分类号 TP957 学号 11069003

UDC 密级 公开

工学博士学位论文

众核加速器的缓存管理

博士生姓名 陈项颢

学科专业 计算机科学与技术

研究方向 计算机系统结构

指导教师 王志英 教授

国防科学技术大学研究生院

二〇一四年十月

Cache Management for Many-core Accelerators

Candidate: Chen Xuhao

Supervisor: Professor Wang Zhiying

A dissertation

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Engineering

in Computer Science and Technology

Graduate School of National University of Defense Technology

Changsha, Hunan, P. R. China

October 27, 2014

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学技术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：众核加速器的缓存管理

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学技术大学有关保留、使用学位论文的规定。本人授权国防科学技术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：众核加速器的缓存管理

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

为了延续摩尔定律，半导体产业开始向高效能的异构芯片或系统的方向发展。以 GPU 为代表的众核加速器得到广泛应用，并且开始集成到通用微处理器中。 GPU 采用 SIMT 执行模型，对于很多访存模式规则的应用程序， GPU 能够通过大规模多线程来隐藏访存延迟。为了支持更多不规则访存模式的应用程序，片上缓存层次结构被加入到 GPU 体系结构中，来捕捉时间和空间局部性，从而缓解不规则访问对系统性能的不利影响。然而， GPU 缓存的效率不高，制约了系统的性能和效能。

GPU 缓存低效的主要原因是其管理策略同面向吞吐率的执行模型不相适应。GPU 生成的大量访存请求引起了缓存冲突和资源拥塞。现有的 CPU 缓存管理策略是针对多核系统设计的，直接应用到 GPU 中效果并不好。这主要是因为 CPU 缓存管理策略无法有效控制工作集和其它资源使用情况。不仅如此，当大规模并行受限于片上资源时，计算部件长时间处于等待数据的状态，系统效能也会因此降低。为了尽可能地减少访存延迟和带宽需求，程序员往往需要对 GPU 代码进行复杂而繁琐的优化，这在很大程度上增加了程序员的负担。

为了解决上述问题，本文提出针对 GPGPU 的执行模式定制其片上缓存管理策略。对线程束调度和流访问模式感知的缓存替换策略能够减少缓存污染和冲突。基于重用距离的缓存旁路策略对缓存层次进行保护，以缓解缓存冲突。动态监测机制在运行时通过计数器采样获取缓存冲突和资源拥塞的信息。为了避免过度使用片上资源，旁路策略协同线程束调节机制对活跃的线程束数目进行动态控制。本文的研究工作和创新点主要包括：

1. 提出了针对 GPGPU 大规模并行执行模式下访存行为的自适应缓存替换和旁路策略。本文对 GPGPU 应用的访存行为进行了详细的模拟和分析，实验结果显示了 GPU 缓存的低效及其根源。针对严重的缓存污染和冲突问题，本文将 CPU 中现有的先进缓存管理策略移植到 GPU 中，实验结果显示，先进的管理策略能够一定程度上提升 GPU 缓存效率，但是仍然存在局限性。本文结合了目前最先进的防污染和防冲突缓存管理策略，提出了针对 GPU 中流访问模式和剧烈冲突问题的管理策略，实验结果表明，该策略能够获得明显的性能提升。

2. 提出了克服单纯旁路策略局限性的协同旁路和线程束调节技术。在缓存旁路策略对缓存层次进行保护的基础上，引入线程束调节机制对活跃的线程

束数目进行动态控制。通过动态缓存冲突和资源拥塞监测机制获取反馈信息，指导线程束调节。本文还提出了一个简单的预测器来动态估计最优的活跃线程束数目，以充分利用缓存容量和其它片上资源。实验结果表明，对于缓存敏感的测试程序，该方法能显著提高缓存效率，且更好地利用片上资源。相比基准 GPU 架构和最优静态线程束调节，系统性能（IPC）分别平均提升了 74% 和 17%，（最多提升达到 661% 和 44%）。

3. 提出了 GPGPU 中缓存感知的功耗管理机制。理想情况下 GPU 的峰值性能同其计算单元的数目和工作频率成正比，但实际的应用程序对系统资源的需求差异很大，访存密集的应用程序很可能受限于存储子系统，因而远远无法达到其峰值的计算性能。本文针对受限于存储子系统的访存密集型应用程序，考虑在不明显降低其性能的情况下，节省系统功耗，提升系统效能。在监控系统资源（存储子系统中的缓存以及 NoC 和 DRAM 带宽）使用情况的基础上，本文通过运用核调整技术来控制活跃核的数目，或者通过 DVFS 技术调节工作电压 / 频率，来达到节省功耗、提升系统效能的目的。

关键词：GPGPU; **缓存管理**; **替换和旁路**; **线程束调节**; **功耗控制**

ABSTRACT

Moore's law drives the semiconductor industry to the direction of developing energy-efficient heterogeneous chips or systems. Manycore accelerators, such as GPUs, have been widely used and integrated into general-purpose microprocessors. With the SIMT execution model, GPUs can hide memory latency through massive multi-threading for many regular applications. To support applications with irregular memory access patterns, cache hierarchies have been introduced to GPU architectures to capture locality and mitigate the effect of irregular accesses. However, GPU caches exhibits poor efficiency due to the mismatch of the throughput-oriented execution model and its cache hierarchy design, which limits system performance and energy-efficiency.

Existing CPU cache management policies that are designed for multi-core systems can be suboptimal when directly applied to GPU caches. This is mainly because CPU cache management schemes are not able to control wroking set and resource utilization. Moreover, when massive parallelism is limited by the on-chip resources, execution units are waiting for the data to return back from memory subsystem, in which case the system energy efficiency is highly hampered. To reduce memory access latency and bandwidth requirements, programmers often need to do complicated and tedious optimization for GPUs, which largely increases programmer's burden.

Therefore, we propose specialized cache management scheme for GPGPUs. The cache replacement policy is aware of warp scheduling and streaming access pattern, and can significantly reduce cache pollution and contention. Reuse distance-based cache bypass policy protects cache hierarchy to mitigate cache contention. Dynamic monitoring mechanism captures cache contention and resource congestion information by the counter sampling at run time. To avoid over-saturating on-chip resources, the bypass policy is coordinated with warp throttling to dynamically control the active number of warps. Innovation and contribution of this thesis include:

1. We propose adaptive cache replacement and bypass policy for the memory access behavior under the massively parallel execution model of GPUs. We carry out a detailed simulation and analysis on the memory access behavior of GPU applications, and the experimental results show GPU cache inefficiency and its root causes. To deal with severe cache pollution and contention, we first port state-of-the-art CPU

cache management schemes to the GPU, which demonstrates that advanced management schemes can improve GPU cache efficiency. It is also illustrated that pure bypass policies have limitations. We then combine the most advanced anti-pollution and anti-thrashing cache management schemes, and propose an adaptive scheme for GPU streaming pattern and severe cache contention. Experimental results show that this scheme can achieve further performance improvements.

2. Given the limitations of pure bypassing, we propose an specialized cache management policy for GPGPUs. The cache hierarchy is protected from contention by the bypass policy based on reuse distance. Contention and resource congestion are detected at runtime. To avoid over-saturating on-chip resources, the bypass policy is coordinated with warp throttling to dynamically control the active number of warps. We also propose a simple predictor to dynamically estimate the optimal number of active warps that can take full advantage of the cache capacity and on-chip resources. Experimental results show that cache efficiency is significantly improved and on-chip resources are better utilized for cache-sensitive benchmarks. This results in a harmonic mean IPC improvement of 74% and 17% (maximum 661% and 44% IPC improvement), compared to the baseline GPU architecture and optimal static warp throttling, respectively.

3. We present cache-aware power management for GPGPUs. Ideally the peak performance of GPUs is proportional to the number of execution units and their operating frequencies, but practically the system resource requirements vary largely for different applications, for example, memory intensive applications are likely to be limited by the memory subsystem, thus unable to reach its peak throughput. Targeting memory intensive applications that limited by the memory subsystem, we consider to save system power and improve energy efficiency without sacrificing performance. Based on monitoring the utilization of system resources (caches and NoC/DRAM bandwidth), we employ core scaling mechanism to control the number of active SIMT cores, and DVFS technology to scale the operating voltage/frequency, and thus to reduce power consumption and improve energy efficiency.

Key Words: GPGPU; cache management; warp throttling; power control

符号使用说明

HPC	高性能计算
CPU	中央处理器
GPU	图形处理器
CMP	片上多处理器
SMP	对称多处理器
SMT	同时多线程
ILP	指令级并行
TLP	线程级并行
DLP	数据级并行
IPC	每周期执行指令数
TBB	线程构建块
NoC	片上网络
MSHR	失效状态保存寄存器
SIMD	单指令多数据
SIMT	单指令多线程
MIMD	多指令多数据
SPMD	单程序多数据
CUDA	计算统一设备架构
OpenCL	开放计算语言
cache	片上高速缓存
line	缓存数据块
GPGPU	通用计算图形处理器
kernel	数据并行应用内核
CTA	协作线程阵列
warp	线程束
thread	线程

RD	重用距离
PD	保护距离
bypss	旁路
RRIP	重用距离预测
PDP	保护距离预测
SWL	静态线程束限定
MAW	最大活跃线程束数目
CBWT	协同旁路和线程束调节
DVFS	动态电压频率调整

第一章　引言

随着半导体产业从多核向众核发展，通过提升 CPU 时钟频率和内核数量而提高计算能力的传统方式遇到了功耗和散热瓶颈 [1]，因而高效能（energy efficiency）已经成为微处理器最为重要的设计目标之一。异构体系结构 [2] 能够针对不同的计算模式定制不同类型的处理引擎（processing engine），从而提升系统效能。比如，图形处理器 GPU 采用 CUDA [3] 或者 OpenCL [4] 进行通用计算编程，即 GPGPU （General-purpose computing on graphics processing units），在处理数据并行应用内核时，具有相比通用 CPU 更好的效能，因此已经被高性能计算 HPC 领域广泛使用。 GPU 厂商正不断加入新的软硬件特征到这类面向吞吐率的加速器中，以更好地支持具有各种访存模式的不规则应用。 Cache 层次结构 [5, 6] 就是其中一种被引入到 GPU 中的硬件特征，它能对以简单方式编写（未优化）的不规则程序提供显著的性能加速。但是目前的 GPU 缓存管理策略缺乏对大规模并行特性的考量，导致在运行访存密集型应用时缓存效率低下，从而制约了系统性能和效能的进一步提升。相比针对特定计算模式定制处理引擎的异构设计方法，本文提出针对特定处理引擎定制其片上存储层次管理策略，使之更好地适应特定计算模式对存储层次的需求，从而提升系统性能和效能。本文选取 GPU 作为特定处理引擎的研究对象，这是因为 CPU+GPU 系统是目前最为常见的异构系统，不论是用于高性能科学计算的分离式 GPU [7–9]，还是同 CPU 集成在一块芯片上的融合式 GPU [10, 11]，本文提出的方法都能适用。

1.1　研究动机

在 GPU 中通常有成百上千的线程同时执行，这种大规模多线程执行环境对缓存层次结构造成了很大的请求压力。大量的请求被发送到缓存系统中，使得缓存冲突（thrashing，或称 contention）频繁发生。当访存分歧（memory divergence）发生时，问题进一步加剧。图 1.1中显示了在采用 32KB 的 L1 缓存情况下 GPU 中重用（reuse）次数的分布情况。对于大多数测试程序，被插入到 L1 缓存中的大部分缓存数据块根本没有被重用过（即重用次数为 0）。这也直接体现为 GPU 中 L1 缓存的失效率非常高。导致缓存低效率的原因通常有两个问题：缓存污染和缓存冲突。缓存污染是指零重用或者重用次数很少的缓存数据块占据了缓存空间，导致重用次数多的缓存数据块得不到足够的缓存空间。缓存冲突则是指程序工作集远大于缓存容量导致存在重用可能的缓存数据块频繁地相互替换而无法得以重用。由于 GPU 的大规模多线程特性，我们发现这两个问题在 GPU 缓存中变得更加严重（详见第 4章），而且直接采用 CPU 缓存管理策略效果并不理想，这

就需要我们针对 GPU 的特性调整其缓存管理策略。本文从异构系统的“定制计算”思想出发，提出“定制存储层次”的概念，并说明其重要性和好处。定制存储层次包括两个方面：一是定制存储层次的结构，包括缓存级数、各级缓存的容量等等；二是定制存储层次的管理策略，包括缓存写策略，缓存替换策略和旁路策略等等。前者决定了缓存呈现出的物理参数特性，需要通过设计空间搜索（design space exploration）进行权衡。本文则主要考虑后者，即针对特定的处理引擎定制相应的缓存管理策略，以达到提高缓存效率，从而提升系统性能和效能的目标。

1.2 研究内容

本文的研究针对并行计算系统的三大问题：内存墙（Memory Wall）、编程墙（Programming Wall）和功耗墙（Power Wall），通过对异构系统中的特定处理引擎定制存储层次管理策略来提升性能、降低编程难度并减少功耗。由于内存的存取速度严重滞后于处理器的计算速度，即内存墙问题，对于延迟敏感的通用 CPU，缓存的性能至关重要。因此研究人员提出很多 CPU 缓存管理策略，以提高缓存的性能，从而减少访存延迟并提高带宽，最终提升系统性能。出于类似的目的，缓存层次结构从 NVIDIA Fermi 架构开始被引入到 GPU 中。本文首先对 GPU 的存储子系统进行了深入细致的评测，并详细分析了导致 GPU 缓存层次结构低效的原因。为了提升 GPU 缓存的效率，本文将 CPU 中的先进缓存替换和旁路（bypass）策略在 GPU 中的实现和改进。这些缓存管理策略能够明显提升 GPU 的系统性能，但是仍然存在局限性。针对这些局限性，本文考虑在多线程系统中，合理控制并行度（并发线程数目）能够进一步提高缓存的效率，同时充分利用其它片上资源。这些改进的缓存管理策略在提升性能的同时，也降低了程序员对显式使用便笺存储器（scratchpad）的依赖性，能够简化编程。最后，对于受限于缓存容量、NoC 或内存带宽的访存密集型应用，合理调节处理器的时钟频率或者活跃核的数目能够有效降低功耗，而且同时保证系统性能不受损害。

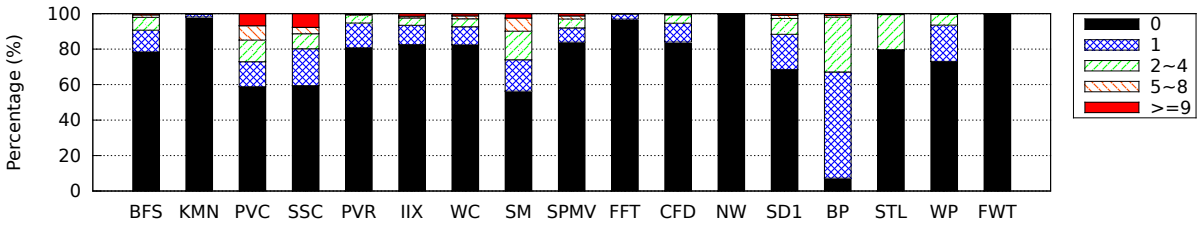

图 1.1 L1D 缓存重用次数分布：显示了在一个缓存数据块被插入 L1 缓存中后对其进行的重复访问的次数。

1.2.1 缓存替换和旁路

缓存替换策略对缓存乃至整个系统的性能影响很大，因此改进替换策略有助于解决 GPU 缓存低效的问题。 LRU （ Least Recent Used ）替换策略选取最近最少使用的缓存数据块进行替换，是 CPU 中最常见的替换策略。 NRU （ Not Recently Used ）替换策略选取最近没使用的缓存数据块进行替换，是 LRU 的一种近似策略。 RRIP （ Re-Reference Interval Prediction ）替换策略[12] 在 NRU 策略基础上进行改进，通过区分失效（ miss ）和命中（ hit ）访问，使被重用的缓存数据块在替换时得到保护。缓存替换策略虽然有助于减少污染、缓解冲突，但是在 GPU 的大规模多线程环境中效果不好。 PDP （ Protection Distance Prediction ）策略[13] 则是在 RRIP 的基础上进一步一般化，其中 PDP 旁路策略对缓存数据块设置一定的保护距离（ protection distance ），让重用概率高的缓存数据块驻留在缓存中，而让一部分请求旁路缓存，从而避免了大量的缓存冲突。本文将 PDP 策略在 GPU 缓存中进行了实现，实验结果表明，将 PDP 策略这种针对 CPU 最末级缓存（ LLC： Last Level Cache ）设计的旁路策略直接应用于 GPU 能够明显提升 GPU 性能，但是仍然存在改进空间。本文在 PDP 的基础上设计针对 GPU 缓存的替换和旁路策略。该策略综合考虑缓存冲突和污染问题，在运行时对缓存冲突和污染情况进行采样监控，然后根据这些信息统筹替换和旁路的决策，以进一步提高缓存效率和系统性能。

1.2.2 线程调节

虽然缓存替换和旁路策略能够缓解缓存冲突问题，但是这些针对 CPU 缓存设计的管理策略没有考虑 GPU 中大规模多线程的特征。 GPU 中成百上千的并发线程使得缓存冲突和污染问题严重加剧，大量的失效和旁路请求会导致片上资源拥塞，从而制约系统性能的提升 [14] 。线程调节（ thread throttling ）技术 [15, 16] 被应用在 CPU 中来有效缓解资源拥塞。 CCWS [17] 也采用了线程束（ warp ）调节技术来缓解线程束间的缓存冲突，从而提高 GPU 中 L1 缓存的命中率。然而，单纯的线程束调节减少了活跃线程束的数目，因而损失了大规模多线程的好处，比如隐藏访存延迟。不但如此，减少线程束数目也可能会导致片上网络 NoC（ Network-on-Chip ）和片外 DRAM 带宽未得到充分利用，因而限制了进一步提升性能的机会。图 5.2中展示了静态线程束限定（ SWL： static wavefront limiting ）和缓存旁路对基准测试程序 BFS 的性能提升。该实验从 1 到 24 静态地改变每个线程束调度器上所允许的最大活跃线程束数目（ MAW： maximum active warps ）。图中两条曲线分别代表没有启用缓存旁路的 SWL （ SWL ）和启用了缓存旁路的 SWL （ SWL+bypass ）。从图中可以看出，当 MAW 大于 2 时，

缓存旁路能够在 SWL 的基础上提升系统性能。然而，将 MAW 限定从 24 变成 7 时， SWL+bypass 的性能提升从 1.36 倍增加到 1.77 倍。图中还显示了 SWL 和 SWL+bypass 的最佳 MAW （即达到最大性能的 MAW ）不一样，分别是 3 和 7 。也就是说当旁路启用后，最佳 MAW 变大了，这是由于缓存旁路允许更多的线程束发射访存请求而不会影响缓存性能，因为缓存数据块得到了抵御冲突的旁路策略的保护，而与此同时，闲置的带宽能够被利用起来将请求发送到存储层次的下一级。这就意味着协同旁路和线程束调节能够提供在单纯缓存旁路或者单纯线程束调节的基础上进一步提升性能的机会。本文针对单纯旁路策略和单纯线程束调节机制的局限性，将二者有机地结合起来，形成互补，提出了协同旁路与线程束调节，并设计了开销非常低的硬件监控和预测机制来实现这种动态自适应策略。

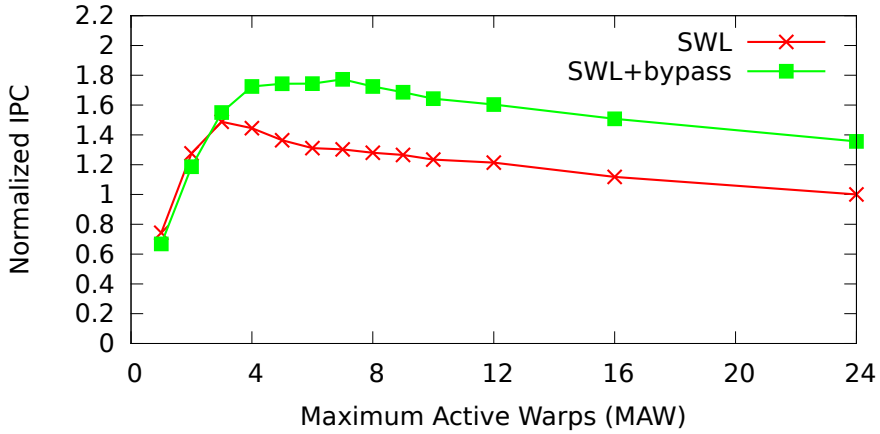

图 1.2　没有缓存旁路的 SWL （ SWL ）和有缓存旁路 SWL （ SWL+bypass ）情况下， BFS （表 2.2）的性能，这是在类似 Fermi 架构的 GPU 模拟器（表 2.1）中进行的实验，所以数据都归一化到 SWL-24 ，即 MAW 为 24 的静态线程束限制（ static warp limiting ） [17] 。这里采用的旁路策略是 PDP 策略 [13] ， PDP 是目前 CPU 中最先进的旁路策略，本文中我们将其移植到了 GPU 中（详见第 4章）。

1.2.3　功耗控制

GPU 通过成百上千的处理单元提供强大的计算吞吐率。理想情况下 GPU 的峰值性能同其计算单元的数目和工作频率成正比，但实际的应用程序对系统资源的需求差异很大，访存密集的应用程序很可能受限于存储子系统，因而远远无法达到其峰值的计算性能。当存储子系统成为系统性能瓶颈时，计算部件处于等待数据从存储子系统返回的状态，没有进行实质性的计算工作，同时又产生了不必要的功耗。在这种情况下，我们可以通过运用核调整（ core scaling ）技术和动态电压频率调整（ DVFS： Dynamic voltage and frequency scaling ）技术来调控处理

器的静态和动态功耗。 DVFS 根据应用程序对计算能力的不同需要，动态调节处理器的运行频率和电压（ P-states ），从而达到节能的目的。核调整则是动态控制活跃的处理器核数目，在不需要太多核的情况下，将部分核转换为休眠状态（ sleep states ）。本文针对访存密集型应用程序，在缓存容量或者 NoC 和 DRAM 带宽成为系统性能瓶颈的情况下，功耗控制模块合理地运用核调整和 DVFS 技术来降低功耗，提升系统效能，同时保证系统性能不受明显影响。

1.3 本文主要贡献

本文针对众核加速器中缓存效率低下的问题，主要开展了三个方面的研究工作:

- **对大规模多线程应用程序进行了详细的评测和分析，并将现有最先进的 CPU 缓存替换和旁路策略在 GPU 中进行了实现和评估**。实验表明对于缓存敏感型应用可以通过改进缓存管理策略提升 GPU 系统性能。本文还结合了现有的防污染和防冲突管理策略，**提出了针对 GPU 的自适应缓存管理策略**，实验结果表明该策略能够进一步提升系统性能。

- **提出了针对 GPU 的自适应缓存管理策略**。由大规模多线程引发的冲突和片上资源拥塞被动态监测，据此缓存旁路策略协同线程束调节技术能够充分利用缓存容量和片上资源。本文还为协同旁路和线程束调节（ CBWT ：coordinated bypass and warp throttling ）设计了一个简单的动态预测器，能够在运行时估计最优的活跃线程束数目。相比基准 GPU 架构和最优静态线程束调节，该方法分别能够提升 GPU 平均系统性能 1.74 倍和 1.17 倍。

- **提出了 GPGPU 中缓存感知的功耗管理机制**。访存密集的应用程序往往受限于存储子系统，因而无法达到 GPU 的峰值计算性能。本文针对这一类应用程序，考虑在不明显降低其性能的前提下，节省系统功耗，提升系统效能。在监控存储子系统中的缓存和（ NoC 、 DRAM ）带宽资源使用情况的基础上，本文通过运用核调整技术来控制活跃核的数目，以及通过 DVFS 技术调节工作电压 / 频率，能够有效地节省功耗、提升系统效能。

1.4 本文结构

本文接下来按如下方式组织：第 2章介绍众核加速器相关背景，着重讨论了现代 GPU 体系结构，特别是其存储层次结构。另外还介绍了 GPU 的编程模型及其应用特征。第 3章比较了与本文相关的研究工作。第 4章提出了针对 GPU 的自

适应替换和旁路策略。第 5章在第 4章的基础上提出了协同旁路和线程束调节策略。第 6章提出缓存感知的功耗控制策略。第 7章总结全文。

第二章　研究背景

摩尔定律 [18] 指出集成电路上可容纳的晶体管数目每隔约 18 个月便会增加一倍。因此从 1971 年 Intel 推出世界上第一款通用微处理器 4004 开始，到 21 世纪初，处理器产业一直通过提高主频和挖掘指令集并行（ILP： Instruction Level Parallelism） [19–24] 来提升计算能力。但是这一趋势在 2005 年左右开始发生变化。当主频接近 4GHz 时 [25]， Intel 和 AMD 发现芯片速度遇到了极限，也就是说单纯的主频提升已经无法明显提升系统整体性能。功耗 [26] 问题也成为制约处理器性能提升的主要因素。在这种情况下，工业界开始向片上多处理器 [27, 28]（CMP: Chip Multi-Processor），即多核处理器的方向发展。 2005 年 4 月， AMD 发布了拥有两个处理器核的 Opteron 和 Athlon 64 X2 处理器。 2006 年 7 月 Intel 推出了基于 Core 架构的双核处理器。同年 11 月，又推出面向服务器的 Xeon 处理器。与上一代处理器相比，双核处理器在性能方面提高了 40%，功耗反而降低 40%。在此之后，处理器产业通过扩展处理器核的数目和增大片上缓存容量来提升新一代处理器的性能 [29]。然而，一方面功耗约束 [1] 和有限的主存带宽 [30–32] 使得这种方式仍然不具有长期的可扩展性。另一方面，通过编写和优化并行程序来开发线程级并行（TLP： Thread Level Parallelism） [33–35] 还是很困难，这就意味着要利用好 CMP 的大量计算资源其实还是存在很大的障碍。为了延续摩尔定律，效能开始成为处理器设计的重要指标，而可编程性也成为软硬件权衡的重要考量。

通用 CPU 虽然具有很好的编程性，但是由于其设计是为了适应各种不同应用的需求，因而效能低下 [36]。要设计高效能的计算系统，最直观的方法是针对不同的应用设计 ASIC 芯片，但是缺乏可编程性。相比而言， FPGA 则具备一定的可编程性，而其效能则处于 ASIC 和通用 CPU 之间。可见，不同的计算单元有各自的优势和缺陷，为了适应不同应用程序的需求，异构体系结构应运而生。异构体系结构使用不同类型指令集和微架构的计算单元组成系统，近年来已经得到学术界和工业界的广泛关注。由于异构计算系统根据特定的计算模式定制相应的处理引擎，因而它具有相比同构系统更高的效能。常见的处理引擎类别包括 CPU、GPU 等协处理器、 DSP、 ASIC、 FPGA 等 [37]。异构计算虽然优势明显，但也给编程带来了很大的麻烦。为了合理高效的利用异构计算资源，程序员往往需要了解底层硬件细节，针对应用特点进行任务划分和调度。同时针对任务的访存模式和硬件提供的存储层次进行优化，这使得异构并行编程极为复杂。本文主要针对异构计算中的存储墙问题，考虑如何尽可能地利用好异构计算资源，同时又简化编程难度。

2.1 众核加速器

加速器并不是一个新概念。早在单核处理器系统中，加速器作为辅助 CPU 的额外的硬件资源，能够显著提升对特定类型应用的性能。浮点协处理器（例如 Intel 8087, FPU ）就是一个典型代表。微处理器中另一种常见的加速器就是 SIMD 扩展单元。 1995 年 Sun 公司在其 UltraSparc I 系统中 [38] 的 VIS 指令集扩展中引入了 SIMD 整形指令， 1996 年 Intel Pentium 处理器加入了 MMX SIMD 扩展单元，用于加速多媒体应用。类似地， 1997 年 IBM 联合 Motorola 发布了 Altivec ， 1998 年 AMD 发布了 3DNow! 。 1999 年发布的 Intel Pentium III 系列处理器将这一技术升级为 SSE （ Streaming SIMD Extensions ）。作为 x86 架构的 SIMD 指令集扩展， SSE 包含 70 条新指令，大都针对单精度浮点运算，支持的向量宽度达到 128 位。 Intel 之后又陆续将 SSE 扩展到 SSE2 、 SSE3 、 SSSE3 和 SSE4 [39] 。 2008 年推出的 Intel Sandy Bridge 中，这一技术被升级为 AVX （ Advanced Vector Extensions ）。 AVX [40] 提供了新的特征、指令和编码策略，并支持 128 位和 256 位向量宽度。 AVX2 （ Advanced Vector Extensions 2 ，又称 Haswell New Instructions ）在最新推出的 Haswell [10] 处理器中开始引入。 AVX2 将大多数 SSE 和 AVX 中的整数向量指令扩展到 256 位，还增加了 gather 指令，能够支持对非连续内存位置的向量元素的同时访问。对于吞吐率敏感型的应用，合理利用 SIMD 单元能够获得明显的性能提升 [41] 。加速器在智能移动终端（例如智能手机）等嵌入式系统中也经常以 SoC （ System on Chip ）的形式被使用。比如，为了加速音频播放，智能手机的处理器中通常会集成音频解码部件。在 Apple iPhone5 的 A6 SoC 芯片上也集成了 GPU 用于加速图形显示。

学术界很早就开始了关于众核处理器的原型研制。流处理器 [42, 43] 是早期众核处理器的雏形。 Imagine [44] 是针对多媒体应用的可编程流处理器。 Imagine 中有 8 个 ALU 簇（ cluster ），而每个 ALU 簇中有 6 个浮点计算单元。 Imagine 中没有采用传统意义上的 cache ，而是采用流寄存器文件 SRF （ Stream Register File ）作为流数据的片上缓冲存储器，以提高存储带宽。 Merrimac [45] 则基于流处理器和高速互联网络构建针对科学计算的超级计算机。 Merrimac 通过互联网络连接 16 个流处理器，每个流处理器中有 16 个 ALU 簇。 Raw [46] 是 MIT 研制的基于 Tile 的高性能可重构原型处理器。 TRIPS [47] 处理器是对显式数据流执行 EDGE （ Explicit Data Graph Execution ）体系结构的实现。与此同时，关于众核处理器的研究 [48–50] 也普遍展开。相比针对延迟优化的传统 CPU ，众核处理器的处理器核通常比较简单，没有复杂的分支预测机制和大容量的片上多级存储层次，更适

合面向吞吐率的计算负载。因此，众核处理器开始被用来作为加速器，同 CPU 一起搭建异构并行系统。

多核 CPU 在市场上开始普及之后，工业界同样也开始积极探索众核处理器的发展方向。为了在 2007 年实现世界上首台 petaflop 级系统， IBM T.J. Watson 研究中心研制了 160 核 Cyclops-64 处理器 [51] 。该处理器目标是实现片上超级计算机（ supercomputer on a chip ），它也是蓝色基因（ Blue Gene ）项目的一部分。这款芯片上没有数据 cache ，取而代之的是可编程的 scratchpad 。由于暴露了很多硬件细节给程序员，要获得高性能则会造成编程相对困难。 Cyclops64 通过大规模并行隐藏访存和功能单元的延迟，这对后来的 GPGPU 产生了重要影响。为探索未来处理器的扩展前景， Intel 提出了 Tera-scale Computing 研发项目，包括 Teraflops 研究芯片和单片云计算机 SCC （ Single-chip Cloud Computer ）。 Teraflops 是 2007 年发布的以 Tile 结构集成 80 个处理器核的处理器原型，其峰值性能能够达到每秒万亿次浮点操作的级别。 2009 年推出的 SCC 则是针对云计算设计的实验性芯片，它集成了 48 个 Pentium 处理器核，以 4×6 2D-mesh 的方式互连。

在 SIMD 扩展部件的普及和众核处理器的蓬勃发展的基础上， GPGPU [52] 的提出真正使众核加速器得到广泛使用。 GPU 作为专门的图形处理单元，其体系结构专门针对图形处理应用进行设计和优化。由于像素点（ pixel ）之间通常不存在相关（ dependency ），对于各个像素点的计算操作可以高度并行。这种无相关或者相关度很低的高度并行计算模式被称为数据级并行（ data level parallelism ）。根据 GPU 的设计特点，研究人员发现 GPU [53] 不仅可以用来进行图形处理，还可以对通用程序中的数据并行内核（ kernel ）进行加速。 2007 年 NVIDIA 推出 CUDA [3] 编程模型，使 GPGPU 开始真正得到编程社区的广泛关注。由于 GPU 针对图形处理应用进行设计和优化，它对加速数据并行应用具有天然的优势，相比 CPU 而言效能更高。 2008 年， Apple 同 AMD 、 IBM 、 Qualcomm 、 Intel 和 NVIDIA 组成 Khronos Group 联合发布了 OpenCL [4] 。相比 CUDA ， OpenCL 并不局限于 NVIDIA 的设备编程，这使其具有更好的可移植性。基于 CUDA 或 OpenCL 的编程虽然得到普及，但是 GPU 的体系结构针对数据并行设计，对于通用计算中需要面对的频繁而复杂的分支操作，以及不规则的访存模式，其效率较低。为了扩展 GPU 应用领域，适应更为复杂的通用计算需求， NVIDIA 和 AMD 开始对 GPU 体系结构及其软件栈进行优化，比如增加 Cache 的层次结构，在保证吞吐率的同时减少延迟。这些设计有效地保证了 CUDA 的性能，同时尽可能地减少程序员的负担。

为了在高端市场与 GPGPU 竞争， Intel 开始考虑为其 x86 架构的 CPU 专门设计加速器。在针对可视计算的 Larrabee 项目 [54] 、 Teraflops 项目和单片云计算机项目的基础上， Intel 提出了 MIC （ Many Integrated Cores ） [55] 体系结构。基于 MIC 架构， Intel 在 2012 年发布了采用 22nm 3D 晶体管工艺技术的 Intel Xeon Phi 协处理器 [56] 。该处理器集成了 61 个核，能够支持多达 244 个线程， SIMD 向量宽度为 512 位。它采用 GDDR5 内存，带宽达到 352GB/s 。同 GPU 类似， Xeon Phi 也是通过 PCI-e 总线同 CPU 进行通信。 Xeon Phi 不但支持类似 GPGPU 的 OpenCL 编程，同时也支持传统的 CPU 并行编程语言，包括 OpenMP [57] ， Intel TBB [58, 59] 和 Intel Cilk Plus [60] 等。这使得它相比 GPU 具有更好的可编程性，因而也得到很多关注。目前在 TOP500 超级计算机排行榜排名第一的天河二号超级计算机就大规模采用了 Xeon Phi 协处理器。不论是 CPU+GPU 还是 CPU+Xeon Phi 的异构系统，虽然其具备异构计算高效能的优势，但是从体系结构角度来看， CPU 和加速器还是分离的，特别是存储空间的分离要求 CPU 和加速器之间大量传输数据，因而不能真正意义上实现 CPU 和加速器的无缝协作，这就催生了片上异构多处理器（ HCMP: Heterogeneous Chip Multi-Processor ）。

异构 CMP 最初以相同 ISA 但处理能力不同的核集成在同一芯片上的形式提出 [61] 。这种将 ISA 相同的大核和小核集成在一起的架构相比同构 CMP 而言能够有更好的效能比，而且编程模型不需要太多的变动。但是，将各个核限制在相同的 ISA 下，很大程度地减少了处理器的异构性。不同的 ISA 通常是针对不同的目标而设计的，比如说，有的是为了使硬件实现简单，有的是为了减少代码规模，有的是为了减少内存访问，有的是为了实现更高效能的硬件设计，有的是支持领域特定指令，等等。不同 ISA 的异构 CMP 允许为各种类型的负载灵活地创建高效的多核处理器。早在 2005 年 IBM 联合索尼和东芝推出了针对 PS3 游戏机的异构多核处理器 Cell [62, 63] 。 Cell 处理器由一个 IBM Power [64, 65] 处理单元（ PPE ： Power Processor Element ）和八个基于 SIMD 的协处理器（ SPE ： Synergistic Processor Element ）构成。 PPE 处理单元是 Cell 的控制与运算中枢，而真正负责浮点运算的是八个 SPE 协处理器，适用于加速多媒体和向量处理应用。 PPE 和 SPE 相互协作，处理各自擅长的计算任务，从而提升系统性能和效能。 Cell 的多处理器技术和运算能力，开创了当时异构计算技术的先河。 Cell 处理器在设计时便将分布式处理考虑在内，它将高性能的计算任务分成更小的部分，并将这些子任务分配到多个处理单元，每个处理单元都以 4GHz 以上的速度运行。正是这种能力使得 Cell 处理器能以 192 Gigaflops 的速度运行。 Cell 处理器是工业界对片上异构处理器体系结构的早期尝试，并在 PS3 上获得了很大的成功。但是 2009 年 IBM 公司停止了 Cell 芯片的研发项目，原因是在商业上除 PS3 游戏机外， Cell 芯

片并未获得大规模的市场应用，而在技术上， Cell 中的协处理器采用特殊结构和指令集，通用性不好，且未采用缓存和虚拟存储技术。尽管如此，这对后来的片上异构处理器的研究和开发都产生了深远影响。

由于单纯地增加片上处理器核数目不具有很好的可扩展性， Intel 和 AMD 并没有继续大规模增加 CPU 上通用处理器核的数目，而是开始将 GPU 集成到处理器中。相比分离式的 GPU ，片上 GPU 性能较弱，但是功耗较低，因而更省电。另一方面，可以利用近几年迅速发展的 GPGPU 技术对应用程序进行加速。 2009 年， Intel 推出 Sandy Bridge [66] 处理器，开始将 GPU 集成到 CPU 芯片中。随后的 Ivy Bridge [67] 和 Haswell [10, 68] 处理器则进一步增强了片上 GPU 的性能。不同于 Intel 的片上 GPU 仅仅专注于图形计算， AMD 将 GPU 集成到 CPU 上用于加速通用计算。 2011 年初， AMD 发布了代号为 Llano 的异构 Fusion 处理器，即 APU （ Accelerated Processing Unit ） [69] 。 2012 年初， AMD 提出异构系统架构 HSA [11] ，为未来 CPU 与 GPU 融合的异构处理器设计了发展蓝图。同年十月， AMD 发布了基于 HSA 架构的 Trinity APU 。 2014 年 AMD 推出了 CPU 和 GPU 更进一步融合的 Kaveri 处理器。同时 Sony 推出的 PlayStation 4 也采用了基于 HSA 的 AMD APU 。 HSA 的一个目标是实现 CPU 和 GPU 真正意义上的融合，即共享内存。将 GPU 集成到芯片上之后，存储层次发生了明显的变化， CPU 同 GPU 开始共享主存和 LLC （ Last Level Cache ），在 CPU 和 GPU 之间合理分配带宽和 cache 资源对性能非常重要。同时，要充分利用好硬件资源，需要重新设计和优化针对共享存储架构的异构编程模型、编译器以及运行时系统支持。

总的来看，一方面，以 GPU 和 Xeon Phi 为代表的众核加速器作为异构系统中的一个重要加速部件，能够有效提升系统的效能。另一方面，将众核加速器集成到 CPU 芯片上是目前微处理器的主流趋势，有助于提升微处理器的效能。但是要充分发挥众核加速器的计算潜力，需要对其片上资源进行合理高效的利用，同时还必须保证编程的简单性。本文以众核加速器体系结构为基础，设计和优化其存储层次的管理策略，从而进一步提升系统效能，同时简化编程。

2.2 GPU 架构

GPU （ graphics processing unit ）的概念最先由 NVIDIA 提出。 1999 年 NVIDIA 发布了被认为是世界上第一款的 GPU： GeForce 256 。这是一款集成了转换（ transform ）、投影（ lighting ）、三角形设定（ triangle setup/clipping ）和成像（ rendering ）引擎的单芯片处理器，每秒至少能够处理一千万个多边形。 ATI 随后提出了类似的概念 VPU （ visual processing unit ）并于 2002 年发布了 Radeon 9700 。经过几年的发展， GPU 的计算能力逐步提升，其峰值性能已经超

过 CPU 。研究人员发现 GPU 不仅可以用于图形计算，还可以用于加速通用计算中的数据并行部分，即 GPGPU [70] 。为此， 2006 年 NVIDIA 推出了支持 CUDA 编程的 GeForce 8 系列（ Tesla 架构 [71] ） GPU 。 GPU 开始真正意义上成为用 GFLOPS 衡量性能的计算处理器。 GT80 首次引入了统一渲染器（ unified shader ）的概念，即完全可编程的统一处理器，也被称为流多处理器（ SM： Streaming Multiprocessor ），并且加入了程序员可见的便笺存储器（ shared memory ）和支持线程间通信的栅栏同步（ barrier synchronization ）机制。

2008 年推出的 GT200 在 GT80 的基础上扩展了功能单元和寄存器文件，引入了硬件内存请求合并（ coalescing ），并开始支持双精度浮点计算。 2010 年推出的 Fermi [7, 72] 架构在体系结构上进行了迄今为止跨度最大的调整。根据 CUDA 用户的反馈， Fermi 首次引入了 cache 存储层次和 ECC 校验，开始支持并发 kernel 执行，并对浮点性能、上下文切换（ Context Switching ）和原子操作（ Atomic Operation ）进行了优化。 NVIDIA 随后又于 2012 年和 2014 年分别推出了 Kepler [8, 73, 74] 和 Maxwell [75, 76] 架构。 Kepler 将 SM 扩展为 SMX 架构，还在 Fermi 的基础上增加了很多新的特征，包括动态并行（ Dynamic Parallelism ）、 Hyper-Q 、 Grid 管理单元（ Grid Management Unit ）、 GPU Direct 等。 Kepler 的设计不仅在性能上带来了明显提升，其效能（ performance per watt ）相比 Fermi 也提升了 3 倍。 Maxwell 的设计则更加注重提升效能，在 SMX 的基础上提出了 SMM 架构。 SMM 中将每个 SIMT 核划分成四个独立的处理块，每个块有自己的指令缓存、调度器和 SIMD 部件。同时，二级缓存被增大到 2M ，以尽量减少 DRAM 访问量。

由于本文不涉及 Kepler 中的新特征，且 Kepler 和 Maxwell 的架构改进主要是扩展功能单元和缓存容量，本文以 Fermi 架构为基准体系结构开展后续的实验，并通过可扩展性实验来说明本文提出的方法对 Kepler 和 Maxwell 同样适用。另一方面，虽然 AMD GPU 的架构 [9] 同 NVIDIA GPU 存在各种细节上的差异，但本文研究的问题同样存在，因此本文的研究对 AMD GPU 同样适用。图 2.1展示了现代 GPU 的体系结构，主要包括 SIMT 核（ SIMT core ）和多级存储层次。 SIMT 核（在 NVIDIA 和 AMD 术语中分别是 SM 和 compute unit ）是 GPU 的计算部件。 Fermi 架构有多达 16 个 SIMT 核。 SIMT 核采用相对简单的流水线，不含有乱序执行和分支预测的硬件特征。 GPU 采用类似 CPU 的多级存储层次，但是针对吞吐率进行了优化设计。 SIMT 核与内存分区（ memory partition ）之间通过片上互联网络（ on-chip interconnection network ） [77] 进行连接。

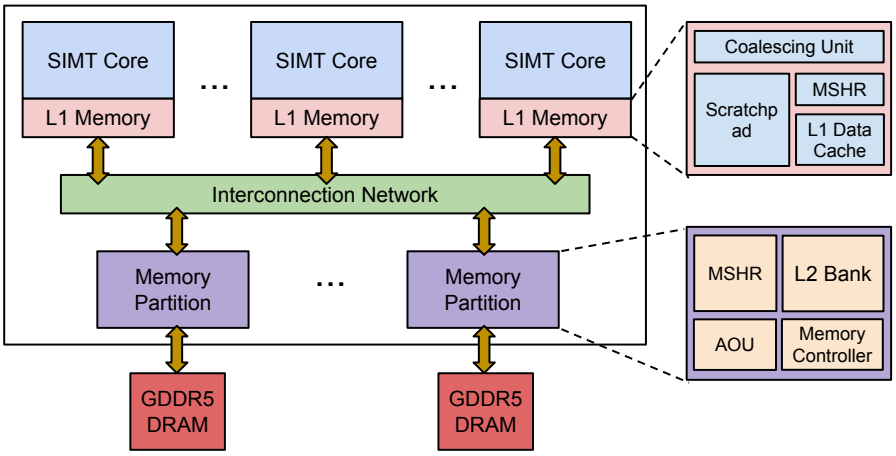

图 2.1　GPGPU 体系结构

2.2.1　线程调度

GPU 采用 SIMT 执行模型，因此其执行单元是包含多个线程的线程束（NVIDIA 和 AMD 术语中分别称为 warp 和 wavefront）。同一个 warp 中的所有线程以 lockstep 的方式并发执行。在 SPMD（CUDA 或 OpenCL）编程模型中，kernel 由多个线程块（NVIDIA 和 AMD 术语中分别称为 thread block 和 work group）组成。线程块在硬件中称为协作线程阵列 CTA（cooperative thread array）。每个 CTA 包含多个 warp。GPU 硬件启动一个 kernel 后，硬件 CTA 调度器以负载平衡（load balance）的方式将 CTA 轮转（round robin）调度到各个 SIMT 核上执行。因此，同一个 CTA 内的 warp 被保证在同一个 SIMT 核上执行。简单起见，本文不讨论 GPU 同时执行多个 kernel 的情况。每个 SIMT 核最多能够支持的 CTA 数目受限于 SIMT 核的资源，比如线程数目、暂存器（scratchpad）的容量和寄存器的数目等等。SIMT 核上的所有 warp 由 warp 调度器负责调度使用执行单元。warp 调度策略决定了访存请求的发送顺序，因而对性能和功耗有很大的影响。目前提出的 warp 调度策略有 LRR（loose round-robin）、GTO（greedy-then-oldest）[17]、two-level 调度 [78, 79]、CTA-aware 调度 [80]、CCWS（cache-conscious wavefront scheduling）[17] 和 DAWS（divergence-aware scheduling）[81] 等等。

- LRR 调度策略：LRR 算法以轮转的方式检查一个 warp 是否就绪执行，因此每个 warp 基本上得到相等的执行机会。LRR 尽可能地让 SIMT 核上所有的 warp 处于活跃状态，让这些 warp 都能访问它们的工作集（working set），因此对于访存密集型的应用经常会导致 L1D cache 冲突，从而使系统性能降级。

- GTO 调度策略： GTO 算法优先调度一个 warp 执行直到它暂停（ stall ），然后根据 warp 的选取先后顺序调度其它 warp 。 GTO 算法尽可能最小化活跃线程（即允许对外发送访存请求的线程）的总数，而且较早的 warp 有较高的执行优先权。

- CCWS 调度策略： CCWS 算法根据 L1D cache 的冲突（ thrashing/contention ）情况动态调整 SIMT 核上活跃 warp 的数目。当某些 warp 因 cache 冲突而频繁失去局部性（ lost locality ）时， warp 调度器选取部分 warp 将其挂起（ suspend ），其它 warp 则得到优先执行的机会，这样就减小了工作集，从而缓解了 L1D cache 的冲突问题。

