基于OpenCL的图像模糊化算法优化研究

张 樱[1,2] 张云泉[1] 龙国平[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)[1]
(中国科学院研究生院 北京 100190)[2]

摘 要 现代GPU一般都提供特定硬件(如纹理部件、光栅化部件及各种片上缓存)以加速二维图像的处理和显示过程,相应的编程模型(CUDA、OpenCL)都定义了特定程序设计接口(CUDA的纹理内存,OpenCL的图像对象)以便图像应用能利用相关硬件支持。以典型图像模糊化处理算法在AMD平台GPU的优化为例,探讨了OpenCL的图像对象在图像算法优化上的适用范围,尤其是分析了其相对于更通用的基于全局内存加片上局部存储进行性能优化的方法的优劣。实验结果表明,图像对象只有在图像为四通道且计算过程中需要缓存的数据量较小时才能带来较好的性能改善,其余情况采用全局内存加局部存储都能获得较好性能。优化后的算法性能相对于精心实现的CPU版加速比为200～1000;相对于NVIDIA NPP库相应函数的性能加速比为1.3～5。

关键词 AMD GPU,Blur,OpenCL,图像对象
中图法分类号 TP302 **文献标识码** A

Research on Image Blur Algorithm Optimization Using OpenCL

ZHANG Ying[1,2] ZHANG Yun-quan[1] LONG Guo-ping[1]
(Laboratory of Parallel Software and Computational Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)[1]
(Graduate University, Chinese Academy of Sciences, Beijing 100190, China)[2]

Abstract Modern GPUs generally provide specific hardware(such as texture, grating components and various on-chip cache) to accelerate the 2D image processing and displaying process. Programming model defines specific APIs to facilitate image applications taking advantage of image-related GPU hardware, such as CUDA's texture memory and OpenCL's Images Object. Taking the optimization of image blur algorithm on AMD GPU as an example, the paper made a deep insight into the using of OpenCL's image object on image applications, especially its advantage and disadvantage compared to the more general optimization method based on global memory and the on-chip local memory. The experimental results demonstrate that the image object can provide better performance only when the processing image is four-channel and the amount of data to be cached is small. For other cases, optimizing with global memory and local memory can get better performance. After optimization, the speedup reaches 200x to 1000x in comparison with the well optimized CPU code, and the speedup over NVIDIA NPP version is upto 1.3x to 5x.

Keywords AMD GPU, Blur, OpenCL, Images object

1 引言

图像滤波在图像处理中有着非常重要的作用,可以对图像进行去噪、模糊、锐化等多种操作。Blur算法是图像滤波中最基本的一种,其作用是对图像进行模糊处理。生物医学、遥感图像处理等领域的数据规模经常达到TB甚至PB量级,单纯用CPU来进行计算耗时太长,无法满足要求。近年来随着GPU通用计算(GPGPU)的快速发展,我们现在可以利用GPU巨大的带宽和计算资源来对图像处理算法进行加速,从而满足大量数据计算的要求。目前主流的GPU通用计算标准主要包括NVIDIA CUDA[1]和后起的OpenCL[2]。对于基本的图像处理算法,CUDA和OpenCL都提供了专门的图像处理部件和相应的API供应用程序使用,如CUDA的纹理内存[13]和OpenCL的图像对象[9]。对于图像处理类算法,使用纹理(图像)部件是编程者最直接的选择。但是编程者也可以不用纹理(图像)部件而选择使用全局内存和片上局部存储进行优化。两种方法是不是前者一定更优是需要研究探讨的。OpenCL作为面向异构系统的开放工业标准,应用范围更加广阔,本文在AMD GPU平台上选择OpenCL来对Blur函数进行加速,并以该函数为例,探究两种编程方法的优劣。

Blur算法的基本原理是把图像中某点的值用该点周围一块区域的平均值来替代。对于输入图像$A(m*n)$和滤波窗

到稿日期:2011-04-13 返修日期:2011-07-28 本文受国家自然科学基金(60303020),国家自然科学基金重点项目(60533020),国家高科技技术研究发展计划(2006AA01A102)资助。

