GPU平台上ADL算法的实现

陈加忠,夏 涛,欧阳昆,黎 单,孙自龙
(华中科技大学 计算机科学与技术学院,湖北 武汉 430074)

摘 要:自适应方向提升小波变换(ADL)利用图像纹理特征进行变换编码,从而获得更高的编码质量,但同时也增加了计算复杂度。为了提高图像编码速率,在统一计算设备架构(CUDA)的图形处理器(GPU)上,提出一种并行实现ADL中的插值和方向变换计算的新方案,对插值部分同时采用粗粒度和细粒度的并行,即把图像数据分成若干个块进行粗粒度的并行,而对块中的每个像素点采用细粒度的并行。对变换部分中的9个变换方向采用粗粒度的并行。实验表明,在GPU上并行实现ADL变换是CPU实现的4倍左右,CPU-GPU整体架构下的ADL变换编码的速度是CPU平台下的3倍左右。

关键词:GPU;并行;提升小波变换;图像编码
中图分类号:TP301.6 **文献标识码**:A **文章编号**:1673-629X(2011)01-0165-04

Implementation of ADL Algorithm on GPU

CHEN Jia-zhong, XIA Tao, OUYANG-Kun, LI Dan, SUN Zi-long
(College of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China)

Abstract: In order to gain much better image quality, ADL (adaptive directional lifting) wavelet transform takes use of the texture property of image to implement the transform coding at the cost of high computation complexity. Implement the interpolation and directional lifting transform of ADL in parallel on GPU (graphic processing unit) with CUDA (compute unified device architecture) to speed up the image encoding. Both fine-grained and coarse-grained granularity parallelization are used for data block and pixels respectively in interpolation, while only coarse-grained granularity is used in nine directions for transform. Experiments results show that implementation of ADL on GPU is 4 times faster than that on CPU. The total time of ADL transform image coding on CPU-GPU framework is almost 4 times faster than on CPU.

Key words: GPU; parallelization; lifting wavelet transform; image coding

0 引 言

图像压缩编码的关键在于去相关方法的效果,2-D小波变换由于其能通过正交分解来消除图像相关性,因而得到广泛的关注。然而,自1980年代2-D小波变换用于图像压缩编码以来,其编码结果一直面临如下尴尬局面:如果给高频子带多分配一些比特,编码的压缩比上不去;如果少分配一些比特,编码质量就变差。究其原因,主要是2-D小波只做两个方向的分解,只能有效地消除0°、45°和90°方向的图像纹理的相关性,而不能有效消除其它方向的图像纹理的相关性,以致部分低频能量混入了高频子带。

近年来,出现了一类沿着图像内容轮廓、纹理方向进行小波分解的图像编码方法,以克服2-D小波变换编码如上的不足[1-6]。其中,ADL (Adaptive Directional Lifting) 通过率失真等手段,先确定变换块的尺寸和形状,然后再在块内按照纹理的方向特征对图像块做分解[1],取得了非常不错的编码效果。基于ADL变换的图像编码步骤包括预处理,自适应方向提升小波变换,量化,比特平面编码,码流信息打包等。由于ADL在进行方向判断和方向提升变换时用到分数像素,从而需要对图像进行插值运算。另外一方面,ADL的算法的计算复杂度非常高。ADL变换的计算量占据整个图像编码的95.3%。因此,充分利用ADL变换的内在的并行特点将有效地缩短图像编码的时间。

过去的几年中,GPU的性能和功能已经有了显著的发展。如今GPU不仅是一个强大的图形处理引擎,也是高度并行可编程处理器,以峰值运算和大大超过同类CPU的内存带宽为特征。由于GPU的处理能力在GPGPU (General-Purpose Graphic Processing Units)

收稿日期:2010-05-26;**修回日期**:2010-08-22
基金项目:部委基金"基于服务定制的智能存储系统研究"(编号略);国家自然科学基金项目(60803112,60873226)
作者简介:陈加忠(1970-),男,博士,副教授,CCF会员,研究方向为图像与视频处理、GPU计算与体系结构;夏 涛,博士,讲师,研究方向为流媒体技术与嵌入式计算。

