GPU-Cuts: Combinatorial Optimisation, Graphic Processing Units and Adaptive Object Extraction

Nandan Dixit
Renaud Keriven
Nikos Paragios

Research Report 05-07
March 2005

Ecole Nationale Des Ponts et Chaussées
CERTIS
Centre d'Enseignement et de Recherche en Technologies de l'Information et Systèmes

CERTIS, ENPC,
77455 Marne la Vallee, France,
http://www.enpc.fr/certis/

GPU-Cuts: Combinatorial Optimisation, Graphic Processing Units and Adaptive Object Extraction

GPU-Cuts : Segmentation d'Objects par Optimisation Combinatoire sur Processeur Graphique

Nandan Dixit[1][2]
Renaud Keriven[1]
Nikos Paragios[1]

[1]CERTIS, ENPC, 77455 Marne la Vallee, France, http://www.enpc.fr/certis/
[2]Indian Institute of Technology Bombay, India, http://www.iitb.ac.in/

Abstract

Object extraction is a core component of computer vision with application to segmentation, tracking, etc. In this paper we propose a GPU graph-based approach to object segmentation. The main contributions of our approach consist of an adaptive, evolving schema to update to statistical properties of the object and the use of a local variant push-relabel algorithm to recover the global minimum of the designed cost function. Furthermore, we propose the implementation of the method on a graphics processing unit where each node of the graph is considered as an independent processor that is connected with the neighborhood nodes. Such a schema recovers segmentation in an optimal and progressive manner. Promising experimental results and important decrease on the computational complexity demonstrate the potentials of our approach.

The authors would like to acknowledge Prof. Boykov [6] from University of Western Ontario, Canada for pointing out the concept of considering graph-based optimization approaches on graphics processing units and provide references for the push-relabel algorithm for the parallel implementation of the mim cut/max flow algorithm.

Résumé

L’extraction d’objets est un sujet central dans la vision par ordinateur et a des applications dans la segmentation, le suivi (tracking), etc. Dans cet article, nous proposons une méthode de segmentation sur GPU basée sur les graphes. Les principaux éléments sont une mise à jour évolutive et adaptative des propriétés statistiques de l’objet, ainsi que l’utilisation de l’algorithme de variation locale des ”push-relabel” pour trouver le minimum global de la fonction de coût choisie. De plus, nous proposons l’implémentation de cette méthode sur un GPU tel que chaque noud du graphe est considéré comme un processeur indépendant connecté à ses nouds voisins. Une telle méthode donne une segmentation de façon optimale et progressive. Des résultats expérimentaux promettants et une importante baisse de la complexité algorithmique sont la preuve du potentiel de notre approche.

Contents

1 Introduction

Image segmentation has been heavily addressed in computer vision. Statistical methods [8], snake-driven approaches [17], variational and level set techniques [22], model-based methods [10] and graph-based techniques [7] are well received formulations to address the problem. Unconstrained segmentation is a problem with important complexity. In the most general case neither the number of classes nor their characteristics are known. On the other hand, object extraction refers to the separation of the object from the background and therefore is a tractable application. In particular model-based segmentation is of increasing interest in domains like medical image analysis.

Snake-driven techniques perform object extraction through the optimisation of a cost function, that consists of image and smoothness terms. Such an approach is quite popular to object extraction and has been exploited in multiple ways. Dynamic programming, Lagrangian formulations and level set methods [21] are some of the techniques considered within this formulation. To this end, an initial surface is propagated towards the object boundaries according to the image term while being constrained to be regular.

Convergence to local minimum is the most important argument against the use of such techniques. Recent advances in the area of combinatorial optimisation have made this domain of increasing interest in computer vision. The use of graph-based techniques to the extraction of minimal paths [4], segmentation and object extraction [7, 27], motion analysis and tracking [23] and stereo [18] are some examples. In [7] connection between minimum cost curs in graphs with MRF-based segmentation [13] is presented while in [3] an interactive approach to dual segmentation is proposed using the max/min flow. In [24] the normalised cuts criterion is introduced that is based on measuring the dissimilarity between the different classes and similarity within the classes. A different graph partitioning approach was proposed in [15] that attempts to find sets with a low isoperimetric ratio that is a more geometric constraint when compared to previous graph-based techniques.

