9

Accelerating Live Graph-Cut-Based Object Tracking Using CUDA

Ismael Daribo, Zachary A. Garrett, Yuki Takaya and Hideo Saito
Keio University
Japan

1. Introduction

Graph cuts have found many applications that address the problem of energy minimization, which occur frequently in computer vision and image processing. One of the most common applications is binary image segmentation, or silhouette extraction.
Image segmentation is the process of applying a labeling to each pixel in an image to determine a list of boundaries of objects, areas of interest, or silhouettes of objects in the scene. The resulting pixel labeling enables higher level vision systems to perform complex actions such as gesture recognition for human-computer interfaces Ueda et al. (2003), real-time 3D volume reconstructions Laurentini (1994), and mixed reality applications Takaya et al. (2009). Without accurate and coherent segmentations, these higher end-to-end systems will fail to perform their required functions. This necessitates a method which can correctly identify and extract objects from a scene with high accuracy. However, accuracy usually comes with serious performance penalties, particularly in image processing which examines hundreds of thousands of pixels in each image. This is unacceptable for high-level vision systems, which generally require the processing to be completed in real time.
By creating a graph using information in the image, as in Fig. 1, researchers have recast the problem of energy minimization and image segmentation as a graph theory problem. Then, methods such as flow analysis and finding shortest paths give researchers information about the structure of the underlying image. The results of these methods have been very promising, although the problem of computation complexity (speed) remains. Techniques such as *livewire* Mortensen & Barrett (1995), *normalized cuts* Shi & Malik (1997), and *graph cuts* have been developed to produce more accurate image segmentations and silhouette extractions.
Applying graph cuts to layered frames of a video sequence has shown to be robust for object tracking and object segmentation. In such scenarios, the entire video sequence is treated as a single 3D volume of layered frames, and a cut is performed across the entire sequence at once Boykov & Funka-Lea (2006). In sight of the large size of the graphs, which is increased with the video sequence length, performance of the cut is of paramount importance. To solve this problem, advances in the graph cut algorithm have produced dramatic performance improvements Boykov & Kolmogorov (2004); Juan & Boykov (2006).
However, regarding live video applications, there is no prior knowledge of subsequent frames. Techniques that build single graph across the entire video Boykov & Funka-Lea (2006) become then inapplicable in live video scenarios. Instead, a graph must be maintained or

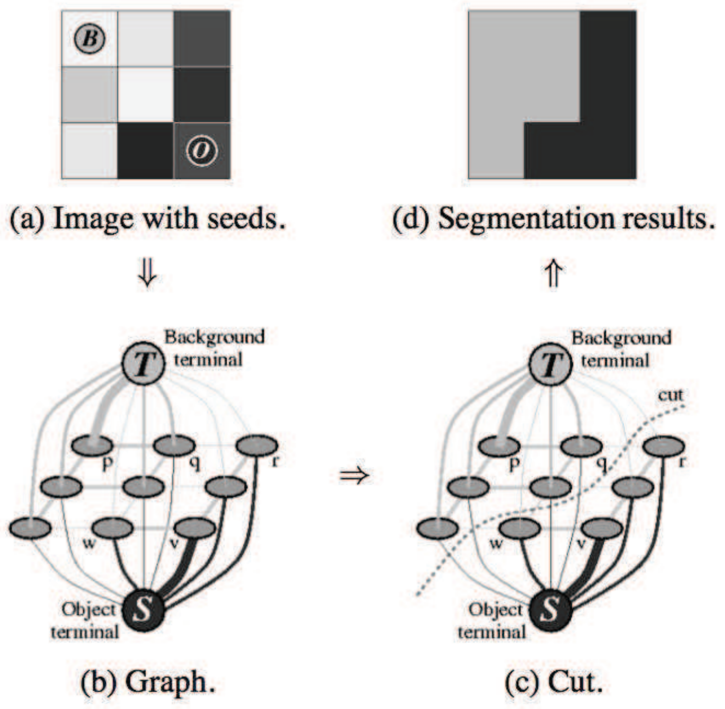


Fig. 1. Graph based silhouette extraction, from Boykov & Funka-Lea (2006).

build at each frame. When considering live real-time video, using faster cut algorithms, is not enough sufficient. The creation of the graph, as well as the structure for utilizing temporal data from previous frames must be reconsidered.
Although recent researches have given much attention to graph cuts algorithms, most of them are too computational expensive to reach real-time, limiting their applications to off-line processing. Moreover, with the introduction of the Compute Unified Device Architecture (CUDA), GPUs are no longer exclusively designed for 3D rendering. In CUDA, a GPU can be explicitly programmed as general-purpose shared memory Single Instruction Multiple Data (SIMD) multi-core processors, which allows a high level of parallelism. Thus, many applications that are not yet able to achieve satisfactory performance on Central Processing Units (CPUs), can get benefit from that massive parallelism provided by such devices. Nevertheless, only specific algorithms can be adapted and efficiently designed on GPUs.
In this chapter, we propose the design and implementation of graph cuts that is capable of handling both tracking and segmentation simultaneously, leading to real-time conditions. We first propose to track and segment objects in live video thereby utilizing regional graph cuts and object pixel probability maps, where the graph is dynamically adapted to the motion of objects. We then introduce a fast and generic novel method for graph cuts on GPUs that places no limits on graph configuration, and which is able to reach real-time performance.

2. Background on graph cuts

Graph-based segmentation has a history originally based in mathematics and graph theory. Since graphs are an abstraction of the problem space, they are often applied to many different domains. In addition to image segmentation, graph cuts have been applied to computer networking, selection maximization, and stereo vision. The graph theory that supports

graph-based segmentation methods is described, followed by a survey of frequently used algorithms, and a discussion of common graph construction techniques.
Before applications to image segmentation were demonstrated, mathematicians had been interested in computation of flows through networks. Networks were modeled as directed graphs with weights (also called *capacities*) assigned to the edges. The first important discovery in graph-based segmentation occurred in 1956, when Elias *et al* Elias et al. (1956), and Ford and Fulkerson Ford & Faulkerson (1956) independently proved the *max-flow min-cut theorem*. This theorem proved that solving for the maximum flow across a network is equivalent to solving for the minimum cut required to separate a network into two disjoint components.

2.1 Max-flow min-cut theorem

The max-flow min-cut theorem defines the relationship between the maximum flow on a *flow network*, and the minimum cut required to separate the network into two disjoint components. A *flow network* is a graph $G =< V, E >$ composed of edge set E and a vertex set V. The vertex set V contains two labeled vertices, the source s, and the sink t, and each edge $\vec{uv} \in E$ has a capacity c_{uv}, and the flow f between two vertices u and v is never more than the capacity, $f(u,v) \leq c_{uv}$. Then for a path $P_{u_0 u_n} = \{\vec{u_0 u_1}, \vec{u_1 u_2}, \ldots, \vec{u_{n-1} u_n}\}$ where $u_0, u_1, \ldots, u_n \in V$, the flow is defined as:

$$f(P_{uv}) = \min_{\vec{ij} \in P_{uv}} c_{ij} \tag{1}$$

Then the maximum flow of the network G is defined as the sum of path flows for all paths from s to t:

$$f(G) = \sum_{\forall P_{st} \in G} f(P_{st}) \tag{2}$$

Next, an *s-t cut* is defined as a partition of the vertices V into two sets $(W, \bar{W})$ such that $s \in W$ and $t \in \bar{W}$ and no edges exists from W to $\bar{W}$, as shown in Fig. 2 if the *forward edges* are removed. The max-flow min-cut theorem states that the value v of the maximum flow is equal to the value of the minimum *s-t cut* Papadimitriou & Steiglitz (1998).

