基于改进的压入与重标记算法的图割在GPU上的实现

李 晔 于双元 罗四维
(北京交通大学计算机与信息技术学院 北京 100044)

摘 要 Graph Cuts一直是应用于图像处理领域的一种重要方法。近些年特别在CUDA出现后,图像处理器逐渐成为能够编程的高层次多核心并行处理器。在GPU高性能计算平台上并行实现基于压入与重标记算法的Graph Cuts能够提高算法的运算性能,对于扩大Graph Cuts在图像处理领域的应用范围很有研究价值。首先将压入与重标记算法在GPU上进行并行化,通过CUDA的纹理内存技术来优化和改进并行化地压入与重标记算法的Graph Cuts。最后经实验证实,改进使算法性能得到有效提高。

关键词 图割,压入与重标记算法,CUDA,图形处理器
中图法分类号 TP312 **文献标识码** A

Realization of Graph Guts Based on Improved Push-relabel Algorithm on GPU

LI Ye YU Shuang-yuan LUO Si-wei
(School of Computer and Information Technology, Beijing Jiaotong University, Beijing 100044, China)

Abstract Graph Cuts has been an important method of computer vision, which can apply to the field of image processing. In recent years, the graphics processing unit (GPU) has progressively and rapidly become a kind of higher parallel computing processing unit with launching the Compute Unified Device Architecture (CUDA) from Nvidia. The parallel implementing of push-relabel algorithm on high-end parallel computing platform has a high level of parallelism to improve the computational efficiency. This technology has an important value of theoretical study and widely actual application background, which expands the range of applications of graph cuts in the field of computer vision. It firstly presents a basic parallel implementation of the push-relabel algorithm for graph cuts on the GPU, which introduces texture memory of GPU to improve the speedup. At last, the experiments test the times of foreground-background segmentation and compare speedup of different groups, which show optimizations can enhance the performance of the implementation.

Keywords Graph cuts, Push-relabel algorithm, CUDA, GPU

1 引言

随着图像技术发展以及人类对于图像处理需求的提高,可获得的图像分辨率越来越高,处理图像的实时性要求也越来越高。传统处理图像的方法,计算量大、效率低、实时性差,影响工作效率。如何提高图像处理效率成为图像领域的一个重点。近年来图像处理的并行化研究成为解决图像处理实时性差的一种有效的方法。有别于过去图像处理对串行算法进行不断优化,并行化的实施从根本上大幅度地提升了对图像处理的效率。

电子科技领域突飞猛进的发展使得图像处理器不仅限于单一的图形处理功能而且逐步向高性能处理和并行计算方向发展。图像处理器具有巨大的运算能力并且能够有效地处理多数据的并行任务。随着NVIDIA公司推出了CUDA体系,GPU发展成为一种不仅仅通过图像API库进行编程的图像处理器。CUDA体系中,GPU被看作一组通用的共享内存单指令多数据多核心处理器。目前在设备上并行的线程数量大约有几千个。许多图像处理算法在CPU上实现的结果效率相对较差,通过GPU的并行化可以得到相对更好的结果。

基于图论的图像处理技术是近年来国际上图像应用领域的一个热点。图论的并行研究一直是一个重要的研究方向,尤其在CUDA出现后,Graph Cut中的经典算法逐步在GPU上实现并且得到了相对更好的结果。Y Shiloach和U Vishkin在文献[1]中第一次提出了要将graph cuts进行并行化,文中采取基于增广路径的方法进行实验。A Goldberg第一个将push-relabel算法进行并行化,文献[2]中的实验通过相互通信的计算机实现并行化的算法。R J Anderson和J Setubal在文献[3]中更进一步地提出将并行化push-relablel算法在共享存储器体系中实现并且提出并发的全局性重标记技术,最后在实验中得到效率更好的结果。在文献[4]中,M Hussein等人在非锁定内存中的GPU上实现并行化的push-relabel算法,并且介绍了两种优化的方法,使得在实验中提高