张 樱(1987－),女,硕士生,CCF会员,主要研究方向为并行计算,E-mail:zhangying913@gmail.com;张云泉(1973－),男,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,主要研究方向为计算机体系结构等。

口 $B(p*q)$，输出图像 $C(m*n)$，C 中第 i 行第 j 列像素值的计算公式为：

$$C[i,j]=\frac{1}{p*q}\sum_{x=-p/2}^{p/2}\sum_{y=-q/2}^{q/2}A[i+x,j+y] \quad (1)$$

对于彩色图像，式(1)中的 $A[i+x,j+y]$ 是一个向量，以向量的方式参与运算，各分量对应相加。对于一个 3*3 的滤波窗口，Blur 函数原理示意图如图 1 所示，其中 $C[i,j]=(A1+A2+A3+A4+A5+A6+A7+A8+A9)*1/9$。

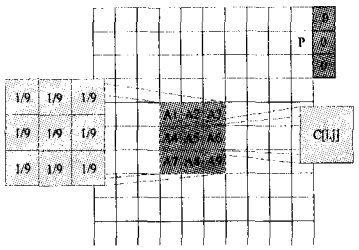


图 1 Blur 函数原理

目前国内外学者对 Blur 函数的 GPU 优化已经做了一些工作[3-5]，处理的数据类型均为 float 型，文献[3]中为了编码方便，舍弃了对图像边缘部分像素点的处理。日常生活中所见到的大部分图像都是彩色的，像素点类型并不完全是单一的 float 型数据。在 AMD GPU 中单字节的 char 型数据比四字节的 float 型数据更难处理和优化。因此本文主要针对 8UC1 和 8UC4(U:unsigned char,C:channel)的图像像素类型进行优化，其中 8U 指数据类型为 8 位无符号整数，C1 和 C4 分别表示像素通道数为 1 和 4，如黑白图像通道数为 1，RGBA 彩色图像通道数为 4。本文实验过程中计算图像边界像素时，将越界的像素点默认为 0，如图 1 中计算图像右上角的 P 点时需要用到超出图像边界的 3 个像素(灰色阴影部分)，我们用填充 0 来处理这种越界情况。

2 算法优化

OpenCL 标准将内核程序中用到的内存分为了几种不同的类型，分别为私有内存、局部内存、常量内存、图像对象(image object)和全局内存[9]。这几种内存类型在 AMD GPU 中对应的硬件资源分别为通用寄存器、LDS、常量内存、L1/L2 Cache 和全局内存[10]。在优化 OpenCL 内核程序时，根据算法本身的特点充分挖掘 GPU 存储层次的潜能，这是优化工作的重要部分。另外 OpenCL 规范中使用 NDRange 来定义 N 维索引空间，指定工作组空间中每个维度上工作节点的个数，针对算法特点选择合适的工作组大小也是至关重要的。

本文用 3 种方法对 Blur 函数进行了实现和优化：第一种使用图像对象存储图像，线程组内多个线程不用 LDS 共享数据；第二种使用图像对象存储图像并且使用 LDS 在线程组内共享数据；第三种使用全局内存存储图像且使用 LDS 共享数据。下面分别对这 3 种优化方式进行具体说明。

2.1 图像对象(Images)

OpenCL 标准为图像处理专门设置了图像对象类型，图像对象中封装了对越界的处理和对过滤模式的处理，而且有专门的读写图像对象的接口函数(如 read_imageui，write_imageui)，对编程者来说使用图像对象可以大大减少代码的复杂性。对于 Blur 函数，最简单、最自然的方式便是使用图像对象进行实现和优化。

单纯使用图像对象实现的程序性能并不理想，需要在这个程序的基础上进行优化。对于简单的 for 循环，AMD 的编译器可以自动进行循环展开，前提条件是循环次数必须在编译时是已知的，即控制循环开始和结束的量必须是常量。我们修改程序中的 for 循环，使其满足编译器自动循环展开的条件，经过测试性能提升一倍。这种循环展开方式对于后面的两种方法同样适用。

