基于OpenCL的图像积分图算法优化研究

贾海鹏[1,2] 张云泉[2,3] 徐建良[1]

(中国海洋大学信息科学与工程学院 青岛 266100)[1]

(中国科学院软件研究所并行软件与计算科学实验室 北京 100190)[2]

(中国科学院软件研究所计算机科学国家重点实验室 北京 100190)[3]

摘 要 图像积分图算法在快速特征检测中有着广泛的应用,通过GPU对其进行性能加速有着重要的现实意义。然而由于GPU硬件架构的复杂性和不同硬件体系架构间的差异性,完成图像积分图算法在GPU上的优化,进而实现不同GPU平台间的性能移植是一件非常困难的工作。在分析不同GPU平台底层硬件架构的基础上,从片外访存带宽利用率、计算资源利用率和数据本地化等多个角度考察了不同优化方法在不同GPU硬件平台上对性能的影响。并在此基础上实现了基于OpenCL的图像积分图算法。实验结果表明,优化后的算法在AMD和NVIDIA GPU上分别取得了11.26和12.38倍的性能加速,优化后的GPU kernel比NVIDIA NPP库中的相应函数也分别取得了55.01%和65.17%的性能提升。验证了提出的优化方法的有效性和性能可移植性。

关键词 OpenCL,GPU,图像积分图算法,跨平台

中图法分类号 TP302.7 **文献标识码** A

Research on Image Integral Algorithm Optimization Based on OpenCL

JIA Hai-peng[1,2] ZHANG Yun-quan[2,3] XU Jian-liang[1]

(School of Information Science and Technology, The Ocean University of China, Qingdao 266100, China)[1]

(Laboratory of Parallel Software and Computational Sciences, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)[2]

(State Key Laboratory of Computing Science, Chinese Academy of Sciences, Beijing 100190, China)[3]

Abstract Image integral algorithm is widely used in fast feature detection, and improving the performance of this algorithm through GPU has an important practical significance. However, due to the complexity of the GPU hardware architecture and the architectural differences between different GPUs, how to complete the optimization of this algorithm and achieve performance portability on different GPU platforms is still a hard work. This paper analysed the differences between theunderlying hardware architectures of GPU, and studied the effects of performance on different GPU platforms using different optimization methods from the utilization of the off-chip memory bandwidth, the utilization of the computing resource, data locality and other aspects. And based on this, we implemented the image integral algorithm based on OpenCL. Experimental results show that optimized algorithm gets 11.26 and 12.38 times speedup on AMD and NVIDIA GPU respectively, and the performance of the optimized kernel improves 55.01% and 65.17% than the CUDA version in NVIDIA NPP library, which verifies the effectiveness and cross-platform ability of optimization methods.

Keywords OpenCL, GPU, Image integral algorithm, Across platform

1 引言

随着计算能力和可编程性的不断增强,越来越多的算法被成功移植到GPU平台上,并取得了很好的加速效果。已有的研究工作一般只针对单一的硬件平台,OpenCL(Open Computing Language,开放式计算语言)[5]是面向异构计算平台的通用编程框架,为实现GPU通用计算程序的跨平台移植提供了解决方案。然而由于GPU硬件体系结构的差异性,在不同GPU硬件平台间实现高效的性能移植是一件非常困难的工作。

图像积分图算法在快速特征检测中有着广泛的应用,而且这些应用大多都有很强的实时要求。随着这些应用数据量的不断增大,对图像积分图算法的性能提出了更高的要求,因此,利用GPU实现图像积分图算法的性能提升有着重要的现实意义和应用价值。目前国内外对图像积分图算法在GPU上的实现已经做了很多工作,然而这些工作大都针对单

到稿日期:2012-04-25 返修日期:2012-06-01 本文受国家自然科学基金资助项目(61133005, 61272136, 61100073),国家863项目(2012AA010902,2012AA010903),ISCAS-AMD联合fusion软件中心资助。

贾海鹏(1983-),男,博士生,主要研究领域为众核环境下编程方法研究,E-mail:jiahaipeng95@gmail.com;张云泉(1973-),男,博士,博士生导师,CCF会员,主要研究领域为高性能计算及并行数值软件、并行计算模型;徐建良(1969-),男,博士,教授,博士生导师,CCF会员,主要研究领域为算法分析与设计、自然语言处理。

一硬件平台，没有考虑不同硬件平台间的性能可移植性。

为指导通用计算程序在 GPU 体系结构上的有效映射，文献[12]定义了 NVIDIA GPU 效率和利用率模型，其通过优化空间的修剪来获得应用程序的最优配置，但是这种优化假设 GPU 上的可用资源不受限制；文献[13]针对 AMD GPU 提出了硬件感知的优化策略并定义了 ALU 利用率、纹理单元利用率和线程利用率 3 个优化空间，但是却没有针对 Global Memory 访存效率进行优化；文献[14]根据程序的访存特征建立了访存优化模型，但是也只针对 AMD GPU，并没有跨平台的实现。

本文在比较和分析当前主要 GPU 架构异同的基础上，针对 OpenCL 编程模型，提出了影响 OpenCL 程序性能的主要因素：片外访存带宽的利用率、计算资源的利用率和数据本地化，并结合 GPU 底层硬件特征给出了 GPU 平台上的通用优化方案。在此基础上实现了基于 OpenCL 的图像积分图算法。实验结果表明，优化后的算法在 AMD 和 NVIDIA GPU 上分别取得了 11.26 和 12.38 倍的性能加速，优化后的 GPU kernel 比 NVIDIA NPP 库中的相应函数也分别取得了 55.01%和 65.17%的性能提升，验证了本文提出的优化方法的有效性和性能可移植性。