2.2.2 流水线微体系结构

图 2.2展示了 SIMT 核内部的流水线微体系结构。每个 SIMT 核可以支持多个 warp 同时执行。每个 warp 的信息包含 warp ID 、 active mask 和程序计数器 PC 。 active mask 中的每一位指示其对应的线程是否活跃。当 warp 刚被创建时，其中的每个线程都是活跃的。 SIMT 核中有数个（ Fermi 有两个， Kepler 有四个） warp 调度器，负责维护每个 warp 的执行上下文，并调度并发的 warp 开始取指并占用流水线资源。被调度的 warp 将经历取指（ fecth ）、译码（ decode ）、指令发射（ issue ）、读操作数（ read operands ）、执行（ execution ）、访存（ memory access ）、写回（ write back ）等操作。 warp 的取指和译码过程跟 CPU 中类似。指令被缓存在 I-cache 中。指令发射阶段由一个轮转仲裁器选择一个 warp 发射指令。每个 warp 有相应的寄存器文件。相比 CPU ， GPU 中的寄存器资源规模大很多，这是为了满足成百上千个并发线程的需求。然而，由于总的寄存器数量毕竟有限，如果每个 warp 对寄存器的需求太高，则会限制并发执行的 warp 的个数。在读取操作数阶段， warp 中所有线程的寄存器值根据其 warp ID 和寄存器 ID 被并发地从寄存器文件中读取出来，然后发送给流水线的后端。 GPU 的计算部件是包含 N 个 lane 的 SIMD 向量单元，每个 lane 具有独立的 ALU 部件进行计算操作。访存单元（ memory access unit ）处理 Load 和 Store 指令的访存操作，必要时将访存请求发射到存储子系统中去。写回部件将计算结果写回到相应的寄存器。然后该 warp 的 PC 和 active mask 被更新，继而重新进入调度队列。

2.2.3 GPU 存储层次

GPU 的存储层次由寄存器、一级缓存、二级缓存和片外 GDDR DRAM 构成。简单起见，本文不讨论纹理缓存（ texture cache ）等特殊用途的缓存。一级缓存包括便笺存储器（ scratchpad ）和 L1 数据缓存（ L1D cache ）。一级缓存是各

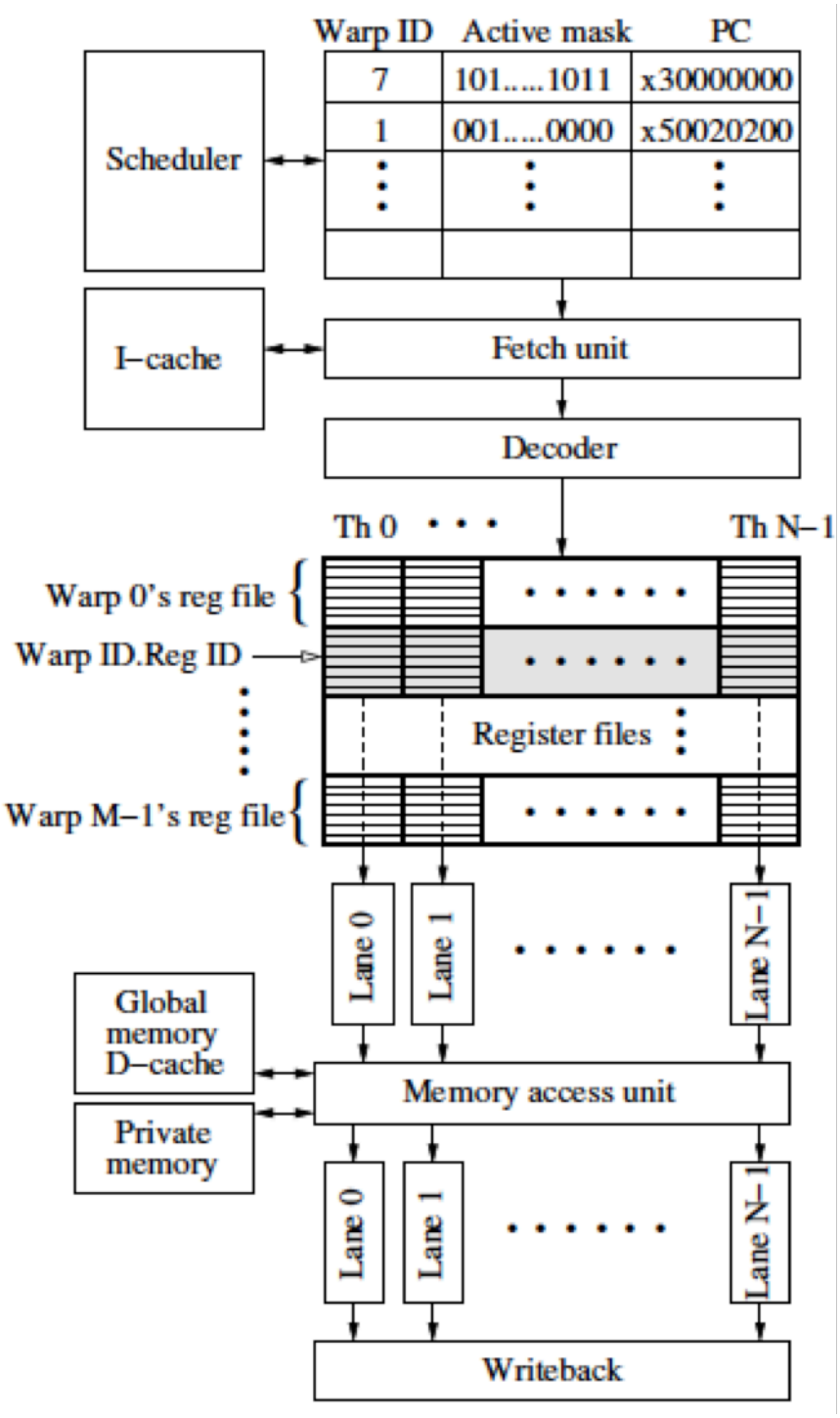

图 2.2　GPU 流水线微体系结构 [78]

个 SIMT 核所私有的，但是被 SIMT 核上的所有线程束（warp）所共享。便笺存储器（NVIDIA 和 AMD 术语中分别称为 shared memory 和 local memory）是程序员可见的。内存被划分成多个分区（memory partition），每个分区包含一个 L2 cache bank。同一个线程束生成的全局（global）访存请求首先在硬件合

并单元（coalescing unit）进行合并，然后发送给 L1 数据缓存。命中的请求将被立即响应，而失效的请求将会被记录到 MSHR （Miss Status Holding Registers）中，然后通过片上互联网络发送给 L2 缓存。 L2 缓存被所有的 SIMT 核共享，因而是整个系统内存一致性的汇合点。在 L2 缓存中失效的请求将被发往内存控制器（MC： memory controller）中的请求缓冲（request buffer）。每个内存分区的 MC 同相应的内存通道（memory channel）相连，按照 FR-FCFS （First-Ready First Come First Serve） [82] 的调度原则将请求发往 GDDR DRAM 。原子操作由内存分区中的原子操作单元（AOU： Atomic Operation Unit）负责完成。根据 Sara 等人 [83] 的测试， Fermi GPU 没有实现硬件预取和 victim cache ，因此本文不予讨论。

为了最大化吞吐率， GPU 对其内存子系统进行了优化 [84] 。第一，请求合并。通过对同一个线程束的访存请求进行合并， GPU 能够节省内存带宽。如果可能的话，所有的请求在被送往 L1 数据缓存之前都会由硬件合并单元进行合并，因此如果程序员恰当设计程序中的访存模式，硬件合并单元能够捕获大量的空间局部性。第二，写策略 [85] 。 GPU 的 L1 缓存通常采用写直达的策略，不同的是， AMD 和 NVIDIA 的 GPU 分别采用写分配和写不分配策略，即在 AMD GPU 中 L1 缓存会对写失效进行缓存块（cache clock）分配 [9] ，而 NVIDIA GPU 中写失效会被 L1 缓存直接旁路 [7, 8] 。由于 GPU 应用中很少有对被写过数据的重用，相比写回策略，写直达能够节省带宽 [86, 87] 。 L2 缓存则采用写回和写分配策略，这同 CPU LLC 中常采用的策略一致。 L2 缓存的写回策略减少对片外 DRAM 的访问量，写分配使得不同 SIMT 核能够进行片上通信（写后读相关）。第三，一致性 [88, 89] 。由于 SPMD 模型不提倡进行 SIMT 核之间的通信，现代 GPU 通常不支持 L1 缓存的硬件一致性协议。这样设计一方面降低硬件复杂度，另一方面，消除了维护一致性协议产生的消息传递减少了片上网络带的数据传输负担，因而也降低了访存延迟 [86, 87] 。第四，包含性 [90] 。当前 GPU 的 L2 缓存没有限定包含性，事实上， NVIDIA 采用的是非包含 - 非不包含 （NINE: non-inclusive non-exclusive）策略 [86, 91] 。具体来说， DRAM 返回的数据块会被插入到 L1 和 L2 缓存中，因此会出现某个缓存数据块（cache line）既在 L1 缓存中，也在 L2 缓存中的情况（non-exclusive）。但是当 L2 缓存中发生替换而移除某个缓存数据块时，不需要通知 L1 缓存，即 L1 缓存能够继续保有这个数据块。因此会出现某个数据块在 L1 缓存中却不在 L2 缓存中的情况（non-inclusive）。这种设计相比包含（inclusive）缓存能够减少冗余数据拷贝，而仍然允许 L1 缓存局部地提供共享数据。

SIMT Core	16 cores, 1.4GHz, 5-Stage Pipeline, SIMT width = 32
Resources / Core	48KB Scratchpad, 32768 Registers, 1536 Threads, 48 warps
L1 Data Caches / Core	32KB, 4-way, 128B line size
L2 Cache Bank	128KB, 16-way, 128B line size, 700MHz
Features	Coalescing enabled, 32 MSHRs/core
Scheduling	LRR warp scheduling, round-robin CTA scheduling
Interconnect	2D Mesh, 1.4 GHz, 32B channel width
DRAM Model	FR-FCFS, 8MCs, 4 DRAM banks/MC, 2KB row size
GDDR5 Timing	1.4 GHz, tCL=12, tRP=12, tRC=40, tRAS=28, tRCD=12, tRRD=6

表 2.1　基准 GPU 的模拟参数配置

2.2.4　实验方法

本文选取周期精确（cycle accurate）的 GPU 模拟器 GPGPU-Sim v3.2.0 [92] 作为实验平台。GPGPU-Sim 支持 CUDA 程序及其 PTX ISA [93] 的模拟，其模拟 NVIDIA GPU 的精度达到 97.3%。表 2.2中列出了本文中基准架构的模拟器参数。基准架构模拟类似 NVIDIA Fermi GPU [7]，并采用了详细的 GDDR5 DRAM 模型。线程束调度器采用最基本的 LRR 策略，但是后续的实验中我们也会使用更先进的 GTO 策略作对比。本文采用 GPUWattch [94] 估算 GPU 系统的功耗。

2.3　编程模型和应用

为了实现多核处理器的性能扩展，普及并行 [95] 是体系结构研究的一个重点问题。虽然并行编程已经发展了几十年的时间，长期以来也积累了很多并行编程语言和相应的软件栈，但是相比编写串行程序，并行编程要求程序员重新设计算法和数据结构，且使用面向底层硬件架构的各种优化技术，使之成为一个复杂耗时且易出错的过程。异构系统集成各种类型的处理引擎，其编程模型就更为复杂，涉及到任务的划分和调度，以及性能可移植性（performance portability）等问

题。因此，可编程性（programability）或者说编程效率（productivity）成为异构并行系统设计的重要指标。

常见的并行编程模型主要分为两类：共享内存（shared memory）和消息传递（message passing）。共享内存模型通过共享变量作为媒介实现处理器之间的通信。由于线程共享内存地址空间，不存在数据传输和冗余拷贝的问题。目前常用的共享内存并行编程模型有 Pthreads、OpenMP 和 TBB 等。对于分布式多处理器，由于并发的任务不属于同一个进程地址空间，通常采用以 MPI [96] 为代表的消息传递编程模型，即程序员通过插入通信原语（primitive）来实现进程之间的同步和通信。现代大规模的并行应用通常采用混合方式编程，在节点之间通过 MPI 实现粗粒度的任务划分，在节点内部则采用 OpenMP 或者 TBB 等细粒度的编程模型来利用多核资源。

为了简化并行编程的难度，提高开发效率，研究人员提出了很多相应的软硬件支持。事务型内存（TM： Transactional Memory）[97] 是针对多线程共享变量锁（lock）机制而提出的一种简单易用的并行任务同步方法。TM 通过软件或者硬件方法保证事务的原子性和隔离性，因而能够达到同粗粒度（coarse-grain）锁一样的易用性，以及同细粒度（fine-grain）锁一样的性能。自动并行（automatic parallelism）是对串行程序采用软件或硬件方法进行自动并行化的技术。例如，线程级猜测技术 [98–100]（TLS： Thread-Level Speculation）就是一种基于线程的自动并行化实现。采用任务模型的细粒度并行机制 [101–103] 由运行时系统通过工作窃取（work stealing）的方式实现动态负载平衡，而程序员则只需要将注意力集中在并行算法上，从而能够提升开发效率。

在异构硬件越来越普及的趋势下，针对未来异构系统的编程模型研究也层出不穷 [104]。EXOCHI [105] 提供对加速器的抽象，允许程序员进行显式管理，帮助程序员对异构硬件进行自动目标化。Merge [106] 在 EXOCHI 的基础上进一步提供关联各种 kernel 与特定加速器的框架，由运行时系统负责选取合适的 kernel，并分派任务到不同异构设备上。Delite [107, 108] 是基于领域特定语言 (DSL: Domain Specific Language) 的隐式并行框架，负责将高级应用代码自动映射到异构硬件设备上。由于 DSL 语言针对特定领域设计，通常简单易用，因而能够显著提升程序开发效率。Delite 不但通过自动目标化降低了异构并行的复杂度，还通过中间层抽象使得 DSL 语言的创建变得容易。Harmony [109] 为整个程序构建数据相关图，然后调度不相关的 kernel 并行执行。

2.3.1 GPU 编程模型

GPU 编程起源于着色语言（shading language）[110]，例如 HLSL、Cg 等。但是着色语言编程困难，程序员不仅要熟悉并行编程技术，而且要了解底层硬

件和图形库细节。流编程语言的出现对 GPU 编程模型的发展起到了重要的推动作用。流编程（streaming programming）模型是针对流处理器的计算映射模型。Stream C/Kernel C [111] 是针对 Imagine 设计的流编程语言，其编译器负责优化调度。StreamIt [112] 则是针对 RAW 处理器开发、以 Java 为基本语法的流处理语言。可编程渲染器（Programmable Shaders）的出现开启了流编程模型在 GPU 上的应用。Brook [113] 流编程语言是针对 GPU 通用计算的编程语言，它在 C 和 Fortran 语言的基础上扩展，为用户提供运行时库，允许其创建和管理流。Brook 为 GPU 通用编程的发展奠定了基础，成为后来 CUDA 的雏形。

CUDA [3]（Compute Unified Device Architecture）是 NVIDIA 针对 GPU 通用编程推出的并行编程框架，采用 SPMD（Single Program Multiple Data）数据并行编程模型，在很大程度上降低了 GPU 编程的难度，这使得 GPGPU 开始得到广泛应用 [114, 115]，特别是在高性能科学计算领域。然而 CUDA 只针对 NVIDIA GPU 设计，缺乏对其它加速器的兼容性。为此，Khronos Group 工作组推出了面向异构系统可移植的并行编程标准 OpenCL [4]（Open Computing Language，开放计算语言）。OpenCL 是一个统一的异构编程环境，便于程序员为高性能计算机、桌面计算系统乃至手持设备编写并行代码，而且广泛适用于多核 CPU、GPU、DSP 和 FPGA 等各种不同的计算设备。OpenCL 由一门基于 C99 用来编写 kernel（设备函数）的语言和一组用于定义并控制平台的 API 组成。

典型的 SPMD 程序由主机（host）即 CPU 启动，并负责将数据发送到设备（device）即 GPU 的显存，并通过驱动（diver）发送命令（command）给 GPU，启动 GPU 计算数据并行内核（kernel），计算结束后再将结果数据传输到 CPU 的主存。图 2.3显示了 SPMD kernel 的编程模型。Kernel 中的所有线程，即工作项（work-item）组成一个线程网格（grid）。线程网格是三维层次结构，一个线程网格包含多个工作组（work-group）。每个工作组又包含多个子工作组（sub-group），即线程束（warp 或 wavefront）。线程束中的所有工作项（即线程）以锁步（lock-step）的方式执行，即在同一时间并发地执行相同的操作。

2.3.2 高级编程语言和编译支持

CUDA 和 OpenCL 虽然很大程度上简化了 GPU 通用编程的难度，但是相比传统的串行编程语言，它们仍然非常复杂，要在 GPU 上获得较好的加速，程序员需要了解 GPU 的硬件细节，针对不同的计算和访存模式进行优化 [117, 118]。为了进一步简化 GPU 编程，研究人员在 CUDA 和 OpenCL 的基础上提供自动的源到源（source to source）编译支持，或者提供高级别的抽象，使程序员尽可能地关注算法本身，而不需要考虑太多并行化的实现细节和底层硬件细节。

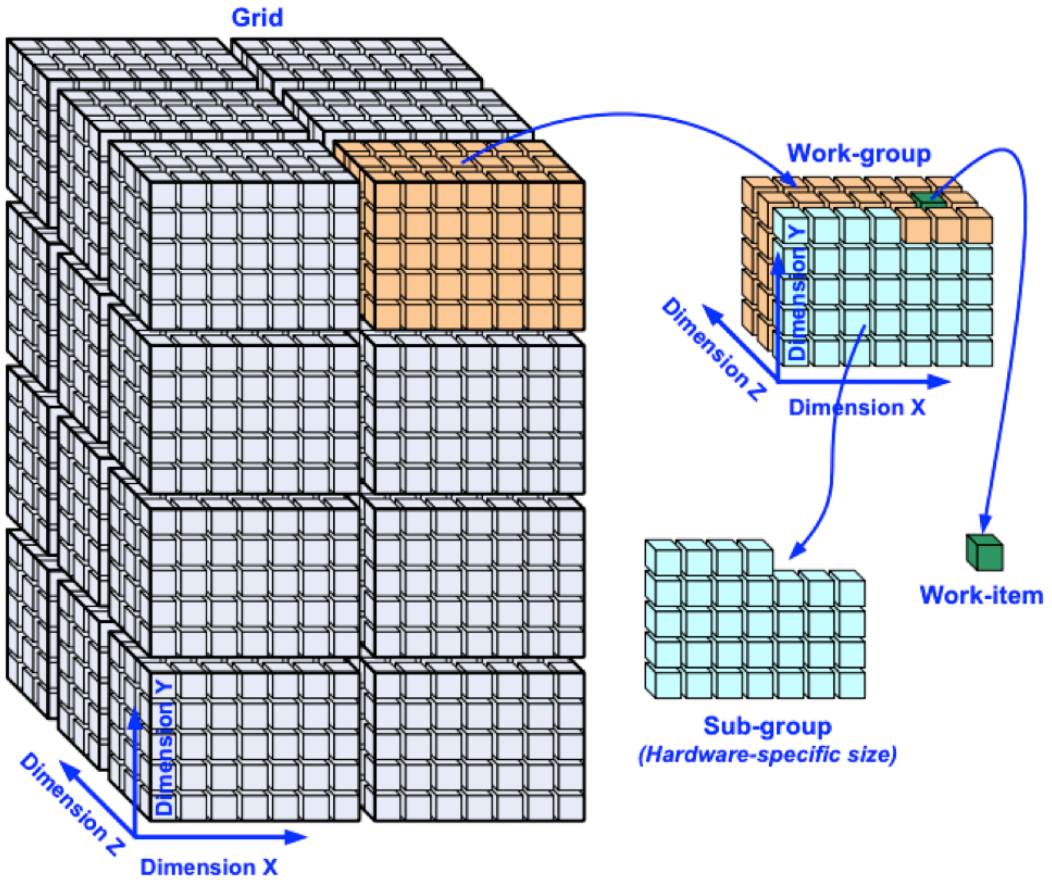

图 2.3　SPMD 编程模型 [116]

MCUDA [119] 和 Ocelot [120] 允许面向 GPU 编写的 CUDA 代码运行在多核 CPU 上，使得在系统中没有 GPU 的情况下，CUDA 代码也能正确执行，而且获得相对串行代码较好的性能加速。与之类似的，MxpA 编译框架 [121] 和 Twin Peaks [122] 运行时系统允许 OpenCL 代码运行在 CPU 上，保证了 OpenCL 的性能可移植性（performance portability）。GMAC [123] 运行时系统负责 host 同设备之间的数据转移，简化了 CUDA 编程，同时通过隐藏数据转移的延迟能够达到同手工代码一样的性能。其它编译支持 [124–126] 自动将 C 或者 OpenMP 程序翻译成 CUDA。Qilin [127] 在 TBB 和 CUDA 之上建立更高层次的抽象，允许程序员在编写代码时不需要考虑底层硬件细节，运行时系统将会在程序执行时根据动态 profile 的信息进行任务划分，使得 CPU 和 GPU 能够有效地协同工作，这样能够更好地利用计算资源，进一步提升性能。

基于 CUDA 的 C++ 模板库 Thrust [128] 和基于 OpenCL 的 C++ 模板库 Bolt [129] 为程序员提供了常用计算模式的模板，简化了编程。C++ AMP [130] 是 Microsoft 针对 CPU-GPU 异构系统推出的编程语言，相比 CUDA 和 OpenCL，C++ AMP 的抽象层次更高，通过扩展 C++ 语法提供对多维数组、索引、数据转移和 tiling

的支持，从而提高了开发效率。由 PGI 和 Cray 等公司牵头推出的 OpenACC [131]，是同 OpenMP 类似的数据并行加速器编程模型， OpenACC 允许在 C 、 C++ 和 Fortran 程序中插入编译制导语句，以指导编译器对 Kernel 代码进行并行化。编译器负责处理数据在 CPU 和加速器之间来回转移的逻辑关系，并将计算映射到适当的处理器上。 Copperhead [132] 是嵌入到 Python 中的高级数据并行编程语言。Copperhead 程序员通过组织数据并行原语来描述并行计算，它支持对数组数据的扁平（ flat ）和嵌套（ nested ）数据并行计算。在语法上， Copperhead 程序使用被广泛接纳的 Python 编程语言的子集来表达。编译器负责将 Copperhead 编译成高效的 CUDA C++ 代码。运行时系统支持 Copperhead 程序同包括数值计算、数据虚拟化和分析在内的标准 Python 模块进行交互，因此相比手工优化的 CUDA 代码， Copperhead 代码量减少了 3.6 倍，但是能够获得其 45%-100% 的性能。

2.3.3 GPU 应用

前面提到， CUDA 和 OpenCL 的推出极大促进的 GPU 通用编程的普及，目前在很多科学、工程计算领域已经积累了大量的 GPU 代码 , 包括石油勘测、天气预报、核模拟、流体力学模拟、分子动力学仿真、生物计算、图像处理、音视频编解码等等。传统的 GPU 硬件非常适合执行规则的数据并行应用，然而通用计算领域中更多的应用程序是不规则的。近几年的研究表明 [133–135] ，通过不同程度的优化，不规则应用也能够在 GPU 上获得很可观的性能加速，但是一方面这严重增加了程序员的编程负担，另一方面程序的过分优化可能导致其性能可移植性很差。更重要的是，不是所有的不规则应用都能够通过软件优化来有效应对的。

GPU 应用的不规则主要分为两种情况：控制和访存。这就对应了 GPU 中两个主要的问题：分支分歧（ branch divergence ） [136] 和访存分歧（ memory divergence ） [137] 。分支分歧是指同一个 warp 中的各个线程执行分支指令的路径不同，这会导致计算资源的利用率不高，因而系统效率低下。由于该问题的重要性，相关的研究非常多 [138–143] 。访存分歧是指同一个 warp 中的各个线程访问不连续的内存位置，因而每条访存指令会产生大量的访存请求，导致存储子系统的低效。访存分歧问题是本文关注的研究点，相关研究在后续章节会详细讨论。

GPU 的一个主要特点是通过大规模多线程掩盖访存延迟。如果访存分歧频繁发生，就会产生大量访存请求，从而导致访存延迟无法被有效掩盖。当计算无法掩盖访存延迟的时候，应用程序的性能开始严重受限于存储子系统的性能。本文首先从标准测试程序集 [144–148] 中选取访存受限的程序。表 2.2中列出了本文中使用的基准测试程序。图2.4中显示了理想存储系统（ Perfect Memory ）下各个测试程序能够获得的性能加速。所谓理想存储系统是指程序中所有的访存请求都在一

Benchmarks	Description	Suite
Highly Cache Sensitive (HCS)		
BFS	Breadth First Search	[144]
KMN	K-means Clustering	[144]
PVC	Page View Count	[145]
SSC	Similarity Score	[145]
PVR	Page View Rank	[145]
IIX	Inverted Index	[145]
WC	Word Count	[145]
Moderately Cache Sensitive (MCS)		
SM	String Match	[145]
SPMV	Sparse Matrix Vector Multiply	[146]
FFT	Fast Fourier Transform	[146]
CFD	CFD Solver	[144]
NW	Needleman-Wunsch	[144]
Cache Insensitive (CI)		
SD1	Graphic Diffusion	[144]
BP	Back Propagation	[144]
STL	Stencil	[146]
WP	Weather Prediction	[147]
FWT	Fast Walsh Transform	[147]

表 2.2　GPU 基准测试程序，根据程序的缓存敏感性分为三类： HCS 、 MCS 和 CI

个周期内被服务（得到数据响应）。如果 L1 数据缓存的命中延迟被理想化地设定为一个周期的话，那么理想存储系统就意味着所有访存请求都会在 L1 数据缓存中命中。图中看出，这些程序的性能都能够明显受益于理想存储系统。

图2.4中的性能收益分为两方面，一是 DRAM 带宽，二是缓存性能。也就是说，对于其中局部性差或者不存在数据重用的程序，例如流式（ streaming ）应用，只能够通过增大 DRAM 带宽来提升系统性能，而对于另一部分程序，提高缓存的效率能够降低访存延迟，同时节省存储带宽，因此也能够获得性能升级。对于第二类负载，由于存在时空局部性特征，它们会呈现出缓存敏感性，即增大缓存容量能够有效提升性能。相反地，因为访存模式不存在数据重用，第一类负载是不存在缓存敏感性的。根据测试程序对缓存的敏感程度，本文将它们分为三组，分别是缓存高度敏感（ HCS： highly cache sensitive ）、缓存微敏感（ MCS：

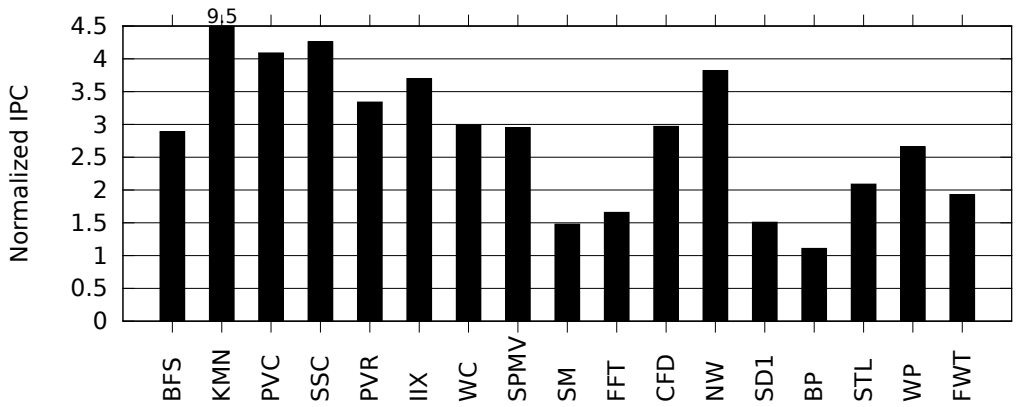

图 2.4　理想存储系统下各个测试程序的性能获得的加速

moderately cache sensitive）和缓存不敏感（CI： cache insensitive）。这些负载的缓存敏感性分析将在第4.1节进行详细介绍。

2.3.4　本章小节

本章首先介绍了众核加速器的起源和发展历程，指出众核加速器已经成为现代计算系统中不可或缺的一部分，并且在未来相当长的一段时间仍将占据很重要的地位。然后，我们介绍了目前计算系统中最常见的众核加速器 GPU 的体系结构，讨论了这种数据并行加速器针对吞吐率的体系结构设计权衡。最后，我们还讨论了这种众核加速器的编程模型及其应用程序，一方面 GPGPU 已经在很多科学和工程计算领域得到了广泛应用，另一方面，我们也可以看出， GPGPU 编程模型对程序员的要求仍然较高，特别是需要程序员对 GPU 的体系结构和存储层次有深入的理解，这在很大程度上阻碍了 GPGPU 的进一步推广。因此，对 GPU 体系结构和编程模型的优化，特别是提供针对不规则应用的支持，是目前针对众核加速器体系结构的主要研究方向。

第三章　相关研究工作

本章介绍与本文相关的研究工作，并将它们与本文中开展的研究工作进行对比。前面提到，处理器产业面临存储墙、功耗墙和编程墙三个主要问题。缓存存储层次 [149] 一方面通过数据重用减少片外 DRAM 访问量，降低访存延迟，节省存储带宽，另一方面硬件自动捕获时空局部性，不需要程序员精心设计访存模式，提高可编程性。但是缓存本身增加了芯片的静态和动态功耗，而且当缓存成为性能瓶颈时，计算引擎的效能也会降低。因此，多线程环境下围绕缓存层次结构管理问题开展的研究主要分为几个方面：

- 缓存替换和旁路策略的优化。为了尽可能多地重用缓存中数据，缓存替换和旁路策略至关重要。在选取替换 vicitm 时，应该尽可能选取未来最不可能被重用的缓存数据块。在处理请求时，如果该请求在未来被重用的可能性远低于目标组（target set）中的所有缓存数据块，则应该将这个请求旁路。

- 调控并行度以充分利用片上资源。芯片的片上资源，包括各级缓存、 NoC 带宽和 DRAM 带宽都是有限的。而当这些资源不足以满足多线程应用程序的需求时，增加并发线程的数目就无法提升系统性能，甚至会造成对共享资源的竞争而使得性能降级。应用程序的资源需求往往还取决于输入数据集的规模，因此需要在运行时动态控制线程的数目。

- 利用功耗控制机制提升效能。现代处理器提供控制功耗的硬件机制，比如 DVFS 和核扩展。当存储子系统成为性能瓶颈时，降低处理器核的工作频率可以节省功耗。同样地，通过门控技术将部分处理器核置于低功耗状态也能达到类似的目的。功耗控制策略的核心问题是如何对工作频率和活跃核进行调整，以降低功耗、提升系统效能，同时又要保证性能不会显著降级。

3.1　替换和旁路策略

CPU 中的 cache 替换策略对 cache 的性能有着直接的影响，因此一直以来都是研究重点 [150–154] 。替换算法的目标是找到目标组（target set）中在未来最不可能被重用的缓存数据块，作为被替换的 victim 。单处理器系统中的 Belady 算法 [155] 是理论上最优的 cache 替换策略，它基于 oracle 的访存序列信息，找出重用距离最远的缓存数据块作为 victim 。 LRU （Least Recently Used） [156] 是最为

常见的 cache 替换算法。它根据每个缓存数据块的时间戳，选择最长时间未被使用的缓存数据块作为替换的候选。 LRU 算法简单，不需要复杂的控制逻辑，而且对于局部性较好的程序该策略非常高效。但是 LRU 的实现开销较大，因此在实际的处理器中常常采用其近似算法 pseudo LRU [157] 或者 NRU （ Not Recently Used ） [158] 。

然而，现实中的应用程序存在着各种各样的访存模式，这使得 LRU 策略没有办法有效应对。主要的改进策略包括防冲突（ thrashing-resistant ） [159, 160] 算法和防污染（ pollution-resistant ） [12, 13, 161–163] 算法。 Qureshi 等 [160] 提出了 BIP （ Bimodal Insertion Policy ）算法，它随机选择一部分缓存数据块驻留在 MRU （ Most Recently Used ）位置，而其它大部分的缓存数据块则被置于 LRU 位置。在工作集远大于 cache 容量时， BIP 能够允许部分工作集驻留在 cache 中。但是对于没有冲突的应用， BIP 会降低性能。因此他们还提出了组竞争（ Set-Dueling ）机制，通过比较在多种替换策略之间动态地选取较优的策略。 DIP 则是在 BIP 的基础上应用组竞争（ Set-Dueling ）机制，在 BIP 和 LRU 之间自适应地选择。

RRIP [12] （ Re-reference Interval Prediction ）是在 NRU 的基础上进行改进的 cache 替换策略。 SRRIP （ Static RRIP ）针对 scan 访问模式（一连串重用距离非常大的请求）设计相应的替换策略以避免 cache 污染。 SRRIP 采用的预测机制的合理性在于：被重用过的缓存数据块很有可能被再次重用。在 SRRIP 策略中每个缓存数据块被赋予一个 n 位的 RRPV （ Re-reference Prediction Value ），并被初始化为 $2^n - 1$ 。在缓存数据块被插入到 cache 中（即失效）时， RRPV 被置为 $2^n - 2$ 。而在 cache 命中发生时，命中的缓存数据块的 RRPV 被置为 0 。 RRPV 表示了缓存数据块的重要性， RRPV 越小，则越有可能在不远的未来被重用，因而越不应该被选为替换的 victim 。每当 cache 替换发生时， $RRPV = 2^n - 1$ 的缓存数据块将被选为 victim 。如果目标组中不存在这样的缓存数据块，则将组里面的每个缓存数据块的 RRPV 都自增 1 ，直到找到一个 $RRPV = 2^n - 1$ 的缓存数据块为止。由于区别对待命中和失效的缓存数据块， SRRIP 能够减轻 scan 访问模式对 cache 的污染效应。值得说明的是，当 $n = 1$ 时， SRRIP 退化为 NRU 。因此 NRU 本质上是 RRIP 的一种特殊形式。而 $n > 1$ 时的 SRRIP 策略相比 NRU 则是通过增加硬件成本提高预测准确度的一种权衡方式。 BRRIP （ Bimodel RRIP ）在 SRRIP 的基础上，用一定的概率在 $2^n - 2$ 和 $2^n - 1$ 之间选择 cache 失效时的 RRPV 赋值，即采用类似 BIP 的思想缓解 cache 冲突。同样类似的， DRRIP （ Dynamic RRIP ）应用组竞争机制在 SRRIP 和 BRRIP 两种策略之间进行动态选择。

然而， RRIP 还是不能完全解决 cache 污染和冲突问题。 SRRIP 的替换策略是静态确定的，无法根据实际程序运行时的情况进行调整，虽然 DRRIP 能够在

SRRIP 和 BRRIP 之间动态选择，但它也不可能覆盖各种不同的访存行为。其根本原因是， RRIP 没有在运行时动态获取程序的阶段性访存行为模式，即重用距离（ RD： reuse distance ）。 PDP cache [13] 则是对 RRIP 进一步的一般化。 PDP 引入保护距离（ PD： protection distance ）的概念。 PD 的作用是保护其对应的缓存数据块在一定次数的访存序列中不被替换。每个缓存数据块会被赋予一个 n 位的 RPD （ remaining PD ）。在缓存数据块被插入到 cache 中时， RPD 被置为初始 PD 。每次访问会使目标组中的所有缓存数据块的 RPD 自减 1 。当 $RPD = 0$ 时，对应的缓存数据块则出于未保护（ unprotected ）状态。除了 RPD 之外， PDP 还使用一个 reuse-bit 来表示对应的缓存数据块是否被重用过。每当 cache 替换发生时，首先查找是否存在未保护的缓存数据块，如果有，则选为 victim ，否则在 reuse-bit 为 0 的缓存数据块中查找 RPD 最大的作为 victim 。如果不存在 reuse-bit 为 0 的缓存数据块，则在所有缓存数据块中查找 RPD 最大的作为 victim 。 SPDP （ static PDP ）使用预先确定好的初始 PD ，而动态 PDP 则在运行时动态计算初始 PD 的值。动态 PDP 在每个组添加一个 FIFO 以记录访存请求的历史信息，同时用一个计数器数组来记录重用距离的 histogram ，即每当一个传入请求（ incoming request ）在 FIFO 中命中，则计算两次访问之间的距离，并将这个距离对应的计数器的值加 1 。在一个采样周期结束之后，一个专用的流水线会根据这个计数器数组中的 histogram 数据计算出最佳的初始 PD 值。这个值将会在下一个采样周期中被应用到所有的缓存数据块中。

cache 替换策略能够在一定程度上缓解 cache 污染和冲突的问题，但是无法完全避免，比如，即使对传入请求的重用预测是 100% 的准确，也无法避免死块被插入到 cache 中，因为每个传入请求都必须插入到 cache 中。如果要解决这个问题，就必须为传入请求提供一个另一个数据通路，即旁路（ bypass ）通路 [164–166] 。在旁路通路的支持下，对于每个传入请求就有两种处理方式：被插入到 cache 中，或者被旁路而直接送至下一级存储层次。这就出现了如何决定哪些传入请求被插入到 cache 中而哪些被旁路的问题，即 cache 旁路策略。研究人员提出了各种 cache 旁路策略 [167–170] 来避免 cache 污染和冲突。 Gaur 等 [171] 基于往返数（ trip counts ）和使用数（ use counts ）提出了针对不包含（ exclusive ） LLC 的选择性旁路算法。 Kharbutli 等 [172] 提出的 LLC 旁路算法采用了基于计数器的预测表（ prediction table ）。 PDP cache [13] 中也提出了基于保护距离的旁路策略：如果不存在未保护的缓存数据块，传入请求就会被旁路。 PDP 的旁路策略实质上是对每个缓存数据块设置一个保护距离，在这个距离之内的缓存数据块都不会被替换。因此，如果 PD 预测是准确的话， PDP 能够最大化 cache 的命中率。本文中

提出的 cache 管理策略从 PDP 旁路策略出发，将其引入到 GPGPU 中进行实现和优化，然后协同 warp 调节机制进一步提升性能。

Anssari [91] 提出硬件监测访存分歧的机制，在某个 warp 产生的访存请求太多时触发旁路机制。 Choi 等 [173] 提出了 GPU 中读请求的旁路策略。该旁路策略主要为了防止 LLC 被只用于 CTA 内部的流数据（ streaming data ）污染，从而保证 LLC 能够有效地用于 CTA 之间的通信。相比而言，本文考虑多级 cache 的旁路策略，而且策略是基于重用距离的预测。 Xie [174] 等提出了利用编译器技术自动旁路部分访存指令发出的请求。该方法不需要任何额外的硬件支持，但是静态分析的准确度较低，且无法应对输入数据相关的访存模式等动态特性，其效果非常有限。死块预测技术通过相应的预测机制找出死块（ Dead block ），目的是尽快将这些死块从 cache 中剔出或者对其旁路，以避免其污染 cache 。现有的死块预测算法包括基于软件的死块识别机制 [175] 、基于指令序列的预测器 [176] 、基于计数器的预测器 [172] 和基于 burst 的预测器 [177] 等。

MRPB [178] 中运用旁路机制减少 GPU 的 L1 数据缓存中 warp 内部（ intra-warp ）的冲突。旁路机制在由于资源短缺而导致的暂停发生时被触发，一种常见的场景是，在很短的时间内一连串请求访问同一个 cache 组，导致目标组中没有足够的路（ way ）可以服务传入请求。在大规模多线程环境中，这种访存模式出现在有大量访存分歧的程序中，而这种情况下该旁路策略的改进效果非常明显。但是这种旁路策略是针对这种特定的访存模式进行设计的，对于其它访存模式效果不明显，本文中的旁路策略则更注重解决一般化的 cache 冲突问题。此外， MRPB 采用的策略也可以被集成到本文中的旁路策略中来，从而相互能够促进。

3.2 线程调节

线程调节（ thread throttling ）技术 [15, 16, 179, 180] 也是多核处理器系统中研究热点。 Suleman 等 [15] 提出反馈驱动的多线程（ FDT： Feedback- driven Threading ）。 FDT 在多线程应用程序性能受限于数据同步或者存储带宽时调整并发线程的数目。在这个框架中，循环的前几次迭代被用于训练（ training ）。在训练过程中，只允许一个线程运行，其数据同步和带宽使用的总量被动态监测。在训练结束之后，该框架根据监测的信息和一个性能解析模型计算出最优的线程数目，接下来的执行则采用这个最优数目启动线程并发。本文没有使用训练的方法来估算 GPU 中每个 warp 对资源的需求情况，这是因为在大规模多线程环境中将线程数目限制得太低会导致很明显的性能降级。不同于 FDT 针对数据同步和存储带宽受限的应用， Cheng 等 [16] 在多核系统中使用线程调节技术以减少访存延迟。他们同样建立了相应的解析模型作为指导，通过限制并发的访存任务的数目来避免访

存请求之间的互相干扰。上述方法都是通过软件方法在运行时系统中做出相应的决策，而本文虽然也考虑访存冲突的问题，但是通过利用旁路策略提供的信息由硬件来调控，预测算法相对简单而易于硬件实现。

GPU 体系结构的研究中也有很多工作探索了线程调节技术。 Rogers 等 [17] 提出 CCWS （ cache-conscious wavefront scheduling ）使用一个 VTA （ victim tag array ）动态监测由 L1 数据缓存中 warp 之间的冲突而丢失的局部性（ lost locality ），当局部性丢失严重时，部分 warp 会被挂起，而不被允许向内存子系统发射访存请求，从而保证其余 warp 的局部性。 DWS [81] 则是在 CCWS 的基础上采用 cache 踪迹（ footprint ）预测进行改进。本文中提出的方法通过应用 cache 旁路策略来缓解 cache 冲突，需要增加的硬件简单且开销很低。相比单纯的 warp 调节机制，本文提出的协同 cache 旁路和 warp 调节为提升系统性能创造了新的机会。其它调度器 [78–80] 也提出了根据 cache 和内存子系统的冲突来限制并发调度的 warp 。不同于 warp 调度， Kayiran 等 [181] 则根据存储系统的冲突情况动态确定最佳的并发 CTA 数目，以获取最高的性能。相比而言， warp 调度是较为细粒度的并发控制机制，而 CTA 调度则是粗粒度的。相比这些调度机制，本文提出的方法能够在保证 cache 效率的前提下，充分利用其它片上资源（ NoC 和 DRAM 带宽），因此能够进一步提升性能和效能。

Li [14] 提出了将 cache 旁路和线程调节相结合的技术 PCAL （ Priority-based Cache Allocation ）。 PCAL 从 CCWS 的最优数目 warp-opt 的活跃 warp 出发，根据系统中的资源使用情况添加或者减少 warp 。在 cache 中分配 cache 块时，额外增加的 warp 会被赋予较低的权限，低权限 warp 发射的访存请求在 cache 块分配时权限较低，在出现冲突时会被 cache 控制器旁路，以避免 cache 冲突。可见，PCAL 的旁路策略是 warp 级别的，即某些低权限 warp 发射的所有请求都会被旁路。相比这种粗粒度旁路策略，本文立足基于重用距离的 PDP 旁路策略，重用距离本质上是对重用概率的一种预测：重用距离越大，重用概率越低。其旁路的粒度是请求（ request ）级别的，故而属于细粒度的旁路策略。细粒度的旁路策略能够更好地利用 cache 空间，因为每个 warp 发射的请求中会有重用概率高和重用概率低的请求。 PCAL 的粗粒度策略无法避免重用概率低的请求被插入到 cache 中，而细粒度策略则总是倾向于让重用概率高的请求驻留在 cache 中。由于不能充分利用 cache 资源，粗粒度方法会制约性能的提升。事实上，实验结果也显示，PCAL 相比 CCWS 并没有太明显的性能提升。此外， PCAL 以 CCWS 为出发点，不仅需要 CCWS 中引入的所有硬件支持，还需要额外增加 PCAL 中的旁路硬件，开销很大。本文提出的方法不需要 CCWS 中的 VTA （ victim tag array ）硬件，很大程度上降低的硬件开销。

3.3 功耗控制

目前许多芯片支持 DVFS，比如 Intel 处理器支持 SpeedStep， ARM 处理器支持 IEM(Intelligent Energy Manager) 和 AVS （Adaptive Voltage Scaling）等。但是要让 DVFS 发挥作用，真正降低功耗，只有芯片的支持还是不够的，还需要软件与硬件的综合设计。

在 DVFS 技术方面， Semeraro 等 [182] 提出将一个处理器划分成多个时钟区域（MCD:Multiple Clock Domain），使得每个区域可以独立地调节频率和电压来细粒度地优化性能和能耗，区域边界的交互通过核内原有的缓冲队列来实现，减少了区域同步的开销。在此基础上，不同于原先根据程序离线 profiling 的数据结果进行电压和频率的调节， Semeraro 等 [183] 随后又提出了针对同一问题的动态在线调节算法。该算法主要用区域间缓冲队列的占用情况的变化作为衡量资源利用的指标，采用 Attack/Decay 算法快速地应对程序运行的突发变化，以达到较低的性能降低并节省较多的能耗。 Kim [184] 分析了传统偏下 DVFS 电压调节器存在的不足，根据当前的技术水平，提出了片上电压调节能带来的优势，如调节器占用空间的减小，调节频率的提高，支持每个核区域独立调节电压。同时，他们也模拟并分析了高频片上调节所带来的转化能耗效率低下的问题。为未来 DVFS 系统与算法设计提供了有用的参考。 Howard [185] 在一个 48 核 IA-32 芯片上实现了单核的独立 DVFS 调控，该能耗管理系统可以提供 8 个电压和 28 个频率区间的灵活调节。

在体系结构方面， Manne 等 [186] 指出，在流水线内部，有部分指令是通过错误的分支预测而被取入的，这部分指令浪费了相当一部分能耗。针对这一问题，他们提出了门控流水线，其基本思想是根据指令是否被执行的概率（用 confidence bit 来衡量），决定执行该指令，还是通过门控，停止对应流水线阶段的运作，从而节省能耗。 Isci [187] 等人提出了动态的基于每个核单独的 DVFS 调节算法。他们根据不同的优化目的，提出了三种不同的调节策略，分别基于任务优先级，各个核的能耗平衡，和优化系统吞吐率。 Li [188] 也提出了根据程序特征的动态能耗调节算法，在程序运行过程中，通过建立搜索空间去寻找全局最优的 DVFS 控制方案，并使用启发式算法去减少空间搜索的复杂度。 Bitirgen [189] 进一步提出了用机器学习的方法协调片上共享资源（包括功耗预算），以达到提升全局性能的效果。他通过学习程序在不同资源配置下的各阶段的性能变化，根据这些学习结果，在多程序并发的情况下，建立算法协调片上资源的分配，快速检测最优配置方案。

在软件支持方面， Magklis 等 [190] 在多时钟区域（ MCD ）方案的基础上，提出了根据程序 profilling 的情况，在程序中自动插入电压调节控制指令，重写二进制执行代码，有效的实现了核内 DVFS 调节。 Hsu 等 [191] 提出了运用 DVS （ dynamic voltage scaling ）来节约程序能耗的编译算法。他们根据程序离线 profiling 的结果来分析程序的不同区域对于电压和频率的需求，运用数学模型计算最优的电压配置，然后在编译时向代码中插入控制电压的指令。 Xie [192] 建立了详细的数学模型来分析，编译期间的 DVS 设置能否带来能耗的节省，以及节省的程度。他们也提出了相应的编译算法，在算法中着重考虑了电压调整的开销和不同数据集输入带来的影响。 Wu [193] 指出了静态编译算法和纯硬件或操作系统支持的 DVFS 调控所存在的不足，并提出了相应的动态编译框架。他们把程序划分为较长的区域，在每段区域开始执行的期间进行测试和硬件反馈信息的收集，这些信息作为 DVFS 决策算法的输入，来决定该区域余下代码的电压频率设置，动态编译负责在相应的代码位置插入电压控制指令。

