

这里我们总结了我们测试的抢占开销，基于历史的限额调整的效果和静态资源管理的效果。总的来说，抢占对 non-QoS kernel 有 1.93% 的性能开销，这是由于大多数抢占导致的内存操作都被其他 TB 的执行掩盖了。在关掉基于历史的限额调整时， $QoS_{reach}$  对所有的情况都大幅下降。开启这一机制可以提升 86.4% 的  $QoS_{reach}$ 。静态资源管理也

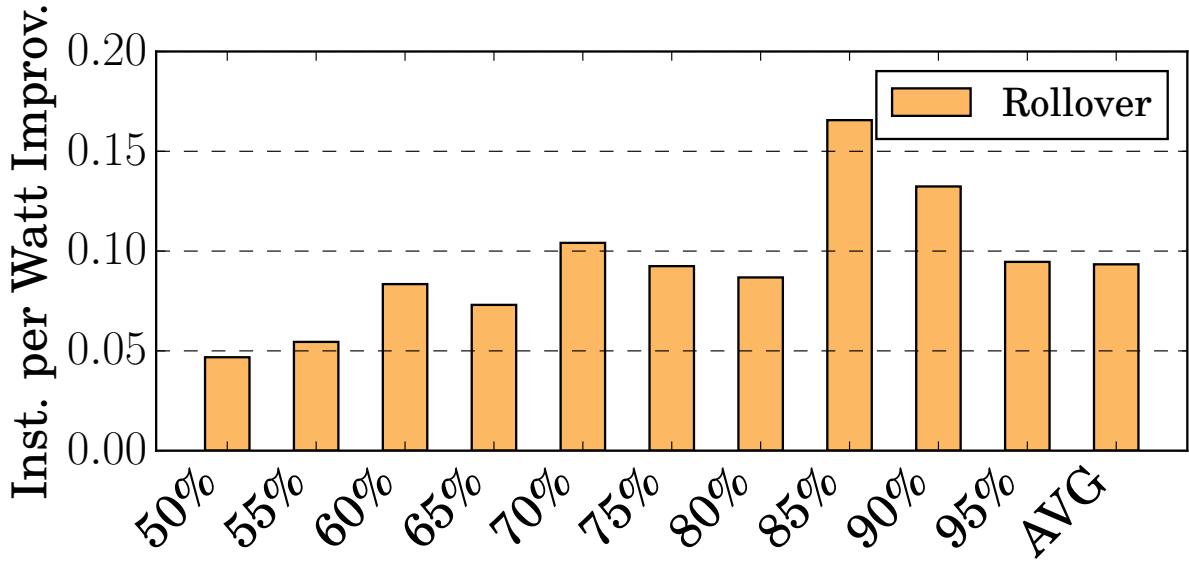


图 3-13 相比 Spart 的能效提升

Figure 3-13 Energy efficiency improvement over Spart.

对性能有正面的影响。由于更好的 TLP 分配，开启静态资源管理可以对 M+M 的组合提升 13.3% 的吞吐率。

### 3.5.9 硬件开销

我们的设计主要对如 SMK 的细粒度抢占机制<sup>[57, 58]</sup>增加了逻辑控制的部分。如图 3-2 所示，附加的逻辑有：(1) 收集每个 SM 上 kernel 数据，更新限额和决定 TB 分配的 QoS 管理器；(2) 具有限额功能的 warp 调度器。这部分逻辑不在关键路径上，因此其对性能没有影响。保存限额，时间段和记录指令的计数器已经在 SMK 设计里提供了。同时这一方案还需要一个标识 QoS kernel 的位向量。总的来说，我们认为这一 QoS 管理机制的硬件开销并不大。

## 3.6 本章小结

当前的 GPU 不支持 QoS 管理，而这对于现代系统中多个应用共享 GPU 是很重要的。我们发现 SM 级别的粗粒度的管理不能满足需求，并且对 kernel 数量的扩展性也不好。我们提出了一套新型设计来为多个 kernel 实现细粒度的 QoS 控制。这个机制让有 QoS 目标的 kernel 只接受刚好满足 QoS 需求的资源，而让其他没有 QoS 目标 kernel 利

用剩下的资源提高吞吐率。并且，我们还提出了一个把应用 QoS 需求转化为体系架构指标的方法。实验结果显示我们的方法在满足 QoS 目标和提高能效方面相比之前的粗粒度 QoS 管理方法都有显著的提升。这一工作发表在 ISCA 2017 上。

## 第四章 基于渐近分析的 CPU+GPU 协同调度算法

### 4.1 研究背景

多核处理器现在依旧是通用处理器市场的主流，但类似 GPU 等众核处理器也在高性能计算和数据集中逐渐流行起来。如果算法适合在 GPU 架构上运行，GPU 可以提供很高的计算性能。因此，配有 CPU 和 GPU 的异构系统逐渐成为了系统设计中的趋势。尽管这样的异构系统已经被广泛使用，但由于程序员手动在 CPU 和 GPU 之间分配负载十分困难，程序员很难高效的利用这样的系统。

数据并行机制假设 CPU 和 GPU 一起处理一个任务。这个任务的数据可以在不同的计算设备上并行处理。调度器会将数据（工作负载）分配到不同的计算设备上以充分利用系统的计算性能。Lee 等人的研究<sup>[80]</sup>表明，即使相比于使用类似 CUDA<sup>[19]</sup>、Brook+<sup>[20]</sup> 和 OpenCL<sup>[21]</sup> 之类的底层语言进行优化的 GPU 程序，经过优化的 CPU 程序依然能提供可观的性能。因此，同时使用 CPU 和 GPU 可以提高系统的整体性能。

负载均衡在数据并行调度中十分重要。之前的研究提出了许多静态或动态的调度算法来平衡 CPU 和 GPU 之间的负载。在静态算法中，CPU 和 GPU 之间的负载是静态分配的。由于在不知道运行时细节的情况下预测一个 GPU 程序的性能十分困难，静态工作负载分配算法经常会导致负载不均衡。GPU 的性能与工作负载的大小有关，而这个信息调度器是无法在运行之前得知的。

另一方面，在动态算法中，工作负载的划分基于 CPU 和 GPU 在运行时的性能进行调整。一种方法是调度器先用一小部分工作负载来分析 CPU 和 GPU 的性能，然后调度器根据分析得到的数据来划分剩下的工作负载。另一种方法是把工作负载切成很多个数据块，并用频繁的同步来一步步的平衡工作负载。前一种方法由于分析的不准确，会导致负载不均衡，而后一种方法则引入了很高的性能开销，降低了整体性能。综上，负载均衡是异构系统性能的重要影响因素，而现有的算法不能很好的解决负载均衡问题。

为了解决这一问题，本章提出了基于渐近分析的 CPU+GPU 协同调度算法（Co-Scheduling Based on Asymptotic Profiling, CAP）。这一算法可以动态的在 CPU 和 GPU 之间平衡工作负载。CAP 只需要很少的几次同步就可以精确的预测出 GPU 和 CPU 之间的性能比。同时，这一操作由运行时系统所管理，无需程序员的干预。CAP 结合了两种动态算法的优点。这一算法针对数据并行设计，但也可以扩展到任务并行。

我们基于 CUDA 和 POSIX 线程（pthread）实现了我们的调度器来测试我们的算法。实验结果表明相比之前最好的协同调度算法，CAP 最高可以达到 42.7% 的性能提升。

本章提出的基于渐近分析的 CPU+GPU 协同调度算法具有如下主要贡献：

- 我们分析了 GPU 的性能特征和现有协同调度算法的优缺点。
- 我们提出了基于渐近分析的动态调度算法。其只需几次同步就可以在 CPU 和 GPU 之间平衡的分配负载。
- 实验结果表明相比之前最好的协同调度算法，CAP 最高可以达到 42.7% 的性能提升。

本章组织如下。第4.2节介绍了 CAP 的相关工作。第4.3节分析了 GPU 的性能特征和现有协同调度算法的优缺点。第4.4节介绍了 CAP 的设计，并和现有算法进行了对比。第4.5节描述了 CAP 的实现细节。第4.6节展示了实验环境，并分析了实验结果。第4.7节总结了本章的成果。

## 4.2 相关工作

GPU 编程正在成为并行编程领域的重要问题。人们提出了例如 CUDA<sup>[19]</sup>, Brooks+<sup>[20]</sup>, 和 OpenCL<sup>[21]</sup> 等编程框架来在高级语言的基础上充分利用硬件的资源。但是由于这些编程框架与硬件架构紧密相关，用它们写出的程序的性能在不同的硬件架构上往往不一致。同时，使用这些编程框架也需要程序员对硬件架构十分的了解。不然，写出的程序就无法充分利用 CPU 和 GPU 的资源。

GPU 的性能特性已经有了深入的研究。Ryoo<sup>[81]</sup> 等人研究了 GPU 程序的各种优化技术，例如区块化，数据预取，数据缓存等技术。他们的研究表明一个充分优化过的 GPU 程序会比一个没优化过的程序快得多。Zhang 等人<sup>[82]</sup> 和 Hong 等人<sup>[83]</sup> 开发了一个性能模型，并通过分析 Nvidia 提供的 GPU 程序编译器 NVCC 所编译出的 GPU 汇编指令来静态的分析 GPU 的性能。Baghsorkhi 等人<sup>[84]</sup> 开发了一个工具来自动化这个过程。Hong 等人<sup>[85]</sup> 之后扩展了他们提出的模型，把功耗分析加入到了他们的模型之中，从而得到了更详细的分析结果。这些模型可以用于静态分析 GPU 程序的性能，但是他们的计算开销太大，不适合在运行时使用。

调度算法在共享内存系统已得到了广泛的研究。OpenMP<sup>[16]</sup> 为 C/C++ 和 Fortran 语言提供了一个扩展，以简单的把串行程序扩展为并行程序并进行调度。WATS<sup>[86]</sup> 是一个可以感知任务负载的调度算法，其可以提高非对称多核心 CPU 架构上并行程序的性能。在分布式计算领域中，MapReduce<sup>[87]</sup> 提供了一个简单而有效的编程模型。使用这个编程模型，那些任务间没有数据依赖的并行程序可以很容易的被转化成分布式程序。许多调度算法上的研究都在试图解决 MapReduce 云计算系统中的异构调度问题。CellMR<sup>[88]</sup> 为非对称异构的 Cell 处理器集群设计了一个可以进行异构调度的 MapReduce 框架。MOON<sup>[89]</sup> 扩展了 Hadoop<sup>[90]</sup> 系统，使其可以应用于高度异构的网格计算环境。

随着 GPU 在数据中心中的使用越来越广泛，CPU+GPU 的协同调度也逐渐成为了

一个重要问题。一些平台和编程框架被设计和实现以结合 CPU 和 GPU 的计算能力。**Mars**<sup>[29]</sup> 使用 MapReduce 作为编程模型来管理 CPU 和 GPU。**StarPU**<sup>[91]</sup> 和 **Scout**<sup>[92]</sup> 提出了为这两个平台编程所使用的新编程语言，程序员需要重写他们的代码才能让程序在这两个平台上的运行。**Mars** 和 **Qilin**<sup>[93]</sup> 则需要程序员使用特定的 API 把任务提交给系统来进行管理。**OmpSS**<sup>[94]</sup> 扩展了 OpenMP 的语义，使 OpenMP 程序只需要很少的改动就可以同时利用 CPU 和 GPU。

异构系统中的负载均衡问题也有很多研究。**Bajaj** 等人<sup>[95]</sup> 利用有向无环图 (DAG) 分析任务并行程序的依赖关系来进行调度。**Radulescu** 等人<sup>[96]</sup> 在编译时静态的对任务进行排序来调度。**Jimenez** 等人<sup>[97]</sup> 在运行时利用任务的历史信息来进行协同调度。这几种方法针对的是任务并行类程序。**Brewe**<sup>[98]</sup> 提出了一种在编译时利用性能模型和历史性能信息，使用 SVM 机器学习算法对任务进行分类，从而切分任务负载的调度方法。这种调度方法受限于 SVM 分类器的预设分类，不一定能够找到最佳的划分。

**Scogland** 等人<sup>[99]</sup> 基于加速 OpenMP<sup>[100]</sup> 提出了两种在 CPU 和 GPU 间动态调度数据并行任务的方法。这两种方法都在运行时动态的采集和分析程序分别在 CPU 上和 GPU 上运行的性能。我们在第4.3节里介绍和讨论了这两种方法。他们的方法的缺点在其论文<sup>[99]</sup> 中的 K-Means 和 Helmholtz 两个应用的测试结果中有体现。

### 4.3 研究动机

这一节介绍了 GPU 相关的背景知识并分析了其性能特性。同时，这一节还介绍和分析了当前三种 CPU+GPU 协同调度的算法<sup>[99]</sup>。

GPU 是一种在性能特征和编程接口上都与 CPU 截然不同的众核架构处理器。CPU 通过操作系统中的硬件驱动与 GPU 进行通信和同步。但这一类操作有很高的开销，在程序运行时应尽量避免。

GPU 具有强大的并行处理能力。如果一个算法在线程间不需要通信或只需要很少的通信，或是这些通信具有很好的空间局部性，那么相比于 CPU，GPU 可以提供显著的加速比。但是 GPU 不适合执行有很多分支或通信的程序。有些未经优化的算法运行在 GPU 上甚至可能比运行在 CPU 上慢。

GPU 的性能与工作负载的大小有关。GPU 可以无开销的进行线程间切换来掩盖内存访问的延迟。由于内存访问延迟较长，GPU 需要大量的线程才能完全掩盖内存访问的延迟。而通常 GPU 上运行的线程数和 GPU 所收到的工作负载的大小有关。如果工作负载太小，GPU 就不能达到最佳的性能。

图 4-1展示了我们实验验证部分中使用的性能评测程序在不同工作负载大小下的归一化性能曲线（归一化到 CPU 的性能）。实验环境在第 4.6 节中有具体描述。X 轴表示

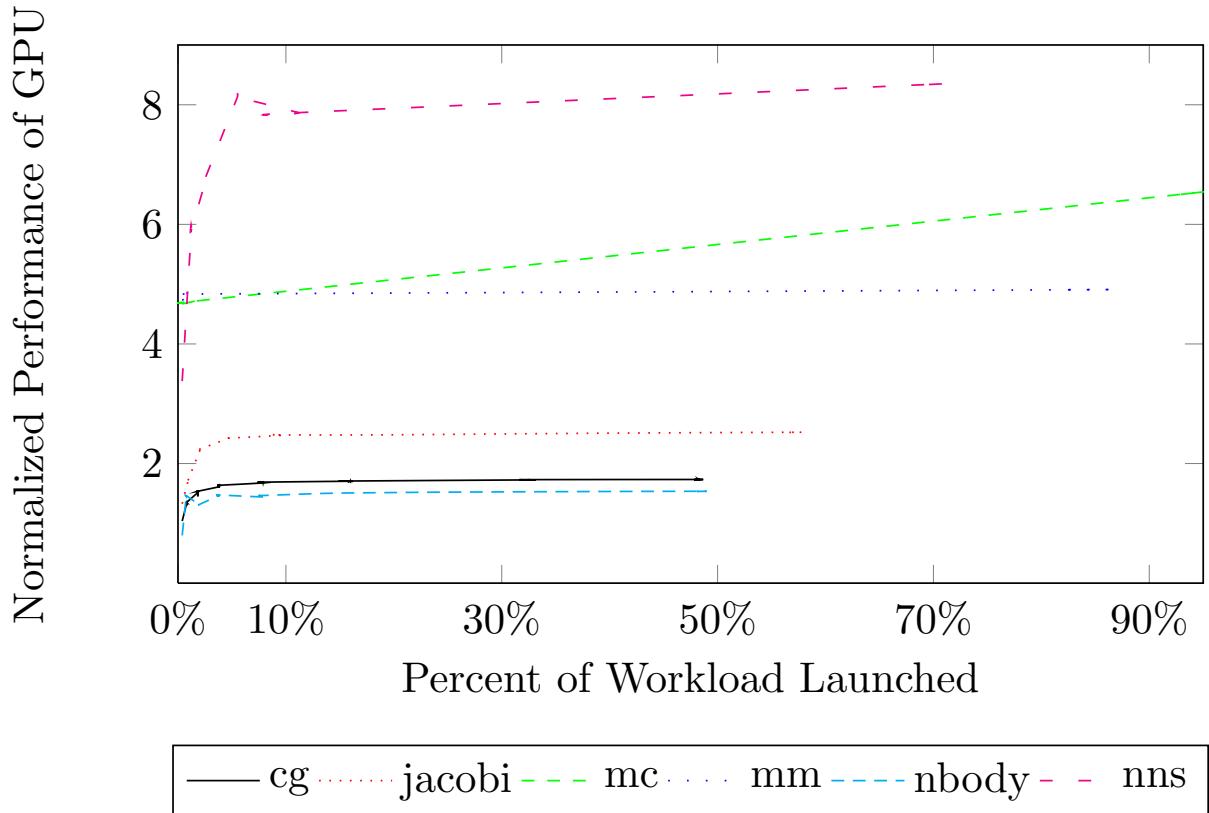


图 4-1 不同性能评测程序在不同工作负载大小下的性能曲线

Figure 4-1 Performance curve of different benchmarks with different sizes of workload

了发送到 GPU 上的工作负载的百分比，而 Y 轴则是对应工作负载大小的 GPU 归一化性能。

GPU 的性能在曲线开始部分快速上升，而在曲线末尾部分保持稳定。当工作负载大小比较小的时候，CPU 和 GPU 之间的数据传输和同步开销占据了绝大部分执行时间，同时 GPU 不能充分发挥其并行计算能力，因此性能较低。在工作负载较大的时候，GPU 的性能可以得到充分利用，程序性能也较高。因此，一次性把工作负载发送到 GPU 上比把工作负载切成许多小的数据块再一个个发送的性能要好。

综上，CPU 和 GPU 之间的同步和数据传输会影响性能。同时 GPU 的性能也受其接收到的工作负载的大小有关。我们设计的调度算法将这两点考虑在内来充分利用 CPU 和 GPU 的计算能力。

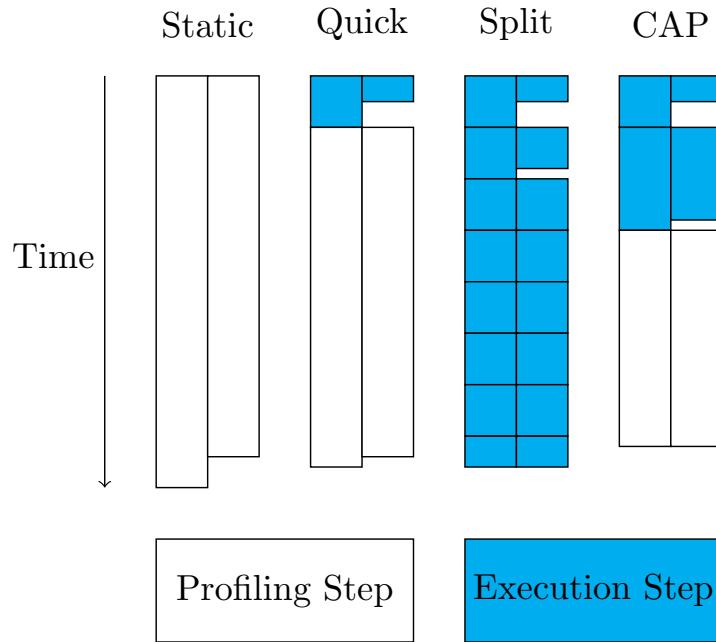


图 4-2 调度算法总览

Figure 4-2 Overview of CPU+GPU scheduling algorithms

### 4.3.1 当前调度算法介绍与分析

这一小节讨论了三种当前的 CPU+GPU 协同调度的算法。图 4-2 展示了这三种算法：静态调度（Static）、快速调度（Quick）和分割调度（Split）。

#### 4.3.1.1 静态调度算法

静态调度是一种传统的调度算法。这一算法在程序执行之前静态的在 CPU 和 GPU 之间划分工作负载。这一划分可以由调度器或程序员指定。

这个算法不能很好的在异构系统上工作。GPU 的性能随着算法，工作负载的大小和具体的实现的不同而不同。程序员很难手动的指定划分方案。调度器可以把之前的执行时间记录在硬盘上，并离线的计算 CPU 和 GPU 的性能比<sup>[93]</sup>，但这一方法不适用于第一次执行的程序。调度器也可以分析 GPU 编译器产生的代码来针对某个特定程序在某块特定的 GPU 上的性能。如果程序执行在另一块 GPU 上，这两种方法就都不能使用了。因为更换了硬件环境之后，之前记录的运行时间和离线分析都变得不准确了。

这一算法不需要运行时额外的同步，运行时开销也非常小。但是负载均衡很差，影响了整体性能。

### 4.3.1.2 快速调度算法

快速调度算法<sup>[99]</sup>通过分两步执行程序来回避静态调度算法的缺点。在第一步，这个算法先用静态划分执行一小部分工作负载。然后采集 CPU 和 GPU 上的运行时间来计算性能比。第二步，这个算法根据第一步算出的性能比来划分剩下的工作负载。

这一算法试图尽可能多的把工作负载一次性的发给 GPU，同时尽量保证负载均衡。算法根据运行时采集的性能信息来计算性能比，但这一信息未必准确。GPU 的性能随着工作负载的大小而变化。增加第一步使用的工作负载的大小可以提高采集的性能的准确性。但第一步的负载划分通常并不均衡，使用过大的工作负载可能影响整体性能。同时，第一步所需要的工作负载的大小也随着具体程序的不同而不同。

