分类号 TP393 学号 14069001

UDC 密级 公开

工学博士学位论文

面向堆叠异构系统的应用透明知策略研究

博士生姓名 李晨

学科专业 电子科学与技术

研究方向 计算机体系结构

指导教师 郭阳 研究员

国防科技大学研究生院

二〇一九年四月

Research on Application-transparent Strategies for Stacked Heterogeneous System

Candidate: Li Chen

Supervisor: Researcher Guo Yang

A dissertation

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Engineering

in Electronic Science and Technology

Graduate School of National University of Defense Technology

Changsha, Hunan, P. R. China

April 23, 2019

独 创 性 声 明

本人声明所呈交的学位论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表和撰写过的研究成果，也不包含为获得国防科技大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文题目：面向堆叠异构系统的应用透明策略研究

学位论文作者签名：_____ 日期： 年 月 日

学位论文版权使用授权书

本人完全了解国防科技大学有关保留、使用学位论文的规定。本人授权国防科技大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档，允许论文被查阅和借阅；可以将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、汇编学位论文。

（保密学位论文在解密后适用本授权书。）

学位论文题目：面向堆叠异构系统的应用透明策略研究

学位论文作者签名：_____ 日期： 年 月 日

作者指导教师签名：_____ 日期： 年 月 日

目　录

表 目 录

图 目 录

摘　要

国防科学技术大学是一所直属中央军委的综合性大学。1984 年, 学校经国务院、中央军委和教育部批准首批成立研究生院，肩负着为全军培养高级科学和工程技术人才与指挥人才，培训高级领导干部，从事先进武器装备和国防关键技术研究的重要任务。国防科技大学是全国重点大学，也是全国首批进入国家“211 工程”建设并获中央专项经费支持的全国重点院校之一。学校前身是 1953 年创建于哈尔滨的中国人民解放军军事工程学院，简称“哈军工”。

关键词: **国防科学技术大学**; 211; **哈军工**

ABSTRACT

National University of Defense Technology is a comprehensive national key university based in Changsha, Hunan Province, China. It is under the dual supervision of the Ministry of National Defense and the Ministry of Education, designated for Project 211 and Project 985, the two national plans for facilitating the development of Chinese higher education.

NUDT was originally founded in 1953 as the Military Academy of Engineering in Harbin of Heilongjiang Province. In 1970 the Academy of Engineering moved southwards to Changsha and was renamed Changsha Institute of Technology.The Institute changed its name to National University of Defense Technology in 1978.

Key Words: NUDT; MND; ME

第一章　绪论

近几十年来，随着半导体技术的不断发展，计算机系统的性能成几何级增长。一方面，这是由于半导体工艺的不断进步，晶体管大小不断缩小，芯片能够集成的晶体管数目不断增大；另一方面，也是因为硬件系统结构、编译技术和算法的不断发展和进步。在这些性能增长中，晶体管大小的缩减起到了决定性作用 [1]。然而，从 2005 年开始，量子隧穿效应使得晶体管漏电现象开始出现。晶体管大小由于受到小晶体管静态功耗的影响，已经难以再按照登纳德缩放比例定律（Dennard Scaling） [2] 继续缩减，时钟频率难以继续提升。为了不断提升性能，这要求我们从更高效的体系结构上下功夫。多核处理器、配有专用硬件加速器的异构系统应运而生。

研究发现，专用硬件加速器能够大大提升处理器的能效比 [3]。无论是传统的 DSP、GPU，还是针对深度神经网络加速的谷歌 TPU [4]、寒武纪 AI 加速器，这些加速器与 CPU 组成的异构系统，这些异构系统体系结构通过将大量并行计算或专用计算任务从 CPU 发送到 GPU、DSP 或其他专用加速器，大大减少了 CPU 指令执行开销和程序员的负担，系统性能和能效均大大提高。这对于图像处理、高性能计算、以及深度学习训练都具有重要作用。在这之中，特别是 GPU 处理器，据 Hameed 等人的研究表明，一个 GPU 的运算处理性能通常不低于 10 个 CPU 核 [3]。虽然起初其出现的主要目标是解决 CPU 对图像渲染加速不足的问题，但由于其架构在处理并行计算的天然优势，使得 GPU 目前更多地被用于通用并行计算。如今无论是手机、平板电脑、笔记本，还是到数据中心和超级计算机，GPU 无处不在。

然而异构系统不断发展，应用规模和数据规模不断增长，又遇到了存储墙问题，访存开销越来越大。在这个背景之下，为了继续维持摩尔定律，提升集成电路性能的同时降低集成电路的开销，集成电路堆叠技术作为一种有效解决方案被提出并广泛应用。三维堆叠集成技术是一种新的集成电路工艺，其将多个硅片垂直堆叠并以三维封装的方式封装成一个芯片，是在工艺尺寸缩减受限的情况下提高系统性能的一种新的方式，从延伸摩尔定律和超越摩尔定律两方面实现芯片上晶体管密度和芯片性能的大幅度提升。其在提升带宽、降低延迟和提高能效方面有许多优势。不过，受限于良率、热效应、设计复杂度、设计测试成本以及 EDA 工具等方面的限制，三维堆叠技术目前主要用于一些如存储器等互连简单、单元排列重复的规则电路。相比于三维堆叠技术这种革命性的革新，基于硅中介层的 2.5 维堆叠技术则属于一种进化技术，避免了三维堆叠技术中的各种问题。2.5 维堆叠集成电路是将多个同构或异构的部件相邻地堆叠在硅中介层上，相邻部件之

间通过硅中介层进行通信。通过 2.5 维堆叠可以将处理器和三维堆叠存储器相邻地堆叠在硅中介层上，为处理器增加内存容量的同时大大提高内存带宽，已经广泛用于目前商业化的异构处理器中。

1.1 研究动机

高性能堆叠异构系统目前已经被引入数据中心，为云计算提供更加强大的计算和存储能力。云计算中多租户技术的应用使得多个用户能够共享高性能堆叠异构处理器，并同时执行和处理多个任务。无论是 IaaS、PaaS、还是 SaaS，均对多租户技术有着强烈的需求。这种多租户任务允许计算资源的共享和存储资源的相互隔离，同时需要保证多租户任务间的安全隔离性。虽然多租户技术保证了安全隔离性，但用户无法了解任务运行的具体情况，难以从程序员的角度对应用程序进行优化，因此，研究面向堆叠异构系统的应用透明策略显得尤为重要。

本文从堆叠异构系统出发，发现并解决堆叠异构系统的三大问题：

内存超额配置造成的性能骤降问题。 云提供商往往会给用户提供超过其硬件资源的存储资源，在用户不同时使用这些资源的情况下提高数据中心的资源利用率。然而随着用户应用规模和数据量的不断增大，内存超额配置越来越普遍。堆叠异构处理器，如 GPU 目前已经能够为用户提供内存超额配置的支持。但通过我们在真实 GPU 的实验发现，当内存超额配置时，GPU 出现了严重的性能下降，在一些情况下甚至发生宕机。

多任务抢占的高上下文切换开销问题。 为在多任务处理中快速响应一些优先级较高，对延迟要求较高的任务，我们必须支持多任务抢占机制进行快速切换。上下文切换是多任务抢占的一种重要方式。然而 GPU 等堆叠异构系统相对于 CPU，由于同时处理大量数据，其上下文大小相对较大。因此，GPU 上下文切换的开销远大于 CPU，传统的上下文切换机制占用的存储带宽将严重影响多任务抢占的性能。

堆叠互连网络结构的负载不均衡问题。 在 2.5 维堆叠异构系统中，传统的方式是在相邻处理器或存储器的四周边缘通过硅中介层进行互连。Enright Jerger 等人提出利用硅中介层大量的连线资源设计 2.5 维片上互连网络。通过上层网络进行一致性协议通信，通过下层网络进行访存通信。然而，我们的实验发现，由于不同类型报文的不均衡性，该方法会导致上下层网络的负载不均衡问题，严重影响整个系统的性能。

我们的研究发现，传统的通过程序员手工调试优化的技术都难以在支持多租户技术数据中心中高效使用。本文针对堆叠异构系统中的上述三大问题，研究对应用透明的硬件或驱动策略，使得堆叠异构系统能够在程序员不修改应用程序的前提之下为这些问题提供高效的解决方案。

1.2 研究内容和主要贡献

1.2.1 研究内容

本文面向堆叠异构系统，研究对应用透明，程序员不感知的优化策略解决上述三大问题，主要包括:

（1）一种内存超额配置管理策略框架（ETC）

现代分离式 GPU 支持统一内存技术和按需取页技术。这种在 CPU 和 GPU 内存中自动的数据拷贝管理大大降低了开发者的负担。但是，当应用程序的在线工作数据集超过 GPU 物理内存时，产生的额外数据移动会导致严重的性能损失。

我们提出了一种内存管理框架，采用了一系列对应用和程序员透明的新技术提升内存超额配置下的 GPU 性能。这些技术的主要思想包括掩藏页逐出延迟、降低内存抖动开销、以及增大有效的内存空间。页逐出延迟可以通过*主动页逐出技术*尽早为将要取进来的数据页腾出空间，掩盖延迟；内存抖动的开销可以通过内存感知的*并行度控制策略*，动态地在页缺失的时候将 GPU 的并行度降低，缓解内存抖动现象；*内存容量压缩技术*在不需要增大物理内存容量的前提下使得更大的在线工作数据集能够被内存容纳。我们发现没有任何一种技术对所有类型的应用程序都有效。因此，我们的 ETC 集成了*主动页逐出技术*、*并行度控制策略*和*内存容量压缩技术* 到一个管理框架，当内存超额配置时针对应用程序类型动态地选择这些策略的组合，对应用程序透明的提升 GPU 的性能。从这个角度出发，ETC 将应用程序划分为无数据共享的规则应用程序、数据共享的规则应用程序以及不规则的应用程序。

我们进行实验分别实现当前的基准结构、一种具有无限内存大小的理想结构以及我们的设计 ETC。ETC 能够几乎完全消除无数据共享的规则应用程序的内存超额配置开销，使之性能与具有无限内存空间的理想情况类似。我们还发现，相比于当前的基准策略，我们的 ETC 能够将数据共享的规则应用程序和不规则应用程序的内存超额配置开销大大降低。

（2）一种动态采用检查点备份技术的 GPU 主动抢占策略（PEP）

无论是空间上的多任务支持还是时间上的多任务支持，GPU 对多任务处理的需求都在不断增加。这要求 GPU 可以随时被抢占，在某一应用程序正在执行的过程中，中断执行并切换上下文到新的应用程序。不同于 CPU，GPU 由于其大量的上下文大小，上下文切换产生的开销非常大。研究人员已经做出了大量工作来降低 GPU 上的抢占开销。例如降低上下文的大小或将上下文切换和执行重叠，同时执行等。而所有之前的这些方法都是被动式的，意味着上下文切换都是在抢占请求到来之后才开始的。

本文提出了一种动态主动的机制来降低抢占的延迟，我们观察到 GPU 内核函数的执行无论是在 CUDA 还是 OpenCL 下都一定是在发射命令只有开始的。因此，抢占请求是可以在其实际到达 GPU 前预期到。我们研究了这一段延迟，并开发了一种预测机制提前进行状态备份。当抢占请求实际到达 GPU 后，我们只需要将相对于上一次状态备份的变化部分再做备份，非常类似于传统的检查点备份技术。我们的设计同时也可以根据 GPU 内核函数在运行过程中的特性动态选择排空执行技术或基于检查点的上下文备份技术。我们进行实验测试，PEP 设计可以有效的降低等待上下文切换产生的停滞延迟。更重要的是，通过我们这种细致的状态备份方法，相比于需要被完整地切换的上下文大小，我们也有效减少需要备份的上下文的大小。

（3）一种动态延迟感知的 2.5 维堆叠片上网络负载均衡策略（DLL）

由于三维堆叠技术依然面临着许多的挑战，当前 2.5 维堆叠技术具有更好的应用前景。通过硅中介层的应用，2.5 维堆叠技术可以提供为异构处理器提供更高带宽和更大容量的存储系统。为了满足 2.5 维堆叠芯片的存储系统通信要求，硅中介层上丰富的连线资源可以被利用来开发并实现一套新的网络。但是，2.5 维堆叠片上网络体系结构的性能受到两层网络之间严重的不均衡负载限制，难以发挥出应有的性能。

为了解决这个问题，我们在本文提出了一种动态延迟感知的负载均衡策略。我们的核心想法是通过最近几个报文的平均延迟来检测整个网络层的拥塞程度，根据收集到的数据在每个源节点来进行报文网络层的路由选择。我们利用了硅中介层上的充足的连线资源实现了一个延迟传播环网。这个延迟传播环网使得在目的节点收集到的报文延迟信息能够传输回源节点。我们采用这些信息达到了负载均衡的目标。

我们的实验与无负载均衡策略的基准设计、一种目的节点检测的策略和一种缓存感知的策略进行比较，我们的 DLL 策略均达到了不少的吞吐率提升，同时只产生非常小的开销。

1.2.2 主要贡献

本文系统深入地研究了应用透明策略以提升堆叠异构系统的性能，做出了许多系统开创性的工作，主要创新点如下：

（1）提出了一种内存超额配置的管理框架

内存超额配置虽然在当前 GPU 中得到了完全支持，但之前的工作没有考虑内存超额配置所带来的严重开销，且内存超额配置的优化工作都需要修改应用程序，难度较大且效果不一定好。本文提出了一种内存超额配置的管理框架，主要贡献包括：1. 据我们所知，这是第一个对 GPU 内存超额配之下的性能开销做深度

分析的工作。我们通过对应用程序访存 trace 的分析，找出了不同应用程序类型在 GPU 内存超额配置下出现严重性能损失的不同原因。2. 本研究提出了一种软硬件结合的对应用透明的解决方案，能够显著降低内存超额配之下的性能损失。该方法对程序员不感知，不需要任何应用程序代码的修改。3. 本研究开发了三种内存超额配置的优化策略。我们发现并没有任何一种单一方法能够对所有类型的应用程序都能见效。从这个角度出发，本研究的策略可以用根据访存特性，在线划分应用程序的类别，并为不同类憋的应用程序采用不同的策略组合进行性能优化。

该部分的研究成果发表在系统结构领域的顶级会议第 24 届 ACM 国际编程语言与操作系统的体系结构支持会议（The 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-19）上。

（2）提出了一种动态采用检查点备份技术的 GPU 主动抢占策略

GPU 中的多任务抢占开销非常大，这是由于相对于 CPU，GPU 的上下文较多，需要备份的上下文数据较大。传统的抢占方案都采用了被动方法，即抢占请求到来时才开始上下文切换。本文开创性的采用了主动抢占方法，提出了一种动态采用检查点备份技术的 GPU 主动抢占策略，主要贡献包括：1. 研究了 GPU 内核函数的额发射工程，观察到抢占请求可以被提前预测到。2. 本研究引入了一种主动的抢占技术来减少抢占内核函数等待上下文切换的时间。通过采用检查点备份技术，当时机抢占请求到来时，只有一小部分更新的上下文需要被存储。3. 本研究使用了一种相对简单的更新数据村粗技术来减少上下文大小，这可以减少不必要的上下文存储开销。4. 本研究开发了一种相对更加精确的线程块执行时间和上下文切换时间的估算方法，设计了实时动态选择算法来确定时机采用的抢占方法。我们可以分别完成长短内核函数的抢占，并使之达到最短延迟和最小开销。

该研究的部分成果发表在了计算机辅助设计领域顶级会议第 55 届设计自动化国际会议（The 55th Design Automation Conference, DAC-18）上，完整内容发表在了体系结构旗舰期刊 IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 上。

（3）提出了一种动态延迟感知的 2.5 维堆叠片上网络负载均衡策略

2.5 维堆叠片上网络作为一种全新的网络结构目前的研究还不多。由于其特有的网络通信特征，传统的多层网络结构以及三维堆叠片上网络方法难以直接应用到 2.5 维堆叠片上网络结构上。本研究发现之前的 2.5 维片上网络研究未考虑到均衡负载的问题。针对该问题，本文研究了一种动态延迟感知的 2.5 维堆叠片上网络负载均衡策略，主要贡献包括：1. 本研究评估并分析了 PARSEC 测试集的通信琉璃，发现 2.5 维堆叠片上网络的通信琉璃在两个网络层极不公平。2. 本研究分析了当前负载均衡策略的缺点。缓存感知的方法无法获取去全局网络的拥塞情

况；当前的延迟感知方法不能准确地检测全局拥塞状态。3. 本研究设计了一个多链路无阻塞环网，通过该环网，将目的节点收集的拥塞信息传输回源节点用于网络选择。

该部分的研究成果发表在了第 34 届国际计算机设计会议（The 34th IEEE International Conference on Computer Design, ICCD-16）上。

1.3 研究框架概述

本文研究了堆叠异构系统的三大问题，包括异构系统的内存墙问题、多任务调度的抢占问题、以及堆叠互连网络的负载均衡问题。本文提出了三种对应用透明的软硬件策略，能够有效解决堆叠异构系统的这三大问题，使其缓解内存超额配置带来的严重性能损失、降低堆叠异构系统多任务调度的高上下文切换延迟、以及均衡堆叠互连网络的负载等。这些应用透明策略均在模拟器中得到了有效验证。

1.4 论文组织结构

第二章　研究背景

2.1　堆叠系统

如今先进的三维堆叠芯片系统结构由于其在减少连线长度方面的天然优势，为减小未来芯片互连延迟和开销提供了非常有效的解决方案。三维堆叠存储器的出现，能够为异构处理器提供更高的访存带宽，解决存储墙问题。堆叠系统的这些优势使得我们有了许多为未来处理器体系结构提供创新设计的机会。

2.1.1　三维堆叠集成

三维堆叠封装技术属于最早的三维堆叠技术，是一种相对成熟的技术，已经在工业界得到了广泛应用。三维堆叠封装技术通过*系统内集成封装*（System-in-package，SiP）或*封装内封装*（Package-on-package，PoP）技术将多个芯片垂直堆叠在一个基板上封装在一起，或者将多个封装号的芯片堆叠起来。目前已经出现了许多成熟的采用三维封装技术的产品，包括在 iPhone6 中开始采用的 Apple A8 处理器，将一个包含封装的双核 CPU 和 4 核 GPU 的处理器和一个 1GB 大小的 LPDDR3 DRAM 封装的存储器采用 PoP 技术封装在一起。三维堆叠封装技术不要求在处理器结构和设计方法学上面做太大修改，因为这些技术的主要目标是为了节省空间，芯片之间的通信仍然通过片外信号完成，因此无论是连接性和延迟都没有任何优势。

不同于三维堆叠封装技术，三维堆叠集成技术则是一种新兴技术。三维堆叠集成技术将两或多层有源器件层，即 CMOS 晶体管层，在垂直方向上集成到一个芯片上。堆叠芯片的层间提供了大量的互连资源，因此这种革命性的系统结构创新要求在设计方法学上的改变。三维堆叠集成技术也可以分为两类：一类是*单晶堆叠方法*（Monolithic Approach），该方法在单一晶元上完成三维集成电路所有的设计制作流程，最后将其切成晶粒。该方法只需要一个衬底，无需对准、削薄、粘接等流程。另一类则是普通的堆叠方法，可以进一步被划分为 wafer-to-wafer、die-to-wafer、和 die-to-die等方法。这些方法都是在分别制造每一层的芯片，最后被组装粘接组成三维堆叠芯片。不同于复杂的*单晶堆叠方法*，不要求全新的设计制造技术，相对更加实际，也是三维集成电路技术的研究重点。

相比于传统的集成电路技术，三维堆叠集成技术具有非常多的优势，我们从线长缩短、高访存带宽、异构集成和低成本四个角度介绍三维堆叠集成技术的优势：

线长缩短。芯片的全局连线延迟并没有按照摩尔定律的速度不断缩短，如今不断增加的全局连线延迟已经成为阻碍性能持续增长的重要原因之一。三维堆叠集成技术能够有效克服全局连线延迟的瓶颈，为集成电路性能持续发展提供了解决方案。连线长度的缩减主要带来两方面巨大优势：延迟和功耗降低。延迟的降低是由于平均线长和关键路径的缩短，而总线长的缩短也自然使得功耗大大降低。

高存储带宽。如今，如何为具有大量计算单元的 GPU 等异构处理器及时提供足够的数据已成为限制性能提升的重大挑战，因此提供高访存带宽尤为重要。传统的片外存储单元由于受到 I/O 引脚的限制，难以提供足够的访存带宽。三维堆叠集成技术作为一种解决方案能为未来的微体系结构设计，特别是多核和众核处理器，解决互连瓶颈、消除“存储墙”问题。通过将存储器堆叠在处理器芯片上，利用 TSVs 带来的通信优势提供远高于二维结构的通信带宽。

异构集成。三维堆叠集成技术为未来的体系结构设计提供了新的机会，即能够在新的维度进行设计空间探索。特别是异构集成能力，让我们从全新的角度研究系统结构的设计。三维堆叠集成技术支持异构集成是由于其每一层都可以分开制造，不同的层也可以采用不同的工艺。甚至可以在处理器层上堆叠光设备层、非易失性存储层或者是相变存储层，以实现高效异构系统结构。这异构集成的实现可以为集成电路和芯片设计提供巨大的灵活性以满足性能、功耗和可靠性等要求。

低成本。随着集成电路规模不断增大，芯片的面积不断增大。但是由于缺陷密度相对恒定，较大的芯片晶粒大小意味着相对较低的良率。而将多个较小的晶粒堆叠成一个相对较大规模的处理器增获得较高的良率，即使三维堆叠集成由于额外的制造复杂度可能会降低一定的良率。另一方面，半导体集成电路的缩减也逐渐达到物理极限，继续缩减不但困难，成本也非常高。因此三位堆叠集成技术潜在地提供了一种相比于传统集成电路更低成本的解决方案。

虽然三维堆叠集成技术带来了巨大的性能优势和在系统结构设计的机会，但在其广泛应用到未来的计算机系统前仍然需要解决几个重要的挑战：

热问题。功耗和热问题在传统的二维集成电路设计中已经成为一个重要的问题。虽然三维堆叠集成技术相对于二维集成电路来说有许多的优势，然而将多个有源器件层堆叠起来大大增加了功耗密度，使得热问题进一步恶化，导致芯片温度升高。芯片温度的升高又会反过来影响电路性能，如互连延迟会由于晶体管温度的升高而延长、静态功耗与温度是指数级依赖关系、温度升高还会导致热逃逸问题，此外，温度的升高会降低可靠性问题。

设计工具和方法学。如果没有辅助设计工具和方法学的支持，三维堆叠集成技术不可能商用化。给定设计目标，高效的辅助设计工具和方法学可以帮助体系

结构和电路设计者权衡三维堆叠电路在性能、功耗和开销。相比于传统的二维集成电路，三维堆叠集成电路需要全新的布局规则，例如 TSV 布局、功耗计划等等。高效的辅助设计工具还能够帮助分析热问题，在布局布线设计中避免热点区域的产生。

测试问题。在三维堆叠集成设计中，不同的测试策略和集成方法都可以严重影响系统的性能、功耗和开销问题。三维堆叠集成电路技术目前没有广泛商用的另一个原因就是测试能力不足和缺乏面向三维堆叠集成技术的可测性设计技术。如果没有在设计阶段考虑好测试问题，高效的三维集成测试时不可能的。现在的三维堆叠集成测试上，一方面现在的探测技术不能够对所有的 TSV 进行探测；另一方面，测试过程很容易损坏被削薄的晶圆。

2.1.2 基于硅中介层的 2.5 维堆叠

三维堆叠集成技术可以将存储器直接堆叠在处理器晶圆之上。不同于三维堆叠集成技术，2.5 维堆叠策略的解决方案则是分别把存储器和处理器相邻地堆叠在硅中介层上，通过硅中介层的连线进行通信。基于硅中介层的 2.5 维堆叠技术虽然是一代进化技术，但其能够在存储带宽和容量、热问题以及制造成本等方面表现出相对于三维堆叠集成技术更多的优势。

存储容量和带宽

对于三维堆叠集成，其主要优势是在两层间的最大潜在带宽受其面积限制。若采用直径 50μs 的 TSV 用作上下层的通信，理论上一个 200cm^2 的芯片能够提供 800 万个 TSVs。相对地，对于 2.5 维堆叠集成，处理器和存储堆叠在硅中介层上的链接取决于处理器芯片的周长。假设硅中介层上的连线直径也是 50μs（全周长约 56cm），则同样 200cm^2 的芯片仅能提供 11000 个连接点。虽然三维堆叠和 2.5 维堆叠能提供的带宽有几个数量级的差异，但一个四通道的传统 DDR3 接口仅要求 960 个封装引脚。硅中介层提供的成千上万的连接点已经可以避免这个瓶颈。堆叠技术最重要的优势是将存储器和芯片封装在了一起，三维堆叠相对于 2.5 维堆叠更多的优势在这里实际并不明显。

而基于硅中介层的 2.5 维堆叠所能提供的存储容量则并不受限于处理器的面积，实际上能够堆叠在硅中介层上的堆叠存储的数目远远大于三维堆叠技术。这是因为三维堆叠仅能从垂直方向上进行堆叠，受到了当前技术的限制。而硅中介层上的 2.5 维堆叠则能够支持多块堆叠存储水平分布，能够提供更大的存储容量。当前的三维堆叠存储标准均为每块堆叠存储器提供了固定的带宽，因此总的存储带宽和容量取决于能够堆叠的存储器的数量。对于当前的三维堆叠存储标准 JEDEC HBM，提供 1024 位数据信号在 1Gbps 的平律师工作，每个堆叠存储能够提供 128GB/s。而若 4 块 HBM 堆叠在硅中介层上，则总共能提供 512GB/s 的带

宽。假设一个 HBM 的大小为 4GB，则基于硅中介层的 2.5 维堆叠能够提供 16GB 的存储容量，无论是带宽还是容量均优于三维堆叠技术。

热问题

基于硅中介层的 2.5 维堆叠相对于三维堆叠技术的另一个优势就是发热问题相对并不严重。因为一般热问题比较严重的是处理器层，在 2.5 维堆叠技术上，处理器一般以单层的形式堆叠在硅中介层上，没有额外的层会堆叠在处理器上。因此可以直接在处理器层上安装散热器。当多块堆叠存储单元堆叠在硅中介层上时，温度虽然不会线性升高，但保持一个相对较低的温度依然对于降低刷新率和提高可靠性有重要意义。

制造成本

不同的堆叠技术要求不同的制造步骤，因此制造开销也会不一样。基于硅中介层的 2.5 维堆叠技术相对于三维堆叠技术来说最明显的不同是 2.5 维堆叠技术需要增加额外的一个硅中介层的制造。由于硅中介层需要更大的面积来堆叠更多的存储和处理器单元，且需要被削薄来支持 TSV 链接到 C4 bumps。因此硅中介层往往采用较老的一代半导体技术来实现。

三维堆叠结构的成本包括需要重新设计支持 TSV 的连接，这都是需要工程师的努力、EDA 工具的支持、以及物理设计等。同时 TSV 的面积和其周围的留空也需要为此增大芯片的面积。而 2.5 维堆叠技术在处理器层不需要 TSV，因此芯片无需削薄和面积的增大。

虽然 2.5 维堆叠和 3 维堆叠解决方案之间还有许多可以比较的优缺点，但根据上述的主要问题，我们认为 2.5 维堆叠技术对于异构系统来说非常令人期待。堆叠异构系统采用 2.5 维堆叠技术可以获得更高的存储带宽和更大的存储容量，同时在热管理上也有更多的优势。因此，本文研究的堆叠异构系统也主要基于 2.5 维堆叠技术，研究其在内存墙问题、多任务切换问题和网络负载均衡上的问题，并提供解决方案。

2.2 GPU 架构与编程模型

计算机辅助绘图最早出现在二十世纪六十年代，如 Ivan Sutherland’s Sketch-pad [5] 从早期的计算机辅助绘图到电影动画的离线渲染和视频游戏的在线渲染，计算机图形处理不断发展。最早的显卡是从 1981 年的 IBM Monochrome 显示适配器（MDA）开始，但当时只支持文字。之后开始出现支持 2D 加速和 3D 加速，以及视频游戏的 3D 加速计算机辅助设计的图形显卡。早期的 3D 图像处理器如 NVIDIA 的 GeForce 3 还只能支持相对单一的功能。从 2001 年开始，NVIDIA 以顶点着色器 [6] 和像素着色器的形式将可编程性引入了 GPU，出现了 NVIDIA Geforce 3 图形处理器。研究人员很快学习到如何在早期的这些 GPU 中通过将矩

阵数据映射到纹理中以实现线性代数计算。这些学术工作将通用计算任务映射到 GPU 上处理，这样程序员就不需要完全了解图形学以使用 GPU。这些努力促使 GPU 制造商在支持图形计算的同事还能够直接支持通用计算。第一个实现通用计算能力的商业化 GPU 产品是 NVIDIA GeForce 8 系列。GeForce 8 系列引入了多项创新，解决了早期 GPU 的不足，包括从 Shader 核写入数据到任意存储地址、便笺缓存等。在 NVIDIA Fermi 架构中，GPU 出现了缓存以暂存读或写的数据到片内。随后的改进包括 AMD 的 Fusion 架构，它在同一芯片上集成了 CPU 和 GPU，以及动态并行性，可以从 GPU 本身启动线程。最近，NVIDIA 的 Volta 推出了 Tensor Cores 等功能，专门用于机器学习加速。

2.2.1 GPU 架构

当前的 GPU 系统，GPU 并不是能够独立工作的计算设备，而是与 CPU 一起合并到一块芯片或作为一个独立显卡连接到 CPU 组成一个系统工作。CPU 主要负责初始化 GPU 端的计算，然后将数据传输到 GPU 进行计算，并在 GPU 端计算完成后将数据结果传输回 CPU。在 CPU 和 GPU 之间进行这种分工的一个原因是计算的开始和结束通常需要访问输入 / 输出（I/O）设备。虽然正在不断努力开发直接在 GPU 上提供 I/O 服务的应用程序编程接口（API），但到目前为止这些 CPU 已经具有 I/O 的能力 [7][8]。这些 API 的功能在 CPU 和 GPU 之间提供了简单的通信接口，隐藏了复杂的管理过程。

如图所示的是包含 CPU 和 GPU 的典型系统的抽象图。左侧是典型的分立式 GPU 配置，包括连接 CPU 和 GPU 的总线（例如，PCIe 或 NVLink），广泛用于 NVIDIA 的 GPU 体系结构，如 NVIDIA 的 Volta GPU。右图是典型的集成 CPU 和 GPU 的逻辑图，如 AMD 的 Bristol Ridge APU 或移动端采用的 GPU。在这之中，包含分立式 GPU 的系统具有分别用于 CPU（通常称为系统内存）和 GPU（通常称为设备内存或显存）的独立 DRAM 存储空间。用于这些存储器的 DRAM 技术通常是不同的（用于 CPU 的 DDR 与用于 GPU 的 GDDR）。CPU 系统内存通常针对低延迟访问进行优化，而 GPU 的显存针对高吞吐量进行了优化。相比之下，具有集成 GPU 的系统具有共同的 DRAM 存储器空间，因此必须使用相同的存储器技术。由于集成的 CPU 和 GPU 经常在低功耗的移动设备上使用，因此共享 DRAM 存储器通常针对低功耗目标（例如，LPDDR）进行优化。

GPU 计算应用程序开始运行应用程序将分配和初始化一些数据结构。在 NVIDIA 和 AMD 的传统分立 GPU 上，GPU 计算应用程序的 CPU 部分通常为 CPU 和 GPU 中的数据结构分配空间应用程序的一部分必须协调数据从 CPU 内存到拷贝到 GPU 内存。最近新的分立式 GPU（如 NVIDIA Pascal/Volta 系列）支持

数据从 CPU 内存到 GPU 内存的自动传输。这可以通过利用 GPU 虚拟内存实现。NVIDIA 称之为“统一内存”（Unified Memory）。对于 CPU 和 GPU 都集成到同一芯片上并共享相同的内存的情况，程序员不需要控制 CPU 内存到 GPU 内存的拷贝。但是，由于 CPU 和 GPU 使用缓存，而其中一些缓存可能是私有的，因此可能存在 Cache 一致性问题，需要开发人员来解决。

在当前 GPU 系统中，CPU 在其运行的驱动程序的帮助下完成对应用在 GPU 计算的初始化。在启动 GPU 计算之前，在应用程序应明确在 GPU 上运行的代码。此代码通常称为内核函数（kernel）。同时 GPU 应用程序的 CPU 部分指定运行多少个线程、以及每个线程所需计算的数据的位置。运行的内核函数，线程数和数据位置通过 CPU 上运行的驱动程序传送到 GPU 硬件。驱动程序将转换信息并将数据和信息存储在 GPU 可查找的位置。然后驱动程序向 GPU 发出信号，表明应用程序的 GPU 部分可以运行计算。

现代 GPU 由许多计算核组成。NVIDIA 将这些计算核心称为流式多处理器（Stream Multiprocessor, SM），AMD 将其称为计算单元 (Compute Unit, CU)。每个 GPU 计算核执行与已经启动以在 GPU 上运行的内核函数相对应的单指令多线程（SIMT）程序。GPU 上的每个计算核心通常可以运行大约一千个线程。在单个核上执行的线程可以通过便笺缓存进行通信，并使用快速栅栏操作进行同步。每个计算核心通常还包含指令和数据高速缓存。它们充当带宽过滤器，以减少发送到较低级别的内存系统的流量。在第一级高速缓存中找不到数据时，核心上运行的大量线程可以互相切换执行以隐藏访问内存的延迟。