到稿日期:2013-05-11 返修日期:2013-07-02 本文受国家自然科学基金(61272354)资助。

李 晔(1988—),男,硕士生,主要研究方向为分布式计算,E-mail:11120457@bjtu.edu.cn(通信作者);于双元(1965—),女,副教授,硕士生导师,主要研究方向为高性能计算、编译技术、中间件技术、软件测试;罗四维(1943—),男,教授,博士生导师,主要研究方向为人工神经网络、计算机并行处理、多媒体计算机技术。

了算法的运算速度。在文献[5]中，Vibhav Vineet 和 P J Narayanan 提出利用 CUDA 快速实现基于 push-relabel 算法的图割，文中作者分别优化了全局内存和共享内存，而且最后提出了有效提升运算效率的随机割理论。

2 CUDA 体系和压入与重标记算法概述

2.1 CUDA 的简介

统一计算架构(CUDA)是一种轻量级的单指令多数据(SIMD)并行计算架构。CUDA 是 GPU 编程语言模型，是在标准 C 语言的基础上进行了一些设备相关指令的扩展，使得开发人员可以更加直接地进行运用。

在 CUDA 编程体系中，通常将 CPU 作为主机而 GPU 作为设备。一个体系中可以同时拥有一个主机和很多个设备。CPU 和 GPU 协同工作完成程序的调度运行。在 CUDA 中，每一个线程拥有自己的私有存储器寄存器和局部存储器；每一个线程块有一块共享存储器；最后 grid 中所有的线程都可以访问同一块全局存储器[6]。除此之外，还有两种可以被所有线程访问的只读存储器：常数存储器和纹理存储器。

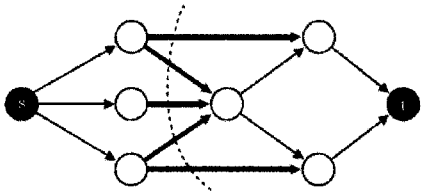
运行在 GPU 上的 CUDA 并行计算函数称为内核(kernel)，kernel 以线程网格(Grid)的形式组织，每个线程网格则由若干线程块(Block)组成，而每个线程块又由若干线程组成。线程是执行单元的基本单位。kernel 是以 block 为单位执行的，CUDA 引入 grid 来表示一系列可以被并行执行 block 的集合。在实际运行中，block 会被分割成更小的线程束(warp)，线程束由硬件计算能力决定[6]。SM 代表流多处理器，即计算核心。一个 block 必须被分配到一个 SM 中，但是一个 SM 中同一时刻可以有多个活动线程块。

在同一个 block 中的线程，可以进行数据通信。在 CUDA 中实现 block 内通信的方法是在同一个 block 中的线程通过共享存储器交换数据，并通过栅栏同步保证线程间能够正确地共享数据。

2.2 压入与重标记算法

2.2.1 图论的相关理论

拥有 N 个节点的有向无环图可以将节点看作图像的像素点。通过设定两个特殊的顶点 s 和 t 将节点划分到特殊顶点对应的分割类中。比如 $G(V,E)$，其中 V 是一组顶点，E 为一组边。s、t 是 V 中的两个终端。在 G 中的割 s/t 将图划分为分离的子集 S 和 T。图 1 中为图割的 s/t 的例子。



终端点 s、t 标记为黑色，割集的值为图中粗线边的权值的集合，S 集合和 T 集合通过割集连接。

图 1 graph 中，割 s/t 的例子

在图论的基础理论中，割集是图中一些被移除后可以使图不连接的边的集合，最小割集则是在割集中边权值最小的割集。在文献[9]中，Ford 和 Fulkerson 证明了网络流中的最小割最大流原理，即最小割问题可以转换为最大流问题。最大流问题是通过将流量值分配到网络中的每个管道来实现从源头到目标最大的流量。解决最大流的方法主要有两种，一种是增广路径的方法，另一种就是压入与重标记算法(push-relabel 算法)。相对于增广路径方法需要不停地在剩余图中搜索增广路径，push-relabel 算法更多的是局部的操作并且更多关注剩余图中顶点的邻居节点，更容易实现算法的并行化。在网络流中压入与重标记算法采用的是一种局部化的方法，即通过检查整个残留网络，每次仅对一个顶点进行操作并且每次仅检查残留网络中该顶点的相邻顶点。