这一算法运行时开销很小。相比于静态调度算法，这一算法能更好的划分工作负载，因此整体性能也较好。但是快速调度算法的性能对第一步所使用的工作负载的大小十分敏感，而在程序运行之前无法得知第一步该使用多大的工作负载。

### 4.3.1.3 分割调度算法

分割调度算法<sup>[99]</sup>把工作负载分成多个等大小的小块，并分多步执行。这一算法根据上一步的性能信息来划分下一步的工作负载。

这一算法试图获得 CPU 和 GPU 之间准确的性能比。这一算法在这节讨论的三个算法里有最好的负载均衡。它每一步都采集 CPU 和 GPU 的执行时间，因此相比其他算法，其在运行时采集了更多的信息。但这一方法需要在 CPU 和 GPU 之间经常同步，因此引入了很高的运行时开销。同时 GPU 的性能也没有得到充分发挥。分割调度算法的性能对切分的大小十分敏感。如果切的太细，运行时开销会很高，GPU 的性能也得不到发挥。如果切的太粗，负载均衡就无法保证。

综上，当前的 CPU+GPU 协同调度算法并不能很好的解决工作负载划分的问题。我们提出了一个新的调度算法：基于渐进分析的协同调度算法（Co-Scheduling Based on Asymptotic Profiling, CAP）来解决这一问题。

## 4.4 基于渐近分析的协同调度算法

这一节介绍了 CAP 的设计，并与上一节介绍的算法进行了对比。

### 4.4.1 CAP 的设计

图 4-2展示了 CAP 的设计概览。CAP 把整个执行过程分成不定的多个步骤。在第一步，CAP 用静态划分执行一小部分的工作负载，并采集执行时间。这一步和快速调度

算法是一致的。不同的是，CAP 在第二步将第一步的工作负载加倍，然后用第一步采集到的数据所算出的性能比来划分这一加倍后的工作负载。利用这一方法，CAP 分析了 GPU 在不同工作负载下的性能。在接下来的步骤里，CAP 一直将工作负载在每一步加倍，直到 GPU 的性能达到一个稳定点。为了找到这个稳定点，CAP 比较了这一步和上一步所计算得到的工作负载划分的方差。如果这个方差小于一个阈值，那么就表明找到了这个稳定点。这个时候 CAP 就会一次性的把剩下的工作负载按照这一稳定的性能比进行划分，并发送到 CPU 和 GPU 上。如果剩下的工作负载不到这一步工作负载的 4 倍，CAP 会按照这一步所计算得到的性能比划分剩下的工作负载。算法 4-2 展示了这一算法的伪代码。

---

#### 算法 4-2 基于渐进分析的协同调度算法

---

```

1: if 这是第一步 then
2:   取一小部分工作负载，按照静态划分方法进行划分
3:   记录下这一工作负载的大小为  $s$ 
4: else if 还有工作负载 then
5:   根据上一步的结果计算性能比
6:   根据性能比划分这一步的工作负载的划分
7:   计算这一步的划分和上一步的划分的方差
8:   if 如果方差足够小或  $s * 4$  大于剩下的工作负载 then
9:     划分剩下的工作负载并执行
10:  else
11:     $s \leftarrow s * 2$ 
12:    划分工作负载  $s$  并执行
13:  end if
14: end if
```

---

下面我们用一个例子来说明 CAP 的工作方式。假设 CAP 要执行一个有 65536 个可在 CPU+GPU 异构系统中并行执行的迭代的程序。CAP 首先取  $\frac{1}{128}$ （这个值在执行前静态的指定）的迭代，即 512。然后 CAP 用静态划分算法划分这些迭代。这里我们假设是 1:1 划分。那么 CAP 就把 256 个迭代分给了 CPU，另 256 个迭代分给了 GPU。CAP 运行时系统会自动只把需要传输的数据传输到 GPU 上。在 CPU 和 GPU 完成了其工作之后，CAP 会同步 CPU 和 GPU，并把数据传输回 CPU 端，并采集 CPU 和 GPU 上的运行时间。然后 CAP 计算 CPU 和 GPU 每单位时间（例如秒）完成的迭代数。

在第二步，CAP 比较根据上一步采集的性能所计算得到的划分和上一步的划分的

方差。如果这个方差小于一个阈值，那么就证明 GPU 已经达到了性能曲线的稳定点，**CAP** 做出的划分是准确的。那么 **CAP** 就把剩下的工作负载（65024 个迭代）根据当前的划分进行分配。如果不是，那么 **CAP** 就会根据当前的划分分配 1024 ( $2 \times 512$ ) 个迭代。

**CAP** 一直重复这个过程，直到下一步会取超过剩余工作负载的  $\frac{1}{2}$ ，即  $s * 4$  大于剩下的工作负载。如果下一步取的工作负载超过了剩余工作负载的  $\frac{1}{2}$ ，那么 **CAP** 就无法在不影响 GPU 性能的情况下采集到稳定点的性能数据。因此，**CAP** 会直接按照当前划分划分剩下的工作负载。这一设计是为了尽可能的提高 GPU 的性能。

**CAP** 使用渐进的方式来采集 GPU 的性能信息。**CAP** 试图通过为 GPU 分配不同大小的工作负载来找到 GPU 性能曲线的稳定点。快速调度算法假设 GPU 的性能不随工作负载大小变化，但实际情况不是这样。**CAP** 一直采集性能数据，直到 GPU 的性能达到了稳定点。**CAP** 在每一步通过调整划分来接近最优划分。在找到稳定点之后，**CAP** 就根据当前的最优划分来分配剩下的工作负载。

#### 4.4.2 与其他调度算法的比较

我们从三个方面来比较调度算法：初始划分、划分方法和工作负载选择。

所有的调度方法第一步都使用静态划分。由于在第一步时调度器不知道 CPU 和 GPU 的性能信息，这是一个很自然的选择。快速调度，分割调度和 **CAP** 都在这一步使用很小的工作负载来避免由于负载不均衡导致的性能损失。

静态调度方法只使用静态划分。快速调度，分割调度和 **CAP** 则使用上一步所采集的性能信息和计算出的性能比来做划分。静态调度和快速调度算法都假设 GPU 的性能不随工作负载的大小变化，而分割调度和 **CAP** 则假设 GPU 的性能会变化。分割调度和 **CAP** 都多次采集执行时间来计算性能比并进行划分。

静态调度算法只执行一步。快速调度算法则把工作负载分成一小块和一大块。其根据执行一小部分工作负载所采集的性能信息来划分剩下的工作负载。分割调度算法把负载分成均等的多份。**CAP** 则使用一个动态策略来决定需要运行多少步。**CAP** 通过计算划分的方差来检查 GPU 的性能是否到达了稳定点。如果 GPU 的性能已经稳定了下来，**CAP** 就不会继续采集性能信息。

图 4-1 表明快速调度算法的不准确主要来自于曲线开始的快速变化。快速调度算法根据曲线开始的部分而不是曲线稳定的部分来估计 GPU 的性能。根据这样采集到的性能执行剩下的工作负载会造成负载不均衡。**CAP** 通过指数性的增加每一步的工作负载来快速的找到曲线的稳定点，并根据每一步的性能信息来调整工作负载划分。一旦划分稳定下来，**CAP** 就可以安全的执行剩下的工作负载，并达到良好的负载均衡。

由于每一步的工作负载大小都以指数级增加，**CAP** 只需要几次同步就可以完成。分

表 4-1 实验环境

Table 4-1 Evaluation environment

名字	描述
CPU	Intel Xeon E5620 @ 2.4GHz
GPU	Nvidia Tesla M2090 @ 1.3GHz
CPU 编译器	GCC 4.6.3
GPU 编译器	NVCC 5.0
操作系统	Debian Wheezy (Linux 3.2)

割调度算法所需的同步次数与最小的工作负载大小线性相关，而 CAP 的同步次数与最小的工作负载的大小对数相关。CAP 也不会影响 GPU 的性能，因为 CAP 每次都在增加 GPU 收到的工作负载的大小。CAP 只在 GPU 的性能进入稳定状态后才使用固定的划分。

## 4.5 具体实现

我们用 `pthread` 和 `CUDA` 实现了一个库作为我们的原型系统。这个库可以结合 CPU 和 GPU 的代码。我们的实现中分别实现了性能评测程序的 CPU 和 GPU 版本。

CPU 可以使用异步操作或额外的线程来管理 GPU。异步操作相比之下更加的复杂。因此为了简化实现，我们采用额外的线程来管理 GPU。首先我们用 `pthread` 为每个 GPU 创建一个新线程，然后由这些线程来管理 GPU。在进入程序之前我们首先初始化 GPU。这是由于我们所使用的 GPU (Nvidia Tesla M2090) 的初始化时间比较长。为了测试的准确，我们把这一部分排除在外。

调度部分的代码是一段关键区域，只有一个线程可以执行这段代码，而其他的线程则等待其完成调度。在我们的实现中，我们使用程序的主线程作为调度线程来执行这段代码。调度算法使用第 4.4 节中描述的算法，调度器负责把工作负载分发到每个线程来开始工作。

如果 GPU 线程在进行部分计算时只需要部分数据，调度器只会传输其需要的数据。我们使用这个技术来减少 CPU 和 GPU 之间的数据传输开销。这部分开销也被计入运行时间。否则调度器会在第一步开始之前把所有的数据传输到 GPU 上。在所有的工作做完之后，线程会在退出点同步并退出。

表 4-2 实验中的性能评测程序

Table 4-2 Benchmarks in the evaluation

Name	Description	Size
cg	共轭梯度方法	16Kx16K 矩阵
jacobi	雅可比法	16Kx16K 矩阵
mc	蒙特卡洛欧洲期权定价	64M 迭代
mm	矩阵乘法	两个 1Kx1K 矩阵
nbody	N-Body 模拟	16K 个物体
nns	最近邻居搜索	16K 点和 16K 次查询

## 4.6 实验验证

这一节用实验测试和验证了我们的算法。实验环境如表 4-1 所示。我们使用了 6 个性能评测程序来测试我们的算法，其列在表 4-2 中。

共轭梯度方法、雅可比法和矩阵乘法是经典的矩阵算法。N-Body 模拟是经典的物理模拟算法。蒙特卡罗方法和最近邻居搜索则被广泛应用于机器学习领域中。

所有的性能评测程序都实现了三个版本：CPU 版本（单线程），GPU 版本（使用 CUDA）和异构混合版本。异构混合版本包括了三种调度算法：快速调度算法、分割调度算法和 CAP。快速调度算法和 CAP 的初始工作负载大小为工作负载的  $\frac{1}{128}$ 。分割调度算法每一块的工作负载大小为 1024。我们实现的程序里应用了 GPU 里常用的优化技术（分块、共享内存归约等）。我们的性能评测程序里 CPU 和 GPU 可以提升相当的性能。其性能比例和完全优化过的版本相近<sup>[80]</sup>。

我们测试了混合执行部分的执行时间，包括 GPU 程序启动开销，数据传输时间，调度开销和计算时间。初始化和数据准备时间没有计算在内。我们使用相对时间（加速比）而非绝对时间（秒）来报告我们的结果。加速比为相对于 CPU 版本的加速比。我们还在实验中探索了初始工作负载大小对 CAP 的影响。

我们比较了 CAP 的性能，快速调度算法的性能和分割调度算法的性能。图 4-3 和表 4-3 中的结果表明，CAP 在 cg, jacobi, nbody 和 nns 评测程序中都比快速调度快。相比分割调度，CAP 则在 mc, mm 和 nns 评测程序中比较快。CAP 相比快速调度平均达到了 33.4% 的性能提升，相比分割调度平均达到了 42.7% 的性能提升。

我们通过如下公式计算执行时间的差距：

$$\frac{|time_{CPU} - time_{GPU}|}{\max\{time_{CPU}, time_{GPU}\}}$$

CAP 的良好性能来自于负载均衡。如图 4-4 所示，在快速调度算法中，除了 mm 程

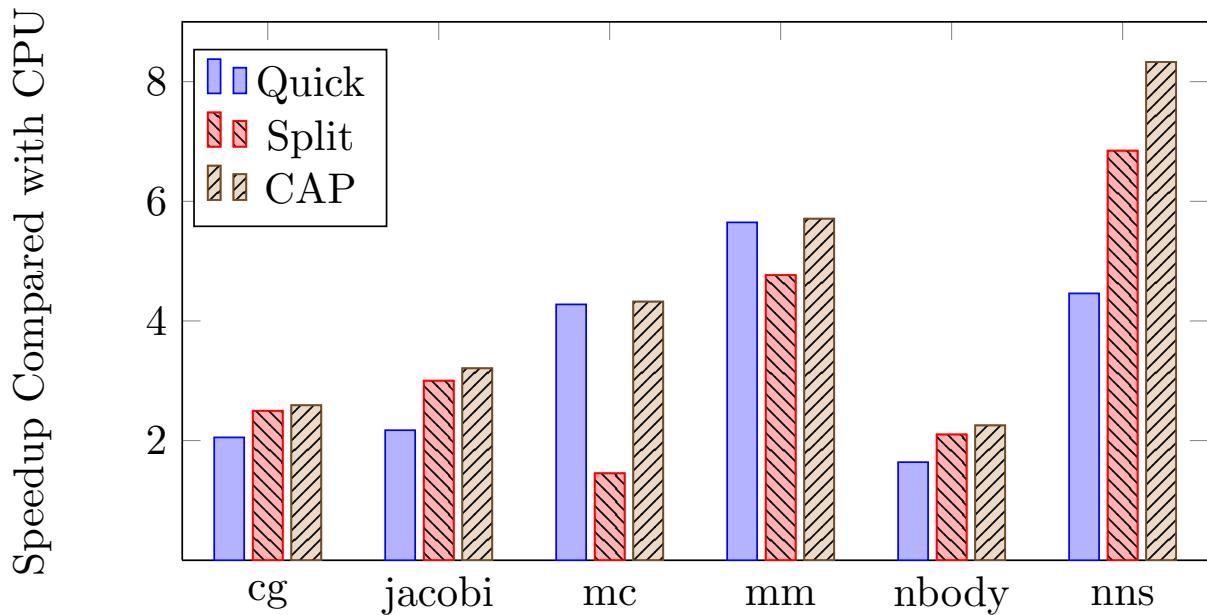


图 4-3 CAP 的加速比  
Figure 4-3 Speedup Of CAP

表 4-3 CAP 的性能提升

Table 4-3 Improvement of CAP

性能评测程序	相比快速调度	相比分割调度	性能采集阶段的比例
cg	26.1%	3.6%	37.9%
jacobi	47.6%	6.9%	26.8%
mc	1.1%	197.3%	3.1%
mm	1.1%	19.8%	3.1%
nbody	37.5%	7.2%	26.8%
nns	86.7%	21.6%	25.0%

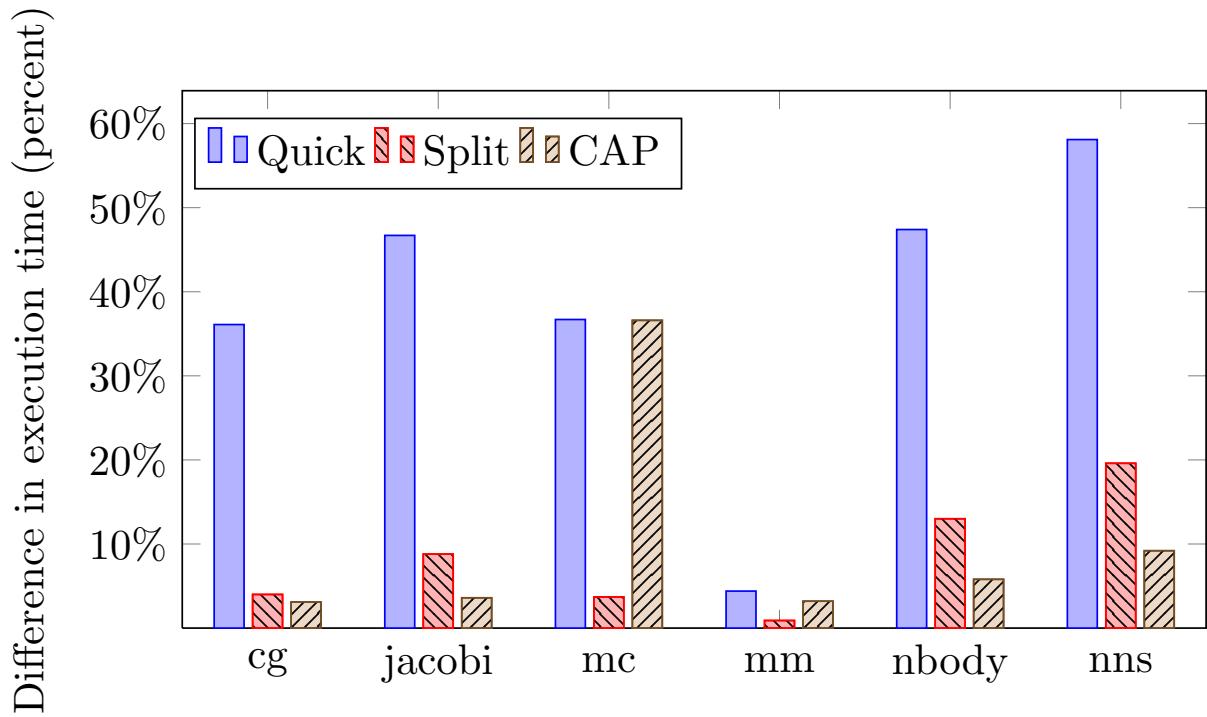


图 4-4 CAP 中 CPU 和 GPU 执行时间的差距（越小越好）

Figure 4-4 Difference in execution time between GPU and CPU of CAP (the smaller the better)

序，其他性能评测程序里 CPU 和 GPU 的执行时间差距都非常大。这意味着在快速调度中，计算设备要花费  $\frac{1}{3}$  在等待上，这样计算性能没有得到充分利用。这反映了一小部分工作负载不足以估计 GPU 在不同工作负载下的性能。每个性能评测程序的工作负载大小都不一样。mc 和 mm 的工作负载较大，一小部分工作负载也能达到性能曲线的稳定点。对于 mm 程序，快速调度已经可以很好的平衡负载，因此 CAP 的提升较小 (1.1%)。CAP 的提升来自于额外一次的性能信息带来的更准确的负载划分。

分割调度算法的负载均衡做的更好，但频繁的同步和过小的工作负载降低了 GPU 的性能。分割调度在 mc 和 mm 程序里性能很差，这是因为其总工作负载比其他程序要大，导致了同步的次数也随之变多。在其他性能评测程序里，分割调度的性能还不错。

CAP 动态的调整采集性能信息的比例。如果程序要采集更多的性能信息，那 CAP 就会采集更多。否则，CAP 就少采集几次来达到更好的性能。表 4-3展示了 CAP 对 cg, jacobi, nbody 和 nns 花费了大概  $\frac{1}{4}$  到  $\frac{1}{3}$  的工作负载来采集性能信息。而对 mm, CAP 只用了  $\frac{1}{50}$  的工作负载。CAP 根据需要采集信息，而分割调度则每步都需要采集。在 mc 上 CAP 的负载均衡略差，但仍可以接受。

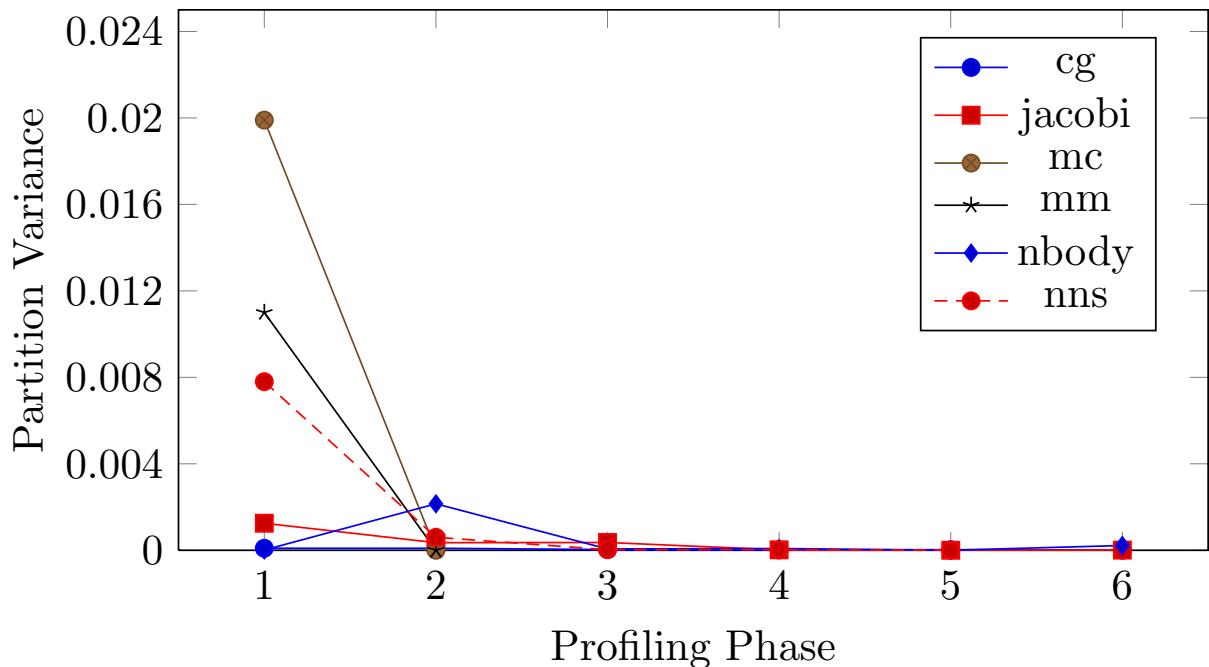


图 4-5 CAP 的每一步的划分之间的方差（越小越好）

Figure 4-5 Partition variance of CAP in each profiling step (the smaller the better)