为了维持高计算吞吐量，必须平衡高计算吞吐量和高存储器带宽。这又需要存储器系统中的并行性。在 GPU 中，一般通过采用多个存储器通道来提供这种并行性。通常，每个存储器通道都将其与存储器分区中的上一级高速缓存的一部分相关联。GPU 计算核心和内存分区通过片上互连网络（如交叉开关）连接。当然，也有其他的组织结构类型。例如，超级计算中广泛使用的 GPU 英特尔至强 Phi 协处理器则直接将最后一级缓存配置给每个计算核心。

2.2.2 GPU 编程模型

GPU 计算应用程序是从 CPU 上开始执行。对于分立式 GPU，应用程序的 CPU 部分通常会分配内存以用于 GPU 上的计算，然后启动输入数据到从 CPU 内存到 GPU 内存的传输，最后在 GPU 上启动内核函数开始计算。对于集成 GPU，可以直接开始启动内核函数进行计算，由于 CPU 和 GPU 共享物理内存，无需进行数据传输。内核函数（Kernel）由数千个线程组成。每个线程执行相同的程序，但是可以运行不同控制流，这取决于计算的结果和运行过程中的判断跳转。下面我们使用 CUDA 编写的特定代码示例详细分析此流程。然而，Seo 等人观察到一

个现象，在 OpenCL 上针对一个体系结构（例如，GPU）仔细优化过的代码可能在另一个体系结构（例如，CPU）上执行效果非常不好 [9]。

图 **??**提供了用于 CPU 实现一个大家熟知的操作，即单精度标量值 A 乘以矢量值 X 加矢量值 Y 的的 C 代码，称为 SAXPY。SAXPY 是众所周知的基本线性代数软件库（BLAS）的一部分，可用于实现更高级别的矩阵运算，如高斯消元法。鉴于其简单性和实用性，它经常被用作教授计算机体系结构的示例。

```
1  void saxpy_serial(int n, float a, float *x, float *y)
2  {
3      for (int i = 0; i < n; ++i)
4          y[i] = a*x[i] + y[i];
5  }
6
7  main()
8  {
9      float *x, *y;
10     int n;
11     // 忽略初始化x和y的操作
12     saxpy_serial(n, 2.0, x, y); //调用串行计算SAXPY函数
13     // 忽略释放x和y的操作
14 }
```

图 2.1　C 代码版本的 SAXPY 计算

```
1  __global__ void saxpy(int n, float a, float *x, float *y)
2  {
3      int i = blockIdx.x*blockDim.x + threadIdx.x;
4      if(i<n)
5          y[i] = a*x[i] + y[i];
6  }
7  int main() {
8      float *h_x, *h_y;
9      int n;
10     //忽略给h_x和h_y在CPU分配空间和初始化的代码部分
11     float *d_x, *d_y;
12     int nblocks = (n + 255) / 256;
13     cudaMalloc( &d_x, n * sizeof(float) );
14     cudaMalloc( &d_y, n * sizeof(float) );
15     cudaMemcpy( d_x, h_x, n * sizeof(float),
           cudaMemcpyHostToDevice );
16     cudaMemcpy( d_y, h_y, n * sizeof(float),
           cudaMemcpyHostToDevice );
17     saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y); //调用并行计
       算SAXPY的GPU内核函数
18     cudaMemcpy( h_x, d_x, n * sizeof(float),
           cudaMemcpyDeviceToHost );
19     //忽略为h_x、h_y、d_x和d_y释放空间的代码部分
20 }
```

图 2.2　CUDA 代码版本的 SAXPY 计算

图 2.1中的示例演示了 SAXPY 的 C 代码形式。代码函数 `main()`开始执行执行。为了使例子专注于计算过程，我们暂时省略分配和初始化数组 x 和 y 的细节。接下来，调用函数 `saxpy_serial`。此函数将参数 n 中向量 x 和 y 中的元素数量，参数 a 中的标量值以及用于表示向量 x 和 y 的数组的指针作为输入参数。该函数迭代数组 x 和 y 的每个元素。在每次迭代中，第 4 行的代码使用循环变量 i 读取值 x[i] 和 y[i]，将 x[i] 乘以 a 然后加 y[i]，然后用结果更新到 x[i]。

图 2.2则提供了相应的 SAXPY 的 CUDA 版本，可以在 CPU 和 GPU 之间分别执行相应部分的代码。与传统的 C 或 C++ 应用程序类似，图 2.2中的代码通过在 CPU 上运行 `main()`函数开始执行。我们将重点放在 GPU 部分执行的代码。在 GPU 上执行的线程是由函数指定的计算内核函数（kernel）的一部分。在 SAXPY 的 CUDA 版本中，如图 2.2所示，第 1 行的 CUDA 关键字 `__global__`表示内核函数 saxpy 将在 GPU 上运行。在该示例中，我们已经并行化了图 2.1中 CPU 示例的 `for`循环。具体来说，图 2.1中原始 CPU 专用 C 代码中第 4 行 `for`循环的每次迭代都被转换为运行图 2.1中第 3-5 行代码的每个独立线程。

在我们的示例中，CPU 在第 17 行调用 CUDA 的内核函数配置语法的启动 GPU 上的计算。内核函数配置语法看起来很像 C 中的函数调用，其中一些附加信息指定了三角括号 («< »>) 之间包含的线程数。组成内核函数的线程被组织成一个层次结构，该层次结构由包含网格（grid）、线程块和 warp 三级组成。warp 一般由 32 个或 64 个线程组成，多个 warp 组成一个线程块，多个线程块组成一个网格。在 CUDA 编程模型中，各个线程执行其操作数是标量值（例如，32 位浮点）的指令。为了提高效率，典型的 GPU 硬件同步执行若干个线程组。这些组被 NVIDIA 称为 warp，AMD 称为 wavefronts。NVIDIA warps 由 32 个线程组成，而 AMD wavefronts 由 64 个线程组成。Warp 被分组为一个更大的单元，称为协作线程阵列（CTA）或 NVIDIA 的线程块。第 17 行表示内核函数应该启动由`nblocks`个线程块组成的单个网格，其中每个线程块包含 256 个线程。CPU 代码传递给内核配置表达式的参数将被分发到 GPU 上正在运行的线程的每个实例。

今天的许多移动设备片上系统将 CPU 和 GPU 集成到单个芯片中，就像今天的笔记本电脑和台式电脑上的处理器一样。然而，传统 GPU 具有自己的 DRAM 存储器，并且今天对于用于机器学习的数据中心内的 GPU 而言仍然是这样。我们注意到 NVIDIA 推出了统一内存，它可以从 CPU 内存透明地更新 GPU 内存，从 GPU 内存中透明地更新 CPU 内存。在启用统一内存的系统中，驱动和硬件负责数据的传输拷贝，而无需程序员手动操作。这种方法极大的减轻了程序员的负担，我们的内存超额配置管理框架也是基于统一内存的概念。但在图 2.2这个例子中，我们为了简单理解，采用了传统的程序员控制内存拷贝的形式。

遵循许多 NVIDIA CUDA 示例中的风格，我们在为 CPU 内存中分配空间时命名指针变量使用前缀 `h_`，在 GPU 内存中分配内存的指针使用 `d_`。在第 13 行，CPU 调用 CUDA 库函数 `cudaMalloc`。此函数调用 GPU 驱动程序并要求它在 GPU 上分配内存以供程序使用。对 `cudaMalloc`的调用将 `d_x`设置为指向 GPU 内存区域，该区域包含足够的空间来容纳 n 个 32 位浮点值。在第 15 行，CPU 调用 CUDA 库函数 `cudaMemcpy`。此函数调用 GPU 驱动程序并要求它从 `h_x`指向 CPU 内存的位置复制数组的内容到由 `d_x`指向的 GPU 内存中的位置。

对于 GPU 上线程的执行，并行编程中采用的通用策略是为每个线程分配一部分数据。为了实现这种策略，GPU 上的每个线程都可以在线程块网格中查找自己的 id。在 CUDA 中执行此操作的机制使用网格，线程块和线程标识符来明确 id。在 CUDA 中，网格和线程块具有 x，y 和 z 维度。在执行时，每个线程在网格和线程块内具有固定的，唯一的非负整数 x，y 和 z 坐标组合，这是每个线程块在网格中的 x，y 和 z 坐标。类似地，每个线程在线程块内具有 x，y 和 z 坐标。这些坐标的范围由内核配置语句设置（第 17 行）。在我们的示例中，未指定 y 和 z 维度，因此所有线程的 y 和 z 线程块和线程坐标的值都为零。在第 3 行，`threadIdx.x`的值标识其线程块和 `blockIdx.x`中线程的 x 坐标。

在我们的示例中，未指定 y 和 z 维度，因此所有线程的 y 和 z 线程块和线程坐标的值都为零。在第 3 行，`threadIdx.x`的值标识其线程块内线程的 x 坐标；`blockIdx.x`指示其网格中线程块的 x 坐标；值 `blockDim.x`表示 x 维度中的最大线程数。在我们的示例中，`blockDim.x`在第 17 行被置为 256。表达式 `blockIdx.x * blockDim.x + threadIdx.x`用于计算在访问数组 x 和 y 时使用的偏移量 i 。正如我们将看到的，使用索引 i，我们为每个线程分配了 x 和 y 的唯一元素。

在很大程度上，编译器和硬件的组合使程序员无需了解 warp 中线程执行的锁步特性。编译器和硬件使得 warp 中的每个线程能够独立执行。在图 2.2中的第 4 行，我们将索引 i 的值与 n（数组 x 和 y 的大小，即总线程数）进行比较。i 小于 n 的线程执行第 5 行。图 2.2中的第 5 行执行图 2.1中原始循环的一次迭代。在网格中的所有线程完成后，内核函数返回，在第 17 行之后继续由 CPU 执行。在第 18 行，CPU 调用 GPU 驱动程序将 `d_y`指向的数组从 GPU 内存复制回 CPU 内存。

下面我们介绍一些 SAXPY 示例未说明的其他细节。一个线程块中的线程可以通过每个流多核处理器的便笺存储器相互高效通信。这个便笺存储器被 NVIDIA 称为共享内存（share memory）。每个流多核处理器包含一个共享内存。共享内存中的空间在该 SM 上运行的所有线程块之间划分。每个线程块拥有一块独立的空间。AMD 的新一代 GPU 架构 GCN 包括一个类似的内存，AMD 称之为

本地数据存储（Local Data Store, LDS）。这些内存很小，每个有 16-64KB，并作为一个特殊的内存空间给程序员控制。程序员在源代码中使用特殊关键字将内存分配到便笺存储器中（例如，CUDA 中的“__shared__”）。便笺存储器可以被理解为软件控制的高速缓存。虽然 GPU 还包含传统的缓存，但通过这些缓存访问数据可能导致频繁的缓存未命中。当程序员可以识别经常重复使用的数据时，应用程序可以预测并使用便笺存储器与 NVIDIA 的 GPU 不同，AMD 的 GCN GPU 还包括 GPU 上所有内核共享的全局数据存储器（Global Data Store, GDS）。便笺存储器用于图形应用程序，以在不同的计算核心之间传递结果。例如，LDS 用于在 GCN 架构的 GPU 中传递顶点和像素着色器之间的参数值。线程块中的线程可以使用硬件支持的栅栏指令有效地同步。不同线程块中的线程可以通过所有线程都可访问的全局内存进行通信。访问全局内存在时间和功耗方面都比访问共享内存的开销要大很多。

2.3 异构系统的架构与策略优化相关研究

2.3.1 GPU 异构系统相关研究

异构系统中的并发管理。Kayiran 等人提出了一种并发限制方案来限制 GPU 多线程的并行度，减少支持多任务的 CPU-GPU 异构系统中的内存和网络竞争。在异构系统上，来自 GPU 的干扰可导致同时执行的 CPU 应用程序出现显著的性能下降。他们提出的线程级并行（TLP）限制方案通过观察共享 CPU-GPU 内存控制器和互连网络中的拥塞度量，以估计应在每个 GPU 内核上主动调度的 GPU warp 的数量。他们提出了两种方案，一种侧重于仅提高 CPU 性能；另一种方案旨在通过平衡由于受限制的多线程和 CPU 干扰引起的 GPU 性能下降来优化整体系统吞吐量（CPU 和 GPU）。作者评估了 warp 调度对异构架构的性能影响，GPU 核心数目与 CPU 核心数目比为 2：1，NVIDIA GPU SM 大约是相同的工艺技术下使用现代无序执行的 Intel 芯片面积的一半。为了使资源利用率最大化，基准配置是让 CPU 和 GPU 完全共享网络带宽和内存控制器。使用这种方案，作者认为限制 GPU TLP 可以对 GPU 性能产生正面和负面影响，但绝不会损害 CPU 性能。

为了提高 CPU 性能，作者介绍了一种以 CPU 为中心的并发管理技术，该技术可监控全局内存控制器中的停顿执行。此技术分别统计由于内存控制器输入队列已满而停止的内存请求数，以及由于从内存管理器到核心的返回网络已满而停止的内存请求数。每个内存控制器上实时监控这些指标，并在集中式单元中进行聚合，该单元将信息发送到 GPU 的计算核心。启发式方案为这些指标设置了高阈值和低阈值。如果两个请求停顿计数的总和低（基于阈值），则增加在 GPU 上

主动调度的 warp 的数量。如果两个值的总和很高，则减少同时运行的 warp 的数量，因为由于 GPU 内存流量较少，CPU 性能会提高。

为了解决 CPU 为中心的技术限制 warp 并行度对 GPU 性能的影响，作者提出了一种以最大化整体系统吞吐量为目标的更均衡的技术。这种均衡的技术在每一个并行工作的间隔（1024 个周期）监视 GPU 在并发无法发出指令的时钟周期数。在当前的限制策略下 GPU 的停顿执行时间将被记录在每个 GPU 核心中，用于决定应该增加还是将降低并行度。这种均衡的技术分两个阶段调节 GPU 的 TLP。在第一阶段，其操作与以 CPU 为中心的解决方案相同，其中不考虑 GPU 停滞并且仅基于存储器争用来限制 GPU TLP。在第二阶段如果它预测继续限制并行度会损害 GPU 性能（一旦 GPU TLP 限制开始导致 GPU 性能下降，因为 GPU 的延迟容忍度已降低），系统会停止限制 GPU 并发性。该方法通过在目标多线程并行级别上查找 GPU 的平均停顿时间的变化进行预测。如果在目标多线程并行级别上观察到的 GPU 停顿时间与当前多线程级别之间的差异超过阈值 k 时，则 TLP 并行级别不会降低。该 k 值可以由用户设置，并且可用于指定 GPU 性能的优先级。

异构系统一致性。Power 等人提出了一种硬件机制，以有效地支持 CPU 和 GPU 集成系统上的高速缓存一致性。他们发现随着 GPU 产生的内存流量增加，目录带宽成为重要瓶颈。它们采用粗粒度区域一致性来减少传统的基于缓存块的一致性目录中导致的过多目录流量。一旦获得了对粗粒度区域的许可，大多数请求将不必访问该目录，并且可以将一致性报文流量发送到可直接访问的总线而不是较低带宽的一致性互连网络。

针对 CPU-GPU 架构的异构 TLP 感知缓存管理。Lee 和 Kim 评估了在异构环境中管理 CPU 核心和 GPU 核心之间的共享最后一级缓存（LLC）的效果。他们证明，虽然缓存命中率是 CPU 应用程序的关键性能指标，但许多 GPU 应用程序对缓存命中率不敏感，因为内存延迟可以通过线程级并行来隐藏。为了确定 GPU 应用程序是否对缓存敏感，他们开发了一种针对每个 GPU 核的性能抽样技术，其中一些 GPU 核绕过共享 LLC，一些 GPU 核使用 LLC。根据这些内核的相对性能，他们可以为其余 GPU 内核设置策略，绕过或直接采用共享最后一级缓存，如果性能不敏感则绕过缓存，否则使用缓存。

其次，他们观察到之前以 CPU 为中心的缓存管理有利于更频繁访问的 GPU 核心。实验显示 GPU 核在最后一级缓存上产生的流量相比于 CPU 增大了五到十倍。这使得缓存容量更多地被 GPU 使用，因而降低了 CPU 应用程序的性能。他们建议扩展以前提出的基于效用的缓存分区的工作，采用最后一级缓存的访问次数比来调整 CPU 和 GPU 之间访问幅度和延迟敏感性的差异。

2.3.2 线程调度相关研究

现代 GPU 与 CPU 的根本区别在于它们依赖于大规模并行性。与具体的程序语言无关（例如，使用 OpenCL，CUDA，OpenACC 等），没有广泛的软件定义并行性的工作负载不适合 GPU 加速。GPU 使用多种机制来聚合和调度所有这些线程。GPU 上的线程主要有三种方式可以进行调度和组织。

将线程分配给 warp。由于 GPU 使用 SIMD 单元来执行由 MIMD 编程模型定义的线程，因此必须将线程融合在一起以 warp 的形式同步执行。在本文中研究的基准 GPU 架构中，具有连续线程 ID 的线程静态融合在一起以形成 warp。除了静态融合方式，研究人员提出了动态 warp 合成（Dynamic Warp Formation，DWF），即将执行相同指令的这些分散线程重新排列到新的动态 warp。在不同的分支处，DWF 可以通过将分散在多个分支 warp 中的线程重组使得每个 warp 的分支减少提高应用程序的整体 SIMD 效率。通过这种方式，DWF 可以在 SIMD 硬件上获得 MIMD 硬件的大部分优势。但是，DWF 要求 warp 在短时间内具有相同的发散分支。这种与时序相关的 DWF 特性使其对 warp 调度策略非常敏感。

将线程块动态分配到计算核。与 CPU 中线程分别一次次地分配给硬件线程不同，在 GPU 中，工作被批量分配给 GPU 核心。该工作单元由线程块形式的多个 warp 组成。在我们的基准 GPU 中，线程块以循环顺序分配给核心。核心资源（如 warp 总数、寄存器文件大小和共享内存大小）以线程块的粒度分配。由于与每个线程块相关联的大量状态，，默认情况下线程块之间不会进行抢占操作，开销过大。线程块中的线程在将资源分配给另一个线程块之前运行完成。然而，为了更好地支持多任务处理，抢占必不可少。本文在第 XXX 章研究如何降低抢占开销。

Kayiran 等人提出限制分配给每个核心的线程块的数量，以减少由线程超额配置引起的内存系统中的竞争。他们开发了一种监控 GPU 核心空闲时间和访存延迟周期的算法。该算法首先为每个 GPU 核心分配其最大线程块的一半数量的线程块，然后监视空闲时间和访存延迟时间。如果 GPU 核心主要等待时间是访存上，则不再分配线程块，并且可能暂停现有线程块并阻止它们发出指令。该技术实现了粗粒度并行性控制机制，使较少的线程块同时处于运行状态，也会限制内存系统干扰并提高整体应用程序性能。

时钟周期驱动的调度决策。在将线程块分配给 GPU 核心之后，一系列细粒度硬件调度程序在每个周期决定哪组 warp 取指令，哪些 warp 发出指令以执行，以及何时为每个流水线中的指令读取 / 写入操作数。传统的 warp 调度方式包括 Loose Round-Robin 调度和 Greedy-Then-Oldest 调度。

Gebhart 等人介绍了使用两级 warp 调度策略来提高能效。他们的两级调度程序将 GPU 核心中的 warp 划分为两个池：一个活动的 warp 池，用于在下一个周期中进行调度，另一个是非活动的 warp 池。每当遇到编译器识别到全局或纹理内存依赖关系非活动 warp 池和活动 warp 池互相替换。每个周期从较小的 warp 池中选择 warp 调度从而减小 warp 选择逻辑的大小和功耗开销。

Narasiman 等人提出的两级调度程序侧重于通过允许线程组在不同时间达到相同的长延迟操作来提高性能。这有助于维护一段时间内的缓存和行缓冲区的局部性。然后，系统可以通过在获取组之间切换来隐藏长延迟操作。

针对硬件线程调度对 GPU 中缓存管理的影响。Rogers 等人提出了提出了缓存意识 warp 调度策略（Cache-Conscious Wavefront Scheduling，CCWS）。这是一种自适应硬件机制，它利用一种新颖的 warp 内部局部性原理检测来捕获由于过度竞争缓存容量而损失的局部性优势。不同于那些改进缓存替换策略的方法，CCWS 优化访存模式以避免共享 L1 缓存的抖动。实验证明 CCWS 优于缓存替换策略的优化。

Jog 等人在 GPU 上探索预取感知的 warp 调度策略。他们的调度策略基于两级调度机制，但是从非连续 warp 形成取指组。该策略增加了 DRAM 中的 bank 级别的并行度，因为预取不会连续访问一个 DRAM bank。他们进一步扩展了这个想法，以根据 warp 组分配操作预取。通过为其他组中的 warp 预取数据，它们可以改善行缓冲区的局部性优势并在预取请求和需求数据之间提供间隔。

Jog 等人还提出了一种线程块感知的 warp 调度策略。在两级调度程序的基础上构建基于有选择地组合线程块的取指组。该方法利用几个基于线程块的属性来提高性能。该方法采用限制优先级技术，限制每个 GPU 核心中同时运行 warp 的数量，类似于其他并行度控制的调度。结合并行度限制方法，他们利用不同核心上的线程块之间数据页局部性原理。在只有局部性感知的线程块调度策略下，连续的线程块通常会同时访问同一个 DRAM bank，因此降低了 bank 级并行性。该方法将此与预取机制相结合，以改善 DRAM 行的局部性。

多个内核函数级别的调度。线程块级的调度和时钟周期驱动的调度决定可以是在一个内核函数中发生的，也可能是当多个内核函数同时在一个 GPU 钟运行时发生的。传统的思路是一个 GPU 在同一时间只能有一个 kernel 在运行。随着 NVIDIA 推出 Stream 和 HyperQ 调度策略后，多个内核函数的并行执行成为可能。这在某些方面也 CPU 的多任务支持有些类似。

Park 等人针对 GPU 上抢占式多任务处理的挑战，采用了更为宽松的幂等性定义，直接丢弃线程块的计算。更宽松的幂等性定义涉及检测执行是否从线程开始执行至今是幂等的。他们的提出的 Chimera 动态地选择了三种方法来实现每个线程块的上下文切换：包括完整的上下文保存 / 存储; 等待线程块排空执行剩余指

令; 如果由于幂等性，可以考虑安全地从头开始重新启动线程块，只需停止线程块而无需保存任何上下文。每种上下文切换技术在切换延迟和对系统吞吐量的影响之间提供不同的权衡。为了实现 Chimera，他们提出的算法算法估计了当前正在运行的线程块的吞吐率和开销，可以在满足用户指定的上下文切换延迟目标的同时对系统吞吐量的影响最小。

异构系统的软件优化 Kim 和 Batten 提出在 GPU 中为每个 SIMT 核心添加细粒度硬件工作清单。他们利用不规则 GPGPU 程序通常在使用数据驱动方法在软件实现时表现最佳的，他们动态生成和平衡线程，而不是拓扑方法。拓扑方法启动固定数量的线程，但通常许多这些线程没有做任何有用的工作。数据驱动方法能够有效提高工作效率和负载平衡，但如果没有配合软件优化，可能会遇到性能不佳的问题。这个工作提出了一个片上硬件工作清单，支持在内核之间进行负载均衡。它们使用线程等待机制并以间隔为基础重新平衡线程生成的任务。他们评估了这些硬件机制在 lonestar GPU 基准测试集中不规则应用程序的各种实现，这些应用程序分别利用了拓扑和数据驱动的方法。核内硬件工作清单的方法解决了数据驱动软件工作列表的两个主要问题：（1）线程在内存系统中的竞争;（2）基于线程 ID 静态分配工作导致的负载不均衡。软件的方法不依赖于静态任务分配，不会造成内存的竞争。硬件工作清单分布在多个结构中，从而减少了争用。它通过在线程变为空闲之前动态地重新生成任务来改善负载平衡。他们向指令集中添加了特殊指令，用于推送和拉出硬件队列。核心中的每个通道都分配有一个小型单端口 SRAM，用作存储器，用于存储由给定通道使用和生成的工作 ID。

Wang 和 Yalamanchili 等人研究了了在 NVIDIA Kepler GPU 上使用 CUDA Dynamic Parallelism 的开销，并发现这些开销可能很大。具体而言，他们确定了几个限制他们研究的应用程序的效率的关键问题。首先，应用程序使用了大量设备启动的内核函数。其次，每个内核通常只有 40 个线程（比一个 warp 略多)。第三，虽然在每个动态内核函数中执行的代码类似，但启动配置不同导致内核配置信息的大量存储开销。第四，为了实现并发，设备启动的内核被放置在单独的流中，以利用 Kepler 支持的 32 个并行硬件队列（Hyper-Q)。他们发现这些因素导致利用率非常低。

Wang 等人之后提出了动态线程块启动策略，他们修改了 CUDA 编程模型使得从设备启动的内核函数可以共享硬件队列的资源。这可以获取更高的并行度和更好的资源利用率。他们的策略的关键是可以动态的组合一起启动的内核函数。这是通过管理一个线程块链表来维护，并需要修改硬件。实验评估通过修改 GPGPU-sim 完成，实验表明这个方法相比于基准方法提升了 1.4 倍的性能，相比于高度优化的 CUDA 方案也提升了 1.2 倍的性能。Wang 等人还探索了线程块在

不同核心被动态启动的影响。他们发现子线程块和和父线程块在一个核心执行相比于简单的轮转分配执行性能提高了 27%。

第三章　一种 GPU 内存超额配置的管理框架

3.1　研究背景

随着面向高性能计算的应用程序对密集型计算需求和程序员对高可编程性需求的不断增加，如今图像处理器已经成为高性能计算领域首选的计算平台。然而，想要使得应用程序发挥出最大的性能，依然要求程序员手动调整代码来适应 GPU 的系统硬件结构并满足内存容量的要求。随着 GPGPU 应用程序的工作数据集不断增大，有限地 GPU 内存容量已经成为影响硬件设计和程序性能的首要瓶颈。

如今内存虚拟化技术支持不断演化进步，GPGPU 程序可以通过虚拟化技术更加容易地扩展在线工作数据集（Working Set），使之可以在超出 GPU 内存物理容量的情况下工作。现代 GPU 配备了统一内存空间和实时请求取页功能，这些新的特性使得开发者不再需要手动地管理数据在 CPU 内存和 GPU 内存之间移动。但是当一个 GPU 应用程序的工作数据集超出了内存的物理容量时（即内存超额配置时），旧数据必须被逐出，为新数据腾出空间。我们在一个真实 GPU 系统的测试显示，当内存超额配置时，真实 GPGPU 应用程序会出现巨大的性能损失，有时甚至发生宕机。

通过程序员的软件优化，能够缓解一部分内存超额配置导致的性能损失。例如，程序员可以将只读数据从 CPU 内存端复制到 GPU 内存而不是简单的搬运。当应用程序需要更多空间存放新数据时，可以直接丢弃这些暂时不用的只读数据，而无需通过传统的逐出方法，写回到 CPU 的内存。程序员可以将预取请求和逐出请求同时执行，新请求的数据无需等待内存逐出操作，有效地减少了延迟。但是，采用软件优化方法来降低内存超额配置导致的性能损失有几个明显缺陷。首先，软件方法要求程序员能够直接分清数据是否为只读数据；其次，程序员必须理解并利用好成千上万条线程的局部性原理，手动地将不同地数据页映射到 CPU 或 GPU 的内存；最后，程序员需要手动地在 CPU 和 GPU 内存之间搬运数据。这些缺陷在云计算环境里会变得更加明显，因为共享一个 GPU 的虚拟机用户无法知道其他用户的应用程序的在线工作数据集大小，即无法预测超额配置的程度，难以对内存超额配置情况进行软件优化。因此，设计一种能够降低内存超额配置开销并且对应用透明的机制的需求非常迫切。

我们观察到利用当前 GPGPU 应用程序的两个关键特性可以更好地管理好超额配置的内存。首先，内存超额配置带来的性能损失大小取决于不同应用程序的不同访存特性。根据 GPU 内存的页访问是否具有可预测性，我们大致将应用程序分为规则应用程序和非规则应用程序；第二，内存超额配置开销的来源与程序类

型相关。内存抖动（Thrashing），即重复地在 CPU 和 GPU 内存之间来回搬运数据页，是非规则程序的内存超额配置开销的最主要来源。而较高的页逐出延迟则是规则程序在内存超额配置情况下的主要开销来源。我们还发现同一个应用程序不同的 GPU 内核函数若存在数据共享，则会进一步影响 GPU 内存超额配置的性能。

基于这些特性，我们提出了一种叫 ETC 的内存超额配置管理框架。该框架以一种对应用程序透明的方式来降低内存超额配置开销。ETC 第一步是高效自动地将应用程序划分为三类，包括数据不共享的规则应用程序、数据共享的规则应用程序以及非规则的应用程序。ETC 之后会基于应用程序的类型选择最有效的策略组合来缓解内存超额配置导致的性能损失。ETC 集成了多个模块，它们协同工作可以有效掩藏或减少内存超额配置带来的开销，主要包括（1）一个应用划分器，能够通过测量的访存合并系数来检测并判断应用程序的类型；（2）一种策略引擎，能够基于应用程序类型选择并采用最合适的策略；（3）一种主动页逐出技术，针对规则应用程序，提前主动逐出无用的数据页，为之后要请求的数据页腾出物理空间；（4）一种内存感知的并行控制策略，针对非规则应用程序，通过降低线程级并行来减少有效在线工作数据集；（5）一种内存压缩引擎，能够为应用程序有效提高可用存储空间。

3.2 相关工作

在我们知晓的范围内，本章的工作是第一个提出应用透明的软硬件结合解决方案，针对不同应用程序的类别采用最有效的技术组合来解决内存超额配置的问题。我们调查了之前的工作，包括（1）提供 GPU 上的统一虚拟内存支持相关工作；（2）降低内存超额配置开销的相关工作；（3）实现良好的线程级并行度的相关工作；（4）增加有效内存容量的相关工作。

GPU 虚拟内存。CPU 上的虚实地址转换的开销已经被广泛和深入地研究。而对于 GPU，Pichai 等人和 Power 等人探索了内存管理控制器的设计来提升虚实地址转换的吞吐率，该方法是基于 GPU 的内存访存特性设计的。Cong 等人提出了针对主机 CPU 和从设备加速器之间的统一虚拟地址空间的 TLB 设计。MASK 是一个 TLB 感知的 GPU 存储层次设计，该方法通过优先内存元数据相关的访问(如页表查询）来加速 TLB 的命中率。Shin 等人提出了一种单指令多线程感知的机制来提高非规则 GPU 应用程序的虚实地址转换性能。

按需取页。传统的 GPGPU 内存占用受到物理内存的容量限制，因为所有的数据都要在内核函数开始执行之前从 CPU 拷贝到 GPU。当前 GPU 能够自动进行 GPU 的内存管理：数据页可以按照需要从 CPU 内存拷贝到 GPU 内存，内核函数的执行与数据的迁移是可以重叠的。这种自动的内存管理方法能够大大降低程序员的负担，并支持应用程序运行更大的数据集。Zhang 等人探索了数据移动的开

销，提出了程序员操作的内存管理来隐藏开销。它们的工作和我们的并不冲突，我们采用该方法作为基准方法在本章所有的配置中，包括 ETC。

GPU 内存超额配置。GPUswap 支持 GPU 内存超额配置，该方法通过重新定位 GPU 应用程序的数据到 CPU 内存，并保持该数据仍旧可以被 GPU 访问。GPUswap 提供了基本的内存超额配置支持，但并不能降低内存超额配置的开销。VAST runtime 基于有效的 GPU 物理内存划分数据并行的应用程序以满足内存容量的要求，但该方法要求程序员进行代码的转换。BW-AWARE 页替换策略利用了异构存储系统的特性和注释方法来指导数据的放置。该方法针对一个全局可以访问的异构内存系统。我们的工作则是采用应用透明的方法降低内存超额配置的开销。

GPU 线程级并行管理。之前的设计工作通过控制 GPU 核心的并行度来实现高的线程级并行和高性能。Rogers 等人提出了一种自适应的硬件策略来控制线程级并行以避免 L1 缓存的内存抖动。Kayiran 等人提出了一种动态的线程块调度机制来调节核心级别的线程级并行，该方法可以降低内存资源的冲突。Mascar 检测到内存饱和并在 warp 之间优先存储访问。Wang 等人提出了一种基于特征的线程级并行管理方法，来调节多应用程序并行的线程级并行。我们的工作降低了内存超额配置下的有效在线工作数据集，这是之前的工作所没有做到的。

GPU 中的内存压缩技术。许多之前的工作都研究了 GPU 内存和缓存的压缩技术。这些工作取得不错的性能均是因为节省了片内或片外的贷款。我们证明了内存容量压缩技术只在特定的场景下能获得性能的提升，并开发了一种机制来决定何时使用内存容量压缩技术。

3.3 研究背景

3.3.1 GPU 工作模型

GPU 通过单指令多线程模型（SIMT）实现了超高的吞吐率。在每个时钟周期，一个 GPU 核心（有时又被成为 SM）执行一组线程，该组线程一般被称为 warp 或 wavefront。一个 warp 中所有的线程以锁步的方式来执行，即在流水线中同时前进执行。GPU 能够通过细粒度多线程掩藏超长的内存访问延迟，即在一个 warp 正在等待内存访问时，新的 warp 指令会被取出并执行，因此在流水线中，并没有来自同一个 warp 的两个指令会同时执行。当没有有效的 warp 能被执行时，这个 GPU 核心将暂停。每个执行的线程可以访问独立的存储地址，因此 GPU 会潜在地产生大量同时发生的存储访问请求。

统一虚拟地址空间。现代 GPU 支持 CPU 和 GPU 的统一虚拟地址空间。这允许 CPU 可以通过 GPU 应用程序的指针访问并管理 GPU 物理内存中的数据。这一

