基于 OpenCL 的均值平移算法在多个众核平台的性能优化研究

庞 旭[1,3] 张云泉[1,2] 龙国平[1] 贾海鹏[1,4] 颜深根[1,2,3]
(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]
(中国科学院软件研究所计算机科学国家重点实验室 北京 100190)[2]
(中国科学院大学 北京 100190)[3] (中国海洋大学信息科学与工程学院 青岛 266100)[4]

摘 要 OpenCL 作为一种面向多种平台、通用目的的编程标准,已经对许多应用程序进行了加速。由于平台硬件和软件环境的差异,通用的优化方法不一定在所有平台都有很好的加速。通过对均值平移算法在 GPU 和 APU 平台的优化,探讨了不同平台各种优化方法的贡献力,一方面研究各个平台的计算特性,另一方面体会不同优化方法的优劣,在优劣的相互转化中寻求最优的解决方案。实验表明,算法并行优化前、后在 AMD 5850、Tesla C2050 和 APU A6-3650 上分别达到了 9.68、5.74 和 1.27 倍加速,并行相比串行程序达到 79.73、93.88 和 2.22 倍加速,前两个平台 OpenCL 版本相比,CUDA 版本的 OpenCV 程序达到 1.27 和 1.24 倍加速。

关键词 GPU,APU,OpenCL,均值平移算法

中图法分类号 TP302 **文献标识码** A

Research on Mean Shift Algorithm Using OpenCL on Multiple Many-core Platforms

PANG Xu[1,3] ZHANG Yun-quan[1,2] LONG Guo-ping[1] JIA Hai-peng[1,4] YAN Shen-gen[1,2,3]
(Laboratory of Parallel Software and Computational Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)[1]
(State Key Laboratory of Computing Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)[2]
(University of Chinese Academy of Sciences, Beijing 100190, China)[3]
(School of Information Science and Technology, The Ocean University of China, Qingdao 266100, China)[4]

Abstract As a general-purpose programming standard for multiple platforms, OpenCL has accelerated many applications. Due to the differences of different platforms in hardware and software environments, general optimization methods may not accelerate the application well for all. Taking the optimization of the mean shift algorithm on GPU and APU platforms as an example, the paper provided several insights on contributions of various optimization methods on different platforms. On one hand, we explored the architectures of different platforms. On the other hand, we compared the pros and cons of different optimization methods. Based on meticulous evaluations of the pros and cons, we looked for the optimal solution. Experimental results show that, on AMD 5850, Tesla C2050 and APU A6-3650 platforms, the optimized algorithm achieves 9.68×, 5.74× and 1.27× speedups, respectively, and 79.73×, 93.88× and 2.22× speedups compared to the serial version, respectively, and 1.27× and 1.24× speedups compared to the CUDA version OpenCV program for the first two platforms, respectively.

Keywords GPU, APU, OpenCL, Mean shift

1 引言

GPU 作为数值及科学计算领域的一颗新星,在大规模数据处理方面表现不俗,相比 CPU,它拥有更高的并行数据处理能力,其理论峰值也在不断飙升,已经远远超过了 CPU。而 OpenCL 作为一个开放的并行编程标准,已经获得多家硬件生产厂商的支持,可以在 CPU、GPU、DSP(Digital Signal Processor)以及其它一些支持 OpenCL 语言的平台上运行,因此具有广泛的用途和广阔的发展前景。

均值平移(Mean Shift)是图像处理领域的一种基础算法。它的概念由 Fukunage[1] 提出,最初含义是偏移的均值向量。之后,Yizong Cheng[2] 对其进行了推广,设定了一族核函

到稿日期:2012-10-19 返修日期:2012-12-19 本文受国家自然科学基金项目(60303020,60533020),国家自然科学基金重点项目(60503020),国家自然科学基金青年基金项目(61100072),国家"863"计划基金资助项目(2012AA010902),ISCAS-AMD 联合 fusion 软件中心资助。

庞 旭(1986-),男,硕士生,CCF 会员,主要研究方向为并行软件设计与实现,E-mail:pangxu010@163.com;**张云泉**(1973-),男,博士,研究员,博士生导师,CCF 会员,主要研究方向为高性能并行计算及并行数值软件;**龙国平**(1982-),男,博士,助理研究员,主要研究方向为计算机体系结构等;**贾海鹏**(1983-),男,博士生,主要研究方向为众核环境下的编程方法;**颜深根**(1986-),男,博士生,CCF 会员,主要研究方向为高性能并行计算。

数和一个权重系数，进一步扩大了算法的使用范围。文献[3]针对求取图像模板均值的问题，提出一种快速计算像素模板均值的方法，以降低计算复杂度。目前，均值平移算法主要应用在聚类分析、图像平移、图像边缘提取、图像分割[4,5]和目标跟踪[6]等方面。

本文基于OpenCL异构并行编程模型对均值平移算法进行了加速。通过在Tesla C2050和AMD 5850两种不同平台进行优化，从该实例角度探索平台特性对性能优化的影响以及OpenCL在各个平台上的一些差异。此外，还对程序在APU平台的性能进行了研究，侧面揭示了APU平台的一些特性。

本文第2节简要介绍NVIDIA Fermi架构、AMD Cypress架构、AMD Llano APU架构以及OpenCL编程模型的一些特点；第3节结合平台，针对算法特点，给出了具体的优化方案；第4节根据测试结果进行性能分析和总结；最后总结全文，并作下一步规划。

2 概述

2.1 平台介绍

2.1.1 NVIDIA Fermi架构

CUDA(Compute Unified Device Architecture，统一计算架构)[7]是NVIDIA推出的一种用于通用计算的GPU系统，是对该公司GPGPU的统一命名。这种系统的GPU经历了G80、GT200和Fermi 3代的发展。

CUDA架构采用了标量的运算模式，每个CUDA core(亦称流处理器)中只有一个ALU(整型)单元，每个时钟周期只能执行一次运算，执行N维矢量就需要N个周期完成。这种设计的好处是，运算单元的运算效率达到100%，而不会有空闲的单元。同时，NVIDIA采用异步方式将流处理器的频率和GPU核心频率分开，流处理器的频率大幅提升，从而提高了标量运算的运算速度。图1为Fermi架构下的CUDA Core设计。