仔细分析算法，可以发现计算相邻两行同一列的两个像素点时，有部分数据和计算是重叠的，对于一个 7*5 的滤波窗口数据和计算的重叠如图 2 所示。计算第 2 行的点 P 和第 3 行的点 Q 时，灰色阴影部分的数据和计算是重叠的，我们可以利用这一特性继续对程序进行优化。安排每个线程计算相邻两行同一列的两个元素，这样做虽然减少了总的线程数量，但是相比于减少的数据读取和计算量，线程数量成为次要因素，程序性能有很明显的提升。

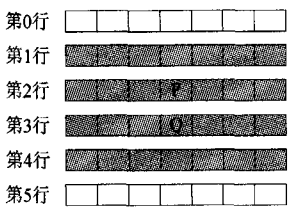


图 2 数据和计算的重叠

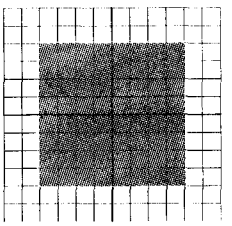


图 3 5*5 滤波窗口，8*8 线程组读取 12*12 像素

另外，对于 8UC1 类型的图像，我们使用 CL_R(或者 CL_A)格式创建图像对象，使用 read_imageui 函数读到的结果为 $(R,0,0,0)$(或$(0,0,0,A)$)。由于有用数据只有 R(或 A)，可以将这一分量从向量中提取出来单独进行计算，这样可以减少 3/4 的计算量，也可以提升部分程序性能。

对于 NDRange 的安排，我们尝试了 8*8 和 16*16 的线程组两种情况，实验结果显示 16*16 的线程组大小可以更好地发挥硬件性能。这主要是因为 GPU 上线程是以 wavefront 为单位并行执行的，GPU 发射一条指令后，一个 wavefront 中的所有线程并行执行这条指令。在 HD 5850 平台上一个 wavefront 有 64 个线程[10]，8*8 的线程组只包含一个 wavefront，当该 wavefront 因为访存操作被挂起时 GPU 资源只能闲置等待，而 16*16 的线程组有 4 个 wavefront，当 1 个 wavefront 被挂起时，其他 wavefront 可以继续执行，这样可以很好地隐藏访存延时。

2.2 图像对象+LDS(Images+LDS)

第 2.1 节中一个线程计算的两个同列相邻像素点可以重用数据和中间计算结果，这是线程内的数据重用。当计算同一行相邻两个像素点时也存在着数据重用的现象，这是线程间的数据重用。我们考虑将数据读取到 LDS 中使得同一个工作组内的线程可以共享数据。对于 5*5 的滤波窗口，8*8 的线程组计算 8*8 的像素点需要读取 12*12 的像素点，如图 3 所示。由于 8*8 的线程组读取的像素点个数为 12*12，因此需要合理安排线程读取这些点，这样做势必要取多次数据，每次取数需要使用分支语句来判断选择哪些线程进行取数操作。处于同一 waveront 中的线程遇到控制流分支时，不同的分支路径会被串行执行，因此大量的分支语句会导致程序执行效率降低。针对这种情况，我们尝试了多种取数和

计算的方法。

以 5 * 5 的滤波窗口和 8 * 8 的线程组为例，我们尝试过的方法包括：1)8 * 8 的线程取 8 * 8 的数据，只计算 4 * 4 的像素点，这种方法要求线程组大小必须大于滤波窗口大小；2)8 * 8 的线程取 12 * 12 的数据，计算 8 * 8 的像素点，取数方法为用线程组中前 6 * 6 的线程，每个线程取 4 个数；3)8 * 8 的线程取 12 * 12 的数据，计算 8 * 8 的像素点，取数方法为将二维的线程组和要读取的数据块均拉成线性的，线程组先读取前 64 个数据，接着读取后面的 64 个数据，最后用前 16 个线程读取最后的 16 个数据；4)8 * 8 的线程读取 32 * 32 的数据，每个线程读 4 个数据，计算 28 * 28 的像素点。我们分别采用这 4 种取数和计算策略实现 Blur 函数并进行优化，实验结果表明第 3)种方法最为简洁有效。