本文第 2 节比较和分析了当前主要 GPU 架构的异同；第 3 节在 OpenCL 抽象模型的基础上，讨论了影响 OpenCL 程序性能的主要因素；在此基础上，第 4 节实现并优化了基于 OpenCL 图像积分图算法；第 5 节对其进行了详细的性能评测和分析；最后总结全文。

2 当前主流 GPU 体系架构

当前，主流 GPU 通用计算架构主要有两种：基于标量的 SIMT(Single Instruction Multiple Thread，单指令多线程)架构，如 NVIDIA 的 Fermi 架构[2]；基于向量的 SIMD(Single Instruction Multiple Data，单指令多数据)架构，如 AMD 的 Cypress 架构[4]。本节将分析两个架构的特点以及它们的异同。

2.1 NVIDIA Fermi 架构

Fermi 架构是 NVIDIA 发布的第三代流多处理器(Streaming Multiprocessor)架构。为了能够更适合通用计算的需求，Fermi 架构从底层计算单元、内存系统到芯片整体架构都进行了全新的设计。Fermi 共集成了将近 30 亿个晶体管，最多可拥有 512 个 CUDA 核心，它们分属于不同的 SM(Streaming Mutiprocessor，流多处理器)，每个 SM 单元拥有 32 个 CUDA 核心。这样，Fermi 架构可最多拥有 16 个 SM。图 1 显示了 NVIDIA Fermi 的整体架构。

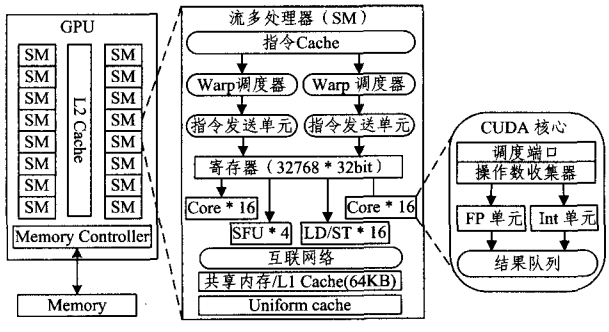


图 1 NVIDIA Fermi 架构

如图 1 所示，NVIDA Fermi 采用了三层的体系架构。第一层由若干共享 L2 Cache 的可扩展流阵列 SM 组成。不同型号的 GPU，SM 的数量可能不同，如 NVIDIA Tesla C2050 GPU 拥有 14 个 SM，不同 SM 的执行是相互独立的。第二层为流多处理器(SM)。SM 是一个高度并行处理器，最多支持 48 个 warp 的并发执行。每个 SM 上由两个 Warp 调度器、32 个 CUDA 核心、4 个 SFU(Special Function Units，特殊函数处理单元和 16 个 LD/ST(Load/Store Unit，存取单元)构成。第三层为 CUDA 核心。CUDA 核心是 GPU 最基本的指令执行单元，每个时钟周期可执行一条逻辑运算指令、单精度浮点数或者 32 位整数运算指令。每个 CUDA 核心由一个 FP(Float Point)计算单元和一个 Int 计算单元组成。这里要强调的是，CUDA 核心中并没有指令部件，其执行的操作由 SM 的指令发送单元控制。在 Fermi 架构中，Int 计算单元已经可以进行 32bit 的整形数运算。

Fermi 架构采用多线程调度机制。执行和调度单元为由 32 个线程组成的 warp。在 Fermi 架构中，每个流多处理有两个 warp 调度单元和两个指令发送单元。这就允许两个 warp 可以同时调度执行，每个 warp 由 SM 内的其中一组 16 个 CUDA 核心调度执行。由于 warp 执行的相互独立性，调度器没有必要检查指令流的依赖性，两个 warp 可并行执行。因此，在 GPU 程序优化中，要保证每个 SM 至少分配两个 warp 才能充分利用硬件资源。

2.2 AMD Cypress 架构

AMD Cypress 架构沿用了传统 GPU 的 SIMD 架构，可以看作是由若干个 SIMD 处理单元组成的多处理器。不同型号的处理器拥有的 SIMD 处理单元的数目是不同的。比如 AMD HD5850 GPU 有 18 个 SIMD 处理单元，而 AMD HD5870 有 20 个 SIMD 处理单元。AMD Cypress GPU 的架构特征如图 2 所示。

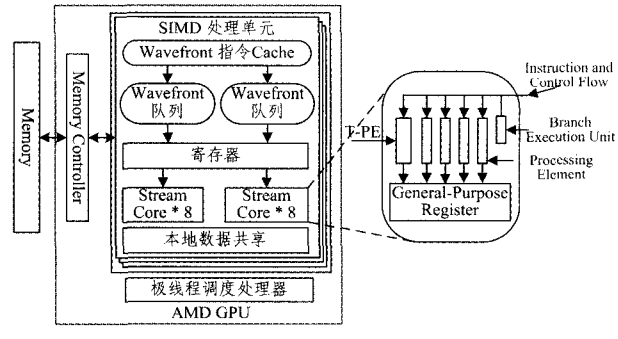


图 2 AMD Cypress 架构

如图 2 所示，同 NVIDIA Fermi 架构一样，AMD Cypress 架构依然采用了层次式的架构设计。其最基本的指令执行单元为 SC(Stream Core，流处理核心)。与 CUDA 核心不同，SC 采用了向量化架构，由 5 个相互独立的处理部件组成，形成了一个五路 VLIW(Very Long Instruction Word，超长指令字)处理器。这样 SC 在一个时钟周期内可最多同时处理 5 条相互独立的指令，这就为开发指令级并行提供了硬件条件。两个处理部件的组合可执行一条双精度浮点数元算，执行速度相对较慢。此外，SC 内部还有一个特殊函数处理单元(T-PE)，用以处理 sin，log 等复杂函数。16 个 SC 组成了 SIMD 处理单元，并共享一个程序计数器，因此，它们的执行完全是同步的。除此之外，SIMD 单元拥有有限数量的寄存器以及

