# 面向存储层次设计优化的GPU程序性能分析

唐 滔 彭 林 黄 春 杨灿群
(国防科学技术大学计算机学院 长沙 410073)

**摘 要** 图形处理器凭借着比传统CPU更高的峰值性能和能效,以及日渐成熟的软件环境,逐渐成为构建异构并行系统的最流行的加速器之一。虽然GPU依靠轻量级线程的灵活切换来隐藏访存延迟,但其超高的并发度仍然给存储系统带来了很大压力,其性能的有效发挥受访存效率的强烈影响。因此GPU程序的访存行为分析及优化一直是GPU相关领域的研究热点,但很少有工作从体系结构的角度分析存储层次的设计对性能的影响。为了更好地指导GPU存储层次的设计和访存优化,从实验的角度详细地分析了GPU各存储层次对程序性能的影响,并总结出若干指导性的优化策略,为未来类似体系结构的存储层次设计和程序优化提供建议。

**关键词** 异构系统,图形处理器,存储层次,性能分析,优化
**中图法分类号** TP302.7 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.12.001

## Performance Analysis of GPU Programs Towards Better Memory Hierarchy Design

TANG Tao PENG Lin HUANG Chun YANG Can-qun
(College of Computer Science, National University of Defense Technology, Changsha 410073, China)

**Abstract** With higher peak performance and energy efficiency than CPUs, as well as increasingly mature software environment, GPUs have become one of the most popular accelerators to build heterogeneous parallel computing systems. Generally, GPU hides memory access latency through flexible and light-weight thread switch mechanism, but its memory system faces severe pressure because of the massive parallelism and its actual performance is enormously impacted by the efficiency of memory access operations. Therefore, the analysis and optimization of GPU program's memory access behavior have always been hot research topics in GPU-related studies. However, few existing works have analyzed the impact of memory hierarchy design on performance from the view of architecture. In order to better guide the design of GPU's memory hierarchy and program optimizations, we analyzed the influence of GPU's each memory hierarchy on the program performance in detail from the view of experiment in this paper, and summarized several strategies for both the memory hierarchy design of future GPU-like architectures and program optimizations.

**Keywords** Heterogeneous system, GPU, Memory hierarchy, Performance analysis, Optimization

## 1 引言

利用CPU和加速器构建异构并行系统,通过CPU来提供通用计算能力,通过加速器来加快核心算法的执行速度,已经成为高性能计算领域的一个非常重要的发展趋势。这类系统兼具以下优势:一方面,CPU具有较强的标量计算性能和非规则应用的处理能力,使得异构系统可以适应多方面的计算需求;另一方面,加速器面向某些特定领域的应用可以提供强大的计算性能,且能效较高,使得异构系统比同构系统拥有更高的性能和能效。2016年6月发布的国际超级计算机TOP500排行榜[1]中,有94台系统采用了异构体系结构,占比18.8%,在TOP100中的占比更是达到24%。

当前技术最成熟、应用最广泛的加速器是原本用于图形计算的图形处理器(Graphics Processing Units,GPU)。GPU以强大的计算性能和日渐完善的软件环境吸引了众多来自非图形计算领域的学者的关注,在高性能计算、桌面计算甚至嵌入式计算领域都得到了广泛应用。因此,有关GPU性能优化的研究也得到了学术界和业界的持续关注。GPU的性能优化研究中最重要的一类是针对其访存行为的优化,这与GPU的体系结构特点紧密相关。

"存储墙"问题一直是困扰处理器性能发挥的难题,对GPU而言也不例外。虽然GPU通过大量并发的线程以及低开销的线程切换尽可能地隐藏访存延迟,但其特殊的多线程执行模式依然决定了访存效率对程序性能的发挥起到了至关

到稿日期:2016-11-22 返修日期:2017-02-21 本文受国家自然科学基金(61402488),教育部博士点基金(20134307120035)资助。
唐 滔(1984-),男,博士,助理研究员,主要研究方向为高性能计算和异构系统编程、编译优化,E-mail:taotang84@nudt.edu.cn;彭 林(1979-),男,博士,助理研究员,主要研究方向为高性能计算和编译优化,E-mail:penglin@nudt.edu.cn;黄 春(1973-),女,博士,研究员,主要研究方向为高性能计算和编译优化,E-mail:chunhuang@nudt.edu.cn;杨灿群(1968-),男,博士,研究员,主要研究方向为高性能计算和系统软件,E-mail:canqun@nudt.edu.cn。

重要的作用。由于同一时刻可能有大量线程同时发出访存请求，若存储层次的设计不能有效应对这种访问模式，造成这些访存请求流向片外 DRAM，则会导致大量线程因等待数据而空转，从而大幅降低 GPU 的计算效率。例如，Fatahalian 等[2]通过实验说明了 GPU 上低带宽的 Cache 设计会使得通用程序难以有效开发 GPU 强大的计算性能。从程序优化的角度看，若程序的访存行为不能很好地匹配 GPU 片上存储层次的设计，没有将潜在的数据局部性发挥出来，则会极大地浪费 GPU 性能。因此 GPU 的访存分析和优化成为了 GPU 性能优化中最受关注的问题。

“存储墙”问题的缓解需要体系结构和程序优化两方面的共同作用。一方面，需要不断改进体系结构，设计更复杂、更高效的片上存储系统来挖掘更多的数据局部性；另一方面，程序通过各种优化使得其访存的行为更加匹配处理器的存储层次设计，以提高访存的效率。然而，现有的 GPU 访存优化工作大多是从程序优化的角度去探索在已有的体系结构上如何更高效地执行程序，而没有回答体系结构应该如何优化设计的问题，因而不能用来指导未来类似体系结构的优化设计。本文正是从这个角度出发，通过大量的实验定量分析在 GPU 这种特殊的执行模式之下各种存储层次的设计参数对程序性能的影响，从而总结若干指导性的优化策略，为未来类似众核体系结构的存储层次设计和程序优化提供建议。

本文第 2 节介绍相关工作；第 3 节介绍 GPU 体系结构和执行模型；第 4 节通过实验分析 GPU 的各种存储层次（包括私有 L1 Cache、shared memory、共享 L2 Cache 和全局 DRAM）对程序性能的影响，并总结体系结构和程序优化的策略；最后总结全文。

## 2 相关工作

近年来，随着 GPU 通用计算的兴起，面向 GPU 的程序性能优化得到了广泛而深入的研究。GPU 程序的性能发挥，尤其是非规则应用的性能发挥严重依赖于其访存操作的效率，这一结论得到了学术界的广泛认可[19-26]，因此关于访存的分析和优化是 GPU 性能优化中最受关注的研究问题。

在 GPU 存储层次的性能分析方面，由于厂商在发布 GPU 时一般不会公布太多存储层次的设计细节和参数，因此大部分工作是从实验的角度构建 benchmark 来进行定量测试，并总结规律以指导程序优化。大部分对存储结构和访问延迟的测试都是基于 P-chase（指针追逐）微 benchmark[12-13]进行的。例如，Mei 等人[14]通过设计一个细粒度的 P-chase benchmark，在 Fermi 和 Kepler 系列的两款 GPU 上对存储层次的一些特性进行分析，包括全局内存、shared memory 和纹理内存，但没有分析数据 Cache，此外他们只观察了 kernel 在单个线程或单个 warp 上的访存行为，而不是完整应用的执行情况。此后，他们又将该工作延续到 Kepler 和 Maxwell GPU 上，通过微 benchmark 对 NVIDIA 的三代 GPU 进行存储层次方面的详细评测[15]，尤其是 Kepler 和 Maxwell GPU 数据 Cache 方面的评测以及 Maxwell GPU 的 shared memory 相较以往结构在体冲突下的性能优势。Candel 等人[16]指出由于 GPU 体系结构进展太快，目前的 GPU 模拟器对存储系统的微体系结构的模拟准确度不够。他们认为，若要精确模拟 GPU 的存储子系统，则需考虑 MSHR（失效状态保持寄存器）、向量访存请求的联合（coalescing）和非阻塞的 GPU 回写三方面的因素。他们在 Multi2sim 模拟器[8]的基础上进行了相应的扩充，通过实验验证了这些因素的加入可以大幅提升部分应用的模拟准确度。事实上，Multi2sim 的主要优势在于支持的平台多，但其对 GPU 存储层次的模拟并没有本文采用的 GPGPUSim 那么详细，GPGPUSim 的劣势在于只支持 NVIDIA 的 GPU 架构。Baghsorkhi 等[17]提出一种基于软件方法的性能监控技术，通过对程序执行过程中的访存轨迹进行抽样来监控存储层次的性能，并重点分析了 Fermi GPU 的 L1 和 L2 Cache 的特性。此外还有一些针对特定 GPU 体系结构的性能分析，如 NVIDIA 8800GTX[18]，GT200[19-21] 和 C2070[22]等。

