基于OpenCL的拉普拉斯图像增强算法优化研究

贾海鹏[1,2] 张云泉[1,3] 龙国平[1] 徐建良[2] 李 焱[1,4]
(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)[1]
(中国海洋大学信息科学与工程学院 青岛 266100)[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]
(中国科学院研究生院 北京 100190)[4]

摘 要 OpenCL是面向异构计算平台的通用编程框架,然而由于硬件体系结构的差异,如何在平台间功能移植的基础上实现性能移植仍是有待研究的问题。当前已有算法优化研究一般只针对单一硬件平台,它们很难实现在不同平台上的高效运行。在分析了不同GPU平台底层硬件架构的基础上,从Global Memory的访存效率、GPU计算资源的有效利用率及其硬件资源的限制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的拉普拉斯图像增强算法。实验结果表明,优化后的算法在不考虑数据传输时间的前提下,在AMD和NVIDIA GPU上都取得了3.7~136.1倍、平均56.7倍的性能加速,优化后的kernel比NVIDIA NPP库中相应函数也取得了12.3%~346.7%、平均143.1%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,通用计算,拉普拉斯算法,跨平台
中图法分类号 TP302.7 **文献标识码** A

Research on Laplace Image Enhancement Algorithm Optimization Based on OpenCL

JIA Hai-peng[1,2] ZHANG Yun-quan[1,3] LONG Guo-ping[1] XU Jian-liang[2] LI Yan[1,4]
(Laboratory of Parallel Software and Computational Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)[1]
(School of Information Science and Technology, Ocean University of China, Qingdao 266100, China)[2]
(State Key Laboratory of Computing Science, Chinese Academy of Sciences, Beijing 100190, China)[3]
(Graduate University, Chinese Academy of Sciences, Beijing 100190, China)[4]

Abstract OpenCL is a general-purpose programming framework forheterogeneous computing platforms, however, due to the differences in hardware architecture, how to achieve performance portability on different platforms based on the function portability is still to be studied. Currently most of the researches on algorithm optimization are aimed at a single hardware platform, and difficult to achieve the efficient running on different platforms. This paper analysed the differences between the underlying hardware architectures of GPU, and studied the effects of different GPU platforms using different optimization methods on performance from the access efficiency of global memory, full use of the GPU compute resource, the constraints with hardware resource and other aspects. Based on this, the Laplace image enhancement algorithm based on OpenCL was implemented. Experimental results show that optimized algorithm gets 3.7~136.1 times and 56.7 times on average speedup(without calculate the data transfer time) on both AMD and NVIDIA GPU, and the performance of the optimized kernel increases 12.3%~346.7% and 143.1% on average than the CUDA version in NVIDIA NPP library, which verifies the effectiveness and cross-platform ability of optimization methods.

Keywords OpenCL, General-purpose computing, Laplace, Across platform

1 引言

随着GPU通用计算的发展,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于单一的硬件平台。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,它为实现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,不

到稿日期:2011-06-27 返修日期:2011-11-16 本文受国家自然科学基金项目(60303020,40806040),国家自然科学基金重点项目(60533020),国家"863"计划基金项目(2006AA01A102, 2R2010FM002)和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,E-mail:jiahaipeng95@gmail.com;张云泉(1973-),男,博士,研究员,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;龙国平(1982-),男,博士,助理研究员,主要研究领域为计算机体系结构等;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;李 焱(1985-),男,博士生,主要研究领域为FFT、并行编程方法。

同的 GPU 硬件平台很难实现高效的移植。因此，如何在功能可移植的基础上实现性能移植仍是有待研究的问题。

图像 Laplace 变换是重要的图像增强算法，基于 Laplace 的图像增强已经成为图像锐化处理的基本工具，同时 Laplace 算法是科学计算领域中常用的核心算法，在微分方程求解和信号处理等领域有广泛的应用[15]。研究 Laplace 算法的 GPU 实现，对开发基于 GPU 平台的图像处理程序和通用科学计算程序都有着重要的意义。目前国内外对 Laplace 算法在 GPU 上的实现已经做了很多工作[7-9]。然而文献[7]只针对 GPU 容易处理的 Float 类型；文献[8]为了编码方便，舍弃了对图像边界像素点的处理，同时这些工作都是针对某一特定的 GPU 硬件平台，并没有进行跨平台的优化和实现。

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，通过优化空间的修剪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，但是也只针对于 AMD GPU，并没有跨平台的实现。

本文针对 OpenCL 模型，提出了影响 OpenCL 程序性能的主要因素：Global Memory 的访存效率、GPU 计算资源的有效利用率及其硬件资源的限制，并结合 GPU 底层硬件特征给出了 GPU 平台上的通用优化方案。并在此基础上实现了基于 OpenCL 的拉普拉斯图像增强算法，其在不考虑数据传输时间的前提下，在 AMD 和 NVIDIAGPU 上都取得了3.7～136.1 倍、平均 56.7 倍的性能加速，优化后的 kernel 比 NVIDIA NPP[20] 库中相应函数也取得了 12.3%～346.7%、平均 143.1%的性能提升。