特性极大地提升了 GPU 的可编程性。因为开发者只需要通过虚拟地址映射就可以便捷地管理 CPU 和 GPU 的存储空间。

统一内存。出现统一虚拟地址空间后，开发者仍然需要通过编程在 GPU 内存分配空间，并在 GPU 内核函数开始执行之前，从 CPU 内存拷贝数据到 GPU 内存。GPU 统一内存技术则将 CPU 内存和 GPU 内存看作一个整体的内存空间，所有的 CPU 程序访问或 GPU 内核函数的访问都可以抽象。因此，当 GPU 内核函数开始执行时，内存可以自动地管理移动数据。当 GPU 应用程序需要访问某一块数据时，数据不存在 GPU 内存中，则会产生缺页中断，从 CPU 内存直接调到 GPU 内存。

GPU 内存超额配置。虽然统一虚拟内存可以大大地提高可编程性，但这也不是万能的。首先，虚实地址转换的硬件结构会引入性能开销，降低 GPU 的吞吐率；第二，从 CPU 内存往 GPU 内存取数据要求频繁地高延迟数据搬移，开销非常大。许多先前的工作都以提升 GPU 虚实地址转换的性能为目标。（如并行地查页表，更高的 TLB 映射率以及更低的查页表延迟）。然而这些方法并没有尝试去直接解决实时按需取页的高开销问题。之前的工作发现预取在掩盖开销方面具有一定的优势，但这并没有考虑如何优化内存超额配置时的性能。

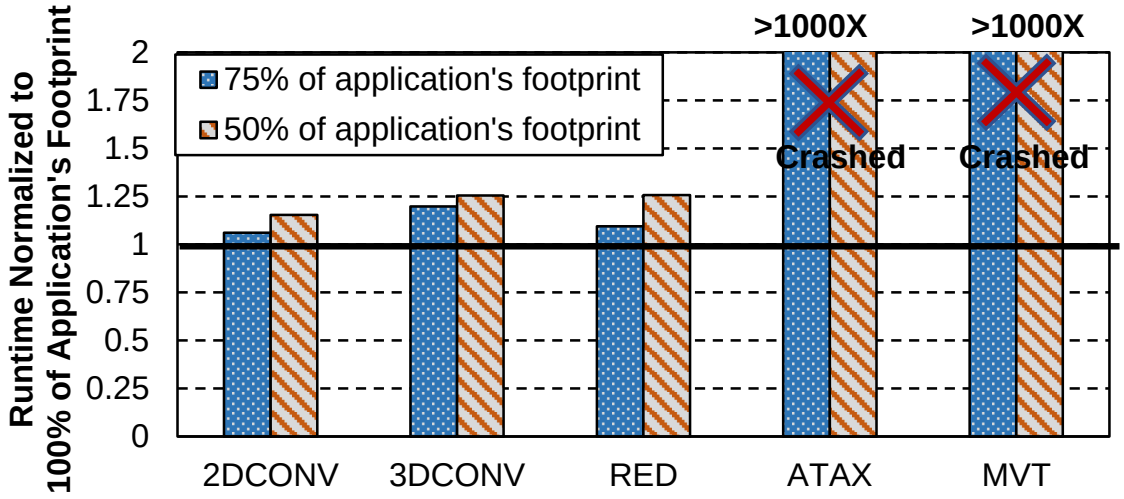


图 3.1　内存超额配置开销

图 3.1显示了内存超额配置下不同应用程序的性能损失。我们观察了了来自 CUDA SDK 和 Polybench 测试集的五个不同的 GPGPU 应用程序。这些应用程序运行在一个拥有 2GB 可用内存的 NVIDIA GTX 1060 GPU。为模拟内存超额配置，我们手动地修改可用内存，使之分别是每个 GPU 内核函数所需内存容量的大约 50% 和大约 75%。从图 3.1中结果我们有三点观察。第一，所有的应用程序在内存超额配置的情况下均出现了显著地性能下降：内存超额配置率越高，性能下降

的程度越大。第二，2DCONV、3DCONV 和 RED 三个应用程序性能平均下降了 17%。我们发现这个性能损失是由于为新拷贝的数据页腾出空间而等待现有物理页的逐出所导致的。这也是规则应用程序的内存超额配置开销的主要来源。第三，当有效内存容量为 ATAX 和 MVT 的内核函数所需内存空间的 75% 时，两个应用程序的性能分别下降了超过 1000 倍；而当 GPU 的有效内存空间仅能满足应用程序所需内存空间的 50% 时，会出现内存抖动的现象，即数据页会不停在 CPU 和 GPU 内存之间来回搬运，该现象产生过多缺页中断直接导致系统宕机。

3.4 设计空间探索：一种应用透明的框架

先前避免内存超额配置的方法。之前有许多技术都可以用来管理内存超额配置。增加存储容量是避免内存超额配置最有效的方法。例如封装内的三维堆叠存储器 HBM 和 HMC 已经被广泛应用到了 NVIDIA P100 和 V100 GPU、AMD Radeon R9 系列 GPU、以及谷歌的 TPUv2 上。然而，增加封装内的三维堆叠存储器的存储容量依然面临着三大主要挑战。首先，堆叠的层数受到了半导体制造技术的限制；其次，在硅中介层水平方向堆叠更多的存储器受到硅中介层引线复杂度和处理器引脚数目的限制；第三，随着 GPU 应用程序数据集持续增长，应用程序开发者仍然需要考虑如何使用有限的存储容量。除此之外，还有一些方法将任务划分并分配到多个 GPU，或者将任务划分成更小的粒度，使得每个任务所需要处理的数据所占的存储空间小一点。但是，这些方法都需要程序员花费很大的精力将一个复杂的大 GPU 任务分解，对程序员的要求非常高。将任务分配到多 GPU 系统还会引入更多的 GPU 之间以及 CPU 和 GPU 之间的通信，这也将带来额外的开销。

初始设计空间探索。为了缓解内存超额配置开销，我们尝试了多种不同的方案和策略进行设计空间探索。这些策略的主要目标是降低缺页中断开销。我们测试了不同的 warp 调度策略。分别为发出缺页中断请求的 warp 和无缺页中断请求的 warp 给定不同的优先级。非缺页中断请求的 warp 将拥有较高优先级，因此该 warp 已在内存中的数据能尽快被继续处理。但是，我们发现这种 warp 调度策略并不能减少缺页中断，只是将集中的缺页中断分散。最终，所有的线程都会在等待缺页中断处理的完成而停滞。由此我们得出结论，虽然 warp 级别的调度策略是掩藏内存访问延迟的有效方法，但这还是难以掩藏缺页中断处理延迟，因为缺页中断处理的延迟比访存延迟大了多个数量级。

我们也实验测试了几种不同的页替换策略来强化局部性原理并将内存抖动最小化。传统想法认为理想的 LRU 策略（Least Recently Used Policy）是页替换策略可以达到的性能上限，然而 LRU 策略在 GPU 内存上实现太过昂贵，需要大量

的查表和元数据存储开销。基于时序的 LRU 策略 (Age-based Least Recently Used Policy) 只需要一个列表存储每一个页从 CPU 内存移入 GPU 内存的时间，相对易于管理维护。我们的实验显示，基于时序的 LRU 策略对于流访问模式的应用程序效果非常显著，这种访问模式的应用程序即为我们定义的具有强序列局部性的规则应用程序。然而对于非规则应用程序，也就是随机访问模式的应用程序，基于时序的 LRU 策略并不奏效。在这种应用程序中，我们观察到内存超额配置会引起严重的内存抖动。因此，没有任何一种页替换策略能够有效减少内存抖动。

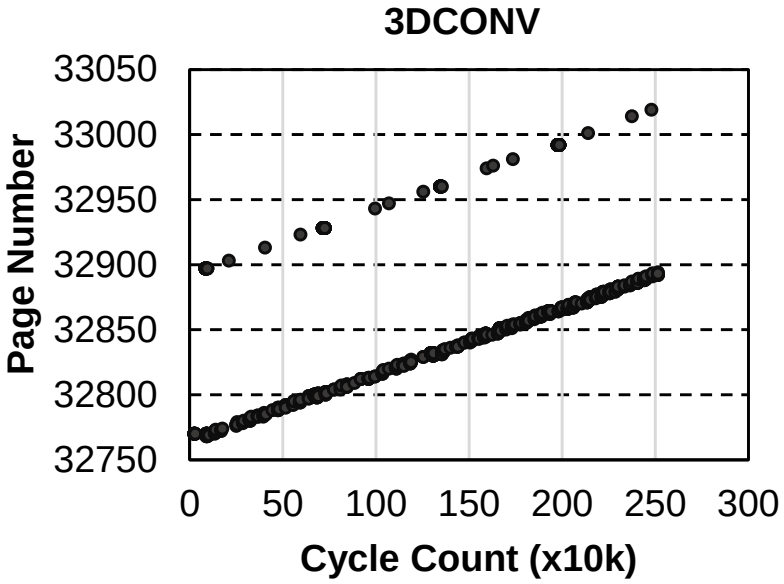
设计目标。我们主要有三个设计目标。首先，尽最大可能恢复内存超额配置下的程序性能到应用程序拥有足够的可用内存时的水平；其次，我们设计的框架必须对应用程序透明，因为我们不希望程序员手动管理控制物理内存；第三，我们的设计必须能够同时满足不同类的应用程序的不同需求。

3.5 GPGPU 应用程序的访存特性研究

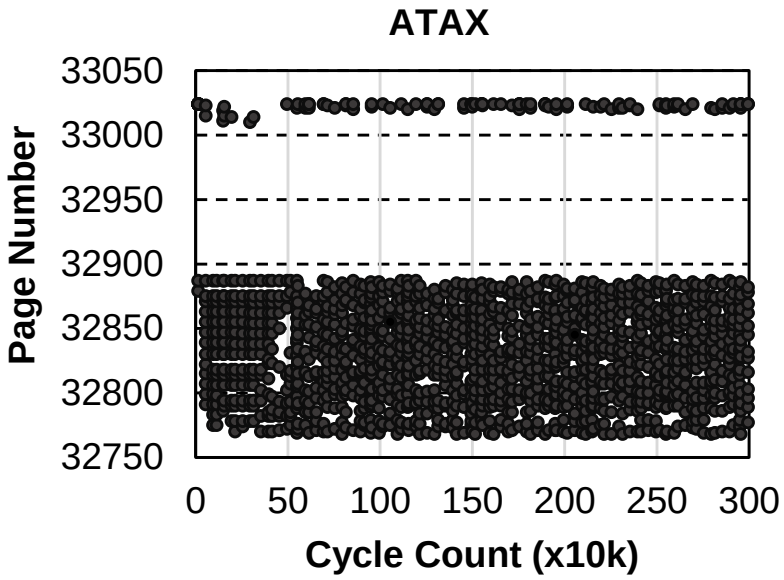
设计一个有效的内存管理框架要求对应用程序访存特性有深入的分析理解，也就是应用程序的特性。从这个角度出发，我们首先收集分析各种不同应用程序的访存 trace，抽取其中最具有代表性的访问模式。我们发现，应用程序一般可以被划分为规则访问模式和非规则访问模式。图 3.2（a）-（b）显示了两个代表性应用程序，3DCONV 和 ATAX。在线程块与数据页访问的关系中，3DCONV 呈现出典型的流访问模式，而 ATAX 则呈现出一种相对随机的访问模式。在任一时间点，3DCONV 只访问非常少数量的页。如图 3.3所示，几乎所有的线程块在一个时间点都在访问少量的几块数据页。相反地，无论在任一时间点，ATAX 都在同时访问大量的页面。如图 3.3（b）所示，ATAX 的不同线程块访问的是不同的页面。我们还发现有许多应用程序都表现出和 3DCONV 类似的内存访问特征。这一类应用程序的在线工作数据集相对较小，也就是说在一段时间内访问的数据量相对较小。相反地，像 ATAX 这一类非规则应用程序的在线工作数据集相对较大，因为每个线程都有可能访问一个新的不同的数据页。这将导致在给定的时间内大量不同的数据页被访问。此外，我们发现规则应用程序的访存是可以被预测的，如流访问模式，而非规则应用程序的访问模式是无法被预测的。

我们观察到对于规则应用程序，流访问模式的可预测性表现在被逐出的页通常都不会在一段时间后被重新请求，这很自然地避免了内存抖动现象。但是，对于非规则的应用程序，预测访问模式非常困难，一旦在线工作数据集超出了有效的内存容量，任何被逐出的页都有可能短时间内再次被请求，导致内存抖动的发生。

GPU 内核函数间数据共享。另一种可能的情况是某些规则应用程序，不同的 GPU 内核函数访问并处理同一块数据。如图 3.4所示，在 LUD 中，每个内核函数



(a) 规则应用程序

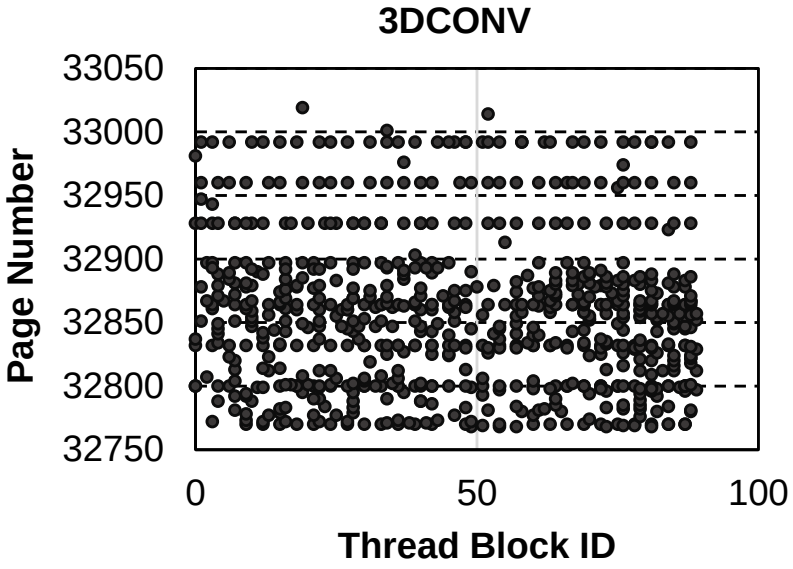


(b) 非规则应用程序

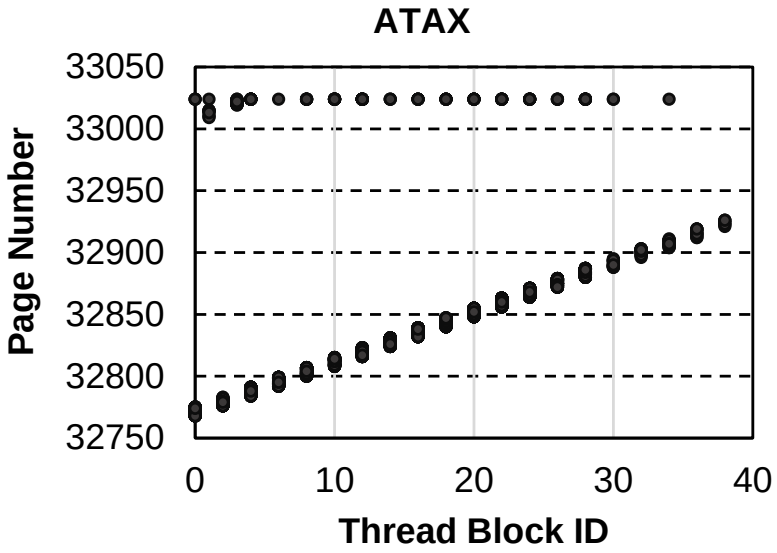
图 3.2　数据页访问特性实例：（a）规则应用程序和（b）非规则应用程序

的访存模式均为流访问模式且在线工作数据集相对较小。但是，当一个应用程序的多个内核函数共享一块数据时，这块数据将被重复多次访问。不同的内核函数之间存在同步栅栏，当数据大于物理内存的容量时，这块数据需要多次在 CPU 内存和 GPU 内存之间来回搬运，导致性能的大幅下降。

基于以上发现，我们得出三点结论。首先，数据页逐出等待时间是规则应用程序内存超额配置的最主要开销来源；第二，内存抖动主导了非规则应用程序的



(a) 规则应用程序



(b) 非规则应用程序

图 3.3　线程块访问数据页关系实例：(a) 规则应用程序 (b) 非规则应用程序

内存超额配置的性能损失；第三，数据共享引入了额外的数据迁移，导致更低的性能。这三点结论将指导我们框架的设计。

3.6　ETC 框架

ETC 的核心思想是为不同类型的应用程序采用不同的内存管理技术以缓解内存超额配置的开销。应用程序主要有三类：无数据共享的规则应用程序、数据共享的规则应用程序以及非规则应用程序。基于这三类应用程序，ETC 主要包含四

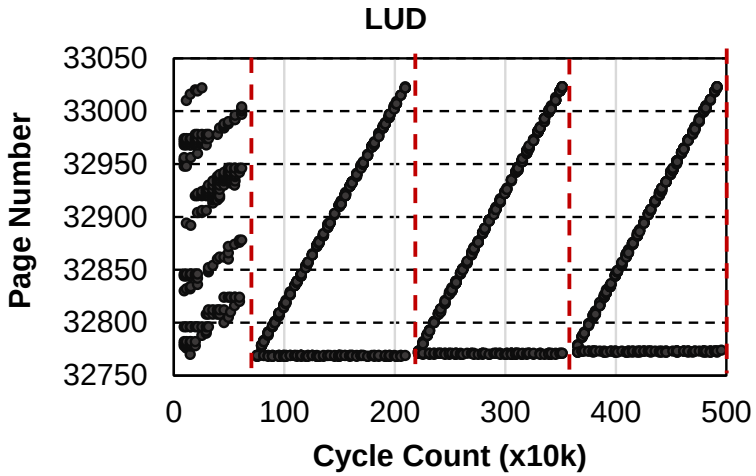


图 3.4　LUD 数据页访问特性：数据共享的规则应用程序（多个内核函数访问同一块数据；虚线代表每个内核函数的结束时间点）

个主要技术：应用程序划分技术、主动数据页逐出技术、内存感知的并行度控制技术、以及内存容量压缩技术。

一旦检测到内存超额配置，ETC 首先对应用程序类别进行判定。根据应用程序的类型，以及数据是否在多个内核函数之间共享，ETC 框架会选择主动数据页逐出、内存感知的并行度控制以及内存容量压缩技术中的一种或多种进行内存超额配置下的性能优化。对于规则应用程序，ETC 会采用主动逐出技术；对于数据共享的规则应用程序，ETC 在采用主动数据页逐出技术的同时还会采用内存容量压缩技术；对于非规则应用程序，ETC 会采用减少一部分运行的 GPU 核心来控制并行度，同时采用内存容量压缩技术来优化性能。

3.6.1　应用程序划分

在 ETC 框架能够选择为哪种应用程序采用哪一种技术之前，它首先需要检测应用程序的类型以及是否存在不同内核函数之间的数据共享。为检测每个 GPU 核心上运行的应用程序的类型，ETC 框架采用访存合并系数作为判断指标，这种应用程序剖析指标在 GPU 等单指令多线程体系结构的系统中被广泛使用。当来自于同一个 warp 的访存请求均访问的是同一个缓存行时，访存合并单元会将这些请求合并为一条访存请求以避免冗余访问，同时也减少了带宽消耗。对于规则应用程序，由于其访存局部性特征非常强，访存合并率也相应非常高。但是对于非规则应用程序，由于其局部性特征不明显，访存请求更多地呈现出低合并率。基于这些观察，ETC 框架采用了一个计数器在每个 GPU 核心的存取单元（LD/ST Unit）取样合并后的访存数。如果合并后的访存数低于一定的阈值，则认为该 GPU 核心上运行的是规则应用程序，否则将该应用程序划分为非规则应用程序。为检测多

个内核函数间的数据共享，ETC 依赖于编译技术。当编译器检测到来自多个内核函数的访存指针指向接近的数据地址，则表明该应用程序的多个内核函数之间存在数据共享。

3.6.2 主动数据页逐出技术

主动数据页逐出的关键想法是预先地在 GPU 用完所有的物理内存空间之前就逐出数据页。这将允许数据页逐出产生的数据迁移与缺页中断产生的数据迁移同时发生。图 3.5给出了主动逐出技术如何工作的例子。当页面在 GPU 内存缺失时，失败的虚实地址转换会产生缺页中断，内存管理单元（Memory Management Unit, MMU）则会向 CPU 内存取该数据页。如图 3.5（a）所示，如果应用程序用尽了物理内存，对新数据的访问需等到其他 GPU 内存中的数据页被逐出回 CPU 内存后才能执行。在目前的商业 GPU 产品系统中，数据页逐出只能被缺页中断触发。从 CPU 到 GPU 的内存迁移只能在 GPU 到 CPU 的数据页被逐出内存，数据迁移完成之后才能开始，如图 **??**（a）所示。如图 3.5（b）所示，我们观察到可以通过重叠数据页逐出和缺页中断处理的延迟来掩盖逐出开销，实现降低内存超额配置开销的目的。

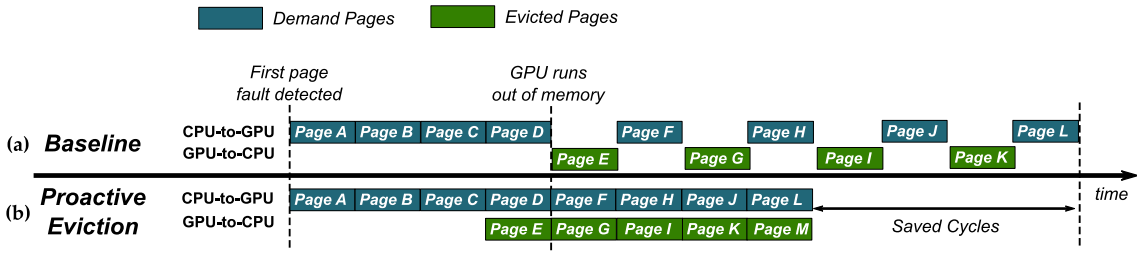


图 3.5 LUD 数据页访问特性：数据共享的规则应用程序（多个内核函数访问同一块数据；虚线代表每个内核函数的结束时间点）

当前 GPU 支持双 DMA 引擎，这使得缺页中断和数据页逐出产生的数据迁移理论上能够同时发生。应用程序开发者可以通过手动操作并行预取和逐出操作来优化应用程序。这种方法仍旧带给程序员巨大的负担，并且也违背了统一虚拟内存和按需取页的关键目标：减轻程序员负担。为了自动的并行页逐出和缺页中断的数据迁移，我们修改 GPU 的驱动使得应用程序消耗完 GPU 所有物理内存之前就逐出页。这样允许缺页中断的处理和页逐出的处理同时发生。

如何在最合适的时机进行主动页逐出是这个设计的重要挑战，也是需要我们解决的首要问题。一方面，如果太早从 GPU 内存中逐出数据页会导致正在使用的

数据页被逐出，该数据页很可能会被再次请求，带来额外的数据迁移开销；另一方面，如果太晚逐出数据页很可能导致主动逐出失效。另一个问题是，GPU 驱动程序需要确定一次逐出操作涉及多少个数据页。主动逐出过多的数据页也许可以减少 GPU 内存中不再使用的页，腾出更多的空间给新数据页。但是，如果主动逐出过多的数据页会导致仍在使用的数据页被逐出，造成额外的开销。我们开发出一种机制来平衡这个问题。

避免过早的数据页逐出。为确定主动数据页逐出的正确时机，我们剖析在 NVIDIA GTX 1060 GPU 上运行的几个不同的 GPGPU 应用程序。我们观察每个应用的内存访问轨迹，即数据页从 CPU 迁移到 GPU 的量是如何增长的。图 3.6显示了五个 GPGPU 应用程序从 CPU 内存移动数据页到 GPU 内存的轨迹图。

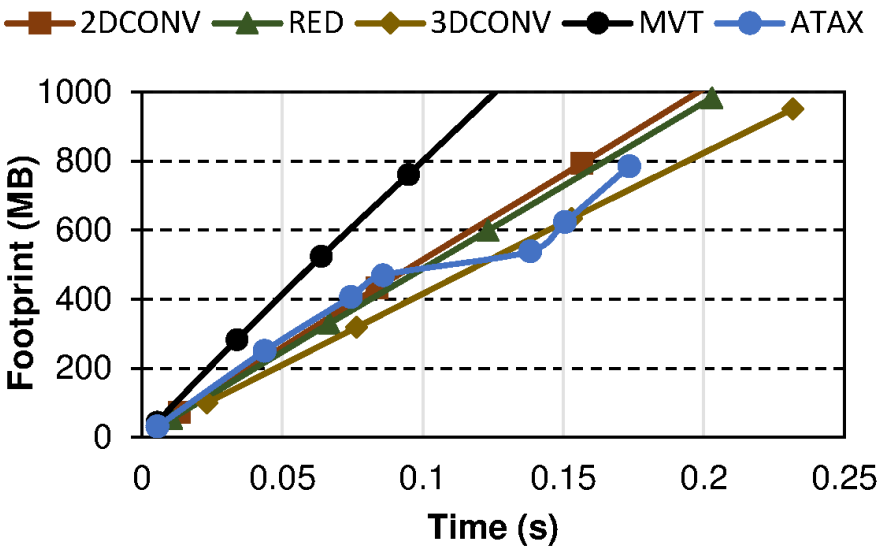


图 3.6　LUD 数据页访问特性：数据共享的规则应用程序（多个内核函数访问同一块数据；虚线代表每个内核函数的结束时间点）

从数据中我们得出四点观察。首先，内存占用大小是随着时间线性增长的；其次，有可能出现多个增长的阶段（如 ATAX 的内存占用情况（图 3.6中的蓝色点线）），但每个阶段内存占用的增长趋势仍然是线性的；第三，GPU 的单指令多线程工作模型显示，不同的 warp 执行相同的指令，但访问不同的数据。当所有这些 warp 并行地执行，共享全局内存带宽，则内存占用会增加直到这个阶段所有的数据被取回。这也解释了内存占用的增长为什么是线性的；第四，缺页中断之间的时间间隔是相对固定的。基于这些观察，GPU 可以预测当一个缺页中断发生时，一系列的缺页中断在短时间内会相继稳定发生。我们也可以触发多个数据页逐出请求，来为新需要的数据页腾出更多的空间。

避免过晚的数据页逐出。我们发现 CPU 内存到 GPU 内存的数据传输速度并不总是和 GPU 内存到 CPU 内存的数据传输速度相同。通过在 NVIDIA GTX 1060 GPU 的测试，我们发现从 GPU 内存到 CPU 内存的传输速度要明显快于 CPU 内存到 GPU 内存的传输速度。因此，传输同样数量的页，从 GPU 内存传输回 CPU 内存的速度相对于从 CPU 内存传输到 GPU 内存速度更快。基于这些发现，我们可以在发生缺页中断的同时开始页逐出避免过晚地逐出页。

非规则应用程序在短时间内访问大量的页面。因此，主动数据页逐出技术在非规则应用程序上使用将变得非常低效。我们发现潜在地缺点是主动逐出数据页技术很可能会加剧内存抖动现象。

实现方案。为了实现主动数据页逐出，ETC 框架修改 GPU runtime 中的虚拟内存管理单元，增加主动数据页逐出单元（PEU)。当缺页中断发生时，PEU 中断 GPU 驱动，使之移动缺失的数据页到 GPU 内存。当 GPU 驱动可成功分配一块新页到 GPU 内存时，PEU 首先开始检查应用程序划分逻辑。然后，PEU 检查内存分配空间大小，并与有效内存容量比较来预测是否内存超额配置。主动数据页逐出单元在以下情况被同时满足时会被触发：1. 内存超额配置；2. 有效内存小于一定的阈值（经验设置小于 2MB)；3. 应用程序被划分为规则应用程序。

3.6.3 内存感知的并行度控制

如 3.3讨论的，数据页层次的内存抖动会大大降低非规则 GPGPU 应用程序的性能。如图 3.2（b）和 3.3（b）所示，通常情况下，非规则应用程序的一个数据页仅被少量的线程块访问，不同的线程块访问的数据页也不一样。当 GPU 许多非规则应用程序的线程块同时执行时，在线工作数据集快速增长，产生严重的内存抖动。这是传统页替换策略无法解决的。为了避免内存抖动，我们的核心思想是限制同时访问内存和取数据页的数量。从这个角度出发，我们设计了内存感知的并行度控制技术，其目标是通过限制并行的线程数量，来减少非规则应用程序的有效在线工作数据集。GPU 并行度控制技术可以有两种实现方式，一种是线程块并行度控制，另一种是 GPU 核心（SM）并行度控制。线程块并行度控制降低每个 GPU 核心内部的线程块的并行度。而 GPU 核心并行度控制则降低每个 GPU 中核心的并行度。我们实验了这两种方法并发现线程块并行度控制方法引入了相对过长的时间来达到最少的内存抖动。与线程块并行度控制相比，SM 并行度控制相对更快限制并行度使之达到最合适的在线工作数据集大小。因此，ETC 采用了 SM 并行度控制的方法来减少 GPU 中的内存抖动现象。

实现方案。当非规则应用程序被检测到且发现内存超额配置时，ETC 会触发我们基于阶段的 SM 并行度控制技术。当并行度控制被触发，ETC 框架首先通过暂停一半 GPU 核心的取指功能，已经在流水线中的指令将继续执行直到完成。

在这个初始阶段，ETC 框架很可能停掉了过多的 GPU 核心，导致较低的利用率；另一种情况是停掉的 GPU 核心数量太少，仍然导致内存抖动。因此在初始阶段之后，ETC 框架需要基于观察到的内存利用率自适应地动态调整可执行的 GPU 核心数量。如图 3.7所示，内存感知的并行度控制将 GPU 的执行划分为两个阶段：检测阶段和执行阶段。在检测阶段，ETC 首先检测是否出现了逐出请求或缺页中断请求来判断初始阶段 ETC 的 SM 并行度控制是否过度。如果一个缺页中断请求被检测，或者检测阶段的运行时间结束，检测阶段将停止。一旦检测阶段停止，执行阶段开始，内存感知的并行性控制技术完成自适应地调整可运行的 SM 数量，不再调整，继续执行直到本阶段结束。

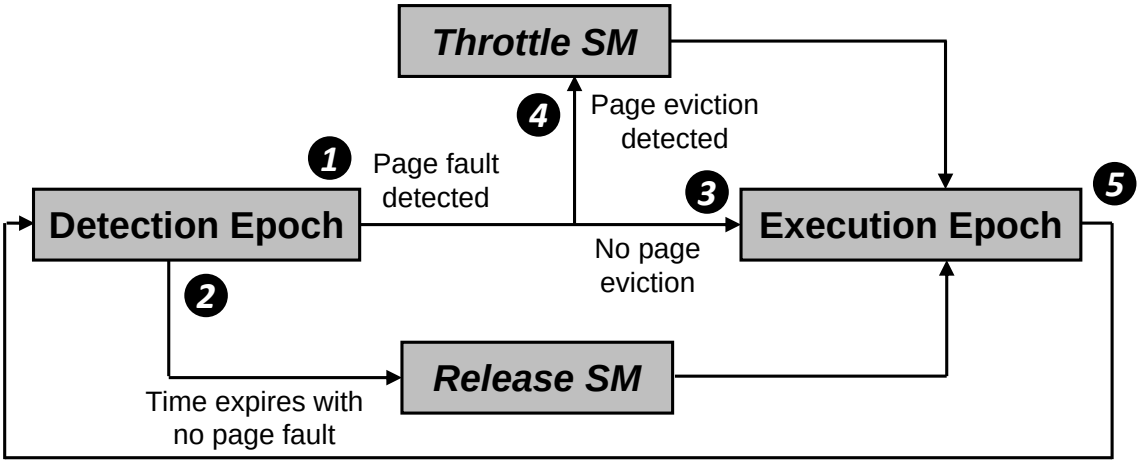


图 3.7　ETC 的内存感知的并行度控制策略)

如果检测阶段的结束是因为检测阶段的时间完成，并没有发生缺页中断。这说明当前的有效内存容量是足够当前的数据集的。GPU 还能够同时执行更多的线程，以更加高的内存利用率运行而无内存抖动。在这种情况下，ETC 框架将释放一个被暂停的 SM。为完成这一操作，ETC 使上一个被停止取指的 SM 重新开始取指来实现 SM 开启。

如果检测阶段的结束是因为一个缺页中断且无数据页逐出发生，这说明 GPU 依然有足够的有效内存空间。

如果检测阶段的结束时因为一个缺页中断，且发生至少一次页逐出，这表明 GPU 的内存容量已无法满足应用程序的在线工作数据集。ETC 框架需要限制更多 GPU 核心的执行，以减少在线工作数据集，使得内存容量足以容纳现有的工作数据集。在这种情况，只要缺页中断处理完成，ETC 将停止产生该缺页中断的 GPU 核心的取指功能，暂停其运行。因为这个 GPU 核心最有可能是刚开始执行处理数据。

在每次调整之后，GPU 会不受打扰地执行一段时间。当所有可运行的 GPU 核心执行完执行阶段的时间后，GPU 重新回到检测阶段，再次检测并调整可运行的 GPU 核心的数量。

通过内存感知的并行度控制方法，非规则应用程序的并行度被调整到在线工作数据集大小能完整地在内存中运行的状态。虽然并行度控制技术在一定程度上降低了线程级的并行度，我们发现它能大大减少 CPU 内存和 GPU 内存之间的数据搬运，并且大大恢复因为内存超额配置造成的性能下降。此外，这种并行度下降产生的性能损失，也可以通过结合容量压缩方法来挽回。这将在 3.6.4中具体介绍。

3.6.4 内存容量压缩

通过之前的介绍可以知道，在内存超额配置的情况下，主动数据页逐出技术可以有效提高规则应用程序的性能，而内存感知的并行度控制技术可以提升非规则应用程序的性能，但依然可能出现以上两种方法独立工作并不能提供足够的性能。首先，ETC 的主动数据页逐出技术可以掩藏数据页逐出延迟，但是在多个内核函数数据共享的情况下，不能减少 CPU 内存和 GPU 内存之间数据页移动的次数；第二，ETC 的内存感知的并行度控制技术虽然对避免非规则应用程序的内存抖动情况非常有效，但它会带来线程并行度下降的问题。