在 CAP 中，最后一步执行的负载均衡和性能对整体性能和负载均衡有主要的影响。相比于最后一步，前面几步的工作负载并不大，其负载均衡对整体性能的影响也不大。因此，CAP 的负载均衡在 nbody 和 nns 上甚至比分割调度还好。

图 4-5展示了 CAP 在每一步里的划分之间的方差。划分之间的方法即当前一步的划分和上一步的划分的方差。这个方差显示了 GPU 性能在不同工作负载下的变化。如果变化很小，那么方差也会很小。否则方差就会比较大。CAP 试图通过计算方差来找到 GPU 性能曲线的稳定点。

性能采集的比例和阈值有关。如果阈值比较高，那么需要采集的工作负载的比例就会较小，但是负载均衡就会差一些。如果阈值比较低，那么就需要采集更多的工作负载。我们的实现里阈值为  $5 \times 10^{-5}$ 。也可以使用其他值来保持负载均衡和同步开销之间的平衡。

如果方差比阈值小，那么 CAP 就会停止性能采集。在这个图中，在 mm 程序里，CAP 的第一步采集就已经很准确了，因此 CAP 只需要多一步就可以保证负载均衡。其他程序也收敛的很快，一般在 5 步之内即可收敛。

CAP 的初始工作负载大小是静态指定的，这个参数也会影响性能。小的初始工作负

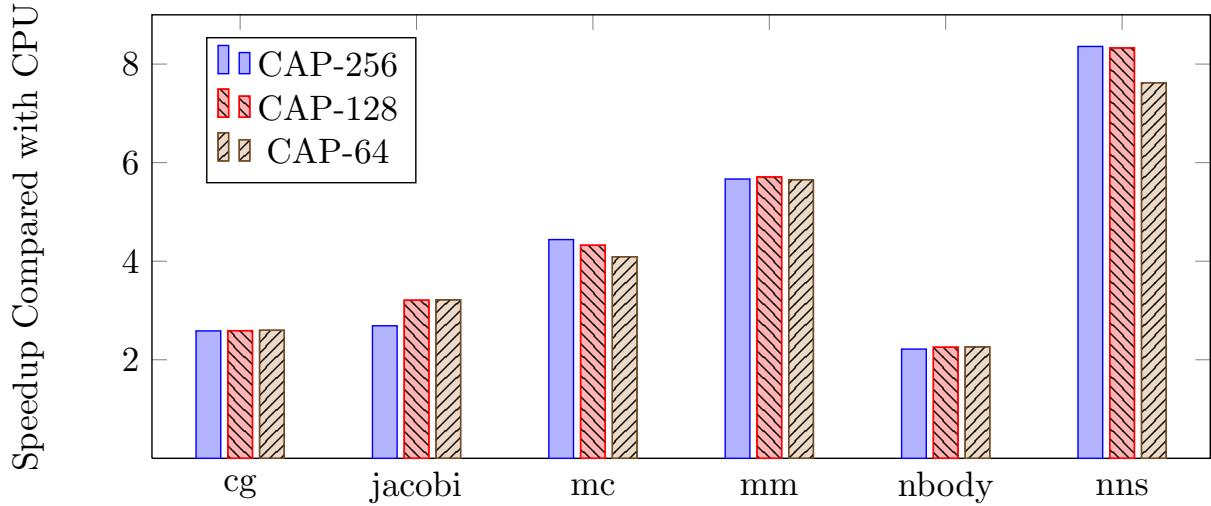


图 4-6 CAP 中不同初始工作负载大小的比较  
Figure 4-6 Comparison of different setting of initial workload

载可以避免第一步里的负载步均衡，但就需要更多次同步才能到达稳定点。由于工作负载较小，这也会降低 GPU 的性能。我们测试了三种不同的初始工作负载大小：总工作负载的  $\frac{1}{256}$ ,  $\frac{1}{128}$  和  $\frac{1}{64}$ 。在图 4-6 中表示为 CAP-256, CAP-128 and CAP-64。

图 4-6展示了不同设定下的加速比。如图所示，这三种设定对性能有影响，但影响都很小。 $\frac{1}{128}$  对各种程序都是一个比较好的设定，其他的值也不会提高多少性能。因此，CAP 不用调整参数就可以适应各种程序。

综上，相比于现有的协同调度算法，CAP 可以显著提高系统的整体性能。CAP 可以在不引入过多同步开销的前提下准确的估计 GPU 的性能。同时 CAP 的性能虽然与参数有关，但并不敏感。

## 4.7 本章小结

本章介绍了用于在 CPU 和 GPU 之间进行负载均衡的的基于渐近分析的 CPU+GPU 协同调度算法。这一工作发表在 PMAM 2013 和 Parallel Computing 上。

CPU 和 GPU 组成的异构系统已经逐渐流行起来。利用数据并行，系统里所有的计算设备都可以被利用起来以解决一个任务。现有的数据并行调度不能充分的利用 GPU 的性能。它们要么引入了过多的开销，要么不能均衡的划分工作负载。

我们提出了基于渐近分析的 CPU+GPU 协同调度算法来解决这个问题。我们的实验测试显示相比与现有调度算法，通过准确的估计 GPU 的性能，CAP 平均可以达到 42.7%

的性能提升。

尽管我们使用数据并行描述这一算法，但这一算法也可以通过记录任务的执行时间和工作负载大小来扩展到任务并行领域。这一领域另一个可能的工作是探索 CAP 的最优参数设定，CAP 有几个影响性能的参数，如每步增加多少工作负载，停止性能信息采集的阈值等。尽管这些参数对性能影响不大，但仍有提升的空间。最优的参数可能与硬件和运行的程序有关。第一步的负载划分方法也有改进的空间。我们使用的是静态划分方法，但基于性能模型的离线分析可以用来获取一个更好的估计。编译器可以为调度器提供这类信息。

节能也是这个领域的重要问题。CAP 没有考虑能耗，而只是根据负载均衡来提高性能。虽然程序运行的更快，但是消耗的能量也更多。在合适的处理器上运行合适的程序比使用所有的处理器的能效更高。调度器可以把功耗和性能一起考虑以获得更好性能功耗比。



## 第五章 EMU: QoS 感知的弹性资源竞争管理机制

### 5.1 研究背景

延迟敏感型 (**LS**) 交互式服务是新型数据中心的商业工作负载的关键组成部分，例如网页服务<sup>[12]</sup>，网页搜索<sup>[13]</sup> 和智能个人助理服务<sup>[9]</sup>。为用户提供稳定，快速的响应对提供互联网服务的公司是十分重要的。与使用单线程处理每个查询的传统数据中心应用（如网页搜索）相比，新型数据中心的应用，如 IPA 服务<sup>[9]</sup>，微软 Bing<sup>[10]</sup> 和金融服务<sup>[11]</sup> 等，对计算量的要求更高。在这些 **LS** 应用中，不同的查询有着不同的工作负载，而每个查询都需要用多个线程同时处理以保证较短的交互延迟。

为了保证 **LS** 应用的服务质量 (QoS)，这些程序经常单独运行在仓库规模数据中心里，并被给予了过量的资源。除此之外，由于新型服务的负载通常服从日间模式<sup>[101, 102]</sup>（即在非峰值时间里利用率很低），把延迟敏感型程序和尽力服务型 (**BE**) 程序放到一个数据中心里可以提高整个数据中心的效率。但是，将 **LS** 应用和 **BE** 应用放到同一台服务器上可能会产生资源竞争，并降低 **LS** 应用的服务质量。

之前有很多工作注意到了这个问题，并提出了在保证 QoS 的同时提高硬件利用率的方法<sup>[103, 104]</sup>。之前的工作要么通过预测一起运行的应用的 QoS 干扰，并识别出安全的一起运行的位置<sup>[105-109]</sup>，要么以粗粒度的方式在应用之间划分资源<sup>[103, 104]</sup>。尽管这些技术对不同查询具有相似的工作负载，并且每个查询都由一个线程处理的传统 **LS** 应用十分有效，它们并不适用于查询延迟受到更多因素影响的新型多线程 **LS** 应用。

图 5-1展示了在单线程/多线程 **LS** 应用 ( $P_1$ ) 和 **BE** 应用 ( $P_2$ ) 在同一个多核服务器上的性能干扰。传统的单线程 **LS** 应用的性能干扰主要来自于内存子系统（即图 5-1里的共享缓存和内存带宽）<sup>[110]</sup>，而多线程 **LS** 应用中，我们观察到性能干扰来自于① 核，② 共享缓存和③ 内存带宽。在计算资源（核）上的竞争对多线程 **LS** 查询的延迟有着更严重的影响。例如，如果一个 **LS** 查询没有被分配到足够多的核，即使没有共享缓存和内存带宽的竞争，由于其没有足够的计算能力，延迟依然会超过 QoS 目标。之前的工作并没有考虑到这一点，因此其不能应用在会受到核竞争影响的新型 **LS** 应用上。

本章试图在保证数据中心中多线程 **LS** 应用的 QoS 目标的前提下提高硬件利用率。多线程 **LS** 应用工作负载的高可变性为保证 QoS 目标和提高硬件利用率带来了三个新的挑战。第一，具有不同输入的查询往往需要不同大小的资源以满足其 QoS 目标。对于新型 **LS** 应用来说，最长的查询的工作负载可能是平均负载的 10 倍之多<sup>[111]</sup>。为了满足 **LS** 应用的 QoS 需求，所有的共享资源都必须针对每个查询来进行分配。第二，用低的

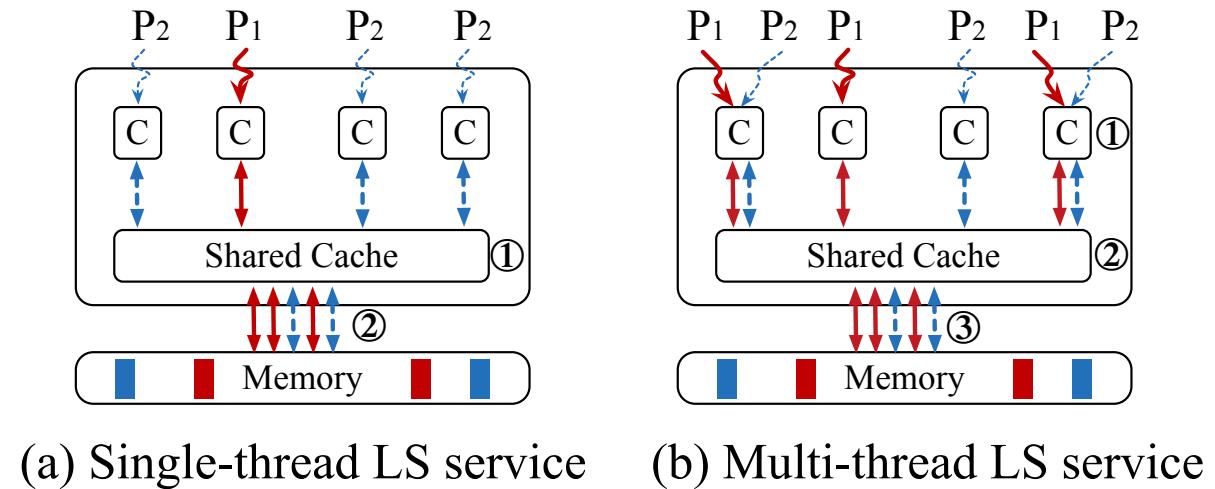
图 5-1 单线程/多线程 LS 应用 ( $P_1$ ) 和 BE 应用 ( $P_2$ ) 放到一个多核服务器上的性能干扰

Figure 5-1 Performance interference between single-thread/multi-thread LS application ( $P_1$ ) and the co-located BE applications ( $P_2$ ) on a multi-core server.

运行时开销来为一个查询找到最优的资源分配并不容易。这是由于核，共享缓存核内存带宽的资源竞争一起影响了查询的延迟。要根据不同的资源配置来预测查询的长度和在巨大的搜索空间中搜索最优的资源分配不容易。第三，要在运行时实时的监控查询的进度以根据实际执行进度重新调整资源分配不容易。

由于每个查询的输入只有在数据中心接收到输入之后才能得知，并且查询的输入各不相同，所以离线方法不能处理这一情况，并且粗粒度的资源分配方法也不能很好的工作（我们会在第 5.3 节详细讨论）。为了解决上述挑战，我们需要一个在线运行时系统来动态监控查询的进度，并根据实际执行情况来调整查询的资源分配，从而在保证 QoS 目标的前提下最大化硬件利用率。因此，我们提出了 **EMU** 系统，其具有三个组成部分：一个性能预测器，一个运行前资源分配器和一个运行时资源分配器。性能预测器利用性能模型来预测一个查询在不同输入和资源分配下的长度和 IPC。基于这一精确的预测，运行前资源分配器给查询分配足够的核和共享内存，令其刚刚好可以在 QoS 目标前完成。内存带宽并没有在应用之间分配，这是因为当前并没有实现这一分配的接口。在查询运行时，运行时资源分配器负责避免由于内存带宽竞争而导致的 QoS 违反。运行时资源分配器会监控查询的执行进度，并根据实际情况在 LS 应用和 BE 应用之间调整资源分配，以保证 QoS 和最大化 BE 应用的吞吐率。

这一章的主要贡献如下：

- 基于机器学习设计了查询性能预测模型。我们建立了一套精确的模型来预测一个 LS

查询的性能，包括其在不同输入和资源配置下的时长和处理的 IPC。

- 一个在应用之间分配各种共享资源的低开销机制。我们设计了一套有效的机制来搜索合适的资源配置，并为 LS 查询分配刚刚好的资源，以把剩下的资源分配给 BE 应用来最大化其吞吐率。
- 设计了一个在线监控 LS 查询进度和根据实际情况调整资源分配的机制。通过增加较慢的 LS 查询的资源，EMU 可以保证其 QoS。反之，通过减少较快的 LS 查询的资源，EMU 可以用多余的资源增加 BE 应用的吞吐率。

我们结合了上述技术实现了 EMU 运行时系统。我们的实验显示在新型 Xeon 处理器上，EMU 可以在保证新型 LS 应用 99% 尾延迟的前提下显著的提升 BE 应用的吞吐率。

表 5–1 EMU 和之前方法的比较

Table 5–1 Comparison between EMU and prior work.

	Loadleveler <sup>[112]</sup>	Bubble-Flux <sup>[106]</sup>	SMK <sup>[58]</sup>	Haque et.al <sup>[113]</sup>	SMiTe <sup>[107]</sup>	Li et.al <sup>[114]</sup>	Baymax <sup>[63]</sup>	Heracles <sup>[104]</sup>	EMU
<b>QoS 感知</b>		✓		✓	✓	✓	✓	✓	✓
<b>利用率提升</b>	✓	✓	✓		✓		✓	✓	✓
<b>适用于 CPU</b>	✓	✓		✓	✓	✓		✓	✓
<b>多线程感知</b>				✓		✓			✓
<b>细粒度资源管理</b>			✓				✓		✓

## 5.2 相关工作

当前已有许多专注于在保证延迟敏感型 (Latency-Sensitive, LS) 应用的 QoS 需求的同时提高系统资源利用率的研究<sup>[104, 105, 109]</sup>。由于数据中心中的负载并非时刻都很重，把其他程序和高优先级的 LS 应用放在一起运行是一个提高资源利用率的主流技术。Loadleveler<sup>[112]</sup> 和 Maui<sup>[115]</sup> 通过回填 (backfill) 算法<sup>[116]</sup> 在服务器中分配任务以提高利用率。但是他们没有考虑一起运行的任务对共享资源的竞争，也无法保证 LS 应用的 QoS。

要保证 LS 应用的 QoS 有两种方法，第一种方法是通过分析程序，在分配时把不会干扰到 LS 应用的程序分配到一起。例如 Bubble-Up<sup>[105]</sup> 和 Bubble-Flex<sup>[106]</sup> 通过分析不会使 LS 应用性能受到影响的“安全”位置来分配程序。SMiTE<sup>[107]</sup> 扩展了 Bubble-Up，这个工作还分析了 SMT 中的干扰，从而进一步提高了资源利用率。Paragon<sup>[108]</sup> 和 Quasar<sup>[109]</sup> 通过给程序的特性分类，从而避免互相干扰的程序分配到一起运行。这些技术和 EMU 是独立的，并可以一起使用。这些方法可以被用于确认合适的让程序一起运行的位置，而 EMU 可以通过对资源的管理最大化 BE 应用的吞吐率。

第二种方法是通过智能化的资源管理保证 QoS。Leverich 等人<sup>[103]</sup> 提出了在 Linux 内核中加入新的 CPU 调度器来保证公平和 QoS。Heracles<sup>[104]</sup> 则通过分析不同程序的资

源需求和在负载较重时增加分配的资源来服务更多的请求。但是这些工作都是为负载稳定的单线程应用设计的，对于新型负载可变的多线程应用并不适用。

对于多线程的新型 LS 应用，已有工作专注于降低每个访问请求的延迟。Haque 等人<sup>[113]</sup> 提出增加长请求的并行度来降低延迟。Li 等人<sup>[114]</sup> 则提出了通过降低长请求的并行度来改善短请求延迟的方案。由于长请求的数量很少，这一方法并不会影响 LS 应用的 QoS。但是这两种方法都只关注保证应用的 QoS，而不能提高系统的利用率。

除了 CPU 的协同定位，也有工作研究如何保障 GPU 上 LS 应用的 QoS 和系统利用率<sup>[59, 63, 69]</sup>。Pai 等人<sup>[30]</sup> 发现同时在一个 GPU 上运行多个程序可以提高性能。Wang 等人<sup>[58]</sup> 提出在一个 SM 内部一起运行程序来提高利用率。Aguilera 等人<sup>[59]</sup> 设计了一个 SM 划分策略来保证 LS 应用的 QoS。Baymax<sup>[63]</sup> 提出了一套任务重排序算法来满足 QoS 需求。Mystic<sup>[69]</sup> 利用机器学习来找到集群里适合让程序一起运行的位置。但是，这些技术都是为 GPU 设计，并不适用于 CPU。

作为总结，表 5-1 总结了 EMU 和之前工作的区别。

### 5.3 研究动机

这一小节中，我们试图回答以下问题：

- LS 应用和 BE 应用一起运行时会有 QoS 违反么？
- 现在的 QoS 管理技术适用于多线程 LS 应用么？
- 我们要做什么才能在保证多线程 LS 应用的 QoS 的同时提高 BE 应用的吞吐率？

#### 5.3.1 真实系统配置

在我们的研究中，我们把 LS 应用和 BE 应用放在一起运行，并观察这些应用的性能干扰。在真实数据中心中，LS 应用一直作为服务运行。这些 LS 应用接收用户请求，并按照严格的 QoS 需求把结果及时返回给用户（QoS 目标通常在 100ms 到 200ms 之间<sup>[111, 117]</sup>）。另一方面，BE 应用并没有 QoS 需求，其运行越快越好。在我们的实验中，我们使用了 Sirius<sup>[9]</sup> 中的新型 IPA 服务作为 LS 应用，并使用 PARSEC 性能评测测试集<sup>[118]</sup> 中的程序作为 BE 应用。在实验中，每个程序用 12 个线程运行在 12 核的计算机上，每个 LS 应用只与一个 BE 应用一起运行来解释这些问题。实验硬件配置和使用的性能评测程序的细节可参考第 5.5 节。

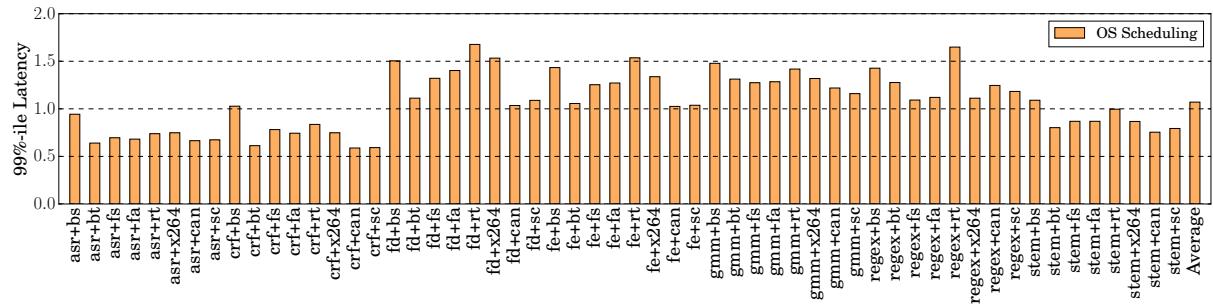


图 5-2 归一化到 QoS 目标的 LS 应用的 99% 尾延迟

Figure 5-2 The 99%-ile latency of LS applications normalized to their QoS target at co-location.