使用 LDS 势必要考虑 bank 冲突的问题，减少 bank 冲突一般可以从两方面着手进行：1) 减少每个线程存取 LDS 的数据量；2) 合理组织线程读写 LDS，尽量使不同线程访问 LDS 不同的 bank。实现 Blur 函数时，read_imageui 的结果为 uint4 类型，每个线程写入 LDS 一个 uint4 型的数据。HD 5850 GPU LDS 有 32 个 bank，每个 bank 4 字节，对 LDS 的访问是以 1/4wavefront 即 16 个线程为一组进行的[10]。连续地址模式下，每个线程向 LDS 写一个 uint4 型的数据势必会产生严重的 bank 冲突。为了减少 bank 冲突，采用了第 1)种方法进行优化。对于 8UC1 类型的图像，uint4 向量中只有 1 个分量有用，我们可以只存入这个 uint 型分量，16 个线程连续写入 16 个 uint 型的数据，无 bank 冲突；对于 8UC4 类型图像，read_imageui 的结果是 uint4，但是实际有效的数据是 uchar4。可以将 uint4 的数据转换为 uchar4 后再存入 LDS，16 个线程连续写入 16 个 uchar4(4 字节)不会产生 bank 冲突。

2.3 全局内存+LDS(Global Buffer+LDS)

第 2.1 节中提到了使用图像对象的好处，比如可以自动处理越界和滤波模式，具有缓存功能，编码简洁等。但是图像对象也有不足的地方，比如在处理 8UC1 类型图像时，1 个像素点会被封装成 1 个四分量的向量进行读写和参与计算，这无形中增加了无用数据的存取和计算。使用 Global Memory 可以更自由地处理各种长度的数据类型，可以更灵活地读写和处理数据。本小节中采用 Global Memory 来存取数据并用 LDS 实现线程间的数据共享，主要利用 AMD GPU 平台的向量特性，用向量存取和向量计算的方法来提升程序性能，以下对处理 8UC1 型图像和 8UC4 型图像分别进行说明。

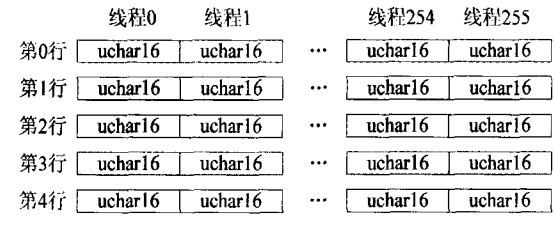


图 4 Global Memory+LDS，8UC1 类型，5 * 5 滤波窗口，线程读取数据示意

对于 8UC1 类型图像，我们采用 256 * 1 的线程组大小，每个线程以 uchar16 为单位读写数据。HD 5850 的内存总线带宽为 256bits[10]，以 uchar16 为单位读写数据刚好可以最大化地利用总线带宽。对于 5 * 5 的滤波窗口，每个线程读取 5 行同一列的 5 个 uchar16，线程读取数据示意图如图 4 所示。HD 5850 GPU 每个 CU 中有 32KB LDS，256KB 寄存器[10]，考虑到 LDS 的容量有限并且这些数据不需在线程间共享，我们将这些数据放到线程私有的寄存器中。

每个线程计算寄存器中 5 个 uchar16 的和，并将求和的结果写到 LDS 中，使线程间可以共享求和的结果，如图 5 所示。当滤波窗口 $k1 * k2$ 中 $k1 < 16$ 时，可以用 254 个线程来计算中间 254 * 16 个像素点的值。对 LDS 非对齐的访问，可以通过 vload 函数进行。当一个 for 循环中含有 vload 语句时，即使循环次数是编译时可知的，编译器也不会做自动循环展开。对此我们使用 # pragma unroll 指导语句进行强制循环展开。另外，第 2.1 节中提到的一个线程计算相邻两行同列的两个像素点的做法在这里同样可以使用，可以让一个线程计算相邻两行同列的两个 uchar16，这样做每个线程需要再多读取一个 uchar16 的数据，总共需要读取 6 个 uchar16。