为了进一步降低内存超额配置的开销，我们的目标是提高内存的有效物理容量。从这个角度出发，我们开发了一种内存容量压缩技术。ETC 内存容量压缩技术的核心想法是，根据应用程序的类型，选择性地使用内存容量压缩来提升性能。以前已经有好几种主存压缩技术被提出，他们都能有效提高内存容量。在本章中，我们采用了线性压缩页技术（Linear Compressed Page，LCP）来压缩 GPU 内存中的数据。

LCP 是一种低延迟的内存容量压缩框架。之前在 CPU 的应用已显示其在提升主存容量方面非常有效。我们发现 LCP 在 GPU 中性能会有严重影响，因为它需要额外的一次访存来获取存储在主存中的压缩相关元数据。如图 3.8所示，额外的一次 LCP 元数据访问可以导致额外的带宽需求。实验显示在无内存超额配置的情况下，这些应用程序在 GPU 的性能平均下降了 13%。

因此，对于 ETC 来说确定何时采用 LCP 压缩技术非常重要。内存容量压缩技术主要应用在两种类型的应用程序中，包括数据共享的规则应用程序和非规则应用程序。因为来自这两种类别的应用程序的线程块会访问大量的数据，而内存压缩技术允许更多的数据存储在主存中。更重要的是，有了内存容量压缩技术，内存感知的并行度控制方法能够一定程度上缓解并行度的下降，这样相比单独并行度控制，更多的线程可以同时执行。

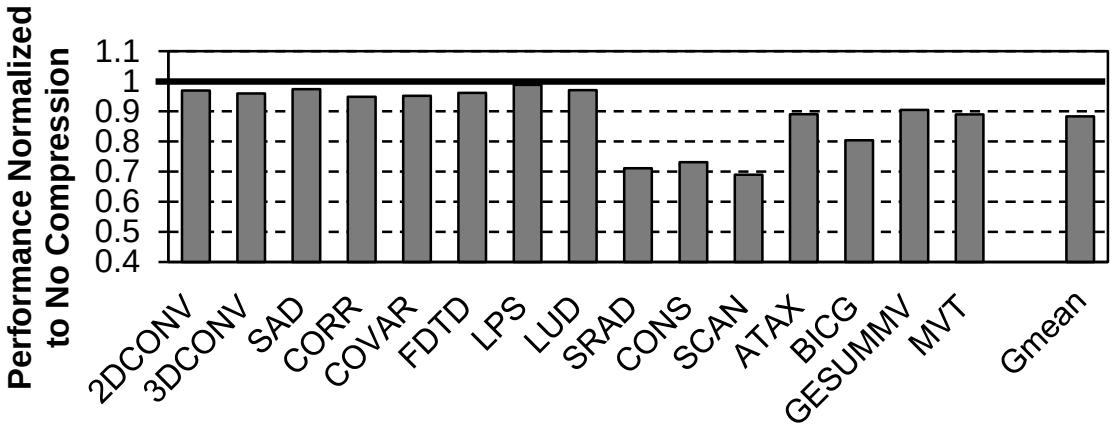


图 3.8　无限内存情况下的 LCP 内存容量压缩技术造成的性能损失

实现方案。由于当前的 GPU 已经在内存控制器和 PCIe 上应用了带宽压缩技术。内存控制器和 DMA 单元已经配置了压缩和解压缩硬件模块。为实现 LCP 压缩框架，ETC 采用了一个具有 512 个存储单元的缓存在内存控制器中以加速压缩元数据的查询，能够一定程度上降低 LCP 的性能开销。一旦应用程序划分单元确定当前执行的应用程序是数据共享的规则应用程序或者是非规则应用程序，同时内存处于超额配置状态，ETC 开始进行内存容量压缩操作。所有写入到 GPU 内存的数据将采用 BDI 压缩算法进行压缩，并通过 LCP 框架存入内存，非常简单高效。

3.6.5　ETC 设计总结

图 3.9是 ETC 的顶层设计方案，主要包括应用划分单元，主动数据页逐出单元、内存感知的并行度控制单元、以及主存容量压缩单元。当待分配的总数据量大于 GPU 的物理内存时，ETC 将被激活，应用程序划分器将开始根据 GPU 的硬件信息和编译信息识别应用程序类型。一旦应用程序被识别为规则应用程序，ETC 将在 GPU 驱动的虚存管理单元开启主动数据页逐出功能。如果应用程序有数据共享，则同时开启内存容量压缩技术。如果应用程序被识别为非规则应用程序，ETC 将同时开启内存感知的并行度控制技术和内存容量压缩技术，避免内存抖动现象的同时提升有效内存容量。

3.7　实验

3.7.1　实验方法

我们修改了基于 GPGPU-sim 3.2.2 的 Mosaic 模拟器来评估 ETC 框架。GPU 核心和存储系统的实验配置如表 3.1所示。

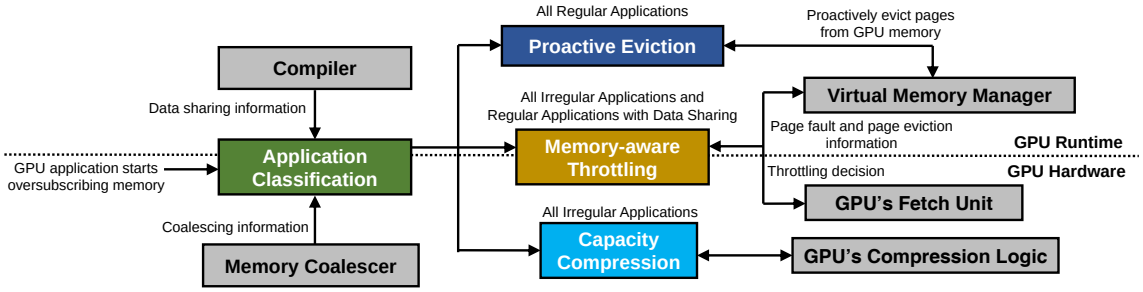


图 3.9　ETC 全局图。包含四个模块，分别是应用程序类别划分单元、主动数据页逐出单元、内存感知的并行度控制单元和内存容量压缩单元。

GPU 核心配置	
系统总览	30 个核心，每个核 64 个执行单元，8 个存储分区
核配置	1020 MHz，9 级流水线
	每个 warp 含 64 个线程，GTO warp 调度
私有一级缓存	16KB，4 路组相联，LRU
	L1 缓存失效将合并请求发送到 L2 缓存
私有一级 TLB	每个核心含 64 个条目，全相联，LRU
共享二级缓存	2MB，16 路组相联，LRU
	2 个缓存块，每个内存块包含 2 个互连端口
共享二级 TLB	总计 512 个条目，1 路组相联，LRU，2 端口
页表查询缓存	16 路 8KB
存储配置	
DRAM 内存	GDDR5 1674 MHz，8 通道，每个 Rank 含 8 个 Bank
	FR-FCFS 调度，burst 长度 8
页表查询	64 个线程共享页表查询器，查询四级页表
统一虚拟内存配置	64KB 数据页大小，2MB 逐出数据大小
	20μs 缺页中断处理时间，16GB/s PCIe 带宽

表 3.1　系统模拟配置

在线按需取页和内存超额配置。我们详尽地模拟了在 CPU 内存和 GPU 内存中按需取页数据迁移的过程，符合 CUDA8.0 的描述。当一个内核函数第一次要访问一块数据页时，TLB 缺失会触发页表查询。如果该数据页并不在 GPU 内存中时，页表查询失败，产生一个缺失页。内存管理单元会中断 CPU 来处理这个缺页错误。我们采用 20μs 的延迟模拟缺页中断处理延迟，并应用了最新的硬件数据页预取器来降低缺页中断开销。当 GPU 内存完全被用满之后，GPU 驱动会通过基于时序的 LRU 页替换策略逐出旧的数据页以足够的空间拷贝新的数据页。我们在实验中为每个应用程序配置完全足够的内存空间、应用程序所需内存的 75% 和 50%。

测试应用程序。我们随机广泛地从 CUDA SDK、Rodinia、Parboil 和 Polybench 选择了 15 个应用程序。我们将应用程序划分为无数据共享的规则应用程序（2DCONV、3DCONV、SAD、CORR、COVAR、FDTD 和 LPS）、数据共享的规则应用程序（LUD、SRAD、CONS 和 SCAN）以及非规则应用程序 (ATAX、BICG、GESUMMV 和 MVT)。它们在运行过程中由应用程序划分单元在线划分类型。这些应用所需的内存占用从 7.28MB 到 22.5MB 不等，平均值为 22.5MB。我们没有模拟更大的内存占用是因为模拟过大的内存占用会导致不现实的模拟延迟。

设计参数。ETC 框架采用了不同的设计参数。我们设置内存合并参数阈值为 10 个缓存行来划分规则应用程序和非规则应用程序。我们设置 2MB 作为剩余 GPU 内存空间的阈值来触发主动页逐出技术。我们设置并行度下降和并行度上升的程度为一次一个 GPU 核心，我们实验发现这个值能取得最高的性能。

3.7.2 实验结果

我们评估 ETC, 并将其与 1）当代采用数据页预取的基准模型（BL）以及 2）一个理想情况下有无限内存的模型作比较。

3.7.2.1 性能结果

图 3.10显示了不同类别应用程序以无限内存的基准模型为性能标准的相对实验结果。基于这些结果，我们得出了三点结论。首先，ETC 在降低内存超额配置开销上作用显著。对于无数据共享的规则应用程序，其性能接近无限内存的基准模型情况。这是因为通过我们的主动数据页逐出技术，这一类应用程序最主要开销来源的数据页逐出延迟可以被完全掩盖。第二，我们发现数据共享的规则应用程序由于不同内核函数之间的同步，数据页的额外搬运不能完全避免。但是 ETC 相比于当前采用数据页预取的基准模型依然平均提升了 60.4% 的性能。第三，对于非规则的应用程序，ETC 相比采用数据页预取的基准模型依然提升了 2.7 倍。我们给出结论，ETC 框架对于内存超额配置的性能恢复非常有效。

3.7.2.2 技术和应用程序细节分析

我们在本小节提供了关于每种 ETC 采用的技术对于每种类型应用程序影响的深度分析。

无数据共享的规则应用程序。图 3.11显示了主动数据页逐出技术和容量压缩技术对于无数据共享的规则应用程序的影响。我们做出三点观察。第一，当主动数据页逐出技术被触发，数据页逐出延迟几乎可以完全被掩藏。从实验中我们可以发现，对于无数据共享的规则应用程序中仅 LPS 因为逐出过多的页而导致性能离理想情况稍有下降。第二，由于页面限制图 3.11并没有完全展现，无数据共享

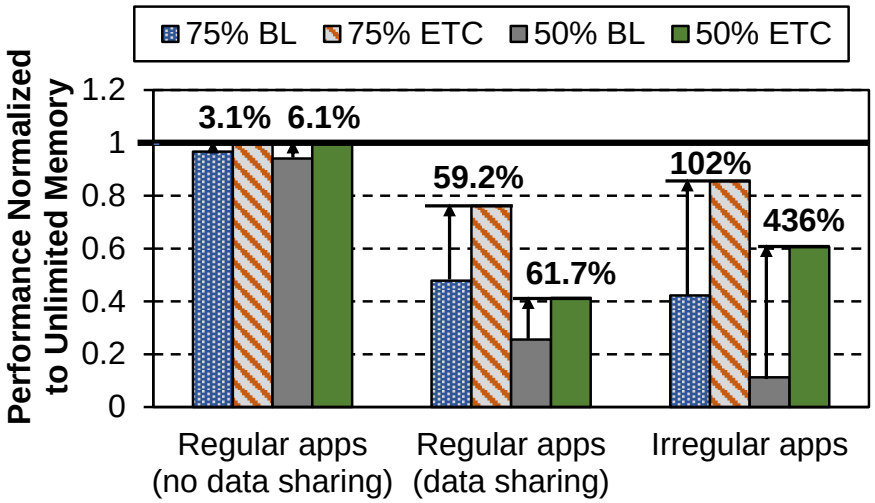


图 3.10　ETC 相对于无限内存的基准模型的性能

的规则应用程序并不能从内存感知的并行度控制技术中获得性能提升。因为该类型的应用程序在线工作数据集本身就非常小，无需降低并行度。同时该类型应用程序的主要开销来源为逐出的延迟。事实上，内存感知的并行度控制技术使得线程级并行下降的同时，延迟掩藏能力也会下降。第三，无数据共享的规则应用程序在采用了内存压缩技术后，性能比基准模型更低。这是因为额外的压缩相关元数据访问所带来的开销，这在 3.6.4节已详细介绍的。

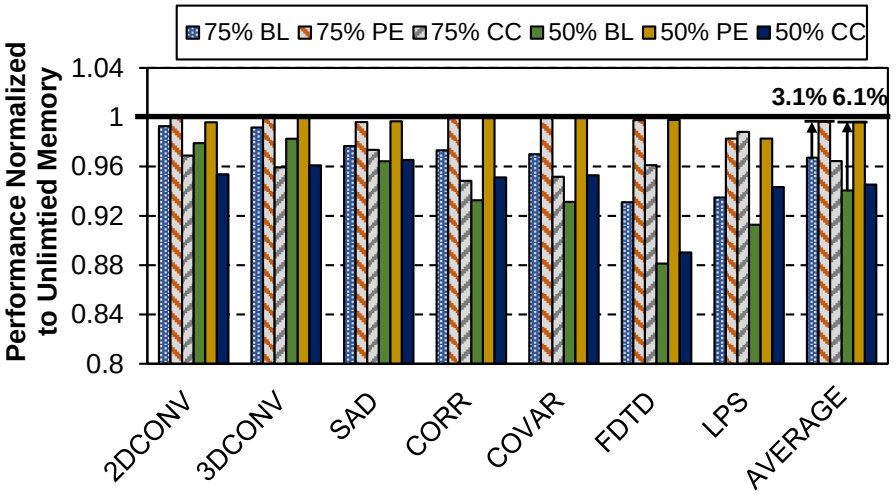


图 3.11　无数据共享的规则应用程序的性能

数据共享的规则应用程序。图 3.12显示了数据共享的规则应用程序在主动数据页逐出技术、容量压缩技术和二者一起使用时的性能。我们得出了四点观察。第一，与无限内存的理想基准情况相比，数据页预取的基准模型在 75% 和 50% 的数据集能被物理内存容纳的情况下性能分别下降了 52.2% 和 74.1%。第二，仅

采用主动数据页逐出技术，性能相比于基准模型仅提升了 9.3%。这是因为由于数据共享导致的数据移动在这种情况下是内存超额配置的最主要开销。第三，仅使用内存压缩技术可以获得 52.8% 的性能提升。这是因为它增加了有效内存容量。第四，当主动数据页逐出技术和内存压缩技术同时使用时，平均可以获得 60.4% 的性能提升。我们得出以下结论，ETC 框架提升了规则应用程序的性能，无论数据是否在多个内核函数之间共享。

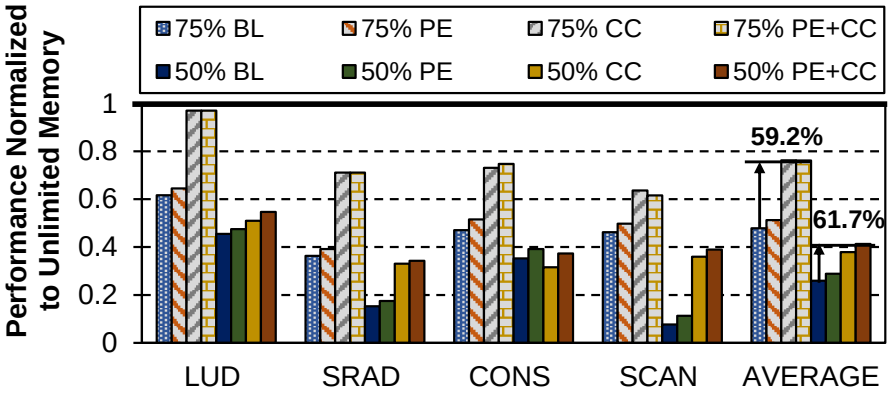


图 3.12　数据共享的规则应用程序的性能

非规则应用程序。图 3.13和图 3.14显示了每个非规则应用程序在 ETC 各个技术下的性能和总的数据页逐出次数。为评估 ETC 框架的并行度控制策略（MT），我们将它和一个简单的并行度控制策略进行比较，该策略静态地在开始执行时减少一半的 GPU 核心数量（如图 3.13和图 3.14所示）。我们得出了三点观察。首先，简单的并行度控制方法在内存能容纳 75% 的数据集的时候性能提升 57.7%。第二，当内存仅能容纳 50% 的数据集的时候，简单的并行度控制方法变得不那么有效，降低了 10.5% 的性能。相反地，我们的内存感知策略，可以动态地调整可运行的 GPU 核心数量。相比于当前的基准模型，我们的内存感知策略提升了 436% 的性能。第三，我们的自适应调整的并行度策略在 BICG 和 GESUMMV 两个应用程序，且内存容量仅能容纳 75% 的数据集的情况下性能不及简单的并行度控制方法。这是因为自适应调整需要一段时间才能达到最佳并行级别，而简单的并行度控制方法在这种情况恰好一次性达到了较好的并行级别。

图 3.15显示了 1000 万个时钟周期里非规则应用程序 ATAX 的缺页中断率。当内存超额配置且 75% 的数据集能存放于内存中，会发生内存抖动，且频繁地发生缺页中断。相反地，当内存感知的并行度控制方法被激活时，缺页中断并不频繁。这说明我们的内存感知的并行度控制方法对于降低在线工作数据集的大小非常有效。同时，图 3.13和图 3.14的数据页表明，内存感知的并行度控制方法可以减少数据页逐出的发生。

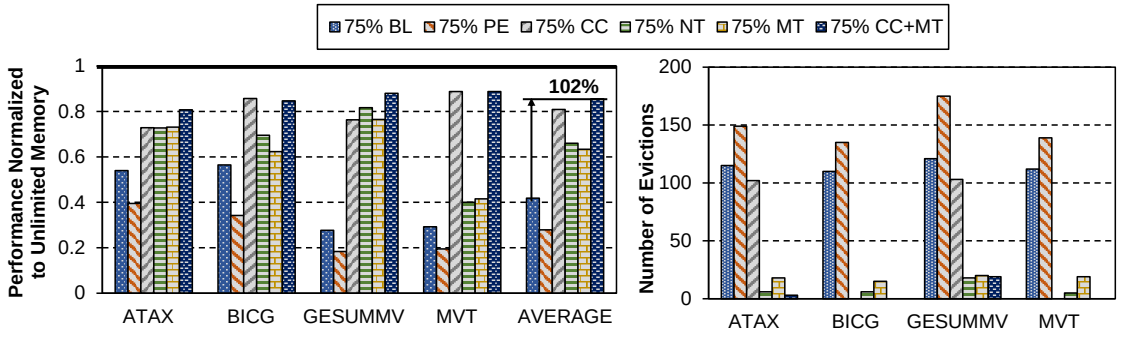


图 3.13　非规则应用程序的性能（75% 的内存占用可以被容纳进内存）

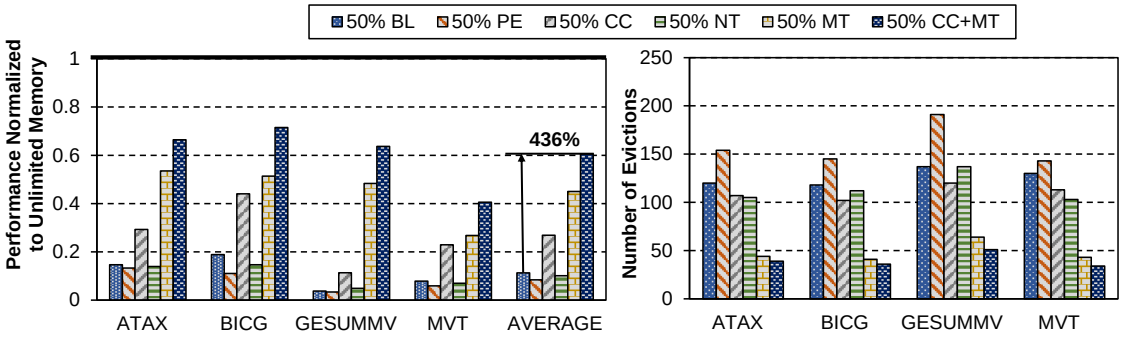


图 3.14　非规则应用程序的性能（50% 的内存占用可以被容纳进内存）

内存容量压缩策略对于非规则应用程序的有效程度取决于压缩率和 GPU 相对于应用程序的内存占用的比例。当一个应用程序完整的内存占用能够在压缩之后被物理内存容纳，缺页中断将不再发生。图 3.13中 BICG 和 MVT 大大减少的数据页逐出数目说明这两个应用程序的压缩率足够高，使得内存能够完全容纳应用程序的内存占用。相比于当前的数据页预取基准模型，内存压缩技术分别提升

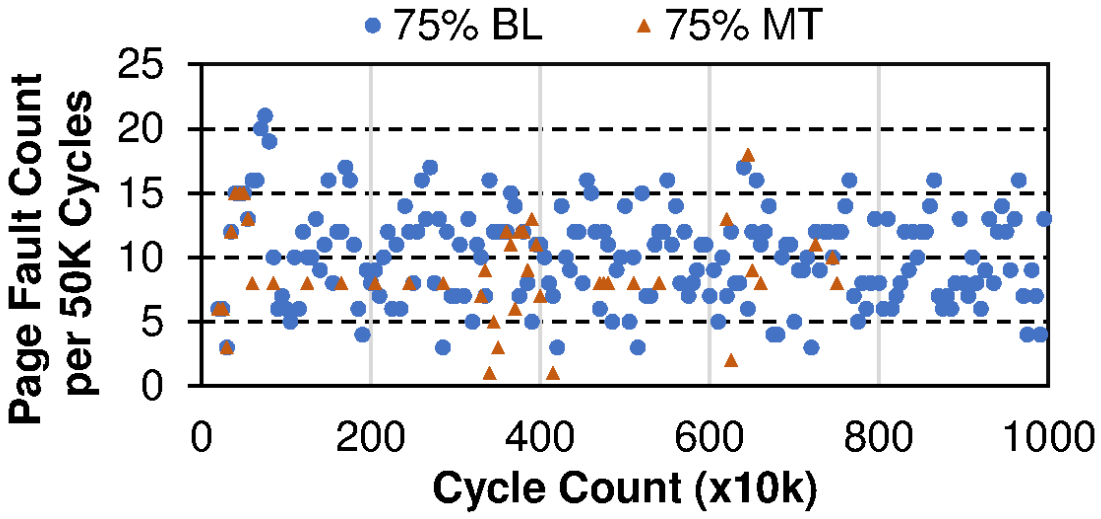


图 3.15　ATAX 的缺页中断率变化趋势

了 51.8% 和 203.6%。相比于无限内存的理想基准情况，BICG 和 MVT 分别恢复了 85.7% 和 88.7% 的性能。图 3.14显示了 GPU 的内存仅能容纳 50% 的数据集的性能情况。即使使用了内存容量压缩技术，所有的应用程序都出现了内存抖动。我们的内存感知的并行度控制技术在和内存容量压缩技术同时使用时，相比于当前基准情况的性能提升了 436%。我们得出以下结论，采用内存容量压缩技术有助于提升内存超额配置下非规则应用程序的性能，然而缺页中断和内存抖动仍然会限制应用程序的性能。因此，需要内存容量压缩技术和内存感知的并行度控制一起工作以在内存超额配置的情况下达到较好的性能。

我们观察到主动数据页逐出技术相比于被动数据页逐出技术性能下降了 29.7%。这是因为数据页被过早地逐出 GPU 物理内存。因此，主动数据页逐出技术并不适合非规则应用程序。

总的来说，内存感知的并行度控制技术和内存容量压缩技术对于提升非规则应用程序的性能非常有效。因此，我们的 ETC 框架采用这两种技术。如图 3.13和图 3.14所示，采用了这两种技术的 ETC 框架能够提升 270% 的非规则应用程序的性能。虽然并行度控制技术在一定程度上降低了线程级并行，但它能够有效降低内存超额配置的开销以及缓解内存抖动。

3.7.3 应用程序划分的精确度分析

ETC 框架依赖于正确的应用程序类别划分。只有确定正确的应用程序类别才能选择最合适的策略（见 3.6.1）。图 3.16比较了从 5 万个时钟周期取样的平均访存合并系数与每个应用程序的真实平均访存合并系数。我们可以观察到规则应用程序和非规则应用程序的访存合并系数相差非常大。我们发现，将访存合并系数阈值设置在 5 到 10 之间都能使得 ETC 的应用程序类别划分准确读达到 100%。因此在本课题研究中我们将阈值设置为 10。

图 3.17显示了每个 warp 指令中 32 个线程访问的平均数据页数目。我们发现非规则应用程序的 warp 一般会访问多个数据页，而几乎大多数的规则应用程序一般只访问一个数据页。因此，采用数据页级别的访存合并系数与缓存行级别的访存合并系数来判定不同的应用程序类别均有非常高的准确率。

3.7.4 敏感度分析

在这一节我们测试了不同敏感度下 ETC 框架的性能。主要包括内存感知的并行度控制策略中调节粒度大小、缺页中断处理延迟大小和内存容量大小对 GPU 内存超额配置下的性能影响。

内存感知的并行度控制度。每个阶段并行度提高或降低所变化的 GPU 核心的数量称为并行度控制的度。它能够直接影响应用程序的性能。图 3.18和图 3.19显

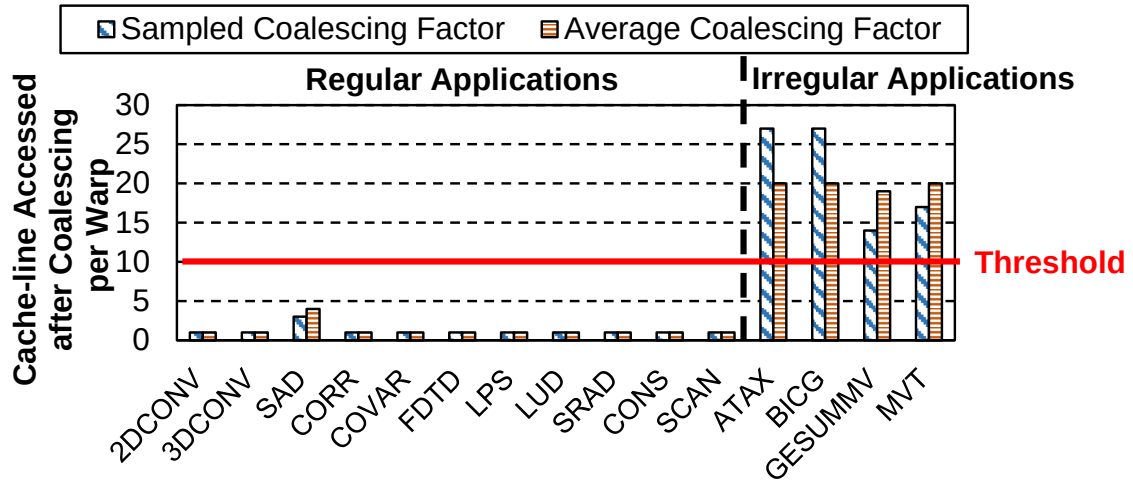


图 3.16　不同应用程序的访存合并系数 (缓存行合并)

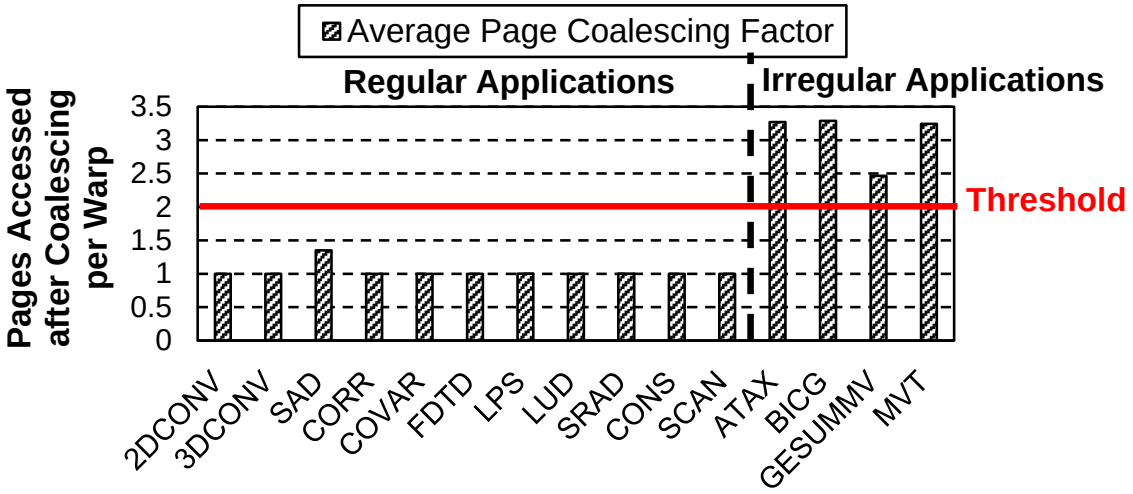


图 3.17　不同应用程序的访存合并系数 (数据页合并)

示了每个阶段调节不同的 GPU 核心数（每次增加或减少的 GPU 核心数量）的相对性能。基于图 3.18和图 3.19，我们得出两点观察。第一，我们的内存感知的并行度控制策略在并行度下降和上升度均为 1 时得到最高的性能。这说明细粒度的调整效果最好；第二，相比于并行度上升，我们观察到 ETC 框架的性能对于并行度下降更加敏感，因为缺页中断产生的开销相比于线程级并行的下降产生的开销更大。

缺页中断延迟。图 3.20显示了 GPGPU 的应用程序在不同的缺页中断处理延迟下的性能。实验设置缺页中断延迟从 20μs 到 50μs 不等。实验结果显示的是相对于缺页中断延迟为 20μs 的性能。我们观察到当缺页中断延迟从 20μs 提高到 50μs，平均性能下降了 31.2%。这些数据说明隐藏页逐出开销对于恢复内存超额配置下的性能非常重要，因为缺页中断延迟是主要的性能瓶颈。

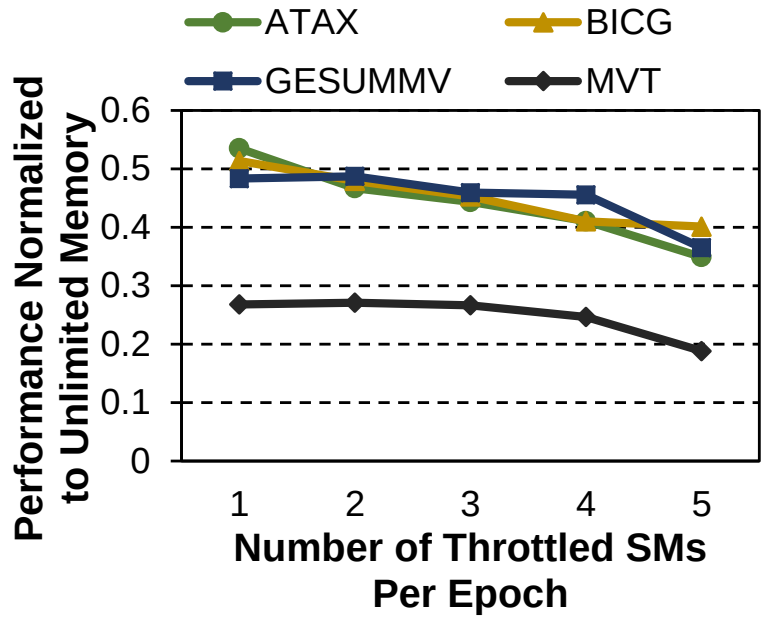


图 3.18　ETC 的性能与减少 GPU 核心数量的关系

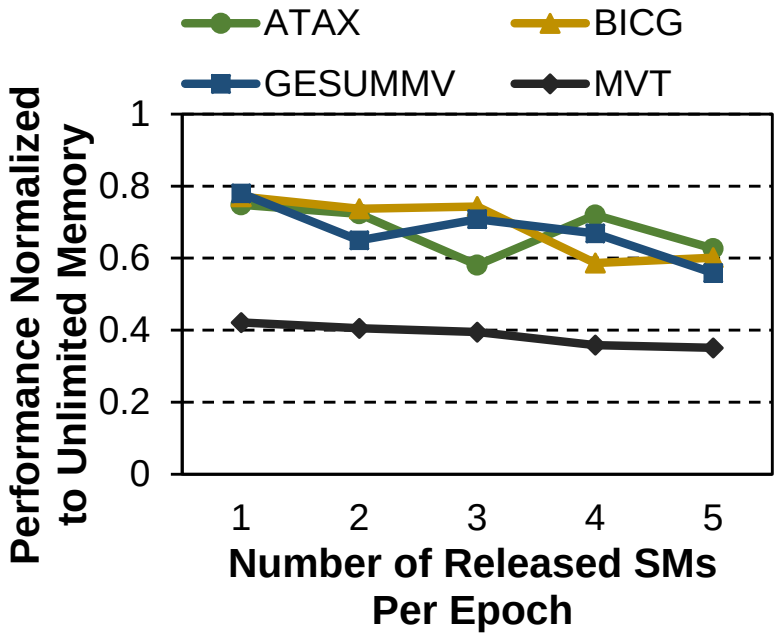


图 3.19　ETC 的性能与增加 GPU 核心数量的关系

压缩率。主存容量压缩技术影响到 GPU 主存能够容纳的数据页的数目。图 3.21显示了所有应用程序采用不同的合成压缩率下的性能，相对于该应用在没有采用压缩技术的性能。该实验在物理内存仅能容纳下 50% 的内存占用的内存超额配置下进行。实验结果显示 GPU 性能随着内存压缩率的增加而线性提高。当所有的内存占用能够被 GPU 内存容纳时，性能有大幅度提升（压缩率为 2）。

