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 11, 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%。

Lee 等人提出了 GPU 上嵌套并行模式的局部感知映射，其中充分利用了嵌套并行计算与 GPU 线程没有通用的最佳映射的观察。具有嵌套并行性的算法（例如 map/reduce 操作）可以使其并行性在不同级别暴露给 GPU，具体取决于 GPU 程序的编写方式。作者利用 GPU 上嵌套并行映射的三种推广：•1D 映射，它将顺序程序的外循环并行化; • 线程块 / 线程映射，它将顺序程序的外部循环的每次迭代分配给线程块，并在线程块上并行化内部模式; • 基于 warp 的映射，它将外循环的每次迭代分配给 warp 并在 整个 warp 中并行化内部模式。这项工作提出了一个自动编译框架，它根据局部性和嵌套模式中暴露的并行度生成预测的性能分数，以选择哪种映射最适合于一组常见的嵌套并行模式。这些模式包括集合操作，如 map，对每个过滤器等进行 reduce。框架尝试将线程映射到集合的每个元素上的操作。框架处理通过首先将应用程序中的每个嵌套级别分配给维度（x，y，z 等）来嵌套模式。双嵌套模式（即，包含缩小的地图）具有两个维度。然后，映射确定 CUDA 线程块中给定维度中的线程数。在设置线程块的维度和大小之后，框架通过使用以下方法为每个线程分配多个元素来进一步控制内核中的并行度。线程跨越和分裂的概念。在二维内核中（即，两个模式嵌套级别），如果为每个维度分配 span（1），则在内核中启动的每个线程仅负责对集合的一个元素进行操作。此映射公开了最大程度的并行性。相反，span（all）表示每个线程对集合中的所有元素进行操作。跨度可以是（1）和（全部）之间的任何数字。Span（all）用于两种特殊情况：直到内核启动之后才知道维度的大小（例如，当动态确定内部模式中操作的元素数量时）以及模式需要同步时（例如，reduce 操作）。

第三章　一种 GPU 内存超额配置的管理框架

3.1　研究背景

3.2　相关工作

3.3　研究动机

3.4　设计

3.5　实验验证

3.6　本章小结

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　研究背景

4.2　相关工作

4.3　研究动机

4.4　设计

4.5　实验验证

4.6　本章小结

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

5.2　相关工作

5.3　研究动机

5.4　设计

5.5　实验验证

5.6　本章小结

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.

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	
$\varGamma$ \varGamma	$\varLambda$ \varLambda	$\varSigma$ \varSigma	$\varPsi$ \varPsi
$\varDelta$ \varDelta	$\varXi$ \varXi	$\varUpsilon$ \varUpsilon	$\varOmega$ \varOmega
$\varTheta$ \varTheta	$\varPi$ \varPi	$\varPhi$ \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	$\digamma$ \digamma
$\upalpha$ \upalpha	$\uptheta$ \uptheta	o \mathrm{o}	$\uptau$ \uptau
$\upbeta$ \upbeta	$\upvartheta$ \upvartheta	$\uppi$ \uppi	$\upupsilon$ \upupsilon
$\upgamma$ \upgamma	$\upiota$ \upiota	$\upvarpi$ \upvarpi	$\upphi$ \upphi
$\updelta$ \updelta	$\upkappa$ \upkappa	$\uprho$ \uprho	$\upvarphi$ \upvarphi
$\upepsilon$ \upepsilon	$\uplambda$ \uplambda	$\upvarrho$ \upvarrho	$\upchi$ \upchi
$\upvarepsilon$ \upvarepsilon	$\upmu$ \upmu	$\upsigma$ \upsigma	$\uppsi$ \uppsi
$\upzeta$ \upzeta	$\upnu$ \upnu	$\upvarsigma$ \upvarsigma	$\upomega$ \upomega
$\upeta$ \upeta	$\upxi$ \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。