上述工作与本文的主要区别在于它们是通过微 benchmark 来测试存储层次的一些未公开的参数和特征，进而指导程序优化；而本文是通过真实应用总结存储层次的参数变化对性能的影响，并总结体系结构优化设计原则。两者的出发点和目标都不同。

性能优化方面的工作主要包括面向 Chared memory 的优化、面向 Cache 的优化、面向全局存储器的优化、包含多种优化策略的综合优化框架以及针对特定应用的优化。

在面向 shared memory 的优化方面，Baskaran 等[23]使用多面体模型将仿射循环中对全局存储器的访问映射到 shared memory 中。Moazeni 等[24]和 Yang 等[25]分别将传统的寄存器分配方法应用到 shared memory 的使用上，通过图着色方法提高 shared memory 的分配效率，以应对 shared memory 的容量限制问题。除了容量限制外，shared memory 面临的最突出问题就是体冲突。Gao[26]针对不同的访问模式对体冲突进行量化评估，并通过数组 padding 和 bank 映射等方式选择合适的数据布局，以减少或消除 shared memory 的体冲突。Gou 等[27]针对 shared memory 的体冲突问题提出一种新的流水线设计，通过降低体冲突产生时的流水线停顿，结合体冲突感知的线程调度策略来提升 shared memory 的使用效率，从而改善程序性能。此外，还有一类工作是针对某些应用算法优化 shared memory 的使用，如 Silberstein 等[28]以典型的和-积算法为例研究 shared memory 的优化策略；Chen 等人[29]利用 shared memory 优化 MapReduce 应用在 GPU 上的实现等。

在 Cache 优化方面，Leverich 等[30]从存储系统的角度分析了硬件管理的一致性 Cache 和软件管理的流式存储器的区别，并指出 GPU 这类编程模型在基于 Cache 的体系结构上同样有效。Fatahalian 等[2]通过对矩阵乘法在 GPU 上的执行分析说明了 Cache 带宽对 GPU 这种依赖大规模数据并行获得高性能的处理器的重要性。Govindaraju 等[31]通过一个解析的 Cache 性能预测模型对 GPU 算法进行评测，然后指出传统的 CPU 存储优化策略可能不适用于 GPU，因此需要面向 GPU 体系结构本身研究 Cache 优化的方法。Xie 等[32]通过

分析程序访存行为评估程序的 Cache 重用性，进而对访存操作是否进入 Cache 进行动态选择，对于局部性表现不佳的访问选择旁路 Cache，这种混合的 Cache 使用策略优于数据全部进入 Cache 或全部旁路 Cache 的策略。Li 等的工作[34]与之类似，其提出了一个访存局部性监控机制，通过对部分访问使用旁路策略来提高 L1 数据 Cache 的使用效率。Jia 等[35]指出 GPU 中 Cache 的引入在某些时刻可能会使性能下降，并提出一种编译时优化方法，通过分析程序的访存行为来判定 Cache 是否会带来性能提升，从而决定是否使用 L1 数据 Cache。

在面向全局存储器的访问优化方面，Hestness 等[36]对 CPU 和 GPU 上的访存行为进行了对比分析，指出由于核心执行模式，不同于 CPU 程序，GPU 程序的性能发挥更加依赖于片外存储器的带宽，而非延迟。Wu 等人[37]提出一种新的算法来优化数组在存储器中的组织，从而减少 non-coalesced 的存储访问，提高片外存储器的带宽利用率。Jog 等[38]针对 GPU 多任务同时执行的应用场景对存储系统面临的任务冲突进行建模，并针对片外 DRAM 提出了新的访问调度机制，改善了多任务情况下存储系统的性能。Che 等[39]和 Sung 等[40]分别在 CUDA 的基础上扩充了编程接口，让程序员可以对数据在内存中的布局进行变换，并相应调整访问的索引，使得访存的局部性更好，充分利用全局存储器访问的联合机制减少访存次数。

Baskaran 等[33]提出了一个比较全面的 GPGPU 编译器优化框架来优化仿射类型的循环在 GPU 上的性能。优化策略分为 3 个方面：1)全局存储器的访问，优化的主要途径是进行访问的联合，以充分利用访存带宽，同时尽可能减少访问次数；2)优化 shared memory 的使用，主要途径是通过数组 padding 等手段避免体冲突；3)并行度的开发。由于 GPU 可以同时执行的线程数和线程对硬件寄存器的需求存在一个折中的关系，对 kernel 进行 unrolling 或 tiling 等循环变换的优化可以提高线程内局部性的开发程度，但也会增加其对寄存器资源的需求量，从而降低程序的并行度，因此该编译器建立了一个评估模型来选择最优的 unrolling 或 tiling 参数。类似的工作还有 Yang 等[41]提出的 GPU 编译框架，其也是面向 GPU 的存储层次优化和程序并行度的管理。该编译器分析 kernel 代码中的访存模式，通过访问向量化和联合等优化方法提高访存带宽的利用率；分析线程/线程块间的数据依赖关系，若发现潜在的共享数据，则将其合并以提高数据的重用性，并以此指导循环的 unrolling 和 tiling 的实现；使用软件预取技术隐藏访存开销；使用 padding 技术或对线程块进行重映射来避免存储器的体访问冲突。Jang 等[42]建立数学模型，对嵌套循环中数组访问的访存模式进行了静态分析，并通过数据变换技术将循环高效向量化，充分利用某些 GPU 中的向量访存部件，通过合理地进行数据-存储空间的映射和任务-线程的映射等提升 GPU 存储子系统的性能。此后，他们还进一步延续该工作，通过选择合适的线程块大小提高访存效率[43]。

针对特定应用优化的研究十分广泛[44-46]，这类工作通常重点探讨特定应用如何面向 GPU 体系结构进行移植以及结合应用特征的性能优化，例如算法相关的线程调度、负载平衡和数据布局优化等。

从以上介绍可以看出，现有工作中绝大多数都是从程序优化的角度去探索在已有的体系结构上如何更高效地执行程序，而没有回答体系结构应该如何优化设计的问题，因此不能用来指导未来类似体系结构的优化设计。本文正是从这个角度出发，研究访存系统的设计因素对程序性能的影响，从而对体系结构的设计提出建议。

## 3　GPU 体系结构与执行模型

不同厂商推出的 GPU 体系结构不尽相同，但总体框架基本类似。为便于介绍，本文综合了多种 GPU 体系结构并抽取其一般特征，如图 1 所示。

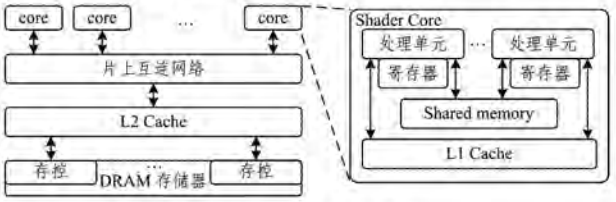


图 1　GPU 体系结构框架

从体系结构的角度看，GPU 通常包含若干个处理核心，每个核心内部包含多个处理单元，可以并发地处理若干线程。这些线程被分成组(通常称为 warp)，以组为单位调度到处理单元上执行，并且每个组在同一时刻总是被调度执行同一条指令，即它们的 PC 总是相同的，这种执行模式称为 SIMT (Single Instruction Multiple Threads)[3]。