The strongest argument to the use of combinatorial techniques is their ability to determine the solution that corresponds to the global minimum at least for the dual case. The Max-Flow/Min-Cut flow algorithm [12] is the most prominent technique. The push-relabel algorithm [14] is an alternative technique to address the same task with certain additional complexity. The solution is recovered through the propagation of maximum information (flow) from a node to its neighbourhood. Such a concept - the one considered in this paper - can make object extraction a rather dynamic process where the statistical properties of the different classes could be updated according to a dynamic fashion.

In the past decade, the computational power that is provided by the graphics card has exploded. In addition to this, the graphics processing unit(GPU) of the modern graphics cards is very easily programmable. In light of this, there has been increasing focus on performing general purpose computation on the GPU in numerous domains [1] including image processing, computer vision [19] and visualisation. In this paper we propose the implementation of a parallel version [2] of the push-relabel algorithm [14] to address image segmentation in a graphics processing unit through a max flow/min cut approach [11]. Once certain approximations are made, one can claim a significant decrease of computational complexity that allows object extraction in a rather adaptive manner. In other words one can tolerate several passes where the statistical properties of the different classes could be updated according to a dynamic fashion.

The reminder of this paper is organised as follows. In section 2 we introduce the push relabel algorithm, while in section 3 an approach to adaptive object extraction is proposed. The use of graphics processing units is considered in section 4 while some comparisons regarding the computational aspect of the approach and some experimental results are presented in section 5.

2 Push Relabel Algorithm

Let us consider a directed acyclic graph with N nodes that when can be seen as the image pixels where segmentation labels are to be attributed. We restrict our approach to the case of binary segmentation that consists separation of an object from the background. One can introduce that within the graph notation through two special vertices the source s and the sink t which correspond to the two segmentation classes.

Given such a graph, a cut-set is the set of edges which when removed make the graph disconnected while the min-cut is a cut-set for which the sum of the weights of edges is minimum. This idea was explored in [7] leading to a pioneering segmentation approach. The Push Relabel Algorithm [14]consists of modelling a directed acyclic graph as a flow network with a source and a sink and capacities on each edge.

Under such a definition, one can consider two main problems: (i) the min-cut problem for a network flow that is equivalent with finding a min-cut that disconnects the source from the sink, (ii) the max-flow problem refers to the assignment of flow values to each pipe in a network such that we achieve the maximal flow from the source to the sink. The max-flow problem is assignment of flow values to each pipe in a network such that we achieve the maximal flow from the source to the sink.

Let $G = (V, E)$ be the a graph with two special vertices the source s and the sink t. One can introduce for any pair of vertices u , v a capacity measure

$c(u, v)$ that is zero in the absence of arc between u to v. It was shown [12] that establishing the maximal flow is equivalent with finding the min-cut on the graph according to the following constraints on the flow between u,v

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

Thus, the max-flow problem is to maximise $\sum_{u \in V} f(u, t)$ where the maximum flow that an arc can carry is the weight(capacity) of the edge in the graph. One can further facilitate the optimisation process through the introduction of pre-flow constraints [16]

$$\sum_{u \in V} f(u, v) \geq 0, \forall\, v \in \{V - \{s\}\},$$

that are like flows, with one exception, that a vertex can have an excess of inflow over the outflow.

2.1 Overview

The central idea behind the push-relabel [14] algorithm is to change the flows that are assigned to arcs, and try to push the excess flow towards the sink along the arc that appears to be on the shortest path. Once such a condition (excess flow be pushed to the sink) cannot be fulfilled, a saturation is observed and the remaining excess has to be pushed back to the source. Upon completion of the process, the resulting flow assignment can then be used to determine the the max-flow from the source to sink [3].

To perform such a push operation, one should be able to determine the shortest path along the available outgoing arcs and therefore, the notion of graph labels are introduced. We assign a numeric label $\omega(u)$ at each graph node u thats refers to an under-estimate of the shortest non-saturated path length from that node to the sink.

During the course of the push-relabel, we will impose mostly the first two min-cut constraints

- $f(u, v) \leq c(u, v)$, $f(u, v) = -f(v, u)$

while allowing the flow conservation to be violated, as we maintain pre-flows. Last, but not least instead of keeping track of the flows in the arcs and the capacities, we only keep track of the residual capacities in all vertex pairs, which we define as $r(u, v) = c(u, v) - f(u, v)$. One can now introduce:

[3] The case of bin segmentation consists of attributing pixels to either class (out of two) and finding the max-flow between them.

Flow excess: The flow excess at a node is the excess of the inflow over the outflow at that node. That is, $excess(u) = \sum_{u \in V} f(u,v)$. The pre-flow constraint keeps the excess non-negative.