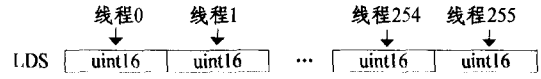


图 5 线程写 LDS 示意图

对于 8UC4 类型的图像，首先用类似 8UC1 的方法以 uchar16 为单位进行实现和优化，但效果不佳。改用 uchar4 为单位进行读写和运算后，由于不需要进行向量的拆分操作，代码变得简洁高效。线程组设置依然是 256 * 1。对于 5 * 5 的滤波窗口，每个线程读取 6 行同列的 6 个 uchar4 数据，计算其中的两个 uchar4 像素。基本算法与处理 8UC1 类型类似，此处不再赘述。

处理 8UC1 和 8UC4 类型图像都用到了向量操作，5 个 uchar16(uchar4)求和时需要用一个 uint16(uint4)型的变量来存储累加结果。OpenCL 标准中规定不同类型的向量不能直接参与运算，uchar16 和 uint16 型的变量相加时，必须要将 uchar16 型的变量用 convert_uint16 强制转换成 uint16 型后才可运算。而 for 循环中包含 convert 语句时编译器不能自动循环展开。针对这种情况，在将 uchar16 的数据取到寄存器中时就将其转换为 uint16 的数据存储起来，为后面直接进行向量求和做准备，这样做可以大大提升程序性能。

在 3 种实现方式的优化过程中，都需要注意指令流的优化。比如：开销较大的整数除法和取模运算可以用位运算来代替；对于精度要求不高的操作，可以用相应的内建函数来代替标准函数[12]；简单的 if else 控制流指令可以用条件表达式来代替。

3 实验环境及评测方法

3.1 实验平台及环境

在测试程序性能时，用 OpenCL 实现的 Blur 函数与 OpenCV 库中的 Blur 函数功能和接口均应一致。OpenCV 库[6]中包含了 CPU 版本的 Blur 函数实现和 CUDA 版本的实现。本文主要针对 AMD Radeon™ HD 5850 架构进行函数实现和优化。为了测试函数性能，将 OpenCL 版的 Blur 函数分别与 CPU 版和 CUDA 版的 Blur 函数进行了对比。在此需要说明的是 OpenCV 库中 CPU 版本和 CUDA 版本的代码均是已经经过优化的高性能版本，其中 CUDA 版本的实现调用了 NVIDIA 开发的高性能 NPP 库[11]。鉴于 CUDA 版的程

序只能在 NVIDIA GPU 平台上运行，我们用配置更高 NVIDIA Tesla C2050 平台进行了比较。两个平台的具体性能指标见表 1，程序的运行环境见表 2。

表 1　HD 5850 与 Tesla C2050 设备参数

GPU 型号	核数	单精度浮点峰值性能	双精度浮点峰值性能	带宽
HD5850	288	2.09Tflops	0.418Tflops	128GB/sec
Tesla C2050	448	1.03Tflops	0.515Tflops	144GB/sec

表 2　程序运行环境

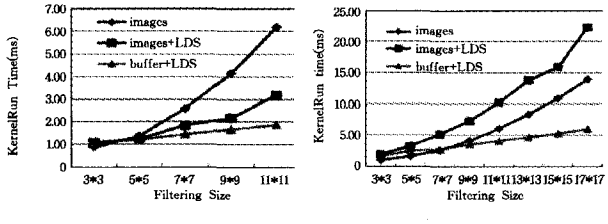
程序	CPU	GPU	GPU SDK	OS
CUDA	Intel® Xeon® X5550 2.66GHZ	NVIDIA Tesla C2050	NVIDIA CUDA SDK 3.2.16	Ubuntu 10.10
OpenCL	AMD PhenomII X4940 0.8GHZ	AMD HD 5850	ATIStream SDK 2.3	Ubuntu 9.04