2.2.2 关于压入与重标记算法

push-relabel 算法的中心思想是改变已经分配给边的流量，并试图沿着最短路径上的边压入节点上的多余流到汇点。如果余流不能压入汇点，剩余的余流将返回源节点。完成上述过程后，剩余流就是源点到目标点的最大流。但是执行这样的压入操作，首先需要引入图标记的概念，这个概念可以确定图中的最短路径。给图中每个节点分配标记数值，这个标记数值代表从该节点到汇点的不饱和最短路径距离的最低估计。

前置流是满足反对称性、容量限制的流守恒特性并且对所有顶点 $f(V,u)\geqslant 0$ 的函数。进入除源点顶点之外的顶点的总净流为非负值并且称进入顶点的总净流为进入 u 的余流即 $e(u)=f(V,u)$。如果 $e(u)>0$，则称顶点 u 溢出。设定 $G=(V,E)$ 是流网络，其源点为 s，汇点为 t。设 f 是 G 的一个前置流。如果函数 $h:V\to N$ 满足 $h(s)=|V|$，$h(t)=0$，且对每条残留边 $(u,v)\in Ef$，有 $h(u)\leqslant h(v)+1$，则该函数为高度函数(*height*)。顶点的高度是与其到汇点 t 的距离相关的，可以通过对转置进行广度优先搜索找到[7]。

对于 push-relabel 算法，在介绍其运算前必须符合一些要求：

$$f(u,v)\leqslant c(u,v) \tag{1}$$

$$f(u,v)=-f(u,v) \tag{2}$$

$$\sum_{u\in V} f(u,v)=0,\ \forall v\in\{V-\{s,t\}\} \tag{3}$$

在算法操作中需要判断节点，所以提出活动节点的概念。所谓活动节点就是余流为正数的并且节点的标记不等于 n 的节点。而压入重标记算法在节点上包含两种操作：push 和 relabel。这些操作只对活动节点进行。

push 操作：如果节点 u 是活动的，节点 u 的邻接边容量为正值($c_f(u,v)>0$)并且到邻接点的高度函数顺差为 1($h[u]=h[v]+1$)，其中高度函数的条件是确保将流压入到路径中估计距离汇点最近的点。设 $d_f(u,v)$ 存储从 u 压入到 v 流量的临时变量，它决定了边和余流的大小，则：

$$d_f(u,v)=\min(e[u],c_f(u,v)) \tag{4}$$

如果流的余量从活动节点压入到它的相邻顶点，则：

$$f[u,v]=f[u,v]+df(u,v) \tag{5}$$

$$f[v,u]=-f[u,v] \tag{6}$$

$$e[u]=e[u]-df[u,v] \tag{7}$$

$$e[v]=e[v]+df[u,v] \tag{8}$$

relabel 操作：重标记操作的作用是增加节点的高度函数值。如果节点 u 是活动节点并且在剩余图中节点 u 具有非零边，节点 u 的高度函数值不大于邻接点的高度函数。则：

$$h[u]=1+\min\{h[v]:(u,v)\in\bar{G}\} \tag{9}$$

压入与重标记算法是通过重复执行这两个操作来处理图中节点得到最终的结果。压入比重标记算法串行伪代码如图 2 所示。

```
    GENERIC-PUSH-RELABEL(G)
Init:源节点 s 无穷大,其他节点=0
While (exist push or relabel operation){
    if(e(u)>0,cf(u,v)>0,h[u]=h[v]+1){
        df(u,v)=min(e[u],cf(u,v));
        f[u,v]=f[u,v]+df(u,v);
        f[v,u]=-f[u,v];
        e[u]=e[u]-df[u,v];
        e[v]=e[v]+df[u,v];
    }
    else if(u 在剩余图中具有非零边,h[u]≤h[v]){
        h[u]=1+min{h[v]:(u,v)是残留网络中的边;}
    }
}
```