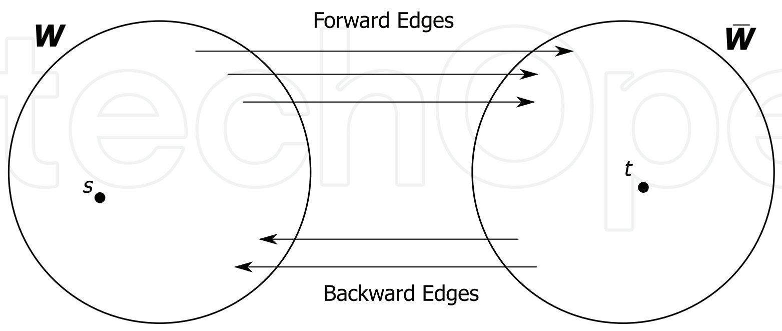


Fig. 2. Graph *s-t* partitioning

While the theorem was proved in 1956, algorithms allowing for efficient computer computation of very large graphs continued to be a focus of much research. Today, general categories of algorithms for solving the max-flow/min-cut exist and are often the basis of additional research.

2.2 Max-flow algorithms

Along with the proof of the max-flow min-cut theorem in Ford & Faulkerson (1956), Ford and Fulkerson presented an algorithm for computing the maximum flow of a flow network. This algorithm forms the basis of a set of algorithms known as *Ford-Fulkerson* or *augmenting paths* algorithms. Then, in 1988 Goldberg and Tarjan presented a new way of approaching the computation of flow with their *push-relabel* algorithm Goldberg & Tarjan (1988).

The time complexities for two *augmenting paths* algorithms and two *push-relabel* algorithms are presented in Table 1. The two classes of algorithms differ in time complexity based on the number of vertices V, or edges E. Ford-Fulkerson algorithms would generally perform better on sparse graphs (fewer edges) than do push relabel algorithms. At the time of writing, one of the fastest (and freely available) implementations is an *augmenting paths* algorithm utilizing breadth-first-search (BFS) search trees Kolmogorov & Zabin (2004).

Algorithm	Complexity	Notes
Ford & Faulkerson (1956)	$O(VE^2)$	
Ford & Faulkerson (1956)	$O(Ef)$	Only for
Edmonds & Karp (1972)	$O(V^2E)$	integer capacities BFS
Push-Relabel, Goldberg & Tarjan (1988)	$O(V^2E)$	
Push-Relabel w/ FIFO Queue, Goldberg & Tarjan (1988)	$O(V^3)$	

Table 1. Time Complexity of Maximum Flow Algorithms.

2.3 Augmenting paths algorithm

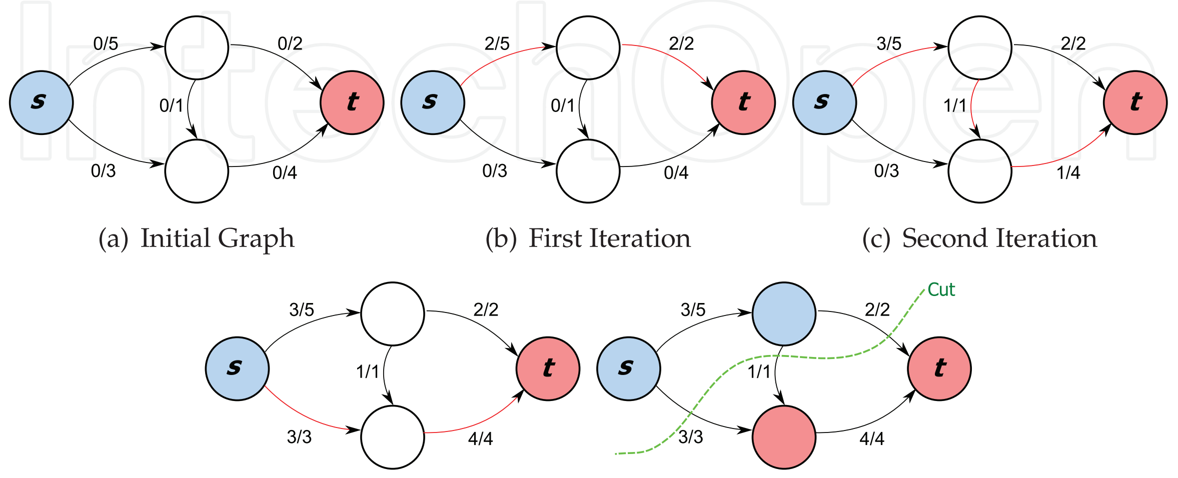
The *augmenting paths* algorithm solves the maximum flow problem by iterating searches for paths from s to t and pushing flow equal to the minimum capacity along the path. The algorithm terminates when no paths with remaining capacity exist from s to t. The pseudo-code for the algorithm is presented in Algorithm 1.

Algorithm 1: Augmenting Paths Algorithm.

```
   input  : Graph $G$ with flow capacity $c$, source vertex $s$, sink vertex $t$
   output: A flow $f$ from $s$ to $t$ which is maximum
1  forall the $\vec{uv} \in E$ do
2      $f(u,v) \leftarrow 0$
3  end
4  while $\exists$ path $P$ from $s$ to $t$ such that $c_{uv} > 0$ for all $\vec{uv} \in P$ do
5      $c_f(p) = \min(c_{uv} | \vec{uv} \in P)$
6      foreach $\vec{uv} \in P$ do
7          $f(u,v) \leftarrow f(u,v) + c_f(p)$        /* Send flow along the path */
8          $f(v,u) \leftarrow f(v,u) - c_f(p)$
9      end
10 end
```

The difference between the Edmonds-Karp algorithm and the Ford-Fulkerson algoirthm in Table 1 lies in the method used to find paths from s to t. In step 2 in Algorithm 1, the Edmonds-Karp algorithm uses a BFS search method originating from vertex s.

Fig. 3 presents a simple case of an augmenting paths algorithm. Fig. 3(a) shows the initial graph. The graph has been constructed with a source vertex and a sink vertex, and all flow has been initialized to zero. In the next panel, the first path from the source to sink is highlighted in red, and the minimum capacity has been pushed as flow. The result is that the link connecting to the sink has been fully saturated, and this path is no longer usable. This continues through panel three and four, iteratively pushing flow along *s-t paths* until no unsaturated path remains. Then the cut separates those vertices that still have an unsaturated path back to the source from those that do not.



(a) Initial Graph (b) First Iteration (c) Second Iteration

(d) Third Iteration (e) Cut

Fig. 3. Example Augmenting Paths Algorithm Execution.

2.4 Push relabel algorithm

Whereas the *augmenting paths* algorithm examines entire *s-t paths*, the *push relabel* algorithm examines individual vertices and attempts to *push* flow through the graph network. The *push relabel* algorithm relies on generating a *pseudo-flow* (a flow that does not obey the properties of the graph) and finding a convergence on a true flow. The algorithm adds two pieces of information to each vertex, a height h and an excess e. The height of the vertex is a distance measurement, reflecting how *close* the vertex is to the source. The excess is the difference between incoming and outgoing flow of a vertex, or in other words, the flow that is trapped or waiting at the vertex. A vertex with $e > 0$ is said to be *overflowing*. The algorithm consists of two main operations: *push* (also called *discharge*) and *relabel* (also called *raise*). Typically a queue of overflowing vertices is checked each iteration, with the first vertex in the list being dequeued and either *discharged* or *relabeled*. The loop terminates once there are no further viable *discharge* or *relabel* operations to perform (i.e. no remaining overflowing vertices).

Discharging a vertex causes some of the excess to be pushed to neighboring vertices when the incident edge has not completely saturated (has residual capacity). The push decreases the excess in the current vertex, increasing excess in the destination, and decreasing the residual capacity of the edge. The destination vertex is then enqueued into the overflowing list, along with the current vertex if it is still overflowing.

A vertex must be *relabeled* when it is overflowing but has no valid neighbors. A neighbor is valid if the edge incident to both vertices has remaining capacity, and the destination vertex's height is lower than the current vertex. The current vertex's height is increased to one more than the lowest neighbor, which allows further *discharging*. If a vertex has no remaining outgoing capacity, then it is relabeled to the height of the source plus one, so that the excess may be pushed back to the source.