在应用方面， Meisner 等 [194] 提出了基于大型服务器（ server ）能耗控制方法，他们指出了传统的 DVFS 检测和调节方法在优化服务器程序能耗方面的不足，并提出了 PowerNap ，来充分减少服务器空闲时的能耗。其主要思想是，将服务器工作分成两个模式，全速（ full performance ）模式和空闲（ idle ）模式，当服务器有需要完成的任务时高速运行，没有任务（ idle ）时，进入休眠模式。而如何快速唤醒服务器则得益于网卡（ NIC ）中新增的检测单元，每次检测到有新任务到达，服务器被唤醒。

针对 GPU 的功耗优化， Leng 等 [94] 提出了时钟精确的 GPGPU 功耗模型，并与真实的硬件做了对比验证。该模型被集成到 GPGPU-Sim 中，支持 DVFS 和门控时钟这些功耗优化方法。 Hong 等 [195] 提出了针对 GPU 的功耗和性能预测模型。该模型能够根据 kernel 和体系结构的特征有效地预测应用在 GPU 上的执行时间和功耗。 Isci 等 [196] 利用 CPU 中访存和计算的阶段性行为特征进行功耗优化。这种动态优化策略对于阶段性特征明显的应用程序非常重要。 Lee 等 [197] 在 GPU 中应用 DVFS 和核扩展两种功耗控制技术进行了实验。他们的工作试图在给定的功耗约束下最大化吞吐率。 Mohammad [198] 指出由于 GPGPU 运行的空闲时间较短，以及调度器实现的贪婪调度算法，使得传统 DVFS 无法在短时间内有效的调节电压频率，据此，他们提出了门控感知的 warp 调度算法（ GATES： Gating Aware Two-Level Warp Scheduler ），每次倾向于发射同种类型的指令，直到不存在可发射的该种指令，才切换到下一种指令类型，这种方法创造了较长的空闲时间，使得 DVFS 可以发挥其作用。

3.4 其它缓存管理策略

cache 划分 [199–201] 技术针对多道程序（multi-programmed）的情况，对共享 cache 进行划分，以提升性能、公平性（fairness）、QoS（quality-of-service）等等。cache 划分的好处是避免程序之间竞争使用 cache 而导致的性能降级，同时也避免了 cache 污染型的应用独占共享 cache 而导致 cache 友好型的应用无法从共享 cache 中受益。TAP [202] 提出了针对 CPU-GPU 共享 LLC 的 cache 划分策略，根据 CPU 负载和 GPU 负载的访存行为动态地划分 cache 容量，能够在不明显损害 GPU 吞吐率的情况下提升 CPU 负载的性能。相比多道程序，本文考虑的是大规模多线程的场景，平均每个线程的 cache 容量非常小，因而没有考虑对线程、warp 或 CTA 进行共享 cache 划分。Gaur 等 [203] 提出了针对 GPU 上 3D 场景渲染负载 LLC 管理策略，而本文主要关注通用计算负载。其它工作研究了异构系统中的 cache 管理策略 [204]. 虽然本文以 GPU 为具体研究对象，但提出的方法对于融合的 CPU-GPU 系统（比如 APU）通用适用。对于 APU 而言，文中提出的防冲突和防拥塞技术其实更为重要，因为 APU 中的 cache 容量通常比离散 GPU 中的小很多。

cache 包含性 [205, 206] 对 cache 的性能和实现复杂度有着重要的影响。包含 cache 和不包含 cache 两种设计有着各自不同的优缺点。包含 cache 实现复杂度相对较低，能够简化一致性协议，但是各级 cache 中存在多个数据副本，cache 容量没有得到最大化的利用，而且在外层 cache 数据块被替换时，需要向内层 cache 发送通知，将其中的副本也剔除出 cache，即所谓（recall）。采用包含 Cache 的另一个优点是可以将在内层 cache 中的未经改写的数据块直接剔除（Silent Eviction）。不包含 cache 则最大可能地利用了片上 cache 的容量，但是一致性协议实现的开销更大。FLEXclusion [206] 提出了一种兼顾两种设计优点的包含性策略，即通过额外的控制电路在两种策略中动态选择，以适应应用程序的不同需求。预取 [207–209] 技术同样被应用到 GPU 中以高效利用片外带宽，减少访存延迟从而提高 GPU 占用率。本文不考虑预取机制，但能够提高 cache 效率、减少片外带宽需求。预取技术能够同本文中的 cache 管理机制协同工作，进一步提高存储子系统的效率。

GPU 中 cache 块的粒度对空间局部性和带宽使用效率有很大影响。粗粒度 cache 块有利于捕捉空间局部性，但对于局部性较差的应用会浪费 DRAM 带宽。细粒度 cache 块对于规则的访存模式会丢失部分局部性，但是带宽使用效率比粗粒度方式要高。针对这个问题，Rhu 等 [210] 提出了 LAMAR，实质上是自适应 cache 块粒度。LAMAR 使用 Bloom Fliter 基于历史访问信息预测 cache 块的空间

局部性，并据此动态地选择使用粗粒度和细粒度方式取 cache 块。 Gebhart [211] 等提出了统一的 GPU 片上存储设计，以适应不同应用对各种类型的存储资源的不同需求。为了预测 GPU 的性能和功耗， Baghsorkhi 和 Kim 等 [83, 212, 213] 提出了相应的 GPU 性能、功耗以及 cache 解析模型，这对于人们理解 GPU 体系结构及其应用程序的行为非常重要，也是为开发出高性能和效能的应用程序提供参考。

研究人员也提出了许多 GPU 软件和编译技术 [135, 214–224] 来优化访存模式，特别是便笺存储器和 cache 访问模式，从而提升 GPU 的性能。虽然静态编译技术能够有效应对规则的应用程序，但是现实中存在更多的是不规则的应用程序，他们的访存模式甚至可能同输入数据相关，而且在运行时呈现出阶段性变化的特征。本文提供的是一中硬件的动态解决方案，能够适应不同的应用程序运行时的访存行为，这意味着该方法能够针对不同的输入数据和阶段性行为进行自适应调整，从而进一步提升性能。

第四章　自适应替换和旁路策略

大规模并行访存模式会引发缓存冲突，导致 GPU 中缓存低效和系统性能降级。这个问题在 CPU 中并不常见，因为通常情况下 CPU 不需要应对成百上千的并发线程。因此 CPU 的缓存管理策略通常不考虑这种严重的缓存冲突和资源拥塞问题，这使得直接将 CPU 缓存管理策略应用于 GPU 效果并不理想。为了解决这种访问模式与管理策略之间的不匹配，本文考虑针对 GPU 的访存特性定制缓存管理策略，本文涉及到的缓存管理策略包括替换策略、旁路策略、线程调节和功耗控制几个方面。其中，替换策略和旁路策略是最常研究的缓存管理策略，它们都在缓存控制器中实现。而线程调节和功耗控制技术则并不是传统意义上的缓存管理策略（即不在缓存控制器中实现），但是运用这两种技术能够间接地管理缓存，使之得到合理有效地利用，从而提升缓存的效率，因此可以说是广义上的缓存管理策略。

本章讨论 GPU 中的缓存替换和旁路策略，也就是通过修改缓存控制器的控制算法，达到提高缓存使用效率的目的。第3.1章提到，通常来说，替换和旁路策略主要用于解决缓存中常见的两个问题：缓存冲突和缓存污染。直接将 CPU 缓存中的防冲突和防污染算法用于 GPU 的效果并不理想，因为这些算法没有考虑 GPU 中大规模并行对存储系统带来的影响。本章首先**对 GPU 中缓存的属性和行为进行分析**，着重突出其不同于 CPU 缓存的模式和特征，然后**在 GPU 缓存中实现目前最为先进的 CPU 缓存管理策略**，通过实验数据分析指出 CPU 缓存管理策略应用于 GPU 中的局限性，最后根据这些局限性，**提出针对 GPU 访存模式定制的管理策略**，进一步改进系统的性能。

4.1　缓存属性和行为分析

表4.1中列出了各代 NVIDIA GPU 中各级存储器的容量。可以看到，寄存器文件、便笺存储器和 L2 cache 都保持稳定增长，这三者对 GPU 性能的重要性不言而喻。 L1D cache 的发展则经历了一个从无到有、从有到无的过程。 L1D cache 从 Fermi 架构开始被引入到 GPU 中，并且同便笺存储器（ SMEM ）紧密联系在一起，即二者共享 64KB 的 SRAM 存储阵列，但是程序员可以显示地选择 48KB 的 SMEM 和 16KB 的 L1D cache ，或者 16KB 的 SMEM 和 48KB 的 L1D cache 。这种设计给了程序员进行存储优化的灵活性。合理利用 L1D cache 也确实能够为很多应用 [135] 带来性能提升。但是在 Maxwell 架构中， SMEM 被独立了出来，而 L1D cache 则同 texture cache 合并到了一起，变为只读（ read-only ） cache 。事实上， NVIDIA 对于这种设计的倾向在 Kepler GPU 中就已经显现了出来 [74]

Mem Unit	Tesla GT200	Fermi GF106	Kepler GK104	Maxwell GM107
REG	16384	32768	65536	65536
L1D $	×	16/48K	16/48K	×
SMEM	16K	48/16K	48/16K	64K
L2 $	×	768K	1.5M	2M
DRAM	141.7 GB/s	177.4 GB/s	192.2 GB/s	86.4 GB/s

表 4.1　各代 NVIDIA GPU 中各级存储层次的容量和 DRAM 带宽

。 Kepler 中 L1 cache 仅用于本地存储器访问（比如寄存器溢出和栈数据），全局 load 只被缓存在 L2 cache 中。只有 GK110B 系列的 GPU 可以通过设置 -Xptxas -dlcm=ca 编译选项来选择将全局数据同时放入 L1 和 L2 中。这种设计趋势具体是出于哪些权衡考量目前并不明确，但可以想象 L1D cache 的低效显然是其原因之一。

表4.1中还能看出， DRAM 带宽随着每一代架构的更新保持着增长，但是非常有限。带宽的扩展明显滞后于 SIMT 核计算能力的扩展，严重制约了系统的吞吐率。需要说明的是， Maxwell 架构的 GPU 目前只推出了 GeForce GTX 750 和 GeForce GTX 750 Ti ，因为不属于高端系列的产品，其 DRAM 带宽较低。后续的高端产品带宽可能达到 300GB/s 左右。即便如此，这也并不足以满足各种计算负载对带宽的需求。 NVIDIA 在 2014 年 GTC 上已经宣布下一代 NVIDIA GPU 架构 Pascal 将采用 3D Memory 技术，其主存带宽将达到 1TB/s 。如果真的能够实现，将会很大程度上缓解计算和存储的矛盾。而另一方面，对于通用计算负载，各级存储器的访存延迟参数将变得越来越重要。

表4.2中列出了 NVIDIA GPU 中各级存储器的命中延迟。总的来看， GPU 中 cache 的访问延迟相比 CPU cache 是很大的。事实上， Intel Haswell 的三级缓存命中延迟也才 36 个周期。 Fermi 架构开始引入两级 cache 层次结构，这使得全局数据的最短命中延迟从 440 个周期缩短到 45 个周期，但同时也导致其各级存储层次的访问延迟明显增大（因为访存请求需要依次穿越各级存储层次），尤其是 L2 cache 和 DRAM 的延迟。在 L1D cache 被逐渐削弱之后， L2 cache 和 DRAM 的延迟开始明显缩小。事实上， L1D cache 的容量相对于 SIMT 核上的线程束数目而言非常有限，大规模并发线程会生成大量的访存请求，导致 cache 资源拥塞，引发访存暂停（ memory stall ）。访存请求无法分配到资源，只能等待。这种情况在 GPU 中频繁发生，结果就是 L1D cache 不但没有能够帮助提升性能，反而严重阻滞了访存请求的发送，造成性能大幅降级。这就是从 Kepler 开始全局数据只缓存在 L2 cache 中的原因之一。第 5章会详细讨论这个问题。

Mem Unit	Tesla GT200	Fermi GF106	Kepler GK104	Maxwell GM107
L1D $	×	45	30	×
SMEM	40	50	35	30
L2 $	×	310	175	195
DRAM	450	680	300	350

表 4.2　各代 NVIDIA GPU 中各级存储层次的延迟参数（单位：时钟周期）[225]

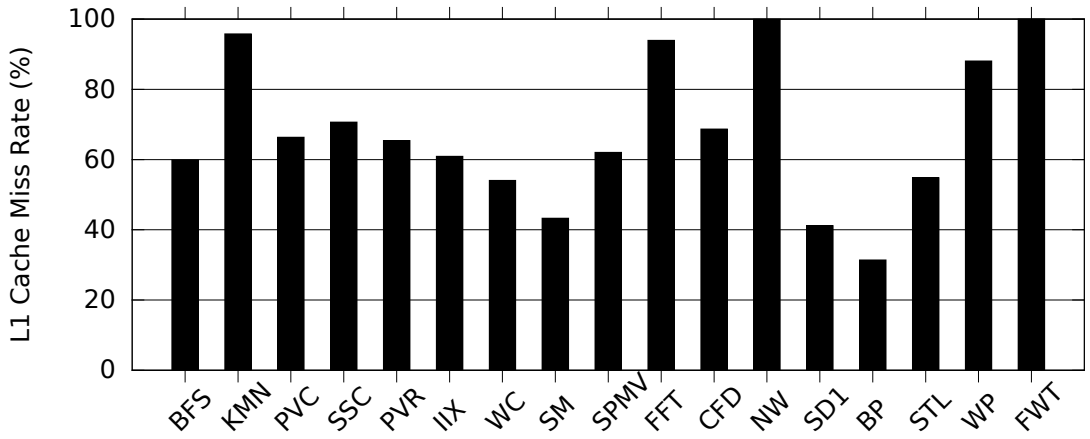

图 4.1　所有负载在基准架构下的 L1 cache 失效率

图 4.1和图 4.2分别显示了在基准架构下所有负载的 L1 cache 和 L2 cache 失效率。不难看出，相比 CPU 中的 cache ， GPU 中的 cache 失效率明显高很多。这是因为一方面， GPU 中采用 SIMD 的方式执行每个 warp ，程序中原有的大多数空间局部性被硬件合并单元捕获。这种情况对于规则的应用程序尤为常见。因此，很多适合 GPU 加速的应用程序都具有流模式的数据访问特性，这样的数据访问模式不具有局部性特征，因此 cache 的效率很低。而另一方面，由于 GPU 采用大规模并行执行模式，数据工作集可以达到几十上百 MB ，这就会导致 cache 出现剧烈的冲突，难以有效捕捉时间局部性。总的来看，传统 CPU 中的 cache 系统与 GPU 的执行模式不相适应，导致了 cache 的效率低下。

图 4.3中显示了基准架构下访存请求的平均往返延迟。可以看出，大部分负载的访存延迟都在 200 到 350 个周期之间，这意味着平均来看，访存请求的延迟大概是访问 L2 cache 的延迟。 L2 cache 能够减少 DRAM 流量，保证了大部分的请求在片上得到服务。少数缓存不敏感（ CI ）负载的访存延迟较大，主要是由于访存频度较大，且局部性特征较差导致的。图 4.4中显示了基准架构下访存流水线（ Memory Pipeline ），即读写单元（ LD/ST Unit ），暂停的比率。每个 SIMT 核有四种不同的一级存储部件：便笺存储器（ shared memory ）、数据缓存

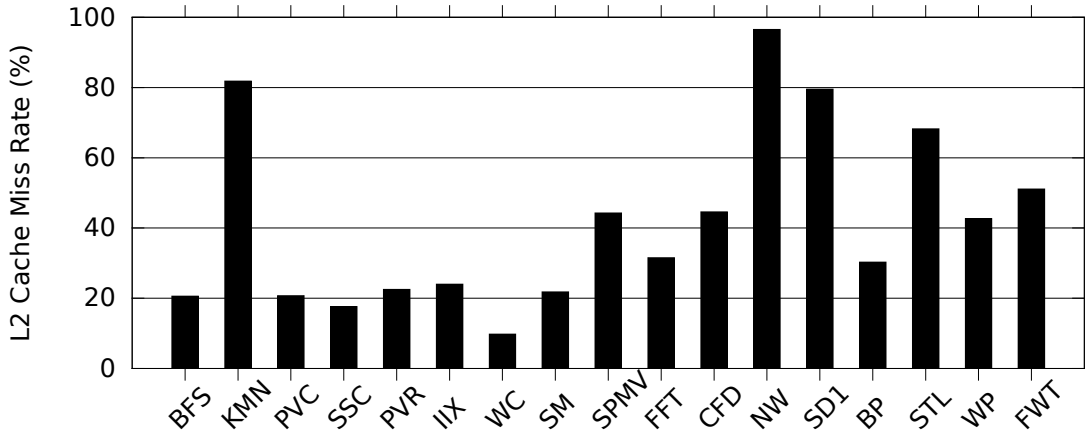

图 4.2　所有负载在基准架构下的 L2 cache 失效率

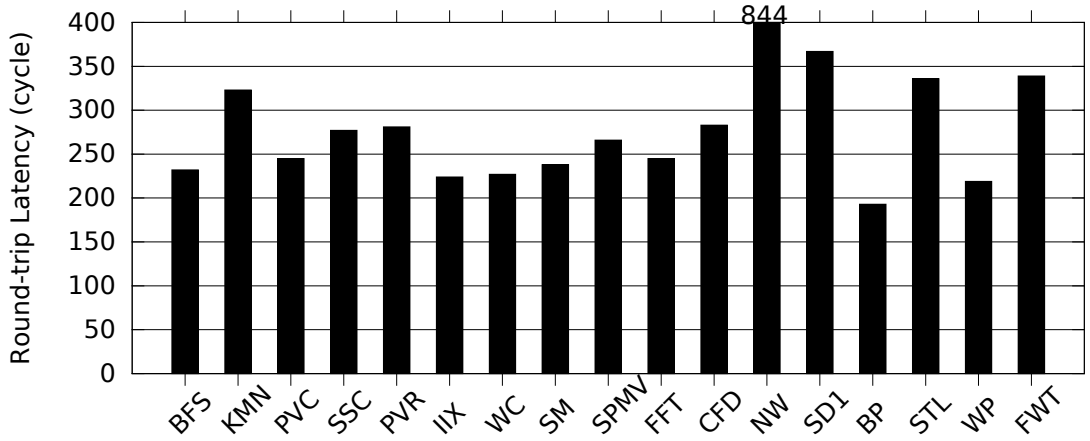

图 4.3　所有负载在基准架构下访存请求的平均往返延迟

（data cache）、常量缓存（constant cache）和纹理缓存（texture cache）。在每个时钟周期读写单元，出现这样几种情况会导致流水线暂停：便笺存储器 bank 冲突、访存请求无法合并（non-coalesced memory access）或者对常量缓存的串行化（serialized）访问。访存流水线暂停的比例说明了存储子系统的工作效率。图中可以看出，很多负载的流水线暂停比例都高于 50%，这说明这些程序执行时，GPU 的存储子系统的效率非常低。

图 4.5显示了所有负载在基准架构下内存分区（memory partition）拥塞的比率。这个比率是指片上网络输出数据到 DRAM 通道时发生暂停的周期数在总周期数中所占的比率。图中可以看出，大多数负载的暂停比率在 10% 以上，部分负载的暂停比率超过了 20% 以上，比如 SSC 和 PVC 等。造成内存分区拥塞的原因可能是资源（L2 cache 块或者 MSHR）不足，或者内存控制器（memory controller）的请求缓冲已经满了，亦或者片外存储带宽受限。图 4.6显示了所有负载在基

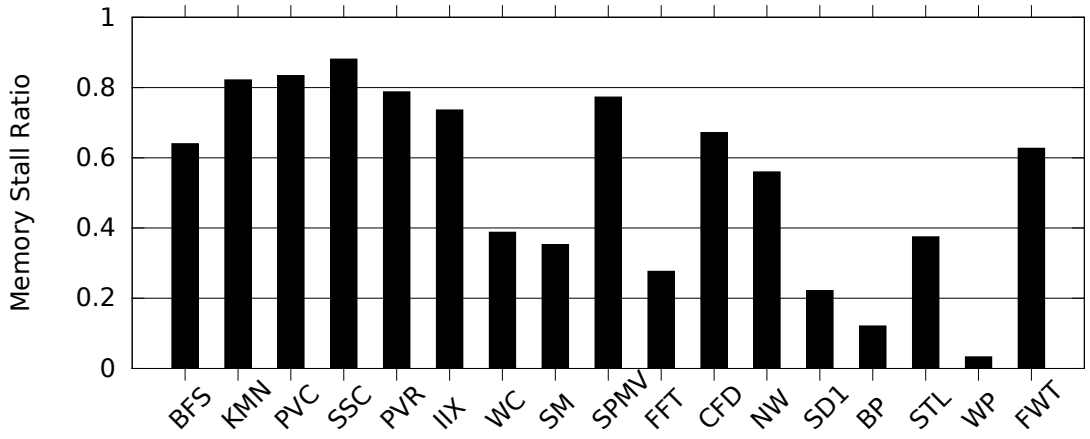

图 4.4　所有负载在基准架构下访存流水线暂停的比率

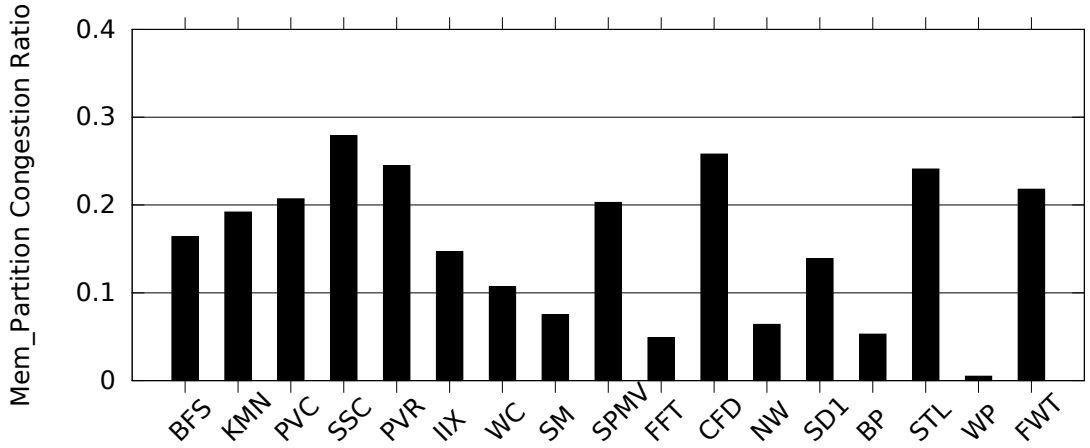

图 4.5　所有负载在基准架构下内存分区拥塞的比率

准架构下的片外存储带宽利用率。图中可以看出，大多数应用的片外带宽利用率并不高，这同这些程序的不规则访存模式有很大的关系。

需要注意的是，带宽利用率较低并不代表片外存储带宽就不是程序的性能瓶颈，因为 GPU 应用可能存在突发访存（burst）的模式，即在一个很短的时间内，SIMT 核向存储子系统发出大量的数据请求，导致带宽受限。但这种访存的情况在整个程序的执行过程中可能只占一小部分，因此总的来看带宽利用率还是会较低。图 4.7中则采用 DRAM 效率对带宽的使用情况进行了统计。DRAM 效率只统计在有请求在等待时的带宽利用率，因而对于具有突发访存模式的应用，DRAM 效率不会给出误导性的数据。图中可以看出，绝大多数负载的 DRAM 效率都在 40% 以上，这同这些应用的访存受限特性是一致的。即便如此，这些负载的带宽利用率远远达不到 100%，当然其主要的原因是 DRAM bank 冲突，因而无法有效挖掘存储级并行（memory level parallelism）。这同它们不规则的访存模式有很大的关系。

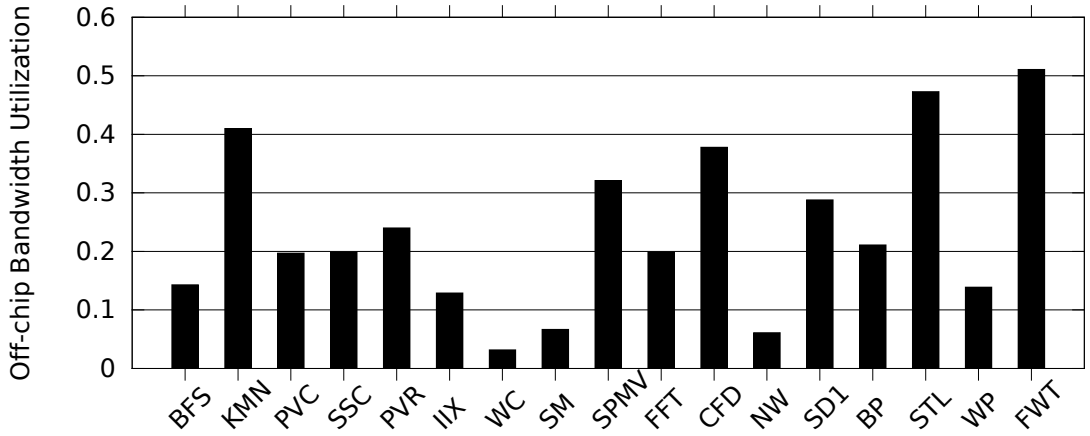

图 4.6　所有负载在基准架构下的片外存储带宽利用率

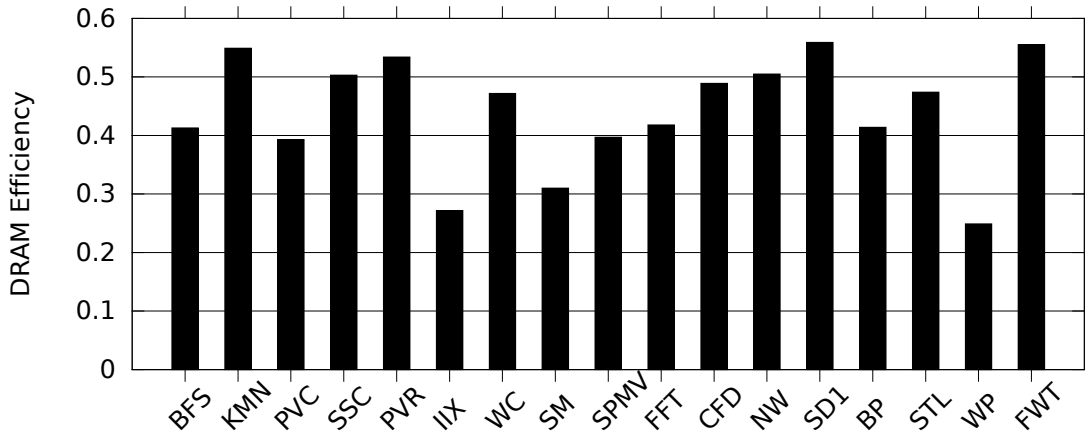

图 4.7　所有负载在基准架构下的 DRAM 效率

图 4.8中显示了采用 GTO 调度策略后，相对 LRR 调度策略的性能变化情况。不难看出，绝大多数程序在 GTO 调度策略下获得了性能提升。特别是 HCS 负载，性能收益很明显。这是因为 GTO 调度策略在一定程度上维护了 warp 内的时间局部性，因而有更好的 cache 性能。图 4.9说明了这一推断。可以看到，GTO 策略下有很多负载的 L1 cache 失效率明显下降了。由于 L1 cache 的访问延迟很短，在 L1 cache 中的命中率提升直接给这些 cache 敏感的应用程序带来了性能提升。与此同时，对于部分程序，由于 L1 cache 性能的提升，发送到 L2 cache 的请求数目锐减，因此也间接地减少了 L2 cache 的冲突，从而使得 L2 cache 的失效率也获得了明显的改善，如图 4.10所示。

图 4.11和图 4.12通过逐步增大 L2 cache 的容量，对这些基准测试程序的工作集进行了估计。在增大 L2 cache 容量的情况下，L2 cache 的失效率会降低，这种降低的变化趋势通常并不是线性的，而是会在增大到某个结点时出现突然的急剧下降，这种变化的原因是，在某个容量下，刚好能够让程序中的某个重要的数据

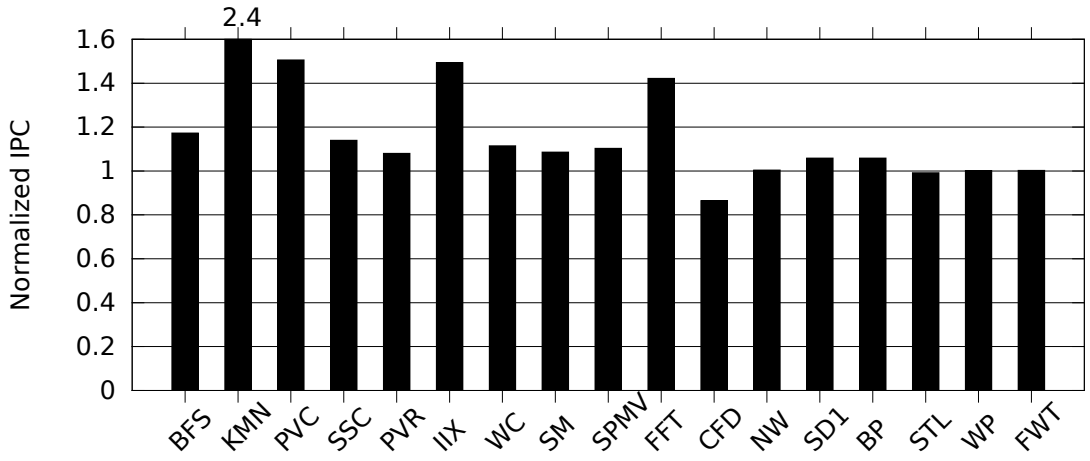

图 4.8　GTO 调度策略下相比基准架构的性能

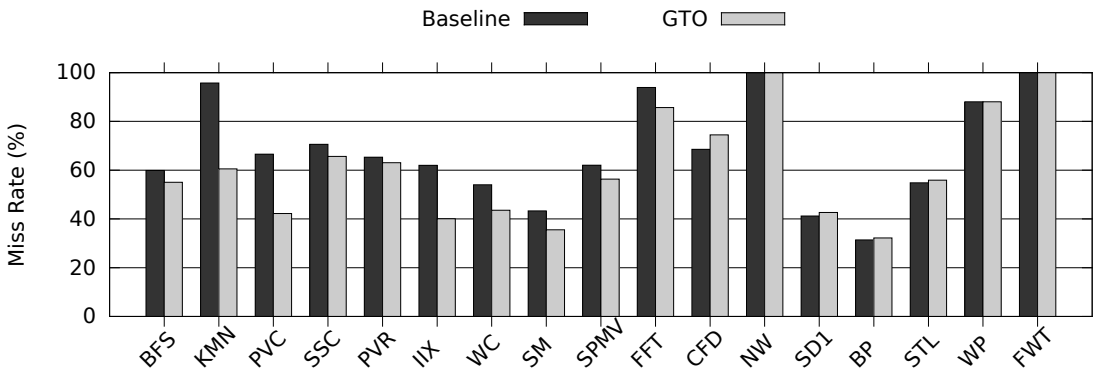

图 4.9　GTO 调度策略下相比基准架构的 L1 cache 失效率

结构驻留在 L2 cache 中，因而能够非常明显地降低 L2 cache 失效率。因此，对于某个给定的应用程序，在其 L2 失效率出现拐点的时候，就是其重要的工作集大小。图中红色箭头所标注的正是各个程序的重要工作集。不难看出，GPU 应用程序的工作集大小通常都在 2MB 以上。个别负载的工作集能够达到几十甚至上百 MB 。但需要说明的是，并不是增大 L2 cache 的容量就能提升 GPU 应用的性能。因为 GPU 的 SIMT 执行模式通过大规模多线程来隐藏访存延迟，因此很多应用程序的性能对 L2 cache 的性能并不敏感。这也是目前商用 GPU 系统中 L2 cache 容量相对较小的原因之一。

4.2　缓存冲突特征化

图 4.13显示了增加 L1D cache 容量的情况下 HCS 测试程序相对基准架构（32KB L1D ）的加速比。可以看出这些 HCS 程序的性能随着 cache 容量的变化而剧烈变化，因为增加 L1D 缓存容量缓解了 cache 冲突，从而提升了性能。图 4.14说明了在 cache 容量增大的情况下，HCS 负载的 cache 失效率明显下降。

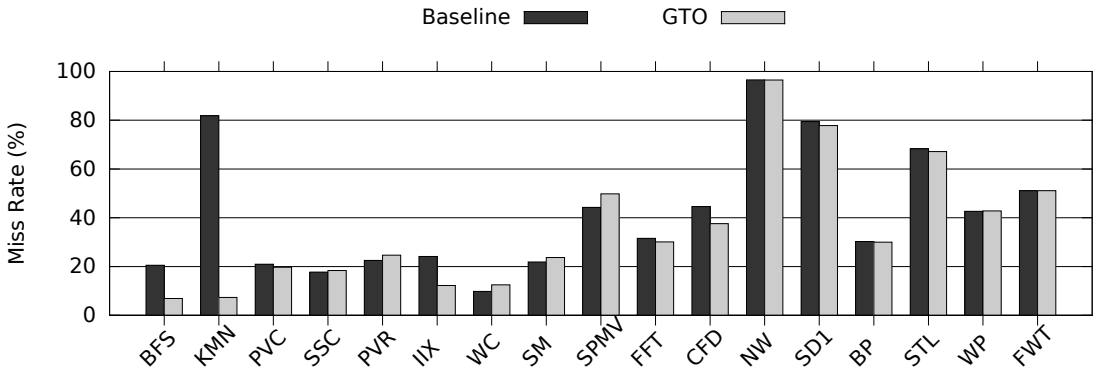

图 4.10　GTO 调度策略下相比基准架构的 L2 cache 失效率

对 MCS 和 CI 的程序我们也进行了类似的实验，如图 4.15和图 4.16所示， MCS 程序显示出了一定程度但不如 HCS 那么剧烈的性能变化，而 CI 类型的负载则显示出对 cache 容量变化的不敏感特征。这个实验本身也是对表 2.2中程序按照其 cache 敏感性进行分类的依据。对于 HCS 负载而言， cache 的性能对系统性能有很大的影响。当 cache 容量变大时， HCS 负载的性能明显上升，这意味着在这类程序中存在大量的数据局部性，特别是时间局部性。在 cache 容量有限的情况下，如何尽可能多地挖掘 HCS 负载中的局部性，这涉及到 cache 管理策略的问题。本文就是试图针对 GPGPU 的特性改进其 cache 管理策略，在给定 cache 容量的情况下提高 cache 的效率。

图1.1中提到 L1D cache 中零重用的缓存数据块占很大比例。造成大量零重用缓存数据块的原因可能是程序访存呈现流（ streaming ）访问模式或者存在重用可能的数据块被过早移除（ early eviction ）。流访问模式是指每个数据都只访问一次，因此不存在数据重用的可能。过早移除则是指 cache 中的数据没有得到充分的重用就因为 cache 冲突而被移除出 cache 。为了说明 cache 冲突在 cache 敏感型应用中的严重程度，图 4.17显示了 HCS 测试程序的 L1D cache 冲突率（ CR ：contention ratio ）。冲突率指的是因过早移除导致的对数据的重复失效请求在所有失效请求中所占的比率。图 4.17的 L1D cache 冲突率是由 L2 cache 检测的，即 L2 cache 累计 L1D 发往 L2 的请求中重复请求的数目（ R_{repeat} ），然后除以 L1D 发往 L2 的所有请求的数目 (R_{all}) ， $CR = \frac{R_{repeat}}{R_{all}}$ 。从图中可以看出过早移除在 cache 敏感型程序中发生得非常频繁。增大 cache 容量能够缓解冲突，这也是 cache 敏感型应用受益于大容量 cache 的原因。值得注意的是，由于冲突率是由 L2 cache 检测的，而 L2 cache 本身的容量也是有限的，因此当 L2 cache 中也存在严重冲突时，这个检测方法会不太准确。比如说， KMN 就产生了大量的请求导致 L1D 和 L2 cache 都出现了严重的冲突问题。而图中显示其 L1D 的冲突率并不是很高，这正是因为 L2 缓存的数据块被频繁的替换而无法准确记录 L1D 的冲突请求。增大

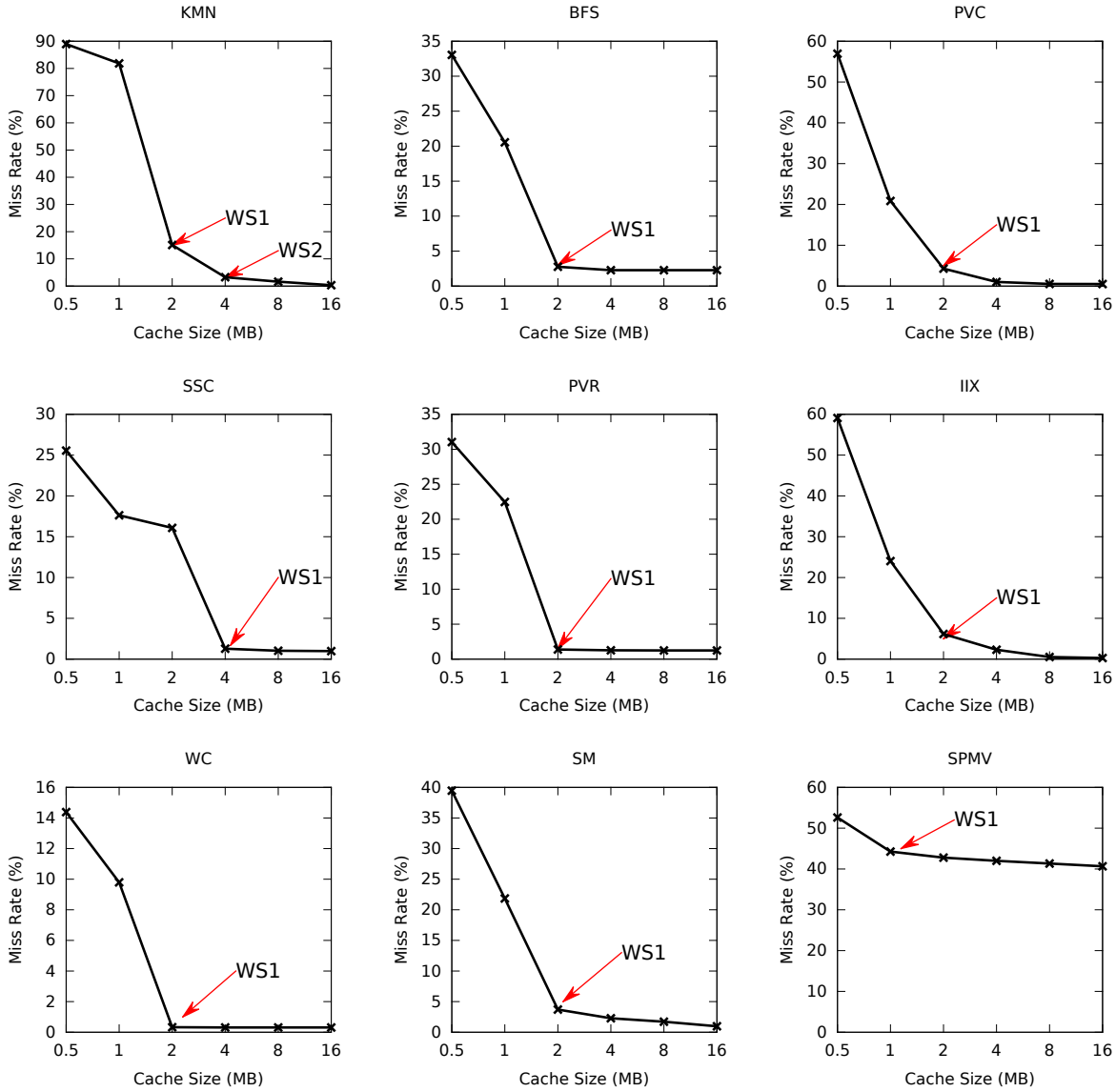

图 4.11　不同 L2 cache 容量下的失效率 (a) 。实验设定 L2 cache 是块大小为 128 字节的 16 路组相联 cache 。 WS1 和 WS2 分别指示重要的工作集。本文选取的 GPU 基准测试程序对 LLC 容量的需求通常在 4MB 以内。部分负载需要 16MB 甚至更大的 LLC 。

cache 容量固然能够提高 cache 效率，但是同时也会增加 cache 的访问延迟、面积和功耗。本文则立足于有限的 cache 容量，通过定制管理策略来尽可能地充分利用 cache 和其它片上资源，从而提高系统性能和效能。

4.3　CPU 缓存管理策略的局限性

上一节可以看出， GPU 的大规模并行执行模式给存储子系统带来了很大的压力，这严重加剧了传统的缓存冲突和缓存污染问题。大规模多线程的访存需求及其线程的相互交织和频繁切换使得缓存管理的难度加大，直接将 CPU 中的管理

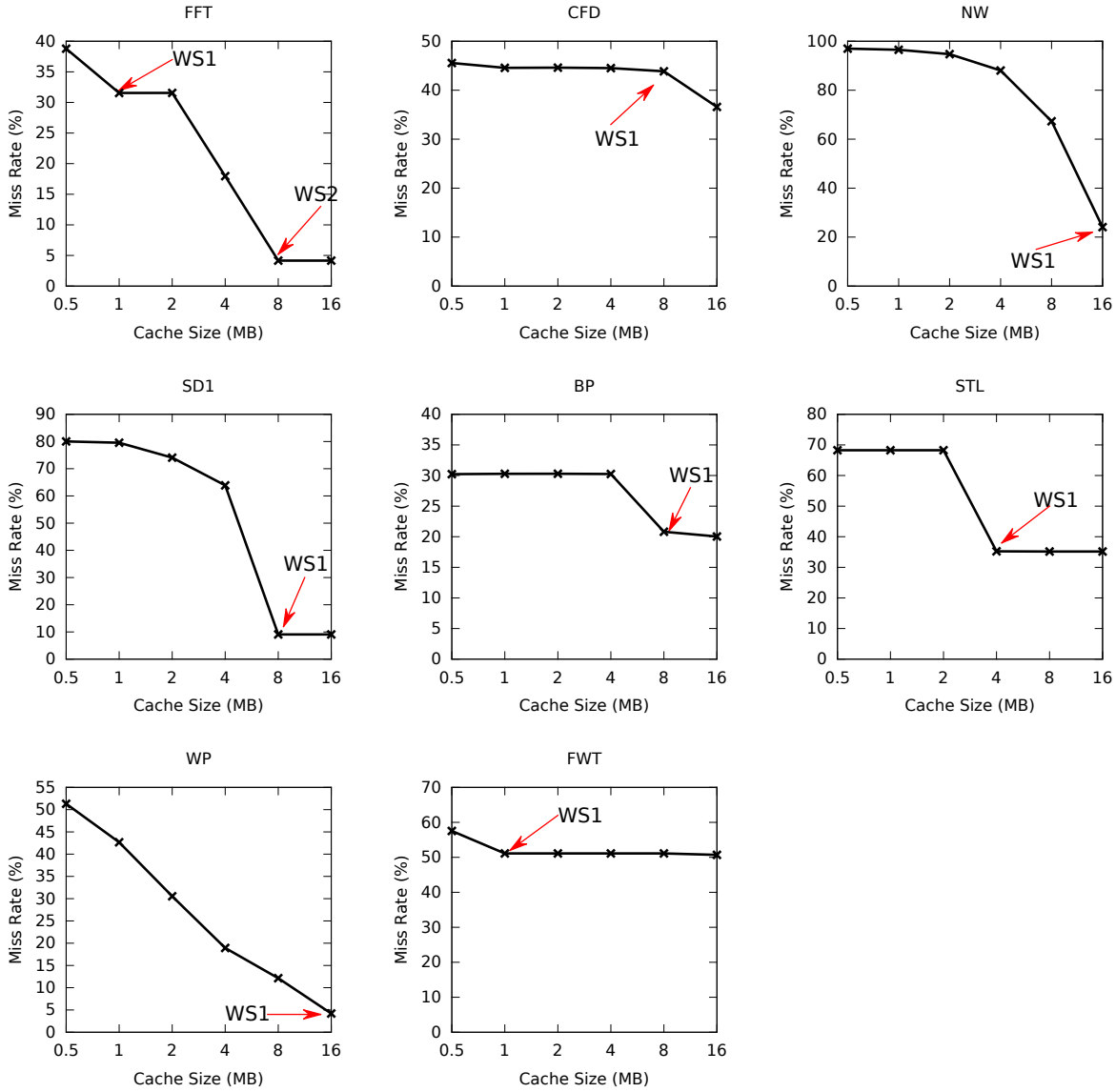

图 4.12　不同 L2 cache 容量下的失效率 (b)

策略应用于 GPU 存在诸多局限性，导致效果并不理想。本章在 GPU 缓存中实现目前最先进的 CPU 缓存管理策略，通过实验数据分析，一方面说明先进策略存在很大的性能潜力，有利于 GPU 系统的性能提升，另一方面这些策略由于没有考虑 GPU 的特征存在明显的局限性，有待进一步的改进。

4.3.1　缓存替换策略

前面提到， cache 替换策略决定了访存请求对有限的 cache 空间占用的优先秩序，因此对性能影响很大。目前 CPU 系统中最常用的 cache 替换策略是 LRU 的近似算法 NRU 或者 Pseudo LRU ，而最为先进的 cache 替换策略是 RRIP [12] 和

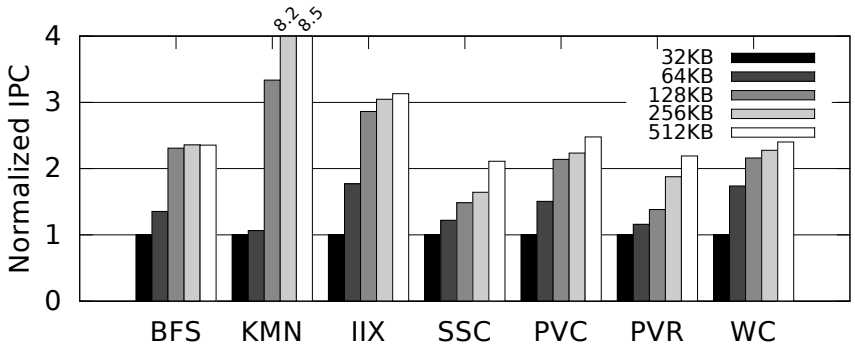

图 4.13　L1D cache 容量大小对 HCS 负载的性能影响，归一化到 32KB L1D cache

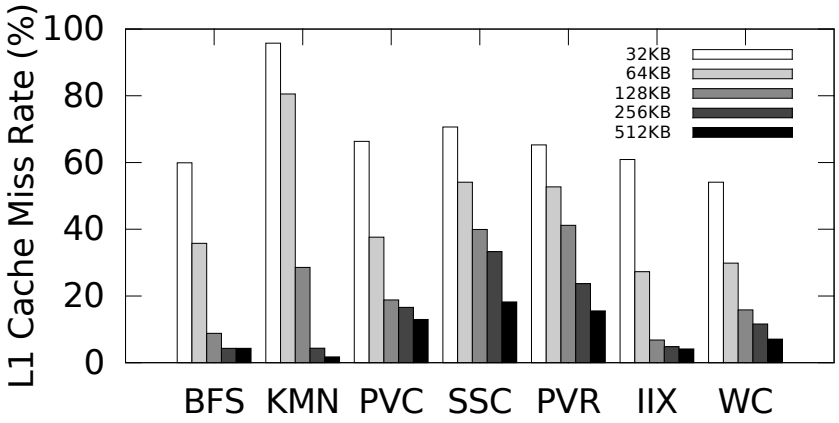

图 4.14　不同 L1D cache 容量下 HCS 负载的 cache 失效率

PDP [13] 。真实系统中采用 NRU 算法是为了降低硬件开销且保持控制逻辑的简单性。可见，硬件开销和逻辑复杂度方面的考量是 cache 管理策略设计中最重要的权衡指标。相比而言，复杂的替换策略能够以额外的硬件开销换取更好的性能，但多数的情况下，有限的性能收益不足以驱使芯片厂商真正将这些策略实现到处理器中。 RRIP 策略保持了相对简单的控制逻辑，而 2-bit RRIP 的硬件开销则是 NRU 的两倍。 PDP 策略则是在 RRIP 的基础上进一步一般化，相比 RRIP ， PDP 增加了采样部件来动态估算重用距离。在 CPU 中，针对 cache 冲突和污染问题，RRIP 和 PDP 策略都能够带来一定的性能提升。