3.2　性能对比方法

现有的 GPGPU 通用计算的流程一般是 CPU 处理少部分逻辑运算，CPU 端通过调用 API 函数将大部分的计算任务交给 GPU 处理。由于 CPU 内存和 GPU 显存位于不同的位置，程序开始时需要将数据从 CPU 端内存传到 GPU 显存上，并且在程序结束时还需要将执行结果从显存拷回到内存中。CPU 和 GPU 间的数据传输是通过 PCIe 总线进行的，尽管 PCIe 总线的带宽已经达到 8GBps[7]，但是对于计算量稍小的程序来说，数据传输仍可能成为程序的主要瓶颈。将 CPU 内存和 GPU 显存融合实现地址空间的统一，是未来 GPU 架构发展的必然趋势，AMD APU[8] 在这方面就已经做出了初步的尝试。未来的 GPU 计算必定不用再考虑数据的传输问题。当然对于某些大型的应用，只进行一次数据传输便可进行大量的数据运算，这种情况下数据传输的开销也可以忽略不计。因此本文中给出的加速比只包括纯 kernel 计算时间，不包括数据的传输时间。OpenCL 版 Blur 和 CUDA 版 Blur 对比时，由于二者均有数据传输的开销，因此只比较了纯 kernel 执行时间。两个版本的 Blur 函数 kernel 执行时间分别通过各自的 profiler 得到。

4　性能评测和分析

我们对第 3 节中提到的 3 种实现方法分别进行了测试，结果如图 6 所示。



(a) 8UC1 图像类型　　(b) 8UC4 图像类型

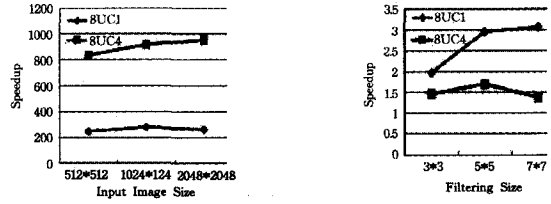
图 6　3 种方法实现的 Blur 函数性能

图 6(a)示出 3 种实现方式处理 8UC1 类型图像的性能测试结果。当滤波窗口为 3 * 3 时，images 实现方法其性能略好于其他两种实现方式。这主要是由于 images 具备缓存功能，当滤波窗口较小时，cache 命中率高，性能比其他两种方法稍好。随着滤波窗口的增大，images 的 cache 命中率下降，使用 LDS 对读取 images 的结果进行缓存，在线程组内共享数据，提高了数据的利用率，这种方法其性能优于单纯使用 images 的程序的性能。从图 6(a)中可以看到使用 buffer+LDS 方法实现的函数性能最好，一方面是因为对于 8UC1 类型的图像 images 将单通道的数据封装成四分量的向量进行读写和计算，无形中增大了数据存取量和计算量，而使用 buffer 可以更灵活地组织数据进行向量读取和向量操作；另一方面，使用 LDS 对数据进行共享，提高了数据的利用率，滤波窗口变大时可以完全替代 images 的缓存功能。

图 6(b)示出 3 种方式处理 8UC4 类型图像的测试结果。images+LDS 的方法完全没有任何优势，滤波窗口比较小时，images 方法比 buffer+LDS 方法好，而当滤波窗口大于 7 * 7 时，buffer+LDS 的方法性能更优。images 在处理 8UC4 类型的图像时有着得天独厚的优势，它可以自动处理越界的情况，以四分量向量为单位对数据进行读写和计算，并具有特殊的缓存功能。当滤波窗口较小时，这些优势得到充分的发挥，其性能优于其他两种方法。随着滤波窗口的增大，cache 命中率降低，数据利用率随之下降，而此时使用 LDS 共享数据仍可维持较高的数据利用率，buffer+LDS 方法胜出。