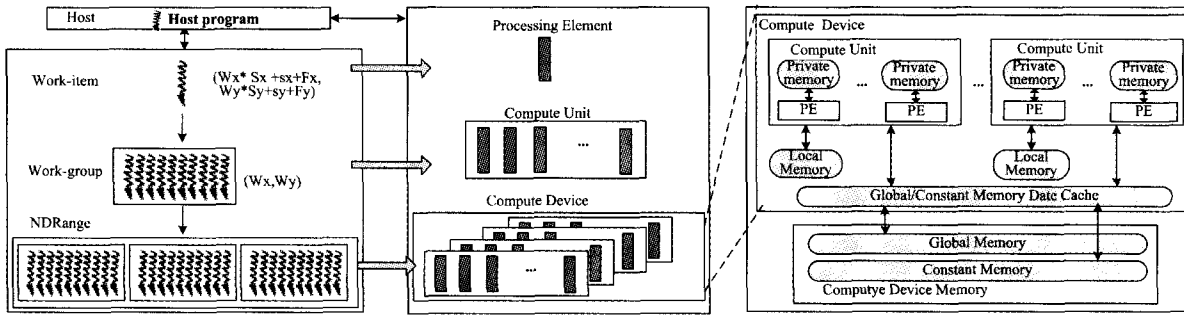
本文第 2 节在分析 OpenCL 抽象模型的基础上，讨论了影响 OpenCL 程序性能的主要因素；第 3 节研究了典型 GPU 通用计算架构底层硬件设计的差异，提出了针对 GPU 平台的通用优化方法；第 4 节实现并优化了基于 OpenCL 拉普拉斯算法的图像增强算法，并在第 5 节对其进行了详细的性能评测和分析；最后总结全文。

2 OpenCL 概述

2.1 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 1 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 1(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 1(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。



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

图 1　OpenCL 抽象模型

2.2 OpenCL 算法优化

OpenCL 算法优化是 OpenCL 程序设计的重要内容，虽然 OpenCL 程序的性能优化方法和具体的硬件平台密切相关，但无论采用何种硬件平台，影响 OpenCL 程序性能的核心因素始终包含以下 3 个方面。

1）Global Memory 的访存效率。在 OpenCL 内存模型中，如图 1(b)所示，Global Memory 是计算设备外内存，容量大，但是访存速度慢，其访存效率的高低直接决定着 kernel 性能的高低。因此，最大限度地利用 Global Memory 的访存带宽，提高 Global Memory 的访存效率是提高 OpenCL 程序性能的主要方法。

2）计算资源的利用率。在 OpenCL 平台模型中，如图 1(a)所示，计算设备由多个计算单元（Compute Unit，CU）组成，计算单元又划分为多个处理单元（Processing Element，PE），PE 作为最基本的处理单元，是执行计算任务的基本单位。因此，最大限度利用 PE 的效能，从而充分利用整个计算设备的计算能力，是提高 OpenCL 程序的关键因素。

3）硬件资源的限制。除计算资源外，计算设备上的 Local Memory、寄存器等硬件资源，在很大程度上决定着能够同时在计算设备上并发执行的 kernels 的数目，从而影响 OpenCL 程序的性能。因此，合理有效地利用计算设备上有限的硬件资源，是提高 OpenCL 程序的有效方法。

充分优化影响 OpenCL 程序性能的 3 个核心要素是提高 OpenCL 程序在 GPU 平台上性能的关键，而 GPU 硬件平台的差异性导致程序在不同 GPU 平台上优化方法的不同。因此，分析 GPU 底层硬件架构设计的异同，找到不同 GPU 平台通用的优化策略和方法，是实现 OpenCL 程序在 GPU 平台上高效移植的关键。

3 典型 GPU 硬件架构比较

目前，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 CUDA 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]。图 3 显示了这两种架构在底层硬件设计上的特点和差异。

3.1 NVIDIA CUDA 架构

传统 GPU 的架构是基于 SIMD 的，NVIDIA GPU 从 G80 架构开始使用全新的设计思想，它将 GPU 性能提升的重点放在了优化 ALU 阵列的结构上，而不是一味地追求数量的增长[18,19]。NVIDIA 将这种全新设计的 GPU 架构称为 CUDA。