从程序执行的角度看，在 GPU 上运行的程序通常称为 kernel，kernel 代码由大量的线程并发执行，这些线程首先被划分为粗粒度的线程块(thread block)，并以线程块为单位调度到处理核心上执行，线程块内部的线程可以实现某种程度的协同操作(例如栅栏同步)，而线程块之间则完全独立，可以以任意顺序调度执行。对线程块内部进行进一步划分，连续的若干个线程(通常是 32 个)称为一个 warp，同一 warp 内的线程以指令锁步的方式执行。

对于这种执行模式，GPU 的存储层次设计一般可以分为私有存储器、局部共享存储器和全局共享存储器。其中，每个线程可以拥有自己的私有存储空间，通常是寄存器文件；一个线程块内部的线程可以共享使用片上的共享存储器(shared memory)完成线程间通信，而同一个处理核心上的线程可以共享使用 L1 Cache；所有的线程都可以访存全局共享的 DRAM 空间并共享 L2 Cache。从控制方式上看，GPU 的片上存储层次可以分为硬件管理和软件管理两类。处理核心私有的 L1 Cache、全局共享的 L2 Cache 与传统 CPU 上的 Cache 一样，由硬件控制，对程序员透明，而 shared memory 则由程序员显式分配、使用和释放。

## 4　存储层次性能影响分析

### 4.1　实验平台和测试程序

由于实验过程中需要调整 GPU 各存储层次的设计参数

并统计 GPU 的多种执行状态，本文选择在 GPU 软件模拟器上进行实验。目前比较成熟的 GPU 软件模拟器有 Attila[4]，Qsilver[5]，Barra[6]，GPGPUSim[7] 和 Multi2sim[8] 等，其中 Attila 和 Qsilver 是面向 GPU 图形流水线体系结构的模拟器，Barra 和 GPGPUSim 是针对 GPU 通用计算体系结构的模拟器，而 Multi2sim 是针对 CPU 和 GPU 异构系统的模拟器。本文采用 GPGPUSim 作为模拟平台。GPGPUSim 是第一款针对通用计算的时钟精确 GPU 模拟器，可以详细地模拟 CUDA[9]/OpenCL[10] 程序在 GPU 上的执行情况，本实验采用 NVIDIA 的 CUDA 编程模型。选择 GPGPUSim 的主要原因在于它可以灵活地对 GPU 存储系统的各项参数进行配置，包括 L1/L2 Cache，shared memory 和 DRAM，并且可以详细输出程序执行的各种统计结果，便于程序性能分析。

GPGPUSim 模拟的 GPU 微体系结构[11] 如图 2 所示，其主要包含一系列 SIMT 核心簇（对应图 1 中的 shader core），通过片上互连网络连接到全局共享的存储系统上。片上存储层次包括寄存器文件、shared memory 和 Cache 等，片外存储系统被划分为多个 memory partition，全局共享的 LLC（Last-Level Cache）和 DRAM 分布在各个 memory partition 中。

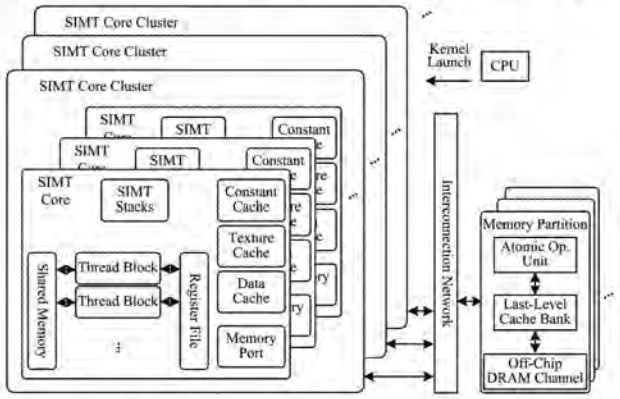


图 2 GPGPUSim 模拟的 GPU 微体系结构[1,11]

本文使用 GPGPU-Sim V3.1.0 版本，缺省配置为模拟器自带的 GTX480 配置文件。GTX480 是 NVIDIA 推出的 Fermi 架构 GPU，使用 GF100 显卡核心，其主要模拟参数如表 1 所列。

表 1 GPGPU-Sim 基本配置参数

| 参数 | 取值 |
|---|---|
| SM(SIMT cluster)个数 | 15 |
| warp 大小 | 32 线程 |
| 存储通道数/个 | 6 |
| 网络端口数/个 | 27 |
| 每个 SM 最大线程块/个 | 8 |
| L1 数据 Cache | 32 组，128 字节 Cache line，4 路组相联 |
| shared memory 容量/字节 | 49152 |
| shared memory bank 数/个 | 32 |
| L2 Cache | 64 组，128 字节 Cache line，8 路组相联 |
| dram 芯片总线宽度/字节 | 4 |
| dram 每通道芯片数/个 | 2 |
| dram burst 长度 | 8 存控周期 |

目前，GPGPUSim 缺省只提供面向 Fermi 架构 GPU 的模拟，但其目标并不是模拟某款特定的 GPU，而是提供一个基础研究平台，其灵活的可配置性允许用户调整各种参数，并在一定程度上模拟更新的架构，如 Kepler 和 Maxwell。

在测试程序方面，本文从 GPGPUSim 模拟器自带的 Benchmark 中选取了 11 个程序作为测试程序集。由于本文采用黑盒方式对程序性能进行评估，为排除因程序本身特征给评测结果带来的影响（例如程序本身是针对某种存储层次优化设计的，因而对该存储层次较为敏感），我们选取的程序涵盖了科学计算、金融、物理、生物等多个应用领域，同时包含了多种访存特征和模式，力求能反映真实应用的行为特点。测试程序的详细信息参见文献[7]。本文重点分析访存对性能的影响，表 2 列出了各测试程序动态访存指令的占比情况。

表 2 测试程序访存指令说明

| 测试程序 | 访存指令数 | Shared Mem 指令数 | 总指令数/M | 访存指令比例/% | Shared Mem 指令比例/% |
|---|---|---|---|---|---|
| AES | 131.6K | 6.645M | 27.24 | 0.48 | 24.4 |
| BFS | 2.483M | 0 | 5.568 | 44.6 | 0 |
| CP | 131.1K | 0 | 125.9 | 0.1 | 0 |
| LIB | 94.45M | 0 | 606.9 | 15.56 | 0 |
| LPS | 2.569M | 13.55M | 72.69 | 3.53 | 18.64 |
| MUM | 2.332M | 0 | 74.67 | 3.25 | 0 |
| NN | 19.05M | 0 | 38.66 | 49.28 | 0 |
| NQU | 319 | 1.075K | 1.233 | 0.026 | 8.72 |
| RAY | 6.035M | 0 | 62.28 | 9.69 | 0 |
| STO | 835.6K | 11.55M | 125.3 | 0.667 | 9.22 |
| WP | 21.02M | 0 | 361.5 | 5.81 | 0 |

表 2 中，BFS，LIB 和 NN 3 个测试程序包含多个 kernel 函数，这里仅选取模拟时间最长的 kernel 函数。

### 4.2 私有存储层次分析

核心私有的存储层次包括 L1 数据 Cache 和 shared memory，前者是硬件管理的，后者是软件管理的。首先关闭了 L1 数据 Cache（记为 L1Off），并与缺省配置进行了对比，图 3 给出了 IPC（Instructions Per Cycle）的加速比（L1Off/Default）。