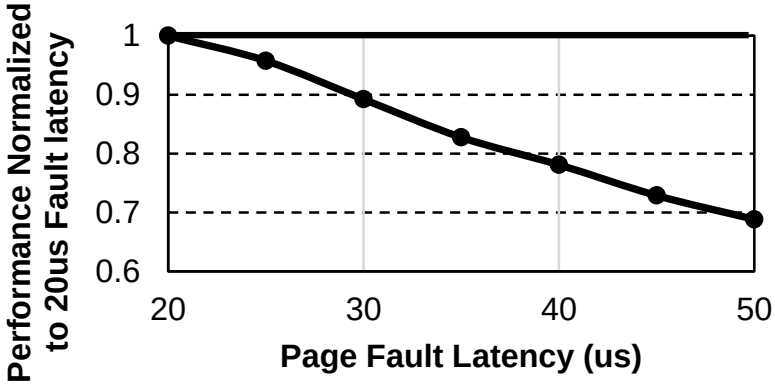


图 3.20　ETC 的性能与缺页中断延迟的关系

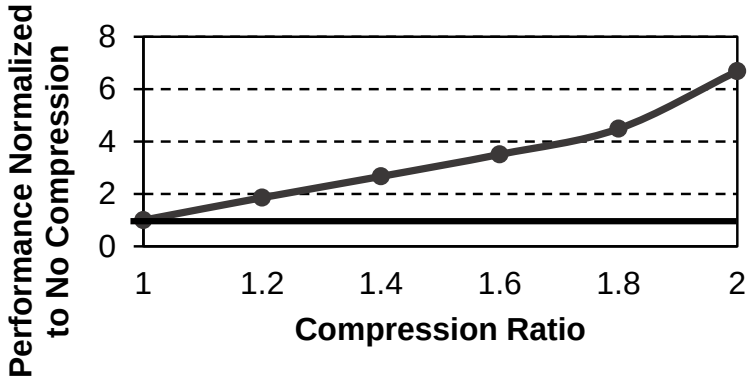


图 3.21　ETC 的性能与内存容量压缩率的关系

3.7.5　硬件开销

我们分析 ETC 框架的每个部分所耗费的硬件开销。主动数据页逐出技术不需要任何硬件开销，可以直接在 GPU 驱动中实现。我们修改了驱动来检测可用的内存大小以触发主动数据页逐出技术。为实现内存感知的并行度控制技术，内存管理单元必须扩展以支持我们的设计。增加两个 32 位的计数器来统计每个阶段的时钟周期。暂停每个 GPU 核心的取指功能需要增加控制逻辑。为实现容量压缩技术，我们增加的硬件开销和为 CPU 设计的 LCP 框类似，主要包括一个有 512 个条目的压缩元数据缓存。内存压缩技术不需要额外的硬件开销，因为当前 GPU 已经具备了内存压缩和解压缩功能。压缩与解压缩单元已经存在于存储控制器中。我们扩展页表项，每个页表项增加 9 位来包含压缩相关信息。最后，应用程序划分单元需要：（1）每个存取单元增加一个 32 位的访存合并计数器；（2）连接取指单元，压缩单元以及内存管理控制器的信号控制线。总的来说，我们的设计增加的硬件开销是有限的。除了逻辑开销外，32KB 的元数据缓存和 482 个 32 位计

数器（30 个 GPU 核心中每个 GPU 核心包含 16 个计数器，内存管理控制器有两个计数器），总存储开销小于 2KB。

3.8 本章小结

我们介绍了 ETC，一种在 GPU 中能够有效减少内存超额配置下的开销的框架。该框架是对应用程序透明的，当内存超额配置情况下，规则应用程序和非规则应用程序呈现出不同的应用特性。规则应用程序的性能受到数据页逐出延迟的影响非常大，而非规则应用程序的性能主要受到内存抖动的影响。ETC 划分应用程序为规则和非规则的应用程序，我们采用（1）主动数据页逐出技术来掩藏页逐出延迟；（2）内存感知的并行度控制技术来缓解内存抖动；（3）容量压缩技术增加了有效内存容量。对于无内存共享的规则应用程序，ETC 消除了内存超额配置带来的性能损失，性能与无内存超额配置下类似。对于数据共享的规则应用程序和非规则应用程序，ETC 相比于当前技术，性能分别提升了 60.4% 和 270%。我们得出结论，ETC 是一种高效低开销的框架，对于 GPU 内存超额配置开销最小化具有重要作用

第四章　一种动态采用检查点备份技术的 GPU 主动抢占策略

4.1　研究概述

GPU 由于其大规模并行处理能力，已经在高性能计算，机器学习和科学计算等领域得到广泛应用。这些领域的计算如今以服务的形式存在于数据中心或云上，而 GPU 则可作为基础硬件资源提供给不同的用户。多任务处理在 GPU 中为支持并行服务和任务已经变得必不可少。已经有些重要的硬件特性支持多任务处理，例如 NVIDIA Kepler 体系结构提供的 HYPER-Q、AMD 支持的命令处理机制等。虽然已经有这些机制支持多任务处理，但还需要更多的工作来支持真正的多任务处理机制。

上下文切换是一种在 CPU 中常用的支持并行性的技术，如今该技术已经被应用到了 GPU 来支持多任务切换。CPU 的进程相对来说非常轻量化，所以在上下文切换和时分任务等应用上非常快速高效。但是，一个 CUDA 线程的上下文相比于 CPU 的线程却是非常巨大的。比如在 NVIDIA GTX980 GPU 上，上下文包括每个流多核处理器的 256KB 的寄存器和 96KB 的共享内存。对于一个含 16 个流多核处理器的 GPU，其上下文大小达到了 5664KB。存储这样大的上下文需要耗费大量的存储带宽，并带来严重的性能损失。

之前已经有许多探索降低 GPU 的上下文切换的开销的方法。最早出现的技术是让上下文切换仅在一部分流处理器核上进行，这样其他的流水里器核就能保持继续执行。被切换的流处理器核会完全停止执行指令，以完成上下文的存取。这些操作对于存储带宽的要求依然非常高。之后出现一种方法，让一部分线程块继续执行直到完成，而只上下文切换一部分线程块。这样可以最大化的利用程序访存和上下文访存的重叠并行性。这个技术被进一步加强扩展，允许在需要抢占时每一个流处理器核内部的不同线程块同时进行执行（直至线程块完成)、丢弃（满足幂等性)、和上下文切换。这些选择均取决于每个线程块对于抢占的终止时间的要求。除了这些工作，一种轻量级的上下文切换技术被设计来降低需要保存在片外存储器的上下文大小。所有这些方法都是通过被动的方法来实现抢占，即只有当抢占请求到来以后才激活所有的操作。因此，如果没有丢弃操作，抢占延迟依然是对性能的一大挑战。

在本章，我们提出了一种动态主动的抢占机制，称为 PEP。这种抢占机制能够大大降低抢占延迟和开销。通过观察内核函数从 CPU 到 GPU 的启动过程，我们发现内核函数的实际执行总是在内核函数启动之后。从一个内核函数在 CPU 启

动到开始在 GPU 执行，大约需要几十个毫秒的数量级，我们可以通过预计抢占请求的到达时间来主动准备上下文切换。当抢占的内核函数真正到达的时候，需要等待完成的上下文切换工作将变得非常小。因此，需要等待的抢占时间将变得非常短。准备上下文切换的工作我们采用了检查点（checkpoint）的概念。第一个检查点，我们在抢占被预计发生时备份当前的上下文。当真正的抢占请求到达 GPU 后，仅需再备份变化的上下文部分。备份变化的上下文相比于完整的内核函数的上下文能够节省大量的时间，减少抢占内核函数的等待时间。平均来看，总的需要备份的上下文不大于所有的上下文。我们还观察到分配的上下文在线程块的生命周期里并不是完全激活的。所以，我们为寄存器设置脏位，来表明该寄存器是否是有效的。只有有效的寄存器才会被备份，这大大减少了需要存储的上下文大小。此外，我们设计了一个动态实时调度策略来确定抢占方法。短的内核函数将要继续执行直到结束，而长的内核函数需要采用检查点备份（上下文切换）来进行抢占。这个算法可以达到最小延迟和开销。我们的贡献主要包括：1）我们研究了内核函数启动的过程，观察到抢占的事件可以被预测。2）我们引入了一种主动的抢占技术来减少抢占的内核函数等待上下文切换的时间。采用主动 checkpoint 技术，当真正的抢占请求到来时，只有一小部分的脏上下文需要被存储。3）我们使用了一种相对简单的脏数据存储技术来减少上下文大小，这可以减少不必要的上下文存储。4）我们开发了一种相对更加精确的线程块执行时间和上下文切换时间估算方法，设计了实时动态选择算法来确定采用的抢占方法。我们可以完成短内核函数和长内核函数的抢占，并使之达到最短延迟和最小开销。

我们实验评估了 PEP，并与之前最好的抢占工作 Chimera 在几种不同类型的测试集里进行比较。实验结果显示，相比之前的工作 Chimera，我们可以将平均抢占延迟从 8.9us 降低到 3.6us。我们采用的简单的减少上下文大小技术，将需要存储的上下文减少了 16.1%。PEP 的总开销，即平均线程块切换延迟，相比 Chimera 减少了 6.3%。

4.2 研究背景

在这一节，我们首先简单介绍了 GPU 的基本结构及其工作模式。我们的基准结构模拟的是一款 NVIDIA 的 GPU 体系结构。因此，我们在本章主要使用 NVIDIA/CUDA 的术语。但是，本章的想法也可以应用到其他厂商的 GPU。此外，我们还介绍了 checkpointing 的方法，该方法在我们的设计中起到了关键作用。

4.2.1 基准结构

GPU 程序执行：典型的 GPU 程序包括两个部分的代码：在 CPU 上运行的主机部分的代码，以及在 GPU 上运行的设备代码（kernels，本文称之为内核函

数)。内核函数是以单指令多线程的模式执行。一个内核函数的执行意味着无数线程同时在 GPU 上并行执行。线程会被程序员组合成线程块。NVIDIA GPU 的 CUDA 编程模型以 CUDA C 的形式展示给程序员，即从 C 语言和实时库中扩展而来。图 4.1是一段 CUDA 程序的例子。一段典型的 CUDA C 程序的操作序列包括：1）声明和分配主机和设备的内存（8-13 行）2）从主机内存向设备内存迁移数据（14 行）3）启动内核函数。在这个例子里，程序员启动 N/256 个线程块，每个线程块包括了 256 条线程（15 行）4）从设备内存向主机内存迁移数据（16 行）5）释放内存空间（18-19 行）

```
1  __global__ void axa(double a, double *x){
2    int i = blockIdx.x*blockDim.x+threadIdx.x;
3    x[i] = a*x[i] + a;
4  }
5
6  void main(){
7    int N = 1048576;
8    double *x, *d_x;
9    x = (double*)malloc(N*sizeof(double));
10   for (int i = 0; i < N; i++) {
11        x[i] = 3.0;
12   }
13   cudaMalloc(&d_x, N*sizeof(double));
14   cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
15   axa<<<N/256, 256>>>(3.0, d_x);
16   cudaMemcpy(x, d_x, N*sizeof(double), cudaMemcpyDeviceToHost);
17   std::cout<<"Output:"<<x<<std::endl;
18   cudaFree(d_x);
19   free(x);
20  }
```

图 4.1　典型 CUDA 代码示例

线程块之间是互相独立的，他们被分别发送到 GPU 核心上。每个 GPU 核心上能够并行的线程块数量受限于设备的资源（包括寄存器，共享内存和线程的数量），这个信息可以在编译阶段获取。大多数之前的抢占策略的相关设计工作是以线程块的粒度完成的，也会采用可用资源的信息帮助不同抢占策略的选择。

GPU 体系结构：图 4.2是 GPU 基准体系结构，我们本章所描述的 GPU 体系结构均基于此结构。当一个 GPU 程序收到主机 CPU 执行时发送的操作指令，用户空间实时引擎将 API 调用指令转换为相关的 GPU 控制数据操作和内核函数的启动。GPU 设备驱动发送这些操作指令到流控制管理器的队列里。流控制管理器通过软件队列来管理多条不同的流（streams）；每一条流里的指令将被串行执行。一般来说，CPU 会先声明并分配存储空间，然后调用 cudaMalloc 来分配 GPU 上的全局内存。之后，一个 cudaMemcpy（H2D）API 的调用将数据从主机内存移动

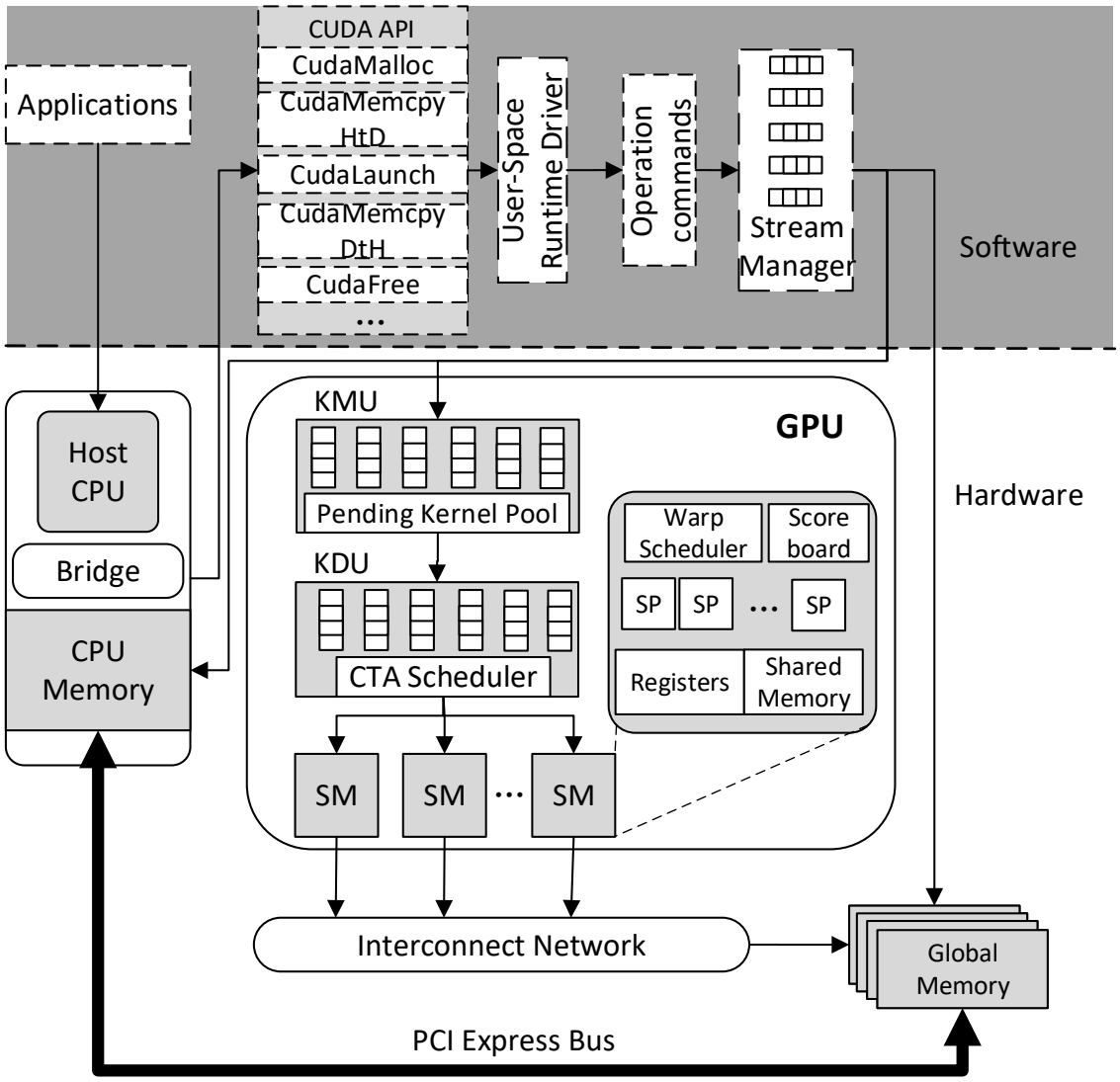


图 4.2　GPU 基准体系结构

到设备内存。一旦所有的数据被传输完成，流控制管理器可以启动内核函数，即传输内核函数相关信息（例如维度配置和每个条目的 PC 地址）到内核函数管理器单元（Kernel Management Unit，KMU)。当所有的信息准备完毕，内核函数会请求 GPU 核心资源。如果内核函数没有足够的资源，内核函数需要等待内核函数等待池有新的空间。如果正在等待的内核函数的优先级高于正在执行的内核函数，则等待的内核函数有可能需要抢占 GPU 核心中正在占用资源的线程块。否则，需要等待之前的内核函数执行完毕再接着执行。一旦内核函数准备好开始执行，它将被传输到内核函数分发单元（Kernel Distributed Unit，KDU)。线程块调度器将相应的线程块分发到不同的 GPU 核心。每个 GPU 核心能够处理的线程块的最大数量取决于资源限制，包括可以执行的线程块数量、线程数量、寄存器数量和共享内存空间。在每个 GPU 核心中的内核函数执行过程中，线程块将被分成

warps，每个 warps 包含 32 或 64 个线程。每个 GPU 核心包含一个或多个 warp 调度器，来选择执行哪一个 warp。在 NVIDIA GTX980 GPU 系统结构中，每个 warp 调度器控制 32 个流处理单元（SP），每个流处理单元处理一个线程。当一个 warp 因为访存或其他耗时较长的操作而停滞时，调度器会切换执行其他的 warp。切换 warp 没有任何开销，因为所有 warps 的上下文已经被放入寄存器和共享内存中。因此，GPU 通过掩藏停滞 warp 的延迟，大大提升了并行性能。

4.2.2 先前的抢占方法

当抢占发生时，每个 GPU 核心的操作是独立的。这意味着有一部分 GPU 核心可能在执行抢占操作，而另一部分 GPU 核心在继续执行当前内核函数直到结束。被抢占的 GPU 核心需要将该核内的上下文存储备份到全局内存。一个 GPU 核心的上下文就是其执行的状态，包括 SIMT 栈、寄存器和共享内存。SIMT 栈存储的是线程执行信息，例如程序计数器和有效线程掩码（用于分支处理）。相比于寄存器和共享内存的大小，SIMT 栈的大小可以忽略不计，因此在本章我们暂不考虑 SIMT 栈。一个线程块在其执行的时候占用 GPU 核心的资源；线程块保持活动状态直到其执行完毕。但是，在其执行的时候，有可能有新的内核函数会被启动。如果该新的内核函数需要满足严格的延迟要求，而等待上一个内核函数执行完毕再开始执行则无法满足该延迟要求。因此我们需要去抢占一些正在运行的线程块来为新的内核函数的线程块提供资源。但是，线程块的上下文相对来说非常大，将他们存到全局内存将引入相当大的开销，抢占延迟也会非常大。如表 4.1所示，抢占延迟（平均上下文切换时间）有可能超过 20us。这些延迟给需要满足延迟要求的新内核函数造成很大隐患。

为达到较低的抢占延迟的目标，Park 等人提出了一种方法，在满足幂等性要求的前提下可以直接丢弃线程块，后续再重新执行。在这个方法里，GPU 核心直接丢弃掉线程块的上下文，并不备份相关上下文。之后直接执行来自更高优先级内核函数的线程块。在这个内核函数运行结束以后，GPU 核心会重新开始执行被丢弃的线程块。丢弃操作几乎没有抢占延迟开销。但是，并不是所有的内核函数都能在任意时间执行丢弃操作。丢弃操作要求内核函数是幂等的（Idempotent），这意味着该内核函数无论执行多少次，运行结果都是相同且独立的，即无原子操作，在丢弃操作发生之前无全局内存的写操作。大多数应用程序并不是幂等的(大约 30% 的 Rodinia 应用程序不满足幂等性条件)。幂等性可以变得相对宽松，但记录幂等性的开销非常大。因此，实现丢弃操作的开销也可能非常大，与需要重复执行的指令成比例。

为了达到较低的抢占开销，GPU 核心排空执行操作被提出。该方法要求在新的内核函数的线程块开始执行之前，当前的线程块继续执行，直到结束。这种方法不要求上下文的存储备份，因此抢占开销能够最小化。但是，这种情况的抢占延迟也可能非常高，这是因为内核函数的执行时间可能非常长，执行时间会非常长。这很可能导致抢占内核函数无法满足其延迟要求。表 4.1包含了我们测量的不同的内核函数的单个线程块运行时间。我们可以看到，有的线程块（如 Kmeans）的执行时间达到接近 1ms。因此，GPU 核心排空执行的方法最适合较短延迟的线程块。

Lin 等人提出一种轻量化的上下文切换方法来减少需要备份到片外的上下文的大小。这些技术主要包括本地上下文切换，将上下文存到未使用的寄存器或共享内存；清除废弃的寄存器，即降低上下文大小；还有寄存器压缩技术，我们在 PEP 中也采用了本地上下文切换技术。但是，采用活跃度（liveness）信息要求为每一条指令的每一个寄存器提供一个活跃位（liveness bit），这将引入一个非常大的活跃度表存储在硬件里。为降低这种较大的开销，抢占操作只能在特点的时间点执行，才能尽可能的使用较少的存储单元来存储活跃度信息。而寄存器压缩技术也需要额外的硬件开销，因此我们也不在 PEP 中采用。

4.2.3 GPU 中的检查点技术

检查点技术即是将一个运行的进程的状态备份的方法，目的是当出现错误时，能够从检查点恢复操作。GPU 检查点技术已经在软件上被实现。虽然检查点技术能够恢复一个进程，但该技术的目的是容错，并不适合于抢占操作。运行进程的设备有可能出现错误，所以有必要将运行状态存储到另一个设备。这是一个非常高延迟的操作，但是相比于错误导致的工作进程丢失，这是非常必要的。对于抢占操作，我们的目标是为抢占内核函数达到一个理想的反应时间，因为抢占内核函数需要满足一定的延迟要求。因此，我们需要存储上下文到设备的全局内存。检查点技术被采用来降低未来的上下文切换的延迟。为引入检查点技术到抢占操作中，限制检查点次数尤为重要，因为我们另一个目标是降低抢占操作的开销。

4.3 研究动机

Chimera 采用了一个选择算法在抢占请求到来时为不同的线程块选择不同的抢占方法；这个选择是基于对前面介绍的三种方法的选择和平衡。Chimera 估计了每一种技术的抢占延迟和开销来选择最有效的抢占方法。因此，GPU 核心中不同的线程块会被不同的抢占方法被抢占。

应用程序	内核函数	平均启动时间	平均线程块执行时间	平均切换时间	平均线程块大小	每个 GPU 核心线程块数量
CUTCP (CP)	cuda_cutoff _potential	5.8μs	516.2μs	10.1μs	16.5KB	8
LBM (LBM)	performStream Collide_kernel	21.8μs	31.7μs	20.9μs	18KB	14
MRI-Q (MRI)	ComputeQ_GPU	10.4μs	865.2μs	11.6μs	18KB	8
STENCIL (ST)	block2D_hybrid _coarsen	4.5μs	41.3μs	4.2μs	12.5KB	4
STREAM CLUSTER(SC)	kernel_ compute_cost	6.7μs	605.6μs	8.3μs	24KB	4
GEMM (GM)	matrixMulCUDA	23.4μs	193.6μs	17.9μs	28KB	8
BLACK SCHOLES(BS)	BlackScholarGPU	3.4μs	387.5μs	16.7μs	12.5KB	16
KMEANS (KS)	invert_mapping	29.7μs	984.7μs	9μs	10KB	8
PATHFINDER (PF)	dynproc_kernel	11.3μs	24.2μs	11.6μs	18KB	8
SRAD_V1 (SRAD1)	extract	5.2μs	1.8μs	4μs	12KB	4
SRAD_V2 (SRAD2)	srad_cuda	15μs	11.5μs	16.4μs	25KB	8
SRAD_V1 (SRAD3)	srad	5.2μs	7.9μs	7.8μs	24KB	4
HOTSPOT (HS)	calculate_temp	33.3μs	4.5μs	7.7μs	38KB	3
LUD (LUD)	lud_internal	4.4μs	5.3μs	10.5μs	16KB	8
BACKPROP (BP)	bpnn_ layerforward	16.7μs	4.7μs	2μs	12KB	1
BACKPROP (BP2)	bpnn_adjust _weights	16.7μs	1.5μs	1.2μs	22KB	1

表 4.1　各应用程序启动时间、线程块切换时间与执行时间.

但是我们发现排空执行方法和上下文切换方法会竞争全局内存的带宽。举例来看，在图 4.3中，访存密集型的应用程序 LBM 会遇到这样的一种冲突：LBM

在每个 GPU 核心中可以执行 9 个线程块；我们展示了所有 10 种切换和排空执行的可能性组合。如果所有的线程块被一个又一个地切换，则不会出现切换和排空执行的竞争。而另一种情况是当一个 GPU 核心在上下文切换时，其他 GPU 核心都在排空执行时，排空执行的时间和上下文切换时间都将远长于 8 个线程块在做上下文切换操作，1 个线程块在排空执行的情况。因此，我们发现带宽竞争会导致 Chimera 时间估计不准确的问题。在这种情况下，当 1 个线程块在排空执行时，其他线程块一个接一个被切换时带宽竞争非常小。另一方面，这个排空执行的线程块不会与其他线程块竞争执行单元。因此，IPC 受到排空执行的线程块的数量影响。

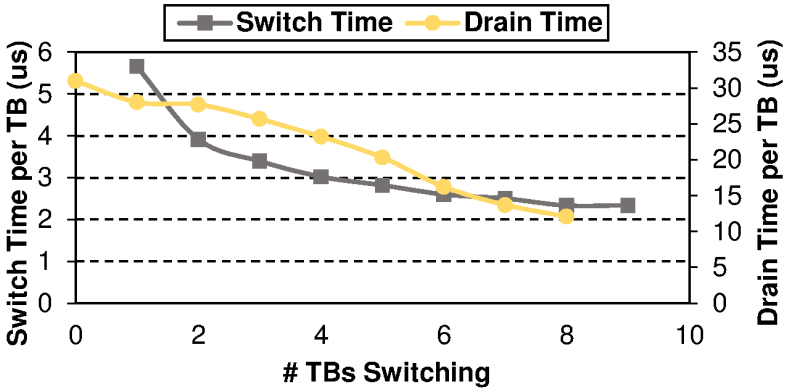


图 4.3　LBM 的上下文切换时间和排空执行时间对比（每个 GPU 核心有 9 个线程块在同时执行）

我们也观察到在每个 GPU 核心中所有的线程块通常选择同一种抢占方法。表 4.1展示了线程块执行时间的范围远大于上下文切换的时间范围。因此，对于长内核函数，排空执行时间和上下文切换时间相差巨大。因此，对于短的抢占内核函数来说，最佳方法是排空执行 GPU 核心内所有的线程块，这可以满足延迟要求的同时接近零抢占开销。相反，对于长的内核函数，如果不满足幂等性要求，则必须做上下文切换。

当前 GPU 的总上下文大小是每个 GPU 核心有 352KB（寄存器包含 256KB，而共享内存包含 96KB）。为了传输所有上下文到全局内存，假设带宽被完全利用需要大约 15us。之前所有的技术都是被动的，他们至少需要这么长时间来完成上下文切换。为了进一步降低上下文切换的抢占延迟，我们不仅需要减少上下文的大小，还需要一种主动的抢占方法。

检查点技术即是一种主动机制广泛应用于容错处理中。该方法周期性地存储当前进程的执行状态到下一级存储中。类似地，我们可以针对抢占操作，将正在运行的线程块的上下文存储到全局内存中。我们介绍一种新的检查点方法，称之

为 PEP。PEP 在抢占发生之前将当前检查点的上下文备份，当真正的抢占操作开始后，我们只需要被动地存储这段时间更新的上下文，这大大减少了抢占等待时间。

4.4 相关工作

在 GPU 多任务抢占研究方面，主要的目标是降低抢占延迟和开销。由于 GPU 上下文的大小相对于 CPU 上下文要大很多，CPU 的抢占方法难以直接在 GPU 上采用。在传统的上下文切换方法的基础上，Tanasic 等人提出了排空执行方法进行抢占。这种方法对于相对较短的内核函数非常有效。Park 等人提出了丢弃执行的方法。这种方法对于满足幂等性要求的抢占内核函数可以实现零抢占延迟。此外，他们的工作还包含将上下文切换、排空执行和丢弃执行基于线程块的执行时间混合选择执行。Wang 等人设计了细粒度动态共享机制，SMK。他们的设计使得基于线程块的细粒度上下文切换能够实现较低的抢占延迟。

在缩减上下文大小方面，iGPU 的观点是，可以在满足幂等性代码区域之间的边界处备份和恢复上下文。他们利用活跃度分析来确定恢复点。这些点的活跃度寄存器数量相对较少。Lin 等人提出三种技术实现轻量级上下文切换，包括本地上下文存储、活跃度分析和上下文压缩等。

检查点技术主要用于容错处理上。传统的检查点软件，如 BLCR，支持通过自定义的 Linux 内核执行检查点备份 CPU 的状态。这并不能在 CPU 片外的 GPU 工作，因为 GPU 的内存是通过设备驱动程序操作的。因此，BLCR 不能恢复其状态。CheCUDA 是第一个尝试去解决 NVIDIA GPU 上的这个问题。它作为一个 BLCR 的附加组件实现，可以与 BLCR 一起工作，但要求重新编译应用程序。NVCR 进一步提升该方法，支持使用实时 API 的更大集合类的应用程序。它替换了 libcuda.so，因此，不需要重新编译应用程序。此外，虚拟化是另一个采用了检查点备份技术的应用。vCUDA 则是一个 GPU 虚拟化方面的重要工作。

我们的检查点备份技术并不是传统的面向容错处理的检查点备份技术。实际上，这种上下文备份类似于检查点备份技术。我们利用了检查点备份的技术在多任务抢占中降低需要存储的上下文的大小。不同于传统的周期性的检查点备份，PEP 采用检查点备份方法最多执行两次（一次初始检查点备份，一次增量备份)。初始检查点备份有系统调用抢占内核函数的启动而触发，来自于硬件信号。当前的检查点备份技术需要上下文的存储和错误恢复单元来保证可靠性，而我们的 PEP 检查点备份则是一种更加轻量化的方法，不要求任何错误恢复单元。

所有之前的抢占工作都是被动的，意味着这些机制都是在抢占内核函数启动并要求资源后才会被触发。因此，任何算法都需要等待 GPU 核心的上下文切换完成或者排空执行完成。我们通过利用内核函数启动过程的特点，PEP 设计了一种

主动触发技术。利用检查点备份方法，PEP 可以实现一个相对于其他方法更短的呃抢占延迟，同时开销可控。

4.5 设计

在本节中，我们首先给出了主动抢占设计的全局概要图。然后我们将证明预测内核函数启动时间和估计抢占时间的可行性。最后，我们提出了基于检查点方法的设计和在线选择算法。

4.5.1 全局设计

我们的方法是来源于一个观察：只要延迟和开销是可以接受的，上下文切换可以在线程块执行的任何阶段发生。为了降低延迟和开销，我们将减少上下文的大小。为了减少抢占延迟，我们可以提前做上下文切换。在合适的时机我们采用了排空执行的方法，因为这个方法几乎没有开销。

为了减少上下文的大小，我们采用了脏位来指明一个寄存器是否被写过。因此，我们不会存储未使用或者已经被释放的上下文。我们还会采用 Lin 等人提出的本地上下文备份的方法，该方法支持让上下文存入空闲的本地内存。在这个方法中，不需要通过互连网络将数据传输到全局内存，因此不占用存储带宽。

我们采用检查点技术来实现主动上下文切换。我们的算法支持在抢占之前的某个检查点存储数据到全局内存。接着继续执行当前内核函数，直到抢占请求到来。此时，我们只需要将这段时间相对于上一个检查点的上下文更新存储到全局内存中。如果一个线程块在第一个检查点和抢占发生直接执行完成，则释放第一个检查点存储的上下文。这个方法可以取得远小于一次完整上下文存储的开销。

为了在抢占中实现检查点技术，我们必须限制检查点存储发生的次数。如果我们备份太多次检查点状态，开销很可能无法接受。另一方面，如果一个线程块在多个检查点之后，抢占之前执行完毕，则之前的检查点备份均浪费掉了，同时还耗费了不少开销。因此，我们有必要预测线程块是否能在抢占点发生的时候依然在执行。此时，我们只需要为在抢占发生时依然在执行的线程块执行检查点技术。我们知道内核函数在 GPU 的执行是在 CUDA API 调用 cudaLaunch 之后发生的。在这个 API 调用之后，一个内核函数启动的命令将被发送到流控制管理器中。如果这个命令进入流队列的队列头，内核函数的相关信息将被发送到内核函数管理单元，开始请求 GPU 核心资源。因此，我们发现内核函数启动的时间是可以被预测的。

我们的检查点技术非常适合打内核函数。对于短的内核函数，我们还是会采用排空执行的方法代替上下文切换。为了采用这两种抢占技术，我们需要估计排空执行和上下文切换的时机以在线选择合适的抢占方法。

4.5.2 预测和估计

内核函数启动时间的预测和排空执行时间和上下文切换时间的估计是 PEP 的关键部分。通过我们对一系列应用程序的研究分析，我们发现有三个时间对我们预测策略的正确性特别关键，分别是内核函数启动时间、上下文切换时间和线程块执行时间。内核函数启动时间可以用来预测什么时候抢占请求会真正启动。上下文切换时间和线程块执行时间可以用来确定采用基于检查点的上下文切换技术还是排空执行技术。表 4.1展示了我们测量的三种延迟。