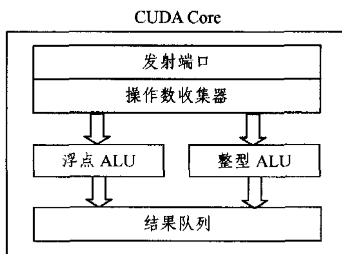

图1 Fermi CUDA Core结构

在第三代Fermi产品中，每个SM(Stream Multiprocessor)有32个CUDA Core，每16个为一组，由于配备有双warp调度器，每个周期能对两个宽度为32线程的warp(CUDA上线程调度的基本单位)进行排程和分发。

CUDA架构中设有纹理内存(Texture Memory)，它是一种只读存储器，由GPU用于纹理渲染的图形专用单元发展而来。纹理内存相比全局内存，有两级缓存，可以通过数据重用来加速数据的读取，因此相比从全局内存读取，速度要快一些，并节约了带宽，也无需按照全局内存对齐的要求访问。纹理内存相比常量内存，不仅支持一维数据，还支持二维及三维数据，十分适合图像数据的处理，同时它的容量比常量内存声明的64kB大很多。

2.1.2 AMD Cypress架构

AMD的GPU采用的是超标量设计，每个流处理单元(Stream Core，见图2)由5个ALU组成，它们共用一个指令发射端口。配合这一结构，采用了5路的超长指令字(VLIW)技术，将指令组合成适合5路运算的超长指令字，以充分利用流处理器的计算资源。这种执行模式被称为SIMD(Single Instruction Multiple Data)。

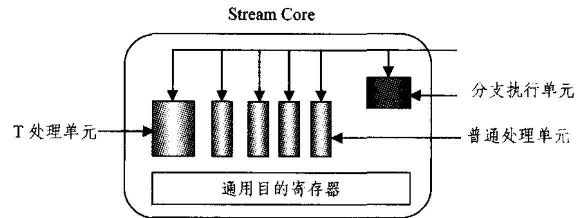

图2 Cypress Stream Core结构

每个流处理单元的5个ALU都可以执行基本的算术运算，而其中一个ALU单元比较特殊，除了能执行简单的算术运算之外，还可以处理一些复杂的运算，比如求差值、取倒数等。5个ALU之间相互独立，通过组合可以处理一维到五维的指令。在Cypress架构中，每个CU(Compute Unit)包含16个Stream Core。

AMD GPU的内存系统设有全局内存和常量内存。访问两种内存都可以使用缓存，以提高程序的读写速度。

2.1.3 AMD Llano APU架构

AMD APU(Accelerated Processing Unit)将中央处理器和AMD的GPU核心集成在一个晶片上，因此其同时具有高性能处理器和独立显卡的处理性能，既可以执行精密的标量运算，又可以执行大规模数据的并行处理。

APU内部的GPU核心没有独立显存，从GPU发出两条连接(见图3)，一条Fusion计算连接(Fusion Compute Link)，一条Radeon内存总线(Radeon Memory Bus)。计算连接将北桥模块(Unified North Bridge)、GPU、IO输入输出串联在一起，允许GPU访问一致性缓存，CPU和GPU之间的通讯时间缩短；而内存总线使得GPU可以通过高速带宽去访问系统内存，比单纯把GPU放在PCI-E总线上的延迟要小很多。

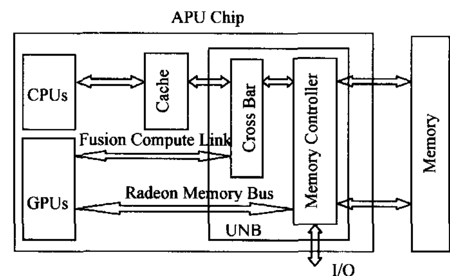

图3 APU芯片结构

代号"Llano"的A系列APU，拥有1个四核处理器，每个CPU核配备128kB一级缓存和1MB二级缓存。AMD A6-3650 APU内置HD6530D显示核心，配有320个流处理器单元。目前，平台并不支持image buffer对象，相关的一些操作是未定义的。

2.2 OpenCL简介

OpenCL(Open Computing Language)[8]是第一个面向异

构系统通用目的的开放、免费的并行编程标准，它提供了一个抽象层，消除了底层的硬件差异，具有平台独立性，可以在多种计算设备上执行，诸如 CPU、GPU、DSP 等。

OpenCL 有 4 种内存类型：私有内存、局部内存、常量内存和全局内存。内存结构如图 4 所示。4 种内存的访问速度整体呈降低趋势，存储容量呈递增趋势，常量内存因寻址方式的不同会优于或次于局部内存的访问速度。此外，全局内存和常量内存可通过缓存重用数据来提高数据访问速度。

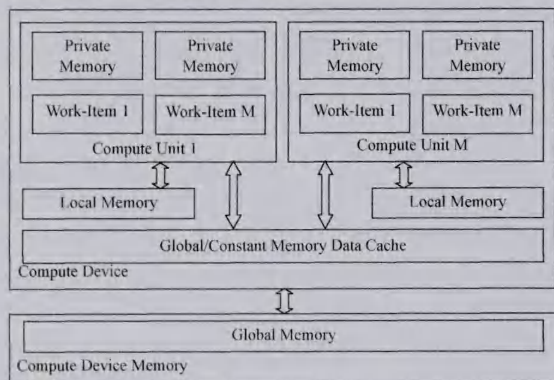

图 4 OpenCL 的 GPU 内存系统结构(引自[Khronos 2011])

image buffer 和 buffer 是 OpenCL 内置的两种内存对象。image buffer 只能读或只能写，而不能同时读写，使用 OpenCL 内建函数访问，支持随机访问，每个像素为四元矢量，尤其适合 4 通道图像数据，同时封装了越界处理和读取时滤波模式，简化了用户操作；buffer 顺序存储，可以同时读写，允许用户通过指针编程访问，适用于一维数据。

image buffer[9] 在 AMD 5850 和 Tesla C2050 平台都有使用 cache 来提高数据访问速度，buffer 在 AMD 5850 有用到 cache，在 Tesla C2050 中则没有这样的操作。