Active nodes: A node is said to be active if its excess is non-zero and the corresponding label is not n. The algorithm consists of two operations on nodes, Push and Relabel. These operations are only carried out on active nodes.

Push operation: A flow Push is applicable from u to v if, u is active, $r(u,v) > 0$ and $d(u) = d(v) + 1$ The last condition is to ensure that we push the flow only in paths that are estimated to be closest to the sink. When we carry out this push, of a flow value δ we decrease $r(u,v)$ by δ and increase $r(v,u)$ by the same amount. The value of δ is dictated by the pre-flow constraint and capacity constraint.

$$\begin{aligned} r(u,v) = r(u,v) - \delta \;\&\; r(v,u) = r(v,u) + \delta \\ excess(u) = excess(u) - \delta \;\&\; excess(v) = excess(v) + \delta \end{aligned}$$

Where, $\delta = min(r(u,v), excess(u))$ which comes from the pre-flow constraint and capacity constraint.

Relabel operation: The Relabel operation on a vertex u involves increasing $d(u)$. A relabel is applicable on u if u is active and for every vertex v which has a non-zero arc from u in the residual graph, $d(u) \leq d(v)$. The new value of $d(u)$ is set as

$$d(u) = 1 + min_{r(u,v)>0}(d(v))$$

that is an under-estimate of the distance to sink.

Once such a set of definitions and operations are available one can introduce the **push-relabel algorithm** as follows:

- *Init:* Set the label of the source to be infinity. Label of all other nodes is initialised as zero. In practice, the label of the source can just be a large enough value(N) that the nodes connected with the sink will never reach.

- *Init Pre-flow:* For all u push a flow δ from the source s to u, where $\delta = c(s,u)$

- *Push-Relabel Loop:* While there is an active node, pick an active node u. If there is a push applicable on an arc (u,v) then do the flow push, else relabel u. When there is no active node u left, exit.

Upon termination of the algorithm - $O(n^3)$ time - there will be no active nodes left, that is, the pre-flow has become a flow. This is also a maximum possible flow in the network [14]. To obtain the partition of the graph $(S, \bar{S})$. We put in S, all nodes which do not have a path to the sink in the residual graph, and all the nodes which can reach the sink in the residual graph form $\bar{S}$.

Although the algorithm is $O(n^3)$, a vanilla implementation of push-relabel turns out to be slow in most cases. This is in large part due to the incorrect estimates for the distance to the sink. And since our relabel operation changes the label of only one node at a time, it takes a large time for the correct estimates to kick in. Attempts like global relabel have been made to cut short on these times and have a proper distance estimate.

The basic idea of this is to simply do a breadth first search (bfs) starting from the sink as the root, and set the label of a node to be nothing but the distance from sink we get in the bfs. If we do this bfs in the residual graph at regular intervals, the performance of the algorithm improves significantly.

2.2 Implementation

The efficiency of the push-relabel method depend on the ordering of the push and relabel operations. In [14] and then in [9], the authors introduce three families of heuristics.

- The first one consists in pushing as many flow as possible from a vertex before eventually relabelling it. This is the *discharge* operation. A efficient implementation of this operation uses, for each vertex, a rotating list of its edges, so that no particular edge is considered more than the others.

- The second issue is the order in which active vertices are processed. Maintaining a FIFO queue of the active vertices seems to be less efficient than processing the vertex with the highest label.

- Because update operations are local, the methods loses the global picture of the distances to the sink. The *global relabelling* heuristics uses a backward breath-first search to restore the labelling to the distance to the sink in the residual graph. Performing it periodically improves the running time. One can also use a *gap relabelling* heuristic, consisting in finding the smallest positive value not used by any label. It such a value exists, all the vertices with a greater label can directly be labelled to N.

Based on these ideas, the authors of [9] present a very efficient ”H_PRF” implementation of the push-relabel method from IG [26].

2.3 Parallel Push-Relabel

In [14], the authors present a simple parallel push-relabel implementation, where each process holds one vertex. The discharge operation is just slightly modified, giving what we will refer to as the *parallel discharge* operation. First, all the vertices push flow at the same time but update their own excess and residual capacities only, ignoring the corresponding updates of their neighbours. In a second time, they all relabel themselves if necessary. As a third and last step, the vertices are informed about what their neighbours had pushed during the first step and eventually update their excess and and residual capacities.