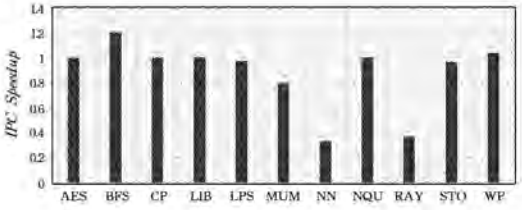


图 3 关闭 L1 数据 Cache 后 IPC 的加速比

从图 3 可以看出关闭 L1 Cache 后，IPC 的变化可以分为以下 3 种情况。

（1）持平，包括 AES，CP，LIB，LPS，NQU，STO 和 WP，关闭 L1 Cache 后对其性能影响不大。除 LIB 外，其他程序的访存指令占总指令的比例都很小（WP 和 LPS 略高，分别为 5.81%和 3.53%，其他都低于 1%），因此 L1 Cache 的变化对其性能的影响非常有限。LIB 访存指令占总指令数的 15.56%，且 L1 Cache 的失效率达到 70%，因此关闭 L1 Cache 后，其平均数据访问延迟有所降低，从 295 个周期降为 246 个周期，但 LIB 的 kernel 函数中包含大量的 sqrt，__fdividef 和 __expf 等长延迟操作，此时计算能力成为瓶颈，因此访问周期的降低对其性能影响也较小。

（2）增加，包括 BFS。在缺省配置中，BFS 的 L1 Cache 失效率达 84.3%，因此 L1 Cache 不仅没有起到挖掘局部性的

作用，反而增加了一级访问层次，因此增加了平均访存延迟。关闭 L1 Cache 后，BFS 的平均访存延迟从 397 个周期减为 296 个周期，因此程序的 IPC 有所增加。

(3)降低，包括 MUM，NN 和 RAY。关闭 L1 Cache 后，这 3 个程序的 IPC 显著下降，在缺省配置下，它们的 L1 Cache 失效率分别为 26.04%，2.28%和 26.35%。注意到，这 3 个程序都没有使用 shared memory，大部分存储访问都命中了 L1 Cache，因此关闭 L1 Cache 后 IPC 显著下降。MUM 比 NN 和 RAY 受此影响较小的原因在于 MUM 中访存指令的比例较小。

从以上分析可以看出，L1 Cache 对于程序性能的影响主要与两个因素有关：访存指令的比例和失效率。对于访存指令比例较高而失效率相对较低的程序而言，L1 Cache 可以有效减少访存延迟从而改善程序性能，而程序本身的数据局部性不好，L1 Cache 难以得到有效利用时反而会因为增加了一级存储层次而给程序性能带来负面影响。对于访存指令占比很小的程序而言，L1 Cache 存在与否对性能影响不大。

接下来，我们调整了 L1 数据 Cache 的容量，分别在缺省配置的基础上增加了 2，3，4，5 倍。GPGPU-Sim 的 L1 Cache 采用组相联设计，通过增加组的个数调整容量。从图 4 可以看出，Cache 容量的变化对所有程序的 IPC 几乎没有任何影响。这说明对于 GPU 而言，程序的性能对 L1 数据 Cache 的容量并不敏感；对于缺省配置的 GPU 内核而言，16kB/核的 L1 Cache 已经足够，Cache 失效率高的主要原因在于程序的局部性不好，而不是 Cache 的容量不够。

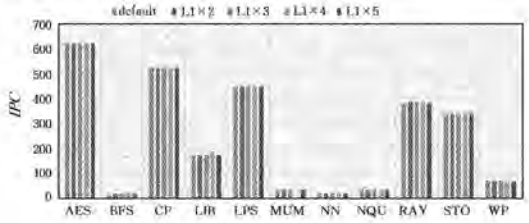


图 4　不同 L1 数据 Cache 容量下各程序的 IPC

继续调节 shared memory 的容量和带宽，分析其对性能的影响。

图 5 给出了调节 shared memory 容量后各程序的 IPC 变化情况。在选取的测试用例中，仅有 4 个程序使用了 shared memory。分别将 shared memory 的容量在缺省配置的基础上增加了 2，3，4，5 倍。如图 5 所示，AES，LPS 和 NQU 3 个程序的 IPC 随着 shared memory 容量的变化没有任何改变，只有 STO 在 shared memory 容量增加 2 倍时 IPC 有所增长，然后就不再改变。为分析具体原因，表 3 列出了各程序执行时每个 SM 上同时执行的线程块个数。

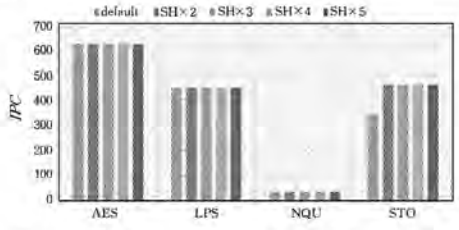


图 5　不同 shared memory 容量下各程序的 IPC

表 3　不同 shared memory 容量下每个 SM 同时执行的线程块个数

| 测试程序 | AES | LPS | NQU | STO |
|---|---|---|---|---|
| default | 6 (threads) | 8 (cta_limit) | 3(shmem) | 3(shmem) |
| x2 | 6 (threads) | 8 (cta_limit) | 6(shmem) | 5(regs) |
| x3 | 6 (threads) | 8 (cta_limit) | 8 (cta_limit) | 5(regs) |
| x4 | 6 (threads) | 8 (cta_limit) | 8 (cta_limit) | 5 (regs) |
| x5 | 6 (threads) | 8 (cta_limit) | 8 (cta_limit) | 5(regs) |

表 3 中，括号中数据为决定该个数的因素，包括以下 4 种：

(1)“threads”，表示受到 SM 上同时允许执行的最大线程数的限制。GTX280 体系结构中默认规定单个 SM 上同时执行的最大线程数为 1536，因此若线程块中包含的线程数越多，则同时可以调度执行的线程块个数就越少。

(2)“cta_limit”，表示 SM 的参数限制。GTX280 体系结构中默认规定单个 SM 上同时执行的最大线程块个数为 8，因此受此因素限制的同时活跃线程块数都是 8。

(3)“shmem”，表示线程块个数受到 shared memory 大小限制。SM 中 shared memory 容量有限，若每个线程块所申请的 shared memory 越大，则同时可以执行的线程块个数就越小。

(4)“regs”，表示线程块个数受到寄存器个数的限制。其原理与 shared memory 相同，每个线程都需要占用一定数目的寄存器，而 SM 中的寄存器个数有限，因此寄存器也可能成为限制同时活跃线程块个数的因素。

从表 3 可以看出，AES 和 LPS 两个程序在所有配置中的同时活跃线程块数始终不变，限制因素分别为 threads 和 cta_limit，与 shared memory 无关，因此在图 5 中调节 shared memory 的容量时，这两个程序的 IPC 没有任何变化。对于 NQU，从表 3 中可以看出，缺省配置和 shared memory 容量增大一倍时，单个 SM 同时活跃线程块个数都受到 shared memory容量限制，shared memory 的容量增大一倍时，同时活跃线程块个数从 3 变为 6，继续增大时，同时活跃线程块个数变为 8，因受到 cta_limit 的限制而无法继续增长。但是从图 5 中发现，即使同时活跃的线程块个数从 3 变为 6 再变为最终的 8，其 IPC 也没有增长，相反还略有下降，分析模拟器的输出结果可以得知原因，SM 上同时活跃的线程块数增加时，其 L1 指令 Cache 的失效率显著增加，导致性能下降，这就抵消了多个线程块同时运行带来的访存延迟隐藏的优势。对于 STO，当 shared memory 的容量增加一倍时，同时活跃线程块个数从 3 变为 5，限制因素从 shmem 变为 regs，之后就不再变化，对应于图 5，IPC 也在 shared memory 容量增加一倍时有所增加，之后没有改变。这是由于更多的线程块在 SM 上同时活跃，因此带来更多的访存延迟隐藏的机会，提升了程序的性能。

从这一组实验可以看出，shared memory 的容量提升对程序性能的影响与程序的行为相关，大部分程序因受到硬件参数的限制而无法在一个 SM 上同时运行更多的线程块，仅少数对 shared memory 需求非常大的程序在初始状态下的同时活跃线程块数受限于 shared memory，在容量提升后性能有小幅提升。