程序员可以控制的本地内存(共享内存,NVIDA GPU),本地内存可被运行在 SIMD 单元的所有线程共享,并可实现位于同一线程块内的所有线程的通信。SIMD 计算单元阵列构成了 GPU,并通过极线程调度处理器(Ultra-thread dispatch processor)维护若干独立的命令队列,以调度大量线程独立计算数据流中不同的数据元素。

同 NVIDIA GPU 一样,AMD GPU 也采用了多线程调度机制。其执行和调度的最小单位是由 64 个线程组成的 wavefront。wavefront 间的执行是相互独立的,并且每个 wavefront 只能在一个 SIMD 计算单元上调度执行。由于每个 SIMD 计算单元内部只有 16 个 SC,因此一个 wavefront 指令的完成需要 4 个时钟周期。同 NVDIA GPU 一样,AMD GPU 可在 SM 上部署大量的 wavefront,这些 wavefront 将以时间片的方式轮转执行,当一个 wavefront 因访存而造成阻塞时,极线程调度处理器会立即调度其他 wavefront 来执行。以此来隐藏访存延迟,减少计算资源的浪费。这也是 GPU 程序优化的通用法则:在 GPU 上部署足够多的线程,以隐藏访存延迟。

AMD GPU 的每个 SIMD 计算单元同样拥有通用计算器和本地内存等有限的片上资源。如 Cypress 架构每个 SIMD 计算单元拥有 32k 的本地内存和 16k 的 128 位的向量寄存器。这些硬件资源的限制,同样会导致 GPU 程序优化空间的不连续性。

2.3 两种架构的异同

通过前两节的论述,从优化的角度讲,NVIDIA Fermi 架构 GPU 和 AMD Cypress 架构 GPU 在架构上的主要差异是基本执行单元设计上的差异:CUDA 核心采用了标量化的设计,而 SC 采用了向量化的设计。此外,NVIDIA GPU 寄存器是 32 位的标量化设计,而 AMD GPU 的寄存器为 128 位的向量化设计。这种设计上的差异也决定了两者在处理向量化指令方式上的不同。

对于 Fermi 架构,每个 CUDA 核心内部都有一个全功能的浮点数处理器。而对于 4D 矢量指令,NVIDIA GPU 会结合 4 个 CUDA 核心来共同完成。如图 3 所示,一条 4D 的矢量指令在 NVIDIA GPU 上执行时,会转换为相互独立的 4 条标量指令,并被分配到不同的 CUDA 核心上执行。这种实现方式的特点就是灵活,无论是 1D,2D,3D 还是 4D 指令,都拆分成相互独立的 1D 指令来执行。CUDA 的这种流处理器架构的设计放弃了单独追求高浮点数计算吞吐量的目标,而是通过优化处理器内部结构来换取更高的执行效率,同时也大大提高了架构的灵活性和可扩展性。

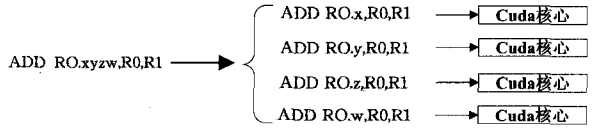


图 3 矢量加法在 NVIDIA GPU 上的运行

AMD GPU 沿用了传统 GPU 的 SIMD 架构,每个 SC 拥有 5 个共享指令发射端口的 1D ALU,并被组织为 5 路 VLIW(Very Long Instruction Word,超长指令字)处理器,在每个时钟周期内,一条 VLIW 指令可同时执行 5 个标量操作。从宏观上看,AMD GPU 确实是 SIMD 架构,因为 SC 内部的 5 个 ALU 共用一个指令发射端口。然而,这 5 个 ALU 与传统 GPU 的 ALU 不同,它们是相互独立的,并能各自组合处理任意的 1D/2D/3D/4D/5D 指令,完美支持 Co-issue 矢量指令和标量指令并行执行,因此,从微观上看,AMD GPU 可以称为 5D 超标量架构。

在 VLIM 体系中,通过将多个短指令合并成一条长 VLIW 指令的方式来提高计算资源的利用率,最大限度地缓解了标量指令效率低下的问题。而另一方面,在通用计算中,并不能将所有的计算指令都组装成合适长度的向量化指令。因此,SC 的这种向量化的设计,也会降低计算资源的利用率,同时也增加了 AMD GPU 在优化过程中的难度。

尽管 NVIDA Fermi 架构和 AMD Cypress 架构存在着差异,但从宏观上看,两者在整体设计上存在着相似性:

1. 相同的硬件层次架构。NVIDIA Fermi 和 AMD Cypress 架构都采用了层次式的硬件架构,基础架构由可扩展的计算单元阵列组成,而计算单元由计算核心构成。计算核心的操作由所属的计算单元的指令发送单元控制。

2. 相似的内存层次架构。两者同样采用了层次式的内存架构:线程私有的寄存器、可被同一线程块内所有线程共享并可被程序员控制的本地内存(NVIDIA GPU 称为共享内存)、可被所有线程访问且大容量的全局内存。前两者为片上内存,具有较高的访存带宽和较低的访存延迟,但容量有限;后者成为片外内存,容量大,但具有较高的访存延迟和相对较低的访存带宽。充分利用片上资源进行数据本地化,以减少对本地资源的依赖,是 GPU 优化的重要方法。