图 2　压入比重标记算法串行伪代码

搜索所有节点,如果没有活动节点算法终止,网络中存在的是最大流。把在剩余图中无法达到汇点的节点归入到 S 并把剩余图中能够到达汇点的节点归入到$\bar{S}$中,得到区分图。

由于经常不能准确地估计到汇点的距离,很多情况下 push-relabel 算法的实现效率并不是很高;而且因为重新标记操作在同一时间内只能改变单个节点的标记,这样就需要花费大量的时间来确定正确的估计。因此文献[3]提出全局重标记方法来缩短运算时间并且得到适当的距离估计。基本方法是简单地以汇点作为起始做广度优先搜索,并对从汇点开始的在广度优先搜索中得到的节点设置标签。如果每隔一段时间在剩余图中做 BFS,算法的性能将有进一步提高。

3　基于压入与重标记的图割 GPU 上的实现

3.1　GPU 上的压入与重标记算法

为了能够得到并行压入重标记算法优化的实现,首先需要了解 GPU 计算平台尤其是关于计算单元的结构。GPU 在出现 CUDA 平台后,已经不仅仅是处理渲染图像的处理器,更作为并行处理器来处理大数据。最值得注意的是它处理数据的方式是按照并行进行的,因此 GPU 上的大量的运算单元可以对压入重标记算法进行并行计算。当进行图像处理时,需要有规则来分配每个像素到每个节点进行处理。由上面提到的 push-relabel 算法原理可知源点和汇点被设定为不活动状态不需要处理,而将剩余的节点对应到完整的网格,图的节点对应所有的像素点。然而 graph 结构的数值变化必须存储在 GPU 上,将 graph 结构传输到 GPU 上相对比较复杂。

3.2　压入与重标记算法 CUDA 上的并行化

CUDA 环境中算法实现的优越性是运用 GPU 的 SIMD 体系结构来展现的。在实验中应该尽量避免线程的发散并且在数据单元上执行相同的操作,这样可以获得更高的运算性能。GPU 是一种高运算能力和低存储带宽的运算器,为了得到更高的实验性能,应该减少通信。根据以上所述,通常认为对所有顶点在 GPU 上并行操作:如果满足 push 操作的条件,就并行所有节点;如果满足 relabel 操作的条件,就并行操作这些节点。但是,在 CUDA 上应该让所有运算器同时执行相同操作来避免发散;而且对于任意节点不能同时进行将流推向其他节点和收到从邻居节点推来的流,因为这些操作同时更新余流。所以下面将讨论这些节点操作。

并行 label 操作:由以上描述可知,任意节点的 label 与其到目标节点的高度有关,但是基本的 relabel 操作降低了算法的运算速度。文献[4]提出了一种称为 global relabeling 的方法,其将节点用它们到目标节点的实际距离作为标记,通过使用从最终节点开始的宽度优先遍历。运用这种方法,所有节点能够得到最佳的并行 label。

并行 push 操作:任意节点不能同时收到来自其超过一个邻接点的流。为了解决这个问题,可以将 push 操作分解为两个阶段:push 和 pull。在 push 阶段,任意节点不能更新其邻接点的流,而是每个节点设置从该节点输出连接的缓存库存储正在从节点输出的流。在 pull 阶段,存在节点能够收到来自其邻接点的流,这些节点更新其余流和剩余图中的边容量。