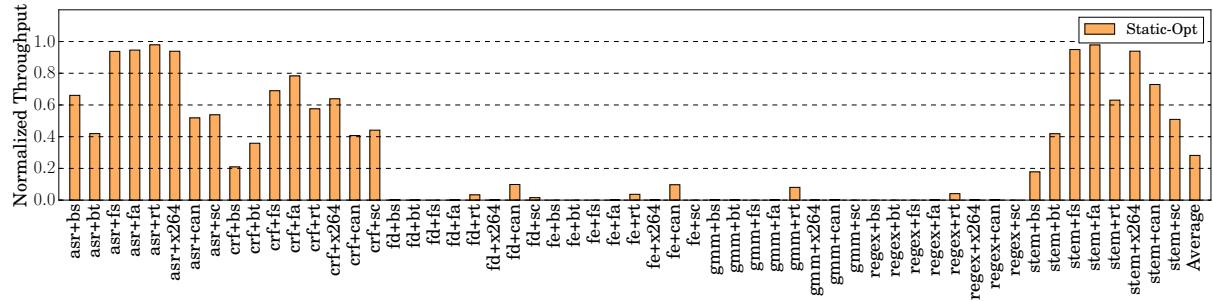


图 5-3 BE 应用的归一化到单独运行时性能的吞吐率

Figure 5-3 The throughput of BE applications at co-locations normalized to their solo-run throughput.

### 5.3.2 OS 调度的长尾延迟

图 5-2 展示了在使用操作系统 (OS) 来调度时 LS 应用归一化到其 QoS 目标的 99% 尾延迟。在图中 X 轴是 LS 应用和 BE 应用的组合；Y 轴是 LS 应用归一化到 QoS 目标的 99% 尾延迟。例如，X 轴上的“asr+bs”代表是 LS 应用 asr 和 BE 应用 bs 在一起运行。与之前的研究相同，我们这里使用 150ms 作为每个 LS 应用的 QoS 目标。如图所示，在 56 个配对中，由 34 个配对的 LS 应用违反了 QoS。即便只与一个 BE 应用一起运行，这些 LS 应用的平均 99% 尾延迟也为 QoS 目标的 1.07 倍，而最差情况是 1.67 倍。

QoS 违反来自于程序之间的共享资源竞争，其导致了 LS 应用的长尾延迟。在上面的实验中，两个程序一共 24 个线程在竞争 12 个核心。另外，图 5-4 展示了每个程序单独运行时的 L3 缓存使用和内存带宽使用。在我们的真实系统中，L3 缓存的大小是 30MB，而根据并行内存带宽测试程序<sup>[119]</sup>测出的实际内存带宽峰值是 54GB。根据图中所示，当两个应用一起运行时，L3 缓存和内存带宽的总需求都超过了硬件的限制（例

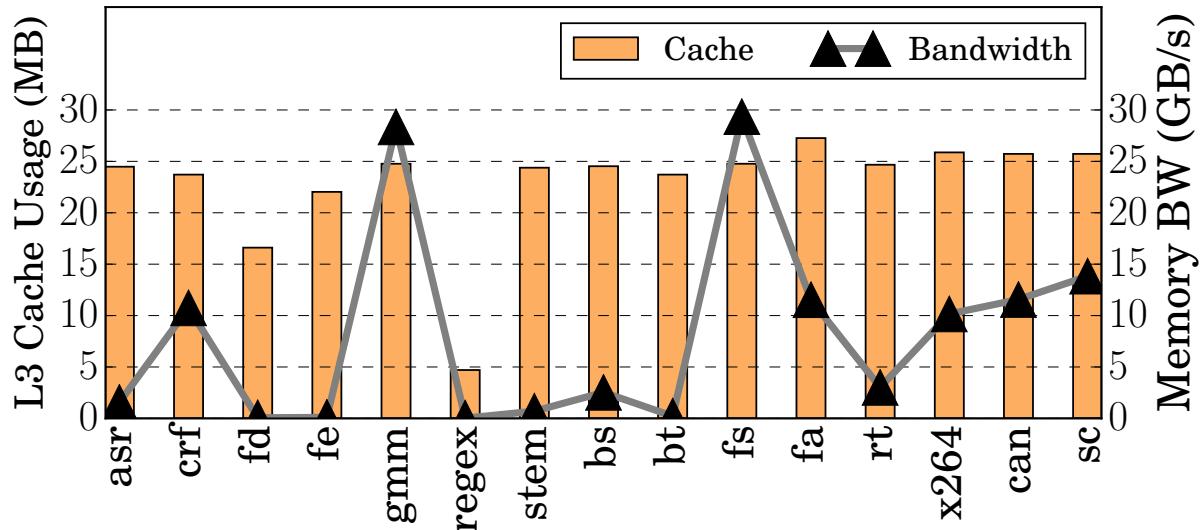


图 5-4 共享缓存与内存带宽使用情况

Figure 5-4 Shared cache and memory bandwidth usage.

如 *gmm+fs*)。因此，在执行中这些程序就会竞争共享的 L3 缓存和内存带宽。

综上，一起运行的应用的性能干扰来自于对核，共享缓存和内存带宽的竞争。这一结论与之前工作的发现一致<sup>[104, 106]</sup>。

### 5.3.3 现有技术的低硬件利用率

正如第 5.2 节中讨论的，现在的保证 LS 应用 QoS 和提高硬件利用率方法可以分为两类：识别安全的运行位置和保留共享资源。在第一种方法里，如果一个 BE 应用可以在 OS 调度不会导致 LS 应用违反 QoS 的情况下和 LS 应用一起运行，那么它们就可以运行在同一个节点上。否则，LS 应用就独自运行以保证 QoS。Bubble-Up<sup>[105-107]</sup> 是这类方法的代表性方法。通过细粒度离线分析，其可以在运行程序之前就识别出两个程序是否可以在一起运行。在第二种方法里，一个运行时系统，如 Heracles<sup>[104]</sup> 会周期性的检查查询的延迟，并在最近的 LS 查询违反了 QoS 的时候为 LS 应用分配更多的资源。

两种方法都只适用于当一个 LS 应用的每个查询都有着相似的工作负载，并只需要相同的资源来满足 QoS 需求的情况。但是，这一假设不适用于新型多线程 LS 应用<sup>[113, 114]</sup>。对于新型 LS 应用，长查询的工作负载可能是平均工作负载的 10 倍<sup>[111]</sup>。

**识别安全位置的技术会导致低硬件利用率。**为了展示这一技术的效果，我们测试了所有的配对，并在图 5-3 中报告了 BE 应用在每个配对中的吞吐率。在图中，Y 轴是 BE

应用归一化到其单独运行时的吞吐率的归一化吞吐率。如果吞吐率是 0，代表由于会违反 QoS，两个应用不能被放在一起运行。也就是说，图 5-3 中报告的吞吐率是这个技术所能达到的理想吞吐率。如图所示，我们发现 56 个配对中只有 33 个是安全的。即使在这 33 个配对中，BE 应用的吞吐率也很低。这是由于这个技术浪费了一些可以在保证 LS 应用 QoS 的同时一起运行的机会。例如，如果一个配对中只有几个查询有会超过 QoS 目标的长延迟，那么这个配对就是不安全的，然后这两个应用就不能在一起运行。但是，如果我们可以处理这些长延迟的时候减少 BE 应用的共享资源分配，并在处理短查询的时候增加 BE 应用的共享资源分配，就可以保证 LS 应用的 QoS 并提高 BE 应用的吞吐率。

**当前保留共享资源的技术会导致 QoS 违反和低硬件利用率。**在这一类技术中最先进的 *Heracles*<sup>[104]</sup> 中，LS 应用和 BE 应用的资源会每隔几秒（LS 查询的长度通常是几百毫秒）重新分配一次。这一粗粒度的资源分配只在 LS 查询的工作负载和资源需求相似时适用<sup>[104]</sup>。但是，在新型 LS 应用的场景中，根据之前短查询的低需求来降低 LS 的资源分配很容易导致在长查询时出现严重的 QoS 违反。另一方面，根据长查询的资源需求来增加 LS 应用的资源分配会降低 BE 应用在执行短查询时的吞吐率。

根据以上分析，现有的技术在处理新型负载可变的多线程 LS 应用<sup>[113, 114]</sup>时，要么会导致硬件利用率低，要么会造成 QoS 违反。

### 5.3.4 EMU 设计指导方针

为了保证多线程 LS 应用的 QoS 并最大化一起运行的 BE 应用的吞吐率，我们提出了 **EMU 运行时系统** 来根据每个 LS 查询动态的调整资源分配。我们根据以下三个方针来设计和实现 EMU：

- EMU 应该可以精确的预测一个查询在不同资源配置（例如核数，共享缓存大小）下的处理时间。利用这一点，EMU 可以找到一个正确的资源配置来保证每个查询的延迟。
- EMU 应该可以在大量可行的资源配置中找到最好的资源配置。利用这一点，EMU 可以为 LS 应用分配刚刚好的资源，并把余下的资源配置给 BE 应用。
- EMU 应该可以监控 LS 查询的执行进度，并避免由于内存带宽竞争产生的 QoS 违反。如果一个查询运行的比预想的慢，应该为其分配更多的资源以加速其运行。如果一个查询运行的比预想的快，可以把多余的资源配置给 BE 应用来提高其吞吐率。

## 5.4 EMU 的设计

图 5-5展示了 EMU 的设计概览，其包括三个部分：一个性能预测器，一个运行前资源分配器和一个运行时资源分配器。对每个查询，性能预测器利用多个低开销的机器学习算法来预测其在不同资源配置下的长度和处理速度（即 IPC）。基于预测得到的性能，EMU 从大量可行的资源配置中选择出一个可以让查询按时完成的合适的资源配置。同时，EMU 会在运行时根据查询的执行情况动态的调整资源分配以避免由于内存带宽竞争和性能预测偏差产生的 QoS 违反。

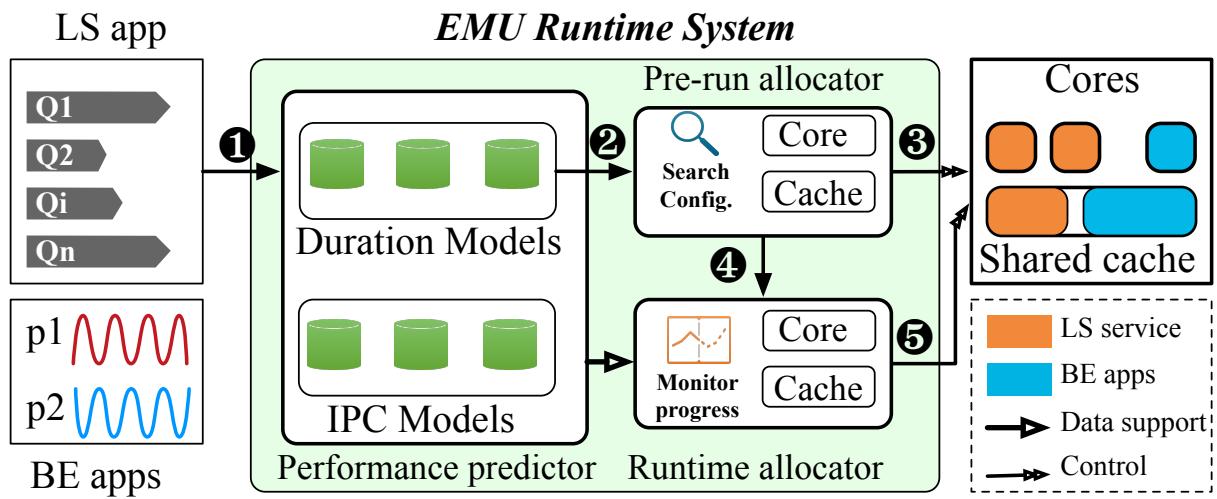


图 5-5 EMU 概览  
Figure 5-5 Overview of EMU.

EMU 用 5 步来保证 LS 应用的 QoS 并提高 BE 应用的吞吐率。当一个 LS 查询被提交时，①其输入被解析，并传递给查询性能预测器。②用输入和不同的资源配置作为性能模型的关键特征，EMU 预测查询在不同资源配置下的长度和对应的 IPC。③在这之后，根据预测得到的长度，运行前资源分配器给 LS 查询分配刚刚好的核和共享缓存，并把余下的资源分配给一起运行的 BE 应用。除此之外，在查询执行过程中，④运行时资源分配器利用硬件性能计数器监控查询的实际执行速度（即 IPC）。⑤通过对比预测的 IPC 与查询实际的 IPC，运行时分配器决定是否周期性的调整一起运行的 LS 应用和 BE 应用的资源分配。

在我们的实现中，我们使用 Linux 内核的处理器关联性接口和最新的 Intel Xeon V4 CPU 中的缓存分配技术（Cache Allocation Technology, CAT<sup>[120]</sup>）来为一个应用分配核和缓存。利用 CAT 技术，我们可以把共享缓存的某几路分配给特定的核，这样这几路

缓存就只能被这几个核访问，从而实现共享缓存划分。值得一提的是 EMU 不能直接在一起运行的应用之间分配内存带宽。这是由于现在的硬件并不支持这一特性。内存带宽的分配是通过调节核和缓存的分配间接实现的。我们会在第 5.4.3 节对此进行详细解释。

### 5.4.1 性能预测器

这一节中，我们展示了用于预测 LS 查询性能的方法，并展示了这一方法的预测准确性。

#### 5.4.1.1 预测方法

性能预测器依赖预先训练的性能模型来预测一个 LS 查询在不同资源配置下的性能（长度和 IPC）。

EMU 为每个 LS 应用单独构建性能模型。这是由于不同的 LS 应用常常具有完全不同的运行时行为和不同的性能特征。要找到一个适合具有不同特征的所有应用的性能模型十分困难。另外，如果 EMU 要调度一个新的 LS 应用，为那个应用训练一个单独的新性能模型要简单一些。相比之下，如果所有的应用都用一个统一的模型，现有的模型可能不适合新的应用，不能提供足够的准确度。并且用新应用的数据来更新统一的模型可能会影响其他应用的预测准确度。

为了构建准确的性能模型，我们选择输入数据大小，核的数量和共享缓存的大小作为输入特征，这是因为这些参数显著的影响一个 LS 查询的性能。尽管内存带宽也对 LS 查询的性能有影响，但现在的 CPU 中没有分配内存带宽的接口，因此我们的性能模型中不考虑这一特征。内存带宽在第 5.4.3 节的运行时资源分配器中进行分配。为了采集训练数据，我们运行了大量具有不同输入和不同资源配置（核数，共享缓存大小）的 LS 查询，并采集了其长度和对应的 IPC。在训练性能模型时，我们使用输入数据大小，核的数量和共享缓存大小作为输入，并使用对应的长度和 IPC 作为模型的输出。

#### 5.4.1.2 选择预测模型

LS 查询的 QoS 目标通常在几百毫秒的量级上以支持平滑的用户交互。因此，为性能预测器选择一个计算复杂度低且预测准确度高的预测算法十分重要。我们测试了下列预测模型来预测 LS 查询的长度和 IPC：*K* 最近邻居 (*K*-Nearest Neighbor, *KNN*)<sup>[121]</sup>，线性回归 (Linear Regression, *LR*)<sup>[122]</sup>，支持向量机 (Support Vector Machine, *SVM*)<sup>[123]</sup>，随机梯度下降 (Stochastic Gradient Descent, *SGD*)<sup>[124]</sup> 和多层次感知神经网络 (Multi-layer Perceptron Neural Network, *NN*)<sup>[125]</sup>。除了上述算法，深度神经网络 (Deep Neural Networks, *DNN*) 也有很高的预测准确度。但是，现有 DNN<sup>[22-25]</sup> 的预测开销太大，不能在运行时

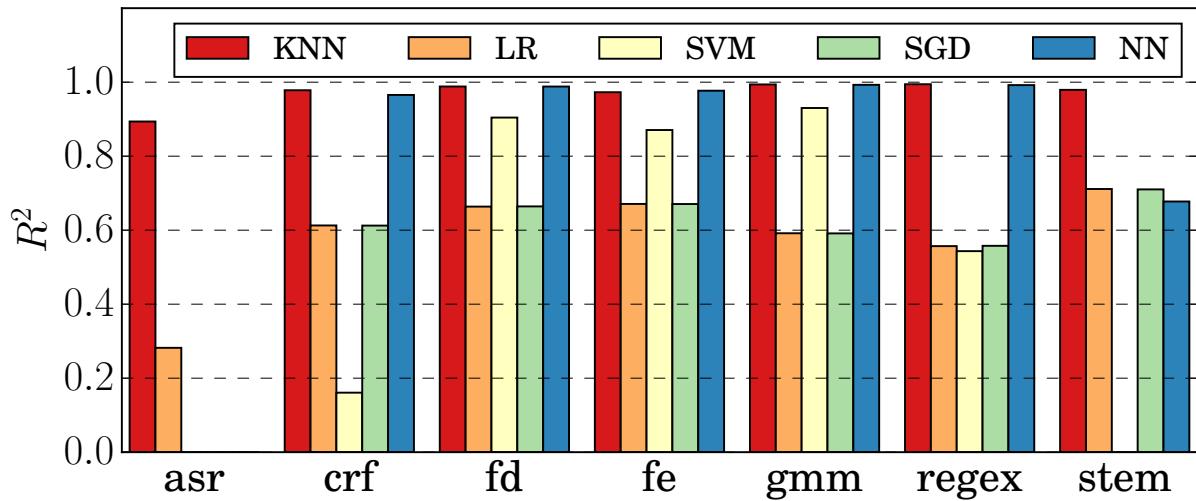


图 5–6 在测试集上预测查询时长的可决系数  $R^2$  的结果（越高越好）

Figure 5–6 The coefficient of determination  $R^2$  for predicting duration on the testing set (higher is more accurate).

使用。之前的工作<sup>[126]</sup> 显示 DNN 在高端 K40 GPU 上也需要几百毫秒才能做出一次预测，而我们真实系统的测试显示在最新的 Intel Xeon CPU 上用 DNN 进行预测需要几秒的时间才能完成。

图 5–6 和图 5–7 展示了上述机器学习算法在我们测试所使用的 LS 应用上的预测准确度。在实验中，我们采集了大量的数据，并随机选择其中的 80% 作为训练集，用剩下的 20% 作为测试集。我们使用可决系数 (*Coefficient of Determination*,  $R^2$ )，一个在机器学习领域被广泛使用的指标<sup>[127–129]</sup> 来评估我们的预测准确度。理论上说，在非线性模型上  $R^2$  可能是负数。在图 5–6 中，负数的情况（例如 SVM, SGD 和 NN 为 asr 的预测）被显示为 0。如图 5–6 和图 5–7 所示，KNN 和 NN 在长度预测和 IPC 预测上表现都不错。LR 和 SGD 的预测效果不好，这是因为它们都是为线性函数设计的。SVM 在 IPC 预测上表现不错，但长度预测准确度比较差。KNN 和 NN 是最适合为 EMU 进行性能预测的算法。但是，它们都不是完全比另一个好。KNN 在一些应用上表现较好（例如 asr 和 stem 的长度），而 NN 在另一些应用上表现较好（如 asr 的 IPC）。

除了预测准确度，我们还测量了每个模型做出一次预测所需的时间。根据我们的实验，用 KNN, LR, SGD 和 NN 做出预测所需的时间都小于 0.04 ms，而用 SVM 预测的用时超过 200 ms。因此，SVM 不能被用作 EMU 的运行时预测算法。

为了实现高预测准确度，我们为每个 LS 应用测试了上述预测算法 (KNN, LR,

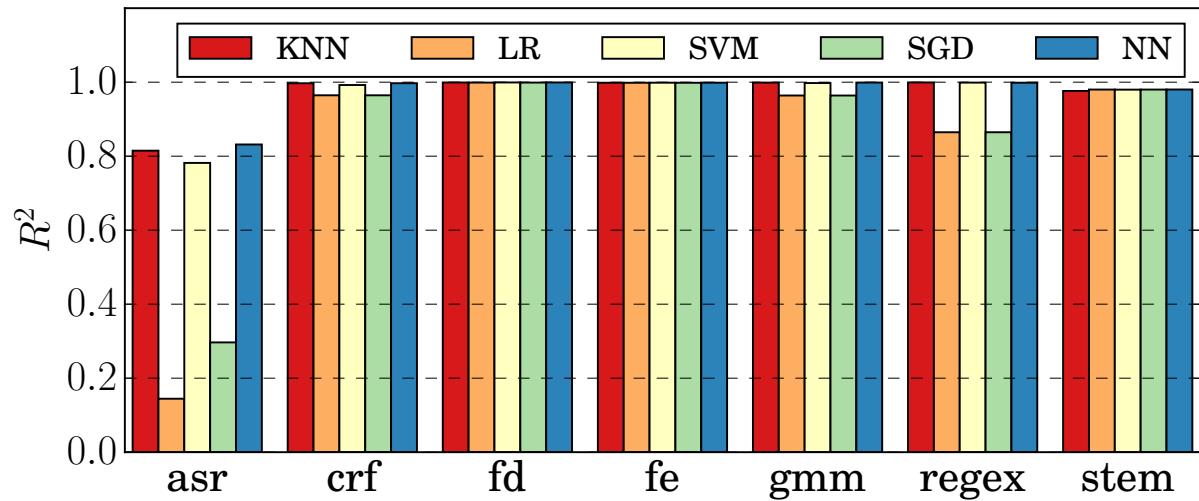
图 5-7 在测试集上预测查询 IPC 的可决系数  $R^2$  的结果（越高越好）

Figure 5-7 The coefficient of determination  $R^2$  for predicting IPC on the testing set (higher is more accurate).