图 4.18显示了 GPU 中采用 SRRIP 替换算法是 HCS 测试程序的性能。可以看出， SRRIP 替换策略能够一定程度上提升 HCS 负载的性能，其中 IIX 和 PVC 获得了 13% 和 7% 的性能提升。对于 MCS 和 CI 负载的性能， SRRIP 策略则基本没有影响。但是 SD1 和 CFD 在 SRRIP-8 策略下出现了明显的性能降级，这是因为 SRRIP-8 对于重用过的缓存数据块的保护时间过长，使得其它有用的数据无法有效利用 cache 空间。总的来看，替换策略能够带来的性能提升非常有限。事实上

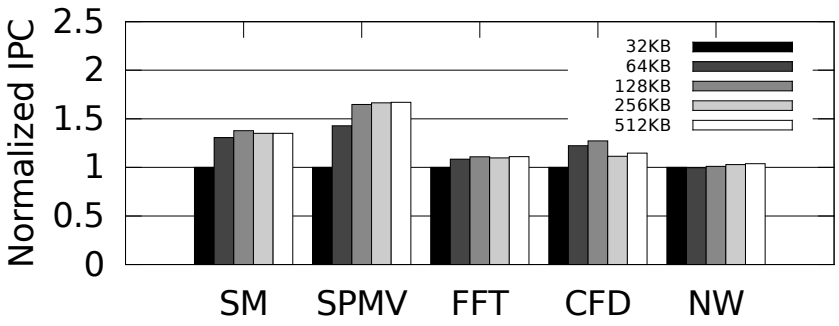

图 4.15　L1D cache 容量大小对 MCS 负载的性能影响，归一化到 32KB L1D cache

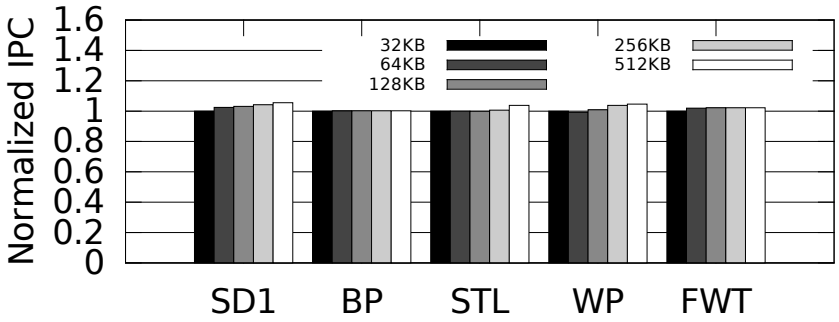

图 4.16　L1D cache 容量大小对 CI 负载的性能影响，归一化到 32KB L1D cache

，即使采用最优的 Belady 算法 [155]，其性能提升也并不明显 [17]。对于 cache 敏感型的应用，其在 GPU 中的有效 cache 容量（即平均每个线程的 cache 容量）非常低，这使得 cache 冲突频繁发生，成为制约性能的主要原因。先进的替换策略能够选择未来被重用概率较低的缓存数据块作为 victim，但是无法避免缓存数据块被过早移除。要想避免过早移除的问题，就需要对缓存数据块进行保护，即确保其不被替换，直到其被重用。当目标组中的所有缓存数据块都处于被保护状态时，传入请求则无法在 cache 中分配空间，因而需要其它的数据通路来保证该请求得到服务而不致使 cache 进入暂停（stall），这个数据通路就是 cache 旁路。

4.3.2　缓存旁路策略

针对 cache 污染（streaming）和 cache 冲突（thrashing）这两个问题，本节探讨相应的缓存旁路策略来缓解污染和冲突。首先我们考虑 cache 污染的问题。前面提到，典型的 GPU 应用中流访问非常频繁，流访问对其数据只访问一次，因而这种访问模式不存在 cache 局部性特征。流数据占用 cache 空间却没有受益于 cache，对于其它数据是一种 cache 污染。本节提出动态预测流访问模式，在分配 cache 空间时，被预测为流访问的访存请求被分配较低优先级，以腾出 cache

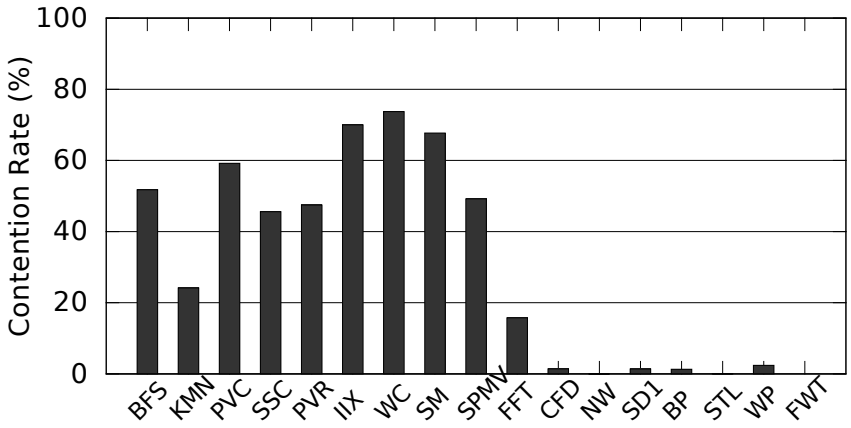

图 4.17　由 L2 cache 检测到的 L1D cache （32KB）冲突率。由同一个 L1D 发出的重复的访存请求被 L2 记录为 victim 请求。冲突率由 L2 中检测到的所有 victim 请求的数目除以所有请求的数目而得出

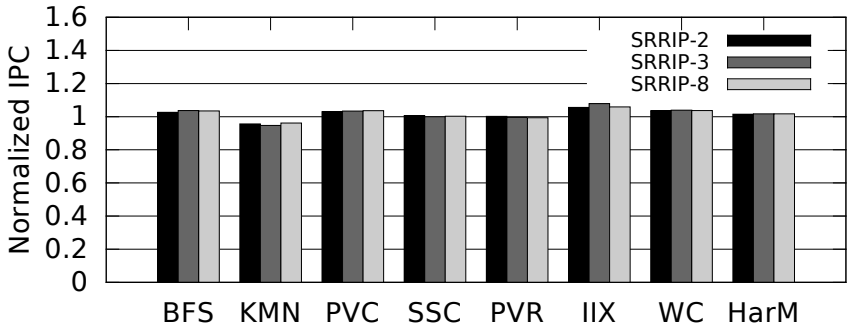

图 4.18　各种替换策略的性能，归一化到基准架构

空间给其它重用可能性较高的数据。在数据局部性频繁流失的情况下，流访问请求将会被直接旁路，避免其污染 cache 。流访问模式的预测机制类似于 SHiP [162]（Signature-based Hit Predictor）中基于签名（Signature）的预测机制。我们采用访存指令的 PC 作为签名，将访存请求的签名存放于 hash 表中， cache 命中和失效将会相应的更新 hash 表中的值。对于一个传入请求，控制器会查找 hash 表中的值同相应的阈值作比较，从而来预测该请求是否为流访问请求。由于被应用到 GPU 中，我们将采用这种预测机制的 cache 旁路策略称为 G-SHiP 。图 4.21显示了 G-SHiP 策略的硬件结构。本文在 GPGPU-Sim 中实现了该策略，图 4.22中显示了 G-SHiP 旁路策略下所有基准测试程序的性能。可以看出， G-SHiP 对 HCS 基准测试程序的性能有不同程度的提升，这种提升的主要原因是 G-SHiP 将部分请求预测为流访问请求，并将它们旁路，从而为其它请求腾出了 cache 空间，保障了这部分请求的局部性，因而提升了 cache 的命中率。

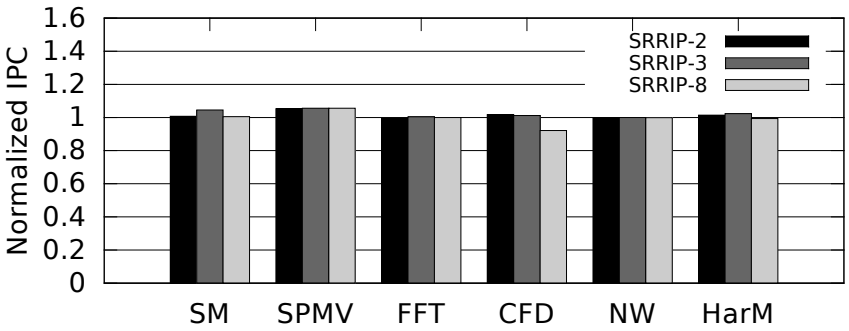

图 4.19　各种替换策略的性能，归一化到基准架构

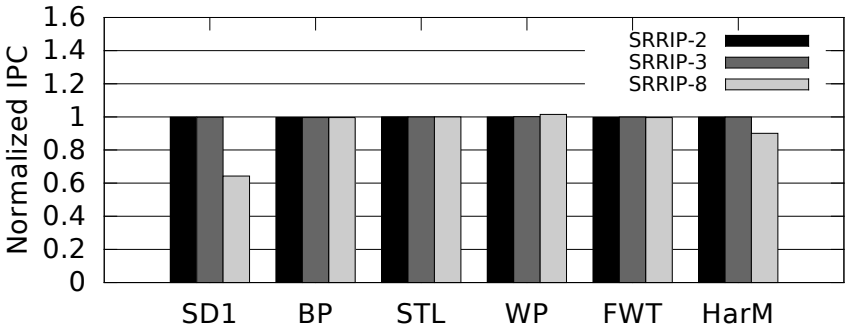

图 4.20　各种替换策略的性能，归一化到基准架构

另一个重要的问题是 cache 冲突，本文引入了 CPU 中的 PDP 旁路策略 [13] 来缓解冲突。本文首先实现了静态 PDP 旁路策略（SPDP-B： static PDP bypass）。SPDP-B 静态选取不同的保护距离 PD 进行实验，然后从中选取获得最佳性能的 PD 值。本文中 PD 的取值范围是从相联度到最大的 PD 值（L1D cache 最大为 64， L2 cache 最大为 256）。图 4.23、4.24、4.25显示了 L1D cache 中随着 PD 的增大，系统性能的变化趋势。对于 cache 敏感型的负载，最优的 PD 能够带来明显的加速。但是相比 CPU 中观察到的结果，这个曲线比较平坦，也就是说性能起伏不是特别大。这是因为 GPU 中 warp 相互交织使得访存行为不太容易预测，而重用距离在这种大规模多线程环境中也就不像 CPU 中那么稳定。不过 PDP 基于重用距离的保护机制对于 GPU 来说还是有作用的，因为毕竟 GPU 的访存行为从统计上来说还是可以预测的 [83]。这种相对平坦的曲线也意味着我们可以采用不那么准确的预测机制而不会损失太多性能收益，从而降低硬件开销。

本文也在 L1D 和 L2cache 中实现了动态 PDP 策略。对于 L1D cache， PDP 有三种设计选择， PDP-private （PDP-P）、 PDP-global （PDP-G）和 PDP-shared (PDP-S)。 PDP-P 在每个 L1D cache 中增加一个 PDP 模块，而每个 L1D cache 都用各自私有的 PD 预测器在本地进行预测。 PDP-G 则是一个集中式的设计，它收

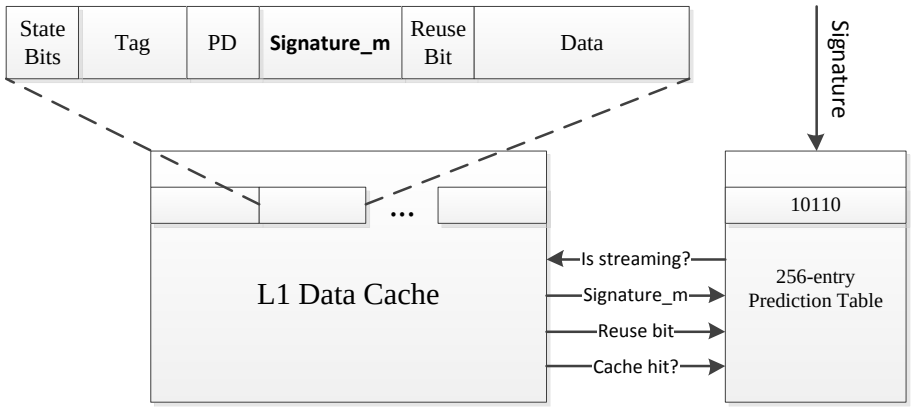

图 4.21　G-SHiP 策略的硬件结构

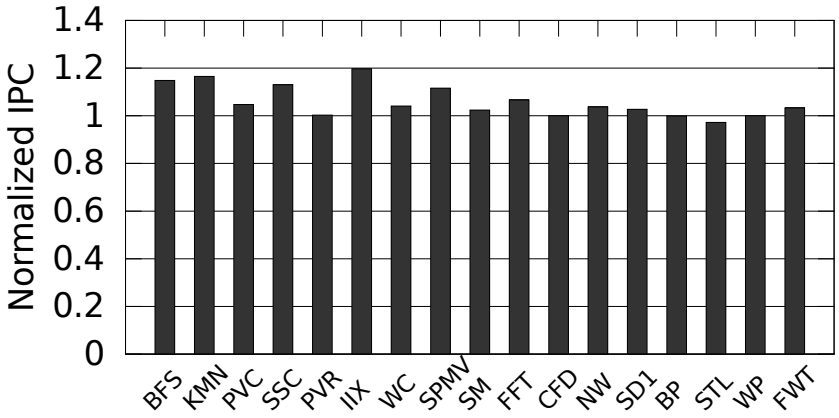

图 4.22　G-SHiP 策略下所有测试程序的性能，归一化到基准架构

集所有 SIMT 核的访存序列，然后为所有的 L1D cache 做出统一的预测。 PDP-S 选择一个（或几个） SIMT 核作为采样核，采样核在本地收集访存序列并做出预测，然后将预测应用到其它所有的核上去。图 4.26、图 4.27 和图 4.28显示了 L1D cache 中采用上述三种 PDP 实现方案所获得的性能。由于缓解了 cache 竞争，这三种 PDP 旁路策略的实现，即 PDP-P 、 PDP-G 和 PDP-S ，都获得了明显的性能提升。 PDP-P 获得了三者之中最好的性能（平均 50.6% ），因为每个 L1D cache 都有自己的预测器，在 SIMT 核之间存在不同的访存行为时，这种设计能够做出更准确的预测。但同时这种设计的硬件实现开销是最大的。 PDP-G 和 PDP-S 获得了比较接近 PDP-P 的性能提升（平均分别是 44.6% 和 45.4% ）。对于大多数负载， PDP-S 表现出同 PDP-P 非常类似的效果。这是因为 SPMD 程序中通常所有的线程的行为都相同或非常类似，故而某一个 SIMT 核所预测的最佳 PD 的值很有可能也是其它 SIMT 核的最佳值。这跟中 Lee 等提出的核采样机制 [202] 所基于的假设非常类似。另一方面，相比这两种实现方案， PDP-G 却展现出不太稳定

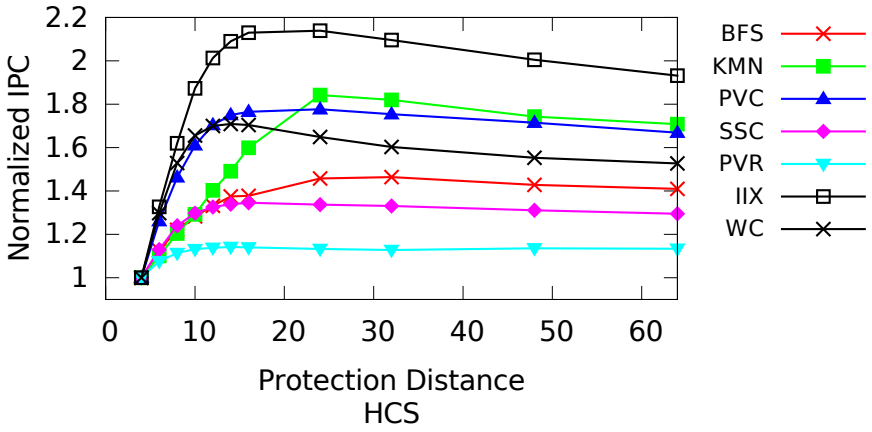

图 4.23　SPDP-B 策略下 HCS 测试程序的性能随着 PD 变化而变化，归一化到基准架构

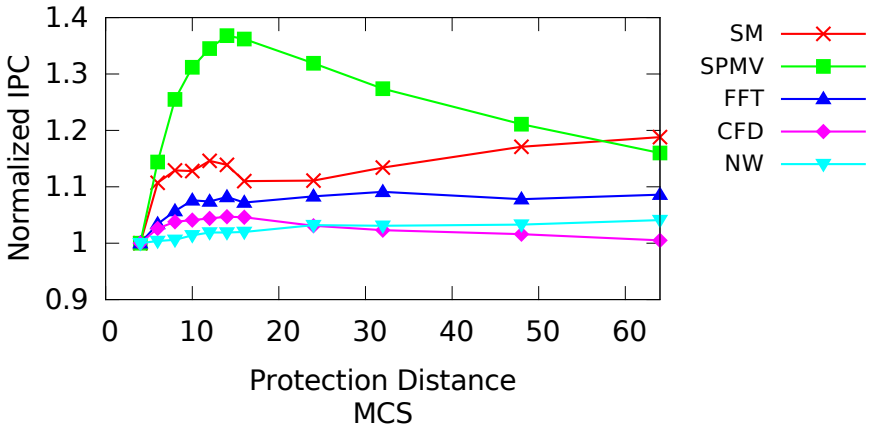

图 4.24　SPDP-B 策略下 MCS 测试程序的性能随着 PD 变化而变化，归一化到基准架构

的性能。本文采用 PDP-S 设计方案作为后续的实验基础，因为该方案能够获得较稳定的性能提升，但相比 PDP-P 其硬件开销要小很多。要注意的是，本文中的动态 PDP 实现采用了每组 32 个 FIFO，而静态 PDP 实现采用了每组 256 个 FIFO。RDD 计数器的数目对于静态和动态 PDP 实现都是 64。

尽管 PDP 旁路策略对于 HCS 负载能够获得明显的性能提升，但这种为 CPU 最末级缓存 LLC 设计的管理策略直接应用到 GPU 中并没有发挥出最佳的效果。原因有以下几个方面：首先，PDP 预测是基于单线程应用的命中率模型，这在 warp 相互交织的 GPU 中并不成立 [226]。正如本文前面讨论的，在 GPU 中由于资源拥塞和 warp 相互交织，cache 的性能同系统的性能并不是直接关联的。因此即便单纯的命中率模型可以指导 cache 的性能提升，它并不一定能带来系统性能的提升。其次，大量的访存请求被发送到 cache 系统中可能会占用掉所有的 MSHR 导致访存暂停（memory stalls）[14, 178]。如果被 cache 旁路的请求同时旁路 MSHR，那么对同一个缓存数据块读访问有可能被多次送往下一级存储层次，

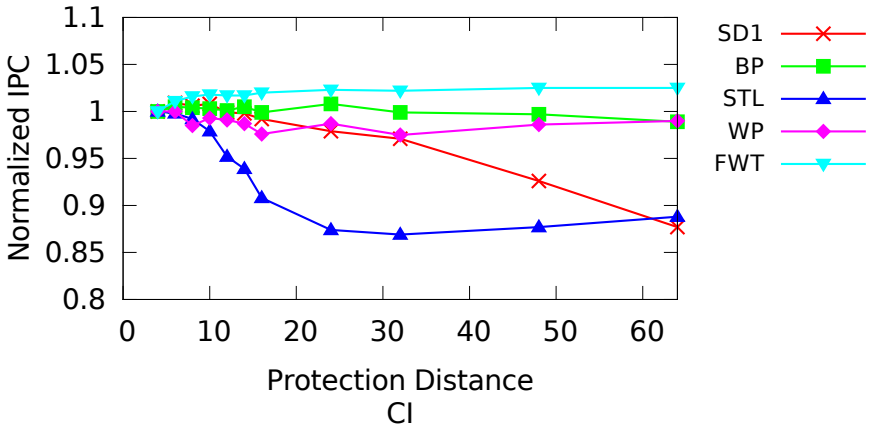

图 4.25　SPDP-B 策略下 CI 测试程序的性能随着 PD 变化而变化，归一化到基准架构

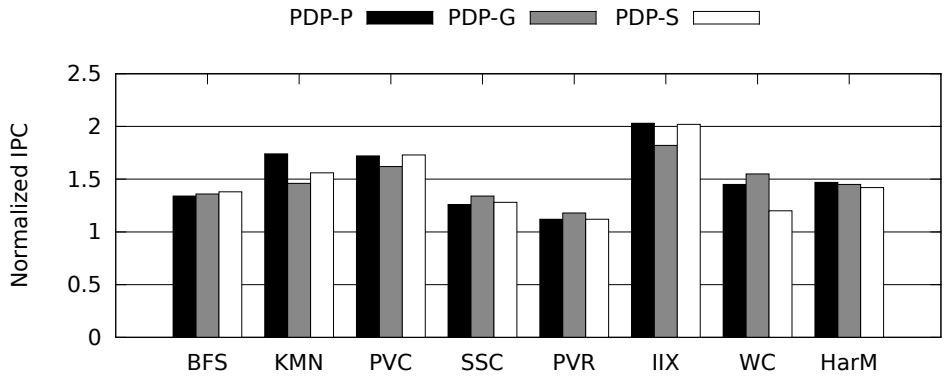

图 4.26　不同 PDP 实现方案下 HCS 测试程序的性能，归一化到基准架构

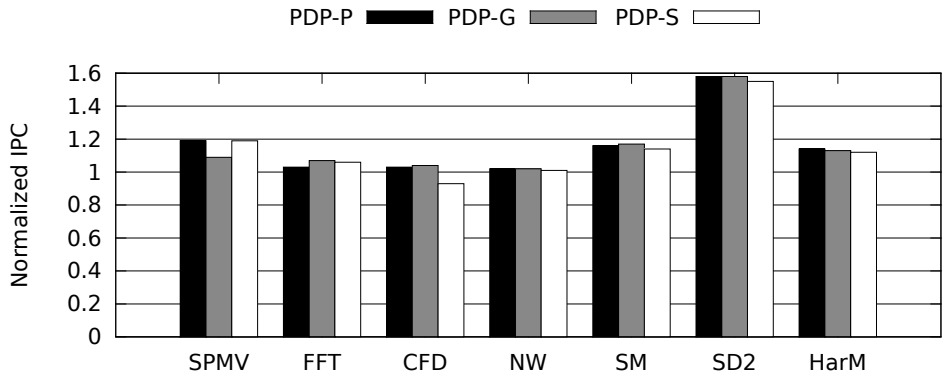

图 4.27　不同 PDP 实现方案下 MCS 测试程序的性能，归一化到基准架构

这会导致局部性的流失，也浪费了 NoC 或者 DRAM 的带宽。第三，当重用距离非常大的时候，PDP 会旁路大部分的请求，而使得大量被旁路的请求发往下一级存储层次，导致 NoC 或者 DRAM 的流量负载非常大。当带宽被消耗完时，就会出现资源拥塞，导致系统性能降级。

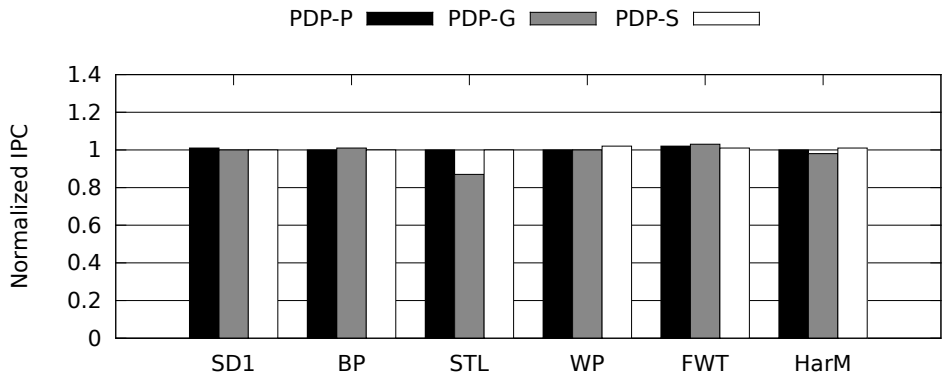

图 4.28　不同 PDP 实现方案下 CI 测试程序的性能，归一化到基准架构

4.3.3　LLC 管理策略

前面提到， NVIDIA GPU 从 Kepler 架构开始将 L1D cache 同纹理 cache 合并，而全局访问的数据只缓存在 L2 cache 中。这种设计我们称为 no-L1 。图 4.29中显示了 no-L1 设计相对于基准架构的性能。不难看出， HCS 负载都获得了一定程度的性能提升，这是因为 L1D cache 冲突太过于剧烈，以致于将 L1D cache 移除后，原先拥堵在 L1D cache 处的大量访存请求可以直接发往 L2 cache ，因而应用程序的性能反而更好。而另一方面， SD1, BP, STL 等程序出现了明显的性能下降，这是因为它们的性能明显受益于 L1D cache ，而移除 L1D cache 之后，其最小数据访问延迟从 L1D cache 的几十个周期变为 L2 cache 的几百个周期，这使得大规模多线程不足以掩盖访存延迟。总的来说， no-L1 设计能够直接避免 L1D cache 的冲突问题，但同时也会使得部分受益于 L1D cache 的程序（有的程序甚至针对 L1D cache 做了优化 [135] ）性能明显降级。

在这种 no-L1 设计的趋势下， L2 cache 的管理变得至关重要，因为 L2 cache 变成了全局访问数据唯一的可读写片上缓存。一旦数据在 L2 cache 中失效，就意味着需要访问片外 DRAM ，而 DRAM 访问量对系统的效能有直接而且明显的影响。如图 4.30所示，在 no-L1 设计中，如果对 L2 cache 使用 PDP 旁路策略，部分程序能够获得明显的性能提升，其中 KMN 获得了 2.56 倍的性能加速。这是因为 PDP 旁路策略能够缓解 L2 cache 的冲突，提升 L2 cache 的命中率，从而直接提升了性能。但总的来看，相比 L1D cache 中的旁路策略， L2 cache 的旁路策略没有那么明显的加速效果，这是因为 L2 cache 的容量较大，因而其冲突并没有 L1D cache 中那么剧烈，另一方面，部分程序的确能够明显受益于 L1D cache ，因而无论 L2 cache 的性能如何优化。对于这些程序， no-L1 设计的性能都会导致一定程度的性能降级。

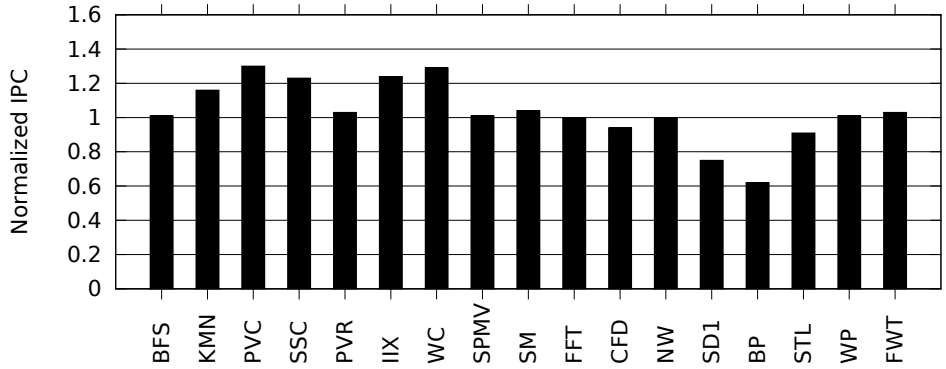

图 4.29　移除 L1D cache 后（ no-L1 ）所有测试程序的性能，归一化到基准架构

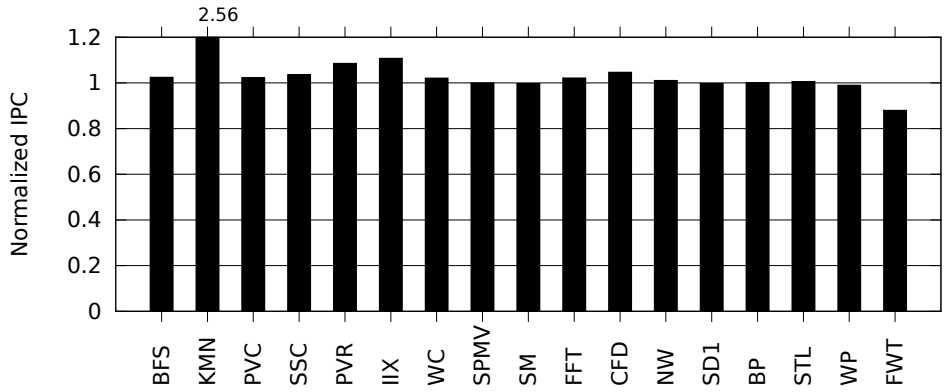

图 4.30　no-L1 架构中对 L2 cache 采用 PDP 旁路策略后所有测试程序的性能，归一化到 no-L1 基准架构

4.4　自适应保护策略

前面提到，在 GPU cache 中应用 CPU cache 管理策略存在很多局限性，约束了先进管理算法的性能潜力。 BIP 算法随机选取部分的缓存块驻留在 cache 中，这能够在一定程度上缓解 cache 冲突，但是 BIP 不能从根本上减小程序的工作集，对于有限的 cache 空间而言， GPU 应用的工作集太大， cache 冲突难以避免。另一方面， DIP 通过组竞争（ set-dueling ）机制在多种替换策略（ LRU 和 BIP ）中动态选取较优的策略。这种组竞争机制在单线程的 CPU 系统中是有效的，因为不同组采用不同的替换策略并不会导致访存序列的先后次序发生变化。即便 cache 被多个线程共享， cache 空间在逻辑上会被划分给各个线程使用，避免了线程之间的相互干扰，以保证线程的服务质量（ QoS： Quality of Service ）。被划分的 cache 分区对于各个线程是私有的，因此在逻辑上仍然不存在访存次序被影响的问题 [14] 。但是在 GPU 中， L1D cache 是被该 SIMT 核上的所有线程束共享的，如果采用组竞争机制，线程束的相互交织会使得不同策略下访存的次序不一样。这是因为 cache 失效使得线程束请求的数据无法到位，这种情况下该线程束

会被调度器挂起，而调度其它线程束执行。不同的调度次序自然就会导致不同的访存请求发送次序。这种 cache 访问次序的不确定性在很大程度上降低了组竞争机制的有效性。

因此，针对 GPU 缓存设计管理策略同 CPU 缓存中得不同点主要是两个方面的问题。一方面， GPU 上有限的 cache 资源同大规模多线程对存储子系统的需求（特别是访存分歧发生时）不相适应，这种情况下只能先保证一部分线程束优先使用 cache ，而另一部分线程束则需要被挂起，或者旁路其发出的请求。否则会出现严重的 cache 冲突。 BIP 替换策略和 PDP 旁路策略能够提升性能，是因为它们在一定程度上缓解了 cache 冲突。但是这些算法没有考虑线程束的优先次序，也就没有考虑最大化线程束内的局部性。而事实上， GPU 应用中的大多数时间局部性都来源于线程束内部的数据重用。另一方面，由于合并单元能够有效捕捉空间局部性， GPU 中流访问模式出现的概率远高于 CPU 。流访问不存在数据重用，因而被缓存在 cache 中没有好处，相反，这些请求只会污染 cache ，浪费 cache 空间。这种情况下，就需要一种流访问模式的监控和预测机制，对于这些请求或缓存数据块，应该尽可能将其旁路或者优先被选为替换的候选，以避免 cache 污染。

因此，从根本上来说，不论是防冲突还是防污染，都是需要对 cache 块进行不同程度的保护。首先，应该优先保护重用可能性较大的缓存数据块，这个策略同 CPU 中的 cache 管理原则是类似的；其次，为了最大可能地维护线程束内的重用，应该优先将部分线程束的访存请求插入到 cache 中，而且在替换发生时这些缓存数据块得到相对较多的保护，而优先级较低的线程束对应的缓存数据块则得到较少的保护；最后，对于流访问模式，应该将其区分对待，减少对这类缓存数据块的保护，类似地，对于流式请求，应该尽可能准确地预测到并将其旁路，以避免 cache 污染。

为了优先保护重用可能性较大的缓存数据块，我们需要重用预测机制，也就是说，需要知道哪些缓存数据块的重用可能性较大。常见的重用预测机制有两种：最近访问过的很可能在不久的将来被重用（例如 LRU 和 NRU ）；最近重用过的很可能在不久的将来被重用（例如 RRIP ）。前者对于数据局部性很高的应用效果很好，但是不能有效应对工作集太大的情况或者 scan 访问模式 [12] 。从硬件实现角度来看，前者需要维护缓存数据块的最近访问时间戳，而后者则需要相应的数据位（ RRPV ）来表示重用的可能性。 RRIP 实质上也是一种保护机制，其保护距离就是 RRPV 。在 scan 模式发生时，这种保护能够让一部分重用可能性较高的缓存数据块驻留在 cache 中，免遭替换。对于重用可能性较高的缓存数据

块，我们应该给予更多的保护。因此，每当缓存数据块被命中之后，其对应的RRPV会被设置为0，即将其保护距离设置为最大。

为了最大可能地维护线程束内的重用，对于新插入的缓存数据块应该根据其对应线程束的优先级给予不同程度的保护。优先级较高的线程束发出的请求应该获得更多的保护。这样，当工作集很大时，这种策略能够在一定程度上缓解cache冲突。 PDP策略是对RRIP的进一步一般化，提出了保护距离PD的概念。PDP通过动态采样估算重用距离，并计算出的最佳保护距离 PD_{opt} ，并据此对缓存数据块设置保护，同时给予最近重用过的缓存数据块更多的保护。但是，在真实的应用程序中可能存在各种访存模式的混合，而PDP对所有缓存数据块使用同一个PD，这就制约了它的性能潜力。比如，在流模式访问频繁出现时（这种情况在GPU中很常见），重用数据的重用距离会很大，这样就使得所有被插入的缓存数据块都得到很长时间的保护，然而这些缓存数据块中可能有很大一部分是不会被重用的，它们不应该得到相同程度的保护。如果能够识别出流模式请求，就可以减少对其相应缓存数据块的保护，而驻留在cache中的就会更多的是重用可能性较大的数据。

为此，我们的替换策略自适应地对不同类型的访问数据给予不同程度的保护，称为自适应优先级保护（APP： Adaptive Priority Protection）替换策略。对于给定的一个缓存数据块被插入到cache中时，保护优先级 PP 的计算涉及两个方面的信息：缓存数据块对应的线程束的优先级 P_{warp} 和缓存数据块是否被预测为流访问请求 B_{stream}。两部分信息对 PP 的贡献也可以动态变化：当线程束内的数据重用占主导地位且线程束间的冲突导致大量局部性流失时，应该增加 P_{warp} 对 PP 计算的影响，以拉大不同优先级的线程束之间的权限差异；当流访问预测的准确率很高时，可以将 PP 设置得尽量低，反之，如果准确率不太高，则应该尽可能减少 B_{stream} 对 PP 计算的影响。

APP策略以PDP策略为基础，也就是说APP也会动态估算重用距离，并计算 PD_{opt}。每当有新的请求被插入到cache中时，该缓存数据块对应的剩余保护距离 RPD 被设置为 PD_{opt}。在每次有请求访问目标组时，组内所有的缓存数据块对应的 RPD 减1，即 $RPD = RPD - 1$。另外，同RRIP和PDP类似，当一个缓存数据块被命中时，控制器会将其对应的 RPD 设置为 PD_{opt}。

当接受到一个新的请求时， cache控制器首先搜索无效状态（invalid）的缓存数据块，如果找到，则直接插入。如果没有找到， cache控制器优先选取 $RPD = 0$ 的缓存数据块作为替换victim，如果有多个，则选取其中 PP 最小的。如果不存在 $RPD = 0$ 的缓存数据块，则在目标组中选取 PP 最小的缓存数据块进行替换。

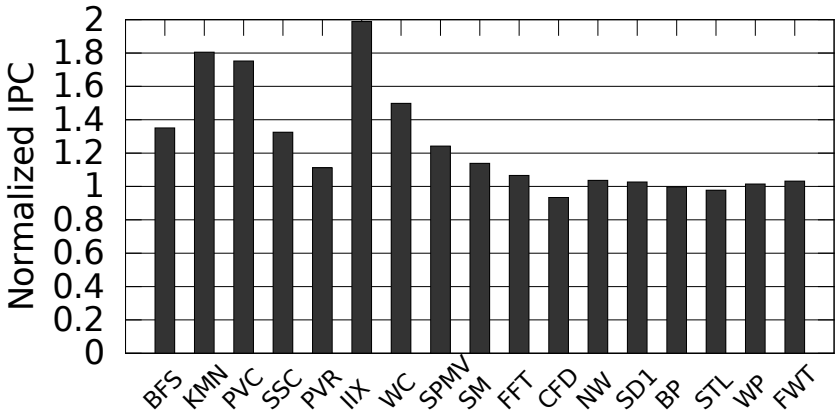

图 4.31　APP 策略下所有测试程序的性能，归一化到基准架构

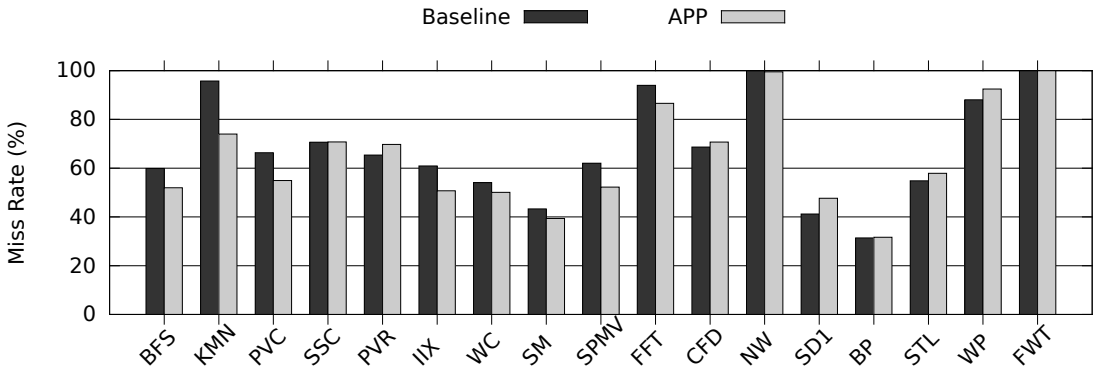

图 4.32　APP 策略相比基准架构的 L1 cache 失效率

旁路策略也是基于同样的框架，区别在于，每当有新的请求被插入到 cache 中时，该缓存数据块对应的剩余保护距离 RPD 被设置为 PP ，而不是 PD_{opt} 。这就意味着， PP 越小的缓存数据块得到的保护就越少。当接受到一个新的请求时， cache 控制器如果能找到无效或者 $RPD = 0$ 的缓存数据块，则直接替换，否则，传入请求将会被旁路，同时组内所有的缓存数据块对应的 RPD 减 1 。

图 4.31显示了 APP 管理策略下所有基准测试程序的性能。可以看出， APP 能够显著提升 HCS 基准测试程序的性能，这主要是因为 APP 同时考虑了缓存污染和冲突两个问题，引入了基于历史信息的流访问模式预测机制和基于重用距离的保护机制，因而能够很好地缓解污染和冲突。相比 G-SHiP 策略， APP 考虑了重用距离，因而更好地保护了数据的局部性。相比 PDP 策略， APP 能够动态识别流访问模式，因而克服了 PDP 策略中对所有缓存数据块只使用一个保护距离的缺陷，有效减少了缓存污染的发生。总的来看， APP 策略获得了相比 G-SHiP 和 PDP 策略更好的性能。图 4.32说明了 APP 策略的性能提升是源于 L1 cache 失效率的明显改善。

4.5 本章小节

本章首先对 GPU 中的 cache 属性和应用程序的 cache 行为进行了详细的分析，指出了 GPU 中由于大规模并行导致的严重的 cache 污染和冲突问题。针对这两个主要问题，本章从替换和旁路策略入手，进行针对 GPU 的定制化设计。首先在 GPU 中实现了 CPU 中现有的先进的 cache 替换策略： RRIP 、 SHiP 和 PDP 替换策略。实验表明，先进的替换策略能改善 cache 污染和冲突的问题，但是效果非常有效。这主要是因为 GPU 的大规模并行环境中程序工作集与 cache 容量极不相称，无法通过单纯改变替换的优先顺序来解决。然后我们在 GPU 中引入了相应的旁路策略： G-SHiP 和 PDP 旁路策略。二者能够更好地缓解 cache 污染和冲突，得到了明显的性能提升。最后我们结合两种策略的思想提出了 APP 管理策略。该策略兼顾了污染和冲突两个问题，从而获得了相比 G-SHiP 和 PDP 旁路策略更好的性能。

第五章　协同旁路与线程束调节

在第 4章中我们讨论了 cache 替换和旁路策略，先进的 cache 替换和旁路策略能够缓解 cache 冲突，提升系统性能。但是，我们也发现大量的旁路请求仍然可能导致其它片上资源（比如 NoC ）出现拥塞。拥塞一旦发生，性能就无法再提升，而且旁路会导致数据局部性的流失，严重时甚至出现性能降级。可见，单纯的旁路策略由于只考虑了 cache 的性能，而没有考虑其它片上资源的使用情况，因而不能最大程度地发挥出 GPU 的性能潜力。为了缓解拥塞，需要根据请求总量和频率动态地调整并发度，避免过度消耗瓶颈资源，这一方面能够减少局部性的流失，另一方面又能够充分利用片上资源。

单纯的线程调节技术 [17] 根据 cache 冲突的情况调整并发度，但事实上并不能达到最佳性能，这是因为为了保证 cache 的命中率，线程调节技术会在检测到冲突时减少并发线程数目，但是这有可能会导致其它片上资源（比如 NoC 带宽）的低利用率，而且减少并发度也意味着减少了大规模多线程隐藏访存延迟的能力，故而不能达到最佳的性能。如果考虑充分利用 NoC 带宽等资源，则有可能造成 cache 性能下降，也会制约系统性能。总的来看，单纯的线程调节技术也不能兼顾各种片上资源的合理利用，因而不能发挥出处理器的最佳性能。本章针对单纯的旁路策略和单纯的线程调节技术存在的局限性，考虑将二者结合起来设计 GPU 的动态执行控制框架，**提出协同旁路与线程束调节技术**，即 CBWT （ coordinated bypassing and warp throttling ）。本章首先将旁路策略同静态线程束调节相结合，以显示 CBWT 的性能潜力。然后介绍动态监测硬件机制，能够在运行时动态获取片上资源使用情况。最后提出动态预测算法， CBWT 预测器在反馈信息的基础上按照该算法估算出最佳的活跃线程束数目，并对线程束调度器进行调控。

5.1　线程束调节的性能潜力

前面介绍了线程调节（ thread throttling ）技术，即通过调节并发线程的数目来合理利用片上资源已达到提升性能、节省功耗的目的。 CCWS [17] 通过线程束调节（ WT： warp throttling ）技术来缓解 GPU 中线程束间冲突（ inter-warp contention ），即将部分线程束挂起，只允许部分线程束向存储子系统发射访存请求，以避免线程束竞争使用片上资源。需要指出的是， GPU 的执行模型中包含线程块、线程束和线程三个并发层次，而其中线程块和线程束都是可供调控的。本文并不考虑线程块的调节技术，因为同一个线程块内的所有线程束是必须被调度到同一个 SIMT 核上执行的，这就意味着较细粒度的线程束调节方案实际是包含了粗粒度的线程块调节方案的，即挂起某个线程块可以通过挂起该线程块内

的所有线程束来实现。不难看出，线程束调节的调整空间更大，其性能潜力要大于线程块调节 [14, 17] 。

WT 同 CPU 中采用的线程调节技术本质上是一样的，都属于对并发度进行调控的技术，区别在于调节并发度的目的有不同的侧重点。 CPU 中并发线程数目本身就不多，其性能瓶颈可能是细粒度的同步开销或者有限的主存带宽，因此调控的主要目的是减少线程间同步和通信的开销或者避免过分消耗主存带宽 [15] 。而 GPU 的应用通常没有太多全局同步操作或者线程间的通信，其性能瓶颈大多数的情况下是主存带宽。而由于 GPU 中访存分歧问题的存在，对于 cache 敏感型应用， cache 容量也常常成为性能瓶颈。不仅如此，大规模的访存请求使得 GPU 的各种片上资源都可能成为性能瓶颈，包括 MSHR 、 NoC 带宽和内存控制器 MC 等等。这种性能瓶颈问题当然在 CPU 中也存在，但并不如 GPU 中这么常见。因此， GPU 中的线程束调控则更侧重提高 cache 的命中率，以及缓解片上资源的拥塞。当然，通过线程束调节来避免过分消耗主存带宽从而节省功耗也是可行的，但是本章先考虑性能问题。

对于任意的 GPU 程序，每个 SIMT 核上支持的并发线程束数目是有上限的（ Fermi 中的 SIMT 核支持最多 48 个线程束同时执行），我们用 n_t 来表示这个上限。程序实际运行时，这个上限并不一定能达到，因为线程束数目还受限于对资源的需求量。这里的资源需求包括寄存器数目、暂存器（ scratchpad ，或称便笺存储器）容量等。我们用 n_{max} 具体某个程序运行时的实际上限。通常情况下，主要有足够的任务量， GPU 会在每个 SIMT 核上启动 n_{max} 个线程束。这种方案实际上就是尽可能多的启动线程束，一方面充分利用计算资源，另一方面利用大规模多线程来隐藏访存延迟。但是正如前面分析的，这种方案并不一定能够带来最佳的性能。为了更直观地说明这个问题， CCWS 中引入了静态线程束限制（ SWL ： static wavefront limiting ）技术。 SWL 允许程序员或编译器静态地设置最大活跃线程束（ MAW ： Maximum Active Warp ）的数目。 SWL 通过遍历所有可能的 MAW 值来找到最佳的线程束数目 MAW_{opt} 。实验表明不同的程序其 MAW_{opt} 不同，即使是同一个程序，如果采用不同的输入数据，其 MAW_{opt} 也可能会发生变化。为此， CCWS 引入了局部性流失监测（ lost locality detector ）机制 [17] 提供反馈信息，以动态调整 MAW 。