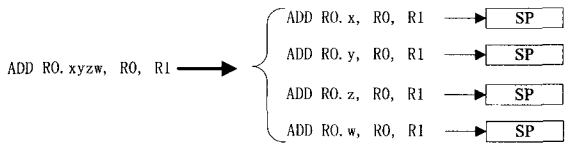


图 2 矢量加法在 G80 架构上的执行

CUDA 是一个基于 SP(Stream Processor，流处理器)的并行众核(many-core)系统，如图 3(c)所示。在 Fermi 架构中，SP 称为 CUDA Core，如图 3(b)所示。作为最基本的执行单元，每个 SP 都包含一个全功能的 1DALU，该 ALU 可在一个周期内完成乘加(MADD)操作，多个 SP 构成了 CUDA 的计算单元 SM(Stream Multiple Processors，流多处理器)。CUDA 架构是完全基于标量的，用 4 个 SP 并行处理完成传统 GPU 的 4D 矢量操作，这种实现的最大好处就是灵活。对 1D，2D，3D，4D 指令，CUDA 都将其拆分成 1D 指令来处理。图 2 为一个普通矢量加法运算在 G80 编译器下的转换：一条 4D 的矢量指令，转换为相互独立的 4 条标量指令，并被分配到不同的 SP 上并行执行。

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。

3.2 AMD Cypress 架构

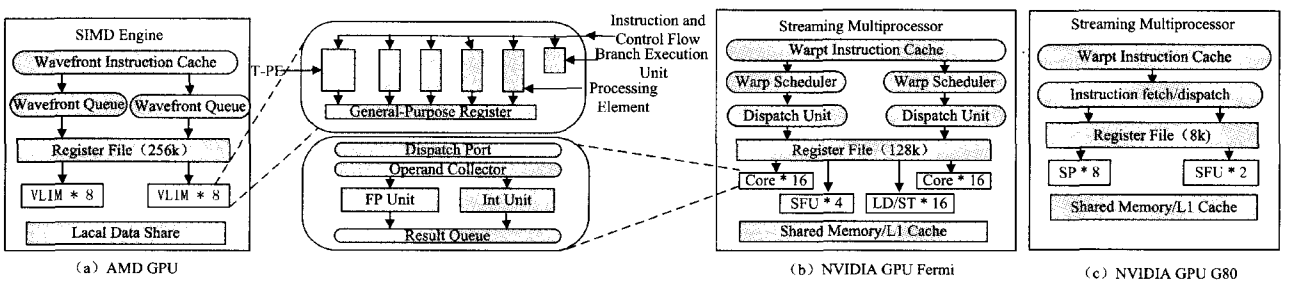
AMD GPU 沿用了传统 GPU 的 SIMD 架构，其基本执行单元称为 Stream Core，简称 SC，每个 SC 拥有 5 个共享指令发射端口的 1D ALU，如图 3(a)所示。SC 被组织为 5 路 VLIW(Very Long Instruction word，超长指令字)处理器，在每个时钟周期内，一条 VLIW 指令可同时执行 5 个标量操作。16 个 SC 组成了 AMD GPU 的计算单元 CU(Computer Unit，计算单元)，同一个 CU 内的所有 SC 都执行相同的指令序列。

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。

3.3 GPU 平台相关优化方法研究

本节主要讨论影响 OpenCL 程序在具体 GPU 平台上性能的两个主要因素：计算资源的有效利用率和硬件资源限制的优化方法，Global Memory 访存带宽利用率优化方法将在 5.1 节中讨论。



(a) AMD GPU　(b) NVIDIA GPU Fermi　(c) NVIDIA GPU G80

图 3 AMD GPU 与 NVIDIA GPU 底层硬件设计

3.3.1 计算资源的有效利用率

两个 GPU 平台在硬件设计上最大的不同是其基本执行单元的设计。在 AMD GPU 平台中，SC 采用了向量化的设计方式，如图 3(a)，每个 SC 是一个 5 路 VLIW 处理器，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。因此，向量化是提高 AMD GPU 计算资源利用率的有效方法。而对于 NVIDIA GPU 平台，由于 SP 是完全标量化的设计，编译器将向量化操作转化成相互独立的标量操作，因此，向量化操作对 NVIDIA GPU 平台影响不大。

虽然两个 GPU 平台在硬件设计上存在着很大的差异，但在整体架构设计和线程调度上两者存在很多相似性，因此有许多通用的提高计算资源有效利用率的优化方法：

a)调整线程组织结构，使 work-group 的大小为 wave-front 或者 warp 的整数倍；

b)每个 CU 上同时并发运行足够数目的 wave-front 或者 warp，以隐藏访存延迟；

c)消除分支对性能的影响；

d)充分利用共享内存、常量内存、Cache 等片上资源减少访问 Global Memory 的次数。

3.3.2 硬件资源的限制

在本文中，GPU 上硬件资源主要是指寄存器、共享内存和缓存等与 GPU 程序性能密切相关的硬件资源。

线程对 GPU 上硬件资源的使用数量在很大程度上决定了在一个 CU 上同时并发运行的 wavefront 或者 warp 的数量，从而影响 GPU 计算资源的利用和访存延迟的隐藏，进而影响程序在 GPU 上的性能。

