基于 CUDA 的快速中值滤波算法

吕亚飞 ， 贾堃阳

（四川大学计算机学院，成都 610207）

摘 要：在众多的快速中值滤波算法中选取一种适合在 CUDA 平台上实现的算法，并针对 GPU 的运算特点，对算法进行很大的改进。改进后的算法采用纹理存储器存储数据源，共享存储器和寄存器存储中间运算结果，并通过同一 block 内的线程共享排序结果减少了排序过程中所需的比较次数，降低了算法的复杂度。实验结果表明改进后的快速中值滤波算法充分发挥了 GPU 强大的并行处理能力，对于分辨率为 4096×4096 的图像其运算速度是基于 CPU 实现的 6597 倍，可有效地应用在实时图像处理中。

关键词：中值滤波；实时算法；GPU；CUDA

0 引 言

目前的主流 GPU 拥有数十至上百个处理核心，拥有 CPU 无法相比的并行运算能力。而 CUDA（Compute Unified Device Architecture，统一计算设备）是 NVIDIA 公司提出的一种将 GPU 作为数据并行设备的软硬件体系[1]。在图像处理的应用中，实时性要求越来越高，如何将 GPU 所提供的强大并行计算能力运用到实时图像处理中，已成为当前的研究热点之一。

中值滤波是图像处理中的一个常用步骤。作为一种非线性滤波，中值滤波既可以有效地消除随机噪声和脉冲干扰，又可以在很大程度上保留图像的边缘信息[2]。然而实现中值滤波需要较大的计算量，在实时图像处理中往往采用专门的硬件实现，例如现场可编程门阵列(Field Programmable Gate Array, FPGA)，需要较高的开发成本。本文结合 CUDA 编程模型的特点，对付昱强[3]等人提出的快速中值滤波算法进行了优化，介绍了在 CUDA 平台上实现高效中值滤波算法的方法。

1 快速中值滤波算法

中值滤波的思想是使用一个有奇数个点的滑动模板，对模板内点的灰度值进行排序，然后选择灰度值序列的中间值作为模板中心位置像素的值。噪声点在排序后会位于灰度值序列的最左端或最右端，不会出现在输出结果中。中值滤波的数学公式表示为：

$$g(i,j)=\mathrm{Med}\{f(i-k,j-h)\},(k,h)\in W \quad (1)$$

其中 $g(i,j)$为输出图像的灰度值，$f(i,j)$为输入图像的灰度值，W 为模板窗口。本文采用的是 3×3 的模板窗口，如图 1 所示。

f (i-1, j-1)	f (i, j-1)	f (i-1, j+1)
f (i, j-1)	f (i, j)	f(i, j+1)
f (i+1, j-1)	f (i-1, j)	f(i+1, j+1)

图 1 3×3 模板窗口

对于图 1 中的模板，中值滤波算法需要对 9 个元素进行排序，可以采用的排序算法有快速排序、冒泡排序和归并排序，快速排序和冒泡排序在最坏情况下需要 36 次比较，2-路归并算法需要 25 次比较。但是中值滤波并不需要对整个序列进行排序，只需找到元素的中间值即可。

收稿日期：2011-05-16 修稿日期：2011-06-16

作者简介：吕亚飞（1991-），男，安徽六安人，本科，研究方向为并行算法设计、图像处理

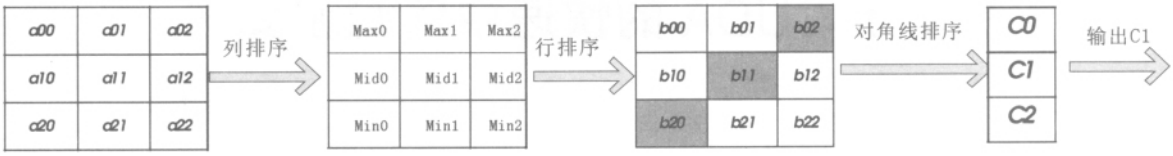

图 2 快速中值滤波过程

针对上述问题,本文采用的是文献[3]中提出的快速中值滤波算法。该算法首先对每列元素进行排序,然后对每行元素进行排序,最后对对角线上的元素进行排序,取中间值作为输出,如图 2 所示。3 个元素排序需要 3 次比较,所以列元素排序需 9 次比较,行元素排序也需要 9 次比较,对角线元素排序需 3 次比较,9 个元素取中间值共需 21 次比较。