相比 CCWS 这种通过降低线程并发度（即 WT 技术）的方法来缓解 cache 冲突， cache 旁路策略是试图选择性地将 cache 空间分配给重用概率高的缓存数据块，而将死块或者重用概率低的缓存数据块直接发往下一级存储层次，从而避免冲突。 WT 的好处是它不但能够缓解 cache 冲突，还能缓解片上资源拥塞的问题，但是这种调控在缓解了 cache 冲突的同时有可能导致片上资源空闲；而旁路的好

处是它只控制 cache 的访问行为，不会导致其它片上资源空闲的问题，但是 cache 旁路策略无法控制其它资源的使用情况，因而无法解决资源拥塞问题。当旁路策略同线程束调节技术一同使用时，就有机会同时保证 cache 的高效和其它片上资源的充分利用，从而创造了进一步提升系统性能的潜力。这就是本章要提出的*协同旁路与线程束调节*技术，即 CBWT [227] 。

资源拥塞。 大规模多线程会引发*资源拥塞*，这在 CPU cache 中并不常见。资源拥塞也是导致 GPU 中 cache 性能不直接与系统性能正相关的原因。在一个 SIMT 核中，当几十个线程束在很短的时间内产生成百上千的访存请求， cache 本身就可能出现 cache 块或者 MSHR 等资源短缺的情况 [178] 。在这种情况下，如果没有旁路的支持，失效的访存请求便既不能被服务，也不能被发送到下一级存储层次。它们只能等待正在占用资源的请求得到服务然后释放资源，这就会引起频繁的访存暂停（ memory stall ），导致计算资源以及 NOC 和主存带宽的空闲，从而严重制约了整个系统的性能。除了 cache 本身的资源以外，还有很多其它片上资源可能成为性能瓶颈，比如 NoC 带宽和 DRAM 带宽。当 SIMT 核产生大量访存请求在 L1D cache 中失效时，这些失效请求会通过 NoC 被发往 L2 cache ，从而迅速耗尽 NoC 带宽。 NoC 带宽一旦被耗尽，继续往 NoC 中注入负载将使得数据包的传输延迟明显增大，这不仅有损性能，还使得效能明显下降，因为程序受限于 NoC 带宽而无法获得任何加速（如果考虑 cache 行为甚至会引起减速），但是计算部件在大部分时间里维持高频率空闲等待的状态，这部分功耗没有转化为吞吐率。值得注意的是，在 GPU 芯片中，计算部件和 NoC 在系统功耗中占到的比例是非常大的（分别是 11.4% 和 9.5% [94] ，这里的系统功耗包含了 DRAM 功耗，如果除去 DRAM 功耗，这两个比例会更高）。图 5.14显示了 HCS 测试程序的平均 NoC 延迟随着 MAW 的变化而变化。这里的 NoC 延迟是指一个数据包通过某个网络所花费的时钟周期数，而平均 NoC 延迟则是所有数据包 NoC 延迟的平均值。从图中可以看出，对于大多数的 HCS 测试程序， NoC 延迟在 MAW 大于 5 时开始明显增大，这是因为并发的线程束越多，单位时间内生成的访存请求就越多，因而迅速地耗尽了 NoC 带宽。

图 5.2中显示了在未启用旁路策略的情况下， MAW 的变化对的系统性能的影响。不难看出， HCS 程序的最佳 MAW 值都小于 24 ，这是因为过多的并发线程束会导致 cache 冲突和资源拥塞。但是 MAW 太小也会导致性能下降，这是因为硬件资源没有得到充分利用。图 5.3中显示了 PDP 旁路策略（详细描述见第 4 章）被启用的情况下，随着 MAW 增加，相对基准架构的系统性能的变化情况。在本文的后续章节中，这个 SWL 和旁路相结合的策略被称为静态协同旁路和线程束调节，即 SCBWT 。 SCBWT 通过遍历所有可能的 MAW 值搜索到的最优性

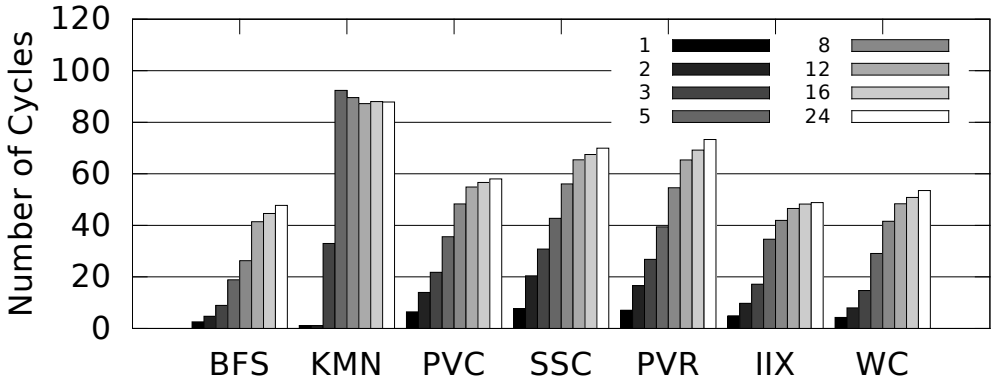

图 5.1　基准 GPU 中数据包的平均 NoC 延迟随着 MAW 从 1 增大到 24 而变化的情况

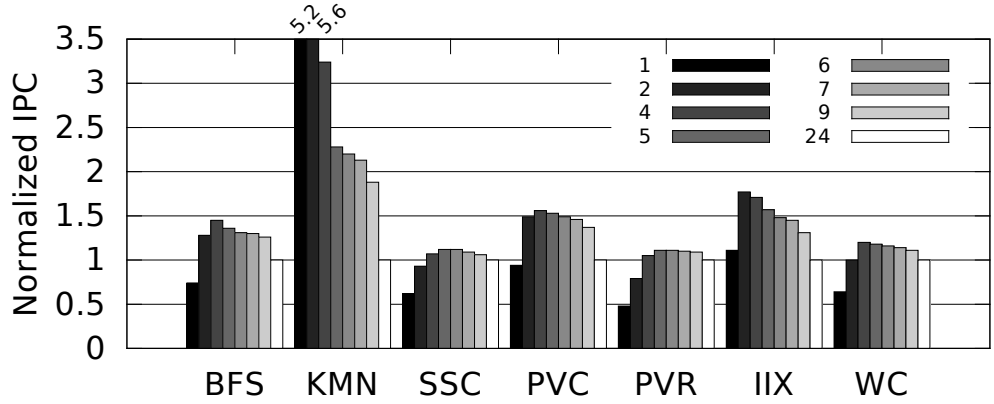

图 5.2　在旁路未被启用的情况下，SWL 在不同多线程限制（即最大活跃线程束 MAW ）下的性能，归一化到未启用旁路的 SWL-24 （即 MAW 为 24 的 SWL 策略）

能对应的 MAW 取值，我们用 MAW_{opt} 来表示。 MAW_{opt} 所对应的 SCBWT 策略是动态 CBWT 策略的上限，在后续的实验中用于对比，称为 Best-CBWT 策略。对于大多数的测试程序，当旁路被启用之后，相比旁路未启用的 SWL ， MAW_{opt} 的值都变大了，原因是旁路策略的启用允许更多的线程束向存储子系统发射访存请求，而不会致使 cache 的性能降级，因为 cache 的访问行为受到了旁路策略的保护。这些额外增加的访存请求则能够有效利用空闲的 NoC 或者 DRAM 带宽，以尽快返回流水线请求的数据，减少了计算部件的空闲等待时间，这就有可能进一步提升系统的性能。此外，同单纯的 SWL 实验中观察的现象类似， SCBWT 中不同的测试程序的 MAW_{opt} 取值不同，并且不同的输入数据集也会导致不同的 MAW_{opt} 取值。这就需要我们采用动态方法来获取 MAW_{opt} 取值。

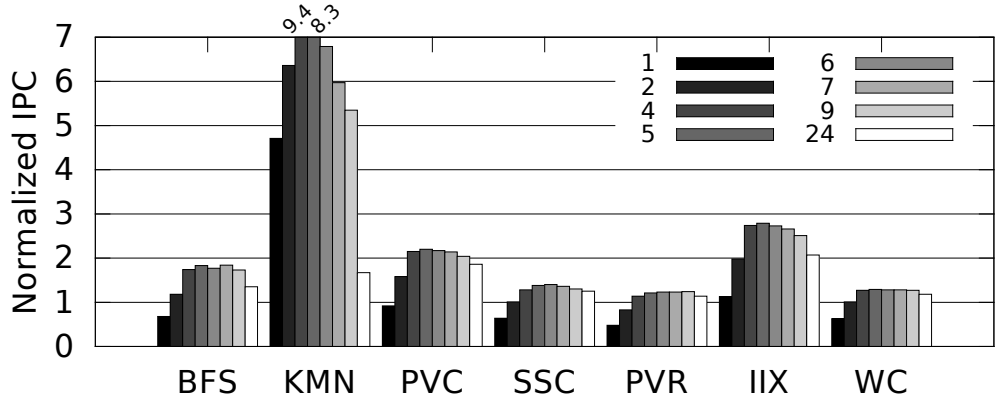

图 5.3　在旁路被启用的情况下，SWL 在不同多线程限制（即最大活跃线程束 MAW ）下的性能，归一化到未启用旁路的 SWL-24 （即 MAW 为 24 的 SWL 策略）

5.2　面向吞吐率的自适应资源管理

理想情况下，即程序中没有分支分歧和访存分歧问题且数据相关很少的情况下， GPU 能够最大程度地发挥出其吞吐率的优势。但是现实中不规则程序的数量远比规则程序要多，要真正实现在 GPU 上执行主流的通用计算，就必须应对分支分歧和访存分歧问题。一旦出现分支分歧， GPU 计算部件的利用率就会明显下降，吞吐率自然就受限。同样地，当出现访存分歧的情况时， GPU 的性能就受限于存储子系统，计算部件则经常处于等待数据的状态。为了最大化吞吐率，在访存分歧发生时，就需要尽可能充分利用存储子系统中有限的资源，发挥出其数据供给的最大潜力，最大程度地减少计算部件等待数据的时间。这种最大化吞吐率的片上存储资源管理就是本节要提出的*面向吞吐率的自适应资源管理*。

5.2.1　访存模式监测

前面提到，单纯的旁路策略或者单纯的线程束调节 WT 技术都无法发挥出 GPU 的最大性能潜力，要充分利用 GPU 的片上资源，就需要将旁路策略和线程束调节技术结合到一起。单纯的 WT 技术根据 cache 的冲突情况来动态调整活跃线程束数目，而当把二者有效结合时，不仅需要动态获取 cache 冲突的情况，还需要其它资源的使用信息。本小节介绍我们为动态监测片上资源使用情况而引入的硬件扩展。监测部件尽可能地利用现有硬件来实现，以最大程度地减少额外硬件开销。

冲突监测。 监测 cache 冲突本质上就是要监测过早移除，或者所谓的局部性流失 [17] 。当一个被移除的缓存数据块被再次访问时，就可以被认为是一次过早移除。本文采用同 CCWS 中同样的术语来表示在一个给定的采样周期内监测到的过早移除发生的次数：局部性流失度（ LLS： lost locality score ）。当 LLS 很

高时，硬件有必要调控并发度来缓解冲突。 CCWS 在每个 SIMT 核的 L1D cache 中添加一个 VTA （ victim tag array ） [17] 来记录过早移除。 VTA 中记录最近被移除的缓存数据块的标签（ tag ），一旦其中某个缓存数据块被重新访问，就意味着过早移除的发生，而相应的 LLS 就会增加 1 。 VTA 的硬件开销是比较大的， CCWS 中的实验表明，要保证有效获取 LLS 的估测值，每个线程束需要 16 个 VTA 记录（ entry ），对于支持 48 个线程束的 Fermi 架构而言，每个 SIMT 核就需要 $16 \times 48 = 768$ 个 VTA 记录，而 VTA 的每条记录需要 40 位，则每个 SIMT 需要 $768 \times 40 \div 8 = 3.75KB$ 的额外硬件。对于一个 16 核的 GPU ，总共的硬件开销就是 60KB ，并且这个开销会根据 SIMT 核数目和每个核支持的线程束数目的扩展而增加。这个开销在 CCWS 的方法中是必需的，因为 CCWS 需要记录每个 SIMT 核中的所有的线程束的局部性流失程度，然后让局部性流失程度较大的线程束优先向存储子系统发射访存请求。而本文采用将旁路和线程束调节相结合的方法，并不需要对线程束的 cache 访问行为进行细粒度的调控，因为旁路策略会保护 cache 免遭冲突，而线程束调节则更注重调控其它片上资源的合理使用。因此，我们只需要监测 cache 冲突是否发生，其严重程度是否达到某一个阈值即可。

为了尽可能地降低硬件开销，本文的监测机制是通过扩展已有的 L2 cache 的标签阵列（ tag array ）来实现。如图 5.4所示，每个 L2 缓存记录（ cache entry ）包含以下条目：状态位（ state bits ）、保护距离（ PD ）、标签（ tag ）、数据（ data ）和 *VB* （ *victim bits* ）。其中状态位、 PD 、标签和数据是 PDP cache [13] 中已有的，而 VB 则是额外添加的条目。 VB 正是用来记录 “ 局部性流失 ” 的，它实质上是一个多位掩码（ bit masks ），其中每一位对应一个 SIMT 核，负责记录在该 L2 cache 的缓存数据块被移除之前来自其对应 SIMT 核的访存历史。这个 VB 位在 L2 cache 完成了来自对应 SIMT 核的一次访存请求时被置上（ set ），而当这个 L2 cache 的缓存数据块被移除时或者初始化时， VB 位将被清零。有了 VB 之后， L1D cache 中发生的过早移除就能被 L2 cache 监测到，即当 L2 cache 发现同一个 SIMT 核向它发送了之前发送过的访存请求时（ L2 cache 命中且对应的 VB 位为 1 ），则认为发生了一次过早移除。在这种情况下， LLS 将被增加 1 ，表示有一次重用机会流失了。

为了表示 cache 冲突的严重程度，我们引入冲突率（ LLS rate ）的概念。冲突率是指由 cache 冲突造成的局部性流失在所有失效请求中所占的比例，其计算方法是用 LLS 除以发送到 L2 的请求总数。如果冲突率高于某个阈值， L2 cache

图 5.4　对 L2 Cache 的硬件扩展

就会向 L1 cache 发送提示信息，将监测到的冲突情况通知 L1 cache 。我们还可以通过采样机制来降低硬件开销，即 L2 cache 的部分（比如每 64 个组中选一个）组被选为采样组（ sampler ），只在采样组中的缓存数据块中添加额外的 VB 条目，其它组的缓存数据块保持不变。值得说明的是，我们的这个设计是低成本有效（ cost-effective ）的，因为 L2 中本来就有的标签阵列被用来收集 L1 cache 的访问历史信息，而不需要像 VTA [17] 那样额外增加一个标签阵列到每个 L1 cache 中。我们还可以通过让多个 SIMT 核共享同一个 VB 掩码位来进一步降低 VB 的硬件开销。

拥塞监测。 可能发生拥塞的片上资源包括 cache 块、失效队列、 MSHR 、 NoC 带宽和 DRAM 带宽等等。 L1 和 L2 cache 拥塞（ cache 块、失效队列和 MSHR ）可以通过记录 cache 资源预订失败（ reservation failure ）来监测。我们可以通过增加相应的计数器来记录资源预订失败的次数。带宽耗尽导致的拥塞（ NoC 或者 DRAM ）则可以通过监控 NoC 和内存控制器来检测。这些监测机制为旁路和并发线程控制提供了反馈信息。当 cache 拥塞被检测到时， cache 控制器可以触发旁路以缓解拥塞 [178] 。而当 NoC 或者 DRAM 拥塞被监测到时，线程束调度器（ warp scheduler ）会得到通知，进而根据拥塞和旁路的统计信息来调控活跃线程束的数目。在本文的实验中，我们主要考虑监测 NoC 的拥塞状况，用来估算 MAW_{opt} （算法详见第 5.2.2节）。 NoC 监控模块随机选取部分 NoC 数据包作为采样样本，样本数据包的 NoC 延迟被硬件计数器记录下来，每过一个采用周期，监控模块会计算该周期内所以样本的平均 NoC 延迟，这个信息将被送往 CBWT 预测器指导预测。

5.2.2 最优并发度预测

前面提到的动态监测机制为预测 MAW_{opt} 提供了必要的信息。本小节则主要考虑基于这些信息如何来动态预测 MAW_{opt} 。如图 5.5所示， CBWT 预测器在每个采样周期内动态搜集的信息包括： L1D cache 旁路发生的频繁程度、 NoC 的拥塞程度以及 L1D cache 的冲突状况。 L1D cache 旁路发生的频繁程度用旁路比率（ bypass rate ）来表示。旁路比率是指被旁路的请求占所有请求的比例。 NoC 的拥塞程度用 NoC 平均延迟来衡量，而 L1D cache 的冲突状况则是同第4章中的冲突率来表征。图 5.5中显示了这些信息的反馈路径。在本文的实验中，每 16K 个发往样本 L1D cache （即被选中用于采样的 L1D cache ）的访存请求作为一个采样周期。 CBWT 预测器试图根据这些信息来估算 MAW_{opt} ，然后让线程束调度器调控并发度，以保持让 NoC 处于繁忙状态但又不至于过度拥塞。

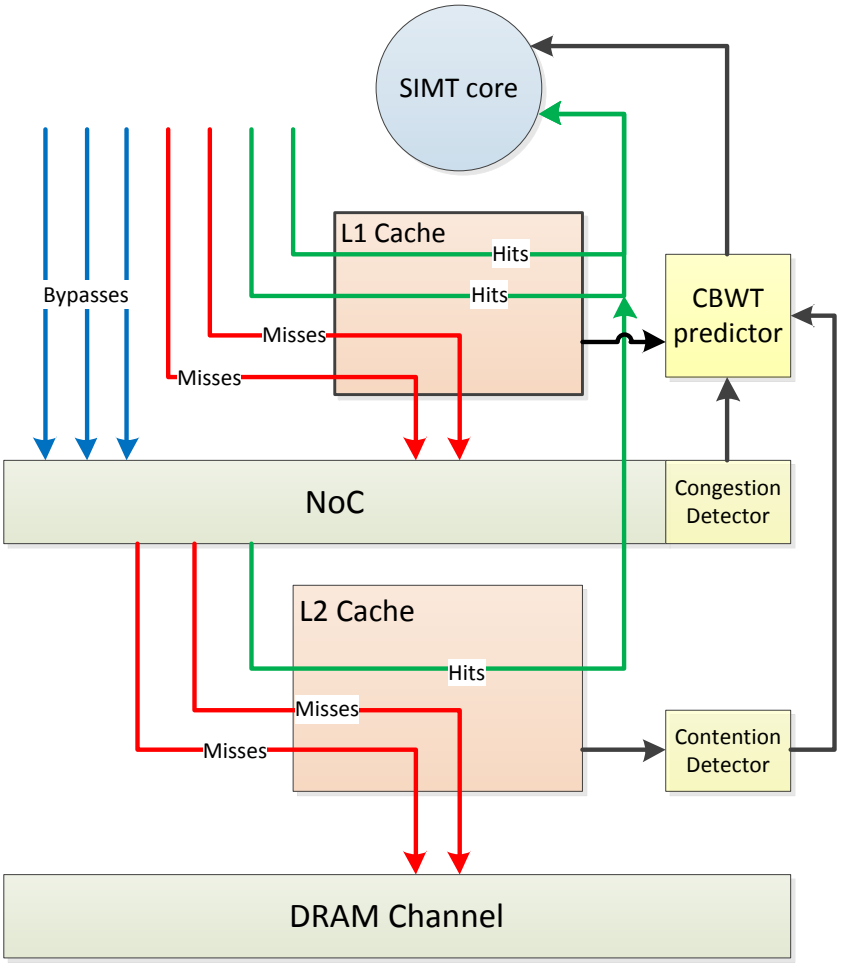

图 5.5　CBWT 的 Cache 层次结构概览。访存请求包括命中请求、失效请求和旁路请求。L1D 和 L2 cache 都由 PDP 旁路策略保护以避免冲突。系统中加入了额外的采样模块来监控冲突和拥塞。CBWT 能够动态自适应地控制活跃线程束数目来充分利用 cache 空间和其它片上资源。

如图 5.6所示，MAW 的调整过程分为多个步骤。Kernel 开始执行时尽可能多地启动线程束，即启动 n_{max} 个线程束。如果 MAW 不小于阈值 T_w，CBWT 预测器对 MAW 进行粗粒度调整，即基于监测到的旁路比率（❶）估算一个最优的 MAW 取值，并通知线程束调度器将 MAW 设置为这个值。粗粒度调整能够保证 MAW 迅速收敛到理论最优值的附近，然后预测器开始进行细粒度调整。当 MAW 下降到低于阈值 T_w，其调整的步幅变为 ±1，即每过一个采样周期增加 1 或者减少 1：当“NoC 的延迟大于阈值 T_{NoC_H}”或“NoC 延迟的变化小于阈值 T_{NoC_G}”时，MAW 减 1（❷）。考虑 NoC 延迟的变化的原因是为了不至于在降低 MAW 对 NoC 延迟作用不大时停止下调。类似地，当 NoC 延迟小于阈值 T_{NoC_L} 时，MAW 则增加 1（❸）。通过这种细粒度调整能将网络延迟控制在目标区域之内，但又不至于因太过激进而导致性能大幅度降级。

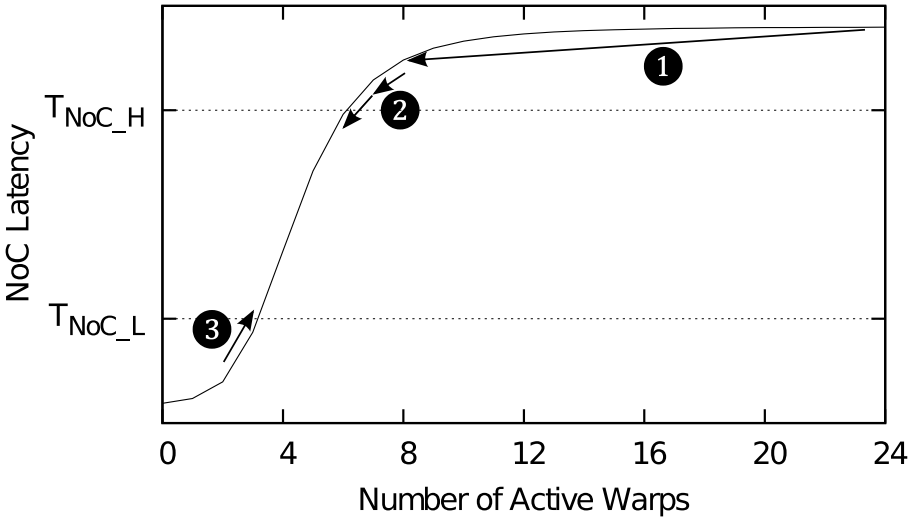

图 5.6 随着 MAW 的上升，NoC 延迟逐渐增大直至带宽耗尽，CBWT 根据反馈信息来维持 NoC 的繁忙，同时也将拥塞程度控制在 T_{NoC_L} 和 T_{NoC_H} 之间。

粗粒度 MAW 预测（❶）是基于一个简化的访存行为模型。值得提出的是，我们不需要精确的模型，一方面是因为在后续的过程中会根据反馈对 MAW 进行细粒度调整，另一方面，简单模型对应简单的硬件逻辑，因而更容易实现。在简化的访存模型中，假定每个 SIMT 核中的每个线程重复地访问单个缓存数据块，线程之间没有共享，并且线程束是按照轮转法进行调度。在这种理想情况下，工作集的规模是同活跃线程束的数目成正比的，而通过降低 MAW 使得工作集与 cache 容量匹配，就能最大化局部性。经过一个采样周期，被旁路的请求数目 m_b（表示没能被 cache 容纳的数据请求）和总共的访存请求数目 m 被硬件计数器记录下来，而 m_b/m 则表示了工作集中没能被 cache 容纳的部分所占的比例。据此，MAW 则被下调到：

$$n_t = (1 - \frac{m_b}{m}) \times n_{max}, \tag{5.1}$$

其中 n_t 是通过线程束调节让工作集同 cache 容量匹配时的最优 MAW 取值。

图 5.7中显示了本文中提出了 MAW 搜索算法的执行细节。当 cache 冲突和资源拥塞被检测到（通过监测冲突率和平均 NoC 延迟）时，预测器开始基于 Eq. 5.1 进行粗粒度的激进式搜索。当 MAW 小于 T_w 时，预测器转而使用细粒度保守式的搜索（❷和❸）。如果平均 NoC 延迟很大或者这个延迟较上一个采样周期的变化不明显，MAW 则会被下调（❷）。另一方面，如果平均 NoC 延迟太小，MAW 则会被上调，以挖掘更多的并发度充分利用带宽资源（❸）。在细粒度搜索阶段，当前一步的调整跨过了最佳点，MAW 也可能会被回滚一步（❹）。搜索过程在拥塞状况进入一个预先设定的理想范围内时停止（❺和❻），算法进入暂时的收敛状态。考虑到程序的阶段性行为特征（phase change），算法在 kernel 结束

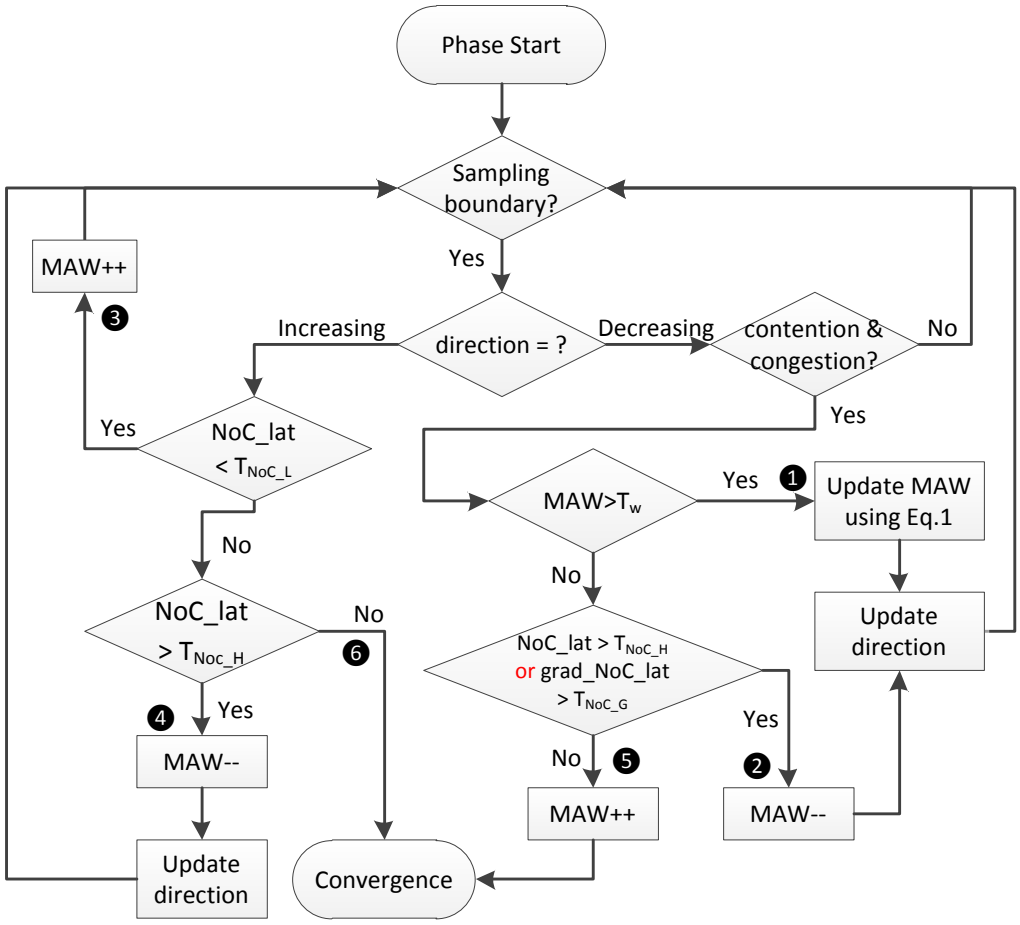

图 5.7　估算最优 MAW 的梯度算法流程图

时或者监测到阶段变迁时，打破收敛状态，将 MAW 重新设为初始的 n_{max}，然后重复上述的搜索过程。通过初始的快速粗粒度 MAW 预测和后续根据反馈信息的渐进式细粒度调整，该算法能够快速有效地找到一个接近最优 MAW 的取值，从而能够获得接近 Best-CBWT 的系统性能。

5.2.3　硬件开销和复杂度

除了能在 CCWS 的基础上进一步提升性能， CBWT 的另一个主要好处就是其较低的硬件开销。在原有硬件的基础上增加的额外硬件不论是存储单元面积还是控制逻辑复杂度都是很低的，因此整体而言，本文提出的存储层次可以很容易地由现在的制造工艺实现出来。旁路策略的开销同 3-bit 动态 PDP cache [13] 是一样的，但是由于是在 L1D cache 中进行实现， PDP 采样的开销比在 LLC 中实现要低很多，而且因为我们选择的是 PDP-S 设计方案，旁路策略只需要一个预测器和采样模块，因此旁路的实现开销几乎可以忽略。对于 CBWT 预测机制，其额外的硬件开销包括三个方面：一是拥塞检测器，即用于计算样本数据包的平均 NoC 延迟的计数器。二是冲突检测器，包括 VB 和两个饱和计数器来累加 LLS 和对 L2

cache 中的样本组的访问总数。假定 VB 的存储开销是由 O_v 来表示，且 L2 cache 有 N 个样本组和 M 个路（ way ），而 P 个 SIMT 核（亦即有 P 个 L1D cache ）。那么存储开销就是 $O_v = P \times N \times M$ 位。对于一个 16 核 GPU ，带有 16 路组相联容量 1MB 的 L2 cache （总共 512 个组），采样比例为 $\frac{1}{64}$ ，则有 8 个组被选为样本组，那么 $O_v = 256B$ 。这相比 CCWS 要少得多，而且它只随着 L2 cache 的扩展而增多。三是预测器，由于搜索算法相对简单，预测器的控制逻辑很容易实现。总的来说， CBWT 的硬件开销要远低于 CCWS ，这是因为 CCWS 没有考虑使用旁路策略，需要依靠精确的局部性流失信息来调整线程束并发度，而 CBWT 使用旁路策略保证了 cache 的性能，其线程束的调整更多地依赖于资源拥塞的信息。

5.3 实验结果

本文在 GPGPU-Sim v3.2.0 [92] 上实现了上述的 CBWT 设计，下面我们按这样的顺序展示实验结果：首先，第5.3.1小节比较了单纯的旁路策略和 CBWT ，说明 WT 技术能够弥补旁路策略的局限性。其次，第5.3.2小节比较了单纯的 WT 技术和 CBWT ，说明启用旁路策略能够帮助 WT 技术进一步提升性能，我们也将 CBWT 同 MRPB [178] 进行了比较，以显示 CBWT 的优势。最后，为了理解不同 cache 容量对 CBWT 有效性的影响，第5.3.3小节给出了敏感性实验的结果。在实验中主要关注的指标包括：系统性能提升、 cache 效率、 DRAM 访问量和效能（ energy efficiency ）。其中系统性能用 IPC （ Instruction Per Cycle ）来衡量，cache 效率则是用失效率来说明， DRAM 访问量即累计发往 DRAM 的请求数目，而效能采用每瓦性能（ performance per watt ）来表征。

5.3.1 对比纯旁路策略

性能。 图 5.8中显示了 PDP-S 、 CBWT 和 Best-CBWT 三种策略下， HCS 负载相对于基准架构的性能（ IPC ）提升。第5.1节介绍过， Best-CBWT 是启用旁路的 SWL 策略。图中可以看出 CBWT 获得的性能明显优于基准架构和 PDP-S 。上一章已经说明了 PDP-S 能够通过缓解冲突提升系统性能，而 CBWT 在旁路的基础上加入了 WT 技术，在监测到冲突和资源拥塞时限制并发活跃线程束的数目，因而进一步缓解了冲突，而且有效控制了 NoC 的拥塞。特别是资源拥塞问题，单纯的 PDP-S 策略无法应对的。 KMN 获得了相比基准架构最大的加速比（ 7.6× ），这是因为 KMN 中存在严重的访存分歧， CBWT 能够缓解大规模访存请求造成的剧烈的 cache 冲突和 NoC 拥塞。相比而言， SWL 和 MRPB 为 KMN 带来的加速则有限得多（分别是 5.6× 和 4.5× ），这是因而这两种策略只考虑了 cache 冲突，无法兼顾资源拥塞的问题。 IIX 获得了仅次于 KMN 的性能提升（ 2.75× ）

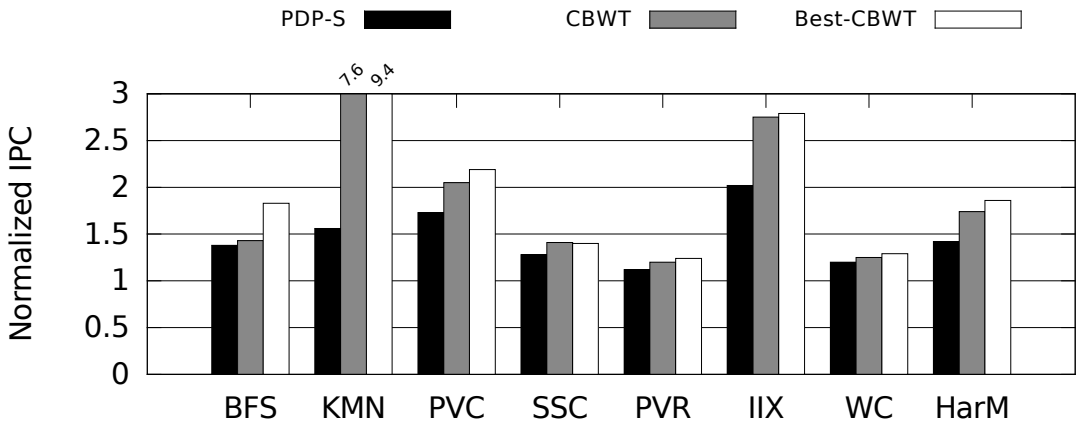

图 5.8　相对基准架构，PDP-S 和 CBWT 策略下 HCS 测试程序的性能提升

，其中主要受益于缓解冲突，而采用 SWL 和 MRPB 策略时，它也获得了明显的加速（分别是 1.91× 和 2.07×）。

同静态的 Best-CBWT 进行对比，能够看出 CBWT 的预测机制是否有效。平均来看，相比 Best-CBWT，CBWT 策略下 HCS 负载仅损失了 8.6% 的性能。对于 SSC，动态 CBWT 甚至比 Best-CBWT 还有略微性能提升，这是因为这个测试程序有比较明显的阶段变迁（phase change）的特性，而静态 CBWT 仅仅支持一个预设的 MAW，因而不能获得最优的性能。但是，CBWT 并不能对所有的负载都获得比静态方法更好的性能，因为预测机制本身有一个由冲突和拥塞监测带来的启动开销。比如，对于 BFS 这种比较小规模的 kernel，CBWT 比 Best-CBWT 的性能要差很多。另一个原因是预测算法也不可能是 100% 准确的，CBWT 可能因为预测偏差而只能获得次优的性能。然而，由于 CBWT 有效控制了冲突和拥塞的程度，总的来说其预测效果还是非常好的，因而获得了非常接近 Best-CBWT 的性能。图 5.9和图 5.10显示了 MCS 和 CI 负载的相对性能。对于其中的大多数测试程序，旁路和 WT 机制没有起到很明显的改进作用，因为这些程序的性能对 cache 行为并不那么敏感。对于所有的程序，尤其是对于 HCS 负载，CBWT 都没有出现明显的性能降级，这保证了 CBWT 能够真正被应用到 GPU 硬件设计中去发挥稳定的效果。

Cache 效率。 如图 5.11所示，PDP-S 和 CBWT 策略为 HCS 测试程序带来明显性能提升的原因是 cache 失效率的大幅降低。在 PDP-S 策略下，HCS 负载的失效率下降的原因是旁路有效地缓解的 cache 冲突。当冲突发生时，缓存数据块会被保护起来，以避免过早移除，因而能够被有效重用。重用次数的上升就带来了较低的失效率。CBWT 在 PDP-S 的基础上进一步地降低了失效率，因为它能够通过限制活跃线程束的数目进一步缓解冲突。特别是在 NoC 拥塞的情况下，CBWT 能够有效缓解 NoC 拥塞，而单纯的旁路策略无法做到。在没有 WT 技术

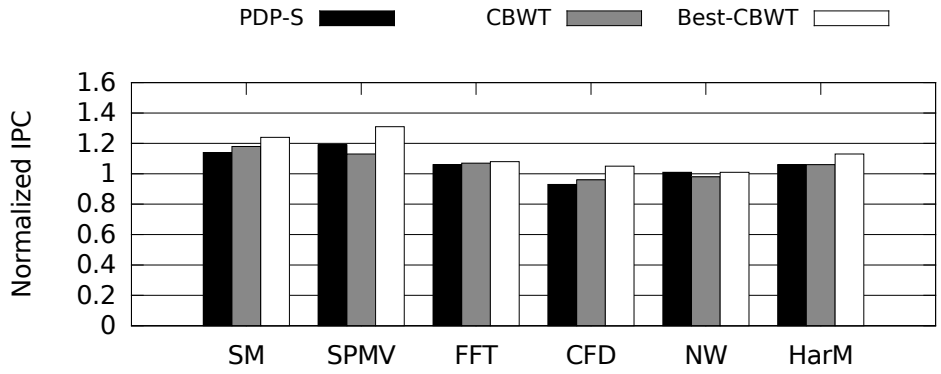

图 5.9　相对基准架构，PDP-S 和 CBWT 策略下 MCS 测试程序的性能提升

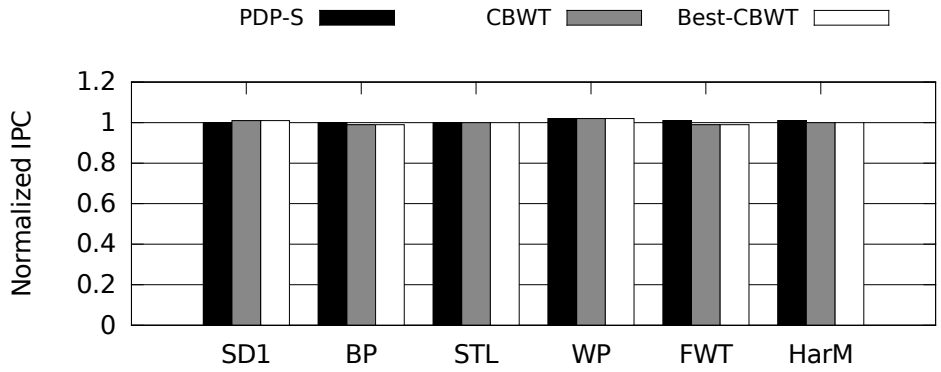

图 5.10　相对基准架构，PDP-S 和 CBWT 策略下 CI 测试程序的性能提升

帮助的情况下，单纯的旁路策略会有大量的请求被旁路，特别是访存分歧严重的负载。这些被旁路的请求中有可能存在重复的请求，也就是存在局部性的流失。而如果引入 WT 技术，通过减少并发线程束的数目，这部分流失的局部性有可能被捕捉到，特别是线程束内的局部性（intra-warp locality），因此 CBWT 能够捕捉更多的局部性。对于 MCS 和 CI 测试程序，如图 5.12和图 5.13所示，不同策略下的失效率则变化不大，这同性能结果是保持一致的。

NoC 延迟。 前面分析了单纯旁路策略无法有效控制访存请求的总量，容易导致 NoC 的拥塞。另一方面，单纯的线程束调节技术能够控制访存请求的发射，但是很难在保证 cache 性能的情况下最大化对 NoC 带宽资源的利用。CBWT 采用旁路策略来保护 cache 资源的免遭剧烈的冲突，同时能够通过调控并发线程束的数目来调整 NoC 的流量，从而在保证 cache 性能的前提下，尽可能地充分利用 NoC 带宽资源。图 5.14中对比了这几种策略下 HCS 负载的 NoC 延迟信息。可以看到，在单纯启用 PDP 旁路策略的情况下，NoC 带宽明显提高，这是因为我们采用的旁路策略将大量 L1D cache 中失效的访存请求通过 NoC 发往 L2 cache，在这些失效访存请求中，被 L1D cache 旁路的请求不会被记录在 MSHR 中，因而避免了 MSHR 成为系统性能平均，但与此同时也造成了 NoC 的拥塞。另一方面，在单纯

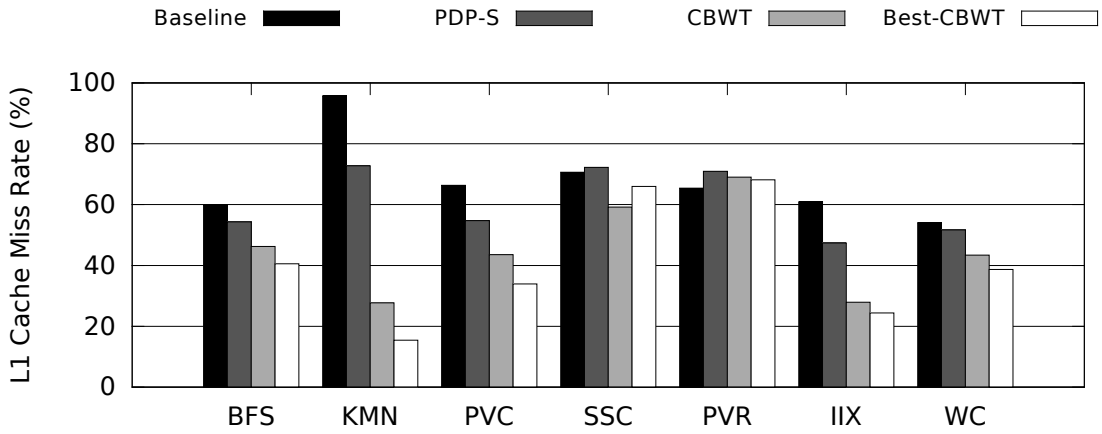

图 5.11　PDP-S 和 CBWT 策略下 HCS 测试程序的 L1D cache 失效率

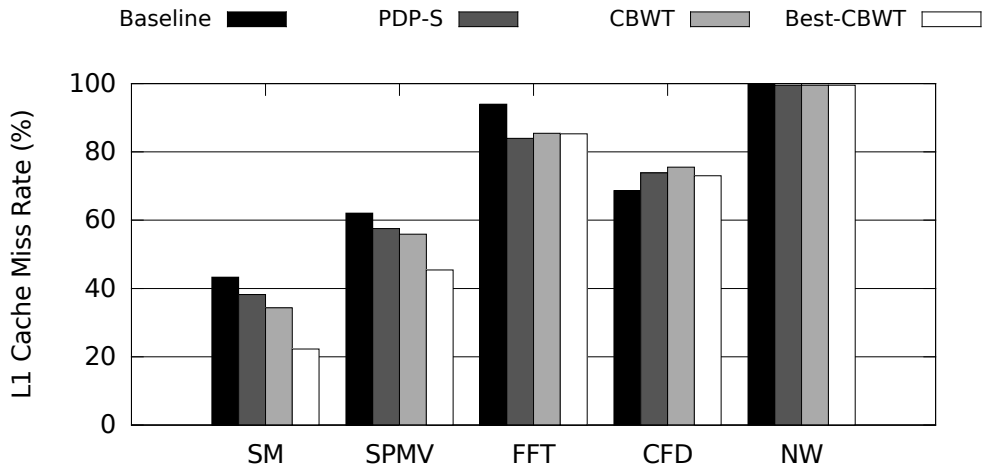

图 5.12　PDP-S 和 CBWT 策略下 MCS 测试程序的 L1D cache 失效率

线程束调节技术 Best-SWL 策略下， NoC 的延迟相对低了很多，这说明 NoC 带宽没有得到充分的利用。相比而言， CBWT 策略下的 NoC 延迟则在二者之中取得了很好地平衡，这也是 CBWT 能够获得比上述两种策略都好的性能的原因。

DRAM 访问量和效能。 图 5.15中显示了各种策略下 HCS 负载中产生的 DRAM 请求的总量。平均来看， PDP-S 相比基准架构减少了 16.5% 的 DRAM 请求总量。而 CBWT 则平均减少了超过一半的片外流量，这是源于旁路策略和线程束调节的综合效果。图 5.16和图 5.17 则分别显示了各种策略下 MCS 和 CI 负载中产生的 DRAM 请求的总量。 DRAM 流量的锐减直接节省了平均 14.4% 的 DRAM 功耗。图 5.18中显示了 HCS 测试程序的系统效能（ Perf/Watt ）。在基准架构的基础上， CBWT 系统效能提升了 58.6% ，而其 DRAM 效能相对于基准架构则是提升了 111% ，即 2.11× Perf/Watt 。对于所有的测试程序，即包括 HCS 、 MCS 和 CI 类型的所有负载， CBWT 相比基准架构也有平均 25.4% 的效能提升，而至于 DRAM 效能， CBWT 则带来了平均 76.7% 的提升。总的来看， CBWT 策略能

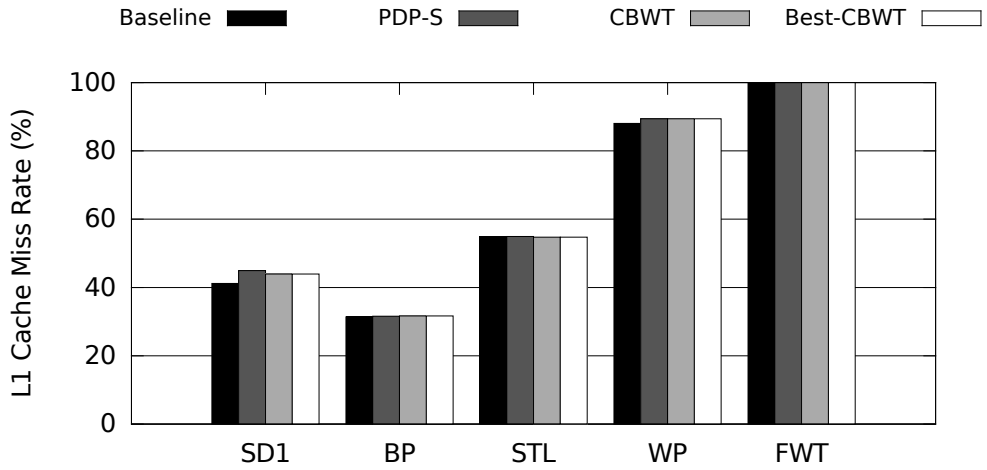

图 5.13　PDP-S 和 CBWT 策略下 CI 测试程序的 L1D cache 失效率

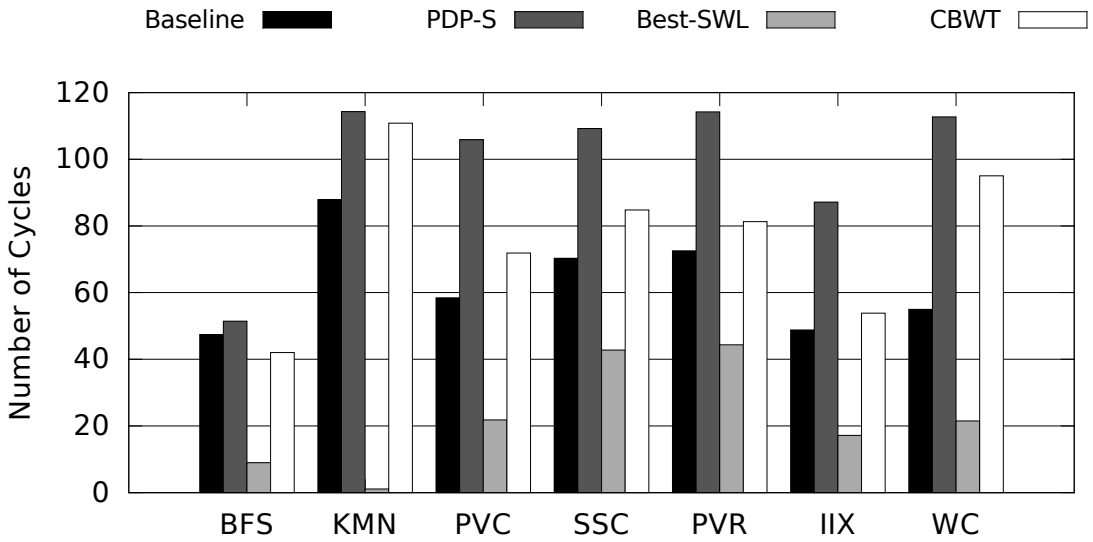

图 5.14　各种策略下 HCS 测试程序的 NoC 延迟

够非常明显地改善 GPU 系统的效能，这对于 HPC 计算系统、大规模服务器系统或者数据中心是至关重要的。

5.3.2　对比其它缓存管理策略

性能。 图 5.19中对比了 SWL-opt(static optimal CCWS)、 MRP 和 CBWT 策略下 HCS 测试程序的相对于基准架构的性能（ IPC ）提升。其中， CBWT （调和）平均获得了 2.09× 的加速效果，明显高于 SWL-opt 和 MRPB 获得的性能提升（分别是 1.72× 和 1.80× ）。相比 SPDP-B ， CBWT 能够控制活跃线程束的数目，因而缩短了重用距离，同时也缓解了 NoC 拥塞。特别是对于 KMN 这种重用距离很长的应用，其生成的大量访存请求造成严重的网络拥塞， WT 技术是很有效的