3. 相似的调度策略。两者都采用了多线程的调度策略,而不同的是,在 NVIDIA Fermi 架构中,调度单元 warp 包含 32 个线程;而在 AMD Cypress 架构中,调度单元 wavefront 包含 64 个线程。两者都能有效地控制大规模线程的并行执行,通过 warp/wavefront 的交替执行来隐藏访存延迟。同时线程间的调度都是通过硬件实现,开销低。

4. 相同的线程组织架构。两者都属于大规模并行处理器,可并行执行大量线程。除硬件层次外,还有将在下节介绍的两者具有相同线程的组织形式,即将要执行的大量线程被组织成 N 维的线程空间,线程空间又进一步切分成不同的线程块,每个线程块只能在一个计算单元上调度执行,线程块间的执行是相互独立的。同一线程块内的线程可通过本地内存进行通信。每个线程移只能在一个处理核心上执行。

这些硬件和软件架构的相似性,为研究分析 GPU 程序在不同 GPU 架构上的优化方法和策略的共性提供了可能,也为进一步实现不同 GPU 架构间的性能可移植创造了条件。

3 OpenCL 简介

OpenCL(Open Computing Language,开放式计算语言)[5],是一个在异构平台上编写并行程序的开放框架标准,包括一个底层程序接口和一个高性能、可移植的抽象层,为并行程序设计提供了一个有效的并行开发环境、一个平台独立的工具和丰富的中间软件层。OpenCL 将 GPU、CPU 以及其他各类计算设备组织成一个统一的计算平台(在本文中,这个计算平台特指由 GPU 和 CPU 组成的异构平台),并提供了基于任务和基于数据的两种并行计算机制,极大地扩展了 GPU 的应用范围,使之不再局限于图形领域。

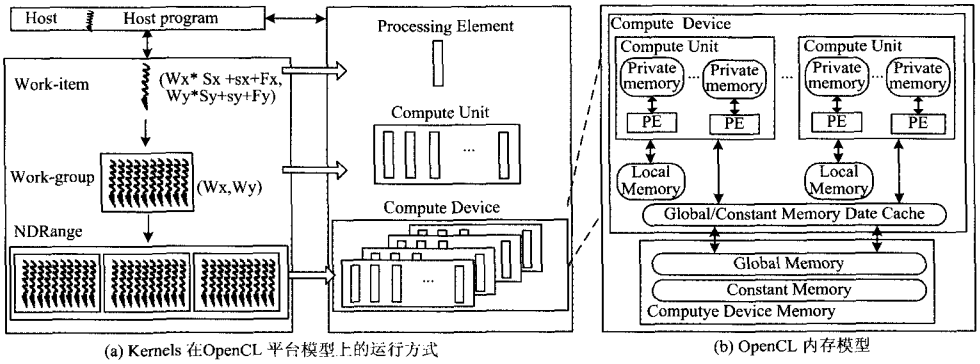
图 4 显示了 OpenCL 定义的抽象模型。OpenCL 执行架构中使用主机端程序(Host program)对多个支持 OpenCL 的计算设备(Compute device)进行统一管理和调度。当主机端提交 kernel 到计算设备时，OpenCL 通过索引空间(NDRange)定义 work-item(一个 kernel 实例)的组织结构并以计算设备上的映射方式定义 kernel 在计算设备上的运行方式，如图 4(a)所示。同时，在内存模型中，OpenCL 根据线程访问权限的不同将内存空间分为如下 4 种：全局内存(Global Memory)、常量内存(Constant Memory)、局部内存(Local Memory)以及私有内存(Private Memory)，如图 4(b)所示，其大小和对应的访问速度各不相同，其使用方式是否得当直接决定程序性能的高低。

OpenCL 算法的优化是 OpenCL 程序设计中的重要内容。通过以上章节对 GPU 硬件架构及 OpenCL 编程模型的分析可知，OpenCL 程序的优化与具体的硬件平台密切相关。但从总体上看，无论采用何种硬件平台，提高 OpenCL 程序性能的主要途径可分为 3 个方面：

1)片外访存带宽的利用率。全局内存容量大，但访存带宽相对较低，具有较高的访存延迟，是限制 GPU 程序性能的主要因素之一。充分考虑 GPU 访存特点，提高片外访存带宽利用率，降低访存延迟，是提高 GPU 程序性能的关键。

2)计算资源的有效利用率。GPU 拥有大量的计算单元，具有强大的计算能力。充分考虑 GPU 架构特点和调度策略，以有效地利用这些计算单元，避免计算单元的闲置，是提高 GPU 程序性能的主要途径。

3)数据本地化。GPU 中提供了本地内存、Cache 等。片上资源具有较高的访存带宽和较低的访存延迟。充分利用这些片上资源实现数据的共享和重用，减少对片外资源的访存次数，是提高 GPU 程序性能的主要方法。



(a) Kernels 在OpenCL 平台模型上的运行方式　　(b) OpenCL 内存模型

图 4　OpenCL 抽象模型

由于 GPU 硬件架构的差异，GPU 程序在优化方法和优化策略的选择上也存在差异性。但无论采用何种 GPU 架构，其优化的方向和途径均相似，这也为 GPU 程序的性能可移植提供了可能。

4　图像积分图算法

4.1　算法概述

图像积分图算法在快速特征检测中有着广泛的应用，主要用于规则区域特征值的计算。在本算法中，我们依然用二维矩阵来表示二维图像。式(1)说明了积分图算法的基本原理：

$$VI(x,y)=\sum_{x'\leqslant x,y'\leqslant y} i(x',y') \tag{1}$$

式中，$VI(x,y)$表示矩阵元素(x,y)的积分图，$i(x',y')$表示矩阵元素(x',y')的值。从式(1)中可以得出矩阵中某一元素的积分图等于该点左上方所有元素值之和。图 5 为其计算示意图。