2 快速中值滤波算法的优化及在 CUDA 上的实现

CUDA 程序的执行包含两部分,一部分是在 CPU 上串行执行的主机(Host)代码,另一部分是在 GPU 上并行执行的设备(Device)代码。运行在 GPU 上的 CUDA 并行计算函数称为 Kernel 函数(内核函数),而运行在 CPU 上的串行代码主要完成 Kernel 启动之前的数据准备和设备初始化工作,以及在 Kernel 之间进行的一些串行运算[4]。CUDA 程序一般包括以下过程:

(1)在 CPU 上初始化数据;

(2)将数据传输给 GPU;

(3)调用 Kernel 进行并行计算;

(4)将结果数据传回给 CPU。

2.1 设备端内存分配

CPU 和 GPU 各自拥有相互独立的储存器地址空间:主机端的内存和设备端的显存。在 Kernel 函数启动之前,需要为 GPU 分配显存并将 CPU 数据拷贝至 GPU,而拷贝至 GPU 上的数据可被 Kernel 函数中的所有线程访问。在 CUDA 存储器模型中能被所有线程访问的存储器有全局存储器(Global Memory)、常数存储器(Constant Memory)和纹理存储器(Texture Memory)。

在快速中值滤波算法中有大量的数据需要重复读取,虽然可在同一个 Block 内使用共享存储器存储相邻点数据,但在不同的 Block 之间依旧需要对数据进行重复读取。全局存储器具有较高的访存时延,对其访问一次大约需 400~600 个时钟周期,而且不具有缓存加速功能。而纹理存储器具有缓存机制,当所要访问的数据已经位于纹理缓存中时,就可以避免对显存的再次读取,访问纹理缓存的时间与访问 GPU 寄存器的速度相当。同时访问纹理存储器不用严格遵守访问全局存储器时所遵守的合并访问条件,提高了编程的灵活性。虽然常数存储器也具有缓存加速,但它只有 64KB 的存储空间,无法存储较大的图像数据。

使用纹理存储器时,首先要用 cudaBindTexture() 函数将线性存储器或 CUDA 数组绑定到纹理,然后便可以使用 CUDA API 中的纹理拾取函数对图形数据进行访问。

2.2 任务划分

Kernel 以线程网格(Grid)的形式组织,每个线程网格包含若干个线程块(Block),而每个线程块又有若干个线程(Thread)组成。不同 Block 之间可以并行执行,但无法进行通信;同一 Block 内的线程不仅能够并行执行,而且能够通过共享存储器和 _synchreads() 函数相互通信。在 Kernel 函数启动之前需要确定 Block 和 Grid 的维度。

在基于 CUDA 的快速滤波算法中,位于同一个 block 内的线程可以通过共享存储器共享数据,从而减少从显存中读入数据的次数。虽然较大的 block 可以有更多的线程进行通信,但较大的 block 会消耗更多的寄存器和共享存储器,从而降低每个 SM(Streaming Multiprocessor)上活动线程块(Active Block)的个数。在本文的程序中,每个线程使用的寄存器数量为 15,而共享存储器的使用数量为 Block 内线程数量的 3 倍,显卡的计算能力为 2.1,通过 CUDA SDK 中的 CUDA Occupancy Calculator 工具可以计算出不同 Block 维度下 GPU 计算单元的占有率曲线,如图 3 所示,Block Size 取 128 是其中的一个最优解。一旦确定了 Block 的维度后,Grid 的维度可用下列公式计算。

$$\begin{cases} GRID\_X=\dfrac{WIDTH+BLOCK\_X-2}{BLOCK\_X-2} \\ GRID\_Y=\dfrac{HEIGHT+BLOCK\_Y-2}{BLOCK\_Y} \end{cases}$$