接下来调整 shared memory 的 bank 个数，以改变 shared

memory 的访问带宽。

shared memory 是按照多 bank 组织的片上存储层次，不同的线程可以并发地访问不同的 bank，如果两个线程的访问落入同一个 bank 中，就会造成体冲突(bank conflict)。在实验中，在缺省配置的基础上(bank=32)分别将 bank 个数增加 2,3,4,5 倍，结果如图 6 所示。从数据上看，AES 和 NQU 的 IPC 有小幅波动，而 LPS 和 STO 保持不变。通过分析模拟器的输出结果列出了各个程序 shared memory 中体冲突造成的停顿周期(见表 4)。

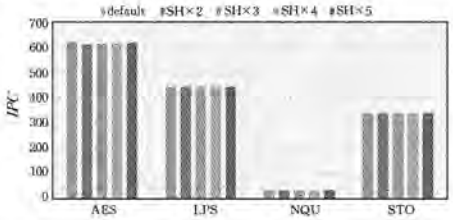


图 6 不同 shared memory bank 个数下各程序的 IPC

表 4 不同 shared memory bank 个数下因体冲突造成的停顿周期

| 测试程序 | default | x2 | x3 | x4 | x5 |
|---|---|---|---|---|---|
| AES | 73648 | 43276 | 32631 | 24266 | 20933 |
| LPS | 0 | 0 | 0 | 0 | 0 |
| NQU | 900 | 373 | 158 | 236 | 380 |
| STO | 0 | 0 | 0 | 0 | 0 |

从表 4 可以看出，AES 和 NQU 存在一定的 shared memory体冲突；增加 bank 个数后，体冲突引发的停顿周期有所下降，但并不严格随着 bank 个数的增加而递减，例如在 NQU 中，当 bank 个数增加至缺省配置的 4 倍和 5 倍时，体冲突造成的停顿周期又有所回升。事实上，在缺省配置中，shared memory 的 bank 个数为 32，已经和一个 warp 的线程数相当，因此理想情况下，如果数据在 shared memory 中合理分布，就不会产生体冲突。在实际程序执行过程中，同一个线程访问的若干数据被分配在同一个 bank 中，从而造成了体冲突。调整 bank 个数实际上改变了数据在各个 bank 中的分布情况，因此不能确定产生的影响是正面的还是负面的，最佳的 bank 个数应该与具体程序访问的方式有关。在 AES 中，虽然体冲突停顿周期随着 bank 个数的增加而明显减少，但从图 6 中可以看出，程序性能反而在 bank 个数增加一倍时略有下降。通过分析模拟器的输出结果可以发现，增加 bank 个数后，虽然体冲突造成的停顿周期减少，但由于非连续(non-coalescing)的全局存储器访问造成的停顿周期增加(这是由于改变了 SM 对 shared memory 的访问行为)，从而间接影响了 SM 对全局存储器的访问行为，而这种影响是难以预测和分析的。

相反，在缺省配置的基础上分别将 bank 个数减少为原来的 3/4,1/2 和 1/4，测试结果如图 7 所示。可以看出，相对于缺省配置，将 shared memory 的 bank 个数减少 1/4 后在所有程序中都产生了大量的体冲突，程序的 IPC 也随之显著下降。与之相对的是，bank 个数在从原来的 3/4 进一步减少至 1/2 时，体冲突停顿周期仅有小幅增长，而在减少至原来的 1/4 时，又显著增加。产生这种不规律性的原因与前文的分析一致，bank 个数对 shared memory 访问的影响来自于它改变了数据在 bank 中的分布方式，但分布方式并不一定直接导致体冲突，而是依赖于具体程序的访问特征。

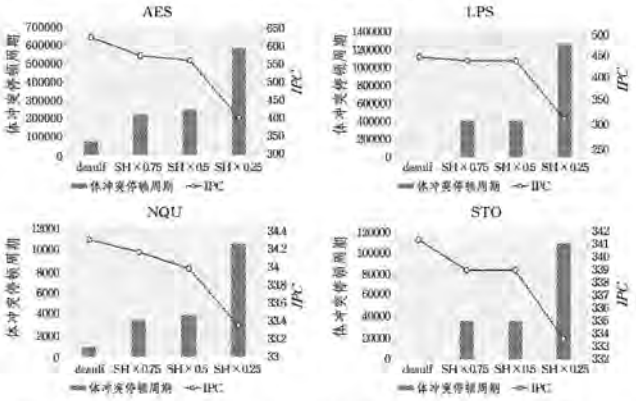


图 7 减少 shared memory bank 个数后各程序的 IPC 和体冲突造成的停顿周期

通过这组实验可以看出，缺省配置中 bank 个数和 warp 中线程个数相当，这对于大部分程序而言已经足够，在此基础上单纯增加 bank 个数对性能优化没有太大的意义，而减少 bank 个数则会给性能带来较大影响，因为当 bank 个数小于 warp 线程数时，即使数据理想分布，一个 warp 内的所有线程同时访问也必然会产生体冲突。

### 4.3 共享存储层次分析

共享存储层次包括 L2 Cache 和全局 DRAM。

与 L1 数据 Cache 类似，首先关闭了 L2 Cache(记为 L2Off)，并与缺省配置进行了对比，图 8 给出了 IPC 和平均存储器访问延迟(AvgMemFetchLat)的加速比(L2Off/Default)。

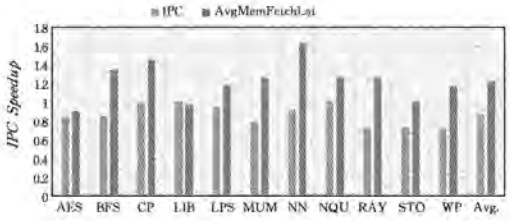


图 8 关闭 L2 Cache 后 IPC 和平均存储器访问延迟的加速比

从图 8 中可以看出，关闭 L2 Cache 后各测试程序的 IPC 加速比主要分为两种情况：

(1)持平，包括 CP，LIB 和 NQU。从测试结果发现，缺省配置下 CP 和 NQU 两个程序的 L2 Cache 失效率都不高，L2 Cache 有效减少了访问全局存储器的次数，因此关掉 L2 Cache 后它们的平均存储器访问延迟增大，但由于 CP 和 NQU 中访存指令的比例很低(小于 0.1%)，因此其对 IPC 的影响不大。LIB 的访存比例为 15%左右，但 L2 Cache 的失效率接近 70%，即大部分访问没有命中 L2 Cache，所以关闭后由于减少了一级存储层次，平均存储器访问延迟还略微降低。综合来看，其 IPC 基本不变。

(2)降低，其他所有程序。在这些程序中，除了 AES 和 STO 外，平均存储器访问延迟都有显著的增加(增幅都在 17%以上)，因此它们的 IPC 有所降低是正常的。AES 的情况比较特殊，它的平均存储器访问延迟不但没有增加，反而减

少,原因与 LIB 一样,其 Cache 失效率较高,减少 L2 Cache 这一级存储层次后,存储器的访问延迟反而减少。但是,AES 的访存指令比例很小,不到 0.5%,因此与 CP 和 NQU 两个程序一样,存储器访问延迟的减小不会对性能产生太大影响。相反地,AES 中包含大量纹理内存(Texture Memory)的访问指令,而纹理内存的数据访问也要经过 L2 Cache,关闭 L2 Cache 后,对这部分的数据访问产生了较大影响,因此其整体性能有所下降。STO 的情况比较特殊,它的平均访存延迟在关闭 L2 Cache 后基本没有变化,而 IPC 却显著下降。分析模拟器的输出发现,关闭 L2 Cache 后,流向 DRAM 的读、写请求个数增加了约 5 倍,但增加的访存主要集中在少数几个体(bank)中,而这些体的平均访问延迟却显著低于缺省配置中的平均访问延迟,原因在于这几个体中的访问失效率较高,禁用 Cache 后由于减少了存储层次,访问延迟反而降低。

L2 Cache 可以缓存全局数据空间、常数空间、纹理空间和指令空间的访问请求,通过这组实验可以看出 L2 Cache 对于性能有比较明显的贡献,如果关闭 L2 Cache,那么 DRAM 的平均访问延迟上升了 22%,性能平均下降了 14%。