1)**预测**：从对表 4.1的研究我们有两点重要的观察。第一，是我们发现内核函数启动的时间和上下文切换的时间（接近检查点技术的延迟）是在同一个数量级。这意味着如果我们在开始在预测的时间进行一次检查点备份时，很可能在抢占请求发生时，我们刚好完成有效上下文的存储备份。第二，无论是上下文切换时间还是内核函数启动的时间，在大多数情况都远小于线程块的执行时间。对于长内核函数的线程块，不太精确的预测不会影响我们是做检查点备份还是排空执行。因此，一次错误的预测不会对最终的延迟性能和开销造成巨大影响。我们将在本节详细讨论这些延时信息。

前面已经提到，我们必须预测何时抢占请求会出现以避免检查点备份作废。一个 CUDA 应用程序一般包括 5 个步骤，我们将其概括为 5 个重要的 CUDA API 调用，包括 cudaMalloc、cudaMemcpy(H2D)、cudaLaunch、cudaMemcpy（D2H）和 cudaFree。cudaLaunch 会触发内核函数启动。该操作会将内核函数的相关信息传输到 GPU，包括线程块的组织信息（网格和块的维度)、指针以及共享内存的分配等信息。我们测试了大量的应用程序，观察到内核函数启动的时间是在几十微秒的数量级。表 4.1展示了一系列我们测试的应用程序，可以发现内核函数启动时间从 3.3us 到 33.3us 不等。这个时间是从 cudaLaunch 被调用到内核函数信息被传输到内核函数管理单元，假设没有在流控制处理器中排队。这种较高的内核函数启动时间包括软件 API 调用和参数数据拷贝到 GPU 的内核函数等待池（Pending Kernel Pool）的过程。因此，这些开销会有很大差异。

此外，每个线程块的平均上下文切换时间从 1us 到 20us 不等，该时间取决于每个线程块的上下文的大小。从表 4.1中我们知道上下文切换的时间和内核函数启动的时间在同一个数量级。上下文切换的时间大致与内核函数启动的时间接近。这意味着如果我们完成第一个检查点备份，抢占请求可能已经启动。在这种情况，我们可以立即释放资源，为新的内核函数腾出空间。我们的设计不要求对内核函数启动时间的精确预测。这是因为如果一个检查点备份完成之后，真正的抢占请求还未到来。GPU 核心可以继续执行线程块直到抢占开始，在抢占时仅需存储更新的上下文。

如表 4.1所示，我们也发现平均线程块的执行时间再 1.5us 到超过 900us 不等。这种变化不定的线程块执行时间完全取决于内核函数的大小。短内核函数的线程块将被排空执行。这种方法几乎没有开销，同时满足抢占内核函数的延迟要求，因为排空执行的时间非常短。只有长内核函数的线程块需要进行上下文切换的操作。因为那些大的线程块的执行时间可以达到几百微秒，因此在内核函数启动的 API 被调用时粗略地预测一个线程块是否会被抢占并不难。为了实现预测的目的，我们设置内核函数启动的时间为 20us。当一个 cudaLaunch 被调用，我们比较预测的内核函数启动时间和每个线程块的剩余执行时间。如果预测的内核函数启动时间小于线程块的执行时间，则我们立刻采用检查点备份技术执行上下文切换。否则，我们将采用排空执行该线程块。注意到与线程块执行时间相比，内核函数启动的时间相对变化较小。因此，即使一个内核函数的真正启动时间不是 20us，也不太可能导致我们选择其他的抢占方法。

现实情况中，内核函数启动的时间可能会因为在流控制处理器中的队列等待而延迟。例如，之前的一个长的内存拷贝操作仍然没有结束，堵住了流控制处理器的队列。但是这个延迟在我们的算法中并不是个问题。在平均线程块执行时间远大于内核函数启动时间的情况，例如对于 CUTCP，这个延迟不太可能比线程块完成执行的时间还快，所以我们的检查点策略不会被浪费。而在平均线程块执行时间小于内核函数启动时间的情况，我们会选择排空执行，所以这个延迟也确定不会影响检查点备份的开销。

我们的预测方法将确定检查点备份的总次数不会超过两次。第一次检查点备份被 cudaLaunch 调用来触发，而第二次检查点备份被抢占请求触发。因此，检查点备份的开销是可以被控制的。

2）**时间估计**：我们可以估计上下文切换的时间和排空执行的时间来选择不同的抢占策略，即基于检查点技术的上下文切换和排空执行。我们还需要时间估计来预测抢占是否在当前线程块执行的时候发生。Chimera 采用时间估计的方法来比较不同的抢占方法的吞吐量开销。Chimera 估计每个线程块的排空执行时间为线程块剩余的指令数乘以该线程块之前的 CPI。上下文切换时间为每个线程块的上下文大小除以全局内存被每个 GPU 核心分享的带宽。但是这种基于线程块的方法在某些情况下并不准确。而当排空执行和上下文切换同时进行的时候，这个时间变得无法估计。如图 4.4所示，当一般的线程块在排空执行，而其他的线程块在做上下文切换时，估计的时间与实际的时间相差巨大。这是因为较少的排空执行的线程块意味着在流处理器（SP）较小的冲突，上下文切换的线程块在全局内存的带宽上出现更多的冲突。

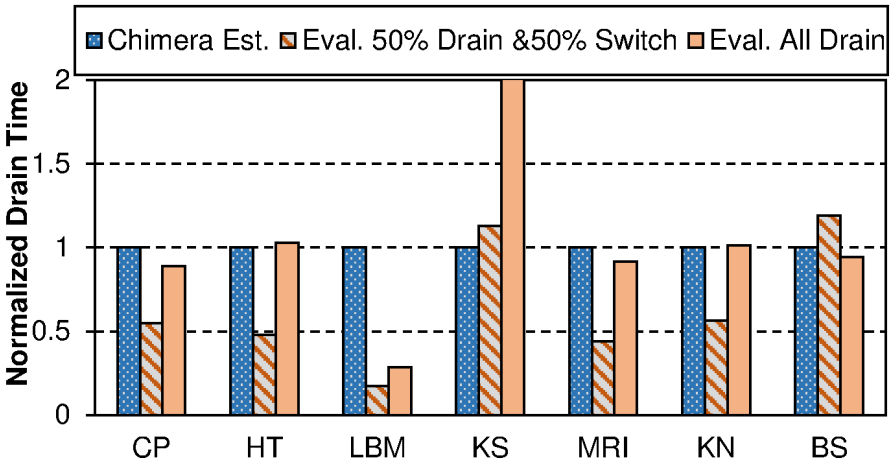


图 4.4　Chimera 的估计排空执行时间

虽然 Chimera 的估计对于全部线程块都做排空执行时更加精确，在某些特定情况仍然有非常不精确的地方。比如说，有的应用程序，如 LBM 和 KS 有多个阶段：它们的 CPI 在不同的时段不一样。在图 4.4中，KS 在开始的时候 CPI 非常低，但其 CPI 随着程序的执行而不断上升。因此，估计的时间和真实测量的时间相差很大。另外，Chimera 估计上下文切换时间来选择抢占策略时，只考虑单个线程块。例如三个线程块在做上下文切换，则需要传输备份的总上下文大小是原上下文大小的三倍。因此真正的尚晓雯切换时间要比 Chimera 估计的每个线程块上下文时间长 3 倍。

从表 4.1中可知，线程块的执行时间和线程块的上下文切换时间相差很大。对于大多数的应用程序，我们会选择排空执行所有的线程块或者上下文切换所有的线程块。因此，排空执行的时间和上下文切换的时间是可估计的。我们不需要担心上下文切换和排空执行的互相干扰。为了避免 CPI 随时间变化的影响，我们在线收集线程块的执行时间。由于不同的线程块执行的指令在大多数情况都是相同的，所以线程块的执行时间是相对稳定的。因此，我们可以采用之前收集的平均线程块执行时间减去已经执行的时间来获取线程块的剩余执行时间。所以，如果在我们需要估计执行时间的时候还未收集到线程块的执行时间，我们将采用 Chimera 的时间估计方法。为了估计上下文切换的时间，我们采用了最坏情况来估计，即估计的时间为当前 GPU 核心所有线程块被上下文切换出去的时间。由于上下文切换的时间范围小于线程块的执行时间，我们使用最坏情况来估计比较有保证。

4.5.3　上下文缩减

传统的上下文切换将所有分配的上下文存储都全局内存中。但是，在某一个特定的时间点活跃的上下文总是小于分配的大小，这使得我们需要存储备份的上

下文大小相对较少。我们采用脏位实时追踪活跃的上下文。但是，线程块主要有两种上下文，寄存器和共享内存。他们的声明周期是不同的。共享内存时每个线程块私有的。由于这是被程序员来管理的，我们将共享内存的生命周期当成线程块的生命周期。另一方面，寄存器是分配给每一个线程的，并且是在每一个 warp 中同时执行。因此，一个寄存器的生命周期与 warp 有关。当一个 warp 结束后，所有这个 warp 相关的寄存器将被完全释放。为了追踪寄存器的利用率，在协会的过程中，一旦一个寄存器被写入数据则需要置脏位为 1。当 warp 运行完成了或者我们采用检查点技术后将重置相应脏位。我们可以用类似的方法追踪共享内存写。

图 4.5展示了多个应用程序的脏寄存器的大小相比于被分配的大小。我们收集了不同执行阶段脏寄存器的百分比。我们的初始收集点是线程块执行的 25% 阶段。Dirty1 和 Dirty2 是脏寄存器在 50% 和 75% 线程块执行时间相比于初始收集点的百分比。对于一些包含了非常多指令的内核函数，MRI 到 PF，Dirty1 和 Dirty2 相比于初始状态，平均下降了 38.2% 和 48%。一般来说，Dirty2 相比于 Dirty1 有较少的脏寄存器，因为在 75

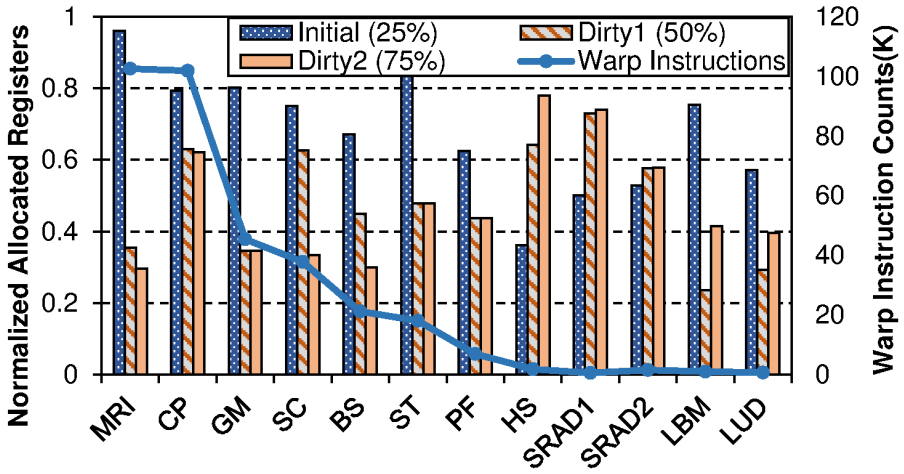


图 4.5　相对于分配的寄存器数量：Initial：线程块执行时间的 25%、Dirty1: 线程块执行时间的 50%、Dirty2：线程块执行时间的 75%

我们还在最后一次检查点备份中采用了本地上下文存储。因为新的内核函数可以采用旧内核函数没有使用的空间，本地上下文切换的方法得以采用。这将进一步降低实际的抢占延迟。

4.5.4　主动抢占方法设计

1）**检查点备份**：检查点备份只在大的被抢占的内核函数中被使用，因为他们排空执行的时间过长。当一个内核函数在多个 GPU 核心上运行时，如果一个

cudaLaunch 被调用，我们知道新的内核函数将要在几十个微秒内被传输到 GPU。在这个时候，GPU 驱动将发送一个信号来激活微程序化陷阱程序。这是通过命令队列和存储映射寄存器。当前的 GPU 通过一些可以被开发者直接访问的寄存器来激活抢占操作，但并不是被终端用户来激活。当检测到一个抢占内核函数启动的请求，则一个初始检查点备份命令被写入命令队列，之后会修改存储映射的寄存器来开始每个 GPU 核心的检查点备份。我们测量了 NVIDIA GTX 1060 GPU 的信号传输时间。这个延迟大约在 1.3us，对于不同的应用程序这个值也相对稳定。这个信号将触发一次检查点备份。我们将暂停取新指令，然后开始完成流水线指令。否则，在检查点备份上下文的时候，上下文也在持续变化。如果当前的内核函数是计算密集型的，完成流水线指令的过程需要几十个时钟周期。如果当前的内核函数是访存密集型的，我们必须等待访存请求返回来完成流水线上的指令。因此每个 GPU 核心中执行完所有流水线上的指令需要几百个时钟周期。第一个检查点需要备份的上下文是相对于初始状态的脏寄存器和共享内存。

当检查点备份完成后，所有的脏位将被重置。之后，GPU 检查是否有新的内核函数被传输到内核函数管理单元。如果内核函数在内核函数等待池，则一旦获得 GPU 核心资源则可以开始执行。则当前内核函数可以被立刻抢占，因为当前的执行状态已经被存储。否则，当前内核函数需要继续执行直到真正的抢占请求到来。当真正的抢占请求到来时，我们只需要存储变化的上下文。因为这一次检查点备份的上下文是相对于上一次检查点备份变化的部分，因此需要备份的上下文非常小，消耗的时间也短很多。因为只有变化的上下文需要被存储，没有重复的数据需要被存储。综上所述，当新的高优先级的内核函数的 cudaLaunch 被调用时，基本检查点备份将开始。因此，新内核函数很快被调用。所以，第二次检查点备份的上下文肯定会非常小。除此之外，由于本地上下文存储的利用，需要被存储的上下文的大小可以进一步减少。

被抢占的内核函数的恢复和传统的检查点恢复类似。如果我们有两次检查点状态需要恢复，则必须先恢复后一次的状态再恢复前一次的状态。但是，在这个时候 GPU 核心不允许执行任何任务。所以，所有的带宽将被用来做上下文恢复。

2)**在线选择器**：从表 4.1中我们可以知道，执行时间、上下文的大小还有内核函数启动时间对于不同的内核函数都不一样。因此，当 cudaLaunch 触发了我们的主动抢占机制，有许多种可能性。图 4.6展示了这些可能性：

（a）两次检查点备份：这是最常出现的情况。内核函数启动的时间比第一次检查点备份的时间长。当真正的抢占请求到来的时候，我们再将更新的上下文做一次备份。

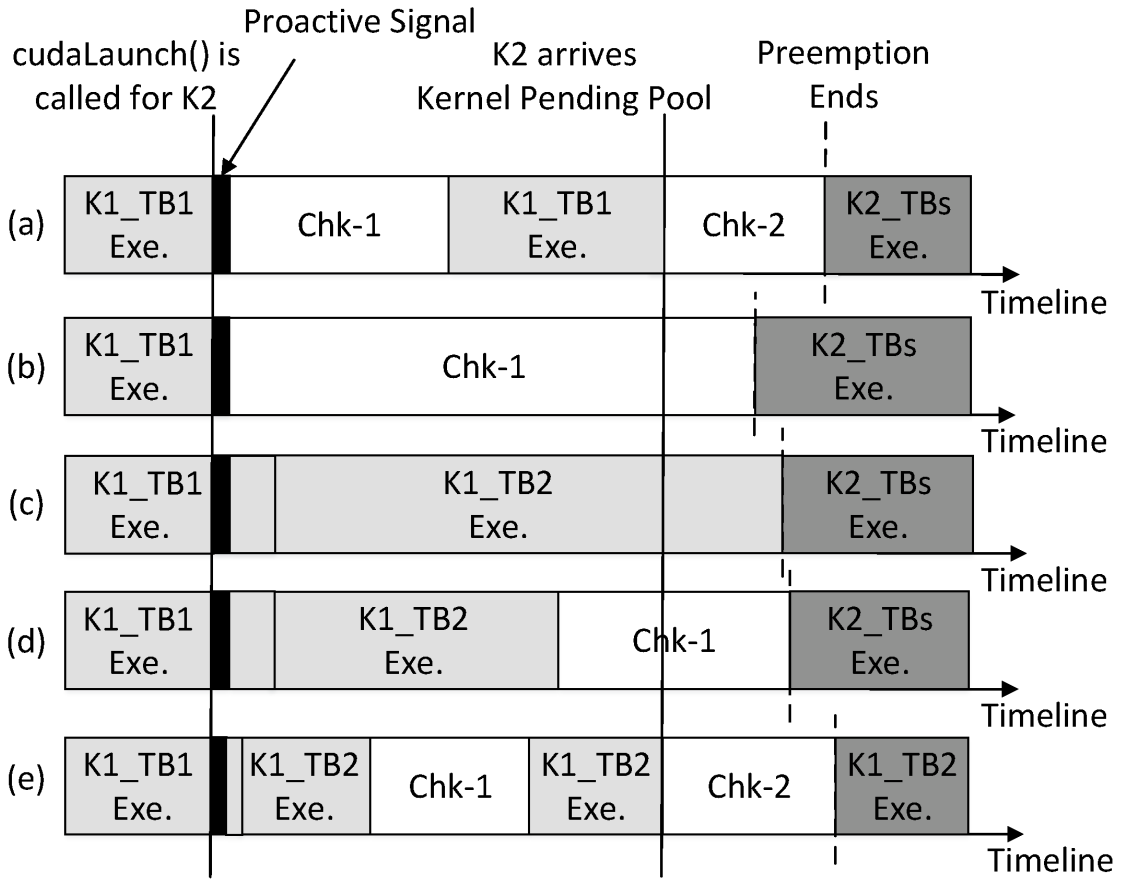


图 4.6　PEP 可能性分析：K1：被抢占的内核函数、K2：抢占内核函数、Chk-1：基础检查点备份、Chk-2：增量检查点备份

（b）单次检查点备份：这实际上和传统的上下文切换是一样的，但是这个检查点备份比传统的上下文切换更早发生。

（c）排空执行：这种情况被抢占的内核函数比较小。线程块的执行时间比抢占内核函数的启动时间要小，很可能会在延迟限制前完成。在这种情况，我们排空执行所有的线程块来达到极小的开销。

（d）排空执行然后单次检查点备份：被抢占的内核函数和（b）的情况一样。如果一个线程块接近执行完毕，则抢占不会发生在当前线程块的执行过程中。因此，我们会先排空执行线程块，然后当新的线程块被发送到这个 GPU 核心。新的线程块会开始执行一定数量的指令后开始检查点备份。本章我们设置这个指令的数量为 1000.

（e）排空执行然后两次检查点备份。被抢占的内核函数和（a）的情况一样。当 cudaLaunch 被调用时，线程块接近执行完毕，和 d 的情况类似。

我们设计了一种动态在线选择机制来处理这些可能性。图 4.7阐述了我们的在线选择方法。当抢占内核函数的 cudaLaunch 被调用，我们比较预测的内核函数启动时间与估计的当前线程块的剩余执行时间。如果线程块的估计排空时间比上

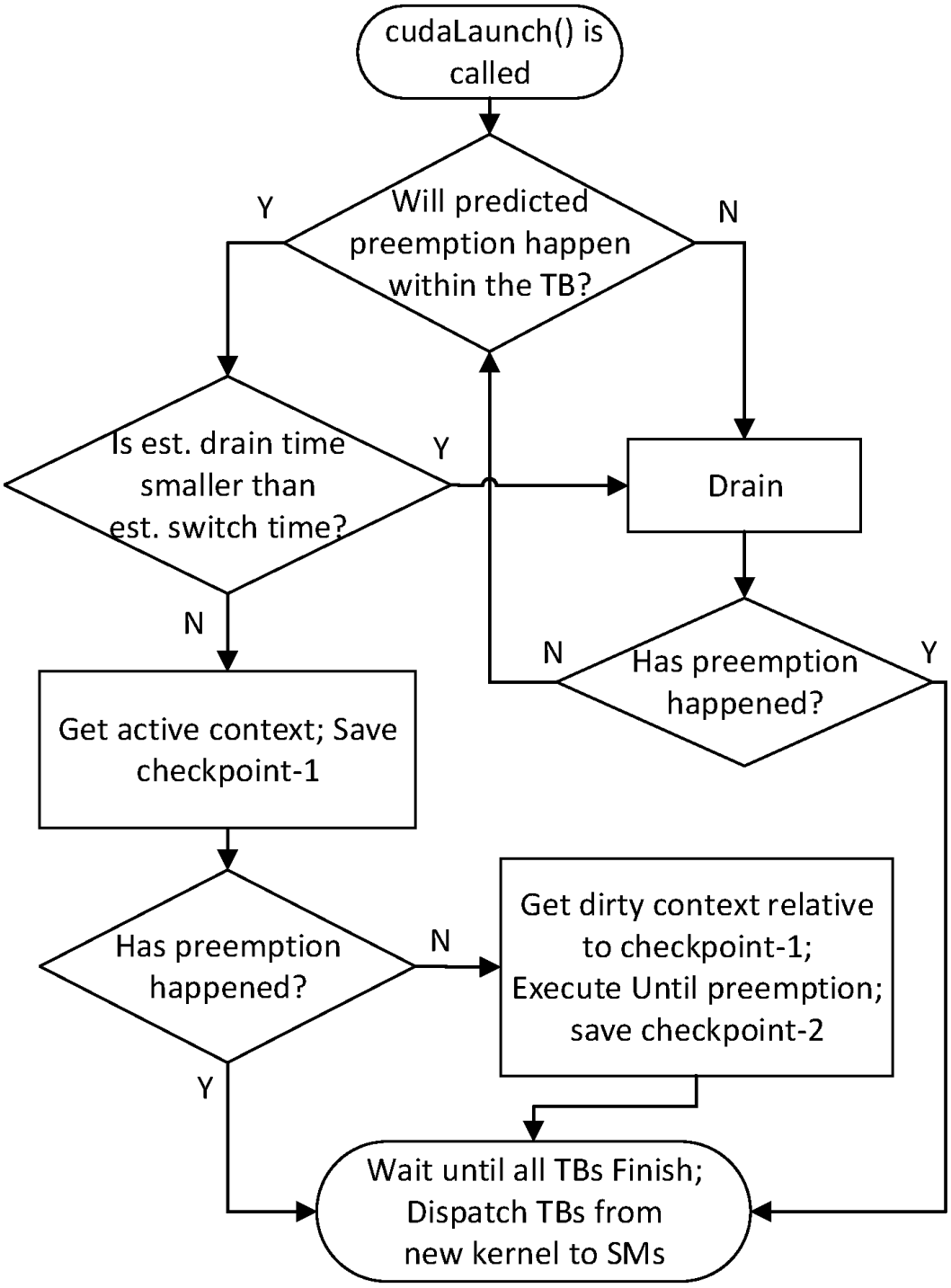


图 4.7　在线选择策略

下文切换时间长，我们定义该内核函数为大内核函数。（a）和（b）均处理的是长内核函数，收集活跃的上下文，并用检查点备份的方法存储到全局内存。对于其他的情况，内核函数预计的抢占点将不会在当前执行的线程块的生命周期中发生。因此我们先排空执行当前的线程块。之后当前内核函数分派新的线程块后再

配置	NVIDIA Geforce GTX980
GPU 核心数量	16
SIMD 宽度	32
SIMT 核心时钟频率	1216MHz
存储时钟频率	7GHz
访存控制器数目	4
调度策略	4 个 warp 调度器采用 Loose-Round-Robin 策略
寄存器大小	256KB
共享内存大小	96KB
每个 GPU 核心的线程块数目最大值	32

表 4.2　GPGPU-Sim 参数配置

重新做时间的预测和估计。如果新的新的线程块可以及时排空执行完毕，才发生抢占，则可以考虑（c）的情况。否则，我们考虑（d）的方案。这是 b 的另一种情况，而 e 是 a 的另一种情况。

4.5.5　硬件开销

为了实现 PEP，GPU 需要增加新的控制逻辑，主要为了实现以下部件：（1）时间预测和估计单元，主要包括一些用来收集数据的计数器和比较器来完成选择；（2）脏位，每个寄存器都需要一位，对于 NVIDIA GTX980 GPU，总计每个 GPU 核心需要 8KB；（3）分析计数器，该计数器用来收集线程块的执行时间。综上，PEP 最主要的硬件开销来源是脏位的存储开销。

4.6　实验

4.6.1　实验方法

我们在最新版本的 GPGPU-Sim 中实现了 PEP，以及 Chimera。系统配置信息总结在了表 4.2。配置中 256KB 寄存器和 96KB 的共享内存反映了当前 GPU 系统结构中较大的上下文。默认情况下，GPGPU-Sim 仅模拟 PTX 指令，这是一种不限制寄存器使用数量的汇编伪指令集。这并无法直接在硬件中执行，而 SASS 才是在硬件上执行的原生指令集。因此，我们采用一种能够和 SASS 一一对应的 PTXPlus 指令集进行模拟，这能够准确的模拟寄存器脏位。

为了比较，我们实现了多个不同版本的 Chimera 和 PEP：原生的 Chimera、采用脏上下文的 Chimera、原生的 PEP 以及采用了本地上下文备份的 PEP。我们测试了许多内核函数，这些内核函数来自于多个测试集的 GPGPU 应用程序，包括 NVIDIA Computing SDK、Parboil、Rodinia 和 Darknet 等。对于 Chimera，我们

设置了不同的内核函数延迟要求。我们观察到上下文切换的平均时间总是小于20.9us，因此我们将延迟要求分别设置为 5us、10us 和 15us。对于 PEP，我们设置了多个抢占内核函数的启动时间，预测的内核函数启动时间以及不同的抢占时机。PEP 的相关参数我们将在稍后讨论。我们在实验中分别比较了这些不同设计的抢占延迟，上下文大小，以及抢占开销。

因为 GPGPU-sim 不模拟从 cudaLaunch API 调用到内核函数在 GPU 真正开始的时机，我们设计了自己的实验方法。我们采用 NVIDIA profiler 来收集分析内核函数启动所需要的时间，这个时间如表 4.1所示，从 3us 到 33us 不等。我们之后设置抢占内核函数的启动时间分别为 5us、15us、25us 或 35us 不等。我们还设置了预测的内核函数启动时间为 20us 或 30us。此外，抢占可以发生在被抢占内核函数执行的任何阶段，因此我们设置了不同的抢占 cudaLaunch 调用的时机。为了试验的目的，我们将抢占内核函数的 cudaLaunch 的调用时间分别设置在被抢占内核函数的平均线程块执行阶段的 25%、50% 和 75%。因此，实验中每个应用程序将跑 24 次，取所有可能性的值的平均值。下面的实验结果均为以各参数运行一次后的平均值。

4.6.2 实验结果

4.6.2.1 策略选择分布

如图 4.8所示，我们收集了每个应用程序采用实时策略选择后所有线程块的策略分布。对于线程块运行时间长于 100us（表 4.1所示）的应用程序，所有的线程块均采用了两次检查点备份的抢占方法。相对地，所有选择排空执行所有线程块的都是小内核函数，即平均线程块执行时间较短。对于 LBM，线程块的平均执行时间为 30.1us，而平均上下文切换时间为 20.9us。他们的排空执行时间和上下文切换时间相差不大。这种相近性使得多种策略选择的出现，一些线程块需要被排空执行，而另一部分线程块则需要检查点备份技术，这都取决于被抢占的线程块的执行阶段。在单次检查点备份的情况中，由于省去了第二次检查点备份，我们节省了开销并降低了延迟。

由于对于大多数的内核函数，平均排空执行时间和平均上下文切换时间相差较大，我们大多数时候对所有的线程块选择一种抢占方法。仅选择一种方法意味着排空执行的线程块的上下文切换的线程块之间没有带宽竞争。因此，我们的延迟估计方法并不会收到访存冲突的影响。

4.6.2.2 抢占延迟

图 4.9展示了抢占延迟，该延迟测量的是从抢占内核函数到达 KMU 开始到最后一个线程块的上下文被存储。这也是内核函数等待池中抢占内核函数的真实等待时间。我们观察到后面 7 个内核函数排空执行每个 GPU 核心中所有的线程

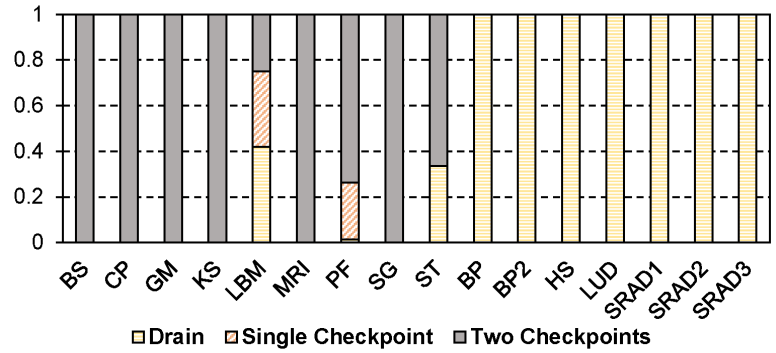


图 4.8　抢占技术选择分布

块并达到了非常低的延迟。这些应用程序的线程块执行时间也都相对较短。虽然 Chimera 的时间估计方法并不准确，但 Chimera 选择的策略和 PEP 一样。这是由于排空执行时间和上下文切换时间相差巨大，这并不要求非常高的精确度。于是，所有四种方法在排空执行时延迟相同，平均排空执行时间为 3.4us。但是，PEP 和 PEP+In-place 分别将总平均抢占延迟从 Chimera 的 8.9us 降低到 4.5us 和 3.6us。较短的抢占延迟使得内核函数能够满足更严格的延迟要求，也增加了多任务处理的可用性。

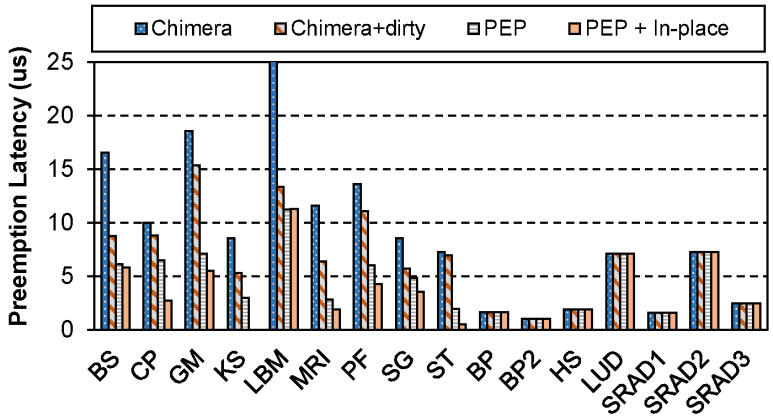


图 4.9　平均抢占延迟

图 4.10展示了上下文切换的抢占延迟。前面的 9 个应用程序不会选择排空执行所有的线程块。两个主要因素会影响他们的抢占延迟：流水线排空的时间和总的上下文大小。这其中，上下文大小为关键因素。与原生 Chimera 相比，脏上下文存储的 Chimera 能够降低 31.8% 的抢占延迟，因为该方法减少了需要存储的上下文大小。而与原生 Chimera 相比，PEP 能够将平均抢占延迟降低了 58.5%，而采用了本地上下文存储后，PEP 能进一步将抢占延迟降低 70.3%。对于 Kmeans

（KS），PEP-In-place 可以达到 0 抢占延迟，因为脏上下文的大小非常小，在第二次检查点备份中可以完全存储于本地。

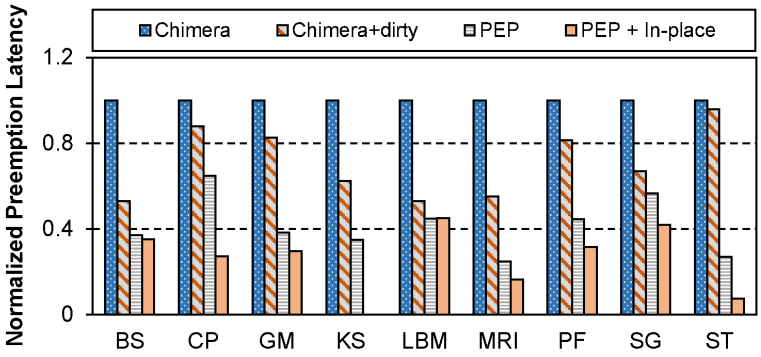


图 4.10　上下文切换应用程序的平均抢占延迟

4.6.2.3　上下文大小缩减

在实验中的上下文，我们只考虑寄存器和共享内存。共享内存位于片内，因此访存开销非常小。程序员采用共享内存的目的是为了减少全局内存访问的开销，经常访问的数据将被放进共享内存。因此，共享内存的脏位经常更新，开销较大。我们仅将脏位应用于寄存器。大多数情况都将存储所有分配的共享内存。在我们实验结果中，“上下文大小”指的是必须存储到全局内存的上下文。

图 4.11比较了不同设计需要存储的上下文，只存储脏上下文，Chimera 每个线程块的平均上下文大小可以被减少 6KB，即平均总上下文大小的 34.4%。因为 PEP 可能会存储检查点状态的上下文两次，PEP 平均总上下文会比 Chimera+dirty 要大. 但是这几乎和原生 Chimera 一样。然而，PEP-in-place 可以进一步降低 16.2% 上下文大小。