3 均值平移算法优化

Mean Shift 是 OpenCV 库中的均值平移处理函数，在取出的数据元素上执行较少的操作，因此属于访存密集的函数。其基本原理是，把图像中某一像素点的值用周围满足一定条件的像素点的均值替代。

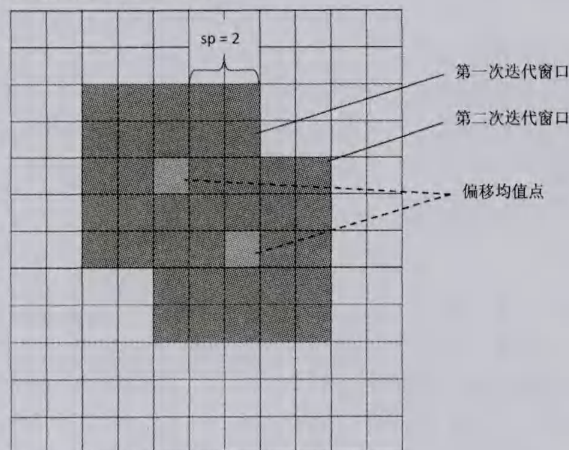

图 5 均值平移处理示意图

对于 8UC4 数据类型，计算只处理前 3 位的 RGB 数据(见图 6)。*sp* 控制窗口半径，*sr* 限制处理的色彩半径，均取整型。初始时，以所在坐标点(x,y)的像素值(r,g,b)作为偏移均值，当前坐标(x,y)作为偏移坐标。将(x,y)所在窗口内的像素点的值(r',g',b')分别和(x,y)的像素值(r,g,b)进行比较，如果满足$(r'-r)^2+(g'-g)^2+(b'-b)^2\leqslant sr^2$，则将该点坐标和像素值累计入新的偏移坐标和偏移均值。当遍历完窗口中所有像素之后，对累计求平均，用平均后的偏移坐标和偏移均值在新的窗口重复上面的过程。当偏移坐标不再变化，或偏移均值满足一定误差，或计算达到最大的迭代处理次数时，则停止。示意流程如图 5 所示。

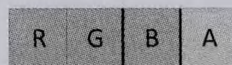

图 6 一个 8UC4 像素

3.1 NDRange 优化

线程的并行分为两种，一种是线程块内部线程之间的并行，属于细粒度的并行；一种是线程块之间的并行，属于粗粒度的并行。一个线程块只能在一个 SM 或 CU 上运行，合理组织线程块大小，使其形成一定的并行规模，有利于隐藏 ALU 延迟和访存时延。

CUDA 线程调度的基本单位是 warp，包含 32 个线程。CUDA 平台使用 Occupancy[9] 描述 SM(Stream Multiprocessor)并行时刻的资源使用率，它由 SM 上并发执行的 warp 数除以理论上 SM 可以并发执行的 warp 数而得。保持较高的 Occupancy 能使硬件资源得到高效的使用。通过设置一定量的 warp 来隐藏访存时延开销。

AMD GPU 的线程调度单位是 wave(或称 wave front)，每个 wave 最多支持 64 个线程。一个 wave 在一个 CU(Compute Unit)上独立执行，每个 CU 由 16 个 SC(Stream Core)组成。这意味着，每个时钟周期只能有 1/4 的 wave 执行，一个完整的 wave 需要连续的 4 个周期完成。AMD GPU 的 RAW (Read After Write)延迟是 8 个时钟周期，两个 wave 即可以隐藏这种 ALU 延迟。当一个 wave 因访存阻塞时，ALU 不必再等待 4 个周期，只要另一个 wave 准备就绪，就立刻执行。两个 wave 交替执行，使 ALU 一直处于忙碌状态，减少了等待时间。

Tesla C2050 和 AMD 5850 平台的 NDRange 优化效果明显，A6-3650 平台优化效果甚微。3 个平台每个线程块支持的最大线程数量分别为 1024、256、1024。由于算法的差异，每个块最佳的线程设置需要具体问题具体分析。通常，较合理的线程块大小设置为 128 或 256 个线程。这种合理性的标准是，存在一个临界值，在这个临界值之前增加线程数量，计算时间缩短；在临界值之后增加线程数量，计算时间保持不变或延长；超过临界值，安排过多线程会增加维护开销。

我们安排一个线程计算一个像素，图像有多少个像素就需要安排多少个线程。当图像行列值不能整除线程块的行列值时，会在每行和每列额外安排一个线程块，这无形中增加了一些空闲线程。合理设置线程块的行列组合，一定程度上可以有效减少空闲无用的线程。同时，在行主序的运算中，数据按行逐个存储，为了保证合并访问，应把一个 warp 或 wave 中连续的线程映射到矩阵的列。所以应尽量增加每列的线程数，避免列值设为 1 这种极端情况出现。通常，较为合理的列

值设置为16、32等，具体需视情况而定。

3.2 内存对象选择

image buffer专用于4通道的图像数据，封装了对越界访问的处理和图像的滤波模式，整体上写出来的kernel更为简洁。同时，无需考虑数据访问对齐，并且可以更好地隐藏寻址计算的延迟。这是image buffer在各平台的共性。

在Tesla C2050平台，image buffer使用CUDA Array。CUDA Array是为纹理拾取而优化的不透明内存布局，并自动绑定到纹理内存，通过纹理拾取来读取[10]，纹理内存使用了两级的纹理缓存，提高了对重用数据的访问速度；buffer绑定在全局内存上，全局内存未使用cache缓存，访问速度较慢。因此，CUDA平台image buffer是较理想的选择。

在AMD 5850平台，image buffer和buffer均使用了数据cache。所以在处理4通道数据时，两种内存的选择没有特别之处，都可以考虑。在处理非4通道数据时，我们更倾向于使用buffer对象，它的数据传输效率更好一些。