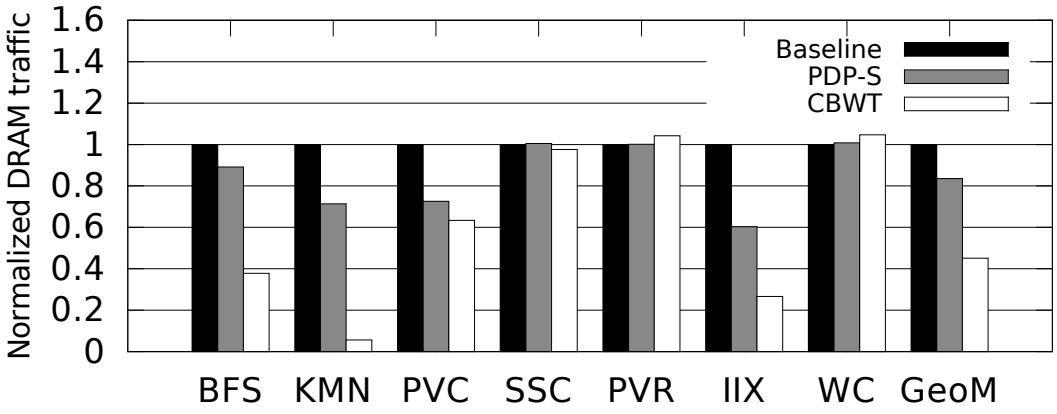

图 5.15　HCS 测试程序的 DRAM 访问量，归一化到基准架构

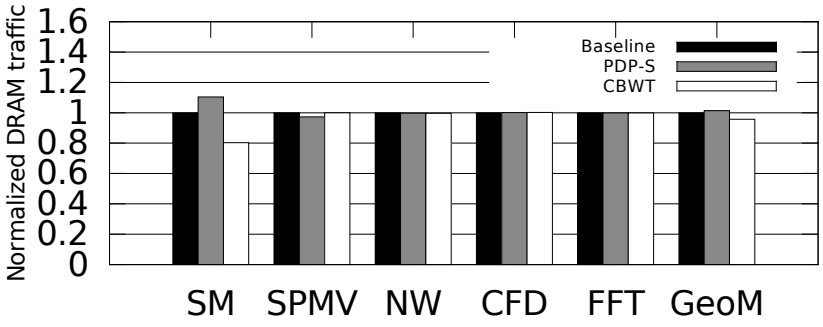

图 5.16　MCS 测试程序的 DRAM 访问量，归一化到基准架构

。相比 CCWS，由于 CBWT 允许更多的活跃线程束并发执行，更高的线程并发度不仅意味着计算的加速，而且能够更有效地隐藏访存延迟。CBWT 中额外增多的线程束不会造成 cache 性能的明显降级，却能够更好地利用空闲的 NoC 带宽以尽早发送访存请求，使之尽早得到服务，因而减少了访存暂停（memory stall）。MRPB 采用请求缓冲（request buffer）来调整访存请求发往 L1D cache 的顺序，以提高线程束内的局部性（intra-warp locality）。但是，同 SPDP-B 类似，MRPB 无法控制并发度，对于访存分歧特别严重的应用，其局限性比较明显。CBWT 能够动态掌握 NoC 的使用情况，在监测到拥塞时挂起部分线程束，因此对于 KMN 和 IIX 这样产生了大规模访存请求致使网络严重拥塞的负载，其性能提升明显较高。

线程束调度器的影响。基准架构采用 LRR（Loose Round Robin）[17] 线程束调度策略，相比其它先进的调度策略，该策略没有考虑线程束的访存行为，性能较差 [17, 80]。为了说明线程束调度策略对 CBWT 的影响，本小节采用 GTO（Greedy Then Old）调度策略重新对 SWL、MRPB 和 CBWT 进行测试。GTO 策略让一个线程束尽可能地执行直到其暂停（stall）然后从就绪（ready）的线程

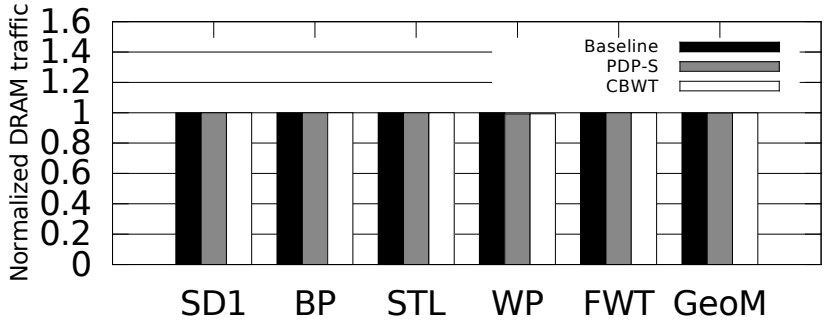

图 5.17　CI 测试程序的 DRAM 访问量，归一化到基准架构

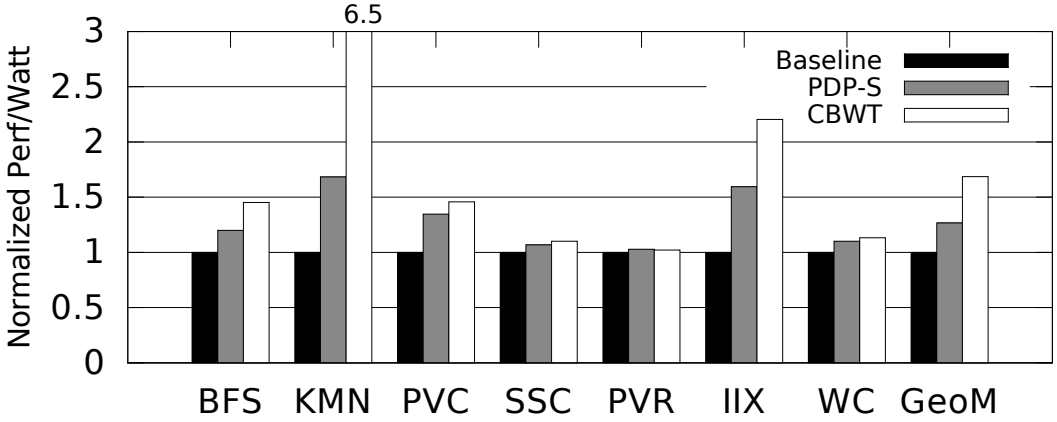

图 5.18　HCS 测试程序的效能，归一化到基准架构

束中选取最早（oldest）的执行。图 5.20中显示了采用 GTO 调度策略时，对于 HCS 测试程序，Best-SWL、MRPB 和 CBWT 三种 cache 管理策略的性能。可以明显看出，除了 SSC 之外，CBWT 的性能都优于另外两种策略。其中，IIX、KMN 和 PVC 的性能优势尤为明显。图 5.21中可以看出，无论采用何种调度策略，CBWT 的性能都要明显优于 Best-SWL 和 MRPB。图中虽然没有显示，但值得一提的是，当归一化到 SWL 时，MRPB 的性能基本上保持不变，而 CBWT 在分别使用 LRR 和 GTO 策略的情况下则获得了 17.3% 和 37.8% 性能提升。也就是说，CBWT 在 GTO 策略下获得了更多的性能提升，这是因为由于 GTO 考虑了维护线程束的时间局部性，致使 SWL 和 MRPB 的性能收益有一定程度的缩减（SWL 和 MRPB 都是注重考虑改善 cache 的性能）。但是 CBWT 不仅仅考虑 cache 的性能，还考虑了 NoC 拥塞，这在即使采用 GTO 来改善 cache 行为的情况下仍然是明显有效的。

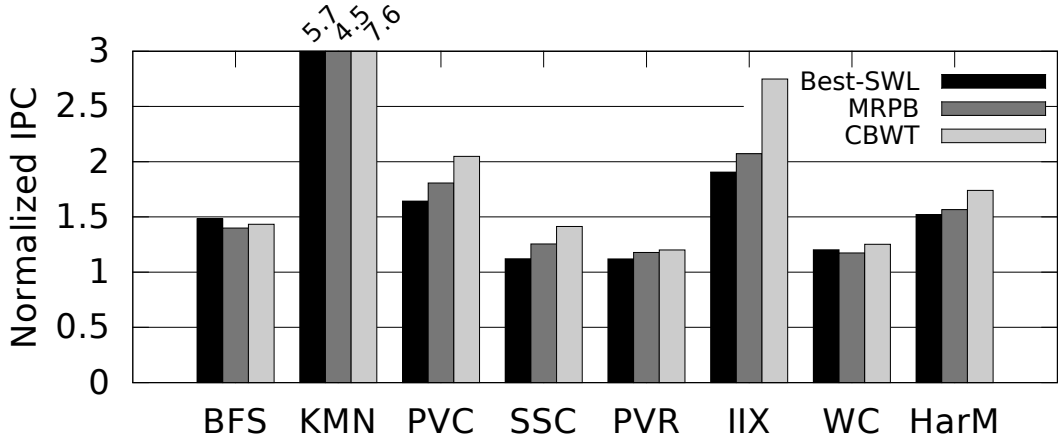

图 5.19　SWL-opt、 MRPB 和 CBWT 策略下 HCS 测试程序的性能提升

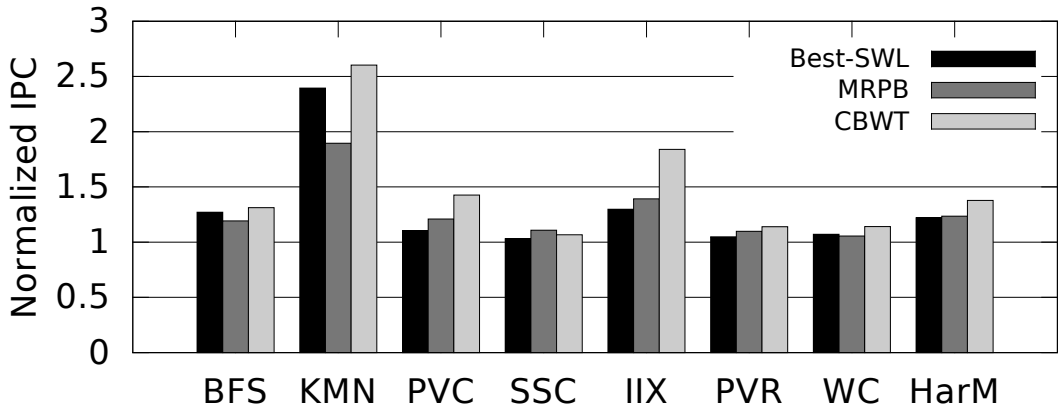

图 5.20　GTO 调度算法下 SWL-opt、 MRPB 和 CBWT 对 HCS 测试程序的性能提升

5.3.3　缓存容量敏感性分析

硬件厂商在未来有可能要增大 cache 的容量，以满足更多潜在应用的性能需求。本小节采用 64KB 的 L1D cache 情况来重新评估 CBWT 的设计，以说明增大 cache 容量对 CBWT 设计的影响。图 5.22中显示的是 PDP-S 和 CBWT 策略下的 IPC，两种策略下都是每个 SIMT 核带有 64KB 的 L1D cache， IPC 都归一化到采用 64KB L1D cache 的基准架构。平均来看， PDP-S 和 CBWT 都显著提升了性能，分别达到 41.3% 和 51.9% 。这就意味着即使未来的芯片将增大 L1D cache 的容量，应用旁路和 CBWT 策略仍然能够有效缓解 cache 冲突和资源拥塞。但是，对于部分测试程序， CBWT 相对于 PDP-S 的性能提升有一定的缩减，这是因为容量较大的 cache 能够吸收更多的请求并减少很多 NoC 的流量负担。然而，对于某些程序，比如 KMN， CBWT 相比 PDP-S 仍然获得了非常明显的加速，这是由于 KMN 产生的请求总量太大，以至于在 L1D cache 容量翻倍的情况下仍然存在

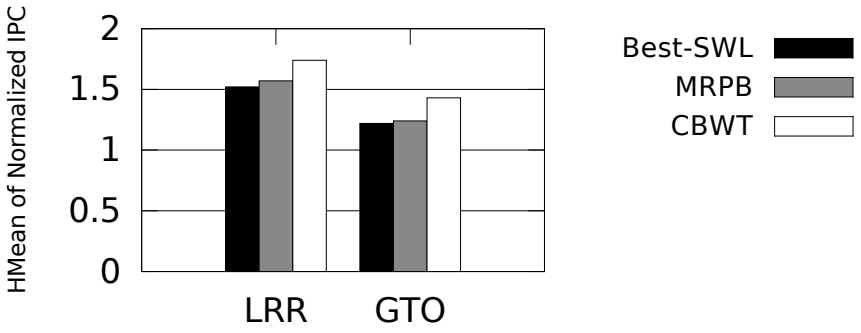

图 5.21　对比不同调度算法下各个策略对 HCS 测试程序的平均性能提升

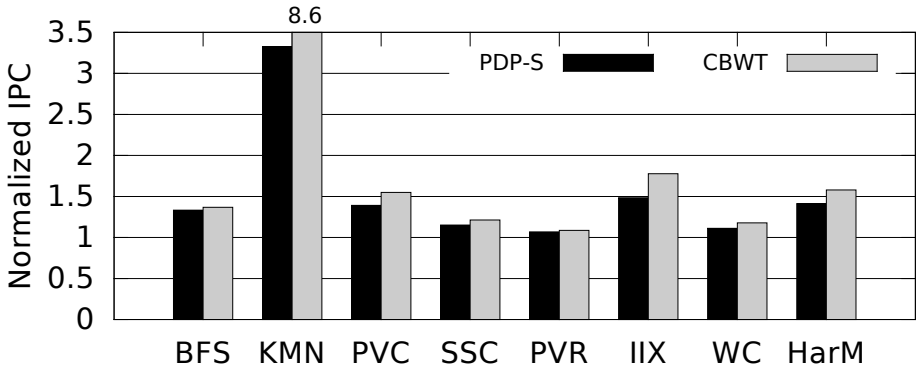

图 5.22　相比基准架构，PDP-S 和 CBWT 策略下 HCS 测试程序的性能提升。其中基准架构、PDP-S 和 CBWT 中的 SIMT 核都采用 64KB 的 L1D cache 。

严重的网络拥塞。对于存在严重访存分歧的程序，这种情况并不少见，因而在增大 cache 容量的 GPU 中采用 CBWT 仍然是有必要的。

5.3.4　MSHR 敏感性分析

我们还通过实验测试了 CBWT 对于 MSHR 数目的敏感性。实验中我们将 MSHR 的数目从 32 增大到 64 和 128 。图 5.23中显示了 HCS 基准测试程序在不同 MSHR 数目下的性能。我们知道，将 MSHR 数目增大到 128 会增加 NoC 的延迟，因为较大的 MSHR 数目意味着更多的访存请求可以被发射到网络中。但是，图中可以看出，CBWT 的性能并没有表现出对 MSHR 数目的敏感性：相对基准架构，HCS 程序在 CBWT 下的平均 IPC 从 1.74 变为 1.70 。这种不敏感性是因为 CBWT 能够很好地调控 NoC 流量，从而保证了在不同 MSHR 数目的情况下都能获得稳定的性能收益。这种稳定性是其它策略无法获得的，因为无论是 CCWS 还是 MRPB ，它们都是单纯以 cache 的性能为优化目标，无法有效地调整 NoC 的使用情况，因而在访存分歧发生时，无法同时兼顾 cache 的冲突问题和其它资

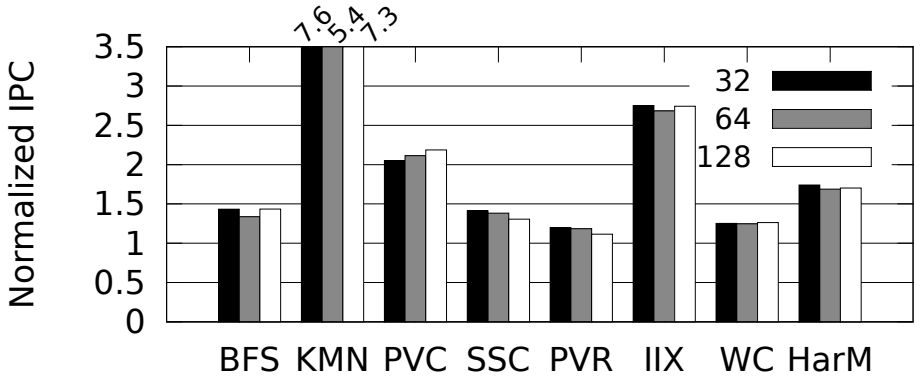

图 5.23　相比基准架构，CBWT 策略下 HCS 测试程序的性能提升。每个 SIMT 核中的 MSHR 数目从 32 增大到 64 乃至 128 。

源的拥塞问题。在 CCWS 和 MRPB 策略下，对于 HCS 程序，系统很可能受限于 MRPB 的数目或者 NoC 的带宽，因而严重制约性能的效能的提升。

5.3.5　本章小节

本章针对单纯旁路策略和单纯线程束调节机制的局限性，设计了协同旁路与线程束调节（CBWT）策略。CBWT 结合了缓存旁路和线程束调节两种技术，一方面能够缓解 cache 冲突，提升 cache 的性能，另一方面，能够有效控制 NoC 的流量，充分利用了片上资源。CBWT 在运行时监控 cache 的冲突和 NoC 的拥塞情况，采样模块周期性地将这些信息反馈给 cache 控制器，CBWT 在这些反馈信息的基础上通过一个简单的预测模块来估计最优的线程束并发度，从而保证充分利用 cache 的容量，同时有效地控制 NoC 的拥塞，最终获得显著的性能和效能提升。与此同时，CBWT 的硬件开销也得到了很好的控制。

第六章　缓存感知的功耗优化

现代处理器的性能扩展开始严重受限于功耗，使得功耗成为在体系结构设计中与性能同等重要的首要设计约束[26]。另一方面，移动智能终端和大规模数据中心系统成为客户端和服务器端的两种主要计算系统，而二者都对节能有很高的要求。前者需要低功耗以提供长时间的电池续航能力，后者则需要通过低功耗来保证系统的正常散热，同时节省能耗能够降低服务系统的成本。对于 GPU 而言，功耗的问题同样很重要。在智能手机和平板电脑的 SoC 芯片中都集成了 GPU，而大规模的数据中心和超级计算机系统中也广泛采用了 GPU 作为加速器来提高系统的吞吐率和计算峰值。相比 CPU，通常 GPU 的芯片面积更大，使用的晶体管数目更多，其功耗也明显大很多。比如，NVIDIA GTX TITAN GPU 的功耗达到了 250W，而 Intel®Core™ i7-4770 处理器的功耗则是 84W。可见，GPU 功耗在整个计算系统功耗中占得比重也很大。总的来看，优化 GPU 功耗的必要性和重要性不言而喻。

传统的功耗控制策略通过监控系统运行状态，对性能瓶颈进行分析，在程序执行受限于片外存储带宽时应用功耗控制技术（降低处理器核的工作频率）以节省功耗。在 GPU 的大规模并行程序执行环境中，程序的性能不仅会经常受限于 DRAM 带宽，也很可能受限于片上资源。因此，本章首先**通过静态调整实验来说明运用功耗调整技术降低 GPU 系统功耗、提升效能的潜力**。然后考虑缓存等片上资源的使用情况，**提出细粒度的资源监测机制**，来指导系统工作状态的调控策略，已达到动态优化系统运行状态、提升系统效能的目的。

6.1　活跃核调节

芯片功耗分为静态功耗（static power）和动态功耗（dynamic power）两部分。静态功耗又称为漏电功耗（leakage power），可以通过关闭相应模块的电源来控制，即所谓*门控电源* (PG: Power Gating)。动态功耗则是晶体管在输入电压切换时产生的耗电，而所有逻辑功能的 0/1 切换归根结底都是时钟信号的切换，如果时钟信号保持不变，那么动态功耗就为 0，这就是所谓*门控时钟* (CG: Clock Gating)。当然，PG 和 CG 都会使得时钟和电源所控制的模块无法工作。二者的区别在于，CG 的恢复时间较短，而 PG 的恢复时间较长。此外，如果单条指令使用多个模块的功能，在恢复该模块功能的时候，并不是最慢的那个模块的时间，而可能是几个模块时间相加，这就是所谓*上电次序*（Power Sequence）的问题，也就是恢复工作时模块间是有先后次序的，不遵照这个次序，就无法恢复。而遵照这个次序，就会使得总恢复时间很长。

因此，在处理器的后端设计中，为了省电可以关闭一些暂时不会用到的处理器模块[1]，但是也不能轻易关闭，否则一旦需要使用该模块，恢复延迟会导致指令的完成时间明显延长，从而导致系统性能降级。此外，子模块的 CG 和 PG 通常是设计电路时就决定的，对于操作系统是透明的，无法通过软件来优化。公式6.1 [94] 中大致描述了 GPU 功耗模型的各个方面，包括漏电功耗、空闲 SIMT 核和各模块（总共 m 个）的动态功耗。其中，各模块的动态功耗是由活动因子 α_i 乘以各模块的峰值功耗 P_{max_i} 计算得出。

$$\begin{aligned} Power &= P_{leakage} + P_{dynamic} \\ &= P_{leakage} + \sum_{i=1}^{m}(\alpha_i * P_{max_i}) + P_{idle\_core} \end{aligned} \tag{6.1}$$

常见的处理器工作状态包含 S-States、 C-States 和 P-States 三种，其中 S-States （ Sleeping states ）指系统睡眠状态， C-States （ CPU Power states ）指 CPU 电源状态，而 P-States （ CPU Performance states ）则指 CPU 性能状态。当然除了这三种外，还有 G-States （全局状态）和 D-States （设备状态）。

S-States 中的 S0 指非睡眠状态，包含了系统正常运作状态以及待机状态，这意味着只有在 S0 状态下， C-States 才会存在。同样地， C0 代表正常工作状态，而 P-States 正是处理器正常运作时的状态，所以 P-States 只存在于 C0 状态下。简单来说，调整 P-States 会改变处理器核的电压和频率，但处理器仍在运作当中；而 C-States 则是改变处理器各个部分的状态，包括核心、缓存、总线以及各种集成的模块，此时处理器应该是工作或待机状态。操作系统频繁地在这些状态下切换，以达到降低功耗和能耗的目的。

对应 P-States 的功耗控制技术是 DVFS （动态电压频率调整： Dynamic Voltage/Frequency Scaling ），即根据系统的负载情况调节处理器核的电压和频率已达到节能的目的。对应 C-States 的功耗控制技术我们称为 CS （核调整： Core Scaling ）， CS 技术实际上就是对 GPU 进行 SIMT 核级别的 CG 或者 PG 控制。功耗控制的目标主要有两类：在给定功耗预算的前提下，最大化系统性能（程序执行时间或吞吐率等）；在给定性能要求（例如执行时间不大于某个阈值）的前提下，最小化功耗。要注意的是 $Energy = Power \times Time$ ，因而单纯地降低系统功耗并不一定能够降低系统能耗，因为降低功耗的同时可能会导致性能下降，程序执行时间变长。因此本章将要提出的功耗控制机制的目标是在不导致明显性能降级的前提下，最大化系统效能（ Perf/Watt ）。

[1]粗粒度的模块包括处理器核、 NoC、 LLC bank 等，而细粒度的模块可以是 SIMD Lane [94] 或者流水线栈 [186] 等

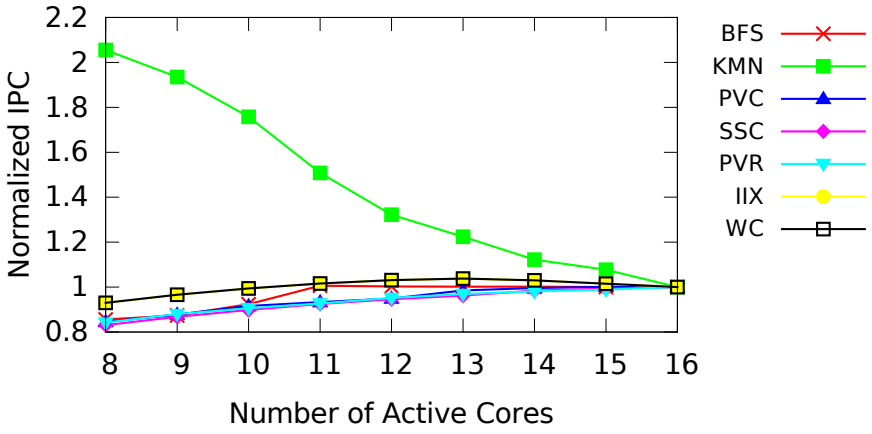

图 6.1　HCS 负载的性能随着活跃核数目变化而变化，归一化到基准架构

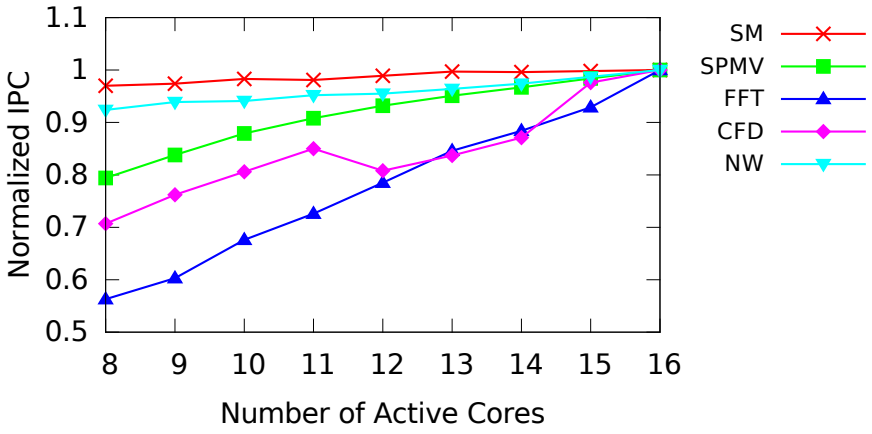

图 6.2　MCS 负载的性能随着活跃核数目变化而变化，归一化到基准架构

本章考虑采用 DVFS 和 CS 技术来优化 GPU 功耗。 Lee 等 [197] 在 GPU 上基于 oracle 研究了 DVFS 和 CS 技术，试图在给定功耗约束下最大化吞吐率，实验数据说明如果合理运用这两种技术， GPU 的吞吐率能够得到有效提升。特别是对于存在阶段性特征的应用程序，如果能够动态改变功耗控制决策，对不同的程序阶段做出不同的响应，系统性能能够进一步提升。本章则以系统效能和低功耗为优化目标，在保证性能的基础上，降低功耗，提高系统效能。要保证系统性能，功耗控制技术需要感知系统资源的使用情况，特别是共享资源，例如 cache 、NoC 和 DRAM 带宽的使用情况，然后根据这些反馈信息来判断程序是否处于资源受限的状态，并据此做出不损害性能的功耗控制决策。

图 6.1、 6.2、 6.3 分别显示了 HCS 、 MCS 和 CI 负载的 IPC 随活跃核数目的变化情况。可以看到，对于 HCS 负载，在一定程度内减少活跃核的数目不会明显降低性能。这是因为这些程序访存非常频繁，其性能明显受限于存储子系统，因而减少计算资源并不会影响性能。其中 KMN 在较少的活跃核数目下获得了更好

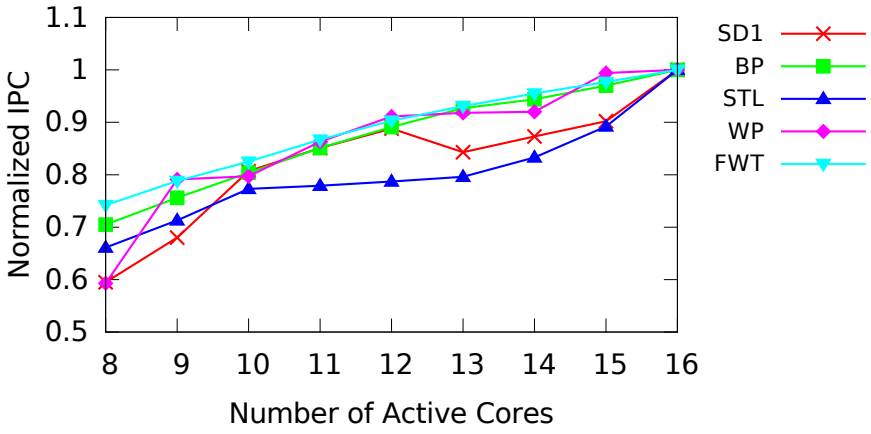

图 6.3　CI 负载的性能随着活跃核数目变化而变化，归一化到基准架构

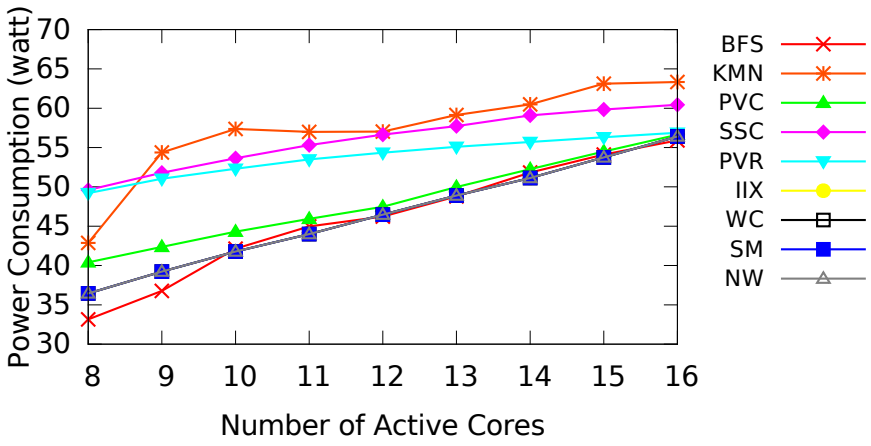

图 6.4　HCS 负载的功耗随着活跃核数目变化而变化，归一化到基准架构

的性能。这是因为较少的并发线程减少了 L2 cache 的冲突以及 NoC 的拥塞，资源受限的问题在 KMN 中尤为突出。 MCS 负载中 SM 和 NW 也展现出了类似 HCS 负载的性能扩展特征。其它测试程序在活跃核数目减少的情况下则表现出明显的性能下降。这些负载的性能对并发度的敏感性很高，因此不适合采取低功耗执行的控制策略。

图 6.4显示了在减少 SIMT 核的情况下， GPU 的功耗也会随着降低。但需要注意的是，由于部分负载的性能在活跃核数目减少的情况下降级了，因此总的能耗（ energy consumption ）不一定会减少。如图 6.5中所示， HCS 负载在活跃核数目减少的情况下的效能变化情况。不难看出，大部分负载在较少活跃核的执行策略下获得了更好的效能。这是因为它们的性能没有受损，但是功耗明显下降了。对于这些访存密集型的应用，我们需要通过对系统的监控获得访存受限等相关的信息，其中比较重要的就是 L2 cache 的冲突情况，因为这些访存密集的程序很可能向 cache 系统发送过多的请求，使得 L2 cache 无法高效地服务这些请求，导致

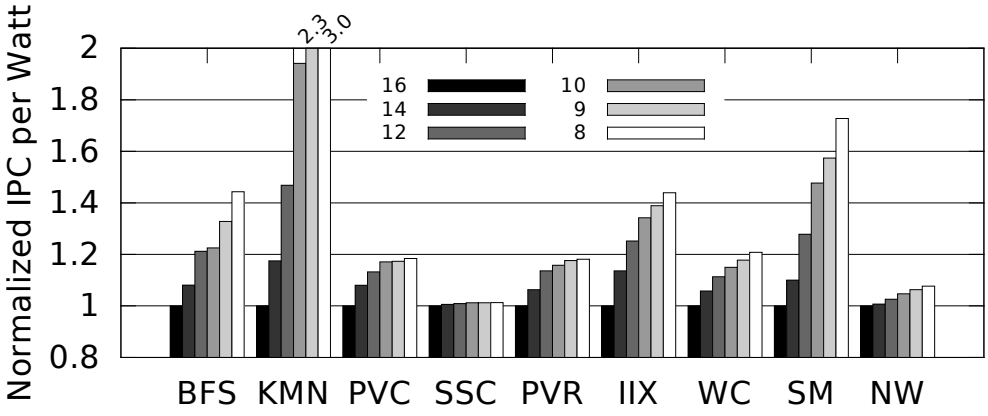

图 6.5　HCS 负载的效能（IPC/Watt）随着活跃核数目变化而变化，归一化到基准架构

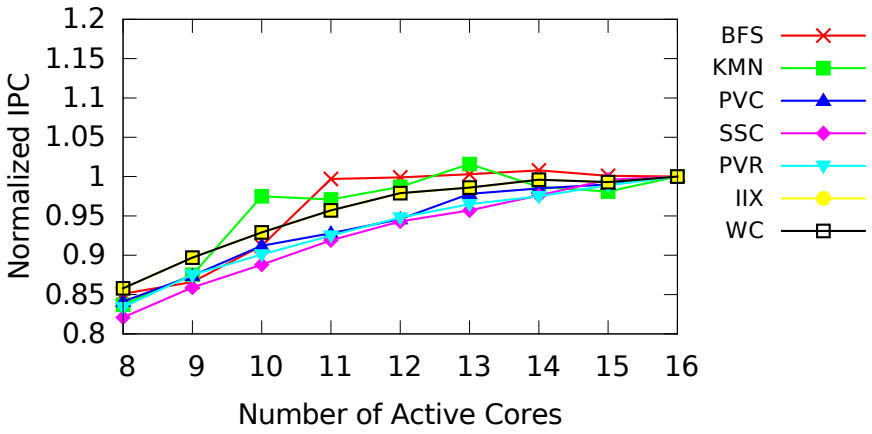

图 6.6　启用 L2 cache 旁路策略时，HCS 负载的性能随着活跃核数目变化而变化，归一化到基准架构

其性能受限于 L2 cache 。这需要对 L2 cache 的冲突情况（即过早移除发生的频度）进行实时监控。当然也存在一部分程序其工作集的大小对 L2 cache 的效率有明显影响，但是其性能并不明显受限于 L2 cache 的效率。这是因为这些程序能够通过大规模多线程有效地隐藏访存延迟，因而其性能对 L2 cache 的效率并不敏感。这也是我们需要获取的信息。根据这些信息进行调控控制活跃核的数目，一方面保证系统性能不会明显下降，另一方面则能够尽可能减少功耗，到达提升效能的目的。

图 6.6显示了启用 L2 cache 旁路策略时，HCS 测试程序的性能随着活跃核数目变化而变化的情况。相比不启用旁路策略的情况，启用旁路之后这些负载在活跃核数目减少时，性能下降的趋势要明显一些。这是因为在启用了旁路策略之后，这些负载的性能不再明显受限于 L2 cache ，而是更多地受限于并行度。减少活跃核的数目降低了程序的并行度，因而使得性能受损。即便如此，所有负载在减

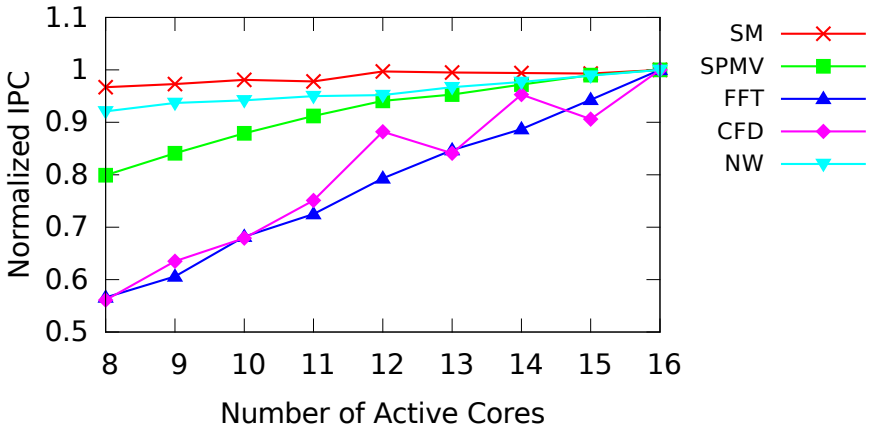

图 6.7　启用 L2 cache 旁路策略时， MCS 负载的性能随着活跃核数目变化而变化，归一化到基准架构

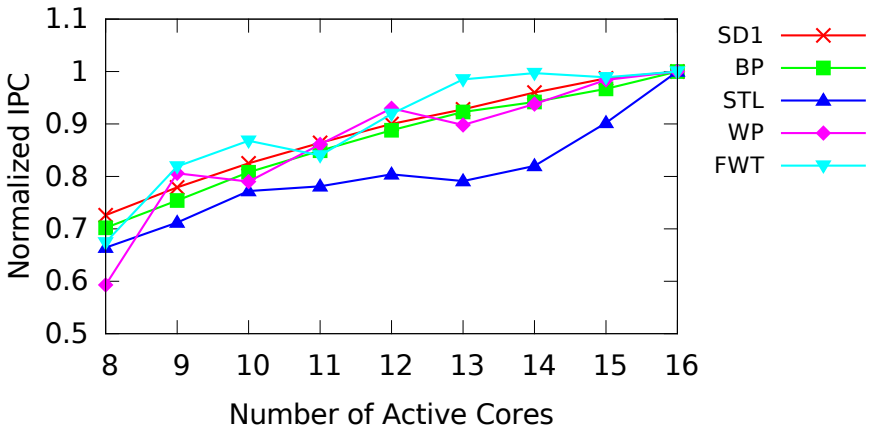

图 6.8　启用 L2 cache 旁路策略时， CI 负载的性能随着活跃核数目变化而变化，归一化到基准架构

少到 11 个活跃核时，都能够保持性能的损耗在 10% 以内。图 6.7和图 6.8则显示了另外两类负载的性能变化情况，这些负载的变化趋势则基本上同前面的情况一致，因为它们的性能大都不受限于 L2 cache 。

图 6.9显示了 HCS 负载在活跃核数目变化时的功耗变化趋势。可以看出，活跃核减少的情况下，功耗下降依然还是很明显的。图 6.10显示了 HCS 负载的效能变化情况。可以看到， BFS 、 KMN 和 SM 等负载存在很大的优化潜力。这说明通过精细的控制，一方面能够有效地控制性能的损耗，另一方面获得明显的效能提升。当然，相比不启用 L2 cache 旁路策略的情况，启用旁路策略之后活跃核调整的效能提升空间变小了。这也是因为旁路策略使得这些负载对 L2 cache 的容量的敏感度减低，因而其性能和效能更多地受限于并发度。因此，在采用了旁路策

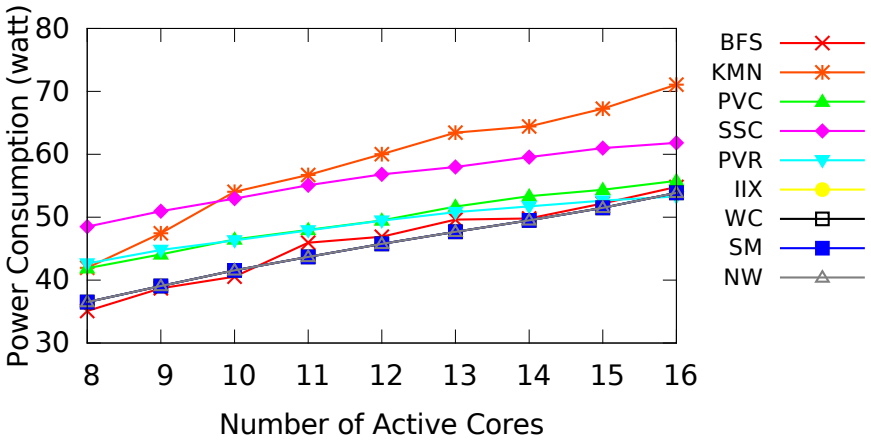

图 6.9　启用 L2 cache 旁路策略时，HCS 负载的功耗随着活跃核数目变化而变化，归一化到基准架构

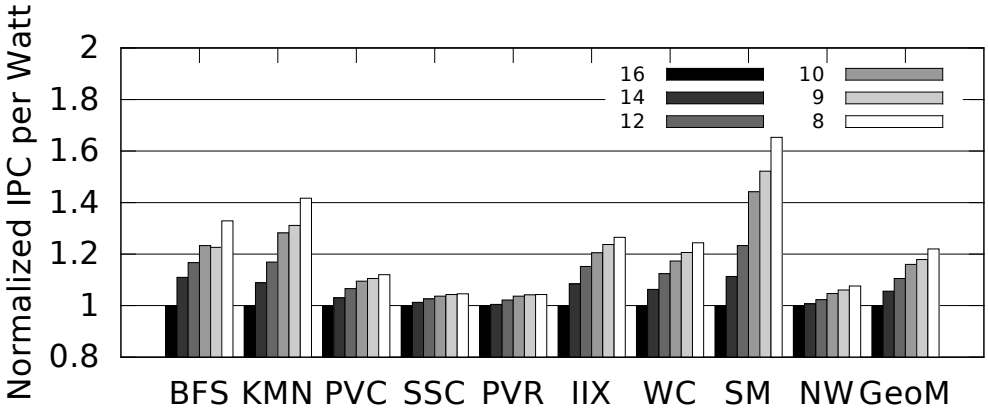

图 6.10　启用 L2 cache 旁路策略时，HCS 负载的效能（IPC/Watt）随着活跃核数目变化而变化，归一化到基准架构

略之后，活跃核的管理策略需要更细致地监控性能的变化，以避免在降低功耗的同时，过分地损耗性能。

6.2　DVFS 调节

DVFS 即动态电压频率调整，它根据处理器所运行的应用程序对计算能力的不同需要，动态调节处理器的运行频率和电压 (对于同一处理器，频率越高，需要的电压也越高) ，从而达到节能的目的。前面我们提到了，降低频率可以减少功耗（power），但是单纯地降低频率并不能节能（energy），而且在现实中的应用程序通常对性能有一定的要求，比如说数据库系统的查询，不满足这个性能要求就无法正常有效地提供服务。因此我们需要在性能和功耗之间做出合理的权衡，在保证一定性能要求的基础上，降低功耗，提升系统效能。

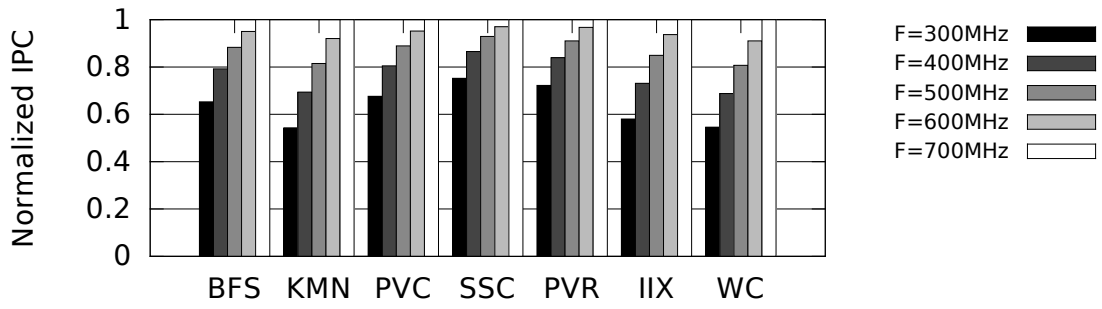

图 6.11　HCS 负载的性能随着 SIMT 核的频率变化而变化，归一化到频率为 700MHz 的基准架构

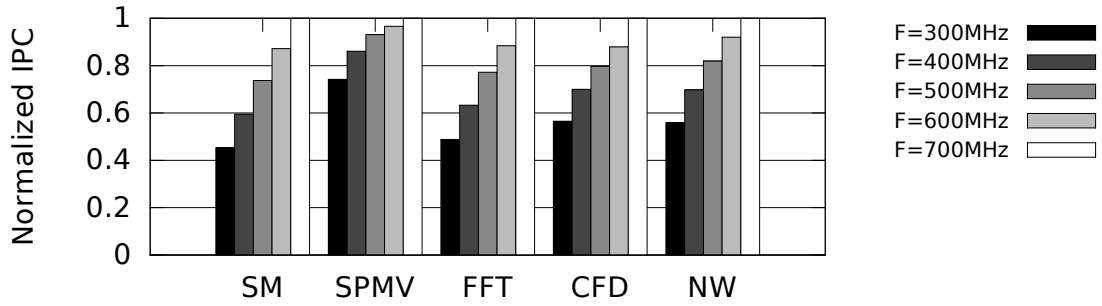

图 6.12　MCS 负载的性能随着 SIMT 核的频率变化而变化，归一化到频率为 700MHz 的基准架构

图 6.11、图 6.12 和图 6.13分别显示了三类负载在不同工作频率下的归一化性能。这里的工作频率指的是 SIMT 核的频率，而内存分区（memory partition），即 L2 cache，和 DRAM 的工作频率是一直保持不变的。可以看出，对于 HCS 程序，由于其性能在一定程度上受限于存储子系统，其性能下降并不是工作频率下降的正比关系。但是，不难看出降低频率对程序的性能影响要大于减少活跃核数目。当活跃核数目减半时，HCS 程序的性能损耗大都在 20% 以内，而工作频率减半则可能带来超过 30%，甚至接近 50% 的性能降级。这是因为 HCS 程序相对来说更多的受限于访存延迟，而不是访存带宽。相反，诸如 FWT 和 STL 等负载，由于其性能受限于存储带宽，因此 SIMT 核的工作频率下降时，性能的损耗并没有减少活跃核时那么明显。