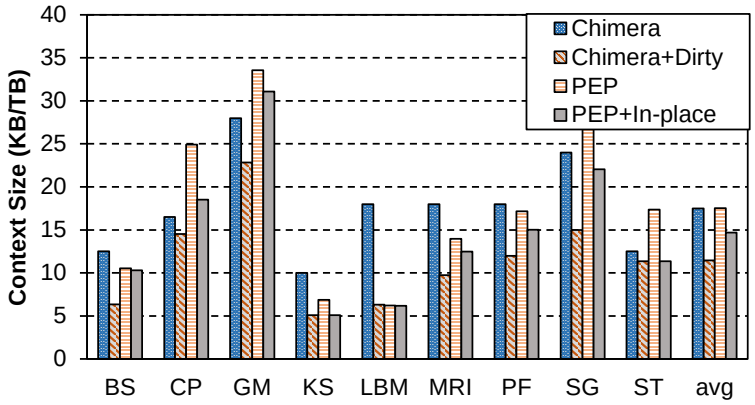


图 4.11　上下文大小比较

图 4.12和图 4.13分别展示了 PEP 和 PEP-in-place 的上下文大小的细节。对于选择上下文切换所有线程块的应用程序，总上下文大小为第一次检查点备份和第二次检查点备份的上下文大小之和。实验结果显示第二次上下文的大小平均只有第一次上下文大小的 56.1%。而本地上下文存储可以进一步降低第二次检查点备份需存储的上下文。图 4.13的实验结果显示第二次检查点备份的上下文大小平均为每个线程块 3.34KB，是第一次检查点备份的上下文的 29.4%。我们可以看到两次检查点备份方法大大减少了上下文的大小。

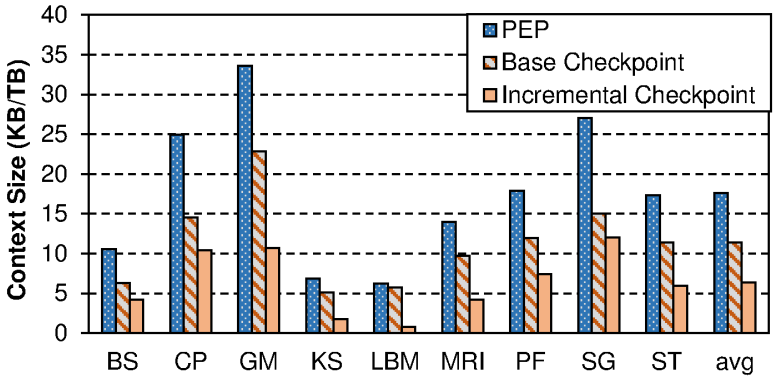


图 4.12　需要备份到片外内存的每个线程块的上下文大小

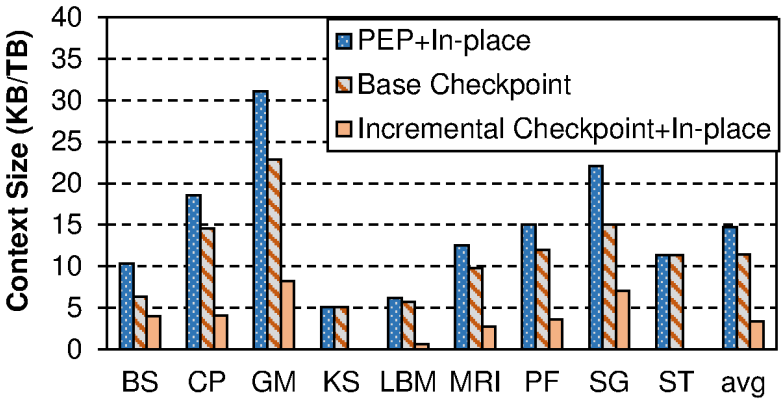


图 4.13　需要备份到片外内存的每个线程块的上下文大小（采用了本地上下文备份技术）

4.6.2.4　性能敏感分析

在这一节，我们测试了带宽和可扩展性对性能的敏感性。

我们保持内存分区的数量和带宽，调整不同的 GPU 核心数量来分析他们对于 PEP 策略的抢占延迟的影响。图 4.14展示了抢占延迟随着 GPU 核心数量的增加几乎线性增加，这是因为访存流量不断升高。如图 4.14所示，32 个 GPU 核心的平均抢占延迟是 16 个 GPU 核心的平均抢占延迟的 2.58 倍，这说明检查点策略对于

存储带宽非常敏感。因此，实验结果进一步说明所有的带宽都提供给上下文切换相比于将执行和上下文切换混合在一起，抢占性能更好。

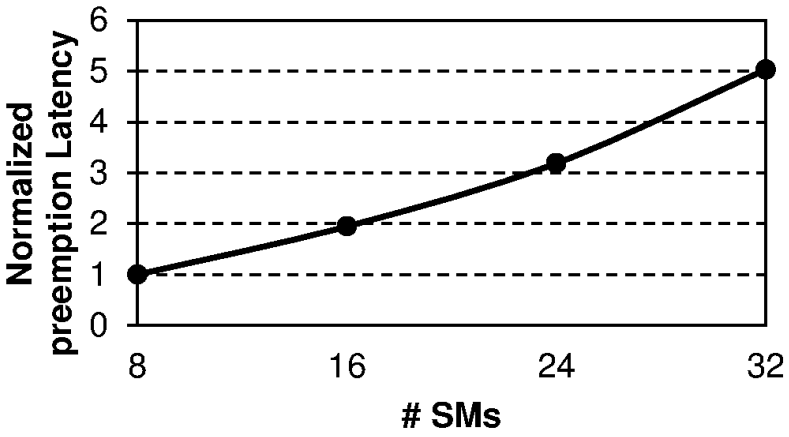


图 4.14　GPU 核心数量对于抢占技术的性能影响

4.6.2.5　抢占开销分析

本章所指的抢占开销是指由于抢占导致的程序执行的停滞。抢占开销的实验结果如图 **??**所示。当 GPU 核心在做上下文切换，切换的线程块必须停止取新指令和执行。GPU 核心停滞等待上下文备份和上下文恢复。上下文备份和上下文恢复的唯一不同是上下文备份前排空流水线的延迟。因此，我们只比较每个线程块的上下文切换延迟作为开销。图 **??**显示第一次检查点备份相比于 Chimera 开销降低了 37.9%。PEP 的开销和采用了脏上下文的 Chimera 的开销类似。两次检查点备份的 PEP 开销仍然比 Chimera 降低了 6.3%。当采用了本地上下文备份能够降低第二次检查点备份的上下文大小，PEP 的平均开销经一部降低了 16.4%。一些应用程序相比于 Chimera 开销可能更高。在这些情况中，寄存器的重用率不高，所以脏上下文的大小更大。此外，采用了两次检查点备份，则更多的时间需要用来排空 GPU 核心的流水线。但是，上下文的大小和上下文切换的开销是正相关的，一些应用程序向 LBM 和 ST 由于脏上下文大小较小，因此节省的开销大于 50%。

4.7　本章小结

在本节，我们介绍了 PEP，一种 GPU 动态主动的抢占机制。只需要一个粗略的抢占内核函数启动时间的预测，我们可以在抢占请求到来之前做好准备。我们借鉴了容错中用到的检查点备份机制，允许我们缩短抢占延迟。更重要的是检查点备份技术能够容忍不精确的预测。为了预测抢占的发，我们利用了 GPU 驱动的内核函数启动的过程。GPU 驱动在收到 CPU 的内核函数启动的命令后会触发第一次检查点备份。这允许我们在真正的抢占请求到来时，只需要再备份相对于第

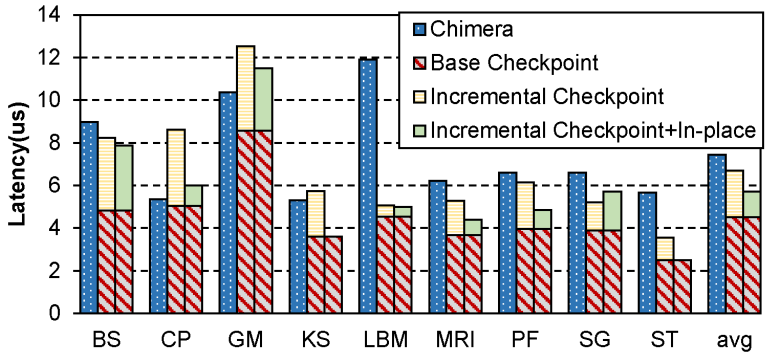


图 4.15　上下文切换开销

一次检查点备份变化的部分。而 GPU 核心可以在两次检查点备份之间正常执行。我们同时支持对小内核函数的 GPU 核心排空执行，还有本地上下文存储的功能来达到最小的抢占开销。对于我们的主动检查点备份的抢占机制，我们平均降低了 58.6% 的抢占延迟和 23.3% 的上下文切换开销。平均抢占延迟也可以被降低到 3.6us，使得抢占应用程序能够满足更严格的延迟要求，更好地支持多任务。

第五章　一种动态延迟感知的 2.5 维堆叠片上网络负载均衡策略

5.1　研究背景

近年来，基于硅中介层的堆叠，即 2.5 维堆叠 [10] 越来越受到业界的重视 [11]。如图 **??**所示，2.5 维堆叠技术将多个晶粒一起堆叠在硅中介载体上。由于三维堆叠是一种革命性的技术，需要全新的设计、测试方法以及工具，2.5 维堆叠则是一种相对渐进的技术。2.5 维堆叠技术可以避免很多三位堆叠技术面临的挑战，并且能够被当前的设计工具链支持。此外，当前的堆叠存储系统为每个堆叠存储单元提供了固定的带宽和容量标准。相比于三维堆叠技术，硅中介层有足够的面积来集成更多的堆叠存储单元，能够达到更高的总存储带宽和更大的存储容量。因此，近几年的商业化 2.5 维堆叠产品已经开始变得越来越流行。

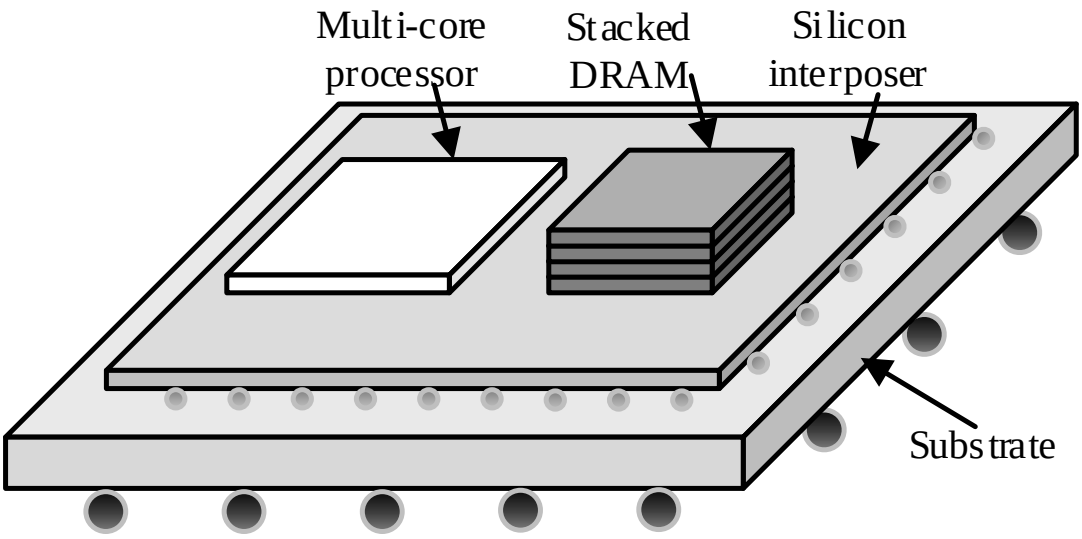


图 5.1　2.5 维堆叠技术

为了更加高效的满足 2.5 维堆叠系统的通信要求，之前的工作显示硅中介层上有足够的连线资源，这些连线资源可以进一步被开发以实现另一层片上网络。当 2.5 维堆叠硅中阶层系统上存在两层片上网络时，比较可取的方式是将核间协议的通信和访存通信划分开。因为这两种通信特性非常不一样，将这两种通信划分到不同的网络层可以得到大量的收益。

但是，我们用 PARSEC 测试集在基准 2.5 维堆叠结构上测试，发现访存通信平均仅占全部通信的 14.8%。这些应用程序的核间通信量远大于访存通信量，出现极大不均衡性。核间通信的网络层出现拥塞，而另一层则未被充分利用。这将

导致较差的性能。因此，需要一种 2.5 维堆叠片上网络的负载均衡策略来平衡两个网络层的负载，并将资源利用率最大化。

缓存感知和延迟感知的方法是比较通用的均衡多层网络负载的方法。传统的缓存感知的方法通过邻居节点的缓存占用率来收集和判断拥塞信息。但是，这种局部的拥塞信息无法完全反映全局的拥塞状态，因此并不适合用于指导 2.5 维片上网络的网络选择。我们知道，基于目的节点检测的延迟感知方法是目前 2.5 维片上网络唯一的负载均衡策略。它在目的节点收集每个报文的传输延迟来检测该节点的拥塞信息，并为该节点发出的报文选择网络层。但是，这种策略采用的是不精确的信息，因为对于每个节点报文接收的路径和报文发送的路径是不一样的。该方法采用了报文接收的路径的拥塞情况来代表报文发送路径的拥塞情况。如图 **??**所示，节点 10 收集的接收报文延迟，即绿色路径作为拥塞信息。但是，该节点的发送的报文是沿红色路径传输的。

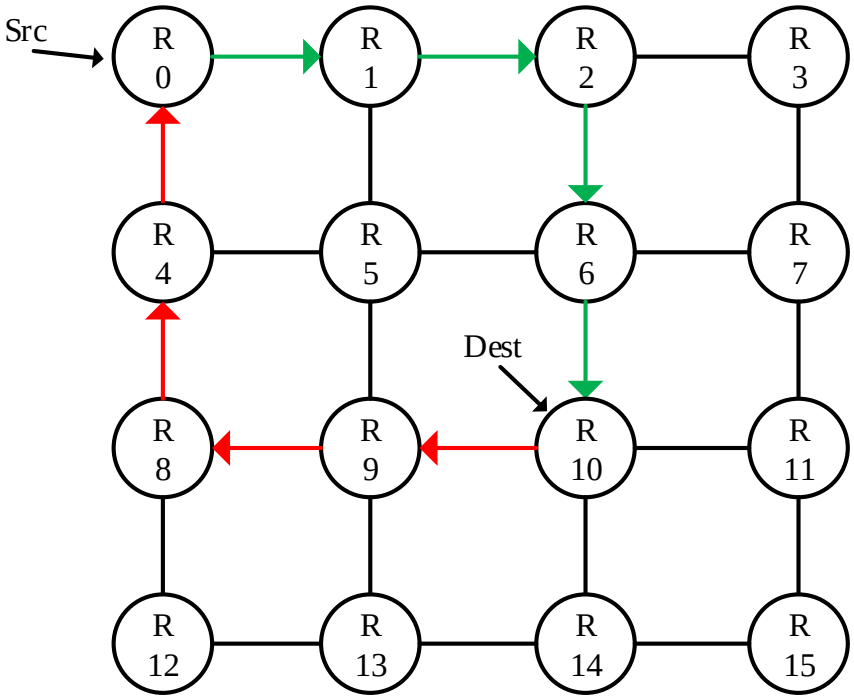


图 5.2　目的节点检测的负载均衡策略（采用 XY 路由算法，绿线为拥塞信息收集的路径，而红线则是节点 10 发送报文到节点 0 所走的路径。在这种情况下，从绿色路径收集的拥塞信息会被当做红色路径的拥塞信息提供给源节点 10 作为拥塞控制依据。）

我们需要研究合适的拥塞检测机制来获取准确的网络状态，并做出高效的网络选择。这是实现 2.5 维片上网络负载均衡的基础。

在本文中，我们提出了 2.5 维片上网络结构的动态延迟感知负载均衡策略（DLL）。DLL 策略首先收集每个目的节点的拥塞信息。然后，通过我们设计的延

迟传输环网，将拥塞信息传输回源节点。最后，每个源节点通过收到的拥塞信息选择网络层来传输报文，我们的 DLL 策略做出了以下几个主要贡献：

我们评估并分析了 PARSEC 测试集的通信流量，发现 2.5 维片上网络的通信流量在两个网络层极不公平。这急需一种高效负载均衡策略。

我们分析了当前负载均衡策略的缺点。缓存感知的方法无法获取全局网络的拥塞状态。当前的延迟感知方法不能准确的检测全局拥塞状态。

我们提出了一种 2.5 维片上网络的动态延迟感知的负载均衡策略。这个策略在目的节点收集拥塞信息，并将这些拥塞信息传输回源节点用于网络选择。我们设计了一个多链路无阻塞的环网用于传输这些拥塞信息。

与没有负载均衡的基准 2.5 维片上网络结构，当前的延迟感知方法以及缓存感知方法相比，我们的策略两个平均提升 45%、14.9% 和 6.5% 的吞吐量，而开销则非常低。

全文的组织结构如下，我们在第二节介绍了相关工作，第三节介绍了我们的策略设计，包括目标系统，设计原理和设计细节。在第四节，我们展示了试验结果。在第五节，我们更除了更多的讨论。最后我们在第六节作了总结。

5.2 相关工作

由于 2.5 维堆叠技术展现出在带宽、延迟和开销上的突出优势，近些年来一系列 2.5 维堆叠的工业街产品被推出。第一个发布的 2.5 维堆叠商业产品是 Xilinx Virtex-7 2000T FPGA, 该平台是由四块相对较小的 FPGA 堆叠在硅中介层上形成的。High Bandwidth Memory (HBM) 和 Hybrid Memory Cube(HMC) 是两个比较先进的堆叠存储技术，它们已经被广泛应用到 2.5 维堆叠的处理器中。当前，他们能够提供每个堆叠单元 128GB/s 的带宽。AMD Radeon R9 Fury Xnew GPU 在其 2.5 维堆叠结构中采用了 HBM 作为存储单元以提供超大的带宽和容量。NVIDIA PASCAL GPU 体系结构则采用了第二代 HBM。2.5 维堆叠技术和第二代 HMC 也将被应用到下一代的 Intel Xeon Phi 协处理器中。

Enright Jerger 等人的工作是第一个提出开发基于硅中介层的片上网络互联设计空间的。他们比较了几种拓扑逻辑结构，提出了一种负载均衡策略。但是，他们的策略收集目的节点的拥塞信息（延迟），并将目的节点的拥塞信息用于指导目的节点的报文发送及网络层的选择。由于报文的发送路径和接收路径并不相同，这个策略采用的实际上是不精确的网络状态信息进行网络层选择。这个策略在本文中将被称为 DestDetect。

在传统的二维片上网络中，关于拥塞控制的工作非常多。Li 等人提出 DyXY 路由机制基于本地网络拥塞状态。相对于静态路由方法，该方法可以达到更高的性能。RCA 是第一个采用了本地和非本地信息来实现负载均衡的工作。但它在拥

塞计算上引入了干扰和冲突。DBAR 通过在收集拥塞信息时保持动态独立解决这个问题。所有这些方法都应用了缓存利用率的统计作为拥塞信息。而缓存利用率作为拥塞信息更适合路由路径的选择而不是网络层的选择。

在三维片上网络结构或者是多片上网络结构的负载均衡问题上，Ranmanujam 等人提出了一种高效的层多路复用（LM）三维结构。他们通过在垂直方向上的多路复用和多路分解结构，替代了传统三维片上网络结构上的一层一跳步的通信方式。在负载均衡逻辑中，每一对输入输出端口采用了一套切片计数器进行拥塞判断和网络层的选择。这种拥塞信息反映了网络层的通信流量，但网络层中的热点区域无法被检测到。Catnap 是一种能效比的多网络结构，该结构的子网络可以通过门控时钟（Power gating）技术关掉，不需要考虑网络连接性。他们实现了一种基于拥塞状态观察的子网选择策略。通过计算每个路由器输入缓存的缓存占用率来检测拥塞。然而，这种方法不能反映全局拥塞状态，在本文中该拥塞检测和网络选择策略我们称为 LocalBuf 策略，将与我们提出的 DLL 策略进行比较。

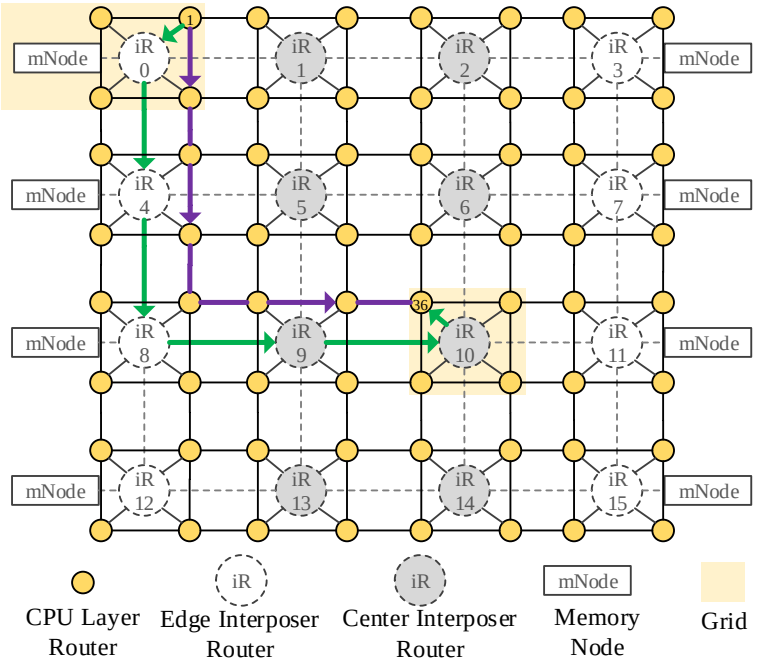
5.3 设计

在本节，我们首先介绍了 2.5 维堆叠片上网络的目标系统。之后我们分析了片上网络的通信和拥塞特性来解释我们的设计动机。最后我们从两个方面介绍 DLL 策略，一方面是拥塞控制和网络选择策略，另一方面是延迟传输环网。

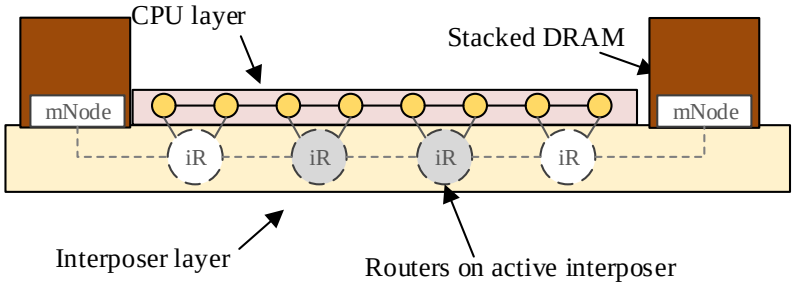
5.3.1 目标系统

DLL 策略可以广泛地应用到基于硅中介层的系统和多网络层系统的选择，因此本文展示了一个例子以更方便具体的呈现我们的策略。我们的 2.5 维堆叠硅中介层系统集成了 64 核的 CPU，其周围集成了 4 个堆叠 DRAM 存储器。在这个系统中包含两个网络层，上面一层是 CPU 层，而下面一层是硅中介层。这两个网络层通过 TSVs 和 μbumps 互连。如图 5.3所示，我们的 2.5 维堆叠片上网络结构，在 CPU 层的拓扑逻辑是 MESH。由于 μbump 会引入较大的开销，在硅中介层，我们采用了集中式 MESH 以减小路由器数量。在硅中介层中共有 16 个节点，每个节点连接 4 个 CPU 节点。在硅中介层网络的两侧总共有 8 个访存控制器。每个方寸控制器包含两个 DRAM 通道，连接到相邻的硅中介层节点。图 5.3也展示了两种类型自的硅中介层实现，包括主动式和被动式。相比于主动式的硅中介层，被动式的硅中介层不包含有源器件（逻辑、门)。所有的硅中介层路由器都实现在 CPU 层。短期来看，被动式的硅中介层仍然是一种比较现实的方案，而主动式的硅中介层则更像是一种三维堆叠的方式。

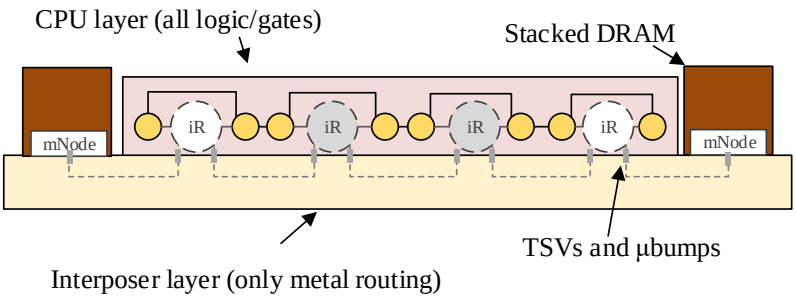
在目标系统中，包含两种类型的通信，核与核之间的通信以及核与存储之间的通信。之前的研究表明，将这两种通信划分到不同的及网络上会带来许多优



(a) 顶视图



(b) 侧视图（有源硅中介层）



(c) 侧视图（无源硅中介层）

图 5.3　2.5 维片上网络结构的目标系统

势。核间协议通信在 CPU 层传输，而核与存储之间的访存通信则可以在硅中介层上进行。但是，这种片上网络通信的划分并不是非常严格的。这依赖于应用程序

的需要以及真实片上网络结构的应用。在下一节，我们将介绍我们的负载均衡策略。该策略基于拥塞状态来划分片上网络的通信流量的传输层次。

5.3.2 设计原则

我们分析片上网络的通信和拥塞特性。这些特性对于设计一个负载均衡策略尤为重要。根据通信和拥塞特性，我们也给出了 2.5 维堆叠片上网络结构负载均衡策略的两个设计原则。

通信特性：许多应用程序（包含 PARSEC 和 SPLASH-2 测试集），都呈现出相对不均衡的特性。在空间特性上，多数的通信流量都是在几对节点之间进行的，大多通信报文都是在几对节点之间传输，与合成流量模型的 permutation 流量模式类似。因此负载较重的流量的通信路径可以通过报文延迟清晰地反映出来。

设计原则 1：以每个路由器的缓存占用率作为拥塞度量，可以反映路由器的拥塞状态。每个节点只能检测到本地拥塞的状态。相反，最近的几个报文的平均通信延迟可以反映出网络相应路径的拥塞情况。因此，网络层中的全局拥塞状态可以通过延迟感知的方式被更准确的检测到。我们采用延迟感知的方法进行拥塞检测而不是缓存感知的方法。

拥塞特性：片上网络的通信呈现出类似 Permutation 通信模式的特点。较重负载的通信路径会很容易变得拥塞，而负载较轻的通信路径则会未被充分利用。对于负载较重的通信路径经过的节点，整个网络层是拥塞的；而对于负载较轻的通信路径经过的节点，整个网络层是未被充分利用的。因此，从不同的节点观察整个网络的拥塞情况，同一个网络层可能是拥塞的也可能是不拥塞的。

设计原则 2：由于不同的节点可能观察到同一个网络层不同的拥塞状态，我们需要采用源节点观察到的拥塞信息用于同一节点的网络层选择。如果该信息用在了目的节点的报文发送上，不准确的拥塞路径将被反映出来。如果从全局的角度来看，这种整个网络层不均匀的拥塞状态无法被直接反映出来。网络层中较轻负载的路径可能会被认为是拥塞路径，反之亦然。因此，网络选择最好是在源节点做决定，而不是目的节点或全局视角。

在本文中，我们利用了延迟感知的方法来检测全局网络层的拥塞，并为 2.5 维片上网络层中选择合适的网络层进行传输。

5.3.3 拥塞检测和网络选择

在设计原则的指导下，我们开发了 DLL 策略。该策略采用最近几个报文的延迟作为网络拥塞的衡量指标。在源节点比较两层网络的平均报文延迟用于网络选择。如图 5.3（a）所示，如果节点 1 发送报文给节点 36，节点 1 首先要比较从节点 1 发送到两层网络的报文的平均延迟。如果上层网络的平均延迟大于下层网络

的延迟且超出一定阈值，则报文会被路由通过绿色路径达到负载均衡，而不是紫色路径。

从技术上看，我们的 DLL 策略需要实现 3 个关键问题：1）如何在路由器记录报文的延迟？2）如何产生并传输拥塞信息？3）如何根据拥塞信息作出网络层的选择？

有我们的 DLL 策略主要包含四个设计的细节。

首先，由于因为报文传输距离较远时，通信延迟肯定长于较短距离的传输，因此采用报文的总传输延迟并不公平，并且可能会误导我们做出错误的网络选择。我们比较不同报文的单跳平均延迟来选择网络层。延迟信息会被记录在每个报文的头切片上。

第二，如果我们传输所有报文的延迟信息回源节点，其开销是无法容忍的。因此，我们采用了粗粒度的方法。网络会被划分为多个格子，每个格子包含多个节点，同一个格子内的节点的拥塞信息将被合并以降低开销。如图 5.3（a）所示，如果节点 1 发送报文到节点 36，我们的粗粒度策略认为该报文是从格子 0 发送到格子 10。我们将连接着同一个硅中介层节点的 4 个 CPU 节点视为一个格子，如图 5.3（a）显示的黄色块描述的。边界的存储节点也被包括在格子内。格子的序号和硅中介层节点的序号是一样的。

第三，因为所有的存储控制器都分布在硅中介层水平方向的两侧，2.5 维堆叠下层网络的边界可能称为瓶颈。我们采用 YX-Z 路由代替 XY-Z 路由来降低边界节点的通信压力。通过这个方法，报文会较少的经过边界的路由到达目的存储控制器。

第四，我们的 DLL 策略可以消除协议层和网络层的死锁。协议层死锁可以通过虚通道来避免，而网络层死锁则通过维序路由来消除。即使报文会从一个网络层传输到另一个网络层，我们依然能够保证无死锁，这是因为 Z 跳一定是第一跳步或者最后一跳步，不可能形成环。

拥塞检测和网络选择总共有 5 个步骤，如图 5.4所示。

第一步在目的节点记录报文延迟。我们为每个报文增加了一个域记录延迟。延迟可以通过每个路由器的时钟来收集。一旦头切片准备完毕可以发送，该域按如下公式进行更新：

$$L = \frac{L_{last} \times hop + t_{out} - t_{in}}{hop + 1}, \tag{1}$$

其中 L_{last} 是上一跳之前的平均跳步延迟。L是当前跳步之前的平均跳步延迟，hop是当前跳步数，t_{out} 是本地路由器发送出该报文的时间，t_{in} 是该路由器收到该投切片时的时间。当收到一个报文时，目的节点首先计数该报文的每一个跳步的

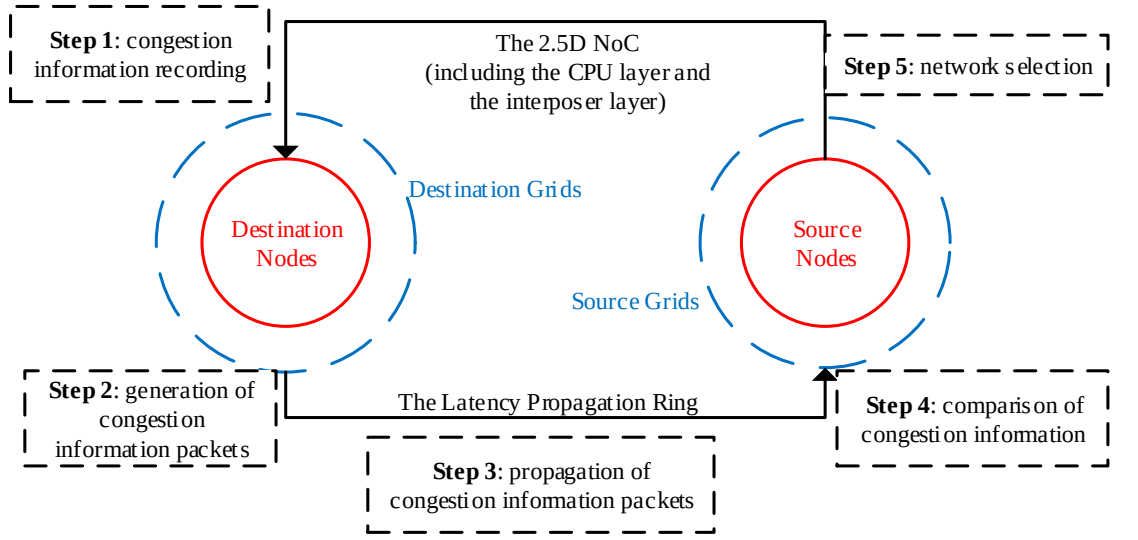


图 5.4　DLL 策略的工作流程

平均延迟。如果平均延迟大于 *15*（*2*‘*b1111*），则将被记录为 *15*（*2*‘*b1111*）。我们相信每跳步 15 个时钟周期已经足以表明其拥塞状态。由于每个跳步的延迟只需要 4 位来表示，因此计算将非常简单和快速。

第二步是在目的格子产生包含拥塞信息的报文。三种类型的数据将被打包存入仅需 9 位的拥塞信息报文，如图 5.5所示，包括了源格子，平均跳步延迟以及传输层。传输层被编码为 0 代表上层，编码 1 代表下层。目的节点发送拥塞信息报文到属于同一格子的硅中介层节点。如果硅中介层的节点的缓存满了，新产生的拥塞信息将被直接丢弃。我们采用的粗粒度方法，并不会严重影响拥塞信息收集的准确度。

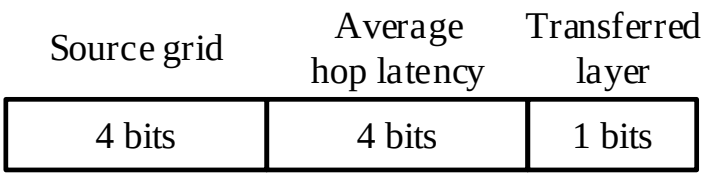


图 5.5　拥塞信息报文

第三步是传输拥塞信息报文回到源格子。我们设计了一个完全独立的环网来传输这些报文。这将在后面详细介绍。

第四步是计算核比较源格子的每层网络的平均延迟。根据传输的网络层，源格子将拥塞信息报文存储到两个特定的 FIFOs 中。这个 FIFO 有 5 个空位，采用了最近的 5 个报文的延迟来计算每个网络层的平均延迟。根据我们的实验，这五个报文的延迟足够反映出网络的拥塞状态。由于网络选择是在每个源节点进行的，延迟比较也是在每个源格子进行的。