3.3 存储层次优化

每个线程初始需要读取一个像素单元，结束时需要将该单元处理结果写回，均值处理的中间过程需要访问单元周围$(x,y)\pm sp$窗口大小的像素集。而中间均值计算过程正是算法的瓶颈所在。我们结合实验平台Tesla C2050和AMD 5850对其进行了深入的分析。

均值平移算法计算过程中，相邻像素之间会出现一些冗余计算，能否用一个线程计算多个像素的均值，将冗余结果保存，以减少后面的重复计算?均值平移算法的难点在于：

1)窗口重叠不规律：每个像素点的均值计算是一个迭代的过程，中间可能会产生多个不同的偏移坐标，每个坐标对应的窗口和相邻像素点的窗口重叠区域是不同的，试图用一个线程计算相邻两个像素的均值并不能减少数据的读取量，总会有一部分数据要从全局内存读取。而如果让局存保存多个窗口，避免从全局读，对局存的容量是一种挑战，窗口大小可以不设限，但局存总是有限的。

2)冗余计算不规律：虽然相邻像素点的窗口会重叠，产生冗余计算，但这种冗余计算也是因数据而异的。在当前窗口范围，某像素(r,g,b)相对于中心像素(r_0,g_0,b_0)在sr^2范围内，但下一次，在新的窗口该像素相对于新的中心像素(r_0',g_0',b_0')不一定在sr^2范围内，况且像素也不一定在这个新窗口内。只有像素在sr^2范围，且(r_0,g_0,b_0)和(r_0',g_0',b_0')的对应坐标相等，这样计算出来的结果才称得上冗余，才有必要保存，显然这样的情况往往是不规律的，很难捕捉。

所以，最简单、有效的设置就是让一个线程计算一个像素的均值。

就像素集的存储位置而言，主要的备选方案有：

1)常量内存，适于存放只读数据，有多级缓存，支持多个线程块(warp或wave)同时访问，带宽很高，但容量有限，整个GPU最大只有64kB；

2)局部内存，访问速度快，让每个线程块共享自己取到的那部分数据，每个流式多处理器(SM或CU)有一个专属局部内存，Tesla C2050为32kB，AMD 5850最大为48kB；

3)直接从全局内存读取，全局内存容量大，访问速度最慢，但可以利用合并读(如NVIDIA CUDA)、合并写(如AMD GPU)来提高访问效率。

就本算法而言，最好的方案是第3)种。由于常量内存容量有限，通常图像数据规模是兆字节(MB)，因此将第1)种方法舍弃。在第2)种方法中，将初始化时每个线程读到的像素存入局部内存，同一个线程块的线程共享局存的内容。不过，线程要处理的数据并非全部在局部内存，总有一部分需要从全局内存读取，所以需要判断数据的读取位置。这样不仅增加了位置计算，而且访存冲突本质上没有减少，所以整体上kernel的运行时间并未缩短。

性能评测中我们选择第2)种和第3)种方法分别进行了对比测试。

3.4 指令优化

算法求均值的过程是一个迭代的循环，通过将最内层循环展开，提高每次循环的计算量，缩减循环次数，以减少循环开销。AMD 5850采用5路VLIW指令，在不存在数据相关的情况下，每个时钟周期可以并行执行5个数据操作。将循环展开，增加单次循环的计算比重，将更多的并行性暴露给编译器，能有效利用GPU的这种特性。Tesla C2050虽然没有这种计算特性，但是将计算展开以后，减少了循环执行条件的判断次数，优化了指令流，有效减少和消除了分支冲突，提高了warp的执行效率。Tesla C2050平台需要显式在循环语句前添加#pragma unroll N($N>1$，常数)制导语句，否则循环不能展开。

通常，循环展开最大以4次为宜。展开会用去较多的寄存器等计算资源，如果寄存器资源紧张，展开过多并不能带来性能的明显提升，同时使程序变得冗长，不利于阅读和维护。

在Tesla C2050平台和AMD 5850平台，全局内存访问速度很慢，减少访存次数和访存冲突成为优先考虑的问题。在线程访存数据量一定，每个线程对全局内存的访问频率很高的情况下，减少访问的次数，能有效减少访存冲突及由此产生的等待。指针访存分$n/4$($n=4$、8、16)次，每次取4个字节，而vloadn要按所取元素对齐(本例按1字节对齐)，因而产生的访存次数比指针矢量化访存多至少一倍。因此，在访存速度不高的情况下，为了减少访存次数，提高访存效率，优先使用指针访存。

在APU平台，GPU绕过PCI-E总线直接从全局内存取数，访问速度是可观的。在APU上使用vloadn指令的性能优于指针访问，这完全不同于前两个平台。由于APU上软件环境还不完善，因此尚不能借助剖分工具确定这种性能差异的根源。

4 性能分析与评测

本文分别在AMD Radeon™ HD 5850、NVIDIA Tesla C2050和AMD A6-3650 APU上进行了优化测试，使用的图像数据大小为2560 * 2560像素，每个像素类型为8UC4(4通道，每个通道8位无符数据)。在4.1、4.4节，sp取5，sr取6；在4.2、4.3节，sr恒取6，sp取5、10、15，对应窗口大小11 * 11，21 * 21，31 * 31。

此次优化，我们着重比较各种优化方法对kernel执行时间的影响。

平台设备参数如表1所列。

表 1 平台设备参数(GPU)

平台	CU数量	Core数量	内存带宽	存储器频率	单精浮点峰值性能
AMD 5850	18	288	128GB/s	1.0GHz	2.09Tflops
Tesla C2050	14	448	144GB/s	1.5GHz	1.03Tflops
A63650	4	64	29.8GB/s	0.8GHz	284Gflops

注:AMD A6-3650 APU 内置独显核心 HD 6530D

表 2 平台设备参数(CPU)

平台	CPU 型号	核心数量	主频	缓存(每核)
AMD 5850	AMD Phenom II X4940	1*4	0.80GHz	512kB
Tesla C2050	Intel Xeon X5550	2*4	2.67GHz	8MB
A63650	x86 系列	1*4	2.60GHz	1MB