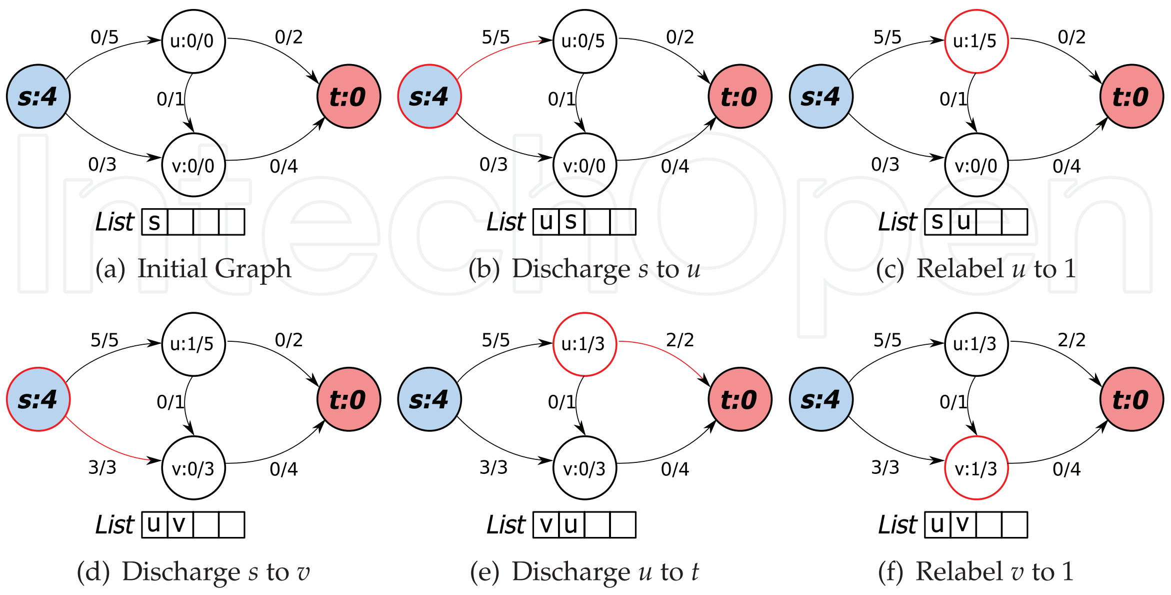
Algorithm 2: Push Relabel Algorithm.

```
input  : Graph $G$ with flow capacity $c$, source vertex $s$, sink vertex $t$
output: A flow $f$ from $s$ to $t$ which is maximum

list.enqueue($s$)
while not list.isEmpty() do
    $u \leftarrow$ list.dequeue()
    if $u$.excess $> 0$ then
        amtPushed $\leftarrow$ discharge($u$)
        if amtPushed $== 0$ then
            relabel($u$)
        end
    end
end
```

In Algorithm 2, the enqueueing of more vertices into the list would occur inside the *discharge* and *relabel* functions. Various techniques to change or improve performance have been proposed, such as pushing to the highest neighbors first, or using a first-in-first-out (FIFO) queue for vertex selection. The general time complexity given in Table 1 would seem to say that the *push relabel* algorithm is more efficient than *augmenting paths*, especially for sparse graphs, but in practice this did not seem to be the case, until recently, with parallelized implementations of the algorithm.

Fig. 4 demonstrates the operations of the *push relabel* algorithm. All vertices have their information printed alongside their label. The source and sink have their height listed after the semicolon, while vertices u and v have their height listed, follow by their current excess (after the slash). The currently active vertex is highlighted in red. There appear to be many more steps than *augmenting paths*, but this is not necessarily the case, since the path finding steps were not detailed in the previous section.



(a) Initial Graph (b) Discharge s to u (c) Relabel u to 1

(d) Discharge s to v (e) Discharge u to t (f) Relabel v to 1

Fig. 4. Example Push Relabel Algorithm Execution.

2.4.1 Parallel push relabel algorithm

As recently as 2008, researchers discovered methods for parallelization of the *push relabel* family of maximum flow algorithms. Most of the strategies implement methods so that the vertices in the overflowing list can be processed concurrently, vastly speeding up the performance of the algorithm. The parallelization of any algorithm often involves many complex concurrency issues, requiring control structures (often referred to as locks) so that the order of execution of certain statements is guaranteed. However, a lock-less implementation of the push relabel has been presented by Hong Hong (2008), which will be presented here. The algorithm constructs the graph with a *pre-flow*, effectively saturating all outgoing links from s.

Algorithm 3: Lock-free, Parallel Push Relabel Algorithm: Master Thread.

```
input  : Graph $G$ with flow capacity $c$, source vertex $s$, sink vertex $t$
output: A flow $f$ from $s$ to $t$ which is maximum
/* Initialization, performed by master thread                         */
1  $s$.height $\leftarrow |V|$
2  foreach $u \in V - \{s\}$ do
3      $u$.height $\leftarrow 0$
4  end
5  foreach $\vec{uv} \in E$ do
6      $f(u,v) \leftarrow 0$
7  end
8  foreach $\vec{su} \in E$ do
9      $f(s,u) \leftarrow c_{su}$
10     $u$.excess $\leftarrow c_{su}$
11 end
```

As stated by Hong, lines 14-17 must be performed by atomic *read-update-write* operations. These operations are usually available on multi-core systems, or are implementable via software (though at a cost). Lastly, Hong notes that while a single thread may terminate because of no available operations, it does not mean that the algorithm has terminated. Other vertices may push flow back into the vertex, making it necessary for the thread to continue operations. This problem will be address in the GPU implementation in Section 3.2.

2.5 Building graphs from images

While the max-flow min-cut theorem was first proved in 1956, it was not applied to computer vision and image segmentation for more than 30 years. The key to applying the max-flow min-cut theorem image segmentation is the ability to represent the image as a meaningful flow network, so that the min-cut can be translated back into an image segmentation. One of the first successful image processing applications was developed by Greig et al Greig et al. (1989), which demonstrated success in estimating solutions for noise removal. Then Boykov et al Boykov et al. (2001) demonstrated that the min-cut of a graph can correspond to the maximum a posteriori distribution of a Markov Random Field, which had been used for image segmentation in the past. By formulating image segmentation as an energy minimization problem, and then a max-flow min-cut problem, Boykov et al created *graph cuts* for image segmentation.

Algorithm 4: Lock-free, Parallel Push Relabel Algorithm: Child Threads.

```
input  : A vertex $u$
output: None
/* Main loop performed per thread, one thread per vertex      */
while $u$.excess $> 0$ do
    $\acute{e} \leftarrow u$.excess
    $\hat{v} \leftarrow$ null
    $\hat{h} \leftarrow \infty$
    foreach $\vec{uv} \in E_f$ do                          /* edges with capacity */
        $\acute{h} \leftarrow v$.height
        if $h\prime < \hat{v}$.height then
            $\hat{v} \leftarrow v$
            $\hat{h} \leftarrow \acute{h}$
        end
    end
    if $u$.height $> \hat{h}$ then                                  /* Discharged */
        $d \leftarrow \min(\acute{e}, c_{uv} - f(u,v))$
        $f(u,v) \leftarrow f(u,v) + d$
        $f(v,u) \leftarrow f(v,u) - d$
        $u$.excess $\leftarrow u$.excess $-d$
        $v$.excess $\leftarrow v$.excess $+d$
    else                                                              /* Relabel */
        $u$.height $\leftarrow \hat{h} + 1$
    end
end
```

The graph cut creation process consists of splitting the edges of the graph into two categories. All edges that are incident to the source or sink are called *t-links* (terminal links), while the other links are called *n-links* (neighbor links). The two different types of links are assigned weights (energy) using different methods, and give rise to the energy equation which is to be minimized by the graph cut.

$$E(A) = \lambda R(A) + B(A) \tag{3}$$

where

$$R(A) = \sum_{p \in \mathcal{P}} R_p(A_p) \tag{4}$$

$$B(A) = \sum_{\{p,q\} \in \mathcal{N}} B_{p,q} \cdot \delta_{A_p \neq A_q} \tag{5}$$

Equation 4 defines the regional energy, which corresponds to the t-links of the graph. This energy is often a model of the object to be extracted. Research has been conducted into various object models such as Gaussian mixture models Rother et al. (2004) and color histograms Garrett & Saito (2008). One simple method for t-link energy computation uses histograms. For a given pixel p and a histogram H, the histogram value for p, $H(p)$, is divided by the maximum histogram value to obtain an energy between $[0.0, 1.0]$ as demonstrated in Equation 6.

$$R(p) = \frac{H(p)}{\max_{\forall q \in V} H(q)} \tag{6}$$

Equation 5 is the boundary energy, corresponding to the n-links in the graph. The most common equation used for the boundary energy is the square of the Euclidean distance between two pixels. Equation 7 demonstrates the energy between two RGB pixels ($p^T = [r, g, b]$), normalized to the range $[0.0, 1.0]$. This produces a linear energy model, which is often not optimized for practical use, as the difference between colors captured in real scenes does not exhibit enough variance in the RGB color space. Instead negative exponential functions (Equation 8) can replace the Euclidean difference.

$$B(\{p, q\} \in V) = 1.0 - ||p - q||/3.0 \tag{7}$$

$$B(\{p, q\} \in V) = e^{-||p-q||} \tag{8}$$

Fig. 5 demonstrates the differences in boundary energy equations. The x axis is the difference between the pixels ($||p - q||$), and the y axis is the resultant energy. The exponential function decreases much faster for small differences of pixel intensity earlier than does the linear function.