领域已成功地得到了应用,因此,除了传统的计算机图形学问题,越来越多的计算密集型的任务正从 CPU 迁移到 GPU,包括解微分方程、视频编码、立体视觉、医学图像处理等[7,8]。因此,文中将根据 ADL 的算法特性,研究其在 GPU 上并行的实现方法。

1 ADL 算法流程与并行性分析

在 ADL 编码中,定义了 4 ×4 为基本的变换块,每个块在水平和垂直方向上各有 3 至 9 个分解的方向。ADL 算法根据率失真准则自适应地决定提升的方向,以利用图像内容在纹理方向上的相关性。在算法的实现中,首先读入图像块数据,然后对读入的图像块数据进行方向提升行分解,紧接着对行分解得到的数据进行方向提升列分解得到最终的分解系数矩阵。不管是行分解还是列分解,分解过程都包括 3 个步骤:分裂,预测和更新,并且行分解和列分解都有 9 个变换方向,其中有 3 个方向对应图像块的整数像素点,其余 6 个方向对应图像块的分数像素点,这些分像素点通过插值得到。ADL 需从 9 个变换方向中选择最优的方向进行提升变换。变换方向选取的依据是率失真准则 $J = D + \lambda R$,采用高频子带中的预测值之和作为率失真优化中失真的度量 D。

在 ADL 图像编码中,分数像素点方向上的变换需要先对块数据进行边界扩展,然后进行插值操作得到分数像素点的值。不同位置的分数像素点使用不同的系数进行线性插值,共有三组系数可供选择。对一个原始图像块数据进行插值将得到一个由分数像素点构成的块数据,而每个分数像素点的插值操作与其他分数像素点插值操作相互独立,但使用相同的插值系数,因而可在 CUDA 架构的 GPU 上指定一个线程完成一个分数像素点的操作,从而并行实现 ADL 中的插值操作。

2 ADL 变换在 GPU 上的实现

2.1 GPU 架构与 CUDA 并行编程模型

如图 1 所示,与传统的 CPU 的顺序编程模型对比,GPU 是大规模并行的高性能处理器,通常抽象为数据的各种形式的平行流模型。这样的流处理模型强调连续地存取接口和内存区域,以及数据并行的效率。为了实现 GPU 的强大性能,设计一个特定应用的流内核是至关重要的,它可以并行处理输入数据,产生一个或多个输出序列,相反,程序中不相关的控制流或者沉重的迭代必然会影响 GPU 运行时的性能。最近,随着下一代 GPU 编程模型已经出现,图形处理器编程 GPGPU 为应用程序开发提供了更广泛的区域。在这些模型中,NVIDIA CUDA 编程模型是比较出名的一个,尽管它只能用来针对 NVIDIA 图形处理器[9]。CUDA 技术是基于流处理模式,但是它也支持线程同步和共享的片上存储器。对于 CUDA 技术平台上的高性能计算,基础的设计和优化原则仍然适用,例如,最大限度地提高数据并行和优化内存访问模式[10]。

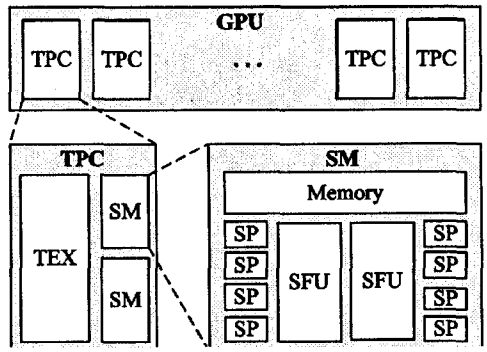

图 1 GPU 硬件架构图

CUDA 并行编程模型强调两点关键设计目标。第一,它要是一种标准的编程语言的延伸,具体来说就是 C/C++,用最低限度的抽象来描述并行。概括来说就是让程序员能集中精力关注关于并行等重要的问题,比如怎样设计有效的并行算法,而不是浪费时间在不熟悉复杂的语言上。第二,它是为编写高度可扩展的代码,与那些运行在上千个并发进程和上百个处理器上程序而设计的。这是因为当前 NVIDIA 的 GPU 在物理上的并行的范围是从 8 个处理器和 768 个线程到 240 个处理器和 30720 个线程,CUDA 模型自然就引导程序员编写清晰、高效、不同层次的并行程序[11]。