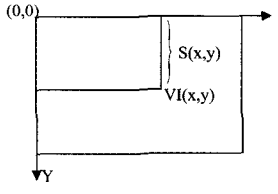


图 5　图像积分图计算示意图

$S(x,y)$表示矩阵元素(x,y)在 y 方向上的所有矩阵元素值之和。这样积分图的计算可分为两部分：

$$S(x,y)=S(x,y-1)+i(x,y) \tag{2}$$

$$VI(x,y)=VI(x-1,y)+S(x,y) \tag{3}$$

从式(2)、式(3)可以看出，积分图算法类似于前缀和计算。实际上，如要同时计算矩阵所有元素的积分图，积分图算法一般被拆分成两步前缀和计算：首先计算矩阵 A 在 x 方向上的前缀和矩阵 A'，然后再计算矩阵 A'在 y 方向上的前缀和矩阵 A''。矩阵 A''即为矩阵 A 的积分图矩阵。在 GPU 实现中，为提高访存性能，计算矩阵 A'在 y 方向上的前缀和矩阵时，通常先将矩阵 A'转置，然后再进行 x 方向的前缀和计算。这样，矩阵 A 的积分图算法可分为 3 步：x 方向上的前缀和计算、矩阵转置、x 方向上的前缀和计算。图 6 显示了 x 方向的前缀和算法的计算过程。也就是说，积分图算法最终转化为两次 x 方向上的前缀和算法和一次矩阵转置算法。本节将重点讨论前缀和算法在 GPU 上的实现和优化。

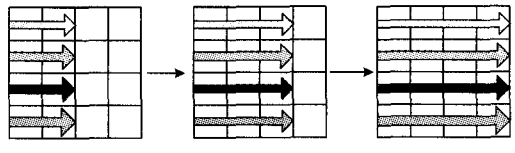


图 6　矩阵 x 方向上的前缀和算法

首先介绍前缀和算法的基本原理，前缀和定义非常简单，即给定一个数值序列：

$[a_0,a_1,a_2,\cdots,a_{n-1}]$

返回计算结果：

$[a_0,(a_0+a_1),(a_0+a_1+a_2),\cdots,(a_0+a_1+\cdots a_{n-1})]$

从前缀和算法的定义上看，前缀和算法具有非常鲜明的串行属性：首先计算完成第 $i-1$ 个元素的前缀和 Sum_{i-1}，然后再计算第 i 个元素的前缀和 $Sum_i = Sum_{i-1} + a_i$。Hillis[15] 最早提出了前缀和算法的并行算法。本文将在下面的内容中，在该并行算法的基础上，详细分析前缀和并行算法在 GPU 上的实现和优化。

4.2 算法优化

积分图算法的核心是对矩阵各行先后进行两次前缀和计算。因此，在计算该算法的计算密度时，可直接使用前缀和算法的计算密度。

下面的伪代码是 Hillis 提出的前缀和并行算法：

```
for(int d=1;d < log2n;d++)
forall k in parallel do
if k≥2^d
then x[k]=x[k−2^(d−1)]+x[k]
```

由算法伪代码可以看出，前缀和算法需要对同一内存位置进行多次读写，且访存模式为非连续访存，这样的访存操作会严重影响程序性能。为此，采取本地分块计算策略：首先以 work-group 所处理的数据元素为单位将各矩阵行分块，在本地内存中完成各数据块的局部前缀和计算；然后将每个数据分块的数据元素和（最后一个数据元素的前缀和）集合到一个数组中，并计算该数组的前缀和（注意此时前缀和计算不要不包含数据元素本身）；最后将该前缀和数组的各个元素加到对应的数据块中完成最终前缀和的计算，图 7 显示了这个计算过程。同时要注意各矩阵行的前缀和计算是相互独立的，可并行执行的。这里需要注意的是，在本算法的实现过程中，采用了固定线程数目的方法：即每个 CU 上部署 4 个 work-group，每个 work-group 的大小为 256 个线程，这样总的 work-group 的数目为 4 * CUs，CUs 为 GPU 所拥有的 CU 数目。所以存储各数据分块元素之和的数组大小为 4 * CUs，数据量非常小。因此，该数组的前缀和计算可在 CPU 上完成。而最后一步加和操作依旧在 GPU 上完成。因此，该算法实际要启动两个 kernel 的运行，并有一个全局同步操作。

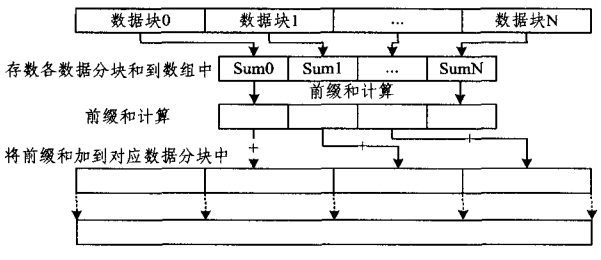


图 7 前缀和本地分块算法

该算法的具体优化过程如下：

1）提高片外带宽资源利用率

由前面讨论的前缀和算法可知，该算法的访存模式为连续访存和对齐访存。因此，提高访存带宽利用率的主要优化方法为向量化访存。在本算法实现中，考虑不同向量长度在不同 GPU 平台上对访存性能的影响，在两个 GPU 平台上采用向量长度为 2 的向量。