GRID_X、GRID_Y 为 Grid 在 x 轴和 y 轴上的数量，WIDTH、HEIGHT 为图片的宽度和高度，BLOCK_X、BLOCK_Y 为 Block 在 x 轴和 y 轴上的数量。需要说明的是在本文的程序中，位于同一 Block 内 threadIdx.x 等于 0 或者等于 BLOCK_X-1 的线程只从显存中读入数据而不计算输出点，因而 GRID_X 的计算是除以 BLOCK_X-2。

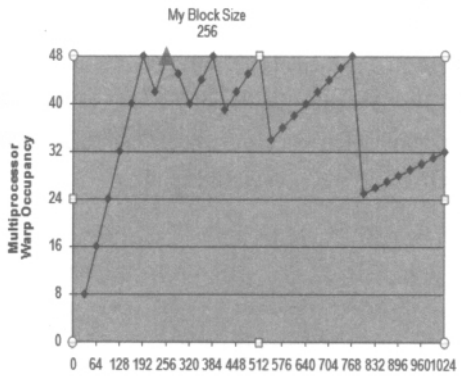

图 3 计算单元占有率

2.3 内核函数的设计

在本文的程序中，一个图像输出点对应 Grid 中的一个线程，如果不针对 GPU 对快速中值滤波算法做任何优化的话，每个线程需要从显存中读取 9 个数据并进行 21 次比较，才能得到计算结果。但是如果将所要访问的数据一次性存入共享存储器中，并在存入之前使用寄存器对数据进行列排序则可大量减少从纹理存储器中读入数据的次数和比较的次数。假设一个线程对应的输出点为 f(i, j)，则该线程将 f(i-1, j)、f(i, j)、f(i+1, j)读入寄存器中，然后对其排序，将排序后的数据存入到共享存储器中的对应位置，如图 4 所示。

在图 4 中，大括号注释标注的是对应数据存放的存储器类型，方括号注释标注的是对数据的操作。经过上述步骤后，位于共享存储器内的数据在每列上都是有序的，后续操作只需每个线程从共享存储其中读取 3 列数据进行行排序，然后取对角线元素的中值即可。优化后每个线程从纹理存储器中读取的数据个数减少为 3 个，进行比较的次数也降到 15 次，程序的执行效率得到了很大的提高。为了避免 Bank Conflict 可将 Block 的维度设计为 32 或 16 的倍数，共享存储器的数量设置为 Block 大小的 3 倍，然后每个线程按照从左到右的顺序读取三列数据。由于边界原因，位于同一 Block 内 threadIdx.x 等于 0 或者等于 BLOCK_X-1 的线程只执行图 4 中的操作，而不产生像素点的输出，这在一定程度上会造成同一 Warp 内的线程产生分支，不同分支中的指令会被串行地执行[5]。在上述原因产生的两个分支中，其中一条分支包含的指令数为 0，可以忽略其对程序性能产生的影响。