如图 2 所示,CUDA 程序被组织为 Host 程序,由一个或多个在主机 CPU 上运行的简单的进程组成,一个或多个适合在像 GPU 那样的并行处理器执行的并行内核。内核在一系列并行线程上执行标量 Sequential 程序。程序员将这些线程组织为 Device 线程块网格。单个线程块的线程允许相互同步,通过访问一个高速,块共享的片上存储器来相互通信,同一个网格中的不同线程块中的线程只能通过所有线程可见的全局存储空间来协调。CUDA 要求线程块相互独立,就意味着不管那个块在运行,内核必须正确的执行,即使所有的块在非抢占式的情况下按任意顺序执行。线程块间的依赖会限制为代码提供可扩展性,对全局通信的需求以及线程块间的同步是将并行任务分配到各个内核主要考虑的问题。

由于 ADL 在 9 个方向上的变换过程是相互独立的,因此,可在 CUDA 架构的 GPU 上分配 9 个 SM (Streaming Multiprocessors),每个 SM 具有 4 种类型的片内存储器:(1)每个 SP 都有一组本地 32 位寄存器;(2)共享内存(Shared Memory),同一 Block 内的线程共享同一片 Shared Memory;(3)只读高速缓存(Con-

stant Cache);(4)只读纹理高速缓存(Texture Cache),用于加速纹理内存的读取。实验中让每个 SM 实现一个方向上的变换,从而并行实现 ADL。由于 9 个方向上的操作是对相同的数据块进行,可以通过重复分配数据,以避免 9 个方向上变换并行实现时的访存冲突。

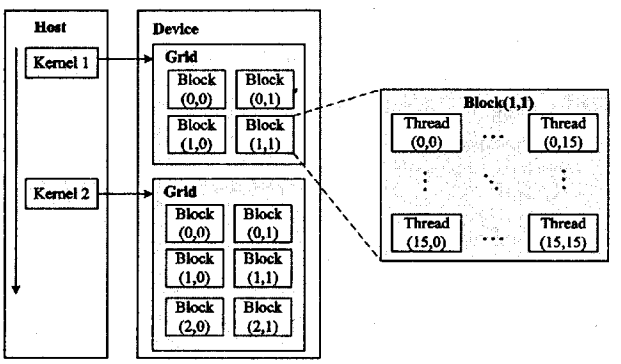

图 2　CUDA 程序执行框架

2.2　ADL 变换中插值操作的并行实现

这部分工作主要是垂直与水平方向上小波提升中预测前的插值。首先利用 CUDA 函数将当前节点的图像块数据以及相邻节点的变换方向从 Host 端拷贝到 Device 端,并将当前节点的块数据绑定到纹理存储器中,同时将三组插值系数存入常量存储器中。然后对原始块进行扩展,且保证得到的分数像素点块(假设大小为 row×col)数据的列数能被 16 整除。在 Device 端分配一个 row×col 大小的数组来保存插值得到的分数像素点块数据。当行数小于 16 时,将使用(col/16)个线程块进行插值操作,其中每个线程块的线程数目为(row×16);当行数大于等于 16 时,将使用(row/16)×(col/16)个线程块进行插值操作,其中每个线程块的线程数目为(16×16)。这部分可以看成是粗粒度并行。经过以上任务划分后,每个线程完成一个分数像素点的插值操作,即实现细粒度并行。根据分数像素点位置的不同,将选用常量存储器中的一组系数与纹理存储器中的数据进行线性插值,得到分数像素点的值。通过线程同步函数,保证整个插值过程结束后,再将插值得到的分数像素点块数据用于 ADL 变换。