总的来说，当滤波窗口较小时，使用 images 方法程序性能较高；当滤波窗口大于 7 * 7 时，使用 buffer+LDS 的方法可以取得更好的性能。下面给出滤波窗口较小时，使用 images 方法实现的程序相对于 CPU 和 CUDA 版 Blur 的加速比，如图 7 所示。图 7(a)显示的是滤波窗口为 3 * 3 时，images 方法相对于 CPU 版 Blur 函数的加速比；图 7(b)展示的是对于 2560 * 2560 大小的输入图像，images 方法相对于 CUDA NPP 版 Blur 函数的加速比。

从图 7(a)可以看出，处理 8UC4 类型的图像，GPU 的优势更加明显。图 7(b)则显示用 OpenCL 实现的 Blur 函数性能至少与 NPP 实现的 Blur 函数相当，甚至可以比 NPP 版更优。



(a)相对 CPU 版 Blur 的加速比　　(b) 相对 NVIDIA NPP 版 Blur 函数的加速比

图 7　加速比

结束语　本文结合 AMD 平台上 3 种实现和优化 Blur 函数的方法，通过对比分析给出了使用 images 和 buffer+LDS 两种编程方式的优劣。只有当图像为四通道且滤波窗口较小，片上需要缓存的数据量较小时，images 的优势才得以充分发挥，代码简洁且高效；而当滤波窗口较大时，使用 buffer+LDS的方式通过合理安排数据的向量读取和向量化计算，能够更好地发挥 AMD GPU 硬件架构的性能，这种情况下，buffer+LDS 的方式完全胜过 images 方式。buffer+LDS 的方式可以更灵活地处理数据，且性能完全可能超越 images 方式，编程者写内核程序时可以不用只局限于 images 的方法。在优化函数的过程中，针对算法的具体特征合理安排 NDRange，尽可能地重用数据和中间计算结果，灵活进行向量运算和循环展开。优化过的 Blur 函数性能可以达到 CPU 版函数性能的 200～1000 倍，可以达到 NVIDIA NPP 版 Blur 函

数性能的1.3～5倍。

参考文献

[1] NVIDIA Corporation. NVIDIA CUDA C Programming Guide version 4.0[R]. 2011

[2] KHRONOS group. OpenCL-The open standard for parallel programming of heterogeneous systems [OL]. http://www.khronos.org/opencl/

[3] Udeepta D, Bordoloi. Optimization Techniques: Image Convolution [R]. 2011

[4] Andrade D. Case study: High performance convolution using OpenCL __local memory [OL]. http://www.cmsoft.com.br/index.php? option=com_content&view=category&layout=blog&id=142&Itemid=201

[5] Kong Jing-fei, Dimitrov M, Yang Yi, et al. Accelerating MATLAB Image Processing Toolbox Functions on GPUs [M]. Pittsburg, 2010: 75-85

[6] OpenCV: Open Source Computer Vision library [OL]. http://opencv.willowgarage.com/wiki/

[7] NVIDIA Corporation. NVIDIA OpenCL Best Practices Guide version 3.2[R]. 2010

[8] AMD Corporation. AMD Fusion™ Family of APUs: Enabling a Superior, Immersive PC Experience [R]. 2010

[9] Khronos OpenCL Working Group. The OpenCL Specification Version 1.1 [R]. 2010

[10] AMD Corporation. AMD Accelerated Parallel Processing OpenCL™[R]. 2011

[11] NVIDIA Corporation. NPP: NVIDIA Performance Primitives library [OL]. http://developer.nvidia.com/npp

[12] AMD上海研发中心. 跨平台的多核与众核编程讲义 OpenCL 的方式[M]. 2010: 59

[13] Sanders J, Kandrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming [M]. 北京: 清华大学出版社, 2010: 115-137

(上接第 250 页)

的分解层数均为5，各方法得到的融合结果如表1所列，局部放大结果见图3。

表1 融合结果的客观指标比较

	波段	IHS法	PCA法	小波	本文方法
UIQI	蓝	0.7730	0.8416	0.9057	**0.9099**
	绿	0.8615	0.9129	0.9464	**0.9475**
	红	0.6790	0.9548	0.9716	**0.9797**
	近红外	0.8917	0.9605	0.9726	**0.9728**
	均值	0.8013	0.9175	0.9491	**0.9501**
Q4	全部波段	0.6722	0.7228	0.7370	**0.7382**
平均梯度	蓝	6.5379	6.9354	7.0113	**7.2491**
	绿	6.3583	6.9047	7.0047	**7.2392**
	红	5.8647	6.9699	7.0976	**7.2008**
	近红外	**11.1818**	9.9872	9.9261	9.9923
	均值	7.4857	7.6993	7.7599	**7.7702**