In [2], a way to get some kind of parallel global relabelling using successive waves is presented. The resulting algorithm is easy to implement (please see the original paper for detail). We will refer to it as the *waves relabelling* heuristic. Both *parallel discharge* and *waves relabelling* are data parallel SIMD algorithms, thus good candidates for a GPU implementation.

3 Image Segmentation & Object Extraction

Once the general optimisation framework has been presented, one now can consider the case of object extraction. Such a problem consists of creating a partition of the image domain into two classes. Therefore, using the notions of combinatorial optimisation, one can consider a graph [7] where each node of the graph corresponds to an image pixel, while the object refers to the sink and the background to the source. Furthermore, one can introduce local dependencies and smoothness constraints through arcs going from a node to the nodes that correspond to the neighbouring pixels in the image. Once such an equivalence is introduced, one can see segmentation as the assignment of a labelling $\mathcal{L}(\Omega)$ at the image domain according to:

$$E(\mathcal{L}(\Omega)) = E_{data}(\mathcal{L}(\Omega)) + \alpha E_{smooth}(p(\mathcal{L}(\Omega))$$

where $E_{data}(p(\mathcal{I}|\mathcal{L}(\Omega))$ is a term that accounts for dependencies between labels and observations, while $E_{smooth}(p(\mathcal{L}(\Omega))$ introduces the notion of smoothness on the labelling process. Let us without loss of generality that some prior knowledge on the statistical properties of the background $p_B(I)$ and the object is known $p_O(I)$ that are the conditional densities with respect to the appearance of the two classes. Then one can introduce the image term as follows:

$$E_{data}(\mathcal{L}(\Omega)) = \sum_{\omega \in \mathcal{L}(\mathcal{O})} -\log\left[p_O(\mathcal{I}(\omega))\right] + \sum_{\omega \in \mathcal{L}(\mathcal{B})} -\log\left[p_B(\mathcal{I}(\omega))\right]$$

where $\mathcal{L}(\mathcal{O})$ (resp. $\mathcal{L}(\mathcal{B})$) are the image pixels assigned to the object (resp. background). Smoothness constraints aim at penalising discrepancies among labels at the local neighbourhood scale;

$$E_{smooth}(\mathcal{L}(\Omega)) = \sum_{\omega \in \Omega} \sum_{\phi \in \mathcal{N}(\Omega)} \mathcal{V}(\omega, \phi)$$

where $\mathcal{V}(\omega, \phi)$ could be defined in the following form:

$$\mathcal{V}(\omega, \phi) = \begin{cases} 0, & if\ \mathcal{L}(\omega) \neq \mathcal{L}(\phi) \\ \alpha, & if\ \mathcal{L}(\omega) \neq \mathcal{L}(\phi) \end{cases}$$

with α being a positive constant. Such a term will force segmentation to be locally smooth even in the cases where discontinuities on the data itself arise. Therefore it is natural to relax such a constraint when the data refers to discontinuities, or

$$\mathcal{V}(\omega, \phi) = \begin{cases} 0, & if\ \omega = \phi \\ \alpha|\mathcal{I}_{\mathcal{S}}(\omega) - \mathcal{I}_{\mathcal{S}}(\phi)|, & if\ \omega \neq \phi \end{cases}$$

where the input image $\mathcal{I}$ was convolved $\mathcal{I}_{\mathcal{S}}$ with a Gaussian operator to account for local discrepancies due to noise. Such a problem can be represented using a directed acyclic graph with N nodes, where its node is connected with the:

- sink according to: $-\log\left[p_O(\mathcal{I}(\omega))\right]$
- source according to: $-\log\left[p_B(\mathcal{I}(\omega))\right]$
- neighbouring nodes according to: $\alpha|\mathcal{I}_{\mathcal{S}}(\omega) - \mathcal{I}_{\mathcal{S}}(\phi)|$.

One can consider the push-relabel algorithm approach for the optimisation of such a cost function on a standard CPU. However, as it was shown in [5], the CPU non-parallel version of the push-relabel algorithm is far more inefficient than its rivals. On the other hand, the algorithm proposed in [5] cannot be considered in a parallel version and therefore it would be interesting to compare it with the GPU parallel version of the push-relabel algorithm.

4 Graphics Processing Units & Push Relabel

Towards optimal implementation of the push-relabel algorithm, it is important to understand the GPU computational platform and in particular the graphics pipeline. Scene rendering on a graphics card is a multi-step process. The specification of geometric shapes in terms of meshes (vertices forming convex hulls) is the first step. Such a mesh is then rotated according to viewing angle, providing a projection where textures are applied at a per-pixel level. Therefore one can introduce two classes of processing

- Operations that involve processing each vertex of the mesh: rotation, scaling, 2D projections, etc.