接下来调整了 L2 Cache 的容量,分别将其增加了 2,3,4,5 倍。注意到,GPGPU-Sim 的 L2 Cache 是作为存控(MC)的一部分实现的,我们没有调整存控的个数,而是直接通过调整 L2 Cache 的组数来调整其大小,从而保持 DRAM 的访存带宽不变。

图 9 给出了调整 L2 Cache 容量后各程序 IPC 的变化情况。可以看出,除个别程序有小幅波动外,绝大部分程序的 IPC 基本没有变化。通过观察模拟器的输出结果发现,L2 Cache 的失效率基本上没有随着其容量的变化而变化,这与私有的 L1 数据 Cache 是一致的,即大部分程序对 L2 Cache 的容量不敏感。

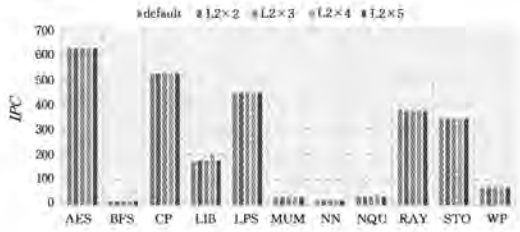


图 9　不同 L2 Cache 容量下各程序的 IPC

最后,调节全局 DRAM 的访问带宽。全局 DRAM 的访问带宽由如下几个参数决定:访存通道数、每通道内芯片个数、每个芯片的总线宽度。全局带宽的计算公式为:

DRAM 带宽=访存通道数 * 每通道芯片数 * 每芯片总线宽度 * DRAM 频率 * 2

分别调整前 3 个参数,观测程序性能的变化情况。

在 GTX480 默认设置中,总线宽度为 4 个字节,将其调整为原来的 2,3,4,5 倍,图 10 给出了调整后的 IPC 变化情况。从图 10 可以看出,只有 AES,LIB,LPS,MUM 和 RAY 等程序的 IPC 在总线宽度增加一倍时有少量增加,之后即使再增加总线宽度,IPC 均保持不变;而其他程序的 IPC 自始至终基本不变。

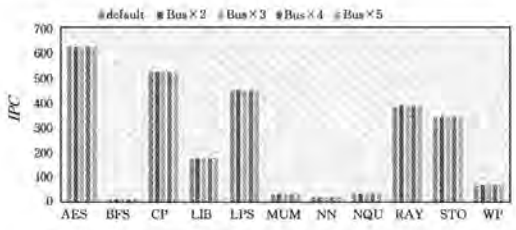


图 10　不同 DRAM 总线宽度下各程序的 IPC

为更进一步分析原因,图 11 给出了调整总线宽度后各程序的平均访存延迟。

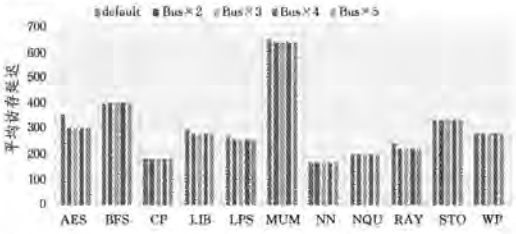


图 11　不同 DRAM 总线宽度下各程序的平均访存延迟

从图 11 可以看出,在总线宽度由 4 字节增大到 8 字节时,上述几个程序的平均访存延迟略有降低,之后不再变化,这与 IPC 的变化规律一致。我们进一步统计了 DRAM 的带宽利用率 *bw_util*,计算公式为:

$$bw\_util = 2 * (n\_rd + n\_write) / n\_cmd$$

其中,*n_rd* 和 *n_write* 分别为存控发出的读请求和写请求次数,*n_cmd* 为存控经历的总周期数,由于每个读/写请求需要占用 2 个存控的命令周期,因此带宽利用率是总请求数的 2 倍除以总周期数。

图 12 给出了各程序的 *bw_util* 加速比随着总线带宽的变化情况(加速比为调整后的 *bw_util* 除以缺省配置下的 *bw_util*,即归一化到缺省配置下的 *bw_util*)。

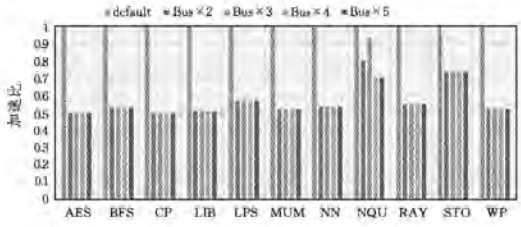


图 12　不同 DRAM 总线宽度下各程序的 *bw_util* 的加速比

从图 12 可以看出,除了 NQU 和 STO,绝大部分程序在总线带宽增大一倍时,*bw_util* 降为原来的一半左右,并且在总线带宽继续增大时其不再变化。这是由于总线带宽增加一倍后,存控发出的请求次数降低了将近一半,因此 *bw_util* 显著下降。继续增加带宽时,请求次数基本不变,原因在于 GTX480 默认设置下,DRAM 的 burst 宽度为 8 个存控周期,因此一次传输请求可以传输 8 * 2 个(DDR)总线数据,即 16 个总线的数据,当总线宽度为 8 字节时,一次请求可以访问 16 * 8=128 字节的数据,正好与 L2 Cache 的宽度(128 字节)相当。换言之,当总线宽度为 4 时,每次 L2 Cache 失效会产生 2 次访存请求;而总线宽度增加一倍时,只会产生一次访存请求,请求数目降低一半;而继续增加总线宽度时,每次 L2 Cache 失效仍然会产生一次访存请求,不会再继续减少。

NQU 和 STO 中 L2 Cache 写失效的次数较多，GPGPUSim 中 L2 Cache 写失效时采用 write no-allocate 策略，即直接将数据写入 DRAM，不在 L2 Cache 中缓存，因此每次写只会产生一次访存请求；当总线宽度增加时，这部分写请求的次数也不会缩减。因此，从总体上看，访存请求的次数高于原始的 1/2。

虽然增加总线宽度后访存请求的次数显著减少，但平均访存延迟以及 IPC 的变化却很小，基本上可以忽略，这说明 L2 Cache 对计算部件流出访存请求起到了很好的缓冲作用，存控流出的 DRAM 访问请求次数即便降为原来的一半也没有显著影响性能。通过这一组实验可以得出结论：在缺省配置下，总线宽度为 4 且 burst 长度为 8 可以很好地应对 L2 Cache 流出的 128 字节长度的访存请求，增加总线宽度对性能影响不大。

接下来继续调整每通道芯片数，在缺省配置的基础上分别将其增加 2，3，4，5 倍，IPC、平均访存延迟和 *bw_util* 的结果与调整总线宽度时的结果基本一致，即增加每个通道内的芯片数的效果和增加每芯片内总线宽度的效果是一致的，当芯片数增加一倍时，访存请求数显著减少，之后不再变化。原因在于在 GPGPUSim 中，全局存储空间在各个存储通道内按照 256 字节交叉存储，这 256 字节再分布到存储通道内的各个芯片上。在缺省配置下，256 字节分配到 2 个芯片上，每个芯片的长度是 128 字节，即一个 L2 Cache 块的长度，如图 13 所示。根据上面的分析，4 字节的总线宽度和 8 周期的 burst 长度需要存控流出 2 次访存请求才能满足一次 L2 Cache 失效请求。

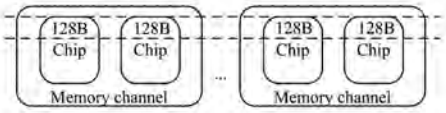


图 13 缺省配置下数据在存储通道内的分布情况

当芯片数由 2 增加到 4 时，每个芯片分配 64 字节，即一个 Cache 块的数据分布在两个芯片上，如图 14 所示。它们可以并发访问，因此一个 L2 Cache 块由一次访存请求就可以全部读取，此时的效果与总线宽度增加为 8 字节时的效果是一致的。