2）数据本地化

正如前面所讨论的那样，前缀和算法的特性决定了该算法要通过使用本地内存来实现数据本地化，增加数据重用，减少对片外访存带宽的依赖，以提高程序性能。

3）提高计算资源的利用率

减少指令数目，提高指令效率。图 8 是 Hillis 提出的前缀和并行算法的计算过程。值得注意的是，该算法存在同一内存位置被多个并行执行的线程多次读写的情况，从而造成数据覆盖。为避免由于数据的重写而导致计算结果错误，在算法的具体实现中，我们采用了双 buffer 的形式。即为每个数据分块申请两块同样大小的本地内存空间交替使用，以避免数据的重写覆盖。

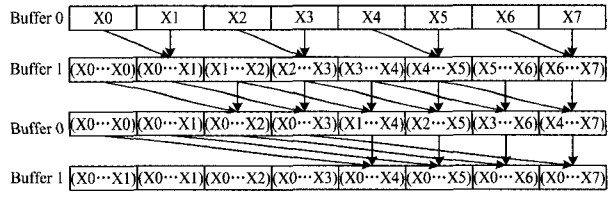


图 8 Hillis 前缀和算法

如图 8 所示，两个 buffer 交替使用，通过使用本地同步操作，避免了一个内存位置的多次读写，保证了结果的正确性。该算法虽然较好地挖掘了数据计算的并行性，但总共需要进行 $O(n\log_2 n)$ 次计算，总计算量与数据增长的比例是非线性的。当 n 较大时，$\log_2 n$ 次计算会大大降低程序的性能。由此可见，线程的效率过低。为此引入平衡树计算方法，将总共需要进行的计算次缩减到 $O(n)$，计算过程如图 9 所示。改进后的前缀和算法共分为两个阶段：规约阶段和交换加和阶段。在规约阶段，从平衡树的叶子节点开始遍历，并在其父节点计算部分和一直到根节点。最终根节点的值为所有元素的和，因此这个过程称为规约阶段。在交换加和阶段则从平衡树的根节点触发，利用规约阶段得到的部分和计算相应节点的前缀和。最终，平衡树的所有叶子节点即为数据元素的前缀和。需要注意的是，通过这个方法获得的前缀和并没有包含最前端和最末端的数据元素。因此，该算法最后还需要进行两步操作：一是需要将各前缀和元素的值加上数组的最前端数据元素 X_0，这个过程可以并行执行；二是将最后的前缀和元素加上最末端数据元素 X_n。改进后的前缀和算法共进行 $2*(n-1)$ 次加法操作和 $(n-1)$ 次交换操作，其复杂度为 $O(n)$，从而使总工作量随着数据元素的正常而呈线性，减少了每个线程的计算指令，提高了线程的效率，提升了程序性能。

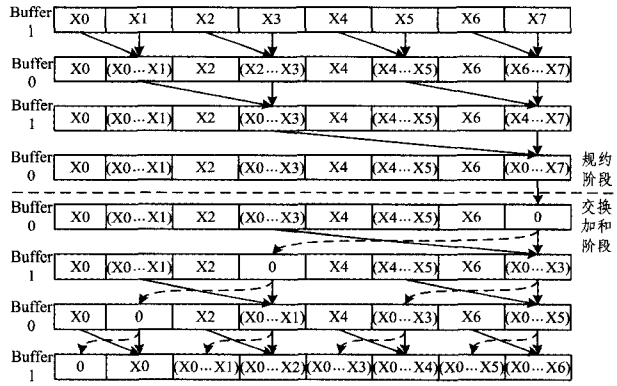


图 9 改进前缀和算法

消除 bank 冲突。改进前缀和算法在本地内存的使用上

存在着严重的本地内存 bank 冲突。例如:规约阶段随着相邻线程数据访问的内存跨度不断增大,发生 bank 冲突的几率不断增加。交换求和阶段,随着同时访问本地内存线程数量的增多,当相邻线程进行数据访问的内存跨度过大时,也会发生本地内存 bank 冲突。在这个阶段,bank 冲突的数量从根节点到平衡树的中间层不断增多,从中间层到叶子节点,发生 bank 冲突的数量开始逐渐减少。解决 bank 冲突的主要方法是添加 padding,使引发 bank 冲突的相邻线程单位(一个 warp 或者 1/4 个 wavefront)所访问的内存跨度不是所有 bank 大小的整数倍。在本算法中,当一个线程处理一个数据,即数据类型为 float 时,每隔 16(AMD GPU)或者 32(NVIDIA GPU)个数据元素添加 4 个字节的 padding,可最大程度地减少 bank 冲突。当采用向量优化方法,例如一个线程处理两个数据,即数据类型为 float2 时,每隔 16(AMD GPU)或者 32(NVIDIA GPU)个 float2 类型的数据元素添加 4 个字节的 padding,也可减少 bank 冲突。以此类推,当数据类型为 float$N$ 时,每隔 16(AMD GPU)或者 32(NVIDIA GPU)个 float$N$ 类型的数据元素添加 4 个字节,可有效减少本地内存 bank 冲突。

开发 ILP(Instruction-Level Parallelism,开发指令级并行)。指令级并行主要考虑的是 work-item 内部指令的并行性开发。开发指令级并行一方面可以减少不同指令束之间的切换,另一方面可以充分利用计算资源,特别是采用向量化架构的 AMD HD5850 GPU,其每个处理单元内部有 5 个处理元件可同时运行 5 条相互独立的计算指令。更重要的是,开发指令级并行可以在每个 CU 上部署较少线程的情况下,达到较高的性能。CU 上部署的线程数目的减少,会增加每个线程使用寄存器和本地内存的数量,从而减轻硬件资源的限制因素和对片外内存资源的依赖,进一步提高 GPU 程序的性能。循环展开和调整代码顺序将相互独立且类型相同的指令放在一起,是开发指令级并行的常用优化方法。