SGD 和 NN)，并选择准确度最高的来在运行时预测每个 LS 查询的性能。由于这个模型选择的过程是离线进行的，其并不会引入任何运行时开销。

#### 5.4.2 运行前资源分配器

在预测了不同资源配置下的查询的长度和 IPC 之后，运行前资源分配器会找到一个刚刚好可以让 LS 查询  $Q$  按时完成的资源配置。

为了做到这一点，当  $Q$  开始运行的时候，EMU 用公式 5-1 计算出  $Q$  处理时间的上界， $T_{ub}$ ，即可以让其在 QoS 目标前返回的处理时间。在这个公式中， $T_{target}$  代表了每个查询的 QoS 目标，这个值由数据中心管理员指定，而  $T_{queued}$  是  $Q$  在开始执行之前的排队延迟。 $T_{queued}$  通过  $T_{start} - T_{issued}$  计算得出。 $T_{start}$  是查询开始执行的时间， $T_{issued}$  是查询发出的时间。

$$T_{ub} = T_{target} - T_{queued} \quad (5-1)$$

通过比较  $T_{ub}$  和查询  $Q$  在不同资源配置下的预测长度，运行前资源分配器可以识别出能够让查询  $Q$  在 QoS 目标内返回的安全的资源配置。对于一个资源配置，只有当查询  $Q$  的预测长度在这个资源配置下小于  $T_{ub}$  时，这个资源配置才是安全的。在我们的实现中，我们给  $T_{ub}$  留出了 20% 的区间来避免由于运行时资源分配带来的额外开销而导致的 QoS 违反。也就是说，在我们的实现中，只有在查询  $Q$  的预测长度小于  $80\% * T_{ub}$

时，这个资源配置才是安全的。这个 QoS 的区间和之前在 Heracles<sup>[104]</sup> 里用的区间一致。

---

### 算法 5-3 运行前资源分配算法

---

**输入：**输入数据大小和 LS 查询的时间限制  $T_{ub}$

```

1: 假设最大核数是  $N_{core}$ , 最大缓存大小是  $N_{cache}$ .
2:  $high \leftarrow N_{core}$ ;  $low \leftarrow 0$ 
3: while  $high > low + 1$  do
4:    $i \leftarrow (high + low) \div 2$ ;  $t \leftarrow predictor_{duration}(input, i, N_{cache})$ 
5:   if  $t < T_{ub}$  then
6:      $high \leftarrow i$ 
7:   else
8:      $low \leftarrow i$ 
9:   end if
10: end while
11:  $n_{core} \leftarrow high$ ;  $high \leftarrow N_{cache}$ ;  $low \leftarrow 0$ 
12: while  $high > low + 1$  do
13:    $i \leftarrow (high + low) \div 2$ ;  $t \leftarrow predictor_{duration}(input, n_{core}, i)$ 
14:   if  $t < T_{ub}$  then
15:      $high \leftarrow i$ 
16:   else
17:      $low \leftarrow i$ 
18:   end if
19: end while
20:  $n_{cache} \leftarrow high$ 
21: 将 LS 查询的线程绑定到  $n_{core}$  个核上;
22: 分配  $n_{cache}$  路缓存给这  $n_{core}$  个核;
23: 把 BE 应用的线程绑定到剩下的核上;
24: 把剩下的缓存分配给剩下的核;
```

---

在一个有很大共享缓存的多核 CPU 上，可行的资源配置有很多（不同数量的核，不同大小的共享缓存）。一个直接的搜索合适的资源配置的方法是穷尽式的搜索整个资源配置空间。也就是说，EMU 要在不同的核数和不同大小的共享缓存里循环，以找到最小的能够让查询  $Q$  的长度小于  $T_{ub}$  的核数和缓存大小。但是，这种穷尽式的搜索在真实系统中是不实际的，很大的搜索空间会带来运行时开销。例如，我们的实验平台有 12

个核和 16 路共享缓存，那么就总共有  $12 \times 16 = 192$  个不同的资源配置。我们的实验显示 EMU 需要  $8\text{ ms}$  的时间才能搜索完 192 个配置。这一穷尽搜索的开销相比 QoS 目标不能忽略。并且，随着核数和共享缓存的路数的增加，这一开销也会显著增加。

为了减少搜索的开销，EMU 使用二分搜索来减少找到合适的资源配置所需的预测数。算法 5-3 展示了运行前资源分配器用于搜索合适的资源配置并根据找到的资源配置分配资源的算法。如算法所示，在第一步中，我们把共享缓存的大小设置为最大，然后在核数上进行二分搜索以找到最小的核数。然后，我们根据找到的核数对共享缓存进行二分搜索以找到最小的共享缓存大小。使用这样的两步二分搜索方法，我们的算法只需要 8 次尝试就可以找到合适的资源配置，这一过程只需要  $0.3\text{ ms}$ 。值得一提的是，由于查询的性能和其被分配的资源之间呈正相关，算法 5-3 的二分搜索所找到的配置和穷尽搜索找到的配置是一样的。

设  $n_{core}$  和  $n_{cache}$  分别为找到的配置中核数和缓存的路数。根据这一配置，运行前资源管理器把  $n_{core}$  个核分配给 LS 查询，然后把  $n_{cache}$  用 Intel CAT 技术分配给这  $n_{core}$  个核。值得一提的是，EMU 允许多个 LS 查询同时在一个服务器上运行。在这种情况下，EMU 会为每个 LS 查询用算法 5-3 搜索合适的资源配置，并为它们分配资源。只有在所有的 LS 查询的资源都被满足了之后，剩下的资源才会被分配给一起运行的 BE 应用。

在找到了合适的资源分配之后，EMU 根据这个资源配置预测查询  $Q$  的 IPC。预测得到的 IPC ( $IPC_{pred}$ ) 用于在运行时（第 5.4.3 节）调整查询  $Q$  的资源分配。

### 5.4.3 运行时资源分配器

运行前资源分配器基于精准的性能预测来为应用分配核和共享缓存。根据我们的研究，除了核和共享缓存的竞争，内存带宽的竞争也会影响一起运行的程序的性能。但是这一因素并未在性能预测器中加以考虑。

由于一起运行的程序的内存带宽竞争，查询  $Q$  在给定资源配置下可能运行的比预想的慢。由于没有考虑内存带宽竞争，运行前资源分配器可能会导致较长的  $Q$  的延迟。为了解决这一问题，我们提出了一个运行时资源分配器来动态的根据运行时信息来更新资源配置。具体地说，我们周期性的收集  $Q$  的实际 IPC 和其已经完成了多少工作。根据比较  $Q$  的实际 IPC 和  $Q$  的预测 IPC，运行时资源分配器可以更新资源配置，来消除 QoS 违反或增加资源利用率。

运行时资源分配器周期性的使用算法 5-4 来重新分配共享的资源。在这个算法中， $W_{pred} = IPC_{pred} \times T_{pred}$ ，其中  $W_{pred}$ ,  $IPC_{pred}$  和  $T_{pred}$  分别代表了查询  $Q$  的预测工作负载， $Q$  的预测 IPC 和  $Q$  的预测执行时间。

在执行  $Q$  的过程中，EMU 收集每个调整间隔（Adjustment Interval）里  $Q$  的实际

**算法 5-4** 运行时资源分配算法

**输入:** 预测 IPC  $IPC_{pred}$ ,  $W_{pred} \leftarrow IPC_{pred} \times T_{pred}$ , 初始核分配  $Core_{initial}$  和缓存分配  $Cache_{initial}$ .

- 1: 在调整间隔结束的时候根据性能计数器计算当前查询  $Q$  的 IPC,  $IPC_{current}$ 。
- 2:  $W_{comp} \leftarrow W_{comp} + IPC_{current} \times T_{interval}$
- 3:  $T_{run} \leftarrow T_{run} + T_{interval}$
- 4: **if**  $T_{run} \geq T_{ub}$  **then**
- 5:     把所有的核和缓存分配给 LS 应用
- 6: **else**
- 7:      $F \leftarrow ((W_{pred} - W_{comp}) / (T_{ub} - T_{run})) / IPC_{current}$
- 8:      $F \leftarrow \max\{F, IPC_{pred} / IPC_{current}\}$
- 9:      $n_{core} \leftarrow n_{core} * F; n_{cache} \leftarrow n_{cache} * F$
- 10:    **if**  $F > 1$  **then**  $n_{core} \leftarrow n_{core} + 1; n_{cache} \leftarrow n_{cache} + 1$
- 11:    **end if**
- 12:     $n_{core} \leftarrow \max\{Core_{initial}, \min\{N_{core}, n_{core}\}\}$
- 13:     $n_{cache} \leftarrow \max\{Core_{initial}, \min\{N_{cache}, n_{cache}\}\}$
- 14:    应用新的资源配置
- 15: **end if**

IPC。 $Q$  的实际运行时间 (设为  $T_{run}$ ) 和  $Q$  完成的工作量 (设为  $W_{comp}$ ) 根据算法 5-4 的第 2-3 行计算得出。根据以上信息, 公式 5-2 计算出了查询  $Q$  要按时完成所需要的 IPC, 即  $IPC_{required}$ 。在这个公式中,  $T_{ub}$  由公式 5-1。

$$IPC_{required} = \frac{W_{pred} - W_{comp}}{T_{ub} - T_{run}} \quad (5-2)$$

设查询  $Q$  在一个间隔里的实际 IPC 是  $IPC_{current}$ 。在这个间隔结束的时候, 公式 5-3 计算出一个缩放系数  $F$  来决定如何在下一个间隔里为  $Q$  重新分配资源。在这个公式里,  $\frac{IPC_{required}}{IPC_{current}}$  反映了所需的 IPC 是否比  $Q$  的实际 IPC 高, 而  $\frac{IPC_{pred}}{IPC_{current}}$  反映了预测的 IPC 是否比  $Q$  的实际 IPC 高。实际 IPC 由于运行时的内存带宽竞争可能会偏低。如果  $Q$  的实际 IPC 比按时完成  $Q$  所需的 IPC 低, 或是比预测的 IPC 低, 那么就应该给  $Q$  分配更多的资源以让其可以保证 QoS 的完成。

$$F = \max\left\{\frac{IPC_{required}}{IPC_{current}}, \frac{IPC_{pred}}{IPC_{current}}\right\} \quad (5-3)$$

在计算得到  $F$  之后, 运行时资源分配器根据算法 5-4 的第 9-14 行调整资源分配。在

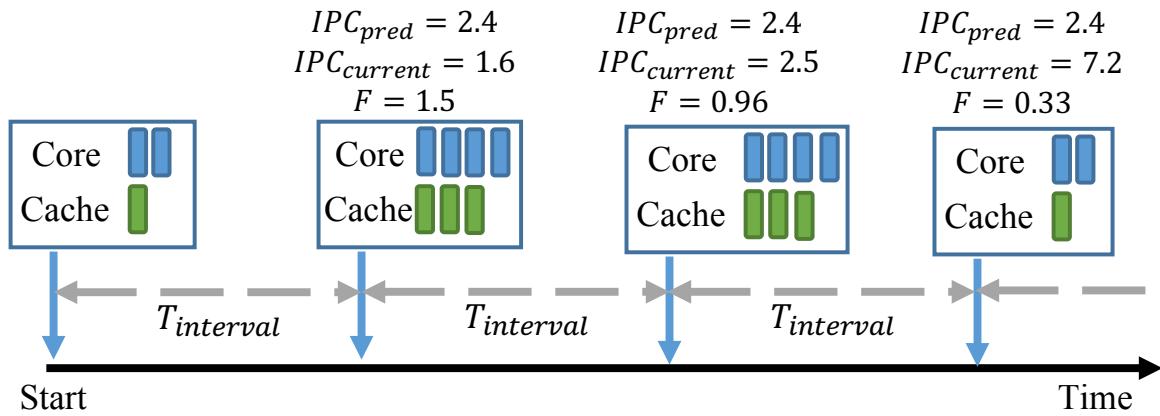


图 5-8 运行时资源分配的例子

Figure 5-8 An example of runtime resource allocation.

这个算法中，我们线性的缩放核数和缓存大小来增加或降低  $Q$  的运行速度。当  $Q$  的核数上升的时候，由于一起运行的 BE 应用的核数减少了， $Q$  会使用更多的内存带宽。但是，只增加核数而不增加共享缓存的大小并不能提高  $Q$  的性能，这在  $Q$  是内存密集型应用的时候尤其突出。因此，我们也同时增加了分配给  $Q$  的共享缓存的大小。

上述方案假设了分配给  $Q$  的资源（核数和共享缓存大小）和  $Q$  的性能呈线性关系以降低运行时资源分配的开销。但是，一个应用的性能通常是核数和缓存大小的次线性关系。由于次线性关系，如果我们减少  $Q$  里  $a\%$  的资源，其性能降低会小于  $a\%$ ；反之，如果我们为  $Q$  多分配了  $a\%$  的资源，其性能增加也会小于  $a\%$ 。因此，在我们减少  $Q$  的资源分配时这不是问题， $Q$  的实际性能会比我们预测的高。但是如果我们需要给  $Q$  分配更多的资源以加速其运行，像算法 5-4 第 9 行里那样线性的增加资源可能不足以让  $Q$  的 IPC 达到  $IPC_{required}$ 。为了解决这个问题，我们在  $F > 1$  的时候为  $Q$  额外增加了一个核和一路缓存（第 10-11 行）。这样调整之后的 IPC 就更有可能达到预测的 IPC。如果一次调整不能满足需求，运行时资源分配器会接着调整资源分配。

运行时资源分配器不仅在  $Q$  的性能不足时增加其资源分配，还在  $Q$  的性能过高时减少其资源分配以提高一起运行的 BE 应用的吞吐率。图 5-8 给出了一个运行时分配的例子。图中的数字只为解释说明，不代表真实执行的情况。在这个例子中，查询  $Q$  开始有 2 个核，1 路共享缓存，预测的 IPC ( $IPC_{pred}$ ) 为 2.4。这个配置是通过运行前资源分配器得到的。在一个调整间隔之后，运行时资源分配器采集到的查询  $Q$  的实际 IPC ( $IPC_{current}$ ) 是 1.6。由于  $IPC_{current}$  小于  $IPC_{pred}$ ，需要给  $Q$  分配更多的核。根据算法 5-4，在新的资源分配中，核数增加到  $2 \times 1.5 + 1 = 4$ ，而共享缓存路数增

表 5–2 软硬件环境和性能评测程序

Table 5–2 Hardware, software, and benchmarks

	参数
硬件	CPU: Intel Xeon E5-2650 V4 @ 2.20GHz 12 个硬件核, 30MB 共享缓存 (16 路)
软件	CentOS 6.8 x86 64 with kernel 2.6.32-642
性能评测程序集	应用
Sirius <sup>[9]</sup>	dnn-asr (asr), crf, fd, fe, gmm, regex, stemmer (stem)
PARSEC <sup>[118]</sup>	blackscholes (bs), bodytrack (bt), facesim (fs), fluidanimate (fa), raytrace (rt), x264, canneal(can), streamcluster(sc)

加到  $1 \times 1.5 + 1 = 3$ 。在下一个间隔中，新的配置是  $\max\{4 \times 0.96, 2\} = 3.84$  个核和  $\max\{3 \times 0.96, 1\} = 2.88$  路缓存。由于核数和缓存数都必须是整数，我们对结果进行四舍五入，并继续为  $Q$  分配 4 个核和 3 路共享缓存。在最后一个间隔里， $Q$  的实际 IPC 比所需 IPC 高很多。在这种情况下，分配给  $Q$  的核数核共享缓存路数分别从 4 和 3 减少到  $\max\{4 \times 0.33, 2\} = 2$  和  $\max\{3 \times 0.33, 1\} = 1$

由于在多个应用之间重新分配资源会引入线程上下文切换和 L3 缓存重填充的开销，调整间隔  $T_{interval}$  的大小对 EMU 的性能有影响。上下文切换的直接开销只有几个微秒，但是重填充 L2 缓存需要几毫秒<sup>[130]</sup>。由于 L3 缓存比 L2 缓存大且慢，重填充 L3 缓存所需的时间可能会更长。如果  $T_{interval}$  太小，重新分配的开销会占据调整间隔很大一部分，这样收集到的 IPC 就不能反映重新分配后的性能。如果  $T_{interval}$  太大，那么 LS 应用的资源就不能及时的根据 BE 应用的干扰而进行调整。我们会在第 5.5.5 节讨论 EMU 对  $T_{interval}$  的敏感性。

## 5.5 实验验证

### 5.5.1 实验方法

我们用一台包含了具有 CAT 机制的 Intel Xeon E5-2650 V4 CPU 的计算机来测试 EMU。详细的配置列在表 5–2 中。我们在实验中使用 12 个核和 16 路缓存。这些核的超线程技术被关闭。如表 5–2 所示，我们使用 Sirius<sup>[9]</sup> 中的 7 个性能评测程序作为 LS 应用，并使用 PARSEC 性能评测程序集<sup>[118]</sup> 中的 8 个程序作为 BE 应用。这些程序覆盖了各种可扩展性，缓存行为和内存带宽使用率的情况。图 5–4 展示了这些程序的缓存使用和内存带宽使用，而图 5–9 用每个程序在 12 个核上相比单核的加速比展示了这些程序

的可扩展性。

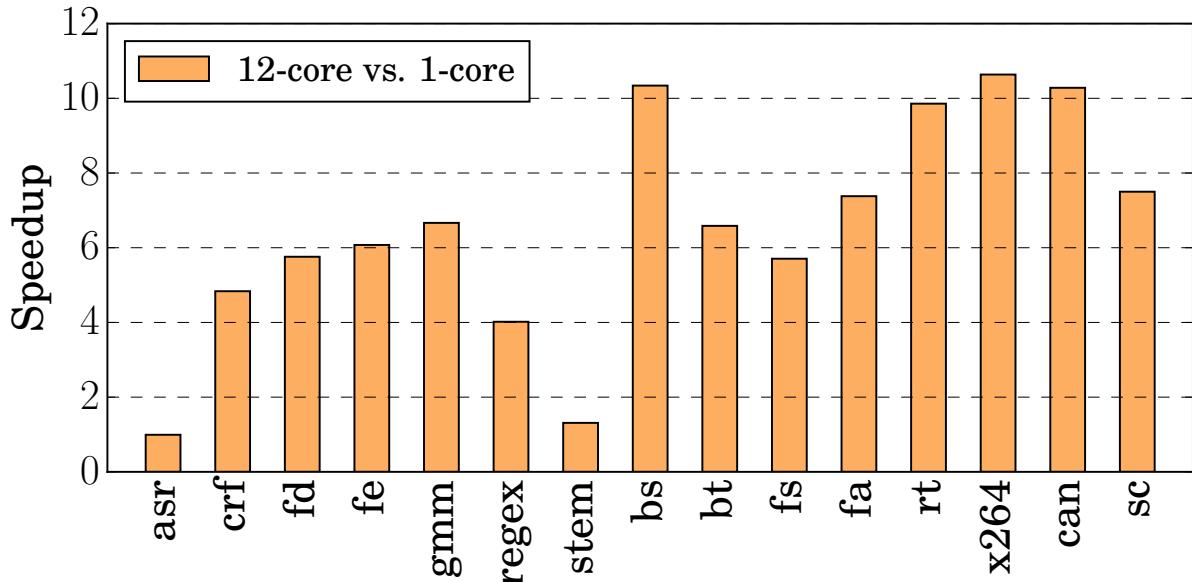


图 5-9 所有 LS 应用和 BE 应用在 12 个核上的加速比。这些性能评测程序有不同的可扩展性。

Figure 5-9 Speedup ratios of all the LS and BE benchmarks on 12 cores. The used benchmarks have various scalabilities.