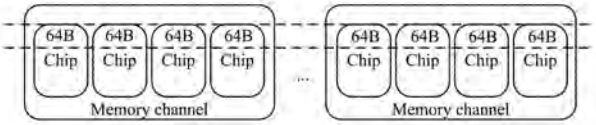


图 14 每通道芯片数增加一倍时数据在存储通道内的分布情况

最后，调整了存储通道数，分别将其增加了 2，3，4，5 倍，各程序的 IPC 如图 15 所示。

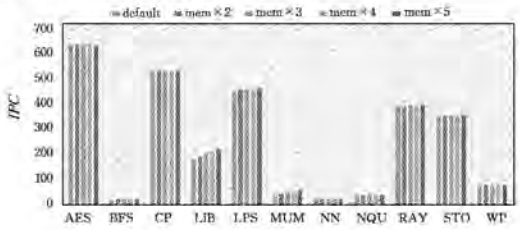


图 15 不同访存通道数下各程序的 IPC

从图 15 可以看出，程序的 IPC 变化情况可以分为两类，除了 BFS，LIB 和 MUM 外，其他基本保持不变。调整存储通道数后，L2 Cache 容量、访问带宽以及全局 DRAM 访问带宽都发生变化，此时上述 3 个程序的性能得到显著提高，说明它们是访存受限的程序；其他程序的 IPC 基本不变，原因可以分为两种：1）它们是计算受限的程序，增大访存带宽对其性能没有帮助；2）访存指令的比例太低，访存性能的改善对性能影响不大，这一点要结合 2.4 节的可扩展性实验结果进行分析。

从本节的实验结果来看，影响访存带宽的 3 个参数中，总线宽度和每通道芯片数对程序性能的影响不大，而访存通道数对部分程序的性能有显著影响。在 GPGPU-Sim 架构中，SM 处理器簇和存控通过片上互连网络连接在一起，网络的端口数为处理器簇数和存控数之和。我们调整前两个参数后，网络的端口数不变，虽然理论上访存带宽有增加，但网络的带宽是固定的，增加的访存带宽并没有真正地被利用起来。当调整存储通道数后，网络的端口数随之改变，网络的带宽也随着访存带宽的提升而同步提升。此时，访存能力能否得到充分挖掘取决于程序的特征：是访存带宽受限还是计算受限。对于访存带宽受限的程序，增加存储通道个数可以显著改善其性能，对于计算受限的程序则影响较小。

### 4.4 可扩展性分析

本节调整 SM 的个数以观察程序性能的变化情况。

GTX480 的缺省配置中包含 15 个 SM，我们依次将其增加到 20，25，30 和 35 个，程序的 IPC 变化情况如图 16 所示。

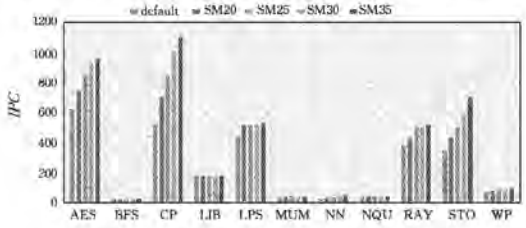


图 16 不同 SM 个数下各程序的 IPC

从图 16 中可以看出，除 BFS，LIB，MUM 和 NQU 外，其他程序的 IPC 随着 SM 个数的增加显著增长。对于 BFS，LIB 和 MUM 3 个程序，通过上一节的分析可以得知，它们是访存受限的程序，因此增加计算资源对其性能的影响不大，即在访存受限的前提下，这 3 个程序的可扩展性受到了很大的影响。NQU 的情况比较特殊，从图 15 中可以看出，增加存储通道数时，其性能基本没有发生变化，这是由于在 NQU 中访存指令的比例非常小，只有 0.026%，在测试集的所有程序中是最低的，因此它可以视为计算受限的程序，但增加 SM 个数时，其性能也没有变化。分析模拟器的执行过程发现，该程序执行过程中出现了负载不均衡的现象，0 号线程块的执行时间过长，远多于其他线程块的执行时间，成为整个程序执行的瓶颈，完全抵消了增加 SM 个数带来的性能提升。换言之，负载的不均衡性导致该程序的可扩展性很差。

从这一组实验可以看出，增加 SM 个数可以明显提升大部分程序的性能。主要原因在于 GPU 程序的并行性很好，线程块之间没有依赖，可扩展性较好，因此增加 SM 个数可以显著减少计算时间；而对于少数存储受限的程序，其性能受限于片上网络和访存性能，增加 SM 个数无法有效提升性能。

### 4.5 小结与讨论

通过大量的实验数据分析可以发现，对于 GPU 的私有

存储层次,在没有经过任何额外优化技术的前提下,使用 shared memory 的效果优于使用 Cache 的效果,且大部分程序对 shared memory 的容量较为敏感,而对 Cache 的容量不敏感。此外,私有存储层次不影响性能的可扩展性;而对于共享存储层次,带宽是影响可扩展性的关键因素。

因此,从可扩展性的角度看,当程序的访存行为一定时,多核/众核体系结构的存储层次优化设计应该重点关注如何提高私有存储层次的命中率和增加共享存储层次的带宽。对于 Cache 结构而言,提高私有存储层次的命中率可以通过提高 Cache 容量来减少容量失效,通过采用更高相联度的设计来减少冲突失效;而对于 shared memory 而言,由于它是由软件控制的,失效率为 0,空间利用率更高,因此提高容量即可有效增加私有存储层次的命中次数。从本文的实验中也可以看出,增加 shared memory 容量比增加私有 Cache 容量可以得到更高的加速比。在不考虑软件开销的前提下,私有存储层次采用软件管理的 shared memory 优于硬件管理的 Cache。对于共享存储层次而言,应该通过多体、多端口等方式来提高带宽,而不是单纯追求存储层次的高容量。对于一般的不带旁路的存储层次设计,越靠近处理器的共享存储层次的带宽越重要,因为远离处理器的存储层次的数据都需要通过它们进入处理器,相对而言,它们对带宽的需求更强烈。

当给定体系结构时,多核/众核上并行程序的优化应该从两个角度展开:首先应当针对核心内运行的程序通过局部性优化技术尽量减少私有存储层次的失效,减少核心对共享存储层次的访问以缓解对共享存储层次的带宽压力;其次,通常情况下越接近处理器的共享存储层次的带宽越高,因此程序优化的第二个方面应通过任务划分、数据划分等技术增加核间程序的数据重用,减小程序在共享存储层次上产生的工作集,使得数据访问向更高层的共享存储层次倾斜,以充分利用存储层次提供的带宽。对于本文研究的 GPU 程序而言,核间任务和数据的划分往往是程序员给定的(线程块的划分),因此程序优化的重点应放在核内串行部分的局部性优化上。

**结束语**　本文通过实验分析了 GPU 上各种存储层次的设计对程序性能的影响,总结了类似体系结构的存储层次设计以及面向访存的程序性能优化应遵循的一般原则。在未来进行类 GPU 众核体系结构设计时,大容量的一级 shared memory 和高带宽的 L2 Cache 应该是一个比较合理的选择。从程序优化的角度看,应该重点从优化核内程序的局部性来提高私有存储层次的命中率和增加核间数据重用以减少程序在共享存储层次上产生的工作集两个方面展开。对于 GPU 这种特殊的体系结构,由于核间的任务和数据划分方式比较固定,优化的重点应放在核内串行部分的局部性优化上。

## 参 考 文 献


[1] Top500[EB/OL]. http://www.top500.org/lists/2016/06, 2016.

[2] FATAHALIAN K, SUGERMAN J, HANRAHAN P. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication[C]// Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. New York, NY, USA, 2004: 133-137.

[3] LINDHOLM E, NICKOLLS J, OBERMAN S, et al. NVIDIA Tesla: A Unified Graphics and Computing Architecture[J]. IEEE Micro, 2008, 28(2): 39-55.

[4] BARRIO V M D, GONZALEZ C, ROCA J, et al. ATTILA: a cycle-level execution-driven simulator for modern GPU architectures[C]// IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2006). Austin, Texas, USA, 2006: 231-241.

[5] SHEAFFER J W, LUEBKE D, SKADRON K. A flexible simulation framework for graphics architectures[C]// ACM Siggraph/Eurographics Symposium on Graphics Hardware 2004. Grenoble, France, 2004: 85-94.