最后一步是在源节点选择合适的网络层来传输报文。我们比较了之前步骤计算的每层网络延迟，然后比较结果被发送到该格子的所有节点。DLL 策略动态地发送报文到较轻负载的网络层。如果 CPU 层的平均延迟超过硅中介层的平均延迟到一定阈值，硅中介层并不拥塞（硅中介层的平均延迟小于一定阈值），报文发送到 CPU 层后将被转到硅中介层，继续传输直到报文到达目的节点所在的格子时。当报文达到目的节点所在格子时，该报文将被发送回 CPU 层。在其他情况，每个源节点将发送报文到其默认的网络层。

延迟传输环网为了能在源节点进行网络层的选择，拥塞信息报文必须从目的节点传输回源节点。如图 5.6所示，我们设计了一个延迟传输环网连接着所有的格子以传输这些拥塞信息报文。这个环网相对独立于当前的网络，因此没有任何的相互干扰存在的同时非常高效。这个独立的环网必须满足两个目标。第一个目标是高带宽。一个高带宽的网络允许更多的拥塞信息报文被发送回源格子以获取更精确的拥塞状态信息。第二个目标则是低开销。由于环网引入了的是额外开销，因此这些开销必须足够低。为了满足这两个目标，我们将延迟传输环网设计成多链路无阻塞的形式。由于环网实现在硅中介层，其面积和连线资源都未被充分利用，环网的开销对于整个系统的影响非常小。

为了获取更高的带宽，我们的延迟环网由四个 7 位的链路组成。报文的传输链路由源格子序号（拥塞信息报文的目标格子）的高两位确定。图 5.7展示了一个格子 0 的路由器的例子。如果一个拥塞信息报文被注入格子 0，其路由单元会首先选择链路 0 来传输报文，这是基于源格子 ID 的高两位（*2‘b00*）来确定的。开始传输后这高两位家昂被丢弃，只有拥塞信息报文剩下的 7 位数据需要在环网的链路上传输。如图 5.6和图 5.7展示的，被发送到不同颜色格子的拥塞信息报文采用的传输链路也不一样。通过 4 条链路，更多的拥塞信息报文可以并行传输。

为了降低硬件开销啊，延迟传输环采用了无缓存路由器结构。在每个路由器并没有缓存。我们在路由器的每条链路上插入一个寄存器，只需要两个时钟周期就能完成一跳步的拥塞信息报文传输。一旦拥塞信息报文被注入环网，报文将无阻塞地传输到源格子。为了避免网络层死锁，环网上传输的报文优先级必须高于等待注入的报文的优先级。换句话说，拥塞信息报文无法在上游路由器有报文的时候注入。

传输拥塞信息报文引入一些延迟。因为我们提出的延迟传输环网是无阻塞的，这个延迟并不会太差，拥塞信息能够及时的反映当前网络的状态。

5.4 实验

我们采用了时钟精确的互联网络模拟器 BOOKSIM 进行实验评估。我们修改了 Booksim 来实现我们的 2.5 维堆叠片上网络结构，通过几种合成流量模型来评

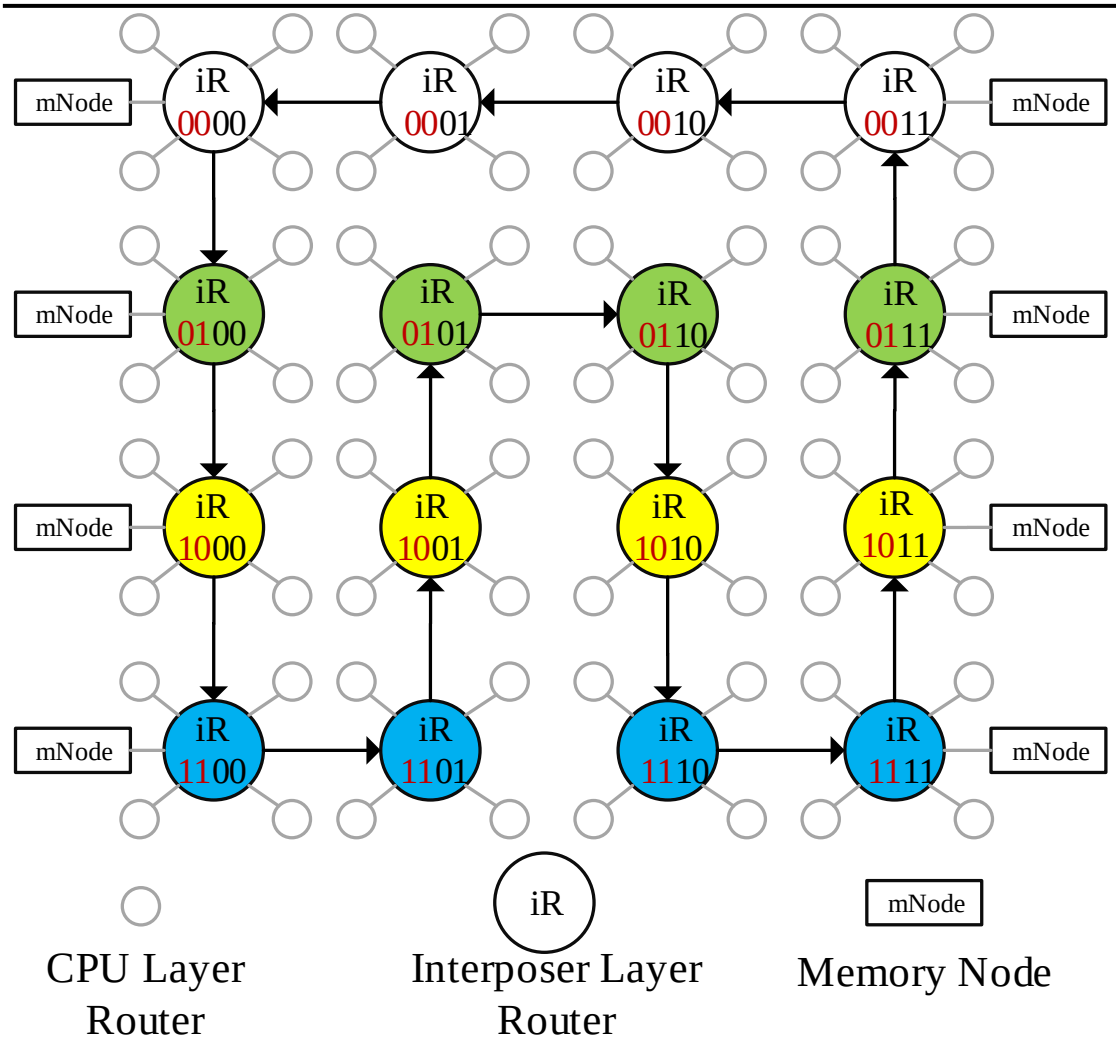


图 5.6　拥塞信息报文

估不同的负载均衡策略。为进一步通过真实应用程序验证实验结果，我们运行了 Netrace 来评估 PARSEC 的性能。我们采用 DSENT 进行面积和功耗模拟。为了更高效的评估我们提出的 DLL 策略，我们首先分析了 2.5 维片上网络结构的瓶颈，因此可以更加深入理解负载均衡策略的性能潜力。我们还分析了相比于 XY-Z 路由，YX-Z 路由的优势。第二，我们比较了 DLL 策略和没有负载均衡设计的基准设计以及其他两种负载均衡策略。最后，我们讨论了 DLL 策略的硬件开销。

5.4.1　性能瓶颈分析

由于 2.5 维堆叠片上网络结构在 CPU 层采用了 MESH 拓扑结构，在硅中介层采用了集中式 MESH 拓扑结构，而内存节点则在两侧，整个 2.5 维片上网络结构是不对称的。因此 2.5 维堆叠片上网络结构有三个可能的区域会出现瓶颈，包括上层网络、下层网络的中心区域、以及下层网络的边沿区域，如图 5.3所示。任一

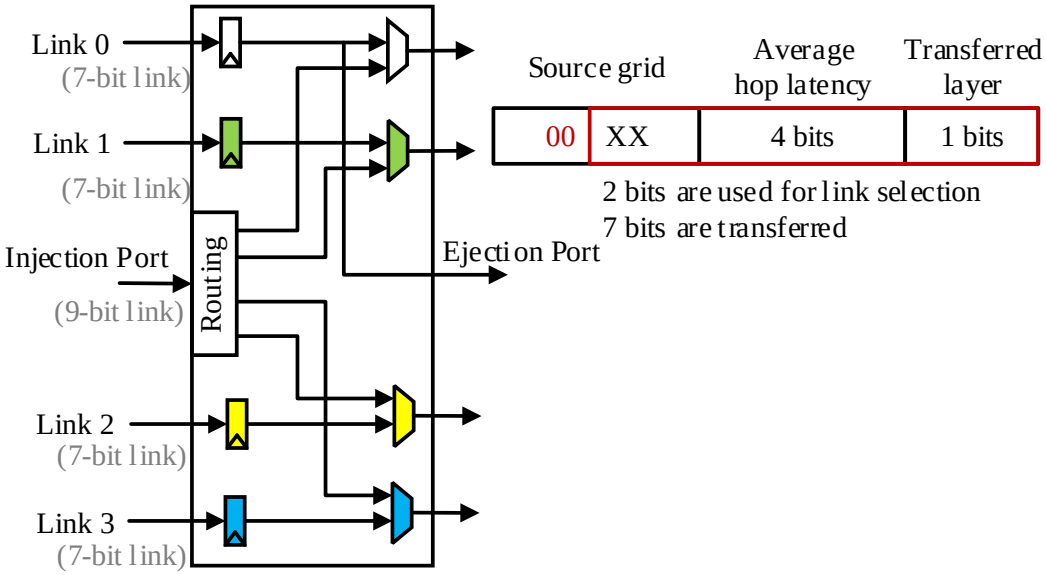


图 5.7　拥塞信息报文

这些区域都可能导致整个 2.5 维堆叠片上网络结构的饱和，而实际上很多区域始终在未饱和状态工作。

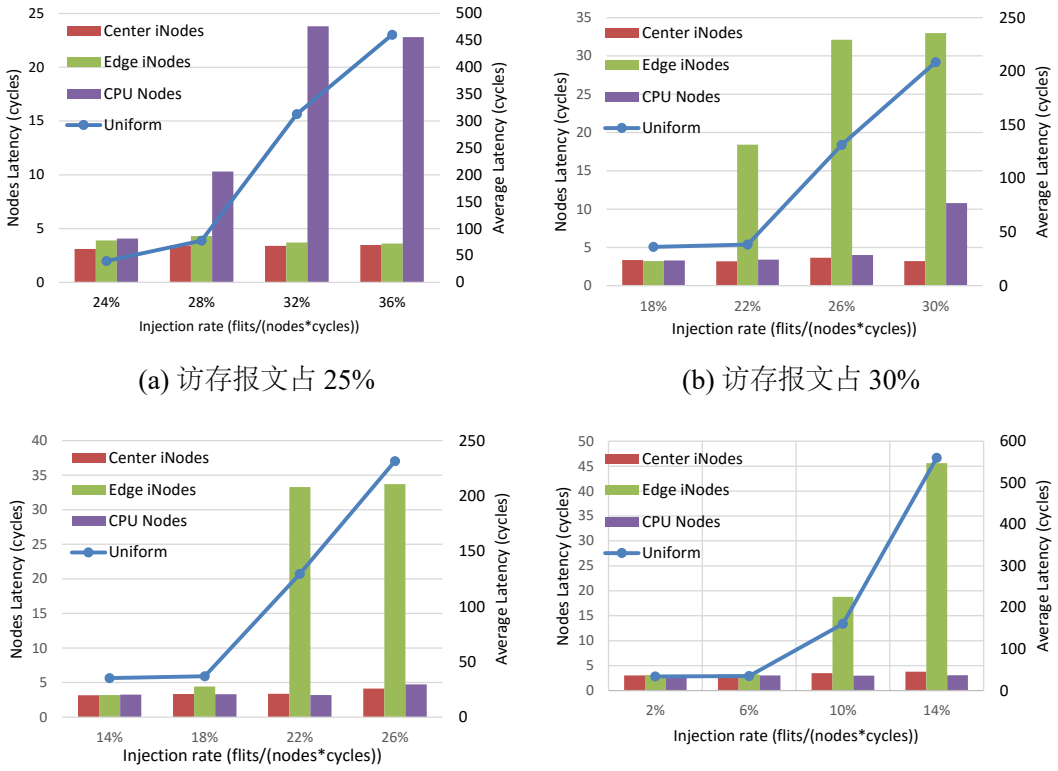
我们评估了报文传输经过这三块区域的平均延迟。在基准设计中，我们采用了 XY-Z 路由，实验结果如图 5.8所示。我们发现上层网络 CPU 节点在访存报文流量占总流量的 25% 会导致全局拥塞。上层网络的等分带宽是下层网络的两倍。因此，当访存通信的流量大于总流量的 30% 时，下层网络将成为瓶颈。图 5.8显示下层网络的边界节点在访存流量大于总流量的 30% 时会导致全局网络饱和。但是，当硅中介层上的边界节点饱和时，硅中介层上的中心节点的报文传输延迟依然非常小，负载也较低。当访存流量占总流量的 75% 时，下层网络的边界部分依然是性能瓶颈。

幸运的是，这个瓶颈可以通过 YX-Z 路由来缓解。YX-Z 路由首先将报文沿 Y 方向传输；这将降低边界列路由器的拥塞。图 5.10显示当访存流量负载非常重的时候，YX-Z 路由相比于 XY-Z 路由有 56.5% 的吞吐率提升。显然，由于报文先被路由到 Y 方向，硅中介层网络的两侧边界节点的流量将被大大降低。

通过上面的分析，负载均衡策略在访存流量小于总流量的 30% 时被应用。我们评估了 PARSEC 测试集的应用程序，发现平均访存流量占总流量的 14.8%。因此，我们的策略对于真实应用程序也会非常有效。

5.4.2　性能比较

在这一小节，我们比较几个不同策略的性能，包括 DLL 策略、基准设计（DOR）和其他两个负载均衡策略，LoacalBuf 和 DestDetect。DOR 设计通过不同



(a) 访存报文占 25%　　(b) 访存报文占 30%

(c) 访存报文占 35%　　(d) 访存报文占 75%

图 5.8　性能瓶颈（蓝线表示在 Uniform Random 通信模型下不同注入率的平均报文延迟；柱形图分别表示不同注入率报文通过不同类型节点的平均延迟。）

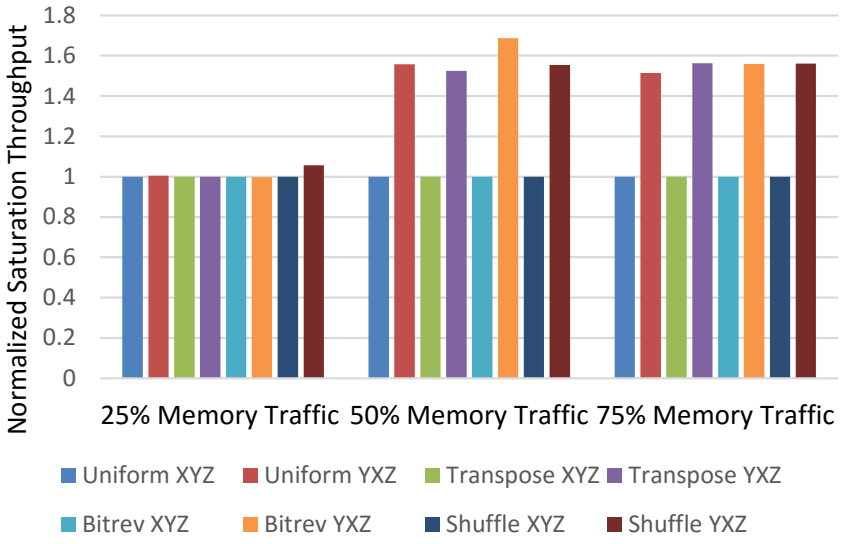


图 5.9　XY-Z 路由与 YX-Z 路由对比（上层网络分别采用四种不同类型的通信模式，下层网络采用 Uniform Random 的通信模式。）

的网络层传输不同类型的报文。LocalBuf 策略通过路由器缓存占用率来检测本地拥塞。如果缓存占用率大于 60%，则该路由器被认为是拥塞的。对于 LocalBuf 策略，如果上层网络中一个格子内超过一个节点是拥塞的，且硅中介层对应节点不拥塞，核间协议通信的流量将被转入下层以达到负载均衡。DestDetect 策略和我们的 DLL 策略类似，都是采用延时作为拥塞衡量指标。该方法收集最近报文的延迟，在目的节点计算每一跳步的平均延迟。然后目的节点将直接采用收到的报文的平均延迟来指导发送报文的网络层选择。如果从上层网络收到的报文的延迟大于下层网络收到的报文的延迟到一定阈值，则核间协议通信流量会被转入下层传输。DestDetect 和 DLL 策略都根据实验选择 8 个时钟周期作为阈值。所有的策略都被应用到第 3A 节描述的目标系统中。基准路由器采用 3 级流水线。为了公平比较的目的，YX-Z 路由被应用到所有的策略。

图 **??**比较了这些策略在不同访存流量占比的情况下的吞吐率。结果显示，相比于 DOR 设计、DestDetect 策略和 LocalBuf 策略，我们的 DLL 策略分别获得 45%、14.9% 和 6.5% 的吞吐率提升。针对不同的流量模型，我们的 DLL 策略在 Uniform Random 流量模型上相比于 LocalBuf 策略优势很小。而在 permutation 流量模型上，我们的 DLL 策略的性能则大大好于其他策略。这是因为 Permutation 的通信模式，流量几乎都是在节点对之间的，这种通信模式对于传输链路的延迟非常敏感。但是，DestDetect 策略的性能相比于 LocalBuf 策略更差。这是因为 DestDetect 策略收集的延迟拥塞信息是收到的报文的路径，而不是报文的发送的路径。因此，DestDetect 策略是一种不准确的方法。虽然 LocalBuf 策略不能够反映出拥塞的路径，但本地拥塞信息是准确的。它比 DestDetect 的性能更好。

针对不同的访存通信流量所占百分比。较小的访存通信流量占比会导致两个网络层更高的不平衡。两个网络层的流量越不均衡，负载均衡策略就能获得更高的收益。当访存流量只占总流量的 5% 时，DLL 策略相比于 DOR 设计平均有 55% 的性能提升。

我们还合并了 Booksim 和 Netrace 进行真实应用程序的评估。Netrace 中的 Trace 是从 M5 模拟器模拟的 64 核系统中收集的，包括一个 64KB 的私有一级缓存，一个共享的 16MB 的二级缓存，以及 8 个片上存储控制器。我们评估了 PARSEC 测试集的所有 traces，发现平均访存流量占总流量的 14.8%。图 5.11的结果显示，DLL 策略相比于 DOR 设计平均降低了 26.1% 的运行时间，而 LocalBuf 策略的性能比 DestDetect 稍高。

5.4.3 硬件开销

图 5.12展示了 DLL 策略相比于 DOR 设计的硬件开销。由于 LocalBuf 策略和 DestDetect 策略仅仅引入了一点点的逻辑和缓存开销，我们主要评估的是延迟传

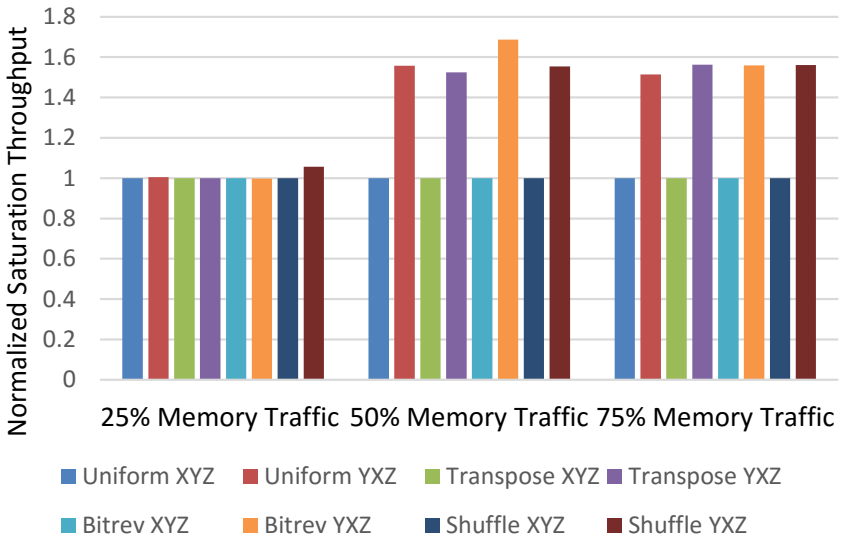


图 5.10　XY-Z 路由与 YX-Z 路由对比（上层网络分别采用四种不同类型的通信模式，下层网络采用 Uniform Random 的通信模式。）

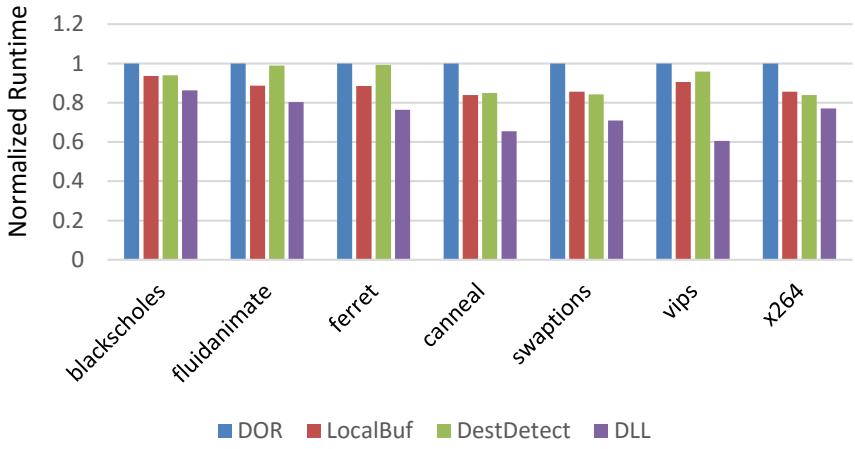
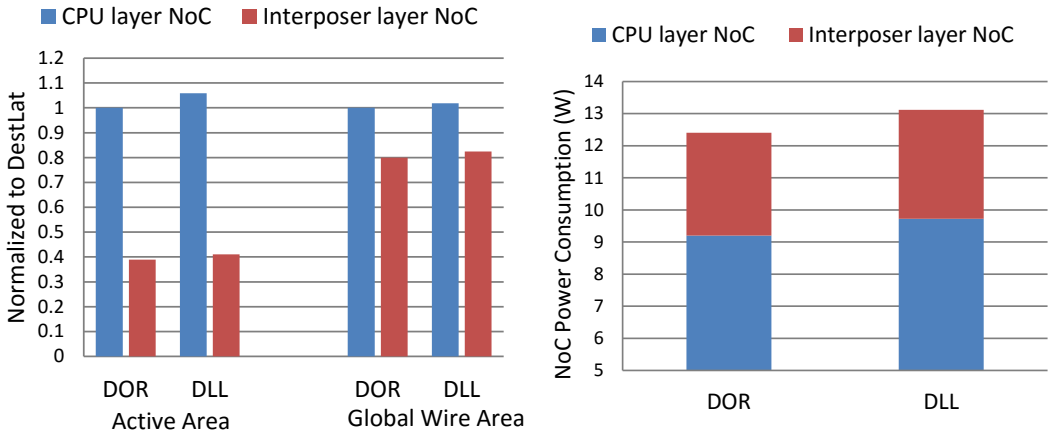


图 5.11　Parsec 测试集的性能

输环网带来的额外开销。实验结果是通过在一个 45nm bulk/SOI low-Vt 处理节点中采用 10% 切片 / 节点 / 时钟周期的注入率收集的。每个路由器中包含 8 条徐通道，每跳虚通道又包含 8 个 128 位的缓存。图 5.12中 CPU 层的片上网络和硅中介层上的片上网络是逻辑片上网络结构。硅中介层的实现在短期内都是被动形式。硅中介层上不存在晶体管。逻辑上的硅中介层片上网络采用金属线互联，而有源器件 (路由器、缓存和中继器) 均位于处理器内。这些大量的资源的部署有效提高了性能。与 DOR 设计相比，我们的 DLL 策略多消耗 7.7% 的面积和 5.8% 的功耗。事实上，主要的硬件开销来自于延迟传输环网的物理链路和相应存储拥塞信息报文的缓存。这些缓存和物理链路的开销占用了有源器件的面积和全局连线的

面积。由于我们的延迟传输环网被设计成果无阻塞形式，这些额外的开销很小，是可接受的。



(a) 面积　　(b) 功耗

图 5.12　硬件开销

5.5　讨论

在本节，我们进一步讨论三个细节问题，包括拓扑逻辑，硅中介层的类型和阈值。

首先，2.5 维堆叠片上网络可以应用其他的拓扑结构。Enright Jerger 和 Kannan 等人研究了集中拓扑逻辑。Double-butterflies 可以用于解决边界节点的性能瓶颈。这个拓扑结构减少了冲突并提高了硅中介层网络边界节点的路径多样性。在本文中，我们采用了集中式 MESH 结构显著的降低了 ubump 的开销，并且应用了简单的维序路由算法来避免死锁。这个拓扑结构仅作为负载均衡策略的实现例子。我们提出的 DLL 策略可以被应用到其他的拓扑结构，包括 Double-butterfly。

第二，硅中介层可以被用来堆叠处理器核存储器。当前的 2.5 维堆叠芯片均采用被动式的硅中介层，并不包含有源器件，这种实现仅需要便宜的实现成本和更高的产量。只有金属连线资源需要被开发，而传输延迟可以在硅中介层网络上被大大降低，这是因为硅中介层采用更宽和更厚的连线。而长远来看，有源硅中介层的实现可以更好的开发硅中介层上的资源。本文中，我们的目标是提高硅中介层上金属连线的资源利用率。由于有源器件需要被实现在芯片层，通过应用粗粒度的的方法和无阻塞环网来降低缓存和逻辑的开销。

第三，设置一个合适的阈值是本文的另一个关键点。本设计的阈值为不同网络层中平均每跳延迟的差值。如果这个阈值太大，网络对拥塞的反应会很慢。如果阈值设置的太小，很可能错误的网络选择判断会发生。但是，不同的流量模型

需要不同的阈值。在本文中，我们通过一系列实验选择一种适应更多应用程序类型的阈值。未来，我们会进一步研究反馈策略来动态地调整不同的阈值。

5.6 本章小结

2.5 维堆叠技术在硅中介层上堆叠芯片和 DRAMs。采用硅中介层上的金属线资源，提供了更多的机会去挖掘 2. 5 维堆叠片上网络结构的新特性。在本文中，我们关注到 CPU 层和硅中介层网络的通信流量不均衡问题，提出了一种动态延迟感知的负载均衡策略 DLL。DLL 在每个目的节点收集报文的平均延迟，利用一个多链路无阻塞的环网将延迟信息传输回源节点，为之后的网络选择提供依据以实现负载均衡。我们评估了不同比例的访存流量来寻找网络瓶颈。实验结果显示，我们的 DLL 策略相比于基准设计、DestDetect 策略和 LocalBuf 策略分别有 45%、14.9% 和 6.5% 的性能提升。同时，相比于基准设计，我们的 DLL 策略仅多消耗 7.7% 的面积资源和 5.8% 的功耗。我们之后的工作将会扩展 DLL 策略以采用适应性阈值设置，并将 DLL 策略应用到更多的拓扑结构中。

第六章　总结

6.1　本文的主要贡献

6.2　未来工作

6.3　结束语

致　谢

衷心感谢导师 xxx 教授和 xxx 副教授对本人的精心指导。他们的言传身教将使我终生受益。

感谢 NudtPaper，它的存在让我的论文写作轻松自在了许多，让我的论文格式规整漂亮了许多。

参考文献

[1] Hennessy J L, Patterson D A. Computer architecture: a quantitative approach [M]. Elsevier, 2011.

[2] Dennard R H, Gaensslen F H, Rideout V L, et al. Design of ion-implanted MOSFET's with very small physical dimensions [J]. IEEE Journal of Solid-State Circuits. 1974, 9 (5): 256–268.

[3] Hameed R, Qadeer W, Wachs M, et al. Understanding sources of inefficiency in general-purpose chips [C]. In ACM SIGARCH Computer Architecture News. 2010: 37–47.

[4] Jouppi N P, Young C, Patil N, et al. In-datacenter performance analysis of a tensor processing unit [C]. In 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). 2017: 1–12.

[5] Sutherland I E. Sketchpad a man-machine graphical communication system [J]. Simulation. 1964, 2 (5): R–3.

[6] Lindholm E, Kilgard M J, Moreton H. A user-programmable vertex engine [C]. In Proceedings of the 28th annual conference on Computer graphics and interactive techniques. 2001: 149–158.

[7] Silberstein M, Kim S, Huh S, et al. GPUnet: Networking abstractions for GPU programs [J]. ACM Transactions on Computer Systems (TOCS). 2016, 34 (3): 9.

[8] Silberstein M, Ford B, Keidar I, et al. GPUfs: Integrating a File System with GPUs [C/OL]. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 2013: 485–498. http://doi.acm.org/10.1145/2451116.2451169.

[9] Seo S, Jo G, Lee J. Performance characterization of the NAS Parallel Benchmarks in OpenCL [C]. In 2011 IEEE international symposium on workload characterization (IISWC). 2011: 137–148.

[10] Deng Y, Maly W P. Interconnect characteristics of 2.5-D system integration scheme [C]. In Proceedings of the 2001 international symposium on Physical design. 2001: 171–175.

[11] Loh G H, Jerger N E, Kannan A, et al. Interconnect-Memory Challenges for Multi-chip, Silicon Interposer Systems [C]. In Proceedings of the 2015 International Symposium on Memory Systems. 2015: 3–10.

作者在学期间取得的学术成果

发表的学术论文

[1] Yang Y, Ren T L, Zhang L T, et al. Miniature microphone with silicon- based ferroelectric thin films. Integrated Ferroelectrics, 2003, 52:229-235. (SCI 收录, 检索号:758FZ.)

[2] 杨轶, 张宁欣, 任天令, 等. 硅基铁电微声学器件中薄膜残余应力的研究. 中国机械工程, 2005, 16(14):1289-1291. (EI 收录, 检索号:0534931 2907.)

[3] 杨轶, 张宁欣, 任天令, 等. 集成铁电器件中的关键工艺研究. 仪器仪表学报, 2003, 24(S4):192-193. (EI 源刊.)

[4] Yang Y, Ren T L, Zhu Y P, et al. PMUTs for handwriting recognition. In press. (已被 Integrated Ferroelectrics 录用. SCI 源刊.)

[5] Wu X M, Yang Y, Cai J, et al. Measurements of ferroelectric MEMS microphones. Integrated Ferroelectrics, 2005, 69:417-429. (SCI 收录, 检索号:896KM.)

[6] 贾泽, 杨轶, 陈兢, 等. 用于压电和电容微麦克风的体硅腐蚀相关研究. 压电与声光, 2006, 28(1):117-119. (EI 收录, 检索号:06129773469.)

[7] 伍晓明, 杨轶, 张宁欣, 等. 基于 MEMS 技术的集成铁电硅微麦克风. 中国集成电路, 2003, 53:59-61.

研究成果

[1] 任天令, 杨轶, 朱一平, 等. 硅基铁电微声学传感器畴极化区域控制和电极连接的方法: 中国, CN1602118A. (中国专利公开号.)

[2] Ren T L, Yang Y, Zhu Y P, et al. Piezoelectric micro acoustic sensor based on ferroelectric materials: USA, No.11/215, 102. (美国发明专利申请号.)

附录 A　模板提供的希腊字母命令列表

大写希腊字母:

Γ \Gamma	Λ \Lambda	Σ \Sigma	Ψ \Psi
Δ \Delta	Ξ \Xi	ϒ \Upsilon	Ω \Omega
Θ \Theta	Π \Pi	Φ \Phi	
Γ \varGamma	Λ \varLambda	Σ \varSigma	Ψ \varPsi
Δ \varDelta	Ξ \varXi	Υ \varUpsilon	Ω \varOmega
Θ \varTheta	Π \varPi	Φ \varPhi	

小写希腊字母:

α \alpha	θ \theta	o o	τ \tau
β \beta	ϑ \vartheta	π \pi	υ \upsilon
γ \gamma	ι \iota	ϖ \varpi	ϕ \phi
δ \delta	κ \kappa	ρ \rho	φ \varphi
ϵ \epsilon	λ \lambda	ϱ \varrho	χ \chi
ε \varepsilon	μ \mu	σ \sigma	ψ \psi
ζ \zeta	ν \nu	ς \varsigma	ω \omega
η \eta	ξ \xi	$\varkappa$ \varkappa	F \digamma
α \upalpha	θ \uptheta	ο \mathrm{o}	τ \uptau
β \upbeta	ϑ \upvartheta	π \uppi	υ \upupsilon
γ \upgamma	ι \upiota	ϖ \upvarpi	ϕ \upphi
δ \updelta	κ \upkappa	ρ \uprho	φ \upvarphi
ε \upepsilon	λ \uplambda	ρ \upvarrho	χ \upchi
ε \upvarepsilon	μ \upmu	σ \upsigma	ψ \uppsi
ζ \upzeta	ν \upnu	σ \upvarsigma	ω \upomega
η \upeta	ξ \upxi		

希腊字母属于数学符号类别，请用\bm 命令加粗，其余向量、矩阵可用\mathbf。