对于任意节点 u 上存在的数据包括:节点的余流 $e(u)$、高度 $h(u)$、节点活动标记 $flag(u)$和剩余图中到其邻居节点的边容量。为了能够使算法并行,将节点分成了 3 种状态,分别为:*Active* 态、*Passive* 态、*Inactive* 态。

$$Active(u):\{e(u)>0;\exists v\{h(u)=h(v)+1\}\} \tag{10}$$

$$Passive(u):\{re(v)\wedge\exists v\{h(u)=h(v)+1\}\} \tag{11}$$

$$Inactive(u):\{e(u)=0\vee(u,v)\notin\bar{G}\} \tag{12}$$

其中,*active* 节点满足有余流 $e(u)>0$、至少有一个邻接点存在 $h(u)=h(v)+1$。passive 节点是存在不满足 $h(u)=h(v)+1$ 但是通过 relabel 操作可以满足条件的节点。*inactive* 节点是 $e(u)=0$ 或者在剩余图中没有邻接点的节点。

在 GPU 上实现并行的压入与重标记算法需要两个 *kernel*:*PushPullKernel* 和 *RelabelKernel*。

PushPullKernel 更新边权值(u,v)和(v,u)、余流 $e(u)$和顶点的 $e(v)$以及剩余图的节点。PushPllkerml 的伪代码如图 3 所示。

```
PushPullKernel(u){
    h(u):global to shared;
    synchronize threads from threads;
    if(exist e(v)>0){
        push operation;
    }
    c(u, v)= c(u, v);c(v, u)= c(v, u);
    e(u) =e(u); e(v)= e(v);
    }
```

图 3　PushPullKernel 的伪代码

具体操作是将 $h(u)$从全局内存中装载到块中的共享内存中;将同步线程确保装载完整;在不违反前流条件的前提下一致性地将流推到合适的邻接点中;一致性地更新剩余图中(u,v)和(v,u)边的权重;一致性地更新剩余图中余流中的 $e(u)$和 $e(v)$;原子性地写入全局内存中保证 grid 中的 block 一致性。

RelabelKernel 作用是得到来自全局内存中的邻接点的 *height*,然后将新的 *height* 值写入全局内存中。RelabelKernel 的伪代码如图 4 所示。

```
RelabelKernel (u){
    h(u):global to shared;
    synchronize threads from relabelkernel;
    min{h[v]:(u, v)属于剩余图}
    h(u)=h(u);
    h(u):shared to global;
}Active(u)=Passive(u)
```

图 4　RelabelKernel 的伪代码

具体操作是将 $h(u)$从全局内存中装载到块中的共享内

存中；同步线程从而能够装载完整；在剩余图中计算 u 的邻接点中的最小 *height*；将更新的 *heights* 写入全局内存中。在 Relabel 操作后，*passive* 节点状态变为 *active*。

并行化的压入与重标记算法通过调用 CUDA 的内核实现对图中节点进行分割，最终形成两个节点集合。GPU 压入与重标记算法并行化伪代码如图 5 所示。

```
Init:图像映射到 graph:e(u),h(u),flag(u),c(u,v)
Parallel graphcuts()
{
    RelabelKernel();
    PushPullKernel();
}
```

图 5　GPU 压入与重标记算法并行化伪代码

4　基于压入与重标记算法 GPU 实现的改进

4.1　对于 GPU 带宽的研究

GPU 存储器的带宽是计算性能的瓶颈，通常图像处理器的计算能力远远超过内存访问的带宽，以存储器访问为主的实现中，应该尽可能高效地利用内存带宽，加速数据存取。过去对于并行化地压入与重标记算法研究并不关注 GPU 存储器带宽，只在文献[4]中提出一种模拟缓存技术，这种技术类似于存储器体系中虚拟内存技术，在访问全局内存时增加结合访问从而减少对于全局内存的访问次数来降低 GPU 存储器的带宽。但是，全局内存并不具有缓存技术，存取时间相对过长，模拟缓存技术只是对于数据结构的改进。事实上在并行化压入与重标记算法中，数据具有空间局部性并且在调用相同内核时更新的数据不需要重新读入，这些特点适合 GPU 的一种特殊的纹理存储器。

4.2　利用 GPU 的纹理存储器

纹理内存是专门为内存访问模式中存在大量空间局部性的算法而设计的，牵涉到显存、两级纹理缓存、纹理拾取单元的纹理流水线[6]。纹理流水线与 GPU 的计算核心相对独立，它的缓存节省带宽和功耗。与标准的全局内存相比，纹理内存存在一些限制，但在某些情况下，使用纹理内存将提升应用程序的性能。