以寄存器为例：

在 NVIDIA G80 架构中，每个 SM 共有 8k 的寄存器，而每个 SM 最多可以同时并发运行 768 个线程，那么每个线程分配的寄存器数量就为 8k/768 = 10 个。如果每个线程使用的寄存器超过 10 个，在一个 SM 运行的线程数量将会以 work-group 的粒度减少。如 work-group 大小为 256，当每个线程使用 11 个寄存器时，在每个 SM 上同时运行的线程数量就会变为 512 个，明显地减少并发运行的 warp 数量，从而无法有效地隐藏访存延迟，造成计算资源的浪费。在 AMD GPU 上，也存在着类似的情况。

在 GPU 平台上，编译器在很大程度上决定了寄存器的使用方式和使用数量。然而，我们可以通过调整程序结构，使用宏定义等方法，尽可能避免中间结果的产生等，节约使用寄存器资源。特别是在 AMD 平台上，一方面向量化是提高程序性能的主要方法，但是，向量化会增加寄存器的使用量。因此在实际的程序优化中，要通过多次的实验，选择合适的向量长度，使程序性能达到最优。

4 拉普拉斯图像增强算法优化

图像 Laplace 变换是基本图像增强算法，原始图像通过 Laplace 变化后会增强图像中灰度突变处的对比度，使图像中的细节部分得到增强并保留了图像的背景色调，图像的细节比原始图像更加清晰。基于 Laplace 的图像增强已经成为图像锐化处理的基本工具。

4.1 算法原理

图像 Laplace 变换的基本理论依据是 Laplace 算子。Laplace 算子是最简单的各项同性微分算子，具有旋转不变性。一个二维图像的 Laplace 算子是各项同性的二阶导数，定义为

$$\nabla^2 f(x,y)=\frac{\partial^2 f}{\partial x^2}+\frac{\partial^2 f}{\partial y^2} \tag{1}$$

该方程的离散形式为

$$\nabla^2 f(x,y)=[f(x+1,y)+f(x-1,y)+f(x,y+1)+f(x,y-1)]-4f(x,y) \tag{2}$$

图 4 是 Laplace 算子的模板表示形式，图 4(a)表示 Laplace 算子模板，图 4(b)表示 Laplace 算子扩展模板，图 4(c)则分别表示其他两种 Laplace 算子的实现模板。

0	1	0
1	-4	1
0	1	0

(a) Laplacian模板

1	1	1
1	-8	1
1	1	1

(b) Laplacian扩展模板

0	-1	0
-1	4	-1
0	-1	0

-1	-1	-1
-1	8	-1
-1	-1	-1

(c) Laplacian其他两种实现模板

图 4 Laplace 算子模板

图像锐化处理的作用是使灰度反差增强，从而使模糊图像变得更加清晰。图像模糊的实质就是图像受到平均运算或积分运算的影响，其高频分量被衰减，因此可以对图像进行逆运算，如微分运算突出图像细节，使图像变得更为清晰。Laplacian 是一种微分算子，它的应用可增强图像中灰度突变的区域，减弱灰度缓慢变化区域。因此，锐化处理选择 Laplacian 对原图像进行处理，产生描述灰度突变的图像，再将拉普拉斯图像与原始图像叠加而产生锐化图像。拉普拉斯锐化的基本方法可以由下式表示。

$$g(x,y)=\begin{cases} f(x,y)-\nabla^2 f(x,y), & \text{掩膜中心系数为负} \\ f(x,y)+\nabla^2 f(x,y), & \text{掩膜中心系数为正} \end{cases} \tag{3}$$

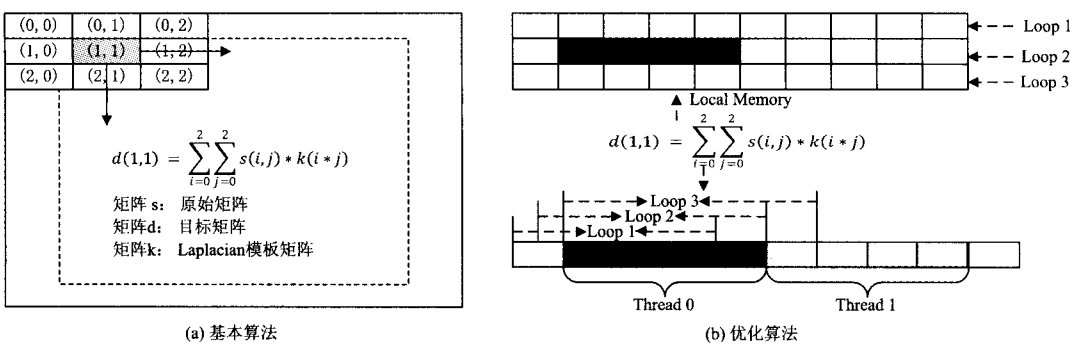
4.2 算法实现与优化

Laplace 变换的核心就是表示图像的二维矩阵根据 Laplacian 模板进行 9 点差分运算，Laplace 变换的目标矩阵和原矩阵不是同一个矩阵，因此，每个点的变换不存在读写依赖，完全可并行执行。