2.3　ADL 变换的并行实现

这部分工作主要包含垂直与水平方向上的 ADL 变换和提升过程中预测后的插值。在经过 2.2 中的插值操作后,整数像素点和分数像素点的块数据都已保存在 Device 端,接下来就可在 Device 端实现 ADL 变换。ADL 先在垂直方向对当前块进行变换,然后对变换得到的块在水平方向进行变换,得到变换系数矩阵。在垂直和水平方向各有 9 个变换方向,所以可用矢量 (x,y) 来表示 ADL 选取的变换方向。在调用计算 ADL 变换方向代价的内核函数时,使用 9×9 个线程构成的线程块,每一个线程的标号(threadIdx.x,threadIdx.y)对应着一个变换方向,每一个线程计算一个变换方向的代价。在使用 ADL 进行变换时,选取的方向不同,进行的操作也不同,如选取的分数像素点方向,则需要进行 sinc 插值操作;而选取的整数像素点方向则不需进行插值操作,因此在并行实现 ADL 变换时,将使用不同线程块中的线程来执行各个方向的变换操作。在垂直方向使用 ADL 对块数据进行变换时,使用 9×1 个线程块构成的网格,网格中的每一个线程块只含一个线程,每一个线程实现一个方向上的变换,变换结束后将得到 9 个临时的系数矩阵,实现粗粒度并行。因为提升操作前后像素点具有依赖性,所以无法实现细粒度并行。

接着使用 2.2 中的并行插值方法对这 9 个矩阵进行插值。插值完成后,就可在水平方向使用 ADL 内核函数对以上 9 个临时系数矩阵和插值数据进行变换,即对每个垂直变换后得到的矩阵沿水平方向进行变换,每个垂直方向上各有 9 个变换方向,所以最终会得到 81 个变换后的矩阵。变换时使用 9×9 个线程块构成的网格,网格中的每个线程块将计算得到一个变换系数矩阵,同时计算其中的线程块标号表示的变换方向 (x,y) 的率失真代价。ADL 变换内核函数运行结束后将得到 81 个率失真值,然后在一个内核函数中求得最小率失真代价和最佳变换方向。最后将最小率失真代价,最佳变换方向,以及使用最佳变换方向进行 ADL 变换得到的变换系数矩阵从 Device 端拷贝到 Host 端。ADL 变换后把数据传回 CPU 作进一步的 SPIHT 编码[12]。

3　实验数据及分析

文中所使用的实验环境为双核、主频为 1.60GHz 的 Intel Celeron (R) CPU (1G 内存),NVIDIA 9800GT 显卡 (1024MB 显存,112 个流处理器,流处理器频率 1.5GHz),CUDA Toolkit version 3.0 for Windows XP。.cu 文件使用的编译选项为 -ccbin " $(VCInstallDir)bin" -I" $(NVSDKCUDA_ROOT)\common\inc" -I" $(CUDA_INC_PATH)" -I. -g -c -m32 -o" $(InputName).obj" " $(InputDir)\$(InputName).cu"。

表 1　在 CPU 和 GPU 上运行插值操作的实验结果比较

图像名	Barbara 512×512	Lena 512×512	Baboon 512×512
CPU 上平均运行时间(毫秒)	18004	17967	17757
GPU 上平均运行时间(毫秒)	2040	2037	2038

由表 1 可知,使用 CUDA 架构的 GPU 进行插值操作的速度是在 CPU 上的 9 倍左右。表 1 统计的是 ADL 变换中进行预测前的插值操作的总执行时间。