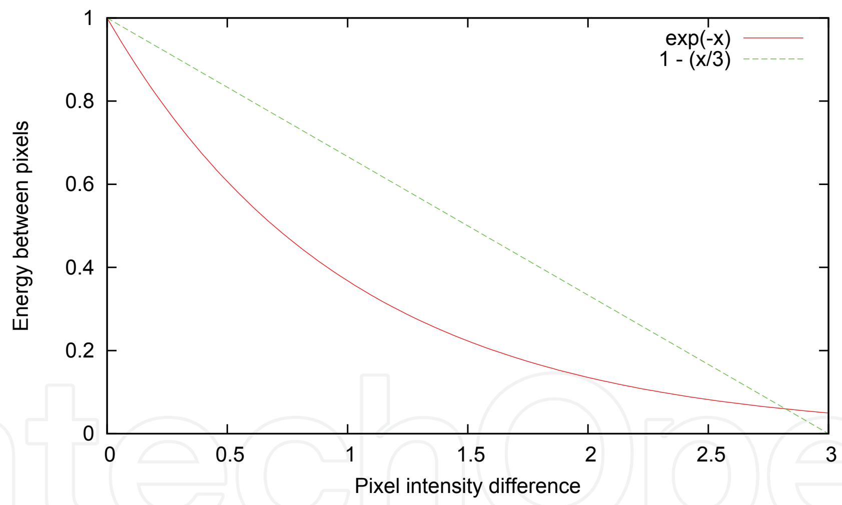


Fig. 5. Boundary Equation Comparisons

Finally, graph construction can be summed up by the energy table by Boykov et al Boykov & Funka-Lea (2006), which has been reproduced here. The $\mathcal{K}$ in Table 2 refers to a maximum energy level, computed as:

$$\mathcal{K} = 1 + \max_{p \in \mathcal{P}} \sum_{q \in \mathcal{N}_p} B_{pq} \tag{9}$$

This equation defines that links may have hard constraints, limiting them to either the foreground or background by ensuring that their neighbor links are fully saturated due to the large amount of energy in the t-link from the source, or the large drain to the sink.

Edge	Weight (Energy)	For
$\vec{pq}$	B_{pq}	$\{p,q\} \in V$
	$\lambda \cdot R_p(\text{"}bkg\text{"})$	$p \in \mathcal{P}, p \notin O \cup B$
$\vec{ps}$	$\mathcal{K}$	$p \in O$
	0	$p \in B$
	$\lambda \cdot R_p(\text{"}obj\text{"})$	$p \in \mathcal{P}, p \notin O \cup B$
$\vec{pt}$	0	$p \in O$
	$\mathcal{K}$	$p \in B$

Table 2. Edge Energy Computations per Edge Type.

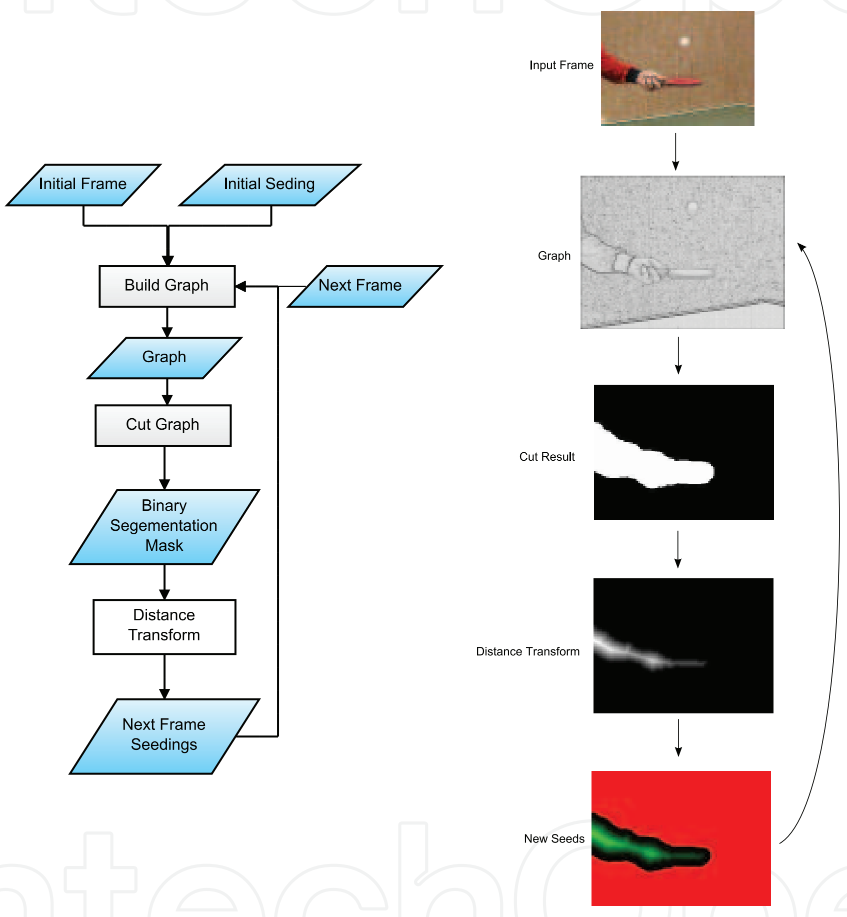


Fig. 6. Flow Chart of Tracking System.

3. Proposed system

3.1 Live video extraction

The method for tracking and silhouette extraction is composed of three main steps: building the graph, cutting the graph, and creating temporal seed information for the next frame. These three steps are performed in a cycle, with the temporal seeds being used in the next build phase, shown in Fig. 6. The images on the right show the image representation of data at each step of the algorithm.

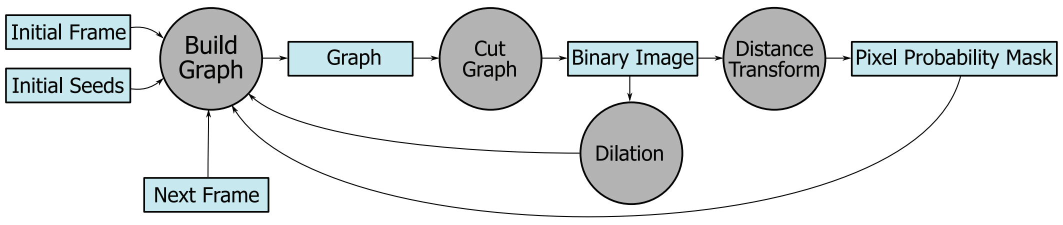


Fig. 7. System Operation Flow Chart.

3.1.1 Build the graph

Each edge in the graph requires an energy computation, and as the size of the image increases, the number of edges increases dramatically. For example, an 8-neighborhood graph on a 640x480-resolution video has almost 2 million edges. This results in poor performance on live video because 2 million edges must be recomputed for every frame. If an object is tracked and segmented that does not fill the entire frame, many of the pixels will not need to be computed, such as those far from the object, since they will not likely change unless the object moves closer. To reduce the number of computations required, the graph is restricted to a region-of-interest.