4.1 NDRange 优化

首先,我们对比了线程块对程序的影响,如图 7 所示。

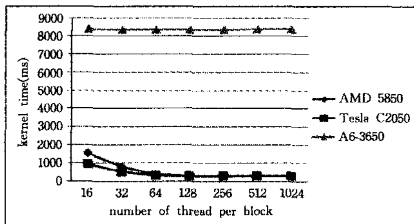

图 7 设置不同的线程块大小的函数性能对比

Tesla C2050 平台:当线程数量达到 128 个线程时,性能最好。进一步增加线程数量,运行时间反而有轻微上浮。从 Occupancy 率来看,当线程设置为 128 时,Occupancy 率是最高的(0.417);当线程数量小于 128 时,每个线程块占用资源减少,SM 上理论可并发的 warp 数量增加(分母增大);当线程数量大于 128 时,每个线程块占用资源增加,SM 上实际可并发的 warp 数量减少(分子减小),这些都会导致 Occupancy 率的降低。

AMD 5850 平台:由图 7 可知,与前述分析一致,当每个块线程设置为 128 时,运行时间达到最优。进一步增加线程,程序时间已基本趋于稳定。

A6-3650 平台:整体上,线程数量的变化对程序性能并没有产生特别明显的效果。经分析,可能是独显核心只有 4 个 CU,CU 内部的并行规模和 AMD 5850 是一样的,但 CU 之间的并行规模相比 AMD 5850 的 18 个 CU 略微弱了些,导致程序性能较差。为了便于后面的对比,设定线程块大小为 128。

其次,当线程块规模为 128 时,不同行列组合对程序性如图 8 所示。

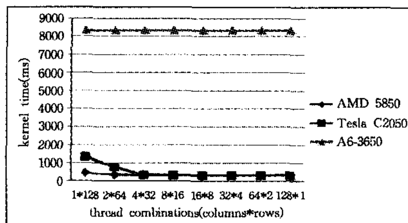

图 8 设置不同线程组合的函数性能对比

Tesla C2050 平台和 AMD 5850 平台:当列值设为 4 时,性能已经较为平稳。当列值设为 32 时,时间降到最低,之后性能将维持在这个水平上下轻微浮动。

A6-3650 平台:各种组合的时间差距不大。整体来说,行列值设置对程序影响甚微。

4.2 内存对象

从图 9 可以看出,两个平台使用 buffer 的性能相当,使用 image buffer 都比使用 buffer 性能要好,这种优势在 Tesla C2050 平台表现得更为明显一些。下面具体对各平台进行分析。

Tesla C2050 平台:纹理内存除了有高速缓存之外,不受访问模式的约束。而全局内存和常量内存需要遵循相应的访问模式才能获得较好的性能,每个 half-warp 访问的数据段首地址须按照 32、64 或 128 字节对齐,否则编译器就会将 half-wrap 的数据访问拆分成两次,而且 buffer 绑定的全局内存没有使用缓存。这些特性使得在 Tesla C2050 平台使用 image buffer 的性能优于 buffer。

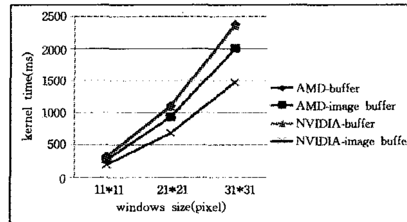

图 9 使用不同内存对象的函数性能对比

AMD 5850 平台:对 11 * 11 窗口大小的用例进行了剖分,从结果来看,使用图像对象,减少了每个线程执行的 ALU 指令数量,这主要归功于 image buffer 对边界检测的封装,无需在 kernel 中编写边界判断语句。另一方面,image buffer 内建函数独有的实现减少了从全局内存读取数据的冲突,使得访存冲突产生的停顿时间为 0,或者维持在 0.01%这样一个很低的水平。而使用 buffer 对象时,整个执行时间会有 5%的停顿。

AMD 的全局内存有两条写回路径,即 complete path 和 fast path[11],前者支持原子操作和非 32 位整数倍的数据操作,当不属于这两种情况时,可以通过后一条路径写回。在图像的读取和写回过程中,分别需要使用内建函数 read_imageui 和 write_imageui。read_imageui 会将每个通道的数据存储在一个 32 位的整型中。每个通道是 uchar 数据类型,在写回时,要将 32 位还原成 8 位的 uchar。由于通过总线传送的 4 个整型需要分别转换为 uchar 写回,使得写回走了 complete path 路径。总线数据有效率只有 25%。

尽管使用 image buffer 有优有劣,但整体上,其优势多于劣势。

A6-3650 平台:目前平台似乎并不支持 image buffer 对象。因此,我们没有在 APU 平台做相关对比。

4.3 存储层次优化

在这一环节,我们在 Tesla C2050 平台和 AMD 5850 平台使用 image buffer 的基础上,对比使用局部内存和不使用局部内存的性能;在 A6-3650 平台使用 buffer 来进行对比。结果如图 10、图 11 所示。

从图 10 可以看出,两个平台使用局部内存后,性能并没有提升。使用局存虽然增加了数据的共享性,减少了全局内存的访问,但由于数据的位置计算增加了 kernel 内部的运算,以及访问局存产生的访存冲突,导致整体计算速度并不比单纯访问全存的速度快。即使在性能差异不是特别明显的 A6-

3650平台也可以看出，故不使用局部内存的性能更好一些。

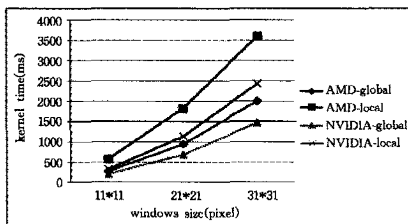

图10 使用和不使用局存的函数性能对比(NVIDIA和AMD)