表 2 为小波提升预测后插值和 ADL 变换两个部分的时间。使用 CUDA 架构的 GPU 实现 ADL 变换的速度是在 CPU 上的 2.5 倍左右。这部分加速效果不明显，主要原因有：ADL 变换内核函数中分支判断语句比较多；ADL 变换内核执行过程中，每个 SM 的活动 Warp 数目较少，因而不能最大程度地隐藏访问延迟；ADL 变换内核只利用 CUDA 架构两层线程模型中的粗粒度即提升方向上并行，由于提升操作中前后数据存在相关性，未能利用细粒度并行以充分发挥 CUDA 架构 GPU 的计算能力。

表 2 在 CPU 和 GPU 上运行 ADL 变换的实验结果比较

图像名	Barbara 512×512	Lena 512×512	Baboon 512×512
CPU 上平均运行时间(毫秒)	29115	29079	29721
GPU 上平均运行时间(毫秒)	12264	12266	12268

在实验中，GPU 和 CPU 之间的数据总的传输时间为 300 毫秒左右，由于大部分能被计算隐藏起来，所以没有计入总时间。GPU 的内存访问模型中的本地内存和全局内存为设备内存的读写区域，且无高速缓存，因此还可以进一步通过 coalesce 的方式(即将多个连续内存访问操作合并成一次完成)来加速全局内存的操作。

4 结束语

文中提出了一种 ADL 变换的并行实现方法。在不损失编码质量前提下，对于存在大量动态分支运算、不规则数据存储以及并行规模较小的 ADL 算法，取得了较好的加速性能。今后的工作中，将利用零树编码中树与树之间的独立性，进一步把 SPIHT 移植到 GPU 上并行实现整个编码过程。

参考文献：

[1] Ding Wenpeng, Wu Feng, Wu Xiaolin, et al. Adaptive Directional Lifting-based Wavelet Transform for Image Coding [J]. IEEE Trans. Image Processing, 2007, 16(2): 416-428.

[2] Wang Demin, Zhang Liang, Vincent A. Curved Wavelet Transform for Image Coding[J]. IEEE Trans. Image Processing, 2006, 15(8): 2413-2421.

[3] Do M N, Vetterli M. The Contourlet Transform: An Efficient Directional Multiresolution Image Representation[J]. IEEE Trans. on Image Processing, 2005, 14(12): 2091-2106.

[4] Chang C, Girod B. Direction-Adaptive Discrete Wavelet Transform for Image Compression[J]. IEEE Trans. Image Processing, 2007, 16(5): 1289-1302.

[5] Liu Yn, Ngan K N. Weighted Adaptive Lifting-Based Wavelet Transform[J]. IEEE Transactions on Image Processing, 2007, 16(7): 1741-1754.

[6] 张 楠，吕 岩，吴 枫，等. 基于方向提升小波变换的多描述图像编码[J]. 自动化学报，2007，33(5)：567-576.

[7] 刘 双，申闫春，狄翠萍. GPU 在实时阴影绘制中的应用[J]. 计算机技术与发展，2009，19(11)：226-229.

[8] Chiang Chen-Kuo, Wang Shu fan, Chen Yi ling, et al. Fast JND-Based Video Carving with GPU Acceleration for Real-Time Video Retargeting[J]. Proceedings of the IEEE, 2009, 19(11): 1588-1597.

[9] NVIDIA Corporation. NVIDIA CUDA Programming Guide, Version 1.1[S], 2007.

[10] Lu Jiangbo, Rogmans S, Lafruit G, et al. Stream-Centric Stereo Matching and View Synthesis: A High-Speed Approach on GPUs[J]. IEEE Transactions on Circuits and Systems for Video Technology, 2009, 19(11): 1598-1611.

[11] Garland M, Grand S L, Nickolls J, et al. Parallel Computing Experiences with CUDA[J]. IEEE Micro, 2008, 28(4): 13-27.

[12] Pearlman W, Islam A, Nagaraj N. Efficient, Low-Complexity Image Coding With a Set-Partitioning Embedded Block Coder [J]. IEEE Trans. Circuits and Systems for Video Technology, 2004, 14(11): 1219-1234.