For the very first cut on the graph, the region-of-interest will cover the entire video frame. However, subsequent regions will be smaller, depending on the object size.

3.1.2 Cut the graph

The second step is to cut the graph. For this step, a freely available graph cut implementation by Kolmogorov *et al* Kolmogorov & Zabin (2004) was used. This implementation relies on a modified *augmenting paths* algorithm, as described earlier.

3.1.3 Temporal seeding

Seeding the graph for the next frame is not a simple matter of marking the previous frame's object pixels as object seeds, and all other pixels as background seeds. If the object were to move within the next frame, then some seeds would be mislabeled, also causing the cut to incorrectly label some pixels. This research describes a new method for seeding pixels based on probabilities of future pixel locations.

When examining any two consecutive frames of a video sequence, many of the pixels of the object will overlap between frames. These will most likely be those pixels closer to the center of the object, as the object must travel farther before these pixels change from object to background. Thus, a probability distribution can be created over the object pixels such that the pixels near the edge will have a lower probability of remaining object pixels in the next video frame. Pixels near the center of the object have a high probability of remaining object pixels in the next video frame. This distribution we call the object pixel probability map.

To compute the pixel probability map, the binary mask of the object segmentation from the previous frame is computed. Then a distance transform is applied to the mask to obtain the pixel probability map, shown on the right hand side of Fig. 7. The distance transform computes the distance from a given white pixel to the closest black pixel. This gives us a gradient from the center of the object (greatest distance) to the cut edge (shortest distance).

Using the probability map, the t-links of the next frame's graph are computed using a modified version of Region Equation R described by Equation 5. Table 3 shows the new energy computations for t-links.

Edge	Weight (Energy)	For
$\{p,s\}$	$\min(\lambda+1, \lambda \cdot R_p(bkg) + \lambda \cdot D_p)$	$p \in P, p \notin O$
	0	$p \in O$
$\{p,t\}$	$\min(\lambda+1, \lambda \cdot R_p(bkg) + \lambda \cdot D_p)$	$p \in P, p \notin B$
	0	$p \in B$

Table 3. Modified Boundary Energy Equation

Where D_p is the value of the probability map for pixel p divided by the largest D, and thus $D_p \in [0.0, 1.0]$. The technique also requires that $R_p \in [0.0, 1.0]$ to make sure that the maximum value of the t-link weight is $\lambda + 1$ (labeled as K in Table 2).
Finally, a region-of-interest is constructed around the object area of the binary image mask. The region-of-interest is then expanded to contain probable future locations of the object. Metrics for determining the size of potential for the region-of-interest include camera frame rate, previous object velocities, and object scaling speeds.

3.1.4 Focused graph cuts

Graph cuts are notoriously computationally expensive. The graph size grows exponentially with image size (image size growth means growing in width and height), and easily includes hundreds of thousands of vertices and millions of edges, as seen in Fig. 8. To alleviate this problem, the proposed system focuses only on those pixels which are most likely to contain the object from the previous frame. By utilizing a dilation of the previous graph cut's resultant binary image, the operations of function graph cuts ca be masked, resulting in a smaller graph on which to operate. Similar methods which use banded graph cuts, cutting only around the boundary of the object, have been proposed Lambaert et al. (2005); Mooser et al. (2007), however these methods have been shown to lose information and misclassify pixels in certain topologies Garrett & Saito (2008); Sinop & Grady (2006).

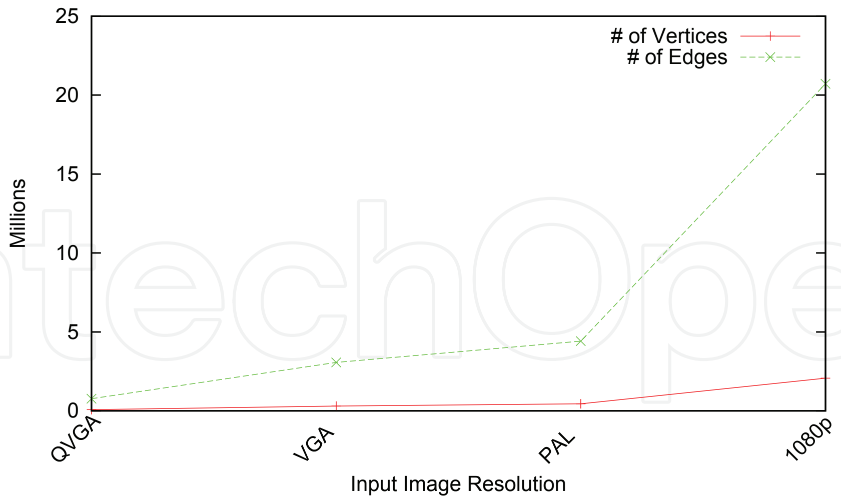


Fig. 8. Graph Growth Size.

Initial timing analysis of graph cuts shown in Fig. 9 demonstrates that the building of the graph consumes a majority of the time, as many floating point operations must be performed (typically slow on CPUs). To speed up the graph cuts even more, the next section details how to harness the parallelization and floating computations of the GPU.

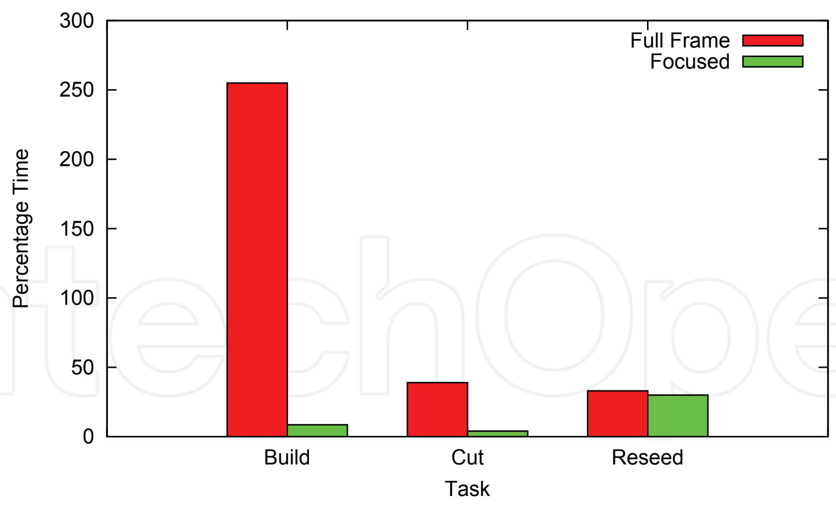


Fig. 9. Graph Timing per Task.

3.2 GPU implementation using CUDA

Driven by the insatiable market demand for real-time, high-definition 3D graphics, the programmable GPU has evolved into a highly parallel, multi-threaded, many-core processor with tremendous computational horsepower and very high memory bandwidth.

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, high-parallel computation, and therefore designed such that more transistors are devoted to data processing rather that data caching and flow control.

NVidia's CUDA allows researchers to easily parallelize their computing task to take advantage of the GPU. CUDA provides a software library used for interfacing with the graphics processor, and a specialized compiler to create executable code to run on the GPU. CUDA presents programmers with two other types of memory access, shared memory and global memory, on top of the texture cache units. Being able to access to all the memory on the GPU (albeit with varying access times) has made it much easier to perform research into GPGPU computing. The method presented in this paper was developed using the CUDA architecture and run on nVidia hardware. There are many technical resources available on the nVidia CUDA homepage nVidia (2009).

The following description of the CUDA model is stated in terms of its hardware and software components. The hardware model, shown in Fig. 10, illustrates how each processor *core* is laid out in relation to the different memory buses of global memory, texture cache, and shared memory. Device (global) memory is the slowest to access while the texture cache is only available for read operations. Shared memory is used by all of the threads in a block, as described by the software model.

The software model, depicted in Fig. 11, shows the how a *block* organizes *threads*, and how the *grid* of a kernel is organized into *blocks*. When an invocation of kernel code is to be called, CUDA requires specifying the dimensions of the *grid* and the size of *blocks* within the grid (each block is the same size). A collection of *threads* that will be run in parallel on the SIMD cores is defined as a *warp*. Operations such as branching will cause threads to diverge into separate warps which must be run serially in time-shared environments. Each thread and each block have a two dimensional ID that can be accessed at any time. This ID is often used to reference memory locations pertaining to the particular thread.