[6] COLLANGE S, DAUMAS M, DEFOUR D, et al. Barra: A Parallel Functional Simulator for GPGPU[C]// 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. IEEE, 2010: 351-360.

[7] BAKHODA A, YUAN G L, FUNG W W L, et al. Analyzing CUDA workloads using a detailed GPU simulator[C]// IEEE International Symposium on Performance Analysis of Systems and Software. IEEE, 2009: 163-174.

[8] UBAL R, JANG B, MISTRY P, et al. Multi2Sim: a simulation framework for CPU-GPU computing[C]// International Conference on Parallel Architectures & Compilation Techniques. 2012: 335-344.

[9] NVIDIA Corp. NVIDIA CUDA C Programming Guide[J]. Nvidia Corporation, 2011, 120(18): 8.

[10] MUNSHI A. The opencl specification[C]// 2009 IEEE Hot Chips Symposium (HCS). IEEE, 2009: 1-314.

[11] AAMODT T M, UNG W W L, HETHERINGTON T H. GPGPU-Sim 3. x Manual, Revision 1. 2. [EB/OL]. http://gpgpu-sim.org/manual/index.php/Main_Page.

[12] SAAVEDRA-BARRERA R H. CPU performance evaluation and execution time prediction using Narrow spectrum benchmarking[D]. University of California, Berkeley, 1992.

[13] SMITH A J, SAAVEDRA R H. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes[J]. IEEE Transactions on Computers, 1995, 44(10): 1223-1235.

[14] MEI X, ZHAO K, LIU C, et al. Benchmarking the Memory Hierarchy of Modern GPUs[M]// Network and Parallel Computing. Springer Berlin Heidelberg, 2014: 144-156.

[15] MEI X, CHU X. Dissecting GPU Memory Hierarchy through Microbenchmarking[EB/OL]. https://arxiv.org/abs/1509.02308.

[16] CANDEL F, PETIT S, SAHUQUILLO J, et al. Accurately modeling the GPU memory subsystem[C]// International Conference on High Performance Computing & Simulation. IEEE, 2015: 179-186.

[17] BAGHSORKHI S S, GELADO I, DELAHAYE M, et al. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors[J]. Acm Sigplan Notices, 2012, 47(8): 23-34.

[18] VOLKOV V, DEMMEL J W. Benchmarking GPUs to tune dense linear algebra[C]// International Conference for High Performance Computing, Networking, Storage and Analysis. SC, 2008: 1-11.

[19] PAPADOPOULOU M M, SADOOGHI-ALVANDI M, WONG H. Micro-benchmarking the GT200 GPU[R]. Computer Group, ECE, University of Toronto, 2009.

[20] WONG H, PAPADOPOULOU M M, SADOOGHI-ALVANDI M, et al. Demystifying GPU microarchitecture through microbenchmarking[C]// IEEE International Symposium on Performance Analysis of Systems & Software. IEEE, 2010: 235-246.

[21] ZHANG Y, OWENS J D. A quantitative performance analysis model for GPU architectures[C]// Proc. of IEEE 17th International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2011: 382-393.

[22] MELTZER R, ZENG C, CECKA C. Micro-benchmarking the C2070[C]// Poster of GPU Technology Conference. San Jose, California, 2013.

[23] BASKARAN M M, BONDHUGULA U, KRISHNAMOORTHY S, et al. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories[C]// Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2008: 1-10.

[24] MOAZENI M, BUI A, SARRAFZADEH M. A memory optimization technique for software-managed scratchpad memory in GPUs[C]// IEEE Symposium on Application Specific Processors (Sasp 2009). San Francisco, CA, USA, 2009: 43-49.

[25] YANG X, WANG L, XUE J, et al. Comparability graph coloring for optimizing utilization of stream register files in stream processors[C]// Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2009: 111-120.

[26] GAO S. Improving GPU Shared Memory Access Efficiency[D]. University of Tennessee, 2014.

[27] GOU C, GAYDADJIEV G N. Addressing GPU on-chip shared memory bank conflicts using elastic pipeline[J]. International Journal of Parallel Programming, 2013, 41(3): 400-429.

[28] SILBERSTEIN M, SCHUSTER A, GEIGER D, et al. Efficient computation of sum-products on GPUs through software-managed cache[C]// Proceedings of the 22nd Annual International Conference on Supercomputing. New York, NY, USA, 2008: 309-318.

[29] CHEN L, AGRAWAL G. Optimizing MapReduce for GPUs with effective shared memory usage[C]// Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing. ACM, 2012: 199-210.

[30] LEVERICH J, ARAKIDA H, SOLOMATNIKOV A, et al. Comparing memory systems for chip multiprocessors [C]// Proceedings of the 34th Annual International Symposium on Computer Architecture. New York, NY, USA, 2007: 358-368.

[31] GOVINDARAJU N K, LARSEN S, GRAY J, et al. A memory model for scientific algorithms on graphics processors[C]// Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. New York, NY, USA, 2006.

[32] XIE X, LIANG Y, SUN G, et al. An Efficient Compiler Framework for Cache Bypassing on GPUs[C]// IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers. 2013: 516-523.

[33] BASKARAN M M, BONDHUGULA U, KRISHNAMOORTHY S, et al. A compiler framework for optimization of affine loop nests for gpgpus[C]// Proceedings of the 22nd Annual International Conference on Supercomputing. New York, NY, USA, 2008: 225-234.

[34] LI C, SONG S L, DAI H, et al. Locality-driven dynamic GPU cache bypassing[C]// Proceedings of the 29th ACM on International Conference on Supercomputing. ACM, 2015: 67-77.

[35] JIA W, SHAW K A, MARTONOSI M. Characterizing and improving the use of demand-fetched caches in GPUs[C]// Proc. of the 26th ACM International Conference on Supercomputing. ACM, 2012: 15-24.

[36] HESTNESS J, KECKLER S W, WOOD D A. A comparative analysis of microarchitecture effects on CPU and GPU memory system behavior[C]// 2014 IEEE International Symposium on Workload Characterization (IISWC). Raleigh, NC, 2014: 150-160.

[37] WU B, ZHAO Z, ZHANG E Z, et al. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU[J]. ACM Sigplan Notices, 2013, 48(8): 57-68.

[38] JOG A, KAYIRAN O, KESTEN T, et al. Anatomy of GPU Memory System for Multi-Application Execution[C]// International Symposium on Memory Systems (MEMSYS). 2015: 223-234.

[39] CHE S, SHEAFFER J W, SKADRON K. DYMAXION: Optimizing memory access patterns for heterogeneous systems[C]// Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011: 1-11.

[40] SUNG I, ANSSARI N, STRATTON J A, et al. Data Layout Transformation Exploiting Memory-Level Parallelism in Structured Grid Many-Core Applications[J]. International Journal of Parallel Programming, 2012, 40(1): 4-24.

[41] YANG Y, XIANG P, KONG J, et al. A GPGPU compiler for memory optimization and parallelism management[C]// Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, 2010: 86-97.

[42] JANG B, SCHAA D, MISTRY P, et al. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures[J]. IEEE Transactions on Parallel & Distributed Systems, 2011, 22(1): 105-118.

[43] JANG B, CHOI M, KIM K K. Algorithmic GPGPU memory optimization[J]. Journal of Semiconductor Technology and Science, 2014, 14(4): 391-406.

[44] TANG M, ZHAO J Y, TONG R F, et al. GPU accelerated convex hull computation[J]. Computers & Graphics, 2012, 36(5): 498-506.

[45] MUYANOZCELIK P, OWENS J D, XIA J, et al. Fast Deformable Registration on the GPU: A CUDA Implementation of Demons[C]// International Conference on Computational Science and ITS Applications. 2008: 223-233.

[46] ERRA U, FROLA B, SCARANO V, et al. An Efficient GPU Implementation for Large Scale Individual-Based Simulation of Collective Behavior[C]// International Workshop on High PERFORMANCE Computational Systems Biology. IEEE, 2009: 51-58.