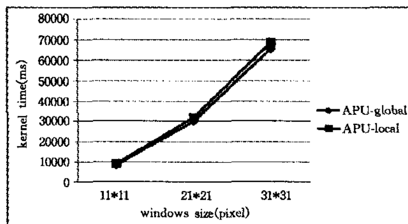

图11 使用和不使用局存的函数性能对比(APU)

注：由于图11的数据间隔较大，表面上看似乎使用global和local的性能差距不大，但实际差距水平与图10是一样的。

4.4 指令优化

4.4.1 循环展开

内存对象的使用和其他优化手段是相互作用的，在上面的优化中性能不是很好的buffer对象，结合其他优化手段，性能并不一定比image buffer差。在不展开的情况下，一次访存只取4个字节，通过在循环中取8字节和16字节将循环分别展开了2次和4次。访存方式为指针访问，没有使用vload操作。结果如图12、图13所示。

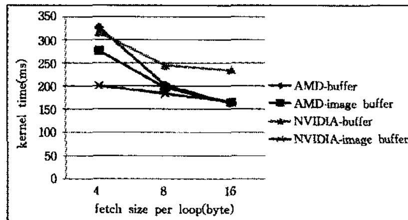

图12 循环展开的函数性能对比(NVIDIA和AMD)

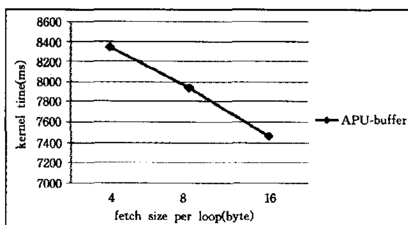

图13 循环展开的函数性能(APU)

Tesla C2050平台：在Tesla C2050平台，使用循环展开后，性能都有所提升，buffer对应的性能提升了34%，image buffer对应的性能提升了20%。

AMD 5850平台：从结果来看，随着展开次数的增加，kernel的时间逐渐缩短，当展开4次时，两种内存对象的性能趋于一致，buffer对象的性能追上了image buffer。对比程序的剖分结果发现，尽管使用buffer时访存产生的停顿时间仍然很高，但整体上ALU Packing率(编译器将ALU运算打包成5路VLIW指令的百分比)从46%(未展开)上升到77.61%(展开2次)，提高了ALU的执行效率(image buffer展开2次时ALU Packing率为67%)，同时数据写回时走fast path，总线数据有效率为100%，相比image buffer使用了complete path，写回速度更快。

A6-3650平台：APU平台使用循环展开后，性能提升了12%。虽然A6-3650的并行规模较小，NDRange优化没有明显的加速效果，但ALU Packing和fast path写回却是可以保证的。

4.4.2 访存指令对比

在使用全局内存的情况下，研究了vload和指针访存的一些差异。以展开4次进行说明，需要使用vload16和分4次指针访存(每次取4字节，uchar4)。分别对比了未展开、展开2次和4次时的性能，结果如图14、图15所示。

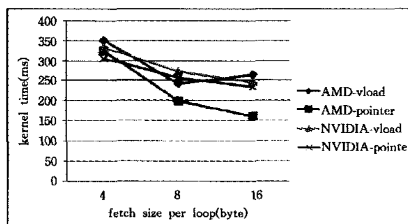

图14 不同访存指令的函数性能对比(NVIDIA和AMD)

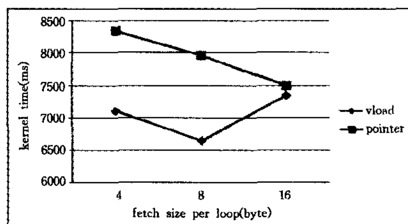

图15 同访存指令的函数性能对比(APU)

Tesla C2050平台：从图14可以看出，vload访存和指针访存时间曲线基本保持平行，而后者效果更为理想。

AMD 5850平台：vload的性能从展开2次后不升反降，以vload16的剖分结果来看，虽然ALU Packing率达到了84.25%，但访存操作相比后者增加了一倍，ALU操作也增加了一半，且因访存冲突产生了较长的停顿时间。这些都抵消了指令打包带来的性能优势。

A6-3650平台：A6-3650使用vload指令之后，性能获得了大幅度提升，在展开2次时，vload指令的访存效率达到了峰值，性能达到最优。当展开到4次时，两种访存方式的性能趋于一致。

4.5 总结

通过上面3个平台的优化，我们看到，由于平台的硬件特性和编译机制等，同一种优化方法在不同的平台可能出现不同的结果；不同的优化组合可能产生相同的优化效果；不同平台最终使用的优化方法可能不同。

最后对优化后的程序进行性能对比，对比了kernel的执行时间。

4.5.1 各种优化方法对程序性能影响

程序优化一开始使用的是全局内存，访问方式为指针访存，而优化过程中加入存储层次对比，旨在对比说明存储层次的选择对性能的影响。从图16、图17可以看出，GPU平台使用NDRange优化和内存对象优化之后，性能变化比较明显；而APU由于粗粒度(线程块之间)的并行规模较小，前两种优化方法并没有使性能有所改善，而循环展开和访存优化挖

掘了 APU 的硬件计算特性和数据传输特点，使性能发生了改观。

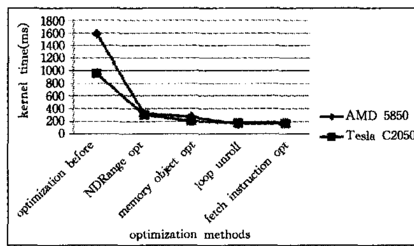

图 16 使用各种优化方法之后的性能走势(NVIDIA 和 AMD)

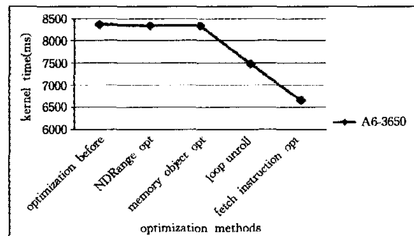

图 17 使用各种优化方法之后的性能走势(APU)