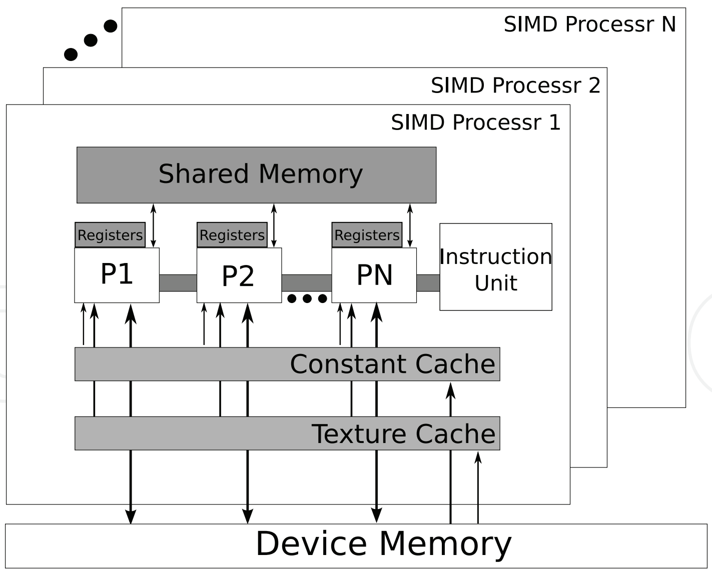


Fig. 10. CUDA Hardware Model

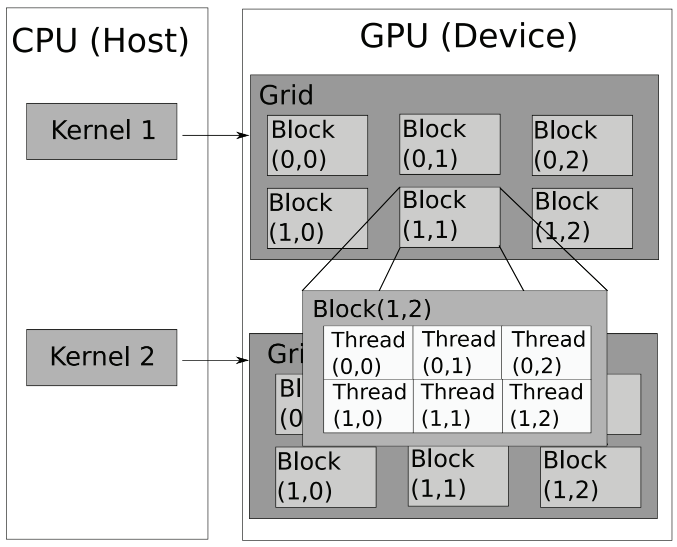


Fig. 11. CUDA Programming Model

3.2.1 GPU Programming paradigm

To develop a GPU implementation of a traditional algorithm, both data structure layout (the graph representation in memory) and concurrency bottlenecks must be carefully considered to realize the full potential of the GPU. CUDA implementations of graph cuts for image segmentation have traditionally paired the 2D lattice structure of graphs with the 2D grid structure of the CUDA programming model. However this leads to restrictions on the types of graphs that can be processed by the routine. Furthermore, traditional graph cut algorithms contain too much cyclomatic complexity in the form of branching and looping. In CUDA, divergent code produces poor performance because the SIMD model cannot perform the instructions in parallel, making new techniques for graph cuts necessary.

Vineet and Narayanan presented a method of graph cuts in CUDA that has shown improved performance by pairing the 2D lattice graphs with the CUDA programming model. They conceded that this method would not be feasible for other types of graphs Vineet & Narayanan

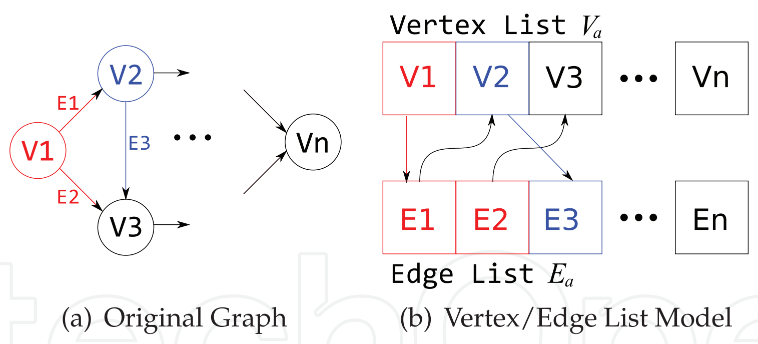


Fig. 12. Conversion of a direct graph into vertex and edge lists.

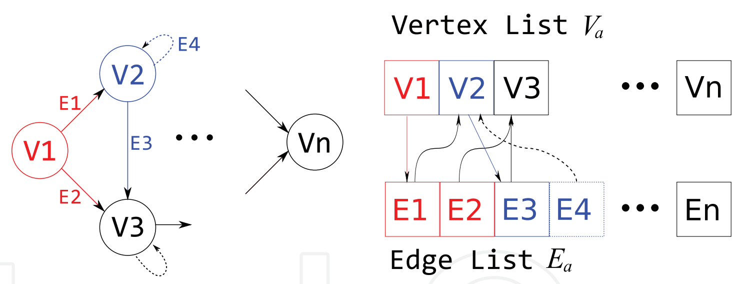
(2008). In this research, the graphs were structured using a technique similar to that presented by Harish et al Harish & Narayanan (2007) that improved performance in distance calculations on arbitrary graphs using the GPU by representing graph $G(V, E)$ as a vertex array V_a and an edge array E_a. Edges in the edge list are grouped so that all edges that originate at a vertex are contiguous, and the vertex contains a pointer to the first edge in the group, as in Fig. 12. Each edge structure holds information about its capacity, current flow, and destination vertex.

3.2.2 Regular graphs

The technique used in this research improved upon the basic implementation of vertex and array lists. To make the list-based graph representation more SIMD friendly, the input graph is made *regular* by adding null edges to any vertex that has fewer edges than the vertex with the highest degree. Second, the code is simplified by only having one kernel, which performs both the discharging and relabeling steps.
Since the GPU is treated as an array of SIMD processors, it is important that the algorithm is able to perform the same operation across multiple data locations. However, this assumes that any kind of graph configuration could be used as input, requiring that each vertex be treated uniquely, as the vertex set is not homogeneous. Particularly in the case of image segmentation, there are three classes of vertices: center vertices, edge vertices, and corners vertices, which have 10, 7, and 5 edges respectively (in an 8-neighborhood graph). Since the vertices must be treated differently, the CUDA framework cannot perform the operations in parallel and it serializes the kernel calls, negating the benefit of using the GPU.
To overcome this problem, the new method used in this research adds extra edges to the graph so that each vertex has the same degree. In Fig. 13, the dotted line of edge $E4$ is an example of a null edge. These null edges point back to the originating vertex and have a capacity of zero. This causes the null edges to be ignored by the graph cut algorithm, as the destination of the edge never has a height lower than the origin (a condition for discharging), and the capacity of the edge restricts the amount of flow discharged to zero.

3.2.3 GPU graph cuts on regular graphs

Algorithm 5 details the steps of the computation that take place on the CPU. The program operates on a list of vertices V, and a list of edges E. The *initialize* function starts by discharging the maximum possible amount of flow from the source to the neighboring vertices. Then the GPU kernel is invoked so that one thread is excuted for each non-source and non-sink vertex. The CPU continues to invoke the GPU kernel until no discharge or relabel operations can be performed.



(a) 2-regular Original Graph (b) Vertex/Edge List Model

Fig. 13. Addition of null edges to achieve a k-regular graph.