如图 6 所示，一个线程读取的位置可能与邻近线程读取的位置很接近。但是从数学角度看，图中的 4 个地址并非连续，一般的存储器不具备这种访问模式。本文中处理的图像数据按照网格的方式二维存储类似于图 6 所示，并行化压入与重标记算法中顶点数据与局部邻接点关联性巨大。

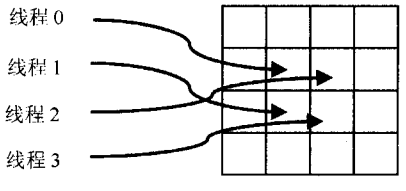


图 6　GPU 中线程映射到二维内存区域

当 GPU 调用内核时，根据纹理内存的特点，纹理缓存预取两个维度上连续分布的数据，纹理流水线计算加载数据的地址，将二维空间映射到一维，过滤对存储器控制器的一部分访问请求以节约带宽。

调用 *PushPullKernel*()：

1. 节点和局部邻接点的标记值、边权值、余流和顶点值从纹理内存进入缓存。

2. 将纹理缓存的标记值传到线程块内的共享内存中。

3. 判断当前节点是否符合条件，符合则其线程调用缓存中剩余数据处理节点。由于前面操作修改了纹理内存中的数据，因此需要重新启动新的 kernel 函数处理其他节点。

调用 *RelabelKernel*()：

1. 节点及其邻接点的高度值缓存到纹理，然后装入共享内存。

2. 如果节点符合条件，则节点的状态 *passive* 变为 *active*，并且更新缓存中的高度值。

纹理存储器是只读的，因此没有数据一致性。如果更改纹理显存中的数据，纹理缓存的数据没有更新，则通过纹理拾取得到错误数据。纹理缓存需要及时刷新，则与 GPU 的内核以及 GPU 的硬件特性相关。

4.3　GPU 任务配置

GPU 硬件特性对于提高算法的性能有重要的影响。在 CUDA 的执行模型中，grid 中的各个 block 会被分配到 GPU 的各个 SM 中执行。由于存储器带宽影响，只有纹理内存的缓存大小与处理缓存中数据的线程充分匹配，才能有效利用资源。所以设置主要参数包括：每个 grid 中 block 的数量 B、每个 block 中线程的数量 T 和纹理缓存的大小 K。

设定算法线程的处理数据量为 I，则每个线程块所需要的内存为 $S_T=T*I$。根据图像处理器的硬件特性可知 GPU 的 warp 为 32，为了保证线程块内的线程形成完整的 warp，每个 block 中线程的数量 T 最好是 32 的倍数。GPU 拥有 30 个 SM，线程块需要平均分配到 SM 中，每个 grid 中线程块 B 的数量最好的配置为 30 的倍数。当 GPU 的 SM 为满载时，需要内存 $Ssm=30*T*I$，则纹理缓存的大小必须满足 $K\geqslant Ssm$ 并且选取 Ssm 整数倍最佳。考虑到 SM 中存在隐藏流处理器指令的延迟，纹理内存的只读性以及修改数据时需要更新整个纹理显存，则纹理缓存设置为 $K=Ssm$ 时效果最佳。

5　实验结果和分析

本文提出的方法是通过测试区分前景背景的实验时间体现的，实现的评判方法是在文献[8]中提出的，它基本上是将图像划分为前景和背景两个分区。

采用压入与重标记算法实现图像前景和背景的区分实际是构建分割区域将图像域分成两类。因此利用组合优化的概念，将图中的每个节点对应于图像的像素，同时将提取对象对应汇点背景对应源点。图像中边的权值一般由两部分构成，平滑项和数据项，这两部分的比重用一个系数来平衡，其中平滑项主要体现顶点像素和其相邻区域像素间值的变化强度，如果变化剧烈说明这两者很有可能出于边缘部分，则被分割开的可能性比较大，而按照最小割的分割原理，这时两者的平滑项权值应该较小。而数据项部分则表示对应顶点属于前景或者背景的惩罚项。