通过对比可知，AMD 平台性能加速最为明显，优化之后，AMD 和 NVIDIA 平台都达到了相当的性能，APU 平台整体计算时间仍然较长。优化后的算法在 AMD 5850、tesla C2050 和 APU A6-3650 平台分别达到了 9.68、5.74 和 1.27 倍加速。

4.5.2 各平台串行与 OpenCL 并行程序加速比

在 3 个平台运行相同串行版本的程序，结果如表 3 所列。与串行程序的性能相比，AMD 和 NVIDIA 平台都达到了几十倍加速，APU 平台只有 2 倍加速。我们认为，程序虽然可以利用循环展开提高程序执行效率，但与其他两个平台相比，APU 并行执行单元的数量还不具规模，无法利用程序潜在的并行性。其次，其没有相配套的多种存储系统，比如像纹理内存这种专门针对图像操作优化过的存储介质。因而 APU 上程序并行加速效果并不明显。

表 3 各平台串行 vs 并行加速比

平台	串行(ms)	并行(ms)	加速比
AMD 5850	13024.0	163.341	79.73
Tesla C2050	15699.0	167.225	93.88
A6-3650	14772.7	6650.40	2.22

4.5.3 AMD 和 NVIDIA 平台 CUDA 与 OpenCL 加速比

OpenCV 库中 CUDA 版本的程序是经过优化的高性能版本，将 CUDA 版本的性能与上面两个平台的 OpenCL 版本性能进行了对比，结果如表 4 所列。从对比结果来看，所提程序相比 CUDA 版 OpenCV 程序获得了 1.2～1.3 倍的加速。OpenCL 版本和 CUDA 版本的程序都使用了纹理内存，线程块设置为 32 * 8，这与程序在优化过程中设为 128 线程时的性能差异甚微。特别地，对 OpenCL 程序使用了循环展开。

表 4 AMD 和 NVIDIA 平台 CUDA vs OpenCL 加速比

平台	CUDA 程序(ms)	OpenCL 程序(ms)	加速比
AMD 5850	—	163.341	1.27
Tesla 2050	207.982	167.225	1.24

4.5.4 各平台数据传输时间对比

由于 APU 核心特殊的结构，我们特别对 3 个平台的数据传输时间作了对比，选择大小从 1280 * 640 像素到2560 * 2560 像素的图像进行了测试，每个像素 4 字节，对比结果如表 5 所列。AMD 5850 和 Tesla C2050 的显存规格均为 GDDR5，存储器频率分别为 1.0GHz 和 1.5GHz，而 LIano APU 作为一款使用 DDR3 规格显存的处理器，在频率不及前面两个平台的情况下，在数据传输时间上却表现出色，优于 AMD 5850，与 Tesla C2050 持平。我们分析，这得益于将 GPU 单元直接挂接在系统内存总线，省去了传统 GPU 计算中内存和显存之间通过 PCI-E 总线数据拷贝的时间。

表 5 各平台数据传输时间(ms)对比

平台	1280 * 640	1280 * 1280	2560 * 1280	2560 * 2560
AMD 5850	4.11	7.24	13.80	27.80
Tesla C2050	1.51	3.09	6.24	12.63
A63650	1.55	2.80	5.55	11.06

结束语 本文通过在 AMD 5850、Tesla C2050 和 A6-3650 平台使用不同的优化方法对均值平移算法进行优化，一方面演示了 OpenCL 优化的基本方法，另一方面展示了各个平台的一些计算特性，结合具体平台特点，综合运用各种优化技术提高程序的性能。

AMD 5850 平台采用 buffer 和 image buffer 都能获得较好的加速；Tesla C2050 由于其独特的纹理内存设计，使用 image buffer 效果更为理想。两个平台再结合 NDRange 优化、存储层次优化和指令优化等，最终均能获得较好的性能。而 APU 作为一个新出现的计算平台，其很多特性还没有被深入了解和挖掘。目前 CPU 和 GPU 还只是浅层次的物理融合，相应的软件环境还不完善，虽然优化之后的性能仍然不是很好，但作为一种尝试，也看到了它在访存这方面的优势。

我们已经使用 OpenCL 优化了多个算法的性能[12-14]，未来将进一步学习各个平台的优化技巧，深入研究 APU 平台的计算，提高程序在 APU 平台的性能。

参考文献

[1] Fukunaga K, Hostetler L D. The estimation of the gradient of a density function, with applications in pattern recognition[J]. IEEE Transactions on Information Theory, 1975, 21(1): 32-40

[2] Cheng Yi-zong. Mean Shift, Mode Seeking, and Clustering[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1995, 17(8): 790-799

[3] 夏永泉，徐洁，崔伟. 均值滤波中邻域均值的快速计算[J]. 郑州轻工业学院学报, 2008, 23(4): 57-59

[4] Comaniciu D, Meer P. Mean shift: a robust approach toward feature space analysis[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2002, 24(5): 603-619

[5] 孙小伟，李言俊，陈义. Mean Shift 图像分割的快速算法[J]. 测控技术, 2008, 27(7): 23-27

[6] Comaniciu D, Meer P. Real-time tracking of non-rigid objects using mean shift[C]// IEEE Conference on Computer Vision and Pattern Recognition, Vol. 2, 2000: 142-149

[7] http://zh.wikipedia.org/wiki/CUDA

[8] KHRONOS OpenCL Working Group. The OpenCL Specification[M]. v1.1, 2010

[9] NVIDIA Corporation. NVIDIA OpenCL Best Practices Guide[M]. v1.0, 2009

(下转第 110 页)

若上次 DGEMM 中发生情况 1，说明 R_{split} 过大，则需根据 *ratio* 减少 R_{split}。*ratio* 越小，R_{split} 减少量越大。具体算法流程图如图 7 所示。

需要注意的是，修正过程中可能因为 GPU 与 CPU 性能突变从而导致修正过度，使得 *ratio* 发散而不能收敛。出现这种情况时需要通过对 *ratio* 变化范围进行控制，在变化较大情况下可以进行适当限制。限制值需根据具体情况设定。