- Operations that have to be done on each pixel of the rendered scene: applying textures, illumination, per pixel processing, etc.

Consequently, most modern GPUs have two programmable processors: (i) the vertex processor that performs operation of the first class and (ii) the pixel processor, that refers to operations of the second class. In other words, the pixel processor - in normal operation - take as input the coordinates of pixels and the corresponding texture, and outputs the colours at each pixel in the final rendered image. It is important to note that processing for each pixel is done in a parallel fashion.

Therefore, one can conclude that the use of the pixel processor is adequate for doing parallel computations on a large number of nodes, that is the case of the push-relabel algorithm. Different type of operations are supported by the pixel shader:

- *Arithmetic Operations:* scalar and vectorial operations are supported with - in most of the cases - limited support for integers.

- *Logical Operations and Conditionals:* scalar and vector logical operations are supported as well as conditional assignments.

- *Texture operations:* The pixel shader have support to look up the texture values of a point using a vector that describes its texture coordinates , the most prominent component of the pixel processor.

Shader Model is a standard for vertex and pixel processor specifications. The newer shader model 3.0 is much more conducive for doing general purpose computation. In order to implement the push-relabel algorithm on the GPU, several aspects are to be addressed:

4.1 Graph nodes and pixels

The pixel processor is used to do parallel computation on the nodes in our Graph. To process nodes on the pixel processor we need to have a scheme of assigning a pixel to every node that is trivial in the case of the rectangular grid nature of the graph when the problem of image segmentation is considered. Since the source and the sink are never active and hence never need processing, the remaining nodes are a perfect grid, where each pixel corresponds to a graph node. However, passing the graph structure to the card is not as trivial as the node assignment since the structure along with all the modifications have to be stored on the card. Textures are the only available storage media, where the graph structure is to be mapped.

4.2 Texture representation of Graphs:

In order to store all the information about a residual graph, in addition to the nodes we also need the residual capacities on the arcs between the nodes and arcs to the sink and from the source. On top of that for push-relabel, we also need to store the excesses at every node and the labels. Such information has to be packed into textures from which the pixel shader can extract all the required values. To this end, we use textures with 4 attributes(rgba) at each coordinate.

The graphics card we used offered eight bits per attribute. Hence the values of the attribute are in the range of $0 - 255$. In our implementation of image segmentation, we only generate planar grid graphs which always have degree 4 [4]. Therefore, the capacities of all 4 outgoing edges (arcs) are packed into a single texture. On the other hand, the capacities of the incoming edges are available through the neighbouring texture locations. The excesses and the labels and stored in a difference texture. Since they can grow and be larger than the 0 to 255 range offered by the 8 bits, we use two attributes to store excess and two to store the label of the graph node.

Last, but not least the source and sink links arcs are stored in a texture with two attributes(16 bits) for each of the source and sink arc.

4.3 Push-Relabel on Pixel shader

One of the major constraints of the pixel shader is that it can render a limited number of targets at a time and often there is a limit on the instructions number on the pixel shader code[5]. The biggest constraint on the pixel shader was the small instruction limit. To counter this, the basic idea was to split the code for flow pushing into many parts, each pushing flow in a particular direction if permitted by the labels and the capacities.

On top of that while for the shader model 3.0 we can renter/modify 4 textures at a time, only one texture can be modified for our original implementation on the ATI Radeon 9800 Pro card which fully supports Shader Model 2.0(SM2). Therefore, it is important to introduce a procedure that effectively splits each operation of pushing or relabelling to a certain number of steps. Such a procedure will be of interest even for the implementation of the algorithm on the Shader Model 3.0 when a neighbourhood system with more than 4 (substantial number) connections is considered.

Without loss of generality we will assume that a single texture can be modified at a time that was the case for the Radeon 9800 Pro card model. Then, we propose to split a single operation of pushing or relabelling according to 3 steps on the

[4]The source and the sink links are introduced in a separate manner.

[5]Radeon 9800 Pro card (Shader Model 2.0(SM2)) can not go beyond 96 instructions.

pixel shader where such a step refers to the smallest unit of processing that can be invoked on the shader. In such a scenario, (i) the first step determines which operation is to be performed, while (ii) subsequent steps modify a texture based on the output of the first technique. One can alternate between these steps (modification of the texture representation) to accomplish push-relabel on the graph. The operation to be performed is dictated by the first step, and is either push or relabel. On top of that one should the push direction and either the push value or the new label.