在本算法中,开发 ILP 的主要方法是向量化操作,每个线程负责多个元素的前缀和计算。但向量化也会增加每个 work-group 的本地内存使用,从而限制同时运行的线程数目的减少,导致程序性能的降低。因此,向量化操作需要权衡本地内存的使用,通过不断试验,在两个 GPU 平台上找到该算法的最佳向量长度。循环展开也可开发 ILP,但在本算法中,循环展开的主要目的是减少动态指令。

指令选择优化。由于指令本身的特点和处理元件数目的差异,无论是 NVIDIA Tesla C2050 GPU 还是 AMD HD5850 GPU,其不同指令的吞吐量是存在差异的。例如,在 AMD HD5850 GPU 上,24bit 的整形乘法和乘加指令的吞吐量就为 32bit 整形乘法指令吞吐量的 5 倍。因此,尽可能选择低延迟和高吞吐量的指令,是提高 GPU kelrnel 性能的有效手段。在本算法中,高吞吐量计算指令的选择主要体现在两点:一是使用位运算指令代替乘法和除法运算指令(操作数为 2 的幂),前者的吞吐量大约是后者的 5 倍;二是使用 mad24 指令代替乘法和加法运算指令。

5 性能评测

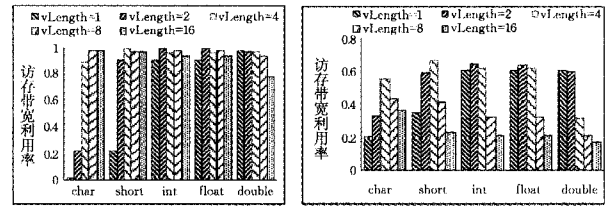
我们使用 AMD 5850、NVIDIA Tesla C2050 两个 GPU 平台测试程序的性能移植,验证了程序在不同 GPU 硬件平台上的性能可移植性。两个 GPU 平台的主要性能参数如表 1 所列。

表 1 两个 GPU 平台的主要性能参数

	NVIIDA C2050 GPU	AMD HD 5850
Clock Rate	1.15 GHZ	0.725 GHZ
#PEs	448	288
#CUs	14	18
Peak Perf.	1030 GFlops	2090 GFlops
Memory	3.0 GB	1.0 GB
Peak Bandwidth	144 GB/s	128 GB/s
#Register/CU	16k	16k
#Local Memory/CU	48k	32k
SDK version	SDK 4.1	SDK 2.6

5.1 向量化对 Global Memory 访存带宽利用率的影响

图 10 显示了不同数据类型的向量化访存分别在 AMD HD5850 GPU 和 NVIDIA Tesla C2050 GPU 平台上对访存带宽利用率的影响。



(a) AMD HD5850 GPU (b) NVIDIA Tesla C2050 GPU

图 10 向量化访存在不同平台上对访存带宽利用率的影响

对于 AMD HD5850 GPU 来说,如图 10(a)所示,向量化访存对性能有较大的影响,特别是当数据类型为 char1,char2、hort1 时,访存带宽利用率非常低。这是因为这些数据类型的大小都小于 32bit,访存请求是通过性能非常低的 CompletePath 完成的。而当数据类型为 short2,int 或者 float 时,性能相对较低,这是因为 AMD GPU 内部支持 128bit 的数据传输,当数据类型的字节数不足够大时,并不能充分利用其内部的访存带宽。同时我们注意到,无论何种数据类型,向量化访存总能提高访存带宽利用率,只是所采用的向量长度有所差异。当数据类型为 char4、short4、int2、float2、double2 时,访存带宽达到最高利用率。综上,在 AMD HD5850 GPU 上,通过对不同数据类型选取合适的向量化长度进行向量化访存,可以实现最高 527.2%(char)、最低 21.2%(doule)的性能提升。

对于 NVIDIA Tesla C2050 GPU,如图 10(b)所示,向量化访存对访存性能也有较大的影响。除 double 外,采用合适长度的向量进行向量化访存,总能提高访存带宽的利用率。特别是 char 和 short 数据类型,向量化访存能大大提高访存性能,但对于 int 和 float 数据类型来说,向量化访存对性能的影响不是那么明显。当数据类型为 char4、short4、int2、float2 和 double 时,访存带宽利用率达到最高。

5.2 图像积分图算法优化分析

图 11 显示了在采用不同优化方法后,图像积分图算法在不同 GPU 硬件平台上的性能提升情况。在性能测试中,为检测方便,使用矩阵代替图像。在本测试用例中,矩阵的大小为 1280 * 1280,数据类型为 8UC1(单通道)类型。

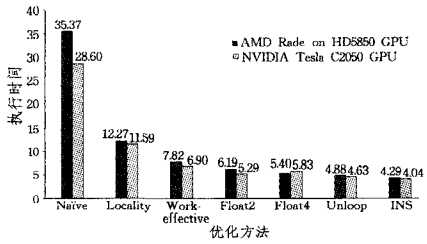


图 11　图像积分图算法在采用不同优化方法后的性能变化

通过图 11 可以得出以下几点结论：

1. 由于算法本身的特性，需要对同一内存位置进行多次读写，并且访存模式为不连续访存。因此，通过数据本地化，在本地内存中实现前缀和计算，有利于数据重用，减少对片外访存带宽的依赖，可提高算法的计算密度，并大大提高算法的性能。