4.2.1 基本实现

在图像 Laplace 变换的基本算法中，每个线程负责一个元素，通过读取其周边的 8 个元素来计算目标矩阵相对应的元素，如图 5(a)所示。在 Laplace 变换中，矩阵元素分为边界元素和中心元素两种情况，在进行边界元素计算时，需要根据边界类型来确定，例如当边界类型是 Reflect 时，第一列元素的“左边”一列的值指矩阵的最后一列。这种基本实现算法的优点是简单直观，容易实现。但是，这种基本算法性能并不理想，主要原因有 3 点：

1)Global Memory 访存效率不高。在本文中，矩阵的数据类型为 char。每个线程处理一个元素，在 NVIDIA GPU 平台上，无法充分利用 DRAMs Line 的数据宽度；在 AMD GPU 平台上，数据对 Global Memory 的写入是通过低效的 CompletePath 进行的。



(a) 基本算法　　(b) 优化算法

图 5 Laplace 基本算法及优化算法

2)由于边界值的计算，产生大量分支。在 GPU 代码中，每个点的计算过程须始终保证一致，一旦产生分支，所有的执行路径就必须被串行执行，条件语句的总执行时间等于各个分支语句执行时间的总和。

3)CGMA ratio(Compute to Global Memory Access ratio)[16]过低。每计算1个元素,需要进行9次访存操作,大大增加了访问片外内存(Global Memory)的次数和开销。而对于相邻元素的计算,没有有效地进行数据重用。如计算(1,1)和(1,2)两个相邻元素时,会使用6个相同的元素。

为提高程序在GPU上的性能,根据算法特征和硬件特性,从提高Global Memory的访存效率,充分利用GPU计算资源以及合理使用GPU硬件资源3个方面对算法进行了优化,如图5(b)所示。

4.2.2 提高Global Memory的访存效率

Laplace算法是数据访存密集型算法。提高访存效率,减少访存延迟,对提高程序性能非常重要。

由于Laplace算法的存储访问模式是连续访存模式,因此,本文提高Global Memory的访存效率的最主要方法是向量化读取。5.1节详细讨论了在访存模式为连续访存的情况下,向量化对访存带宽利用率的影响。

对于NVIDIA GPU平台,可以充分利用DRAM Line的数据宽度,根据5.1节的访存带宽利用率的测试结果,当向量长度为4时可以达到较高的访存效率。对于AMD平台,当每个线程访问的字节数大于4个字节时,向量长度对访存利用率的影响较小,在不同向量长度下,都能达到较高的访存效率。

结合两个平台的特性,为了在两个平台上都能达到较高的访存利用率,本文采用长度为4的向量进行访存操作。

4.2.3 提高计算资源的有效利用率

在本算法中,主要使用了消除分支、使用共享存储器、常量存储器以及向量化的方法来充分利用计算资源。

消除分支,减少分支语句带来的性能损失主要有两种方法:

1)数据填充法。即根据边界类型,对矩阵进行边界填充,通过扩大矩阵规模来消除边界元素,将边界元素转化为中心元素。这种方法的优点是进行数据填充后,GPU代码实现简单,完全避免了分支对性能的影响。缺点是边界填充值在程序运行时才能根据边界类型进行确定(边界类型通常由用户作为参数传入),边界填充有一定的时间开销。

2)用?:语句代替if…else…语句,同时延缓取数操作。即在?:语句中计算操作数的地址,不进行任何实际的取值操作,在所有地址都确定之后,再从源矩阵中获取操作数。这样用额外的计算代替额外的访存,无疑会缩短执行时间,提高函数性能。本文采用了这种方法。

使用共享内存存储重用数据和中间结果,如图5(b)所示。将数据放在共享内存中,通过数据重用提高有效数据率,减少对Global Memory的访问次数和时间开销,同时使用常量存储器存储表示拉普拉斯算子的矩阵,减少对Global Memory的访问。

实际算法优化中,在两个平台上采用向量化计算方法都能显著提高函数性能,这是因为,在AMD GPU平台下,向量化可以充分利用5路VLIW计算资源;而在NVIDIA GPU平台下,虽然向量指令转化为相互独立的标量指令,不会对程序性能造成实质的影响,但是向量化操作使每个线程处理的元素增多,提高了有效数据比。通过在不同GPU平台上的测试,最终选用了长度为16的向量进行向量化计算。在实际算法设计中,将向量作为一个整体看待,以work-group为单位,统一计算向量各个分量,以利用GPU的计算资源。