Action Instruction: The Push Relabel involves two operation on active nodes: Push and Relabel. This technique determines which operation is applicable on the node. On top of that if the ”push” is applicable this action determines the δ while for the ”relabel” case it determine the new label. This happens on the pixel shader, hence is done for all the nodes in parallel. The first step involves looking up arc capacities and labels from the textures where texture lookup functions are used. Initial action consists of pushing the flow to the sink. If such an action is not possible, we try to push flow to any of the 4 neighbouring nodes and the source. If the push of flow is not possible, we relabel with the new label while for non active nodes no action takes place.

Subsequent Techniques: Once the action to be performed in *action instruction* has been determined, the subsequent techniques consists of bookkeeping, modifying the residual arcs, labels and excesses in agreement with the *action instruction*. It is worth mentioning that such techniques are to be synchronised with the operations performed at the neighbouring pixels of u. Pushes from neighbouring pixels change the excess and arc capacities of arcs going out of u as well. The above techniques are applied in a loop until no active nodes are left. For checking completion, we have to move the textures to the cpu, where we check the excesses of all nodes. When the excesses of all nodes become zero, we are done, the nodes with labels greater than N are on the source side (background) and the remaining ones are with the sink (foreground).

As it was earlier explained, one can perform **global relabel** to further accelerate the convergence to the optimal solution. Such a step can be considered either on the graphics card or at the CPU. Both methods have been implemented and an experimental comparison between the two methods was considered. We came to the conclusion that for well structured graphs related with image segmentation problems their difference is marginal and therefore for global relabelling takes place on the CPU. Based on the experimental results presented in [FIG. (1)] the speed up factor of the parallel version is marginal when compared to its CPU version. On the other hand using some theoretical notions from parallel and dis-

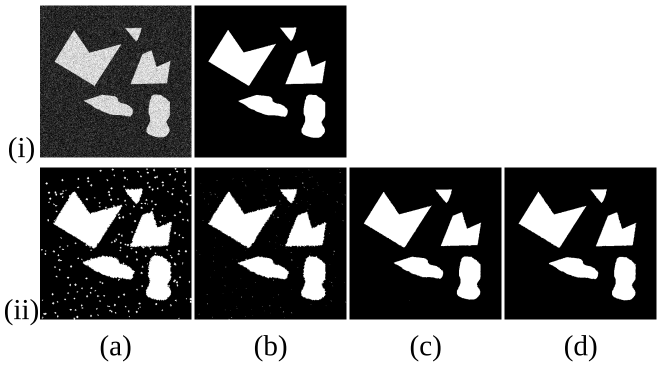

Figure 1: (i.a) input image, (ii.b) optimal segmentation, (ii) approximate segmentation using : (a) 5 discharges, 35 ms, (b) 10 discharges, 70ms, (c) 15 discharges, 105 ms, (d) 20 discharges, 140ms.

tributed programming and certain approximations one can overcome the need of global relabelling and can boost the GPU version of the algorithm.

4.4 Notions & Approximations from Parallel and Distributed Programming

One can claim that relaxing the constraint of having an efficient vertices ordering as well as efficient relabelling scheme, does not eliminate the ability of recovering the global minimum. In [25] the authors observe that the minimum cut is most of the time near to the source or near the sink.

In computer vision applications like segmentation, it seems to be so where most of the minimum cut turned out to be near the sink. This means that one practically does not have to wait for the vertices that will be with the source to get a high label. Starting from a 0 label, after a very few parallel discharges, all the vertices that have a label greater than a small constant, can be approximatively considered as sources vertices, and the other ones as sink vertices. Examples of such an approach are shown in [FIG. (2,3)] where results obtained through different numbers of discharges are presented.

4.5 Incremental Segmentation

Once such an implementation is available, one can consider the use of various image and smoothness terms to perform object extraction. The piece-wise constant Mumford-Shah approach [20] consists of segmenting and reconstruction the image using piece-wise linear functions.

$$E_{data}(\mathcal{L}(\Omega), \mu_{\mathcal{O}}, \mu_{\mathcal{B}}) = \sum_{\omega \in \mathcal{L}(\mathcal{O})} (\mathcal{I}(\omega)) - \mu_{\mathcal{O}})^2 + \sum_{\omega \in \mathcal{L}(\mathcal{B})} (\mathcal{I}(\omega)) - \mu_{\mathcal{B}})^2$$

Furthermore, one introduce the edge weights without taking into account data discontinuities [4] leading to a graph-based definition of the piece-wise constant Mumford-Shah approach [20], that was used for our experiments.