该算法使得 R_{split} 根据上次迭代计算的 *ratio* 以及 GPU 与 CPU 的性能比(P'_{gpu}/P'_{cpu})进行调整，有效减少了情况 2 的发生，可以使得单节点内多 GPU 情况下有明显的性能提升。采用该算法实验结果进行分析。

3.4 测试平台

本文使用浪潮 MF5588 服务器平台进行实验。每个节点的主要配置如表 1 所列。

表 1 MF5588 服务器软硬件配置情况

类别	项目	型号	个数
硬件	CPU	Xeon 5675 (6 core @3.068GHz)	2
	GPU	Tesla C2070	2
	内存	Samsung 8G DDR3 1333MHz	16
软件	OS	RHEL 6.2	—
	MKL	Intel MKL 10.2	—
	Compiler	Intel ICC 12.1	—

实验主要考虑在单节点内 CPU 总的计算能力 P_{cpu} 与 GPU 总的计算能力 P_{gpu} 比不同的情况下，原版 NVIDIA HPL、采用 ED 算法的 HPL、同时采用 ED 算法与 MA 算法两种算法的 HPL 的加速效果。测试代码是在基于 NVIDIA 针对费米架构的 HPL 最新代码 hpl-2.0_feimi_v11 上进行修改优化。

3.5 性能

在 P_{gpu}/P_{cpu} 不同的情况下对上述方法进行测试。测试结果如表 2 所列。

表 2 不同 P_{gpu}/P_{cpu} 情况下的测试结果

GPU 个数	CPU 核数	内存占用率	P_{gpu}/P_{cpu}	原始 (Gflops)	ED 算法 (Gflops)	ED 算法+MA 算法 (Gflops)
1	12	80% (N=100000)	2.4	389.6	391.2	391.4
2	12	80% (N=100000)	4.8	770.7	786.1	795.4
2	6	40% (N=70000)	9.6	625.2	639.9	663.9

其中，原始算法为 NVIDIA 提供的 hpl-2.0_feimi_v11 代码中使用的算法。

3.6 数据对比与分析

单节点内 CPU 总的计算能力与 GPU 总的计算能力比(P_{gpu}/P_{cpu})由节点硬件结构决定。当节点内 CPU 性能较强而 GPU 性能较弱时，(P_{gpu}/P_{cpu})较小，传统的同构集群(P_{gpu}/P_{cpu})=0；当节点内 CPU 较弱而 GPU 性能强，GPU 数量多时，(P_{gpu}/P_{cpu})较大，单节点 4 GPU 的情况下(P_{gpu}/P_{cpu})可以超过 11。

对表 2 数据进行分析。以 NVIDIA 原始算法为对比，比较采用 ED 算法的 HPL 以及同时采用 ED 算法与 MA 算法的 HPL 的加速效果，如图 8 所示。

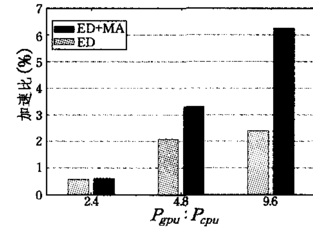

图 8 各算法在不同 P_{gpu}/P_{cpu} 时的加速效果

从图 8 可以看出，随着(P_{gpu}/P_{cpu})的上升，采用 ED 算法与 MA 算法得到的加速比也逐渐上升。当(P_{gpu}/P_{cpu})=9.6 时，同时采用 ED 算法与 MA 算法可达到 6.3%的加速比。

加速比变化趋势反映出 ED 算法与 MA 算法在节点内 GPU 数较多、性能较强的情况下可发挥出更大的优势。而 NVIDIA 本年底将推出的新一代开普勒架构 GPU 在性能方面将有 3 倍提升，(P_{gpu}/P_{cpu})将进一步增大，使得上述算法将取得更好的加速效果。

结束语 本文主要分析现有 GPU 加速的 HPL 代码运行时特性，提出该代码采用的动态负载均衡算法在单节点多 GPU 情况下不能很好地实现负载均衡。针对该现象，本文提出了经验指导的动态负载均衡算法(EB 算法)与多 GPU 自适应负载均衡算法(MA 算法)。上述两种算法在单节点多 GPU 情况下，相比 NVIDIA 最新针对费米架构的 HPL 可取得 6.3%的加速比，并通过对实验数据的分析判断上述算法可在单节点多 GPU 集群环境取得较好的效果。

参考文献

[1] NVIDIA CUDA Compute Unified Device Architecture Programming Guide[Z]

[2] Wang Feng, Yi Hui-zhan. Optimizing Linpack B enchmark on GPU-Accelerated Petascale Supercomputer[J]. Journal of Computer Science and Technology, 2011, 26(5): 854-865

[3] HPL-A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers[OL]. http://www.netlib.org/benchmark/hpl/

[4] Phillips E. CUDA Accelerated Linpack on Clusters, SC09[OL]. http://developer.nvidia.com/get-started-parallel-computing

[5] NVIDIA Tesla GPGPU- Introduction of tesla C2070 and C2050 [OL]. http://www.nvidia.cn/object/product_tesla_C2050_C2070_cn.html

[6] Linpack on NVIDIA GPU [OL]. http://tech.it168.com/a2010/0723/1081/000001081479_all.shtml

[7] NVIDIA. Fermi compute architecture whitepaper[Z]. 2009

[8] HPC Challenge Benchmarks[OL]. http://icl.cs.utk.edu/hpcc

(上接第 85 页)

[10] NVIDIA Corporation. NVIDIA CUDA 计算统一设备架构编程指南[M]. v2.0, 2008

[11] AMD Corporation. ATI Stream Computing OpenCL Programming Guide[M]. 2010

[12] 贾海鹏，张云泉，龙国平，等. 基于 OpenCL 的拉普拉斯图像增强算法优化研究[J]. 计算机科学，2011，39(5)

[13] 颜深根，张云泉，龙国平，等. 基于 OpenCL 的规约算法优化[J]. 软件学报，2011

[14] 张樱，张云泉，龙国平. 基于 OpenCL 的图像模糊化算法优化研究[J]. 计算机科学，2012，39(3): 260-264