$$E(A)=\lambda\cdot R(A)+B(A) \tag{13}$$

其中

$$R(A)=\sum_{p\in P}R_p(A_p)$$

$$B(A)=\sum_{\{p,q\}\in N}B_{\{p,q\}}\cdot\delta(A_p,A_q)$$

并且

$$\delta(A_p,A_q)=\begin{cases}1, & \text{if } A_p\neq A_q\\ 0, & \text{otherwise}\end{cases}$$

式中，$R(A)$ 表示的是区域数据项，$B(A)$ 表示的是边界平滑项，$E(A)$ 表示的是权值，即损失函数，也叫能量函数，图割的

目标就是优化能量函数使其值达到最小。

本文的实验中没有设定复杂的约束条件，比如增加一些属于前景或背景的像素。实验是在对于前景和背景区分结果没有明显差异的前提下，比较不同的实现时间。

CPU 实现的硬件平台为 Intel Core i5 2.8GHz 处理器、8G 内存；GPU 实现的硬件平台为 NVIDIA GTX570、4G 设备内存。实验实现的软件平台是 Ubuntu12.04 LTS 、CUDA toolkit5.0。实验采用压入与重标记算法进行图像区分测试运算时间。在 3 幅不同的图像中分别进行 CPU 实现、GPU 并行实现以及改进的 GPU 实现。

实验在图 7 中展现算法处理不同图像得到的区分结果。表 1 记录了对于处理不同图像时得到的测试时间，其中分别列出 CPU 与改进 GPU 算法实现和普通 GPU 算法实现的加速比。表中的运算时间排除了产生图形本身时间之外的时间，对于 CUDA 实现图形数据从主机传送到设备的时间和从设备将结果传送到主机的时间也被排除在外。

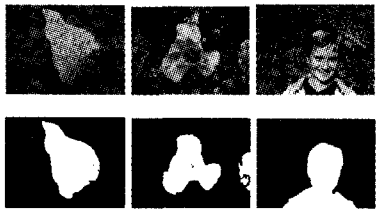


图 7　前景背景分割：牵牛花，海绵，女士

表 1　CPU 实现结果分别与在 GPU 下一般实现和优化实现下的对比

图像	尺寸	算法分别在 CPU 和 GPU 实现的时间比较			算法分别在 CPU 和改进 GPU 的时间比较			性能提升
		CPU (ms)	GPU (ms)	加速比	CPU (ms)	Improved GPU(ms)	加速比	
海绵	640*480	11.02	4.50	2.45	11.02	3.65	3.01	22.9%
牵牛花	600*450	18.59	7.22	2.58	18.59	5.05	3.68	42.6%
女士	600*450	31.89	8.94	3.58	31.89	6.98	4.57	27.8%

由于使用 GPU 计算平台，并行化的算法在同一时间可以处理大量节点的数据，相比采用 CPU 串行算法性能明显提高，其中加速比=CPU 实现时间/GPU 实现时间。对于普通的 GPU 实现，由于采用了全局存储器，其并没有缓存机制。GPU 上数据传输量巨大，储存器的带宽成为程序的瓶颈。优化的 GPU 实现采用了纹理内存的缓存技术并且结合了 GPU 的硬件特性，因为缓存的数据重复利用减少了存储器和计算单元的传输次数，很大程度上影响了整个算法的性能。表 1 中，性能提升=(GPU 改进实现的加速比－GPU 普通实现的加速比)/GPU 普通实现的加速比。

从表 1 中的数据得出，在图像处理结果相同时，相比单独采用 CPU 而言，利用 GPU 进行并行化压入与重标记算法能够将图像处理性能提高 3～4 倍；相比利用 GPU 普通的实现而言，优化的 GPU 并行压入与重标记算法性能提升了 23%～43%。实验结果表明，对于 GPU 下压入与重标记算法的改进有效，提高了图像处理的速度。