EMU 的性能模型用 scikit-learn 库<sup>[131]</sup>进行训练。为了训练和测试我们的预测器，我们收集了很多数据，并随机选择其中的 80% 作为训练集，余下的 20% 作为测试集。每个 LS 应用都有不同长度和大小的多种数据。对于 KNN 模型，我们选择最近邻居的数量为 5 ( $K = 5$ )。默认情况下，EMU 使用的调整间隔为 20ms。

在这一节中，QoS 定义为 99% 尾延迟，而硬件利用率用 BE 应用的吞吐率来衡量。吞吐率越高，硬件处理 BE 应用的利用率就越高。在下面的实验中，BE 应用的吞吐率归一化到其单独运行时的吞吐率。值得一提的是，一个应用的归一化吞吐率可能超过 1，这是由于我们同时运行了多个 BE 应用的实例来最大化 Static-Opt 和 EMU 的资源利用率。

### 5.5.2 EMU 的性能

在这一节中，我们测试了 EMU 在提高 BE 应用吞吐率和保证新型 LS 应用的 QoS 上的效果。

图 5-10 分别展示了 Static-Opt 和 EMU 方法管理下的 LS 查询的 99% 尾延迟和一起运行的 BE 应用的吞吐率。在 Static-Opt 中，我们利用离线分析已经知道了最多能和每

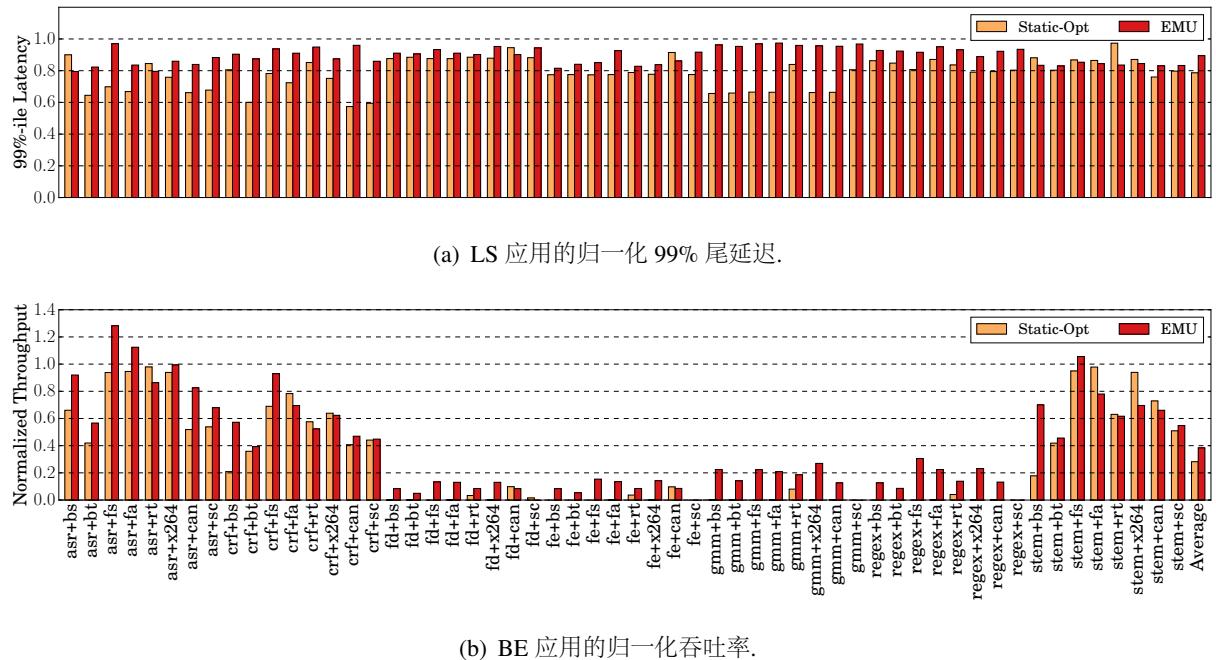


图 5-10 使用 Static-Opt 和 EMU 时 LS 应用的 99% 尾延迟和 BE 应用的吞吐率

Figure 5-10 The 99%-ile latency of LS applications and the throughput of BE applications with Static-Opt and EMU.

一个 LS 同时运行几个 BE 应用实例而不影响其 QoS。如果一起运行 BE 应用会导致 QoS 违反，那么 BE 应用就会被停掉。实际上，这种离线分析会影响真实系统数据中的 LS 应用的 QoS，因此其实际上是不可行的<sup>[105]</sup>。

从图 5-10(a)中可知，Static-Opt 和 EMU 都可以保证 LS 应用的 QoS。与需要预先知道两个应用是否可以一起运行的 Static-Opt 相比，EMU 不需要这种知识。EMU 利用准确的性能预测器核两个自适应的资源分配器动态的根据不同 LS 查询的资源需求分配资源。

从图 5-10(b)中可知，在大多数情况下，EMU 都比 Static-Opt 可以达到更高的 BE 应用吞吐率。在 Static-Opt 里，56 个配对中有 30 个配对是不安全的，因此它们不能一起运行。例如，大多数 BE 应用都不能在 Static-Opt 下和 *fd*, *fe* 和 *gmm* 一起运行。结果就是，Static-Opt 在这些情况下的吞吐率很低，或是 0。相反，EMU 可以让这些 LS 应用和 BE 应用一起运行。EMU 动态的根据每个 LS 查询的输入和 BE 应用的干扰来动态的调节资源分配。这一细粒度资源管理方法允许 BE 应用尽量多的使用资源，但不会导致 QoS 违反。平均来看，BE 应用在 EMU 下的吞吐率为 0.383，相比 Static-Opt 有 36.02% 的提升。

细心的读者观察到 Static-Opt 在有些情况下（如 *stem+x264*）的吞吐率比 EMU 高。

这是由于如果应用可以在不违反 QoS 的情况下一起运行 (BE 应用的负载很轻), 不划分资源的运行可能会达到更好的资源利用率。但是, 这并不意味着 Static-Opt 在所有安全的情况下都可以超过 EMU。EMU 在有些安全的情况下也比 Static-Opt 好, 例如 *asr+fs*, *asr+fa* 和 *crf+fs*。EMU 更高的 BE 应用吞吐率来自于 EMU 可以支持更多的 BE 应用实例一起运行。例如, Static-Opt 只能支持两个 *fs* 实例和 *asr* 一起运行, 而 EMU 可以支持更多的 *fs* 实例处于激活状态。

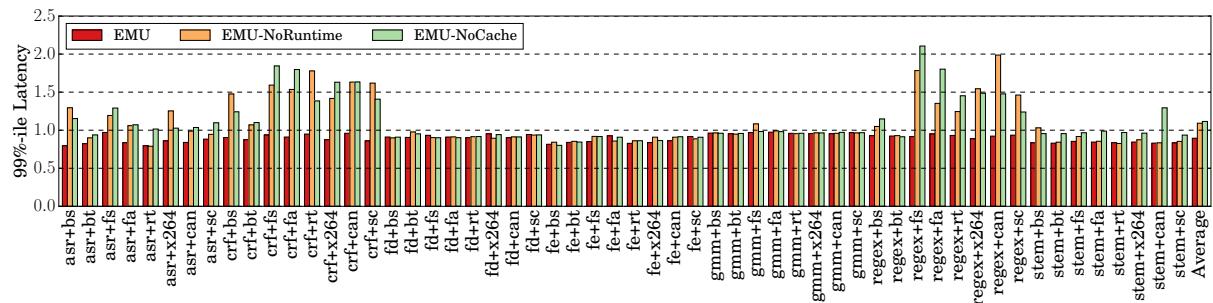


图 5-11 使用 EMU 及其变体时 LS 应用的 99% 尾延迟

Figure 5-11 The 99%-ile latency of LS applications with EMU and its variants.

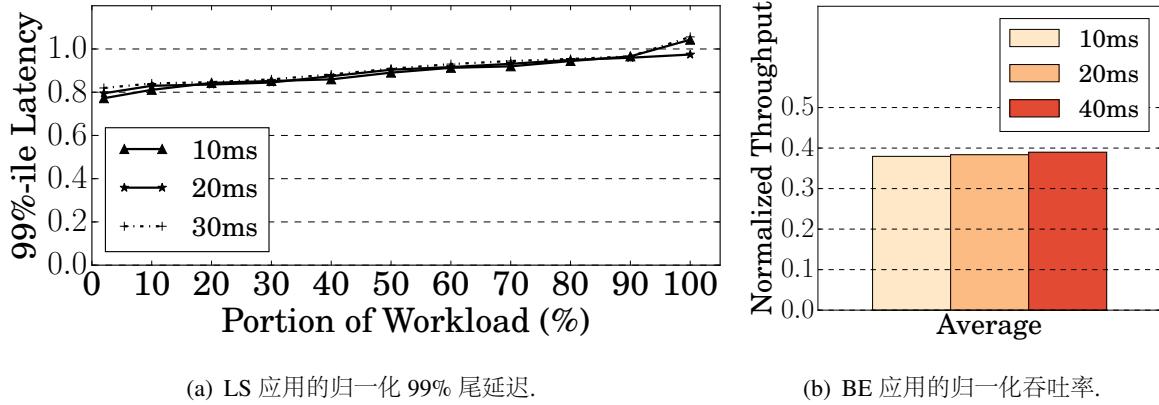
### 5.5.3 运行时机制的效果

在这个实验中, 我们测试了 EMU 运行时资源分配器的有效性, 其负责根据内存带宽竞争来调整资源分配以避免 QoS 违反。我们报告了一个关闭运行时资源分配器的 EMU 版本的性能 (设为 EMU-NoRuntime)。

图 5-11 展示了 EMU 和 EMU-NoRuntime 的 LS 应用的 99% 尾延迟。如图所示, 56 种情况里 EMU-NoRuntime 有 21 个 QoS 违反。在最差的情况下, *regex+can* 里的 *regex* 有 2 倍的 QoS 违反。这一结果显示来自 BE 应用的内存带宽竞争会显著的影响一起运行的 LS 应用的性能。即使 BE 应用使用的内存带宽不多 (如 *can* 和 *sc*), 其多个实例依然可以用尽内存带宽。因此, 运行时资源分配器对避免由于内存带宽竞争导致的 QoS 违反是必要的。

### 5.5.4 动态缓存分配的效果

在运行时资源分配器中, 分配给 LS 查询的共享缓存的大小和核数一起增长。为了测试这一设计选择的有效性, 我们使用一个运行时资源分配器不增加共享缓存大小的 EMU 版本 (设为 EMU-NoCache)。



(a) LS 应用的归一化 99% 尾延迟.

(b) BE 应用的归一化吞吐率.

图 5-12 使用不同调节间隔时 LS 应用的 99% 尾延迟和 BE 应用的吞吐率

Figure 5-12 The 99%-ile latency of LS applications and the throughput of BE applications with EMU using different adjustment intervals.

如图 5-11 所示，在使用 EMU-NoCache 时，56 个配对里有 23 个配对的 LS 应用有 QoS 违反。这一 QoS 违反情况与 EMU-NoRuntime 类似。因此，在调节核数的时候如果不调节共享缓存大小是不能避免来自 BE 应用的干扰的。

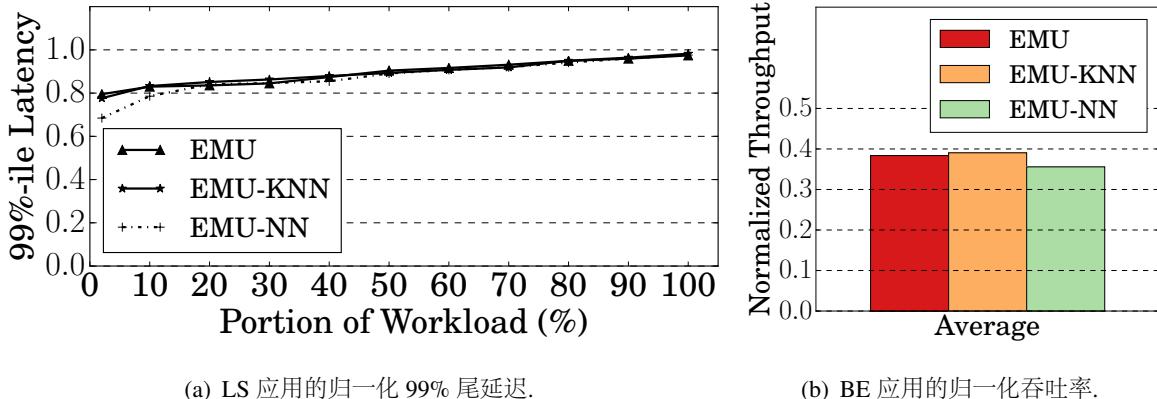
减少 BE 应用的核数可以减少其发射的内存请求的个数，降低其内存带宽使用。但是，增加 LS 应用的核数也会增加 LS 查询在 L3 缓存的工作集的大小。由于 L3 缓存在多个核之间共享，更多的并发线程意味着更多的数据需要被加载到 L3 缓存里。如果分配给 LS 查询的缓存的大小是固定的，其会导致更高的 L3 缓存未命中率和更高的内存带宽使用。这一因素导致了 EMU-Nocache 尽管为 LS 查询分配了更多的核，其性能还是较差。因此，随着核数缩放缓存大小对 EMU 是必须的。这一设计可以有效的避免由于高缓存未命中率而导致的 QoS 违反。

### 5.5.5 调整间隔敏感性

在这个实验中，我们测试了调整间隔 ( $T_{interval}$ ) 对 EMU 性能的影响。图 5-12 展示了 LS 应用在 3 种不同调整间隔的 EMU (10ms, 20ms 和 40ms) 下的 LS 应用的 99% 尾延迟和 BE 应用的吞吐率。

$T_{interval}$  较小对 EMU 有两个影响。首先，由于很大一部分时间都在进行上下文切换和缓存重填，在线性能监控就变得不准确了。其次，频繁的资源重分配导致了较高的性能开销。如图所示，在 10ms 的调整间隔下，这两个因素导致了 56 个配对里有 1 个配对出现的轻微的 QoS 违反 (crf+rt 里的 crf)，并且吞吐率也比使用默认的 20ms 的 EMU 低了 1%。

另一方面，更大的  $T_{interval}$  可以降低资源重分配的频率。但是较大的间隔也会降低根据 LS 查询的需求动态伸缩资源的能力。使用  $40ms$  的调整间隔时，EMU 也产生了一个 QoS 违反 ( $asr+fs$  里的  $asr$ )，但是吞吐率比默认的  $20ms$  的 EMU 高了  $1.6\%$ 。综上，只要调整间隔在合理范围内，其对 EMU 的性能影响很小。



(a) LS 应用的归一化 99% 尾延迟.

(b) BE 应用的归一化吞吐率.

图 5-13 使用不同模型算法时 LS 应用的 99% 尾延迟和 BE 应用的吞吐率

Figure 5-13 The 99%-ile latency and the throughput for EMU with different performance modeling algorithms.

### 5.5.6 建模算法的敏感性

默认情况下，EMU 根据  $R^2$  自动选择用于预测查询长度和 IPC 的建模算法。另一种方法是为预测查询长度和 IPC 选定一种算法。我们通过比较 EMU 和 EMU 的两个分别使用 KNN 和 NN 算法来进行性能预测的变种（分别设为 EMU-KNN 和 EMU-NN）来测试这一方法，结果展示在图 5-13 中。

由图可知，EMU-KNN 和 EMU-NN 都能保证所有的 LS 应用的 99% 尾延迟。这主要是因为 KNN 和 NN 都有很好的预测精确度（见第 5.4.1 节）。由于 EMU 的运行时资源分配器，预测时的一点不准确性对 99% 尾延迟影响很小。例如，尽管 NN 预测  $asr$  的长度的准确性不好，但是其准确的 IPC 预测让运行时资源分配器在运行前分配之后为  $asr$  分配了更多的核和缓存。因此其 QoS 依然可以被满足。

但是，不好的运行前资源分配会导致更多的运行时开销。因此，EMU-NN 的吞吐率比 EMU 低了  $8.2\%$ 。EMU-KNN 在查询长度和 IPC 上的准确度都很好，因此其和默认的自动选择模型方法结果相近。EMU 所使用的自动选择方法适用性更好，而指定一个模型的方法在模型足够好时也可以给出很好的结果。

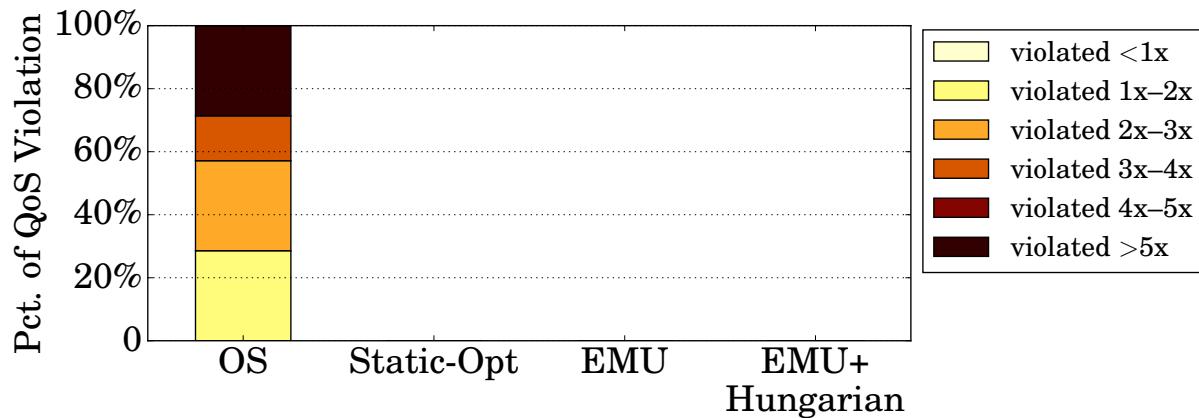


图 5-14 使用 OS 调度, Static-Opt 和 EMU 时所有协同定位的违反 QoS 的不同程度的比例

Figure 5-14 Percentage of QoS violation at different levels in all co-locations with OS, Static-Opt, and EMU.

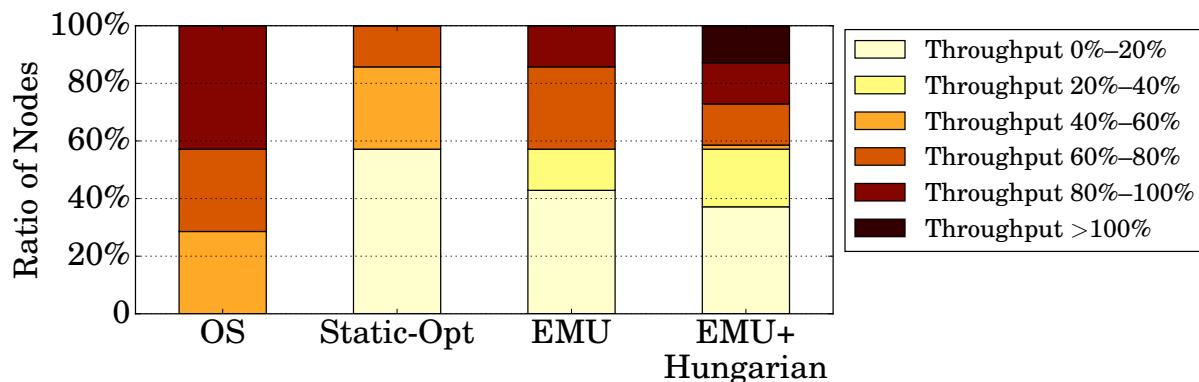


图 5-15 使用 OS 调度, Static-Opt 和 EMU 时所有协同定位的 BE 应用的不同吞吐率的比例

Figure 5-15 Percentage of nodes on which the BE applications' throughputs at different levels with OS, Static-Opt, and EMU.