5 Conclusions

In this paper we have proposed a fully distributed discrete optimisation approach to image segmentation on graphic processing units. Such an approach is based on recent advances on combinatorial optimisation related with the max flow/min cut problem. In most of the cases [FIG. (2)] we can only claim a marginal decrease of the computational cost when comparing the approach with its CPU version.

Under certain approximations one can observe a substantial decrease of computational cost. On the other hand a more de-favourable comparison with the graph-cut algorithm [5] was observed when the optical solution was the objective. One can argue though that when approximations are considered such comparisons turns in favour for the GPU approach. Furthermore, given the expected evolution of the GPU in terms of speed, memory and operations (dynamic branching, etc.) the proposed approach inherit enormous potentials.

5.1 Experimental Evaluation

All the tests are done on a simple graph issued from a 2D segmentation problem using 4 neighbours only. We do perfectly know that on such graphs, the graph-cuts method is more efficient than the push-relabel one. Tests on more complex regular graphs are perfectly feasible using the multi-target rendering capacities of the recent GPUs. On such graphs, we expect the push-relabel to be a serious candidate.

We first used an ATI Radeon 9800 Pro GPU with no multi-target rendering and a limited number of instructions in each pass. With a 256x256 image, each parallel discharge operation takes 43ms. On an Nvidia GeForce 6800GT GPU, the same operation takes 15ms only. Yet, this GPU permits multi-target rendering and a less limited number of instructions version which takes 2.6ms only. In the same conditions, a 512x512 image needs 9.2ms for each parallel discharge and 29ms are needed for a 1024x1024 image. This gives us a discharge rate of more than 36 millions of single vertex discharges per second. On a Pentium4 running at

SIZE	PR	GPU	GPU+GR	GPU+*
256x256	170ms	910ms	950ms	10-50ms
512x512	870ms	9200ms	3200ms	35-140ms
1024x1024	1820ms	125000ms	95000ms	150-700ms

Figure 2: *Execution times for the results presented in [*FIG. *(1)] using the Nvidia6800Pro card; (PR) Push Relabel on the CPU [26], (GPU) Push Relabel on the GPU, (GPU+GR) Push Relabel on the GPU and Global Relabelling on the CPU, (GPU+*) approximation of the Push Relabel on the GPU with limited number of discharges.*

3.0Ghz, the sequential algorithm [26] gives a less than 0.6 millions of discharges per second.

Yet, our parallel algorithm lacks of any heuristics on the ordering of the updated vertices, and will obviously never have one, just because all the vertices are continuously updated. As a result, our implementation reaches the global minimum latter than the sequential H_PRF version, just because a lot more discharges are needed! On a 256x256 image, we need almost 350 parallel discharges and nearly 1 second, whereas the H_PRF terminates in no more than 180ms. With a 512x512 image, its even worse: 1000 discharges and nearly 9 seconds instead of 870ms for the sequential version.

Global relabelling is also a crucial heuristic in order to obtain the global minimum. We have tested the *wave relabelling* of [2]. Yet, this algorithm needs a nearly N parallel discharges to get the waves reach all the vertices. As a consequence, it is more adapted to complex graphs where the sequential algorithm would have passed through a lot of updates, anyway. The running times we get with waves relabelling are similar to the ones without it: it sometimes reduces the number of discharges, but handling the waves gives a nearly 25 percent slower discharge operation. Another solution is to perform global relabelling and gap relabelling of the CPU. This really decreases the number of discharges. Yet, it supposes to read the data from the GPU back to the CPU, which is still a relatively slow operation. On an 512x512 image, each global relabel takes 80ms, for a total running time of nearly 3 seconds, which is still more than 3 times slower than the CPU H_PRF version.