结束语　本文基于近年来热门的 CUDA 并行计算平台，以图论的压入与重标记算法为基础，关注 GPU 带宽对于算法的影响，从而提出运用纹理缓存改进算法的实现，提高了 Graph Cuts 中基础的图论算法在图像处理的性能。

图像处理的实时性要求在逐年提升，对于 GPU 存储器的数据管理成为继续提高并行算法速度的关键。未来将继续研究如何提高 GPU 计算核心和 GPU 存储器之间的通讯带宽问题，从而设定管理 GPU 内存的数据存储机制，将其广泛运用到图像处理领域，使在 GPU 上已经并行化的方法的效率继续提升。

参 考 文 献

[1] Shiloach Y, Vishkin U. An $o(n^{\log n})$ parallel max-flow algorithm [J]. Journal of Algorithms, 1982, 3: 128-146

[2] Goldberg A. Efficient graph algorithms for sequential and parallel computers[D]. Cambridge: MIT, 1987

[3] Anderson R J, Setubal J. On the parallel implementation of goldbergs maximum flow algorithm[C]// Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures. 1992: 168-177

[4] Hussein M, Varshney A, Davis L. On implementing graph cuts on cuda[C]// First Workshop on General Purpose Processing on Graphics Processing Units. 2007

[5] Vineet V, Narayanan P J. CudaCuts: Fast Graph Cuts on the GPU[C] // CVPR Workshop on Visual Computer Vision on GPUs. 2008

[6] 张舒，褚艳利. GPU 高性能运算之 CUDA[M]. 北京：中国水利水电出版社，2009：15-20

[7] Cormen T H. Introduction to algorithms(第二版)[M]. 潘金贵，译. 北京：机械工业出版社，2009：411-415

[8] Boykov Y, Jolly M P. Interactive graph cuts for optimal boundary & region segmentation of objects in n-d images[C]// International Conference on Computer Vision (ICCV). 2001

[9] Wu Zhen-yu, Leahy R. An Optimal Graph Theoretic Approach to Data Clustering: Theory and Its Application to Image Segmentation [J]. IEEE Trans. on Pattern Analysis and Machine Intelligence, 1993, 15(11): 1101-1113

[10] Shi Jian-bo, Malik J. Normalized Cuts and Image Segmentation [J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2000, 22(8): 888-905

[11] 卢凤顺，宋君强，银福康，等. CPU/GPU 协同并行计算研究综述 [J]. 计算机科学，2011，38(3)：5-9

[12] Grady L, Schwartz E L. Isoperimetric Graph Partitioning for Image Segmentation[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2006, 28(3): 469-475

[13] 杨帆，廖庆敏. 基于图论的图像分割算法的分析与研究[J]. 视频技术应用与工程，2006(7)：80-83

[14] 许新征，丁世飞，史忠植，等. 图像分割的新理论和新方法[J]. 电子学报，2010(2A)：18-82

[15] 刘丙涛，田铮，李小斌，等. 基于图论 Gomory-Hu 算法的 SAR 图像多尺度分割[J]. 宇航学报，2008，29(3)：1002-1007

[16] Feng Wen-juan, Xiang Hui, Zhu Yan. An Improved Graph-based Image Segmentation Algorithm and Its GPU Acceleration [C]// Workshop on Digital Media and Digital Content Management. 2011

[17] Bader D A, Sachdeva V. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic [C] // ISCA PDCS. 2005: 41-48

[18] 张庆科，杨波，王琳，等. 基于 GPU 的现代并行优化算法[J]. 计算机科学，2012，39(4)：304-311

[19] 陶文兵，金海. 一种新的基于图谱理论的图像阈值分割方法[J]. 计算机学报，2007，30(1)：110-119

[20] Boykov Y, Veksler O, Zabih R. Fast approximate energyminimization via graph cuts [J]. IEEE Trans. Pattern Anal. Mach. Intell., 2001, 23(11): 1222-1239