### 5.5.7 应用到仓库级数据中心

在这一节中，我们不在单个节点上测试 EMU，而是在数据中心的场景下测试 EMU 的效果。在这个实验中，我们模拟了一个包含有 700 个具有 Intel Xeon E5-2650 V4 CPU 的节点的数据中心，每个 Sirius 里的 LS 应用分配到 100 个节点上。BE 应用由从表 5-2 中 PARSEC 程序的 5600 个实例组成（每个节点 8 个实例）。在这个实验中，我们让应用配对运行，并随机选择和 LS 应用一起运行的 BE 应用。

图 5-14 展示了一起运行时不同策略下 QoS 违反的百分比。前面的三个柱状图表示

了当每个节点的应用由操作系统, Static-Opt 和 EMU 管理时的 QoS 违反。如图所示, 在使用操作系统管理时, 所有的 LS 应用都有 QoS 违反, 并且由 71.4% 的 LS 应用承受着严重的 QoS 违反 (大于 2 倍的违反)。相比之下, Static-Opt 和 EMU 可以维持所有情况下 LS 应用的 QoS。除了随机把 LS 应用分配给集群里的 CPU (前三个柱状图), 我们还展示了使用匈牙利算法<sup>[132]</sup>进行任务分配的结果。当 BE 应用和每个 LS 应用配对的吞吐率通过离线分析已知的情况下, 匈牙利算法, 一个在集群级别进行组合优化的任务分配算法可以选择最佳的任务分配以达到最好的吞吐率 (设为 EMU+Hungarian)。也就是说, 这代表了 EMU 所能达到的最高的吞吐率。EMU+Hungarian 不会引发任何 QoS 违反。

图 5-15展示了节点上的 BE 应用分别在操作系统策略, Static-Opt 策略和 EMU 下的的吞吐率在不同级别的百分比。如图所示, BE 应用在默认的 OS 调度, Static-Opt, EMU 和 EMU+Hungarian 分别达到了 0.836, 0.231, 0.397 和 0.453 的吞吐率。尽管默认的 OS 调度有最高的吞吐率, 其也引发了图 5-14中所示的严重的 QoS 违反。相比 Static-Opt, EMU+Hungarian 通过应用匈牙利算法来在数据中心层面上选择的合适的配对, 并通过 EMU 来管理资源显著的提高了 BE 应用的吞吐率。

图 5-14和图 5-15展示了 EMU 可以在数据中心里在保证 LS 应用 QoS 的同时显著的提升 BE 应用的吞吐率。相比之下, 默认的 OS 调度会导致严重的 QoS 违反, 而 Static-Opt 则会导致较低的 BE 应用吞吐率。

## 5.6 本章小结

本章提出了一个新型资源管理系统 EMU 来在保证多线程 LS 应用的 QoS 目标的同时提高一起运行的应用的吞吐率。基于精确的运行时性能预测, EMU 识别一个查询所需要的资源量, 并为其分配刚刚好的资源来保证其按时完成。EMU 同时还监控每个用户查询的执行进度, 并根据运行时的反馈调整资源分配来在保证 QoS 的同时最大化吞吐率。实验测试结果表示 EMU 相比之前的方法提升了 BE 程序 36.02% 的吞吐率, 并满足了 LS 应用的 99% 尾延迟。



## 第六章 总结和展望

### 6.1 主要结论

本文围绕异构数据中心上的新型应用，从 GPU 的高效共享、GPU 的 QoS 支持、异构任务的负载均衡和负载可变的多线程应用的调度四个方面，深入的研究了异构数据中心中的硬件架构和调度算法。为了提高数据中心的性能和资源利用率，针对这四个问题，我们在体系结构层提出了同时多任务 GPU (*Simultaneous Multikernel GPU, SMK*) 和 GPU 细粒度共享的 QoS 支持机制，在调度层提出了基于渐进分析的 CPU+GPU 协同调度算法 (*Co-Scheduling Based on Asymptotic Profiling, CAP*) 和 EMU: QoS 感知的弹性资源竞争管理机制。经过深入细致的研究，本文取得了如下主要研究成果。

1. GPU 的高效共享机制研究：之前的研究表明单独运行的 GPU 程序可能会导致 GPU 资源利用率低下，而多个程序共享 GPU 可以显著的提高资源利用率。但是，当前的 GPU 架构并不支持硬件级的共享。而之前的研究工作提出的方法要么不能动态的共享 GPU (不适用于数据中心环境)，要么可以共享 GPU，但不能提高资源利用率。我们提出了同时多任务 GPU 架构来细粒度的共享 GPU。通过利用 GPU 应用的异构性，我们的方法可以充分利用 GPU 上的资源。具体来说，我们提出了公平静态资源分配策略和公平 warp 调度策略来在提高系统吞吐率的同时保障共享应用的公平性。实验表明在使用同时多任务 GPU 时，系统吞吐率相比单独执行平均提高了 37%，相比之前的方法平均提高了 12.7%。
2. GPU 的 QoS 支持机制研究：数据中心中的延迟敏感型应用会有 QoS 的需求，而当前的 GPU 并没有在硬件上支持保障应用性能的机制。之前有研究工作试图使用粗粒度资源分配的方法保证应用的 QoS，但这些方法只能满足较低的 QoS 需求，而不能适应更高的 QoS 需求。我们提出了细粒度共享的 QoS 支持机制，通过精确的控制静态资源和动态资源的分配，我们的方法可以在时钟周期级别上控制应用的执行进度。具体来说，我们提出了 QoS 感知的 warp 调度机制和线程级并行度控制机制来保证 QoS。实验表明我们提出的方法相比之前的方法可以达到的 QoS 目标多了 43.8%，且系统吞吐率平均提高了 20.5%。
3. 异构任务的负载均衡算法研究：异构数据中心中的同时使用 CPU 和 GPU 的异构任务的工作负载需要根据处理器的性能均衡的分配到 CPU 和 GPU 上才能达到最佳的性能。但 GPU 复杂的性能模型使在运行时准确预测 CPU 和 GPU 的性能比变得比较困难。之前的方法要么需要承受很高的运行时性能开销，要么不能准确的找到 CPU

和 GPU 的性能比。我们提出了基于渐进分析的协同调度算法，通过对 GPU 性能特征的分析，我们的方法只用几次性能采样就可以准确的分析出 CPU 和 GPU 之间的性能比，从而在处理器之间均衡的分配工作负载。具体来说，我们根据 GPU 的性能特征，快速的接近 GPU 性能的稳定区间来采集 GPU 的性能，从而找到 CPU 和 GPU 的性能比。实验表明我们提出的方法相比之前的方法平均提高了 42.7% 的性能。

4. 面向负载可变的多线程应用的 QoS 调度算法研究：当前数据中心需要保证新型负载可变的多线程应用的 QoS。如果单独运行这些应用，数据中心的利用率就不是很高。而如果让它们和其他程序一起运行，则 QoS 难以保证。之前的调度算法都只针对负载稳定的单线程应用设计，不适用于负载可变的多线程应用。我们提出了 EMU，一个基于机器学习的运行时资源管理机制，来为负载可变的多线程应用根据其输入分配资源，从而保证其 QoS。具体来说，我们设计了性能预测器、运行前资源分配器和运行时资源分配器来在运行前和运行时为应用准确的分配刚刚好够用的资源。实验表明我们提出的方法相比最优的静态方法提高了 36.02% 的系统吞吐率，同时保证了应用的 QoS。

## 6.2 研究展望

虽然本文对异构数据中心上的新型应用的硬件架构和调度算法进行了深入细致的研究，但仍有一些问题可以进行进一步的探索和研究。

1. GPU 的低功耗 QoS 调度机制与算法。我们可以研究在保证延迟敏感型应用 QoS 的同时降低 GPU 的功耗的方法（例如动态的调整 GPU 的频率，或是关闭不需要的核），从而降低系统的整体功耗。
2. 负载可变的多线程应用的能耗优化策略。我们可以研究根据应用的 QoS 需求和处理器的特性来最优的选择处理器的频率，或是让处理器尽快进入睡眠状态来降低系统的整体功耗。
3. 其他加速器的负载分配策略。我们可以研究其他加速器（例如 Intel Xeon Phi）的性能特征，并把我们的负载分配算法扩展到其他加速器上，使之更加通用。

## 参考文献

- [1] COMPUTING A, COMPUTING G. Torque resource manager[J]. Online] <http://www.adaptivecomputing.com>, 2012.
- [2] OPEN M. Open source high performance computing. 2012.
- [3] MELL P, GRANCE T, et al. The NIST definition of cloud computing[J]. 2011.
- [4] BARHAM P, DRAGOVIC B, FRASER K, et al. Xen and the art of virtualization[C]//ACM SIGOPS operating systems review. Vol. 37. 5. ACM. [S.l.]: [s.n.], 2003: 164–177.
- [5] HELSLEY M. LXC: Linux container tools[J]. IBM developerWorks Technical Library, 2009: 11.
- [6] VAQUERO L M, RODERO-MERINO L, CACERES J, et al. A break in the clouds: towards a cloud definition[J]. ACM SIGCOMM Computer Communication Review, 2008, 39(1): 50–55.
- [7] TURNER M, BUDGEN D, BRERETON P. Turning software into a service[J]. Computer, 2003, 36(10): 38–44.
- [8] KOOMEY J. Growth in data center electricity use 2005 to 2010[J]. A report by Analytical Press, completed at the request of The New York Times, 2011, 9.
- [9] HAUSWALD J, LAURENZANO M A, ZHANG Y, et al. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers[C]//Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ASPLOS '15. New York, NY, USA: ACM, 2015.
- [10] JEON M, HE Y, ELNIKETY S, et al. Adaptive Parallelism for Web Search[C/OL]//Proceedings of the 8th ACM European Conference on Computer Systems. EuroSys '13. Prague, Czech Republic: ACM, 2013: 155–168. ISBN: 978-1-4503-1994-2. <http://doi.acm.org/10.1145/2465351.2465367>. doi: 10.1145/2465351.2465367.
- [11] Parallelizing a computationally intensive financial R application with zircon technology. [J]. 2010.

- [12] Apache HTTP Server. [J/OL]. 2016. <https://httpd.apache.org>.
- [13] Apache Solr. [J/OL]. 2016. <http://lucene.apache.org/solr>.
- [14] BUTENHOF D R. Programming with POSIX threads[M]. [S.l.]: Addison-Wesley Professional, 1997.
- [15] BLUMOFE R D, JOERG C F, KUSZMAUL B C, et al. Cilk: An efficient multithreaded runtime system[M]. Vol. 30. 8. [S.l.]: ACM, 1995.
- [16] DAGUM L, MENON R. OpenMP: an industry standard API for shared-memory programming[J]. IEEE Computational Science and Engineering, 1998, 5(1): 46–55. doi: 10.1109/99.660313. issn: 1070-9924.
- [17] SHREINER D, GROUP B T K O A W, et al. OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1[M]. [S.l.]: Pearson Education, 2009.
- [18] GRAY K. Microsoft DirectX 9 programmable graphics pipeline[M]. [S.l.]: Microsoft Press, 2003.
- [19] Nvidia. Programming Guide. 2014. <http://docs.nvidia.com/cuda/cuda-c-programming-guide>.
- [20] BUCK I, FOLEY T, HORN D, et al. Brook for GPUs: Stream Computing on Graphics Hardware[C/OL]//ACM SIGGRAPH 2004 Papers. SIGGRAPH '04. Los Angeles, California: ACM, 2004: 777–786. <http://doi.acm.org/10.1145/1186562.1015800>. doi: 10.1145/1186562.1015800.
- [21] MUNSHI A. The OpenCL Specification, version 1.2[J].
- [22] SIMONYAN K, ZISSERMAN A. Very Deep Convolutional Networks for Large-Scale Image Recognition[C]//. ICLR '15. [S.l.]: [s.n.], 2015.
- [23] KRIZHEVSKY A, SUTSKEVER I, HINTON G E. ImageNet Classification with Deep Convolutional Neural Networks[G/OL]//Advances in Neural Information Processing Systems 25. Ed. by PEREIRA F, BURGES C J C, BOTTOU L, et al. [S.l.]: Curran Associates, Inc., 2012: 1097–1105. <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.
- [24] SZEGEDY C, LIU W, JIA Y, et al. Going Deeper With Convolutions[C]//The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). [S.l.]: [s.n.], 2015.

- [25] ZEILER M D, FERGUS R. Visualizing and Understanding Convolutional Networks[M/OL]//Computer Vision – ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6–12, 2014, Proceedings, Part I. Ed. by FLEET D, PAJDLA T, SCHIELE B, et al. Cham: Springer International Publishing, 2014: 818–833. [http://dx.doi.org/10.1007/978-3-319-10590-1\\_53](http://dx.doi.org/10.1007/978-3-319-10590-1_53). doi: 10.1007/978-3-319-10590-1\_53. ISBN: 978-3-319-10590-1.
- [26] HE J, ZHANG S, HE B. In-cache query co-processing on coupled CPU-GPU architectures[J]. Proceedings of the VLDB Endowment, 2014, 8(4): 329–340.
- [27] BOLZ J, FARMER I, GRINSPUN E, et al. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid[C]//ACM SIGGRAPH 2003 Papers. SIGGRAPH '03. [S.l.]: ACM, 2003: 917–924.
- [28] FAN Z, QIU F, KAUFMAN A, et al. GPU Cluster for High Performance Computing[C]//Proceedings of the 2004 ACM/IEEE Conference on Supercomputing. SC '04. [S.l.]: IEEE Computer Society, 2004.
- [29] HE B, FANG W, LUO Q, et al. Mars: A MapReduce Framework on Graphics Processors[C]//Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. PACT '08. [S.l.]: ACM, 2008: 260–269.
- [30] PAI S, THAZHUTHAVEETIL M J, GOVINDARAJAN R. Improving GPGPU Concurrency with Elastic Kernels[C]//Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '13. [S.l.]: ACM, 2013: 407–418.
- [31] LEE M, SONG S, MOON J, et al. Improving GPGPU resource utilization through alternative thread block scheduling[C]//High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on. [S.l.]: [s.n.], 2014: 260–271. doi: 10.1109/HPCA.2014.6835937.
- [32] PARK J J K, PARK Y, MAHLKE S. Chimera: Collaborative Preemption for Multi-tasking on a Shared GPU[C]//Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM. [S.l.]: [s.n.], 2015: 593–606.

- [33] TANASIC I, GELADO I, CABEZAS J, et al. Enabling Preemptive Multiprogramming on GPUs[C]//Proceeding of the 41st Annual International Symposium on Computer Architecuture. ISCA '14. [S.l.]: IEEE Press, 2014: 193–204.
- [34] ADRIAENS J, COMPTON K, KIM N S, et al. The case for GPGPU spatial multitasking[C]//High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on. [S.l.]: [s.n.], 2012: 1–12.
- [35] BRADLEY T. Hyper-Q example[J/OL]. 2012. [http://docs.nvidia.com/cuda/samples/6\\_Advanced/simpleHyperQ/doc/HyperQ.pdf](http://docs.nvidia.com/cuda/samples/6_Advanced/simpleHyperQ/doc/HyperQ.pdf).
- [36] NVIDIA. Sharing a GPU between MPI processes: multi-process service(MPS)[J]. 2012.
- [37] WU B, CHEN G, LI D, et al. Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations[C]//ICS' 15. [S.l.]: [s.n.], 2015.
- [38] WANG G, LIN Y, YI W. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU[C]//Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom). [S.l.]: [s.n.], 2010: 344–350.
- [39] GREGG C, DORN J, HAZELWOOD K, et al. Fine-grained resource sharing for concurrent GPGPU kernels[C]//4th USENIX Workshop on Hot Topics in Parallelism (HotPar). Berkeley, CA. [S.l.]: [s.n.], 2012.
- [40] ROSSBACH C J, CURREY J, SILBERSTEIN M, et al. PTask: Operating System Abstractions to Manage GPUs As Compute Devices[C]//Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. SOSP '11. [S.l.]: ACM, 2011: 233–248.
- [41] MENYCHTAS K, SHEN K, SCOTT M L. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators[C]//Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '14. [S.l.]: ACM, 2014: 301–316.
- [42] Kato, Shinpei and Lakshmanan, Karthik and Rajkumar, Raj and Ishikawa, Yutaka. TimeGraph: GPU scheduling for real-time multi-tasking environments[C]//Proc. USENIX ATC. [S.l.]: [s.n.], 2011: 17–30.

- [43] MENYCHTAS K, SHEN K, SCOTT M L. Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack.[C]//USENIX Annual Technical Conference. [S.l.]: [s.n.], 2013: 291–296.
- [44] KIM S W, OOI C.-L, EIGENMANN R, et al. Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution[C]//PPoPP '01. [S.l.]: ACM, 2001: 2–11.
- [45] GHODSI A, ZAHARIA M, HINDMAN B, et al. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.[C]//NSDI. Vol. 11. [S.l.]: [s.n.], 2011: 24–24.
- [46] NARASIMAN V, SHEBANOW M, LEE C J, et al. Improving GPU Performance via Large Warps and Two-level Warp Scheduling[C]//Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-44. [S.l.]: ACM, 2011: 308–317.
- [47] ROGERS T G, O'CONNOR M, AAMODT T M. Cache-Conscious Wavefront Scheduling[C]//Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-45. [S.l.]: IEEE Computer Society, 2012: 72–83.
- [48] EYERMAN S, EECKHOUT L. System-level performance metrics for multiprogram workloads[J]. IEEE micro, 2008, 28(3): 42–53.
- [49] BAKHODA A, YUAN G, FUNG W, et al. Analyzing CUDA workloads using a detailed GPU simulator[C]//Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on. [S.l.]: [s.n.], 2009: 163–174.
- [50] NVIDIA. NVIDIA Geforce GTX980 Whitepaper. 2014. [http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce\\_GTX\\_980\\_Whitepaper\\_FINAL.PDF](http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF).
- [51] STRATTON J A, RODRIGUES C, SUNG I.-J, et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing[J]. Center for Reliable and High-Performance Computing, 2012.
- [52] FUNG W W L, SHAM I, YUAN G, et al. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow[C]//Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 40. [S.l.]: IEEE Computer Society, 2007: 407–420.

- [53] ZHONG J, HE B. Medusa: Simplified Graph Processing on GPUs[J]. *Parallel and Distributed Systems, IEEE Transactions on*, 2014, 25(6): 1543–1552.
- [54] HAUSWALD J, KANG Y, LAURENZANO M A, et al. DjiNN and Tonic: DNN As a Service and Its Implications for Future Warehouse Scale Computers[C/OL]//Proceedings of the 42Nd Annual International Symposium on Computer Architecture. ISCA '15. Portland, Oregon: ACM, 2015: 27–40. ISBN: 978-1-4503-3402-0. <http://doi.acm.org/10.1145/2749469.2749472>. doi: 10.1145/2749469.2749472.
- [55] WANG M, XIAO T, LI J, et al. Minerva: A scalable and highly efficient training platform for deep learning. 2014.
- [56] FOUNDATION H. HSA Platform System Architecture Specification. 2015.
- [57] WANG Z, YANG J, MELHEM R, et al. Simultaneous Multikernel: Fine-grained Sharing of GPGPUs[J]. *Computer Architecture Letters*, 2015, PP(99): 1–1. doi: 10.1109/LCA.2015.2477405. issn: 1556-6056.
- [58] WANG Z, YANG J, MELHEM R, et al. Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-Grained Sharing[C]//2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). [S.l.]: [s.n.], 2016: 358–369. doi: 10.1109/HPCA.2016.7446078.
- [59] AGUILERA P, MORROW K, KIM N S. QoS-aware dynamic resource allocation for spatial-multitasking GPUs[C]//Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific. [S.l.]: [s.n.], 2014: 726–731. doi: 10.1109/ASPDAC.2014.6742976.
- [60] AGUILERA P, MORROW K, KIM N S. Fair share: Allocation of GPU resources for both performance and fairness[C]//Computer Design (ICCD), 2014 32nd IEEE International Conference on. [S.l.]: [s.n.], 2014: 440–447. doi: 10.1109/ICCD.2014.6974717.
- [61] XU Q, JEON H, KIM K, et al. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming[C]//Proceeding of the 43st Annual International Symposium on Computer Architecuture. ISCA '16. [S.l.]: IEEE Press, 2016.

- [62] MARGIOLAS C, O'BOYLE M F P. Portable and Transparent Software Managed Scheduling on Accelerators for Fair Resource Sharing[C/OL]//Proceedings of the 2016 International Symposium on Code Generation and Optimization. CGO 2016. Barcelona, Spain: ACM, 2016: 82–93. ISBN: 978-1-4503-3778-6. <http://doi.acm.org/10.1145/2854038.2854040>. doi: 10.1145/2854038.2854040.
- [63] CHEN Q, YANG H, MARS J, et al. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers[C/OL]//Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016: 681–696. ISBN: 978-1-4503-4091-5. <http://doi.acm.org/10.1145/2872362.2872368>. doi: 10.1145/2872362.2872368.
- [64] DUDA K J, CHERITON D R. Borrowed-virtual-time (BVT) Scheduling: Supporting Latency-sensitive Threads in a General-purpose Scheduler[C/OL]//Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles. SOSP '99. Charleston, South Carolina, USA: ACM, 1999: 261–276. ISBN: 1-58113-140-2. <http://doi.acm.org/10.1145/319151.319169>. doi: 10.1145/319151.319169.
- [65] YUAN W, NAHRSTEDT K. Energy-efficient Soft Real-time CPU Scheduling for Mobile Multimedia Systems[C/OL]//Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. SOSP '03. Bolton Landing, NY, USA: ACM, 2003: 149–163. ISBN: 1-58113-757-5. <http://doi.acm.org/10.1145/945445.945460>. doi: 10.1145/945445.945460.
- [66] GOYAL P, GUO X, VIN H M. A Hierarchical CPU Scheduler for Multimedia Operating Systems[C]//Proceedings of the 2nd USENIX Conference on Operating Systems Design and Implementation. OSDI '96. [S.I.]: USENIX Association, 1996.
- [67] CHANDRA A, ADLER M, GOYAL P, et al. Surplus Fair Scheduling: A Proportional-share CPU Scheduling Algorithm for Symmetric Multiprocessors[C/OL]//Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4. OSDI'00. San Diego, California: USENIX Association, 2000: 4–4. <http://dl.acm.org/citation.cfm?id=1251229.1251233>.
- [68] STOICA I, ABDEL-WAHAB H, JEFFAY K, et al. A Proportional Share Resource Allocation Algorithm for Real-time, Time-shared Systems[C/OL]//Proceedings of the 17th IEEE Real-Time Systems Symposium. RTSS '96. Washington, DC, USA: IEEE Com-

- puter Society, 1996: 288-. ISBN: 0-8186-7689-2. <http://dl.acm.org/citation.cfm?id=827268.828976>.
- [69] UKIDAVE Y, LI X, KAEKI D. Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning[C]//2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). [S.I.]: [s.n.], 2016: 353–362. doi: 10.1109/IPDPS.2016.73.
- [70] BAUTIN M, DWARAKINATH A, CHIUEH T.-C. Graphic engine resource management[J/OL]. Proc. SPIE, 2008, 6818. <http://dx.doi.org/10.1117/12.775144>. doi: 10.1117/12.775144.
- [71] YU M, ZHANG C, QI Z, et al. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming[C/OL]//Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing. HPDC '13. New York, New York, USA: ACM, 2013: 203–214. ISBN: 978-1-4503-1910-2. <http://doi.acm.org/10.1145/2462902.2462914>. doi: 10.1145/2462902.2462914.
- [72] PAI S, GOVINDARAJAN R, THAZHUTHAVEETIL M J. Preemptive Thread Block Scheduling with Online Structural Runtime Prediction for Concurrent GPGPU Kernels[C/OL]//Proceedings of the 23rd International Conference on Parallel Architectures and Compilation. PACT '14. Edmonton, AB, Canada: ACM, 2014: 483–484. ISBN: 978-1-4503-2809-8. <http://doi.acm.org/10.1145/2628071.2628117>. doi: 10.1145/2628071.2628117.
- [73] KAYIRAN O, NACHIAPPAN N C, JOG A, et al. Managing GPU Concurrency in Heterogeneous Architectures[C/OL]//Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-47. Cambridge, United Kingdom: IEEE Computer Society, 2014: 114–126. ISBN: 978-1-4799-6998-2. <http://dx.doi.org/10.1109/MICRO.2014.62>. doi: 10.1109/MICRO.2014.62.
- [74] SETHIA A, MAHLKE S. Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution[C/OL]//Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-47. Cambridge, United Kingdom: IEEE Computer Society, 2014: 647–658. ISBN: 978-1-4799-6998-2. <http://dx.doi.org/10.1109/MICRO.2014.16>. doi: 10.1109/MICRO.2014.16.