4.2.4 硬件资源的限制

在本算法中,硬件资源的限制主要表现在寄存器的使用上。由于采用向量化算法,导致每个线程使用的寄存器数量大大增加,而且寄存器的使用数量会随着向量长度的增长而增加。实验表明,当向量长度不超过16时,在两个GPU平台上,寄存器的使用会成为硬件资源限制的主要因素。

4.2.5 向量化对性能的影响

在本算法中,向量化是主要的优化方法,但向量化也有其缺点和不足,主要表现在:

1)容易造成共享内存bank冲突。AMD GPU共享内存有32个bank,NVIDIA GPU有16个bank,每个bank都是4个字节,而AMD GPU与NVIDIA GPU访问共享内存的单位都是16个线程,也就是说每个线程访问共享内存的字节数大于4(NVIDIA GPU)或者大于8(AMD GPU)就会造成bank冲突。

2)增加GPRS(General Purpose Registers)的使用,从而减少在CU(Compute Unit)上同时并发执行的wavefront或者warp的数量。这可能会造成不能有效地隐藏访存延迟,同时不能充分利用CU上的ALU资源。

虽然向量化有这两个缺点,但从整体来看,采用向量化操作确实能够提高函数的性能,特别是对于AMD GPU。在实际的性能测试中,当向量的长度为16时,在两个GPU平台上,都能达到比较好的加速比。

5 性能评测

我们使用AMD 5850,NVIDIA Tesla C1060以及NVIDIA Tesla C2050(Fermi架构)这3个GPU平台测试程序的性能移植。在实际测试中,共进行了4个方面的测试:1)向量化访存在不同硬件平台上对Global Memory访存带宽利用率的影响;2)不同优化方法在不同GPU平台上对性能的影响;3)在两个NVIDIA GPU平台上,测试了与NVIDIA NPP库中相应实现的性能对比;4)3个平台上相对于CPU程序的加速比。

5.1 向量化对Global Memory访存带宽利用率的影响

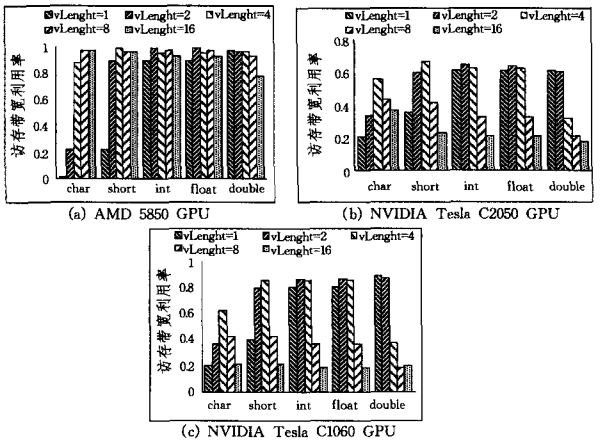
我们使用长度为1,2,4,8,16的向量实现copy函数,以此来说明向量化读取在3个平台上对访存带宽利用率的影响。通过实验发现AMD GPU平台向量化对访存带宽利用率的影响较小,NVIDIA GPU向量化对访存带宽的利用率影响较大,如图6所示。

图6(a)显示了在AMD 5850上,向量化对访存带宽利用率的影响:除了char和short外,向量化访存对带宽利用率的影响非常小,主要有两个原因:

1)在AMD GPU平台上,当kernel对Global Memory进行写操作时,CU与Global Memory之间上有两条相互独立的内存通道:Complete Path和Fast Path。Complete Path因需要进行额外的操作而在性能上与Fast Path差异非常明显。而当kernel存在着原子操作或者小于32bit的数据传输时,写入操作将通过性能低下的Complete Path进行。所以,

char1、char2 以及 short1 的带宽利用率非常低。

2) AMD GPU 的 Global Memory Path 一个时钟周期内支持 128bit 数据的传输，而 copy 函数的访存请求满足内存合并访问的要求，同一个访存请求内所有线程的数据访问通过一次数据传输就可以完成，合并存取后无论是否使用向量都能充分利用 128bit 的传输位宽，因此，向量化访存对 Global Memory 访存带宽利用率的影响非常小。



(a) AMD 5850 GPU　(b) NVIDIA Tesla C2050 GPU

(c) NVIDIA Tesla C1060 GPU

图 6　向量化对访存带宽利用率的影响

由此可以看出，在 AMD GPU 平台下，向量化读取对 Global Memory 访存带宽利用率的影响很小，向量化写入可显著提升 char 及 short 数据类型的写入效率，对其他数据类型的影响很小。

图 6(b)、(c)显示了在 NVIDIA Tesla C1060 和 C2050 平台上，向量化访存对带宽利用率的影响：向量化访存对两个平台的带宽利用率影响非常明显。原因有以下 3 点：