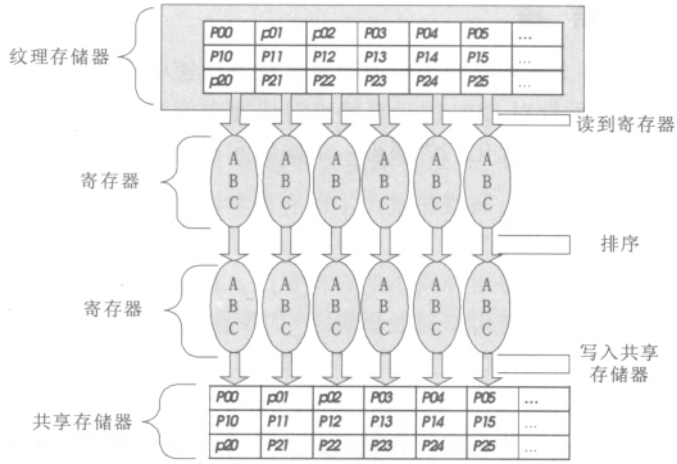

图 4 对快速中值滤波算法的优化

3 实验结果与分析

实验平台：软件环境为 Windows 7、Visual Studio 2008 和 CUDA 4.0；硬件环境为 Intel Pentium E5300 2.6GHz，GeForce GTS 450。实验数据为不同分辨率的灰度图像，表 1 列出了 GPU 和 CPU 在不同图像规模下实现快速中值滤波算法的时间。

表 1 GPU 与 CPU 实现快速中值滤波算法的运行时间对比

图像分辨率	GPU 处理时间/ms	CPU 处理时间/ms	加速比
128×128	0.03	45.53	1651
256×256	0.06	173.58	3141
512×512	0.13	713.69	5347
1024×1024	0.46	2718.51	5973
2048×2048	1.65	10626.95	6435
4096×4096	6.43	42422.78	6597

分析表 1 数据可知在图像分辨率低于 1024×1024 时,随着数据规模的增大,GPU 的加速比迅速提升。而当图像的分辨率高于 1024×1024 时,GPU 获得的加速比稳定在 6500 倍左右。这是由于数据规模较小时,每个多处理器上分配的 Block 数量不够多,没能充分利用 GPU 的计算资源;而当数据规模增加到一定程度时,每个多处理器上分配的 Block 数量也达到了饱和,GPU 的计算资源也得到了充分的利用,这时如果继续增加数据量,GPU 计算时间的增长速度便会与数据量的增长速度相同。在利用 GPU 进行运算时,大规模的数据能够获得更好的加速比。

4 结 语

本文首先介绍了中值滤波算法及快速中值滤波算法,然后针对 GPU 的运算特点对快速中值滤波算法进行了进一步优化,将获得一个输出元素所需的比较次数从 21 次降低为 15 次,算法的性能得到了很大的提升。实验结果表明该算法在 GPU 上获得了高达数千倍的加速比,具有实际意义。

参考文献

[1]张舒,褚艳利. GPU 高性能运算之 CUDA[M]. 北京:清华大学出版社,1997

[2]王宇新,贺圆圆. 基于 FPGA 的快速中值滤波算法[J]. 计算机应用研究,2009,26(1):224~226

[3]付昱强. 基于 FPGA 的图像处理算法的研究与硬件设计[D]. 南昌: 南昌大学, 2006

[4]CUDA_C_Programming_Guide.pdf[EB/OL]. [2011 -04]. http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/ CUDA_C_Programming_Guide.pdf

[5]CUDA_C_Best_Practice_Guide.pdf [EB/OL]. [2011 -04].http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/ CUDA_C_Best_Practice_Guide.pdf

Fast Median Filtering Algorithm Based on CUDA

LV Ya-fei , JIA Kun-yang

(Department of Computer Science and Technology, Sichuan University, Sichuan 610207)

Abstract: To realize median filtering algorithm efficiently on CUDA platform, uses a special fast median filtering algorithm. A big improvement has been made on the selected algorithm according to the features of GPU computing. The improved algorithm uses texture memory to store data sources, shared memory and registers to store operations result. And by the method that threads in the same block can share sorted results, the number needed to compare in the computing process is rapidly reduced. Experimental result shows that the improved fast median filtering algorithm makes full use of the parallel processing ability of GPU, with a speed which is at most 6597 times faster than the algorithm based on CPU, proves that it can be effectively applied in real-time image processing.

Keywords: Median Filter; Real Time Algorithm; GPU; CUDA