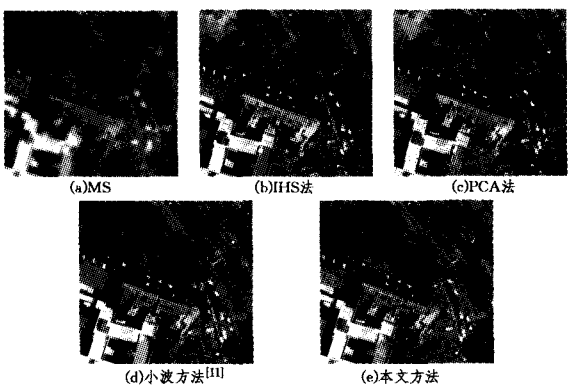


图3 融合结果局部放大图

从图3可以看出，提出的融合方法能得到富含空间细节信息的比较清晰的融合图像。而IHS法和PCA法存在一定的光谱扭曲，对光谱分析不利；而小波变换的空间分辨率差于本文提出的方法。

表1的客观评估结果证实了我们的主观评估，即本文方法在光谱信息的保留程度和空间清晰度方面都优于其余几种方法。

结束语 作为多尺度几何分析工具之一的 Directionlets 已经被证明能够比小波变换更好地捕获图像的方向信息。本文提出了一种新的基于 Directionlets 和 PCA 的全色和多光谱图像融合方法。将待融合的低空间分辨率多光谱图像与低光谱分辨率的全色图像均进行线性 PCA 变换之后，使用 Directionlets 提取全色图像的空间细节信息，将其“注入”到多光谱图像的主分量中，从而能够得到更加清晰并且光谱分辨率较高的融合图像。实验结果表明，在 UIQI 指数、整体图像质量指数 Q4、平均梯度等主观视觉效果和客观评价指标上，该方法均优于基于小波变换的方法。

参考文献

[1] 王建梅，李德仁. QuickBird 全色与多光谱数据融合方法用于土地覆盖分类中的比较研究[J]. 测绘通报，2005(10)：37-40，43

[2] 陈蜜. 基于独立分量分析的影像信息融合方法与应用研究[D]. 武汉：武汉大学，2006

[3] Audicana M G, Saleta J L, Catalan R G, et al. Fusion of Multispectral and Panchromatic Images Using Improved IHS and PCA Mergers Based on Wavelet Decomposition [J]. IEEE Transactions on Geoscience and Remote Sensing, 2004, 42(6): 1291-1299

[4] Wang Z J, Ziou D M, Armenakis C, et al. A comparative analysis of image fusion methods[J]. IEEE Transactions on Geoscience and Remote Sensing, 2005, 43(6): 1391-1402

[5] Leung H H, Li Z Z. Fusion of multispectral and panchromatic images using a restoration-based method: geoscience and remote sensing[J]. IEEE Transactions on Geoscience and Remote Sensing, 2009, 46(5): 1482-1491

[6] Zheng S, Shi W Z, Liu J, et al. Remote sensing image fusion using multiscale mapped LS-SVM[J]. IEEE Transactions on Geoscience and Remote Sensing, 2008, 46(5): 1313-1322

[7] Velisavljevi V, Beferull-Lozano B, Vetterli M, et al. Low-rate reduced complexity image compression using Directionlets[C]// IEEE International Conference on Image Processing, ICIP'06. Atlanta, GA USA, 2006: 1601-1604

[8] Velisavljevi V, Beferull-Lozano B, Vetterli M. Space-frequency quantization for image compression with Directionlets[J]. IEEE Transactions on Image Processing, 2007, 16(7): 1761-1773