1)与 AMD GPU 不同，尽管在计算能力大于 1.2 的设备上已经放松了内存对齐的限制，但是内存对齐对 NVIDIA GPU 的 Global Memory 访问有着明显的影响。因此如果满足内存对齐要求的访存请求的性能都比较低，如 char1，char2，short1 等。

2)在 NVIDIA GPU 平台下，每个线程访问的字长只有是 1，2，4，8 和 16 字节的情况下，一次 Global Memory 访存请求才会被编译成一条访存指令。因此，当向量数据类型大于 16 字节的时候，带宽利用率会非常低，如 int8，double4 等。

3)在 Fermi 架构中，对 Global Memory 访存添加了 cache 功能，并且 L1 与 L2 的 cache line 都为 128 个字节，因此内存请求的单位也为 128bit。当内存请求大于 128bit 时，一次内存请求会被拆分为几次独立的内存访问请求。因此，在 Fermi 架构下，当一次访存请求字节数为 128bit 的倍数且向量数据类型不大于 16 时，访存效率最高。

因此，在 NVIDIA GPU 平台下，向量化读取通过影响内存对齐方式和每个线程访问的字长的方式明显影响了访存带宽利用率。

从以上的分析可以得出，在跨平台的程序优化中，向量化是提高访存带宽利用率的有效方法，通过调整向量长度，在 NVIDIA 平台下达到较高的内存访问效率时，AMD 平台也会达到较高的内存访问效率。

5.2　图像 Laplace 变换算法性能测试

5.2.1　不同优化方法在不同 GPU 平台上对性能的影响

由于不同优化方法在 NVIDIA C2050 GPU 和 C1060 GPU 两个平台的表现相似，因此，在本节中，只比较了 AMD 5850 和 NVIDIA C2050 GPU 两个平台下，不同优化方法对性能的影响。

表 1 说明了在 AMD 5850 和 NVIDIA C2050 GPU 平台下，图像大小为 2560 * 2560 时，分别采用常量内存、共享内存、分支消除、向量化和循环展开等优化方法后，kernel 性能的变化。

表 1　不同优化方法对性能的影响(单位 ms)

	基本实现	常量内存	共享内存	分支消除	向量长度				循环展开
					2	4	8	16	
AMD 5850	31.25805	30.17836	23.27605	18.86332	6.86626	4.65939	3.05269	2.72524	2.11236
NVIDIA C2050	15.2075	14.26536	7.38626	5.86295	3.388828	2.5951	2.91296	1.92605	1.93163

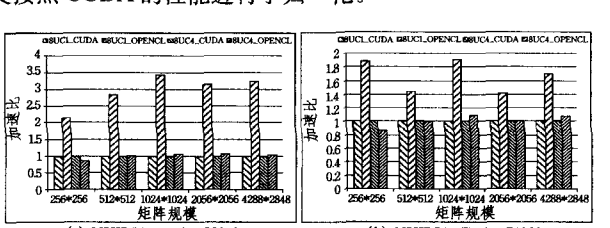
从表中可以得出，无论是AMD GPU还是NVIDIA GPU，通过使用常量内存、共享内存提高 CGMA ratio，通过分支消除有效利用 GPU 计算资源能有效地提高 kernel 的性能。向量化操作也能明显地提升 kernel 在两个平台上的性能，但在 AMD 平台下对性能的影响要显著得多，这是因为 AMD GPU 硬件架构是基于向量的，向量化能更加充分地利用 AMD GPU 的计算资源，而 NVIDIA GPU 硬件架构是基于标量的，向量化对其计算资源的利用影响不大，而性能的变化主要是由访存带宽的利用率引起的。循环展开在AMD平台下有效地提升了kernel的性能，而在NVIDIA GPU平台下对性能几乎没有影响。有两方面的原因，一是在两个平台下，编译器对循环自动展开的优化不同；二是循环展开后，增加了寄存器等硬件资源的使用，影响了程序的执行效率。

总之，GPU 程序优化，是多种因素综合的结果，是这样一种取舍的过程，即通过多次试验，找到影响程序性能主要因素的平衡点，使程序达到最优性能。

5.2.2　与 CUDA 实现的性能对比

为提高测试结果的可信性，在 CUDA 版本的选择上，使用了 NVIDIA NPP 库中对该函数的实现。

图 7 显示了在 NVIDIA Tesla C2050 和 NVIDIA Tesla C1060 两个 GPU 平台下，不同数据规模，不同图像类型(8UC1(黑白图像)；8UC4(彩色图像))，OpenCL 版 kernel 与 CUDA 版 kernel 的性能对比。为更好地比较两者的性能，本文按照 CUDA 的性能进行了归一化。