Algorithm 5: Proposed System Host Graph Cut Loop

```
finished $\leftarrow$ false
Initialize(V, E)
while not finished do
    // GraphCutKernel() is performed in parallel on the GPU

    finished $\leftarrow$ GraphCutKernel(V,E)
end
```

The GPU kernel is detailed in Algorithm 6. Since the graph is regular, the loop over the neighbors is fully unrolled. The null edges are ignored because the height of u will never be less than the current vertex's height (since they are equal). In addition the discharge can be modified to push a conditionally assigned amount of flow. Conditional assignment allows us to use of the ternary operator, which prevents divergent code since it can be optimized to a zeroing of either operand of an addition.

Algorithm 6: Proposed System GPU Kernel

```
tId $\leftarrow$ GetThreadID()
amtPushed $\leftarrow$ 0
foreach Neighbor $u$ of V[tId] do
    if $u$.height $<$ V[tId].height and $\overrightarrow{(\text{V[tId]}, u)} \in E$ has capacity $> 0$ then
        amtPushed $\leftarrow$ amtPushed + Discharge(V[tId], $u$)
    end
end
if amtPushed $> 0$ and V[tId].excess $> 0$ then
    V[tId].height = FindLowestNeighbor(V[tId]).height + 1
end
```

4. Experimental results

The GPU experiment analyzes the performance of the GPU push relabel implementation described in Section 3.2. The goal of the implementation is to be able to realize the results of the graph cut faster than the traditional CPU implementations. To measure the performance

benefit of the GPU approach, a pure GPU implementation was compared against the previous CPU implementation and against a hybrid CPU/GPU system.
This section begins with the drawbacks of the GPU approach. Then, as remarked at the end of the CPU test analysis, the two areas identified for improvement are the graph building and the graph cutting areas, which are covered in the following two sections respectively.
These tests utilized two video sequences scaled to QVGA (320x240), VGA (640x480), HD 720p (1280x720), and HD 1080p (1920x1080) resolutions.
All tests in this section were performed on an 2.5 GHz Intel Xeon processor with 2 GB of RAM and a nVidia GTX 280 graphics card running Windows XP. This information is important in analyzing the performance results in the following sections, as results are likely to differ with hardware.

4.1 GPU transfer latency

Due to computer hardware architecture, the memory on the GPU is not directly accessible from the CPU. An important limitation of the older GPUs was the time required to transfer the input to the GPU memory and the time required to transfer the GPU output back to the CPU. This latency from transferring data is not applicable to the CPU-only implementations of graph cuts, since all the data resides in the CPU main memory. Traditional Advanced Graphics Port (AGP) technology facilitated fast data transfer to the GPU, but the return path to the CPU was considered unimportant, since the output would then be written to the display device. Even with the new Peripheral Component Interconnect Express (PCIe) bus architecture, at very large image resolutions, minor latency is observed. Fig. 14 demonstrates that for even large HD 1080p (1920x1080) resolution images the absolute maximum frame rate (due to transfer latency) would be approximately 30 frames-per-second.

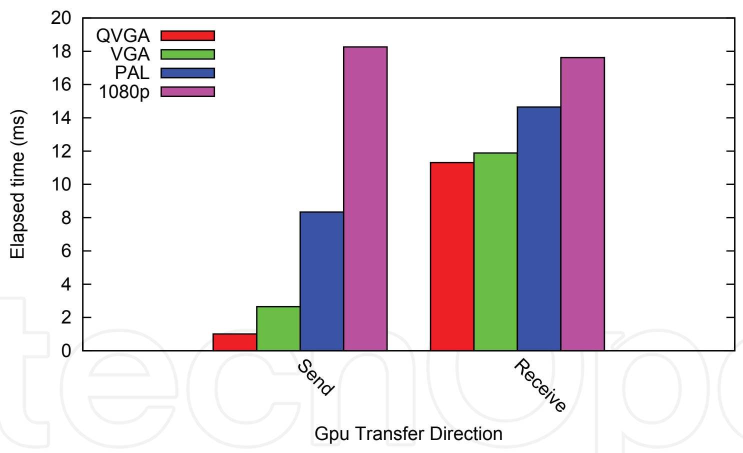


Fig. 14. Graph of Elapsed Time per GPU Task

4.2 GPU graph building

Due to the parallelism in GPUs, even greater performance benefits can be obtained using CUDA. The GPU performed so well during this task that the histogram bars cannot be seen, and instead the values are written. Furthermore, unlike focused graph cuts, this task is content independent since it works on the entire frame at the same time.
Table 4 gives exact performance numbers for the results shown in Fig. 15. In the final row, the speed up multiplier is computed as the ratio between the GPU implementation and the focused graph cuts implementation. For HD 1080p video, GPU graph building achieves as

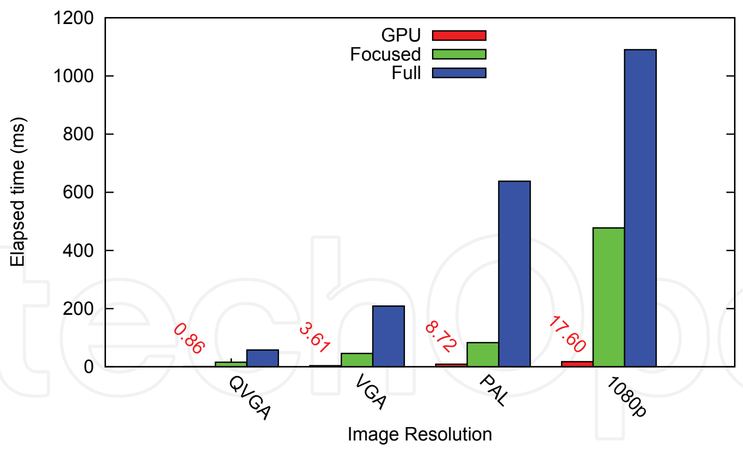


Fig. 15. Build Times for Various Image Resolutions

high as a 26x speed increase over the focused graph building, and even greater over traditional graph building methods.

	QVGA	**VGA**	**720p**	**1080p**
Full	57.96	208.91	638.07	1090.52
Focus	15.59	45.82	82.99	477.55
GPU	0.86	3.61	8.72	17.60
Speedup	17.5x	12.7x	9.5x	26.1x

Table 4. Speed Results for GPU Graph Building.

4.3 GPU push relabel

This section compares the cut speeds of the GPU implementation of the system. Fig. 16 shows that at smaller resolutions, the GPU implementation does not see significant improvement in speed. As the raw clock speed of the CPU is higher than that of the GPU (4x greater), this is to be expected. However, at the higher resolutions (720p and 1080p), the GPUs parallelization capabilities start to have an impact on the performance.

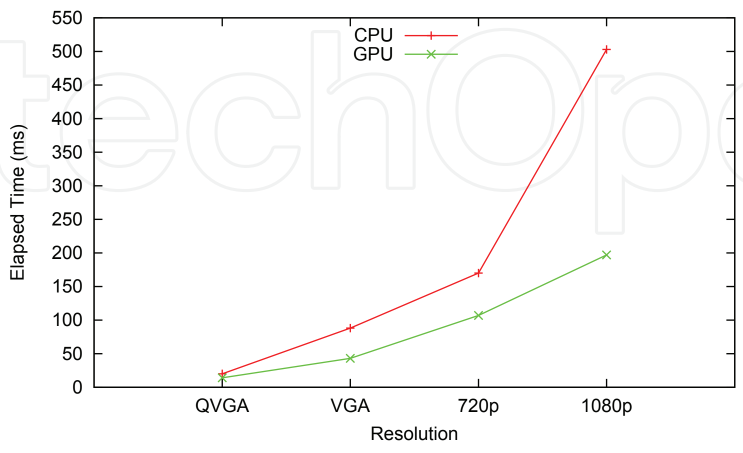


Fig. 16. Cut Times for Various Image Resolutions.

Table 5 gives exact performance numbers for the results shown in Fig. 16.

		QVGA	VGA	720p	1080p
	Avg	20	88	170	503
CPU	Min	15	78	156	453
	Max	32	94	170	503
	Avg	14	43	107	197
GPU	Min	11	39	91	180
	Max	18	47	117	207

Table 5. Speed Results for GPU Graph Building.