2. 改进的前缀和算法的一个重要特点是减少了指令数量和工作量，提高了指令效率，节约了计算资源，同样较大地提升了程序的性能。

3. 通过向量化开发 ILP，依然是一个需要权衡的过程。如本算法在 NVIDIA Tesla C2050 GPU 上，当向量长度超过 2，如为 4 时，性能就会下降。这是因为较长的向量长度不仅会造成本地内存和寄存器的过量使用，而且也会增加发生本地内存 bank 冲突的几率。在 AMD HD5850 GPU 平台上，虽然采用长度为 4 的向量依然会改进程序的性能，但在此基础上进行循环展开，反而降低了程序性能。这主要是因为在本算法中，循环展开本身对性能提升的影响很小，却增加了寄存器的使用。而在 NVIDIA Tesla C2050 GPU 上，在采用长度为 2 向量情况下进行循环展开，可小幅提高程序性能。

4. 通过选择吞吐量较高的指令，也可小幅提高程序的性能。

总之，通过精心优化，图像积分图算法在 AMD HD5850 GPU 和 NVIDIA Tesla C2050 GPU 上分别实现了 8.24 和 6.51的加速比。

为提高测试结果的可信性，在 CPU 版本上，我们选择了 OpenCV2.3 库中的图像积分图算法；在 CUDA 版本上，我们选择了 NVIDIA NPP 库中该函数的实现，这两个版本都进行了精心的优化，与它们进行性能比较，更能体现出我们优化工作的成绩。性能比较结果如图 12 所示。

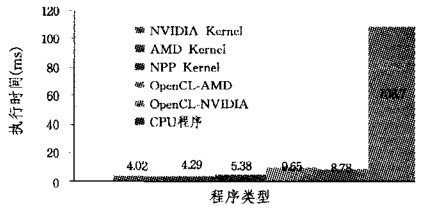


图 12　图像积分图算法多种版本的性能比较

优化后的 kernel 在 NVIDIA GPU 和 AMD GPU 上都达到了较高的性能，与 NPP[16] kernel 相比，分别有 65.17%和 55.01%的性能提升。在加入 kernel 启动时间和 CPU 端程序的时间后，OpenCL 程序在两个 GPU 平台上也达到了 12.38 和 11.26 倍的性能提升，充分证明了优化的有效性。

结束语　本文给出了影响 OpenCL 程序在 GPU 平台上性能的 3 大核心因素：片外访存带宽的有效利用率、计算资源的有效利用率以及数据本地化；并指出通过向量化提高访存带宽利用率，通过合理安排线程组织结构、避免分支、合理利用片上资源，以访存次数提高 GPU 计算资源的有效利用率，是提高程序性能的通用方法。本文实现了基于 OpenCL 的图像积分图算法，验证了向量化访存和计算在 NVIDIA 和 AMD 两个 GPU 平台上都能取得良好的加速效果，但在 AMD GPU 平台上的效果要明显得多。实验结果表明，优化后的算法在 AMD 和 NVIDIA GPU 分别取得了 11.26 和 12.38倍的性能加速，优化后的 GPU kernel 比 NVIDIA NPP 库中的相应函数也取得了 55.01%和 65.17%的性能提升，验证了本文提出的优化方法的有效性和性能可移植性。

参 考 文 献

[1] Owens J D, Luebke D, Govindaraju N, et al. A survey of general-purpose computation on graphics hardware [J]. Computer Graphics Forum, 2007, 26(1): 80-113

[2] NVIDIA Corporation. NVIDIA CUDA C Programming Guide V3.2[M]. Sept. 2010

[3] NVIDIA Corporation. OpenCL programming Guide for the CUDA Architecture V3.2[M]. Aug. 2010

[4] AMD Corporation. Accelerated Parallel Processing OpenCLTM [M]. Jan. 2011

[5] KHRONOS OpenCL Working Group. The OpenCL Specification V1.1[S]. Sept. 2010

[6] Lindholm E, Nickolls J, Oberman S, et al. NVIDIA Tesla: A Unified Graphics and Computing Architecture[J]. IEEE Micro, 2008, 28(2): 39-55

[7] Bordoloi U D. Optimization Techniques: Image Convolution [Z]. 2011

[8] Chen Ying, Lin Qian-jin. Implementation of LU decomposition and Laplace algorithms on GPU[J]. Journal of Computer Applications, 2011, 31(3): 851-855

[9] Tang Tao, Lin Yi-song. Design and Implementation of Jacobi andLaplace Algorithms on GPU Platform[J]. Computer Engineering & Science, 2009, 31(1): 93-97

[10] Taylor R, Li Xiaoming. A Micro-benchmark Suit for AMD GPUs[C]// 39th International Conference on Parallel Processing Workshops. 2010: 387-396

[11] Papadopoulou M, Sadooghi-Alvandi M, Wong H. Micro-benchmarking the GT200 GPU[R]. Computer Group. ECE, University of Toronto, 2009

[12] Ryoo S, Rodrigues C I, Stone S S, et al. Programoptimization space pruning for a multithreaded GPU[C]// 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization. New York: ACM Press, 2008: 195-204

[13] Jang B, Do S, Pien H, et al. Architecture-Aware optimization Targeting Multithreaded Stream Computing [C] // The 2nd Workshop on General Purpose Processing onGraphics Processing Units. New York: ACM Press, 2009: 62-70

[14] Jang B, Schaa D, Mistry P, et al. Exploiting Memory Access Patterns to Improve Memory Performance in Data Parallel Architectures[J]. Parallal and Distributed Systems, 2011, 22(1): 105-118

[15] Hillis W D, Steele G L Jr. Data Parallel Algorithms[J]. Communications of the ACM Special issue on parallelism, 1986, 29(12): 1170-1183

[16] NVIDIA NPP Library[OL]. http://developer.nvidia.com/npp