此外，我们观察到在工作频率从 700MHz 降低到 600MHz 的情况下 SSC、PVR 和 SPMV 等负载的性能损耗在 5% 以内。这些应用程序存在较大的潜力受益于 DVFS 调节。当然，除了 L2 cache 和 DRAM 带宽之外，NoC 带宽也可能成为 GPU 应用程序的性能瓶颈。后面我们会详细讨论各个程序的性能瓶颈。从图中还可以看出，采用静态选取工作频率的方法节省功耗或者提升效能的潜力有限。

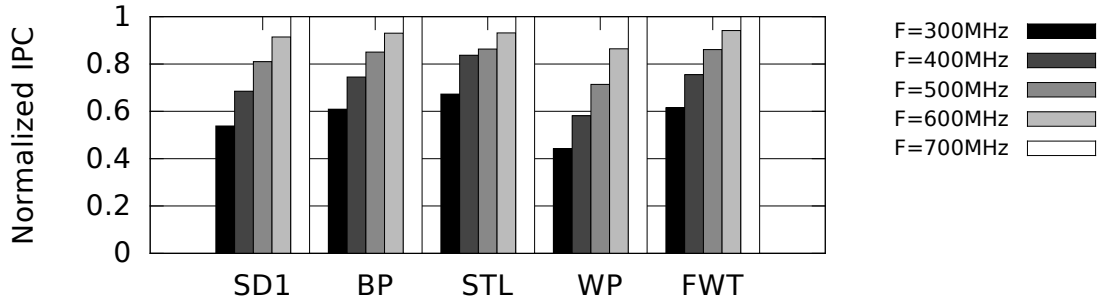

图 6.13　CI 负载的性能随着 SIMT 核的频率变化而变化，归一化到频率为 700MHz 的基准架构

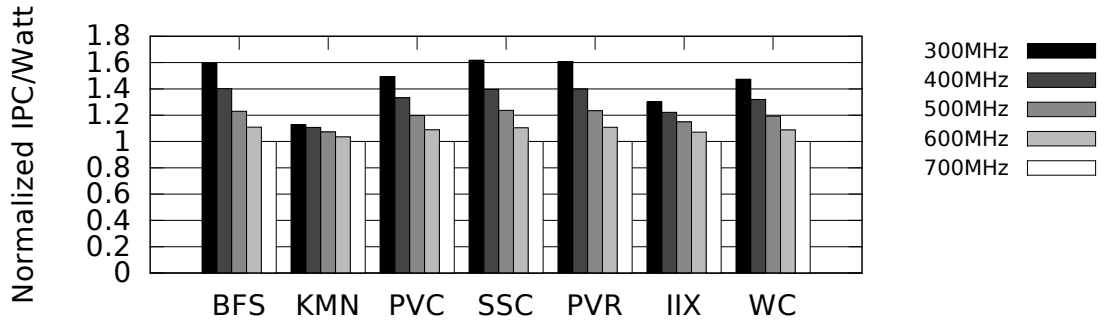

图 6.14　HCS 负载的效能随着 SIMT 核的频率变化而变化，归一化到频率为 700MHz 的基准架构

事实上，很多程序在执行过程中呈现出阶段性变化的特征（phase change property），在受限于存储子系统（memory bound）的执行阶段，选取较低的工作频率能够减低功耗且不明显损耗性能，但是在计算受限（compute bound）的执行阶段，选取较低的工作频率将会大幅度损耗性能。虽然静态降低工作频率对程序的性能损耗很明显，但这并不意味着 DVFS 调节无法得到有效运用。我们只需要在运行时监控程序的执行状态，当发现程序在执行到某一阶段受限于存储子系统资源时，便可以降低 SIMT 核的工作电压 / 频率，从而节省功耗，提升效能。

附录 A中的图 A.1、图 A.2和图 A.3 分别显示了三类负载在不同工作频率下的功耗情况。可以看出，在频率降低的情况下，这些负载的功耗明显下降，因为降低的频率意味着较低的工作电压，而功耗同工作电压的平方成正比。因此，理想情况下，功耗会随着频率的降低而呈指数下降。三类负载对比来看，CI 负载的功耗下降幅度最为明显。而 HCS 负载的下降趋势则相对较缓慢。总的来说，由于功耗随着工作电压的下降速度比性能要快，因此 DVFS 调控能够有效提升系统的效能。

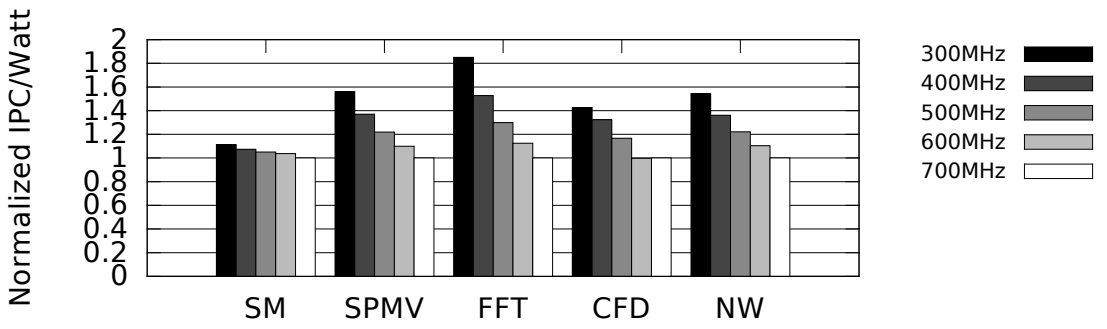

图 6.15　MCS 负载的效能随着 SIMT 核的频率变化而变化，归一化到频率为 700MHz 的基准架构

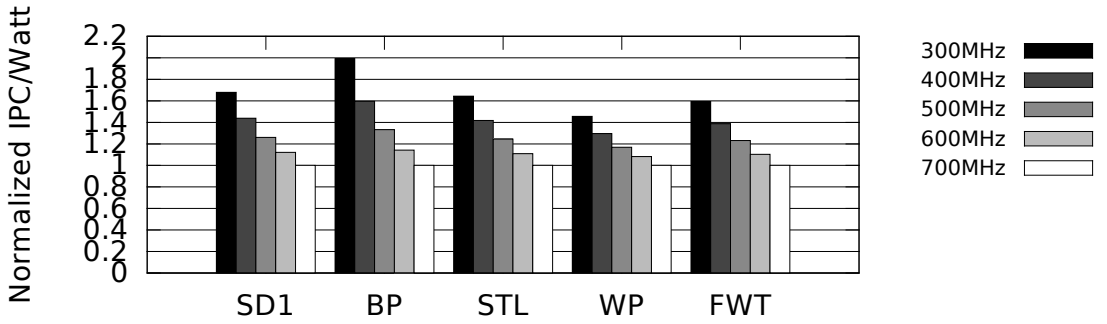

图 6.16　CI 负载的效能随着 SIMT 核的频率变化而变化，归一化到频率为 700MHz 的基准架构

图 6.14、图 6.15 和图 6.16则分别显示了三类负载在不同工作频率下的归一化效能。图中可以明显看出，较低的工作频率通常会有较高的效能，这是因为我们选取的测试程序都是访存受限的，因此提升 SIMT 核的工作频率并不能提升效能，相反，由于 SIMT 核长时间处于等待数据的状态，其工作频率越高，效能反而越低下。因此，对于这些负载，采用 DVFS 提升效能的潜力是很大的，例如 FFT、SD1 和 BP 等负载。当然，过分降低工作频率会导致性能明显下降，因而激进的调控策略存在一定的风险。为了尽可能将性能损耗控制在较低的范围内，我们需要对系统运行状态进行细粒度的监控，有效掌握片上各部分资源的使用情况，从而做出合理的调控决策。

6.3　资源监控与功耗控制

前面的实验可以看出，在 GPU 中运用 CS 和 DVFS 技术进行功耗控制的优化潜力很大，但是要做出合理的功耗控制决策，需要我们对系统运行状态进行细粒度的监控。本章节主要考虑三个方面的资源监控：LLC（在本文中即 L2 cache）、DRAM 带宽和 NoC 带宽。

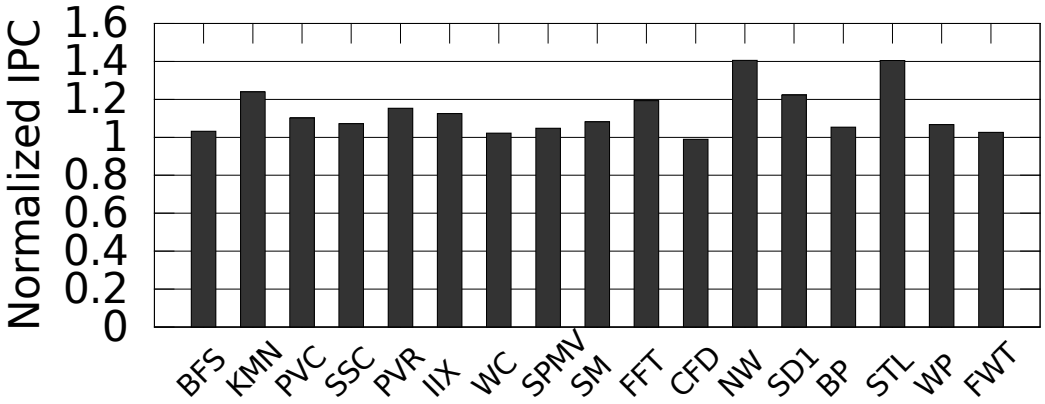

图 6.17　16MB 的 L2 缓存相比 512KB 的 L2 缓存的性能加速比

附录 B中图 B.1和图 B.2 显示了程序执行过程中访存暂停的比例。图中可以看出，不同的应用程序对存储子系统的要求差异很大。其中， HCS 负载的访存暂停比例都很高，这是因为 cache 层次结构无法高效服务这些应用的访存需求，因而出现大量的访存暂停。这在很大程度上损害了系统的性能和效能。而另外一部分程序，例如 FWT 等，则是受限于片外存储带宽。不仅如此， NoC 的带宽也成为制约部分程序性能的主要因素之一。本章节不考虑 L1 cache 对系统性能的影响，因为目前 GPU 中 SIMT 核内部的流水线和 L1 cache 采用相同的频率，因此 DVFS 无法有效利用 L1 cache 的访问行为模式来优化功耗。但是对于 HCS 负载，如果能够分别调控流水线和 L1 cache 的频率，则存在很大的功耗优化空间。

我们还分别评估了这三个方面的性能瓶颈。图 6.17中显示了增大 L2 cache 容量给系统性能带来的影响。部分程序的性能明显受限于 L2 cache 的容量，比如 NW 和 STL 。从图 4.12中我们已经知道， NW 和 STL 的工作集比较大，比如，STL 的工作集是 4MB ，因此在 L2 容量增大到 4MB 时， STL 的性能出现了跃升，这是因为 4MB 的 L2 cache 能够将 STL 的重要数据结构驻留在其中，因而大幅减少了片外访存流量，不但降低了 DRAM 带宽需求，同时由于程序的时间局部性特征，也减少了平均访存延迟。这种改善对于 NW 和 STL 这样的负载非常重要，因为 GPU 的大规模多线程不足以掩盖它们的访存延迟。

HCS 负载对 L2 cache 容量的敏感性反而不那么明显。 HCS 负载之所以对 L1 cache 容量敏感，是因为 SIMT 核上运行的线程的工作集远大于 L1 cache 容量。但是这些负载总的工作集并不大，通常小于 L2 cache 的容量，因而性能也就通常不受限于 L2 cache 的容量。当然，还有一部分程序 L2 cache 的命中率会随着 L2 cache 的容量增大而升高，但是其性能却不会有明显的变化，这是因为对于这些程序， GPU 的大规模多线程能够有效地隐藏访存延迟。

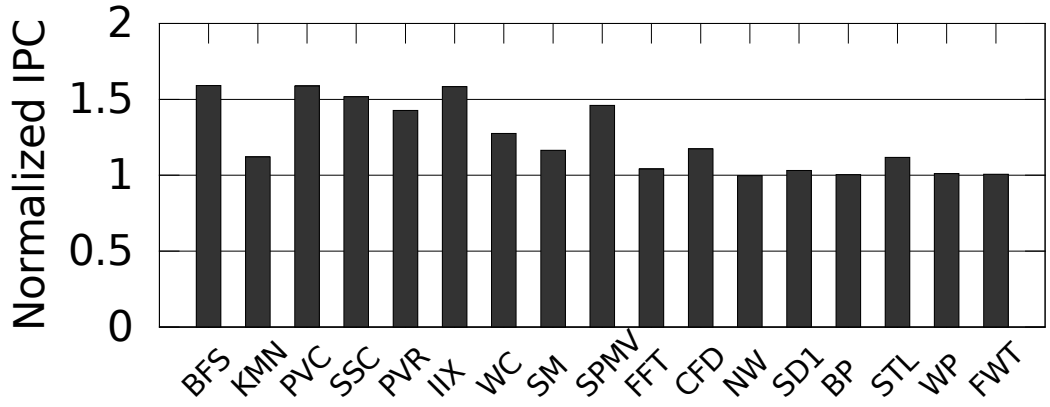

图 6.18　NoC 频率为 2GHz 相比 NoC 频率为 1GHz 的程序性能加速比

图 6.18和图 6.19 分别显示了 NoC 和 DRAM 带宽对程序性能的影响。可以看出，带宽对这些防存密集的应用程序的性能影响很大。在增大带宽的同时，很多程序的性能有明显的上升。另一个明显的观察结果是 HCS 负载大都受限于 NoC 带宽，而其它负载则更多地受限于 DRAM 带宽。由于 HCS 负载存在剧烈的 L1 cache 冲突，因此在 L1 和 L2 cache 之间存在大量的数据传输，导致其性能受限于 NoC 的带宽。而其它程序则或者存在 L2 cache 冲突问题，比如 NW 和 STL ，或者其计算 / 访存的比例太低，因而对 DRAM 带宽的需求更为迫切。总的来说，对 NoC 和 DRAM 带宽的使用情况进行动态监控，能够更好地理解应用程序对于除了 cache 之外的系统资源的需求，从而针对性地实施 CS 和 DVFS 调控策略。

为了动态地做出合理地功耗控制决策，我们提出细粒度的系统资源监控策略。对于 cache 的访问行为监控，我们采用同 PDP cache [13] 类似的策略，即对被指定为采样的组增加请求 FIFO ，以记录 cache 访问的序列。通过统计 LLS 来获取 cache 中局部性流失的程度。另外，可以增加相应地计数器来统计 cache 块和 MSHR 导致的访存暂停。当然，由于大规模多线程隐藏访存延迟的特性，导致 L2 cache 的命中率同系统的性能之间不一定存在直接关联，因此即便 L2 cache 出现严重的局部性流失，也不能说明 L2 cache 就是当前系统的性能瓶颈。因此我们还需要通过监测系统性能来区分不同的程序执行特征。对于 NoC 带宽使用情况的监测，我们可以采用前述采样 NoC 传输延迟的方法，也可以用更为精确地 NoC 性能监控技术，比如监控网络中 buffer 的占用情况等。 DRAM 带宽则使用类似的计数器来记录带宽利用率以及带宽受限导致的暂停。基于上述监控技术，我们可以获得详细的系统资源使用情况，对系统性能瓶颈做出全局性的估计，从而制定出合理高效的功耗控制决策。

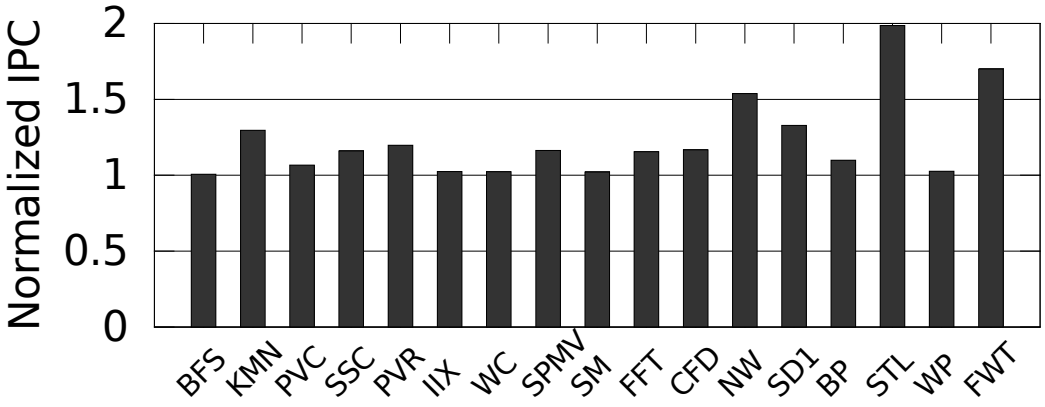

图 6.19　DRAM 频率为 1.6GHz 相比 DRAM 频率为 0.6GHz 的程序性能加速比

6.4　本章小结

本章主要讨论在 GPU 上进行功耗优化以提升系统效能，主要涉及的功耗控制技术是 CS 和 DVFS 。我们首先通过静态实验说明运用这两种技术来调控系统功耗存在很大的效能优化潜力。另一方面，如果没有合理的控制策略，减少活跃 SIMT 核的数目或者降 SIMT 核的工作频率会导致性能的大幅降级，从而不可能在实际系统中真正得到应用。为此，本章研究了 cache 等存储子系统中的重要资源对系统性能的影响，为合理地进行动态功耗控制，还提出细粒度的系统资源监控策略。

第七章　总结

经过几十年的发展，摩尔定律仍然引领着处理器产业的发展。工艺的不断进步使得芯片的设计越来越趋向复杂。为了在工艺进步的同时向用户提供持续提升的性能和效能，处理器体系结构开始朝着异构并行体系结构的方向发展。并行处理器要求程序员编写并行程序来挖掘线程级并行，在提高了编程难度的同时，对存储带宽和功耗的要求也越来越高。异构处理器通过融合针对不同类型应用的处理引擎来提升效能。 GPU 就是一种针对数据并行应用的众核加速器处理引擎。GPU 集成大量简单地处理器核构成大规模 SIMT 并行系统，通过提供这种大规模的计算能力和存储带宽来高效执行数据并行应用程序。同时， GPU 通过大规模多线程来隐藏访存延迟。这种解决方案能够有效处理控制和访存规则的应用程序，但是当运行不规则的应用程序时， GPU 的效率低下，这使得在真正将 GPU 推广到通用计算领域时变得不那么实用。

cache 层次结构是 NVIDIA Fermi 体系结构开始被引入到 GPU 中的一个重要体系结构革新。虽然 cache 在 CPU 中已经被采纳和优化了很长的时间，但是在 GPU 中得到应用还存在很多工程和研究问题亟待解决。 GPU 的大规模并行计算模式对存储子系统提出了更高的要求。不规则应用出现访存分歧问题时，会产生大量的访存请求导致存储子系统的低效。传统的 cache 层次结构设计无法有效服务这种大规模访存模式，不但无法发挥出 cache 系统降低延迟、增大带宽的好处，反而会导致系统性能和效能的下降。本文针对 GPU 中的 cache 层次结构低效的问题，着眼于存储墙、编程墙和功耗墙三个方面，提出针对大规模并行众核加速器的 cache 管理策略。

7.1　本文的主要贡献

本文首先对 GPU 上的大规模多线程应用程序进行了详细的评测和分析，实验说明了 GPU 的 cache 层次结构无法高效服务不规则应用程序的访存需求。其主要原因是大规模并行应用出现访存分歧时，大量的访存请求造成了 cache 污染和 cache 冲突，导致 cache 的性能很差。然后，本文将现有最先进的 CPU 缓存替换和旁路策略在 GPU 中进行了实现和评估。实验表明对于 cache 敏感型应用可以通过改进 cache 管理策略提升 GPU 系统性能，但另一方面，将现有的 CPU 缓存替换和旁路策略直接应用到 GPU 中仍然存在局限性。本文还结合了现有的防污染和防冲突管理策略，提出了针对 GPU 的自适应缓存管理策略，实验结果表明该策略能够进一步提升系统性能。

针对单纯缓存旁路策略和单纯线程束调节技术的局限性，本文提出了针对GPU的自适应缓存管理策略。由大规模多线程引发的cache冲突和片上资源拥塞在运行时被动态监测，片上资源的使用情况作为反馈信息指导管理策略。据此缓存旁路策略协同线程束调节技术（CBWT）能够充分利用缓存容量和片上资源。本文还为协同旁路和线程束调节设计了一个简单的动态预测器，能够在运行时估计最优的活跃线程束数目。实验表明，CBWT能够有效提升cache的效率，同时合理充分地利用其它片上资源，以达到最佳的系统执行效率。相比基准GPU架构和最优静态线程束调节，该方法能够平均分别提升GPU系统性能1.74倍和1.17倍。

本文针对处理器设计中的功耗墙问题，提出了GPGPU中缓存感知的功耗管理机制。GPU的大规模执行模式导致访存密集的应用程序往往受限于存储子系统。当系统性能受限于访存时，计算部件处于低效能的执行状态。本文针对访存受限的应用程序，考虑在保证性能不明显降级的前提下，通过在运行时使用核调整技术CS和动态电压频率调整技术DVFS，节省系统功耗，提升系统效能。为了保证对系统性能的有效控制，本文提出细粒度资源监控机制，对存储子系统中的缓存和（NoC、DRAM）带宽资源使用情况在运行时通过采样进行细致的监控。在此基础上，本文用实验说明了通过运用核调整技术来减少活跃核的数目，以及通过DVFS技术降低工作电压/频率，能够有效地节省功耗、提升系统效能。

7.2 未来工作

本文针对GPU中cache层次结构效率低下的问题进行了深入的研究。在此基础上，我们还有两个方面的工作可以延伸开展。第一，本文的研究工作主要集中在L1缓存的管理。虽然涉及到了L2缓存，但是并没有深入研究。在GPU中，相比L1缓存，L2缓存的延迟大很多，而且相比CPU的LLC，这种延迟的量级也是非常大的。因此L2缓存的管理策略对性能的影响没有L1缓存那么显著。但是L2缓存的性能对片外访存总量的影响很大，由于DRAM的能耗在系统中占很大的比重，提升L2缓存的效率对系统的效能提升有非常明显的影响。我们可以延续在L1缓存上的研究思路，也就是说，一方面通过改进L2缓存控制器的管理策略（例如替换和旁路）来提升L2缓存的效率，另一方面考虑其它资源的使用情况，进行协同的全局优化管理。这将给管理策略进一步复杂度，因此我们还需要考虑如何控制开销，在收益和成本之间作出合理的权衡。

第二，功耗优化需要进一步的实验验证。本文提出的功耗优化技术通过静态实验说明了CS和DVFS的优化潜力。同时，本文还通过实验对应用程序的性能瓶颈做了详细的分析，并提出了细粒度的监控策略。下一步工作需要在监控策略

的基础上，实现 CS 和 DVFS 的动态控制。功耗的动态调控策略需要应对各种不同的应用行为，根据不同的程序运行时行为作出不同的应对调整。比如说，如果程序在运行时表现出稳定的阶段性特征，比如计算受限和访存受限的执行阶段交替出现，且每个执行阶段的持续时间足够长，调控策略则可以简单地针对不同的执行阶段作出相应地系统工作状态调控，比如在访存受限时减少活跃核数目或者降低 SIMT 核的工作频率。但是对于不稳定的行为模式，我们需要尽可能减少调控策略的变化，维护策略的鲁棒性，避免出现大幅度的性能降级。

7.3 结束语

现代 GPU 架构擅长加速规则或访存行为静态可预测的应用程序。然而，对于访存不规则的通用负载， GPU 性能则高度受限于存储子系统，使得 GPU 的效能甚至低于 CPU 。要真正让 GPU 架构的在通用计算领域有用武之地，需要充分利用好 GPU 芯片上的资源，并使得存储子系统（而不仅仅是计算单元）适应不规则的应用程序行为。本文的解决方案是将缓存和其他片上资源利用情况考虑在内，并协调运行时信息利用好这些资源。

本研究的长远目标是使新兴应用能够在大规模并行处理器（比如 GPU ）上高效运行。运行在大规模数据中心系统，以及服务器，台式机，甚至移动智能终端上的许多重要的商业应用程序都表现出不规则的访存行为。由于存储墙问题将持续存在，处理器体系结构设计者仍然需要精心设计内存子系统以满足新兴应用的需求。本文中用到的缓存敏感的基准测试程序包括现代数据中心系统上的 Map-Reduce 应用，以及在许多网络服务器系统中常用的 graph 处理和数据挖掘应用。对片上高速缓存层次结构的自适应管理可以显著提高这些应用的性能，这使得大规模并行处理器在未来的市场上更具有吸引力和竞争力。本研究也是最先将目前最先进的 CPU 缓存管理策略移植到 GPU 中，其实验结果为工业界决定是否值得将这些策略在真实硬件中实现提供了参考。此外，本文中提出的 CBWT 方法引入很少的硬件改动，这使得这种动态的缓存管理方案的硬件支持更容易被工业界所接受。

为了提高效能，研究人员提出了很多针对特定应用或者计算类型进行专门化或定制的处理引擎。而本研究的主要想法则是为 GPU 这种为数据并行应用定制的处理引擎来定制缓存层次结构。本研究通过实验说明了定制存储子系统来提高系统效能的重要性。由于存储墙问题在可预见的未来将持续存在，这种定制存储子系统的研究方向变得非常重要。本研究还从存储子系统的角度给出了如何在通用计算和定制计算之间做出合理权衡的经验。将缓存旁路和线程束调节相结合的方法为研究者们在全局范围进行资源管理从而获得系统性能上升提供了一个新的

视角。本文提出的方法说明了结合不同的管理策略能够克服单独使用其中任何一种的局限性。由于 GPU 的大规模执行环境，在 GPU 中进行协同的资源管理比在 CPU 中要重要很多，也要困难很多。本研究提出的硬件解决方案能够有效应对输入数据相关（input dependent）的访存行为以及程序的阶段性变化特性（phase change property）。此外还存在同系统软件（编译器和运行时系统）进行协作的可能性，以此来进一步提升系统性能、效能和可编程性。因此，针对未来 GPU 和 HSA 系统的体系架构和编译器协同设计（architecture-compiler co-design）的研究也有很大潜力。

致　谢

衷心感谢导师王志英教授。自 2007 年加入师门以来的九年时光里，王老师在学习、科研和生活各方面一直给予我悉心的指导和真诚的关怀。王老师不但为我的研究工作指明了方向，而且创造了良好的工作环境，让我能够集中精力攻坚克难。生活中，王老师待人宽厚、积极乐观，是真正为人师表的典范。王老师多年来对我的言传身教将使我终生受益。

衷心感谢 Wen-Mei Hwu 教授。在 UIUC 的两年时间里， Hwu 老师悉心指导我的研究工作，给予了我很多中肯的建议。 Hwu 老师学识渊博，治学严谨，对学术问题的细节刨根问底，让我明白如何扎实严谨地开展学术研究。 Hwu 老师在课堂上和学术讲座上大方稳健、谈笑风生、游刃有余的风格也给我留下了很深的印象。

衷心感谢沈立老师。沈老师在我学术道路的起步过程中给予了我很多引导，让我最初开始认识到了科学研究的魅力。沈老师在科研项目中对我细心指导，不但让我提升了科研能力，更重要的是培养了迎难而上的科研素养。沈老师乐观开朗、待人热诚，也是生活中的良师益友。

感谢在科研上给予我帮助的老师和同学。感谢陈微师姐给了我在科研上的启蒙指导，师姐不但在学术上不厌其烦地指导我，在生活上也常常给予帮助。感谢黄立波师兄对我在学术上的启发和帮助，也让我从他身上学习到很多成功的经验。感谢陆鸿毅老师在我遇到技术难题的时候总是能够在百忙之中抽出时间帮忙解决。感谢郑重、苏博、徐帆、朱琪等师门的同学们，虽然在追寻博士学位的征途上一路艰辛，但我们共同学习、共同娱乐，相互关心、相互支持，这段美好的时光值得一生珍藏。感谢 IMPACT Group 的同学们在这两年期间对我在科研上的帮助： Li-Wen、 Shengzhao、 Chris、 Ray、 Xiao-Long、 John、 Wei-Sheng、 Carl、 Hee-Seok、 Abdul、 Tom、 Izzat、 Simon、 Jie、 Zhihao、 Sitao 等。特别感谢 Xiaolin 对我在生活上的关心和帮助，让我在异国他乡不会感到孤单无助。

最后要感谢我的家人，你们是我在风雨阴霾中仍然坚定前行的动力。是你们给予我最无私的亲情，让我知道无论什么时候我都会有你们在身后默默地支持和关心。虽然有时候我没能耐心倾听你们的关切言语，虽然我很少抽出时间陪伴你们，但是你们从来都没有责备过我，而是一如既往地支持我、关心我。在此，我要特别向你们说一声：谢谢！衷心祝福你们身体健康、幸福快乐！

参考文献

[1] Esmaeilzadeh H, Blem E, St Amant R, et al. Dark Silicon and the End of Multicore Scaling [C/OL]. In Proceedings of the 38th Annual International Symposium on Computer Architecture. New York, NY, USA, 2011: 365–376. http://doi.acm.org/10.1145/2000064.2000108.

[2] Kumar R, Tullsen D, Jouppi N, et al. Heterogeneous chip multiprocessors [J/OL]. Computer. 2005, 38 (11): 32–38. http://dx.doi.org/10.1109/MC.2005.379.

[3] NVIDIA Corporation. CUDA C Programming Guide v5.5, July, 2013. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[4] Khronos OpenCL Working Group. The OpenCL C SpecificationVersion: 2.0, July, 2013. https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf.

[5] Przybylski S, Horowitz M, Hennessy J. Characteristics of Performance-optimal Multi-level Cache Hierarchies [C/OL]. In Proceedings of the 16th Annual International Symposium on Computer Architecture. New York, NY, USA, 1989: 114–121. http://doi.acm.org/10.1145/74925.74939.

[6] Jouppi N P. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers [C/OL]. In Proceedings of the 17th Annual International Symposium on Computer Architecture. New York, NY, USA, 1990: 364–373. http://doi.acm.org/10.1145/325164.325162.

[7] NVIDIA Corporation. NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™, 2009. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[8] NVIDIA Corporation. NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110, 2012. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[9] AMD Corporation. AMD Graphics Cores Next (GCN) Architecture white paper, 2012. http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[10] Kurd N, Chowdhury M, Burton E, et al. 5.9 Haswell: A family of IA 22nm processors [C/OL]. In Proceedings of the IEEE International Solid-State Circuits Conference, Digest of Technical Papers. Feb 2014: 112–113. http://dx.doi.org/10.1109/ISSCC.2014.6757361.

[11] Kyriazis G. Heterogeneous System Architecture: A Technical Review [R/OL]. 2012. http://developer.amd.com/wordpress/media/2012/10/hsa10.pdf.

[12] Jaleel A, Theobald K B, Steely S C, Jr, et al. High performance cache replacement using re-reference interval prediction (RRIP) [C/OL]. In Proceedings of the 37th Annual International Symposium on Computer Architecture. New York, NY, USA, 2010: 60–71. http://doi.acm.org/10.1145/1815961.1815971.

[13] Duong N, Zhao D, Kim T, et al. Improving Cache Management Policies Using Dynamic Reuse Distances [C/OL]. In Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2012: 389–400. http://dx.doi.org/10.1109/MICRO.2012.43.

[14] Li D. Orchestrating Thread Scheduling and Cache Management to Improve Memory System Throughput in Throughput Processors [D/OL]. [S. l.]: University of Texas at Austin, 2014. http://www.cs.utexas.edu/~cart/publications/dissertations/dongli.pdf.

[15] Suleman M A, Qureshi M K, Patt Y N. Feedback-driven Threading: Power-efficient and High-performance Execution of Multi-threaded Workloads on CMPs [C/OL]. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 2008: 277–286. http://doi.acm.org/10.1145/1346281.1346317.

[16] Cheng H-Y, Lin C-H, Li J, et al. Memory Latency Reduction via Thread Throttling [C/OL]. In Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2010: 53–64. http://dx.doi.org/10.1109/MICRO.2010.39.

[17] Rogers T G, O'Connor M, Aamodt T M. Cache-Conscious Wavefront Scheduling [C/OL]. In Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2012: 72–83. http://dx.doi.org/10.1109/MICRO.2012.16.

[18] Moore G E. Cramming More Components Onto Integrated Circuits [J/OL]. Electronics. 1965, 38 (8): 114–117. http://dx.doi.org/10.1109/JPROC.1998.658762.

[19] Yeh T-Y, Patt Y N. Two-level Adaptive Training Branch Prediction [C/OL]. In Proceedings of the 24th Annual International Symposium on Microarchitecture. New York, NY, USA, 1991: 51–61. http://doi.acm.org/10.1145/123465.123475.

[20] Patt Y N, Hwu W M, Shebanow M. HPS, a New Microarchitecture: Rationale and Introduction [C/OL]. In Proceedings of the 18th Annual Workshop on Microprogramming. New York, NY, USA, 1985: 103–108. http://doi.acm.org/10.1145/18927.18916.

[21] Patterson D A, Sequin C H. RISC I: A Reduced Instruction Set VLSI Computer [C/OL]. In Proceedings of the 8th Annual Symposium on Computer Architecture. Los Alamitos, CA, USA, 1981: 443–457. http://dl.acm.org/citation.cfm?id=800052.801895.

[22] Palacharla S, Jouppi N P, Smith J E. Complexity-effective Superscalar Processors [C/OL]. In Proceedings of the 24th Annual International Symposium on Computer Architecture. New York, NY, USA, 1997: 206–218. http://doi.acm.org/10.1145/264107.264201.

[23] Chang P P, Mahlke S A, Chen W Y, et al. IMPACT: An Architectural Framework for Multiple-instruction-issue Processors [C/OL]. In Proceedings of the 18th Annual International Symposium on Computer Architecture. New York, NY, USA, 1991: 266–275. http://doi.acm.org/10.1145/115952.115979.

[24] Sohi G S, Breach S E, Vijaykumar T N. Multiscalar Processors [C/OL]. In Proceedings of the 22nd Annual International Symposium on Computer Architecture. New York, NY, USA, 1995: 414–425. http://doi.acm.org/10.1145/223982.224451.

[25] Hinton G, Sager D, Upton M, et al. The Microarchitecture of the Pentium® 4 Processor [C/OL]. In Intel Technology Journal. 2001. http://www.ecs.umass.edu/ece/koren/ece568/papers/Pentium4.pdf.

[26] Mudge T. Power: a first-class architectural design constraint [J/OL]. Computer. 2001, 34 (4): 52–58. http://dx.doi.org/10.1109/2.917539.

[27] Olukotun K, Nayfeh B A, Hammond L, et al. The Case for a Single-chip Multiprocessor [C/OL]. In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 1996: 2–11. http://doi.acm.org/10.1145/237090.237140.

[28] Kuskin J, Ofelt D, Heinrich M, et al. The Stanford FLASH Multiprocessor [C/OL]. In Proceedings of the 21st Annual International Symposium on Computer Architecture. Los Alamitos, CA, USA, 1994: 302–313. http://dx.doi.org/10.1145/191995.192056.

[29] Hill M D, Marty M R. Amdahl's Law in the Multicore Era [J/OL]. Computer. 2008, 41 (7): 33–38. http://dx.doi.org/10.1109/MC.2008.209.

[30] Burger D, Goodman J R, Kägi A. Memory Bandwidth Limitations of Future Microprocessors [C/OL]. In Proceedings of the 23rd Annual International Symposium on Computer Architecture. New York, NY, USA, 1996: 78–89. http://doi.acm.org/10.1145/232973.232983.

[31] Huh J, Burger D, Keckler S W. Exploring the Design Space of Future CMPs [C/OL]. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. Washington, DC, USA, 2001: 199–210. http://dl.acm.org/citation.cfm?id=645988.674164.

[32] Rogers B M, Krishna A, Bell G B, et al. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling [C/OL]. In Proceedings of the 36th Annual International Symposium on Computer Architecture. New York, NY, USA, 2009: 371–382. http://doi.acm.org/10.1145/1555754.1555801.

[33] Amdahl G M. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities [C/OL]. In Proceedings of the Spring Joint Computer Conference. New York, NY, USA, April 1967: 483–485. http://doi.acm.org/10.1145/1465482.1465560.

[34] Tullsen D M, Eggers S J, Levy H M. Simultaneous Multithreading: Maximizing On-chip Parallelism [C/OL]. In Proceedings of the 22nd Annual International Symposium on Computer Architecture. New York, NY, USA, 1995: 392–403. http://doi.acm.org/10.1145/223982.224449.

[35] Tullsen D M, Eggers S J, Emer J S, et al. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor [C/OL]. In Proceedings of the 23rd Annual International Symposium on Computer Architecture. New York, NY, USA, 1996: 191–202. http://doi.acm.org/10.1145/232973.232993.

[36] Hameed R, Qadeer W, Wachs M, et al. Understanding Sources of Inefficiency in General-purpose Chips [C/OL]. In Proceedings of the 37th Annual International Symposium on Computer Architecture. New York, NY, USA, 2010: 37–47. http://doi.acm.org/10.1145/1815961.1815968.

[37] Chung E S, Milder P A, Hoe J C, et al. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? [C/OL]. In Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2010: 225–236. http://dx.doi.org/10.1109/MICRO.2010.36.

[38] Tremblay M, O'Connor J M. UltraSparc I: A Four-Issue Processor Supporting Multimedia [J/OL]. IEEE Micro. 1996, 16 (2): 42–50. http://dx.doi.org/10.1109/40.491461.

[39] Intel Corporation. Intel® SSE4 Programming Reference, July, 2007. http://home.ustc.edu.cn/~shengjie/REFERENCE/sse4_instruction_set.pdf.

[40] Intel Corporation. Intel® Advanced Vector Extensions Programming Reference, June, 2011. https://software.intel.com/sites/default/files/m/8/a/1/8/4/36945-319433-011.pdf.

[41] Lee V W, Kim C, Chhugani J, et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU [C/OL]. In Proceedings of the 37th Annual International Symposium on Computer Architecture. New York, NY, USA, 2010: 451–460. http://doi.acm.org/10.1145/1815961.1816021.

[42] Khailany B, Dally W, Kapasi U, et al. Imagine: media processing with streams [J/OL]. IEEE Micro. 2001, 21 (2): 35–46. http://dx.doi.org/10.1109/40.918001.

[43] Yang X, Yan X, Xing Z, et al. A 64-bit Stream Processor Architecture for Scientific Applications [C/OL]. In Proceedings of the 34th Annual International Symposium on Computer Architecture. New York, NY, USA, 2007: 210–219. http://doi.acm.org/10.1145/1250662.1250689.

[44] Kapasi U, Dally W J, Rixner S, et al. The Imagine Stream Processor [C/OL]. In Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors. Washington, DC, USA, Sep 2002: 282–288. http://dl.acm.org/citation.cfm?id=846216.846937.

[45] Dally W J, Labonte F, Das A, et al. Merrimac: Supercomputing with Streams [C/OL]. In Proceedings of the ACM/IEEE Conference on Supercomputing. New York, NY, USA, 2003: 35–. http://doi.acm.org/10.1145/1048935.1050187.

[46] Taylor M B, Lee W, Miller J, et al. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams [C/OL]. In Proceedings of the 31st Annual International Symposium on Computer Architecture. Washington, DC, USA, June 2004: 2–13. http://dl.acm.org/citation.cfm?id=998680.1006733.

[47] Sankaralingam K, Nagarajan R, Liu H, et al. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture [C/OL]. In Proceedings of the 30th Annual International Symposium on Computer Architecture. New York, NY, USA, 2003: 422–433. http://doi.acm.org/10.1145/859618.859667.

[48] Kelm J H, Johnson D R, Johnson M R, et al. Rigel: an architecture and scalable programming interface for a 1000-core accelerator [C/OL]. In Proceedings of the 36th Annual International Symposium on Computer Architecture. New York, NY, USA, 2009: 140–151. http://doi.acm.org/10.1145/1555754.1555774.

[49] Karpuzcu U R, Greskamp B, Torrellas J. The BubbleWrap Many-core: Popping Cores for Sequential Acceleration [C/OL]. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA, 2009: 447–458. http://doi.acm.org/10.1145/1669112.1669169.

[50] Dreslinski R G, Fick D, Giridhar B, et al. Centip3De: A Many-core Prototype Exploring 3D Integration and Near-threshold Computing [J/OL]. Communications of the ACM. 2013, 56 (11): 97–104. http://doi.acm.org/10.1145/2524713.2524725.

[51] del Cuvillo J, Zhu W, Hu Z, et al. Toward a Software Infrastructure for the Cyclops-64 Cellular Architecture [C/OL]. In Proceedings of the 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment. Washington, DC, USA, May 2006: 9–15. http://dx.doi.org/10.1109/HPCS.2006.48.

[52] Nickolls J, Dally W J. The GPU Computing Era [J/OL]. IEEE Micro. 2010, 30 (2): 56–69. http://dx.doi.org/10.1109/MM.2010.41.

[53] Keckler S W, Dally W J, Khailany B, et al. GPUs and the Future of Parallel Computing [J/OL]. IEEE Micro. 2011, 31 (5): 7–17. http://dx.doi.org/10.1109/MM.2011.89.

[54] Seiler L, Carmean D, Sprangle E, et al. Larrabee: a many-core x86 architecture for visual computing [J/OL]. ACM Transactions on Graphics. 2008, 27 (3): 18:1–18:15. http://doi.acm.org/10.1145/1360612.1360617.

[55] Duran A, Klemm M. The Intel® Many Integrated Core Architecture [C/OL]. In International Conference on High Performance Computing and Simulation. July 2012: 365–366. http://dx.doi.org/10.1109/HPCSim.2012.6266938.

[56] Chrysos G. Intel® Xeon Phi™ Coprocessor - the Architecture [R/OL]. 2012. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner.

[57] The OpenMP® API specification for parallel programming. https://www.http://openmp.org.

[58] Intel® Threading Building Blocks (Intel® TBB). https://www.threadingbuildingblocks.org.

[59] Reinders J. Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism [M/OL]. O'Reilly Media, 2007. http://books.google.com/books?id=do86P6kb0msC.

[60] Intel® Cilk™ Plus. http://www.cilkplus.org.

[61] Kumar R, Tullsen D M, Ranganathan P, et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance [C/OL]. In Proceedings of the 31st Annual International Symposium on Computer Architecture. Washington, DC, USA, 2004: 64–. http://dl.acm.org/citation.cfm?id=998680.1006707.

[62] Flachs B, Asano S, Dhong S, et al. A streaming processing unit for a CELL processor [C/OL]. In Proceedings of the IEEE International Solid-State Circuits Conference, Digest of Technical Papers. Feb 2005: 134–135 Vol. 1. http://dx.doi.org/10.1109/ISSCC.2005.1493905.

[63] Hofstee H P. Power Efficient Processor Architecture and The Cell Processor [C/OL]. In Proceedings of the 11th International Symposium on High Performance Computer Architecture. Washington, DC, USA, 2005: 258–262. http://dx.doi.org/10.1109/HPCA.2005.26.

[64] Kalla R, Sinharoy B, Tendler J M. IBM Power5 Chip: A Dual-Core Multithreaded Processor [J/OL]. IEEE Micro. 2004, 24 (2): 40–47. http://dx.doi.org/10.1109/MM.2004.1289290.

[65] Kalla R, Sinharoy B, Starke W J, et al. Power7: IBM's Next-Generation Server Processor [J/OL]. IEEE Micro. 2010, 30 (2): 7–15. http://dx.doi.org/10.1109/MM.2010.38.

[66] Yuffe M, Knoll E, Mehalel M, et al. A fully integrated multi-CPU, GPU and memory controller 32nm processor [C/OL]. In Proceedings of the IEEE International Solid-State Circuits Conference, Digest of Technical Papers. Feb 2011: 264–266. http://dx.doi.org/10.1109/ISSCC.2011.5746311.

[67] Damaraju S, George V, Jahagirdar S, et al. A 22nm IA multi-CPU and GPU System-on-Chip [C/OL]. In Proceedings of the IEEE International Solid-State Circuits Conference, Digest of Technical Papers. Feb 2012: 56–57. http://dx.doi.org/10.1109/ISSCC.2012.6176876.

[68] Hammarlund P, Martinez A J, Bajwa A A, et al. Haswell: The Fourth-Generation Intel Core Processor [J/OL]. IEEE Micro. 2014, 34 (2): 6–20. http://dx.doi.org/10.1109/MM.2014.10.

[69] Branover A, Foley D, Steinman M. AMD Fusion APU: Llano [J/OL]. IEEE Micro. 2012, 32 (2): 28–37. http://dx.doi.org/10.1109/MM.2012.2.

[70] Tarditi D, Puri S, Oglesby J. Accelerator: Using Data Parallelism to Program GPUs for General-purpose Uses [C/OL]. In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 2006: 325–335. http://doi.acm.org/10.1145/1168857.1168898.

[71] Lindholm E, Nickolls J, Oberman S, et al. NVIDIA Tesla: A Unified Graphics and Computing Architecture [J/OL]. IEEE Micro. 2008, 28 (2): 39–55. http://dx.doi.org/10.1109/MM.2008.31.

[72] Wittenbrink C, Kilgariff E, Prabhu A. Fermi GF100 GPU Architecture [J/OL]. IEEE Micro. 2011, 31 (2): 50–59. http://dx.doi.org/10.1109/MM.2011.24.

[73] NVIDIA Corporation. NVIDIA GeForce GTX 680 Whitepaper v1.0, 2012. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[74] NVIDIA Corporation. Tuning CUDA Applications for Kepler, July, 2013. http://docs.nvidia.com/cuda/pdf/Kepler_Tuning_Guide.pdf.

[75] NVIDIA Corporation. Tuning CUDA Applications for Maxwell, Feb, 2014. http://docs.nvidia.com/cuda/pdf/Maxwell_Tuning_Guide.pdf.

[76] NVIDIA Corporation. NVIDIA GeForce GTX 750 Ti Whitepaper v1.1, 2014. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf.

[77] Dally W J, Towles B. Route Packets, Not Wires: On-chip Inteconnection Networks [C/OL]. In Proceedings of the 38th Annual Design Automation Conference. New York, NY, USA, 2001: 684–689. http://doi.acm.org/10.1145/378239.379048.

[78] Narasiman V, Shebanow M, Lee C J, et al. Improving GPU performance via large warps and two-level warp scheduling [C/OL]. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA, 2011: 308–317. http://doi.acm.org/10.1145/2155620.2155656.

[79] Gebhart M, Johnson D R, Tarjan D, et al. Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors [C/OL]. In Proceedings of the 38th Annual International Symposium on Computer Architecture. New York, NY, USA, 2011: 235–246. http://doi.acm.org/10.1145/2000064.2000093.

[80] Jog A, Kayiran O, Chidambaram Nachiappan N, et al. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance [C/OL]. In Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 2013: 395–406. http://doi.acm.org/10.1145/2451116.2451158.

[81] Rogers T G, O'Connor M, Aamodt T M. Divergence-aware Warp Scheduling [C/OL]. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA, 2013: 99–110. http://doi.acm.org/10.1145/2540708.2540718.

[82] Rixner S, Dally W J, Kapasi U J, et al. Memory Access Scheduling [C/OL]. In Proceedings of the 27th Annual International Symposium on Computer Architecture. New York, NY, USA, 2000: 128–138. http://doi.acm.org/10.1145/339647.339668.

[83] Baghsorkhi S S, Gelado I, Delahaye M, et al. Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors [C/OL]. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2012: 23–34. http://doi.acm.org/10.1145/2145816.2145820.