- [75] JOG A, KAYIRAN O, CHIDAMBARAM NACHIAPPAN N, et al. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance[C/OL]//Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '13. Houston, Texas, USA: ACM, 2013: 395–406. ISBN: 978-1-4503-1870-9. <http://doi.acm.org/10.1145/2451116.2451158>. doi: 10.1145/2451116.2451158.
- [76] LIU J, YANG J, MELHEM R. SAWS: Synchronization Aware GPGPU Warp Scheduling for Multiple Independent Warp Schedulers[C]//Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-48. [S.l.]: [s.n.], 2015.
- [77] JOG A, KAYIRAN O, PATTNAIK A, et al. Exploiting Core Criticality for Enhanced GPU Performance[C/OL]//Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science. SIGMETRICS '16. Antibes Juan-les-Pins, France: ACM, 2016: 351–363. ISBN: 978-1-4503-4266-7. <http://doi.acm.org/10.1145/2896377.2901468>. doi: 10.1145/2896377.2901468.
- [78] NVIDIA. GP100 Pascal Whitepaper[J/OL]. 2016. <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>.
- [79] LENG J, HETHERINGTON T, ELTANTAWY A, et al. GPUWattch: Enabling Energy Optimizations in GPGPUs[C/OL]//Proceedings of the 40th Annual International Symposium on Computer Architecture. ISCA '13. Tel-Aviv, Israel: ACM, 2013: 487–498. ISBN: 978-1-4503-2079-5. <http://doi.acm.org/10.1145/2485922.2485964>. doi: 10.1145/2485922.2485964.
- [80] LEE V W, KIM C, CHHUGANI J, et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU[C/OL]//Proceedings of the 37th Annual International Symposium on Computer Architecture. ISCA '10. Saint-Malo, France: ACM, 2010: 451–460. ISBN: 978-1-4503-0053-7. <http://doi.acm.org/10.1145/1815961.1816021>. doi: 10.1145/1815961.1816021.
- [81] RYOO S, RODRIGUES C I, BAGHSORKHI S S, et al. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA[C/OL]//Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '08. Salt Lake City, UT, USA: ACM, 2008: 73–82. ISBN: 978-1-

- 59593-795-7. <http://doi.acm.org/10.1145/1345206.1345220>. doi: 10.1145/1345206.1345220.
- [82] ZHANG Y, OWENS J D. A quantitative performance analysis model for GPU architectures[C]//2011 IEEE 17th International Symposium on High Performance Computer Architecture. [S.l.]: [s.n.], 2011: 382–393. doi: 10.1109/HPCA.2011.5749745.
- [83] HONG S, KIM H. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness[C/OL]//Proceedings of the 36th Annual International Symposium on Computer Architecture. ISCA '09. Austin, TX, USA: ACM, 2009: 152–163. ISBN: 978-1-60558-526-0. <http://doi.acm.org/10.1145/1555754.1555775>. doi: 10.1145/1555754.1555775.
- [84] BAGHSORKHI S S, DELAHAYE M, PATEL S J, et al. An Adaptive Performance Modeling Tool for GPU Architectures[C/OL]//Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '10. Bangalore, India: ACM, 2010: 105–114. ISBN: 978-1-60558-877-3. <http://doi.acm.org/10.1145/1693453.1693470>. doi: 10.1145/1693453.1693470.
- [85] HONG S, KIM H. An Integrated GPU Power and Performance Model[C/OL]//Proceedings of the 37th Annual International Symposium on Computer Architecture. ISCA '10. Saint-Malo, France: ACM, 2010: 280–289. ISBN: 978-1-4503-0053-7. <http://doi.acm.org/10.1145/1815961.1815998>. doi: 10.1145/1815961.1815998.
- [86] CHEN Q, CHEN Y, HUANG Z, et al. WATS: Workload-Aware Task Scheduling in Asymmetric Multi-core Architectures[C]//2012 IEEE 26th International Parallel and Distributed Processing Symposium. [S.l.]: [s.n.], 2012: 249–260. doi: 10.1109/IPDPS.2012.32.
- [87] DEAN J, GHEMARAT S. MapReduce: Simplified Data Processing on Large Clusters[J/OL]. Commun. ACM, 2008, 51(1): 107–113. <http://doi.acm.org/10.1145/1327452.1327492>. doi: 10.1145/1327452.1327492. ISSN: 0001-0782.
- [88] RAFIQUE M M, ROSE B, BUTT A R, et al. CellMR: A framework for supporting mapreduce on asymmetric cell-based clusters[C]//2009 IEEE International Symposium on Parallel Distributed Processing. [S.l.]: [s.n.], 2009: 1–12. doi: 10.1109/IPDPS.2009.5161062.

- [89] LIN H, MA X, ARCHULETA J, et al. MOON: MapReduce On Opportunistic eNvironments[C/OL]//Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. HPDC '10. Chicago, Illinois: ACM, 2010: 95–106. ISBN: 978-1-60558-942-8. <http://doi.acm.org/10.1145/1851476.1851489>. DOI: 10.1145/1851476.1851489.
- [90] BIALECKI A, CAFARELLA M, CUTTING D, et al. Hadoop: a framework for running applications on large clusters built of commodity hardware[J]. Wiki at <http://lucene.apache.org/hadoop>, 2005, 11.
- [91] AUGONNET C, THIBAULT S, NAMYST R, et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures[J]. Concurrency and Computation: Practice and Experience, 2011, 23(2): 187–198.
- [92] MCCORMICK P, INMAN J, AHRENS J, et al. Scout: a data-parallel programming language for graphics processors[J]. Parallel Computing, 2007, 33(10): 648–662.
- [93] LUK C.-K, HONG S, KIM H. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping[C/OL]//Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 42. New York, New York: ACM, 2009: 45–55. ISBN: 978-1-60558-798-1. <http://doi.acm.org/10.1145/1669112.1669121>. DOI: 10.1145/1669112.1669121.
- [94] BUENO J, PLANAS J, DURAN A, et al. Productive programming of GPU clusters with OmpSs[C]//Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. IEEE. [S.I.]: [s.n.], 2012: 557–568.
- [95] BAJAJ R, AGRAWAL D P. Improving scheduling of tasks in a heterogeneous environment[J]. IEEE Transactions on Parallel and Distributed Systems, 2004, 15(2): 107–118. DOI: 10.1109/TPDS.2004.1264795. ISSN: 1045-9219.
- [96] RADULESCU A, van GEMUND A J C. Fast and effective task scheduling in heterogeneous systems[C]//Proceedings 9th Heterogeneous Computing Workshop (HCW 2000) (Cat. No.PR00556). [S.I.]: [s.n.], 2000: 229–238. DOI: 10.1109/HCW.2000.843747.
- [97] JIMÉNEZ V J, VILANOVA L, GELADO I, et al. Predictive runtime code scheduling for heterogeneous architectures[C]//International Conference on High-Performance Embedded Architectures and Compilers. Springer. [S.I.]: [s.n.], 2009: 19–33.

- [98] GREWE D, O' BOYLE M F. A static task partitioning approach for heterogeneous systems using OpenCL[C]//International Conference on Compiler Construction. Springer. [S.l.]: [s.n.], 2011: 286–305.
- [99] SCOGLAND T R W, ROUNTREE B, c. FENG W, et al. Heterogeneous Task Scheduling for Accelerated OpenMP[C]//2012 IEEE 26th International Parallel and Distributed Processing Symposium. [S.l.]: [s.n.], 2012: 144–155. doi: 10.1109/IPDPS.2012.23.
- [100] BEYER J C, STOTZER E J, HART A, et al. OpenMP for accelerators[C]//International Workshop on OpenMP. Springer. [S.l.]: [s.n.], 2011: 108–121.
- [101] VASAN A, SIVASUBRAMANIAM A, SHIMPI V, et al. Worth their watts? - an empirical study of datacenter servers[C]//HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture. [S.l.]: [s.n.], 2010: 1–10. doi: 10.1109/HPCA.2010.5463056.
- [102] BARROSO L A, HÖLZLE U. The Case for Energy-Proportional Computing[J/OL]. Computer, 2007, 40(12): 33–37. <http://dx.doi.org/10.1109/MC.2007.443>. doi: 10.1109/MC.2007.443. issn: 0018-9162.
- [103] LEVERICH J, KOZYRAKIS C. Reconciling High Server Utilization and Sub-millisecond Quality-of-service[C/OL]//Proceedings of the Ninth European Conference on Computer Systems. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014: 4:1–4:14. isbn: 978-1-4503-2704-6. <http://doi.acm.org/10.1145/2592798.2592821>. doi: 10.1145/2592798.2592821.
- [104] LO D, CHENG L, GOVINDARAJU R, et al. Heracles: Improving Resource Efficiency at Scale[C/OL]//Proceedings of the 42Nd Annual International Symposium on Computer Architecture. ISCA '15. Portland, Oregon: ACM, 2015: 450–462. isbn: 978-1-4503-3402-0. <http://doi.acm.org/10.1145/2749469.2749475>. doi: 10.1145/2749469.2749475.
- [105] MARS J, TANG L, HUNDT R, et al. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations[C/OL]//Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-44. Porto Alegre, Brazil: ACM, 2011: 248–259. isbn: 978-1-4503-1053-6. <http://doi.acm.org/10.1145/2155620.2155650>. doi: 10.1145/2155620.2155650.

- [106] YANG H, BRESLOW A, MARS J, et al. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers[C/OL]//Proceedings of the 40th Annual International Symposium on Computer Architecture. ISCA '13. Tel-Aviv, Israel: ACM, 2013: 607–618. ISBN: 978-1-4503-2079-5. <http://doi.acm.org/10.1145/2485922.2485974>. doi: 10.1145/2485922.2485974.
- [107] ZHANG Y, LAURENZANO M A, MARS J, et al. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers[C/OL]//Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-47. Cambridge, United Kingdom: IEEE Computer Society, 2014: 406–418. ISBN: 978-1-4799-6998-2. <http://dx.doi.org/10.1109/MICRO.2014.53>. doi: 10.1109/MICRO.2014.53.
- [108] DELIMITROU C, KOZYRAKIS C. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters[C/OL]//Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '13. Houston, Texas, USA: ACM, 2013: 77–88. ISBN: 978-1-4503-1870-9. <http://doi.acm.org/10.1145/2451116.2451125>. doi: 10.1145/2451116.2451125.
- [109] DELIMITROU C, KOZYRAKIS C. Quasar: Resource-efficient and QoS-aware Cluster Management[C/OL]//Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014: 127–144. ISBN: 978-1-4503-2305-5. <http://doi.acm.org/10.1145/2541940.2541941>. doi: 10.1145/2541940.2541941.
- [110] TANG L, MARS J, VACHHARAJANI N, et al. The impact of memory subsystem resource sharing on datacenter applications[C]//Computer Architecture (ISCA), 2011 38th Annual International Symposium on. [S.l.]: [s.n.], 2011: 283–294.
- [111] DEAN J, BARROSO L A. The Tail at Scale[J/OL]. Commun. ACM, 2013, 56(2): 74–80. <http://doi.acm.org/10.1145/2408776.2408794>. doi: 10.1145/2408776.2408794. ISSN: 0001-0782.
- [112] KANNAN S, ROBERTS M, MAYES P, et al. Workload management with loadleveler[J]. IBM Redbooks, 2001, 2(2).
- [113] HAQUE M E, EOM Y H, HE Y, et al. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services[C/OL]//Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Op-

- erating Systems. ASPLOS '15. Istanbul, Turkey: ACM, 2015: 161–175. ISBN: 978-1-4503-2835-7. <http://doi.acm.org/10.1145/2694344.2694384>. doi: 10.1145/2694344.2694384.
- [114] LI J, AGRAWAL K, ELNIKETY S, et al. Work Stealing for Interactive Services to Meet Target Latency[C/OL]//Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '16. Barcelona, Spain: ACM, 2016: 14:1–14:13. ISBN: 978-1-4503-4092-2. <http://doi.acm.org/10.1145/2851141.2851151>. doi: 10.1145/2851141.2851151.
- [115] JACKSON D, SNELL Q, CLEMENT M. Core algorithms of the Maui scheduler[C]//Workshop on Job Scheduling Strategies for Parallel Processing. Springer. [S.l.]: [s.n.], 2001: 87–102.
- [116] MU'ALEM A W, FEITELSON D G. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling[J]. IEEE Transactions on Parallel and Distributed Systems, 2001, 12(6): 529–543. doi: 10.1109/71.932708. ISSN: 1045-9219.
- [117] PETRUCCI V, LAURENZANO M A, DOHERTY J, et al. Octopus-Man: QoS-driven task management for heterogeneous multicore in warehouse-scale computers[C]//2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). [S.l.]: [s.n.], 2015: 246–258. doi: 10.1109/HPCA.2015.7056037.
- [118] BIENIA C, KUMAR S, SINGH J P, et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications[C/OL]//Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. PACT '08. Toronto, Ontario, Canada: ACM, 2008: 72–81. ISBN: 978-1-60558-282-5. <http://doi.acm.org/10.1145/1454115.1454128>. doi: 10.1145/1454115.1454128.
- [119] BINGMANN T. Parallel Memory Bandwidth Benchmark[J/OL]. 2013. <https://panthemain.net/2013/pmbw>.
- [120] Intel. Intel Resource Director Technology[J/OL]. 2016. <https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html>.

- [121] COVER T, HART P. Nearest neighbor pattern classification[J]. *IEEE Transactions on Information Theory*, 1967, 13(1): 21–27. doi: 10.1109/TIT.1967.1053964. issn: 0018-9448.
- [122] SEBER G A, LEE A J. *Linear Regression Analysis*[M]. Vol. 936. [S.l.]: John Wiley & Sons, 2012.
- [123] SMOLA A J, SCHÖLKOPF B. A tutorial on support vector regression[J/OL]. *Statistics and Computing*, 2004, 14(3): 199–222. <http://dx.doi.org/10.1023/B:STCO.0000035301.49549.88>. doi: 10.1023/B:STCO.0000035301.49549.88. issn: 1573-1375.
- [124] GARDNER W. Learning characteristics of stochastic-gradient-descent algorithms: A general study, analysis, and critique[J/OL]. *Signal Processing*, 1984, 6(2): 113–133. <http://www.sciencedirect.com/science/article/pii/0165168484900136>. doi: [http://dx.doi.org/10.1016/0165-1684\(84\)90013-6](http://dx.doi.org/10.1016/0165-1684(84)90013-6). issn: 0165-1684.
- [125] HORNIK K, STINCHCOMBE M, WHITE H. Multilayer feedforward networks are universal approximators[J/OL]. *Neural Networks*, 1989, 2(5): 359–366. <http://www.sciencedirect.com/science/article/pii/0893608089900208>. doi: [http://dx.doi.org/10.1016/0893-6080\(89\)90020-8](http://dx.doi.org/10.1016/0893-6080(89)90020-8). issn: 0893-6080.
- [126] REN S, HE K, GIRSHICK R, et al. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks[G/OL]//*Advances in Neural Information Processing Systems 28*. Ed. by CORTES C, LAWRENCE N D, LEE D D, et al. [S.l.]: Curran Associates, Inc., 2015: 91–99. <http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.pdf>.
- [127] CAMERON A C, WINDMEIJER F A. An R-squared measure of goodness of fit for some common nonlinear regression models[J/OL]. *Journal of Econometrics*, 1997, 77(2): 329–342. <http://www.sciencedirect.com/science/article/pii/S0304407696018180>. doi: [http://dx.doi.org/10.1016/S0304-4076\(96\)01818-0](http://dx.doi.org/10.1016/S0304-4076(96)01818-0). issn: 0304-4076.
- [128] GUYON I, ELISSEEFF A. An introduction to variable and feature selection[J]. *Journal of Machine Learning Research*, 2003, 3(0): 1157–1182.

- [129] GYIMOTHY T, FERENC R, SIKET I. Empirical validation of object-oriented metrics on open source software for fault prediction[J]. IEEE Transactions on Software Engineering, 2005, 31(10): 897–910. doi: 10.1109/TSE.2005.112. issn: 0098-5589.
- [130] LI C, DING C, SHEN K. Quantifying the Cost of Context Switch[C/OL]//Proceedings of the 2007 Workshop on Experimental Computer Science. ExpCS '07. San Diego, California: ACM, 2007. isbn: 978-1-59593-751-3. <http://doi.acm.org/10.1145/1281700.1281702>. doi: 10.1145/1281700.1281702.
- [131] PEDREGOSA F, VAROQUAUX G, GRAMFORT A, et al. Scikit-learn: Machine learning in Python[J]. Journal of Machine Learning Research, 2011, 12(Oct): 2825–2830.
- [132] KUHN H W. The Hungarian method for the assignment problem[J]. Naval Research Logistics, 2005, 52(1): 7–21.

## 致 谢

时光飞逝，转眼间我在上海交通大学五年的博士生生涯即将结束。在此，我要感谢陪伴我度过这五年的各位老师、同学、朋友和亲人们。

感谢我的博士生导师过敏意教授。本文从论文选题、算法设计、实验方案设定、实验环境搭建和最后的论文撰写都凝聚了过敏意老师大量的心血。在我攻读博士学位的五年时间里，过敏意老师为我的研究提供了先进的实验平台，营造了宽松的学术氛围、提供了丰厚的生活津贴，还为我提供了出国交流学习的机会。这使我能够无后顾之忧地进行科学研究。过敏意老师宽广的研究视野和严谨的治学风格深深地影响了我，让我受益终身。感谢过敏意教授的夫人李力老师。李力老师非常关心我们的生活状况，使我们在繁忙的科研工作之外能够感受到别样的温暖。

感谢美国匹兹堡大学的杨峻教授和张有弢教授。在我到匹兹堡大学交流的二十一个月时间里，两位教授对我的研究工作提供了大量的帮助。两位教授带我接触到了世界最先进的研究方法和研究内容。他们为我的算法设计、实验方案和论文撰写提供了细致的指导。他们的帮助使我的研究和论文写作能力有了大幅的进步。我还要感谢美国匹兹堡大学的 Rami Melhem 教授和 Bruce R. Childers 教授。他们对我的研究和论文提出了许多指导意见。感谢美国匹兹堡大学的同学和室友在我交流期间对我的帮助和支持。

感谢陈全老师对我研究和论文的帮助和支持。他严谨的科研精神和优秀的论文写作技巧让我受益匪浅。他一直是我学习的榜样。感谢郑龙、卢彦超两位师兄在我初入实验室时对我的指导和鼓励。他们带我进入了科学的研究的大门。如果没有他们的引领，我可能不会攻读博士学位。感谢 EPCC 实验室的各位老师以及师弟师妹们为我的实验提供了大量的支持。

感谢我的女友金思惠子女士对我的陪伴，支持和鼓励。她给了我很多生活上的帮助和精神上的支持。感谢我的父母对我一直以来的无私付出和支持。他们一直支持我跟随自己的兴趣学习下去。我的所有的成果以及每一点成功都来自于我的亲人的无私关怀和支持。

感谢 A1203391 班的所有同学们。感谢我的论文评审人。感谢他们为我的论文提出了中肯的修改意见。感谢系办及研究生院的老师为我的论文提交和评审提供的便利及帮助。感谢所有帮助和支持我的人。



## 攻读学位期间发表的学术论文

- [1] ZHENNING WANG, LONG ZHENG, QUAN CHEN, MINYI GUO. CAP: Co-Scheduling Based on Asymptotic Profiling in CPU+GPU Hybrid Systems[C]. Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores, ACM, 2013: 107-114
- [2] ZHENNING WANG, LONG ZHENG, QUAN CHEN, MINYI GUO. CPU+GPU Scheduling with Asymptotic Profiling[J]. Parallel Computing, 2014, 40(2): 107-115.
- [3] ZHENNING WANG, JUN YANG, RAMI MELHEM, BRUCE CHILDERS, YOUTAO ZHANG, MINYI GUO. Simultaneous Multikernel: Fine-Grained Sharing of GPUs[J]. IEEE Computer Architecture Letters, 2016, 15(2): 113-116.
- [4] ZHENNING WANG, JUN YANG, RAMI MELHEM, BRUCE CHILDERS, YOUTAO ZHANG, MINYI GUO. Simultaneous Multikernel GPU: Multi-Tasking Throughput Processors via Fine-Grained Sharing[C]. High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on. IEEE, 2016: 358-369.
- [5] ZHENNING WANG, JUN YANG, RAMI MELHEM, BRUCE CHILDERS, YOUTAO ZHANG, MINYI GUO. Quality of Service Support for Fine-Grained Sharing on GPUs[C]. Proceedings of the 44th International Symposium on Computer Architecture (ISCA'17). ACM, 2017.



## 攻读学位期间参与的项目

- [1] 973 项目 “城市大数据三元空间协同计算理论与方法” 子课题 “智慧城市服务平台与应用验证”
- [2] 自然基金项目 “众核平台的并行编程模型及其运行时支持技术的研究”
- [3] 自然基金项目 “网络-物理空间中物件的搜询”
- [4] 长江学者创新团队项目 “可扩展数据中心关键技术研究”