(a) NVIDIA Tesla C2050　(b) NVIDIA Tesla C1060

图 7　与 CUDA 实现性能比较

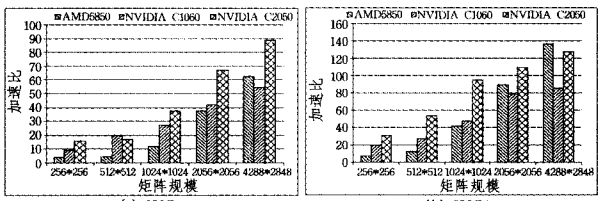
从图中可以看出，对于 8UC1 图像，优化后的 kernel 性能

普遍比 CUDA 版本提高了 1～2 倍，对于 8UC4 图像，两个版本的实现相差不大，性能基本相同，在数据量较小的情况下，CUDA 实现略好，但当数据量增大时，OpenCL 版本性能略高。这就证明了，优化后的性能并不比 CUDA 差，甚至比 CUDA 还要好。由于 NPP 库没有开源，因此无法知道 CUDA 版的具体实现方式，但从数据上分析，对于 8UC1 图像，CUDA 实现还有很大的优化空间。

5.2.3 与 CPU 实现的加速比测试

CPU 程序选用了 OpenCV 库中对该函数的实现程序，因为 OpenCV 库的 CPU 代码都经过了相当的优化，因此，能够比较客观地反映 OpenCL 实现的加速比。

图 8 显示了不同数据规模、不同图像类型下，图像 Laplace 变换在采用 GPU 加速后相对于 CPU 程序的加速比。在这里需要说明的是，OpenCL 程序的测试时间为程序的整体运行时间，即 CPU 代码的执行时间与 GPU 代码的执行时间之和，并没有计算数据在显存和内存之间的传输时间。之所以没有计算数据传输时间，原因有两点：一是本文的主要目的是 OpenCLkernel 的跨平台优化，添加数据传输的比较对本文而言没有任何意义，而不添加数据传输时间，可以显示 GPU 强大的加速能力；二是随着技术的进步，数据传输瓶颈会逐渐得到解决，或者通过不断提高 PCI-E 传输带宽，或者融合 CPU 和 GPU，即 APU 技术，使两者共享内存，最大程度地减少或消除数据传输时间。



(a) 8UC1　　(b) 8UC4

图 8　Laplace 算法加速比

从图中可以看出，GPU 加速比随着数据规模的扩大而增大，GPU 的良好性能来自于其所拥有的众多计算单元以及巨大的吞吐量提供多点计算。只有提供足够的计算量，充分利用 GPU 众多的计算资源，才能充分发挥其性能，这也是加速比随着计算规模增加而增大的原因。另一方面，GPU 加速程序在 3 个平台上都取得了良好的加速性能，充分验证了程序的跨平台性的高效移植性。

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：Global Memory 的访存效率、GPU 计算资源的有效利用率及其硬件资源的限制，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 的拉普拉斯图像增强算法，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，虽然在 AMD GPU 平台上要明显得多。实验结果表明在没有考虑数据传输时间的前提下，在 AMD 和 NVIDIA GPU 上都取得了 3.7～136.1 倍、平均 56.7 倍的性能加速，优化后的 kernel 比 NVIDIA NPP 库中相应函数也取得了 12.3%～346.7%、平均 143.1%的性能提升。

程序在 GPU 上的性能不仅与程序特征有关，而且与硬件特性密切相关。因此，根据硬件特性和程序特征建立性能优化模型，实现运行环境感知的自适应优化算法是本文将来的研究重点和方向。

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[Z]. Sept. 2010

[3] NVIDIA Corporation. OpenCL programming Guide for the CUDA Architecture V3.2[Z]. AUG, 2010

[4] AMD Corporation. Accelerated Parallel Processing OpenCL™ [Z]. Jan. 2011

[5] KHRONOS OpenCL Working Group. The OpenCL Specification V1.1[Z]. 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: 851-855

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

[10] Taylor R, Li Xiao-ming. 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, et al. Architecture-Aware optimization Targeting Multithreaded Stream Computing[C]// The 2nd Workshop on General Purpose Processing on Graphics 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]. IEEE Tra, 2011, 22(1): 105-118

[15] NCSABench[CP/OL]. http://www.ncsa.uiuc.edu/UserInfo/Perf/NCSAbench/, 2009-04-13

[16] Kirk D B, Hwu Wen-mei W. Programming Massively Parallel Processors[M]. Beijing: Tsinghua University Press, 2010

[17] Dav I B, Mart I B. Numerical inversion of the Laplace transform: A survey and comparison of methods[J]. Journal of Computational Physics, 1979, 3(1): 1-32

[18] The analysis of AMD parallel computing technology[EB/OL]. http://vga.zol.com.cn/192/1927855_all.html#p1929487

[19] AMD's Cayman GPU Architecture [EB/OL]. http://www.realworldtech.com/page.cfm? ArticleID=RWT121410213827&p=1

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