[84] Hughes C, Kim C, Chen Y-K. Performance and Energy Implications of Many-Core Caches for Throughput Computing [J/OL]. IEEE Micro. 2010, 30 (6): 25–35. http://dx.doi.org/10.1109/MM.2010.83.

[85] Jouppi N P. Cache write policies and performance [C/OL]. In Proceedings of the 20th Annual International Symposium on Computer architecture. New York, NY, USA, 1993: 191–201. http://doi.acm.org/10.1145/165123.165154.

[86] Singh I, Shriraman A, Fung W W L, et al. Cache Coherence for GPU Architectures [C/OL]. In Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture. Washington, DC, USA, 2013: 578–590. http://dx.doi.org/10.1109/HPCA.2013.6522351.

[87] Hechtman B A, Che S, Hower D R, et al. QuickRelease: A throughput-oriented approach to release consistency on GPUs [C/OL]. In Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture. Feb 2014: 189–200. http://dx.doi.org/10.1109/HPCA.2014.6835930.

[88] D J Sorin D A W, Mark D Hill. A Primer on Memory Consistency and Cache Coherence [M/OL]. 1st ed. Morgan and Claypool Publishers, 2011. https://class.stanford.edu/c4x/Engineering/CS316/asset/A_Primer_on_Memory_Consistency_and_Coherence.pdf.

[89] Martin M M K, Hill M D, Sorin D J. Why on-chip cache coherence is here to stay [J/OL]. Communications of the ACM. 2012, 55 (7): 78–89. http://doi.acm.org/10.1145/2209249.2209269.

[90] Baer J-L, Wang W-H. On the inclusion properties for multi-level cache hierarchies [C/OL]. In Proceedings of the 15th Annual International Symposium on Computer Architecture. Los Alamitos, CA, USA, 1988: 73–80. http://dl.acm.org/citation.cfm?id=52400.52409.

[91] Anssari N. Using hybrid shared and distributed caching for mixed-coherency GPU workloads [D/OL]. [S. l.]: University of Illinois at Urbana-Champaign, 2012. https://www.ideals.illinois.edu/bitstream/handle/2142/42361/Nasser_Anssari.pdf?sequence=1.

[92] Bakhoda A, Yuan G, Fung W, et al. Analyzing CUDA workloads using a detailed GPU simulator [C/OL]. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software. Boston, MA, 2009: 163–174. http://dx.doi.org/10.1109/ISPASS.2009.4919648.

[93] NVIDIA Corporation. PTX: Parallel Thread Execution ISA Version 2.3, 2011. https://www.cs.drexel.edu/~jjohnson/2010-11/summer/cs680/resources/doc/ptx_isa_2.3.pdf.

[94] Leng J, Hetherington T, ElTantawy A, et al. GPUWattch: Enabling Energy Optimizations in GPGPUs [C/OL]. In Proceedings of the 40th Annual International Symposium on Computer Architecture. New York, NY, USA, 2013: 487–498. http://doi.acm.org/10.1145/2485922.2485964.

[95] Asanovic K, Bodik R, Catanzaro B C, et al. The Landscape of Parallel Computing Research: A View from Berkeley [R/OL]. 2006. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.227.1678.

[96] Open MPI: Open Source High Performance Computing. http://www.open-mpi.org/.

[97] Herlihy M, Moss J E B. Transactional Memory: Architectural Support for Lock-free Data Structures [C/OL]. In Proceedings of the 20th Annual International Symposium on Computer Architecture. New York, NY, USA, 1993: 289–300. http://doi.acm.org/10.1145/165123.165164.

[98] Steffan J G, Colohan C B, Zhai A, et al. A Scalable Approach to Thread-level Speculation [C/OL]. In Proceedings of the 27th Annual International Symposium on Computer Architecture. New York, NY, USA, 2000: 1–12. http://doi.acm.org/10.1145/339647.339650.

[99] Prabhu M K, Olukotun K. Using Thread-level Speculation to Simplify Manual Parallelization [C/OL]. In Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2003: 1–12. http://doi.acm.org/10.1145/781498.781500.

[100] Krishnan V, Torrellas J. A Chip-Multiprocessor Architecture with Speculative Multithreading [J/OL]. IEEE Transactions on Computers. 1999, 48 (9): 866–880. http://dx.doi.org/10.1109/12.795218.

[101] Frigo M, Leiserson C E, Randall K H. The Implementation of the Cilk-5 Multithreaded Language [C/OL]. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, 1998: 212–223. http://doi.acm.org/10.1145/277650.277725.

[102] Kumar S, Hughes C J, Nguyen A. Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors [C/OL]. In Proceedings of the 34th Annual International Symposium on Computer Architecture. New York, NY, USA, 2007: 162–173. http://doi.acm.org/10.1145/1250662.1250683.

[103] Sanchez D, Yoo R M, Kozyrakis C. Flexible Architectural Support for Fine-grain Scheduling [C/OL]. In Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 2010: 311–322. http://doi.acm.org/10.1145/1736020.1736055.

[104] Saha B, Zhou X, Chen H, et al. Programming Model for a Heterogeneous x86 Platform [C/OL]. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, 2009: 431–440. http://doi.acm.org/10.1145/1542476.1542525.

[105] Wang P H, Collins J D, Chinya G N, et al. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System [C/OL]. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, 2007: 156–166. http://doi.acm.org/10.1145/1250734.1250753.

[106] Linderman M D, Collins J D, Wang H, et al. Merge: A Programming Model for Heterogeneous Multi-core Systems [C/OL]. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 2008: 287–296. http://doi.acm.org/10.1145/1346281.1346318.

[107] Chafi H, Sujeeth A K, Brown K J, et al. A Domain-specific Approach to Heterogeneous Parallelism [C/OL]. In Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2011: 35–46. http://doi.acm.org/10.1145/1941553.1941561.

[108] Sujeeth A K, Brown K J, Lee H, et al. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages [J/OL]. ACM Transactions on Embedded Computing Systems. 2014, 13 (4s): 134:1–134:25. http://doi.acm.org/10.1145/2584665.

[109] Diamos G F, Yalamanchili S. Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems [C/OL]. In Proceedings of the 17th International Symposium on High Performance Distributed Computing. New York, NY, USA, 2008: 197–200. http://doi.acm.org/10.1145/1383422.1383447.

[110] Owens J, Houston M, Luebke D, et al. GPU Computing [J/OL]. Proceedings of the IEEE. 2008, 96 (5): 879–899. http://dx.doi.org/10.1109/JPROC.2008.917757.

[111] Mattson P. A Programming System for the Imagine Media Processor [D/OL]. [S. l.]: Stanford University, 2002. http://cva.stanford.edu/publications/2001/pmattson_final2_pre.pdf.

[112] Thies W, Karczmarek M, Amarasinghe S P. StreamIt: A Language for Streaming Applications [C/OL]. In Proceedings of the 11th International Conference on Compiler Construction. London, UK, UK, 2002: 179–196. http://dl.acm.org/citation.cfm?id=647478.727935.

[113] Buck I, Foley T, Horn D, et al. Brook for GPUs: Stream Computing on Graphics Hardware [J/OL]. ACM Transactions on Graphics. 2004, 23 (3): 777–786. http://doi.acm.org/10.1145/1015706.1015800.

[114] Kirk D B, Hwu W-m W. Programming Massively Parallel Processors: A Hands-on Approach [M/OL]. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. http://books.google.com/books/about/Programming_Massively_Parallel_Processor.html?id=qW1mncii_6EC.

[115] Hwu W. GPU Computing Gems [M/OL]. Morgan Kaufmann, 2012. http://books.google.com/books?id=dNvantWW7HMC.

[116] Hower D R, Hechtman B A, Beckmann B M, et al. Heterogeneous-race-free Memory Models [C/OL]. In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 2014: 427–440. http://doi.acm.org/10.1145/2541940.2541981.

[117] Ryoo S, Rodrigues C I, Baghsorkhi S S, et al. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA [C/OL]. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2008: 73–82. http://doi.acm.org/10.1145/1345206.1345220.

[118] Ryoo S, Rodrigues C I, Stone S S, et al. Program Optimization Space Pruning for a Multithreaded Gpu [C/OL]. In Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization. New York, NY, USA, 2008: 195–204. http://doi.acm.org/10.1145/1356058.1356084.

[119] Stratton J A, Stone S S, Hwu W-M W. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs [C/OL]. In Proceedings of the Workshops on Languages and Compilers for Parallel Computing. Berlin, Heidelberg, 2008: 16–30. http://dx.doi.org/10.1007/978-3-540-89740-8_2.

[120] Diamos G F, Kerr A R, Yalamanchili S, et al. Ocelot: A Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems [C/OL]. In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. New York, NY, USA, 2010: 353–364. http://doi.acm.org/10.1145/1854273.1854318.

[121] Kim H-S, El Hajj I, Stratton J A, et al. Multi-tier Dynamic Vectorization for Translating GPU Optimizations into CPU Performance, IMPACT-14-01 [R/OL]. 2014. http://impact.crhc.illinois.edu/paper_details.aspx?paper_id=227.

[122] Gummaraju J, Morichetti L, Houston M, et al. Twin Peaks: A Software Platform for Heterogeneous Computing on General-purpose and Graphics Processors [C/OL]. In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. New York, NY, USA, 2010: 205–216. http://doi.acm.org/10.1145/1854273.1854302.

[123] Gelado I, Stone J E, Cabezas J, et al. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems [C/OL]. In Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 2010: 347–358. http://doi.acm.org/10.1145/1736020.1736059.

[124] Lee S, Min S-J, Eigenmann R. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization [C/OL]. In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2009: 101–110. http://doi.acm.org/10.1145/1504176.1504194.

[125] Jablin T B, Jablin J A, Prabhu P, et al. Dynamically Managed Data for CPU-GPU Architectures [C/OL]. In Proceedings of the 10th International Symposium on Code Generation and Optimization. New York, NY, USA, 2012: 165–174. http://doi.acm.org/10.1145/2259016.2259038.

[126] Jablin T B, Prabhu P, Jablin J A, et al. Automatic CPU-GPU Communication Management and Optimization [C/OL]. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, 2011: 142–151. http://doi.acm.org/10.1145/1993498.1993516.

[127] Luk C-K, Hong S, Kim H. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping [C/OL]. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA, 2009: 45–55. http://doi.acm.org/10.1145/1669112.1669121.

[128] Hoberock J, Bell N. Thrust. http://thrust.github.io/.

[129] Corporation A. Bolt C++ Template Library. http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-libraries/bolt-c-template-library/.

[130] Microsoft Corporation. C++ AMP: Language and Programming Model, December, 2013. http://download.microsoft.com/download/2/2/9/22972859-15C2-4D96-97AE-93344241D56C/CppAMPOpenSpecificationV12.pdf.

[131] OpenACC: Directives for Accelerators. http://www.openacc-standard.org.

[132] Catanzaro B, Garland M, Keutzer K. Copperhead: Compiling an Embedded Data Parallel Language [C/OL]. In Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2011: 47–56. http://doi.acm.org/10.1145/1941553.1941562.

[133] Merrill D, Garland M, Grimshaw A. Scalable GPU Graph Traversal [C/OL]. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2012: 117–128. http://doi.acm.org/10.1145/2145816.2145832.

[134] Yan S, Long G, Zhang Y. StreamScan: fast scan algorithms for GPUs without global barrier synchronization [C/OL]. In Proceedings of the 18th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2013: 229–238. http://doi.acm.org/10.1145/2442516.2442539.

[135] Chang L-W, Stratton J A, Kim H-S, et al. A scalable, numerically stable, high-performance tridiagonal solver using GPUs [C/OL]. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. Los Alamitos, CA, USA, 2012: 27:1–27:11. http://dl.acm.org/citation.cfm?id=2388996.2389033.

[136] Fung W W L, Sham I, Yuan G, et al. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow [C/OL]. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2007: 407–420. http://dx.doi.org/10.1109/MICRO.2007.12.

[137] Meng J, Tarjan D, Skadron K. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance [C/OL]. In Proceedings of the 37th Annual International Symposium on Computer Architecture. New York, NY, USA, 2010: 235–246. http://doi.acm.org/10.1145/1815961.1815992.

[138] Fung W W L, Aamodt T M. Thread Block Compaction for Efficient SIMT Control Flow [C/OL]. In Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture. Washington, DC, USA, 2011: 25–36. http://dl.acm.org/citation.cfm?id=2014698.2014893.

[139] Diamos G, Ashbaugh B, Maiyuran S, et al. SIMD Re-convergence at Thread Frontiers [C/OL]. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA, 2011: 477–488. http://doi.acm.org/10.1145/2155620.2155676.

[140] Vaidya A S, Shayesteh A, Woo D H, et al. SIMD Divergence Optimization Through Intra-warp Compaction [C/OL]. In Proceedings of the 40th Annual International Symposium on Computer Architecture. New York, NY, USA, 2013: 368–379. http://doi.acm.org/10.1145/2485922.2485954.

[141] Rhu M, Erez M. CAPRI: Prediction of Compaction-adequacy for Handling Control-divergence in GPGPU Architectures [C/OL]. In Proceedings of the 39th Annual International Symposium on Computer Architecture. Washington, DC, USA, 2012: 61–71. http://dl.acm.org/citation.cfm?id=2337159.2337167.

[142] Rhu M, Erez M. The Dual-path Execution Model for Efficient GPU Control Flow [C/OL]. In Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture. Washington, DC, USA, 2013: 591–602. http://dx.doi.org/10.1109/HPCA.2013.6522352.

[143] Rhu M, Erez M. Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation [C/OL]. In Proceedings of the 40th Annual International Symposium on Computer Architecture. New York, NY, USA, 2013: 356–367. http://doi.acm.org/10.1145/2485922.2485953.

[144] Che S, Boyer M, Meng J, et al. Rodinia: A benchmark suite for heterogeneous computing [C/OL]. In Proceedings of the IEEE International Symposium on Workload Characterization. 2009: 44–54. http://dx.doi.org/10.1109/IISWC.2009.5306797.

[145] He B, Fang W, Luo Q, et al. Mars: A MapReduce Framework on Graphics Processors [C/OL]. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. New York, NY, USA, 2008: 260–269. http://doi.acm.org/10.1145/1454115.1454152.

[146] Stratton J A, Rodrigrues C, Sung I-J, et al. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing, IMPACT-12-01 [R/OL]. 2012. http://impact.crhc.illinois.edu/Shared/Docs/impact-12-01.parboil.pdf.

[147] NVIDIA CUDA C/C++ SDK code samples. 2011. http://docs.nvidia.com/cuda/cuda-samples/index.html#axzz38JEwl6lY.

[148] Grauer-Gray S, Xu L, Searles R, et al. Auto-tuning a high-level language targeted to GPU codes [C/OL]. In Proceedings of the Innovative Parallel Computing. May 2012: 1–10. http://dx.doi.org/10.1109/InPar.2012.6339595.

[149] Balasubramonian R, Jouppi N. Multi-Core Cache Hierarchies [M/OL]. 1st ed. Morgan & Claypool Publishers, 2011. https://class.stanford.edu/c4x/Engineering/CS316/asset/Multi-core_Cache_Hierarchies.pdf.

[150] Qureshi M K, Lynch D N, Mutlu O, et al. A Case for MLP-Aware Cache Replacement [C/OL]. In Proceedings of the 33rd Annual International Symposium on Computer Architecture. Washington, DC, USA, 2006: 167–178. http://dx.doi.org/10.1109/ISCA.2006.5.

[151] Subramanian R, Smaragdakis Y, Loh G H. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads [C/OL]. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2006: 385–396. http://dx.doi.org/10.1109/MICRO.2006.7.

[152] Jaleel A, Najaf-abadi H H, Subramaniam S, et al. CRUISE: Cache Replacement and Utility-aware Scheduling [C/OL]. In Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 2012: 249–260. http://doi.acm.org/10.1145/2150976.2151003.

[153] Qureshi M K, Thompson D, Patt Y N. The V-Way Cache: Demand Based Associativity via Global Replacement [C/OL]. In Proceedings of the 32nd Annual International Symposium on Computer Architecture. Washington, DC, USA, 2005: 544–555. http://dx.doi.org/10.1109/ISCA.2005.52.

[154] Lee D, Choi J, Kim J H, et al. LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies [J/OL]. IEEE Transactions on Computers. 2001, 50 (12): 1352–1361. http://dx.doi.org/10.1109/TC.2001.970573.

[155] Belady L A. A Study of Replacement Algorithms for a Virtual-storage Computer [J/OL]. IBM Systems Journal. 1966, 5 (2): 78–101. http://dx.doi.org/10.1147/sj.52.0078.

[156] Mattson R L, Gecsei J, Slutz D R, et al. Evaluation Techniques for Storage Hierarchies [J/OL]. IBM Systems Journal. 1970, 9 (2): 78–117. http://dx.doi.org/10.1147/sj.92.0078.

[157] Handy J. The Cache Memory Book [M/OL]. San Diego, CA, USA: Academic Press Professional, Inc., 1993. http://books.google.com/books/about/The_Cache_Memory_Book.html?id=-7oOlb-lCpMC.

[158] Intel Corporation. Inside the Intel Itanium 2 Processor, July, 2002. www.dig64.org/about/Itanium2_white_paper_public.pdf.

[159] Jaleel A, Hasenplaugh W, Qureshi M, et al. Adaptive Insertion Policies for Managing Shared Caches [C/OL]. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. New York, NY, USA, 2008: 208–219. http://doi.acm.org/10.1145/1454115.1454145.

[160] Qureshi M K, Jaleel A, Patt Y N, et al. Adaptive Insertion Policies for High Performance Caching [C/OL]. In Proceedings of the 34th Annual International Symposium on Computer Architecture. New York, NY, USA, 2007: 381–391. http://doi.acm.org/10.1145/1250662.1250709.

[161] Keramidas G, Petoumenos P, Kaxiras S. Cache replacement based on reuse-distance prediction [C/OL]. In Proceedings of the 25th International Conference on Computer Design. Oct 2007: 245–250. http://dx.doi.org/10.1109/ICCD.2007.4601909.

[162] Wu C-J, Jaleel A, Hasenplaugh W, et al. SHiP: Signature-based Hit Predictor for High Performance Caching [C/OL]. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA, 2011: 430–441. http://doi.acm.org/10.1145/2155620.2155671.

[163] Chaudhuri M. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches [C/OL]. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA, 2009: 401–412. http://doi.acm.org/10.1145/1669112.1669164.

[164] Tyson G, Farrens M, Matthews J, et al. A Modified Approach to Data Cache Management [C/OL]. In Proceedings of the 28th Annual International Symposium on Microarchitecture. Los Alamitos, CA, USA, 1995: 93–103. http://dl.acm.org/citation.cfm?id=225160.225177.

[165] Johnson T L, Hwu W-m W. Run-time Adaptive Cache Hierarchy Management via Reference Analysis [C/OL]. In Proceedings of the 24th Annual International Symposium on Computer Architecture. New York, NY, USA, 1997: 315–326. http://doi.acm.org/10.1145/264107.264213.

[166] Johnson T L, Connors D A, Merten M C, et al. Run-Time Cache Bypassing [J/OL]. IEEE Transactions on Computers. 1999, 48 (12): 1338–1354. http://dx.doi.org/10.1109/12.817393.

[167] Khan S M, Tian Y, Jimenez D A. Sampling Dead Block Prediction for Last-Level Caches [C/OL]. In Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2010: 175–186. http://dx.doi.org/10.1109/MICRO.2010.24.

[168] Albericio J, Ibáñez P, Viñals V, et al. The Reuse Cache: Downsizing the Shared Last-level Cache [C/OL]. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA, 2013: 310–321. http://doi.acm.org/10.1145/2540708.2540735.

[169] Sandberg A, Eklöv D, Hagersten E. Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses [C/OL]. In Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. Washington, DC, USA, 2010: 1–11. http://dx.doi.org/10.1109/SC.2010.44.

[170] Kurian G, Khan O, Devadas S. The locality-aware adaptive cache coherence protocol [C/OL]. In Proceedings of the 40th Annual International Symposium on Computer Architecture. New York, NY, USA, 2013: 523–534. http://doi.acm.org/10.1145/2485922.2485967.

[171] Gaur J, Chaudhuri M, Subramoney S. Bypass and Insertion Algorithms for Exclusive Last-level Caches [C/OL]. In Proceedings of the 38th Annual International Symposium on Computer Architecture. New York, NY, USA, 2011: 81–92. http://doi.acm.org/10.1145/2000064.2000075.

[172] Kharbutli M, Solihin D. Counter-Based Cache Replacement and Bypassing Algorithms [J/OL]. IEEE Transactions on Computers. 2008, 57 (4): 433–447. http://dx.doi.org/10.1109/TC.2007.70816.

[173] Choi H, Ahn J, Sung W. Reducing off-chip memory traffic by selective cache management scheme in GPGPUs [C/OL]. In Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units. New York, NY, USA, 2012: 110–119. http://doi.acm.org/10.1145/2159430.2159443.

[174] Xie X, Liang Y, Sun G, et al. An Efficient Compiler Framework for Cache Bypassing on GPUs [C/OL]. In Proceedings of the International Conference on Computer-Aided Design. Piscataway, NJ, USA, 2013: 516–523. http://dl.acm.org/citation.cfm?id=2561828.2561929.

[175] Wang Z, McKinley K S, Rosenberg A L, et al. Using the Compiler to Improve Cache Replacement Decisions [C/OL]. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. Washington, DC, USA, 2002: 199–. http://dl.acm.org/citation.cfm?id=645989.674328.

[176] Lai A-C, Fide C, Falsafi B. Dead-block prediction & dead-block correlating prefetchers [C/OL]. In Proceedings of the 28th Annual International Symposium on Computer Architecture. New York, NY, USA, 2001: 144–154. http://doi.acm.org/10.1145/379240.379259.

[177] Liu H, Ferdman M, Huh J, et al. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency [C/OL]. In Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2008: 222–233. http://dx.doi.org/10.1109/MICRO.2008.4771793.

[178] Jia W, Shaw K A, Martonosi M. MRPB: Memory request prioritization for massively parallel processors [C/OL]. In Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture. Feb 2014: 272–283. http://dx.doi.org/10.1109/HPCA.2014.6835938.

[179] Lee J, Wu H, Ravichandran M, et al. Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications [C/OL]. In Proceedings of the 37th Annual International Symposium on Computer Architecture. New York, NY, USA, 2010: 270–279. http://doi.acm.org/10.1145/1815961.1815996.

[180] Zhuravlev S, Blagodurov S, Fedorova A. Addressing Shared Resource Contention in Multicore Processors via Scheduling [C/OL]. In Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 2010: 129–142. http://doi.acm.org/10.1145/1736020.1736036.

[181] Kayiran O, Jog A, Kandemir M T, et al. Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs [C/OL]. In Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques. Piscataway, NJ, USA, 2013: 157–166. http://dl.acm.org/citation.cfm?id=2523721.2523745.

[182] Semeraro G, Magklis G, Balasubramonian R, et al. Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling [C/OL]. In Proceedings of the 8th International Symposium on High Performance Computer Architecture. Washington, DC, USA, 2002: 29–. http://dl.acm.org/citation.cfm?id=874076.876477.

[183] Semeraro G, Albonesi D H, Dropsho S G, et al. Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture [C/OL]. In Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture. Los Alamitos, CA, USA, 2002: 356–367. http://dl.acm.org/citation.cfm?id=774861.774899.

[184] Kim W, Gupta M, Wei G-Y, et al. System level analysis of fast, per-core DVFS using on-chip switching regulators [C/OL]. In Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture. Feb 2008: 123–134. http://dx.doi.org/10.1109/HPCA.2008.4658633.

[185] Howard J, Dighe S, Hoskote Y, et al. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS [C/OL]. In Proceedings of the IEEE International Solid-State Circuits Conference, Digest of Technical Papers. Feb 2010: 108–109. http://dx.doi.org/10.1109/ISSCC.2010.5434077.

[186] Manne S, Klauser A, Grunwald D. Pipeline Gating: Speculation Control for Energy Reduction [C/OL]. In Proceedings of the 25th Annual International Symposium on Computer Architecture. Washington, DC, USA, 1998: 132–141. http://doi.acm.org/10.1145/279361.279377.

[187] Isci C, Buyuktosunoglu A, Cher C-Y, et al. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget [C/OL]. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2006: 347–358. http://dx.doi.org/10.1109/MICRO.2006.8.

[188] Li J, Martinez J. Dynamic power-performance adaptation of parallel computation on chip multiprocessors [C/OL]. In Proceedings of the 12th International Symposium on High Performance Computer Architecture. Feb 2006: 77–87. http://dx.doi.org/10.1109/HPCA.2006.1598114.

[189] Bitirgen R, Ipek E, Martinez J F. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach [C/OL]. In Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2008: 318–329. http://dx.doi.org/10.1109/MICRO.2008.4771801.

[190] Magklis G, Scott M L, Semeraro G, et al. Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor [C/OL]. In Proceedings of the 30th Annual International Symposium on Computer Architecture. New York, NY, USA, 2003: 14–27. http://doi.acm.org/10.1145/859618.859621.

[191] Hsu C-H, Kremer U. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction [C/OL]. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, 2003: 38–48. http://doi.acm.org/10.1145/781131.781137.

[192] Xie F, Martonosi M, Malik S. Compile-time Dynamic Voltage Scaling Settings: Opportunities and Limits [C/OL]. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, 2003: 49–62. http://doi.acm.org/10.1145/781131.781138.

[193] Wu Q, Martonosi M, Clark D W, et al. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance [C/OL]. In Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2005: 271–282. http://dx.doi.org/10.1109/MICRO.2005.7.

[194] Meisner D, Gold B T, Wenisch T F. PowerNap: Eliminating Server Idle Power [C/OL]. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 2009: 205–216. http://doi.acm.org/10.1145/1508244.1508269.

[195] Hong S, Kim H. An Integrated GPU Power and Performance Model [C/OL]. In Proceedings of the 37th Annual International Symposium on Computer Architecture. New York, NY, USA, 2010: 280–289. http://doi.acm.org/10.1145/1815961.1815998.

[196] Isci C, Contreras G, Martonosi M. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management [C/OL]. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2006: 359–370. http://dx.doi.org/10.1109/MICRO.2006.30.

[197] Lee J, Sathisha V, Schulte M, et al. Improving Throughput of Power-Constrained GPUs Using Dynamic Voltage/Frequency and Core Scaling [C/OL]. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. Washington, DC, USA, 2011: 111–120. http://dx.doi.org/10.1109/PACT.2011.17.

[198] Abdel-Majeed M, Wong D, Annavaram M. Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs [C/OL]. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA, 2013: 111–122. http://doi.acm.org/10.1145/2540708.2540719.

[199] Qureshi M K, Patt Y N. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches [C/OL]. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2006: 423–432. http://dx.doi.org/10.1109/MICRO.2006.49.

[200] Xie Y, Loh G H. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-core Shared Caches [C/OL]. In Proceedings of the 36th Annual International Symposium on Computer Architecture. New York, NY, USA, 2009: 174–183. http://doi.acm.org/10.1145/1555754.1555778.

[201] Chang J, Sohi G S. Cooperative Caching for Chip Multiprocessors [C/OL]. In Proceedings of the 33rd annual international symposium on Computer Architecture. Washington, DC, USA, 2006: 264–276. http://dx.doi.org/10.1109/ISCA.2006.17.

[202] Lee J, Kim H. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture [C/OL]. In Proceedings of the IEEE 18th International Symposium on High Performance Computer Architecture. Washington, DC, USA, 2012: 1–12. http://dx.doi.org/10.1109/HPCA.2012.6168947.

[203] Gaur J, Srinivasan R, Subramoney S, et al. Efficient Management of Last-level Caches in Graphics Processors for 3D Scene Rendering Workloads [C/OL]. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA, 2013: 395–407. http://doi.acm.org/10.1145/2540708.2540742.

[204] Mekkat V, Holey A, Yew P-C, et al. Managing shared last-level cache in a heterogeneous multicore processor [C/OL]. In Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques. Piscataway, NJ, USA, 2013: 225–234. http://dl.acm.org/citation.cfm?id=2523721.2523753.

[205] Jaleel A, Borch E, Bhandaru M, et al. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies [C/OL]. In Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. 2010: 151–162. http://dx.doi.org/10.1109/MICRO.2010.52.

[206] Sim J, Lee J, Qureshi M K, et al. FLEXclusion: balancing cache capacity and on-chip bandwidth via flexible exclusion [C/OL]. In Proceedings of the 39th Annual International Symposium on Computer Architecture. Washington, DC, USA, 2012: 321–332. http://dl.acm.org/citation.cfm?id=2337159.2337196.

[207] Lee J, Lakshminarayana N B, Kim H, et al. Many-Thread Aware Prefetching Mechanisms for GPGPU Applications [C/OL]. In Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2010: 213–224. http://dx.doi.org/10.1109/MICRO.2010.44.

[208] Jog A, Kayiran O, Mishra A K, et al. Orchestrated Scheduling and Prefetching for GPGPUs [C/OL]. In Proceedings of the 40th Annual International Symposium on Computer Architecture. New York, NY, USA, 2013: 332–343. http://doi.acm.org/10.1145/2485922.2485951.

[209] Sethia A, Dasika G, Samadi M, et al. APOGEE: Adaptive prefetching on GPUs for energy efficiency [C/OL]. In Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques. Sept 2013: 73–82. http://dx.doi.org/10.1109/PACT.2013.6618805.

[210] Rhu M, Sullivan M, Leng J, et al. A Locality-aware Memory Hierarchy for Energy-efficient GPU Architectures [C/OL]. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA, 2013: 86–98. http://doi.acm.org/10.1145/2540708.2540717.

[211] Gebhart M, Keckler S W, Khailany B, et al. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor [C/OL]. In Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA, 2012: 96–106. http://dx.doi.org/10.1109/MICRO.2012.18.

[212] Kim H, Vuduc R, Baghsorkhi S, et al. Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU) [J/OL]. Synthesis Lectures on Computer Architecture. 2013, 7 (2): 1–96. http://impact.crhc.illinois.edu/shared/papers/sara2012.pdf.

[213] Baghsorkhi S S, Delahaye M, Patel S J, et al. An Adaptive Performance Modeling Tool for GPU Architectures [C/OL]. In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2010: 105–114. http://doi.acm.org/10.1145/1693453.1693470.

[214] Sung I-J, Stratton J A, Hwu W-M W. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications [C/OL]. In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. New York, NY, USA, 2010: 513–522. http://doi.acm.org/10.1145/1854273.1854336.

[215] Sung I-J, Liu G, Hwu W-M. DL: A data layout transformation system for heterogeneous computing [C/OL]. In Innovative Parallel Computing. May 2012: 1–11. http://dx.doi.org/10.1109/InPar.2012.6339606.

[216] Sung I-J, Gómez-Luna J, González-Linares J M, et al. In-place Transposition of Rectangular Matrices on Accelerators [C/OL]. In Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2014: 207–218. http://doi.acm.org/10.1145/2555243.2555266.

[217] Ueng S-Z, Lathara M, Baghsorkhi S S, et al. CUDA-Lite: Reducing GPU Programming Complexity [C/OL]. In Proceedings of the Workshops on Languages and Compilers for Parallel Computing. Berlin, Heidelberg, 2008: 1–15. http://dx.doi.org/10.1007/978-3-540-89740-8_1.

[218] Yang Y, Xiang P, Kong J, et al. A GPGPU Compiler for Memory Optimization and Parallelism Management [C/OL]. In Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, 2010: 86–97. http://doi.acm.org/10.1145/1806596.1806606.

[219] Zhang E Z, Jiang Y, Guo Z, et al. Streamlining GPU Applications on the Fly: Thread Divergence Elimination Through Runtime Thread-data Remapping [C/OL]. In Proceedings of the 24th ACM International Conference on Supercomputing. New York, NY, USA, 2010: 115–126. http://doi.acm.org/10.1145/1810085.1810104.

[220] Zhang E Z, Jiang Y, Guo Z, et al. On-the-fly Elimination of Dynamic Irregularities for GPU Computing [C/OL]. In Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 2011: 369–380. http://doi.acm.org/10.1145/1950365.1950408.

[221] Che S, Sheaffer J W, Skadron K. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems [C/OL]. In Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis. New York, NY, USA, 2011: 13:1–13:11. http://doi.acm.org/10.1145/2063384.2063401.

[222] Calder B, Krintz C, John S, et al. Cache-conscious Data Placement [C/OL]. In Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 1998: 139–149. http://doi.acm.org/10.1145/291069.291036.

[223] Kandemir M, Yemliha T, Muralidhara S, et al. Cache Topology Aware Computation Mapping for Multicores [C/OL]. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, 2010: 74–85. http://doi.acm.org/10.1145/1806596.1806605.

[224] Jia W, Shaw K A, Martonosi M. Characterizing and improving the use of demand-fetched caches in GPUs [C/OL]. In Proceedings of the 26th ACM international conference on Supercomputing. New York, NY, USA, 2012: 15–24. http://doi.acm.org/10.1145/2304576.2304582.

[225] Wong H, Papadopoulou M-M, Sadooghi-Alvandi M, et al. Demystifying GPU microarchitecture through microbenchmarking [C/OL]. In Proceedings of IEEE International Symposium on Performance Analysis of Systems Software. March 2010: 235–246. http://dx.doi.org/10.1109/ISPASS.2010.5452013.

[226] Chen X, Wu S, Chang L-W, et al. Adaptive Cache Bypass and Insertion for Many-core Accelerators [C/OL]. In Proceedings of International Workshop on Manycore Embedded Systems, in conjunction with the 41st International Symposium on Computer Architecture. New York, NY, USA, 2014: 1:1–1:8. http://doi.acm.org/10.1145/2613908.2613909.

[227] Chen X, Chang L-W, Rodrigues C I, et al. Adaptive Cache Management for Energy-efficient GPU Computing [C]. In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. 2014.

作者在学期间取得的学术成果

发表的学术论文

[1] Chen X H, Chang L W, Rodrigues C I., et al. Adaptive Cache Management for Energy-efficient GPU Computing. Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47), 2014. (EI 待收录)

[2] Chen X H, Shen L, Wang Z Y, et al. Binary Compatibility for Embedded Systems using Greedy Subgraph Mapping. Science China Information Sciences, 2014, 57(7):1-16. (SCI 收录 , 检索号 :WOS:000337259300006.)

[3] Chen X H, Wu S, Chang L W, et al. Adaptive Cache Bypass and Insertion for Many-core Accelerators. Proceedings of International Workshop on Manycore Embedded Systems, in conjunction with the 41th Annual IEEE/ACM International Symposium on Computer Architecture, 2014, 1-8. (EI 收录 , 检索号：20143017981223.)

[4] Chen X H, Chen W, Li J, et al. Characterizing Fine-Grain Parallelism on Modern Multicore Platform. Proceedings of 17th IEEE International Conference on Parallel and Distributed Systems, 2011, 941-946. (EI 收录 , 检索号：20120614751534.)

[5] Chen X H, Li J, Zheng Z, et al. Evaluating Scalability of Emerging Multithreaded Applications on Commodity Multicore Server. Proceedings of International Conference of Information Technology, Computer Engineering and Management Sciences, 2011, 332-335. (EI 收录 , 检索号：20120614754105.)

[6] Chen X H, Zheng Z, Shen L, et al. GSM: An Efficient Code Generation Algorithm for Dynamic Binary Translator. Proceedings of 4th International Symposium on Parallel Architectures, Algorithms and Programming, 2011, 231-235. (EI 收录 , 检索号：20120614745690.)

[7] Zheng Z, Chen X H, Wang Z Y et al. Performance model for OpenMP parallelized loops. Proceedings of International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE), 2011, 383-387. (EI 收录 , 检索号：12728988.)

研究成果

[1] 王志英 , 徐帆 , 沈立 , 赖鑫 , 陈微 , 陈项颢 , 等 . 多处理器平台下的动态二进制翻译方法：中国 , CN102087609A. (中国专利公开号 .)

[2] 王志英 , 赖鑫 , 沈立 , 徐帆 , 陈微 , 陈项颢 , 等 . 支持动态二进制翻译的多核体系结构 : 中国 , CN102073533A. (中国专利公开号 .)

附录 A　DVFS 对系统平均功耗的影响

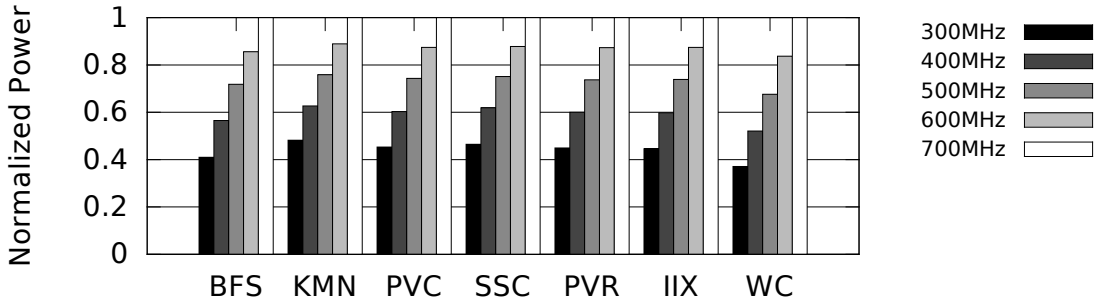

图 A.1　HCS 测试程序的平均功耗随着 SIMT 核的频率变化而变化，归一化到频率为 700MHz 的基准架构

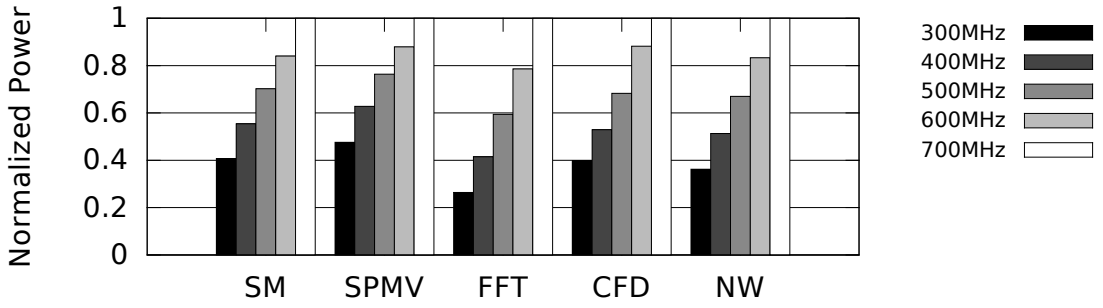

图 A.2　MCS 测试程序的平均功耗随着 SIMT 核的频率变化而变化，归一化到频率为 700MHz 的基准架构

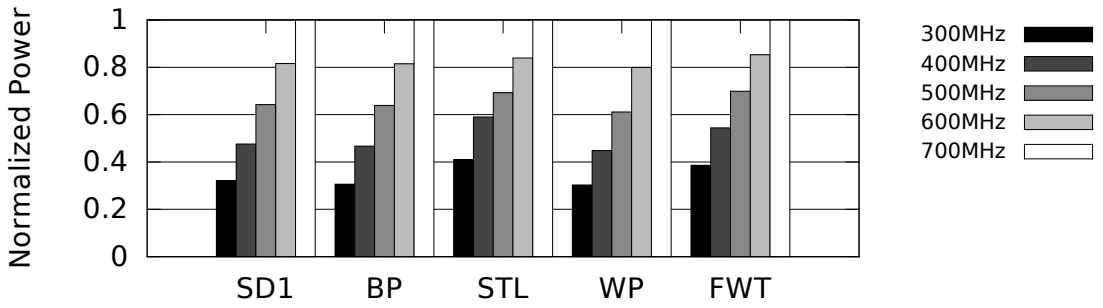

图 A.3　CI 测试程序的平均功耗随着 SIMT 核的频率变化而变化，归一化到频率为 700MHz 的基准架构

附录 B　访存暂停比例

图中显示了各个负载在执行过程中的*访存暂停比例*R_{stall}，即系统中出现访存暂停的时钟周期数占总的时钟周期数的比例。通过周期性采样，即每 10000 个时钟周期内累计访存暂停的时钟周期数 C_{stall}，然后用 C_{stall} 除以 10000，即得到 R_{stall}。

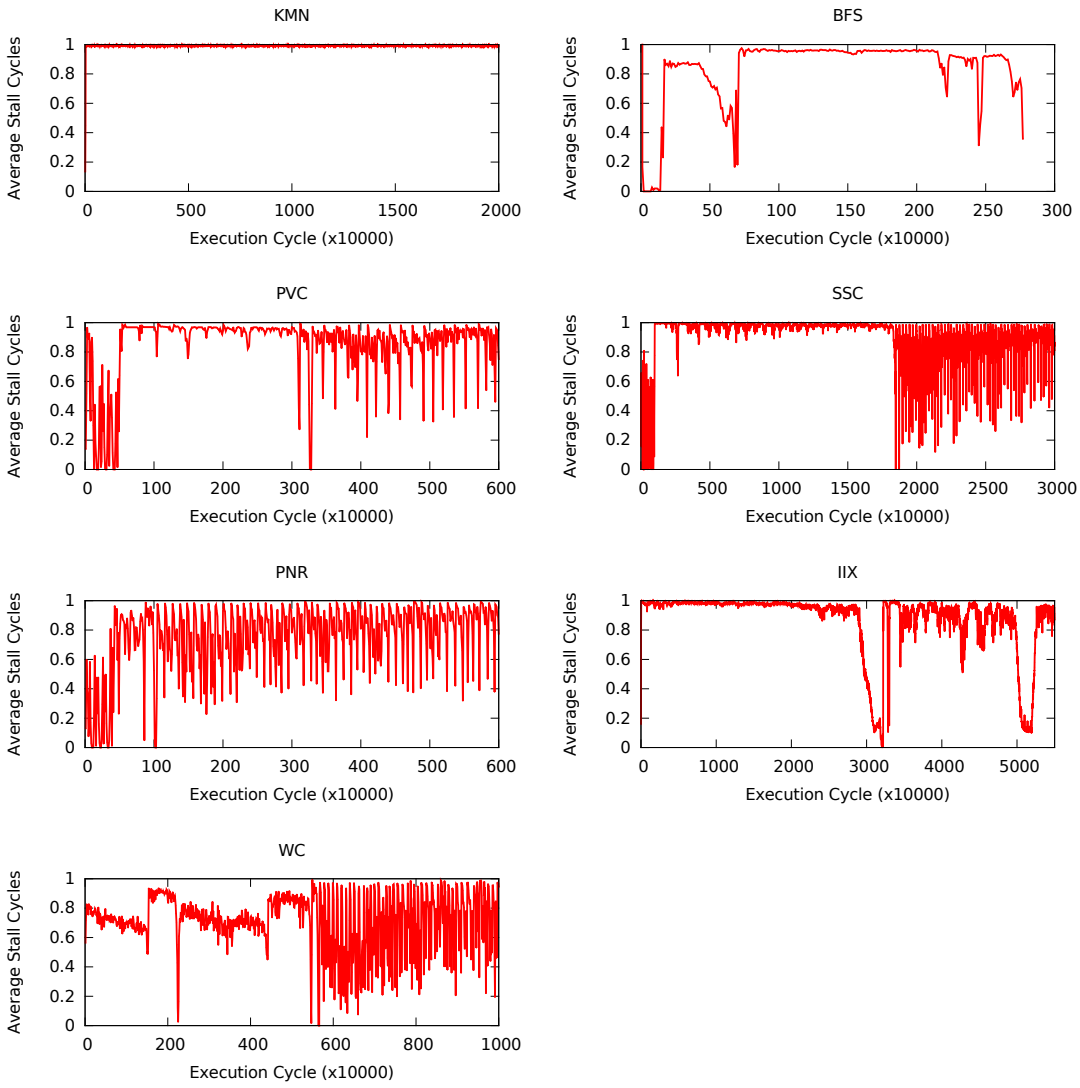

图 B.1　随着 HCS 测试程序的执行，系统中出现访存暂停的比例

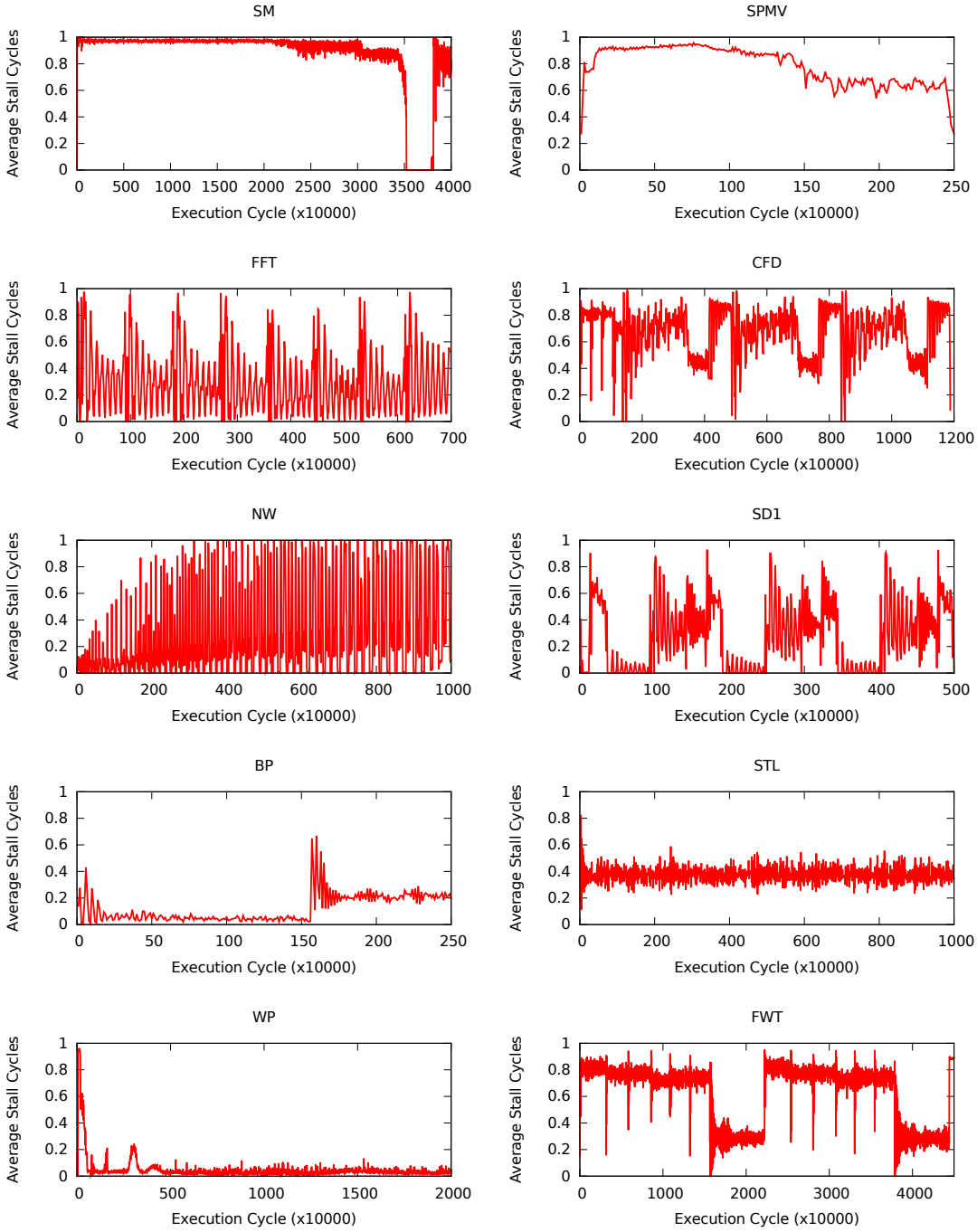

图 B.2　随着 MCS 和 CI 测试程序的执行，系统中出现访存暂停的比例