5.2 Future Directions

The implementation of the method to account for 3D data is a natural extension of our approach. Furthermore, it was shown that for more complicated neighbourhood systems [5] the push-relabel approach is favourable compared to the graph-cut approach and therefore is a natural future direction. Real-time incre-

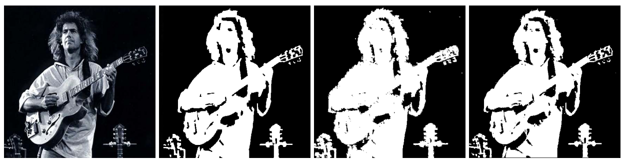

Figure 3: (a) input image, (b) optimal segmentation, (c) approximate segmentation using 5 discharges (35 ms), (d) 10 discharges (70ms).

mental stereo reconstruction graphics processing units was the purpose of our investigation. To this end, adaptation of the push-relabel algorithm to address reconstruction is the ultimate objective.

References

[1] General-Purpose Computation Using Graphics Hardware. *http://www.gpgpu.org/*.

[2] R. Anderson and J. Setubal. A Pralallel Implementation of the Push-Relable Algorithm for the Maximum Flow Problem. *Journal of Parallel and Distributed Computing*, 29:17–26, 1995.

[3] Y. Boykov and M-P. Jolly. Interactive Graph Cuts for Optimal Boundary and Region Segmentation of Objects in N-D images. In *IEEE International Conference in Computer Vision*, volume I, pages 105–112, 2001.

[4] Y. Boykov and V. Kolmogorov. Computing Geodesics and Minimal Surfaces via Graph Cuts. In *IEEE International Conference in Computer Vision*, volume I, pages 26–33, 2003.

[5] Y. Boykov and V. Kolmogorov. An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:1124–1137, 2004.

[6] Y. Boykov, P. Torr, and R. and Zabih. Discrete Optimization Methods in Computer Vision, 2004. Tutorial, European Conference in Computer Vision.

[7] Y. Boykov, O. Veksler, and R. Zabih. Fast Approximate Energy Minimization via Graph Cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23:1222–1239, 2001.

[8] Y. Cheng. Mean Shift, Mode Seeking, and Clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17:790–799, 1995.

[9] B. Cherkassky and A. Goldberg. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19:390–410, 1997.

[10] T. Cootes, C. Taylor, D. Cooper, and J. Graham. Active shape models - their training and application. *Computer Vision and Image Understanding*, 61:38–59, 1995.

[11] E. Dinic. Algorithm for Solution of a Problem of the Maximum Flow Problem. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

[12] L. Ford and D. Fulkerson. *Flows in Networds*. Princeton University Press, 1962.

[13] S. Geman and D. Geman. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

[14] A. Goldberg and R. Tarjan. A New Approach to the Maximum Flow Problem. *Journal of the Association for Computing Machinery*, 35:921–940, 1988.

[15] L. Grady and E. Schwartz. Faster graph-theoretic image processing via small-world and quadtree topologies. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2004.

[16] A. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.

[17] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active Contour Models. In *IEEE International Conference in Computer Vision*, pages 261–268, 1987.

[18] V. Kolmogorov and R. Zabih. Multi-camera Scene Reconstruction via Graph Cuts. In *European Conference on Computer Vision*, volume 3, pages 82–96, 2002.

[19] A. Lefohn, J. Cates, and R. Whitaker. Interactive, GPU-Based Level Sets for 3D Segmentation. In *Medical Imaging Copmuting and Computer-Assisted Intervention*, volume 1, pages 564–572, 2003.

[20] D. Mumford and J. Shah. Boundary detection by minimizing functionals. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 22–26, 1985.

[21] S. Osher and J. Sethian. Fronts propagating with curvature-dependent speed : Algorithms based on the Hamilton-Jacobi formulation. *Journal of Computational Physics*, 79:12–49, 1988.

[22] N. Paragios and R. Deriche. Geodesic Active Regions: A New Framework to Deal with Frame Partition Problems in Computer Vision. *Journal of Visual Communication and Image Representation*, 13:249–268, 2002.

[23] J. Shi and J. Malik. Motion Segmentation and Tracking Using Normalized Cuts. In *IEEE International Conference in Computer Vision*, pages 1154–1160, 1999.

[24] J. Shi and J. Malik. Normalized Cuts and Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22:888–905, 2000.

[25] J. Sibeyn. The Parallel Maxflow Problem is easy for almost all Graphs. *http://citeseer.ist.psu.edu/sibeyn97parallel.html*.

[26] IG SYSTEMS. Efficient implementation of a push-relabel algorithm for the maximum flow/minimum cut problems. *http://www.igsystems.com/hipr/*.

[27] N. Xu, R. Bansal, and N. Ahuja. Object Segmentation Using Graph Cuts Based Active Contours. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 46–53, 2004.