4.4 GPU Summary

Experimental data show that the GPU implementation is able to perform on average 9 times faster than the CPU implementation for large resolutions (see Table 6). Standard video broadcasts (up to VGA) could be segmented with very little delay as the frame rates approach 15 fps. However, HD video is still beyond the reach of real-time systems, averaging only 3 fps.

	QVGA	VGA	720p	1080p
CPU	150	739	1739	3094
GPU	23	69	186	336
Speedup	6.5x	10.7x	9.3x	9.2x

Table 6. Summary Results for Full System Performance Tests.

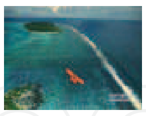
Fig. 17 contains sample frames from output video, scaled to fit the page, and scaled relative to each other in order to realize the size difference in resolutions. The segmentation boundary is designated by the red border around the yellow airplane. The ocean and islands are segmented as background, and the airplane is segmented as the object of interest. The videos are samples obtained from the Microsoft *WMV HD Content Showcase* website Microsoft (2009).

5. Conclusions

This chapter has presented research on a novel system for object silhouette extraction. The originality of the system comes from three distinct method improvements. First, the research developed a method for tracking and segmenting objects across a sequence of images by using temporal information and distance transforms. Second, the method performance was increased by focusing the graph cut on likely image pixels, reducing the graph size to only those pixels that are considered useful. Finally, creating an implementation on the GPU using vertex and edge lists, as opposed to 2D memory arrays, and making the graph regular by adding *null* edges further improved performance.
Through testing on various image sequences and the standardized Berkeley segmentation data set, the research showed the new system provides fast and accurate results. Future research extensions include alternative color models for objects and new boundary energy equations. Both hold promise for better segmentation in the occlusion case, but must be balanced with computational complexity, otherwise the improvement in speed realized by the system will be negated.

6. Acknowledgments

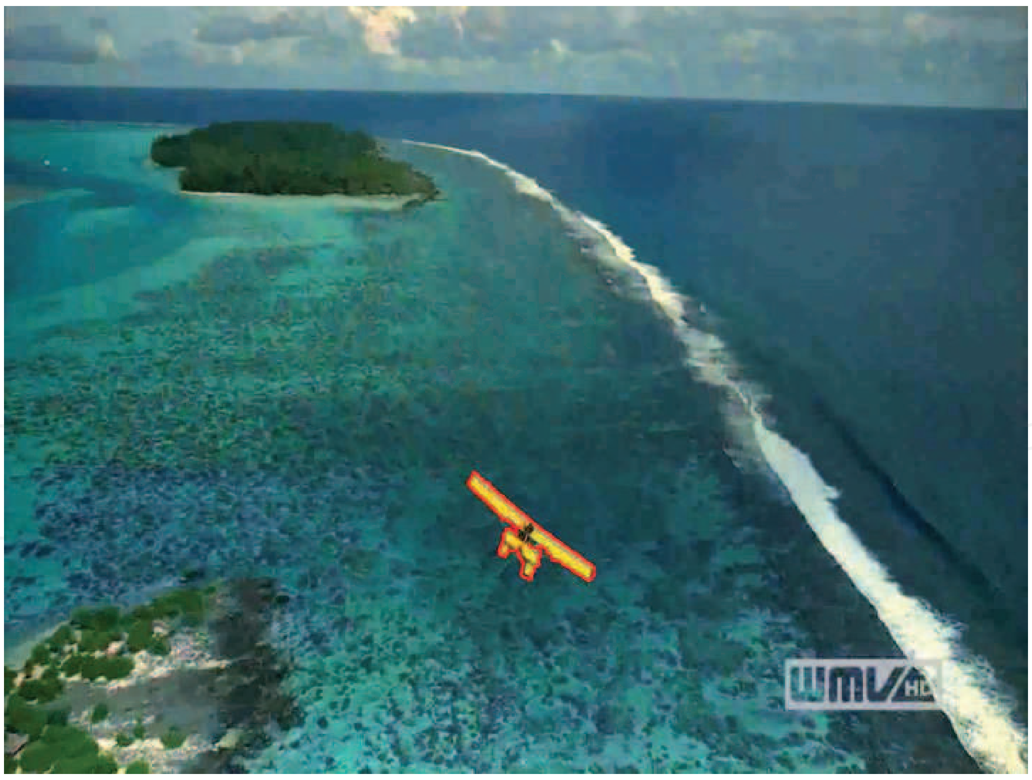
This work is supported by the National Institute of Information and Communications Technology (NICT), Japan.



(a) QVGA

(b) VGA

(c) 720p



(d) 1080p

Fig. 17. Sample Output of GPU Video Sequence Performance Experiment.

7. References

Boykov, Y. & Funka-Lea, G. (2006). Graph cuts and efficient n-d image segmentation, *International Journal of Computer Vision* 70(2): 109–131.

Boykov, Y. & Kolmogorov, V. (2004). An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision, *IEEE Trans. Pattern Anal. Mach. Intell.* 26(9): 1124–1137.

Boykov, Y., Veksler, O. & Zabih, R. (2001). Fast approximate energy fast approximate energy minimization via graph cuts, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23(11): 1222–1239.

Edmonds, J. & Karp, R. (1972). Theoretical improvements in algorithmic efficiency for network flow problems, *Journal of the ACM* 19(2): 248–264.

Elias, P., Feinstein, A. & Shannon, C. (1956). A note on the maximum flow through a network, *IRE Transactions on Information Theory* 2(2): 117–119.

Ford, L. & Faulkerson, D. (1956). Maximal flow through a network, *Canadian Journal of Mathematics* 8: 399–404.

Garrett, Z. & Saito, H. (2008). Live video object tracking and segmentation using graph cuts, *International Conference on Image Processing*, IEEE, pp. 1576–1579.

Goldberg, A. & Tarjan, R. (1988). A new approach to the maximum-flowproblem, *Journal of the ACM* 35(4): 921–940.

Greig, D., Porteous, B. & Seheult, A. (1989). Exact maximum a posteriori estimation for binary images, *Journal of the Royal Statistical Society* 51(2): 271–279.

Harish, P. & Narayanan, P. J. (2007). Accelerating large graph algorithms on the gpu using cuda, *IEEE High Performance Computing*, pp. 197–208.

Hong, B. (2008). A lock-free multi-threaded algorithm for the maximum flow problem, *Workshop on Multithreaded Architectures and Applications, in Conjunction With International Parallel and Distributed Processing Symposium*, IEEE Computer Society.

Juan, O. & Boykov, Y. (2006). Active graph cuts, *CVPR '06: Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, IEEE Computer Society, Washington, DC, USA, pp. 1023–1029.

Kolmogorov, V. & Zabin, R. (2004). What energy functions can be minimized via graph cuts?, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26(2): 147–159.

Lambaert, H., Sun, Y., Grady, L. & Xu, C. (2005). A multilevel banded graph cuts method for fast image segmentation, *IEEE Conference on Computer Vision*, Vol. 1, IEEE Computer Society, Los Alamitos, CA, USA, pp. 259–265.

Laurentini, A. (1994). The visual hull concept for silhouette-based image understanding, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16(2): 150–162.

Microsoft (2009). Wmv hd content showcase, *Microsoft Corporation* .
URL: *http://www.microsoft.com/windows/windowsmedia/musicandvideo/hdvideo/contentshowcase.aspx*

Mooser, J., You, S. & Neumann, U. (2007). Real-time object tracking for augmented reality combining graph cuts and optical flow, *International Symposium on Mixed and Augmented Reality*, Vol. 00, IEEE, IEEE Computer Society, pp. 1–8.

Mortensen, E. & Barrett, W. (1995). Intelligent scissors for image composition, *22nd Annual Conference on Computer Graphics and Interactive Techniques*, pp. 191–198.

nVidia (2009). Cuda zone, *nVidia* .
URL: *http://www.nvidia.com/object/cuda_home.html*

Papadimitriou, C. & Steiglitz, K. (1998). *Combinatorial Optimization: Algorithms and Complexity*, Courier Dover Publications.

Rother, C., Kolmogorov, V. & Blake, A. (2004). Grabcut: Interactive foreground extraction using iterated graph cuts, *ACM Transactions on Graphics* 23: 309–314.

Shi, J. & Malik, J. (1997). Normalized cuts and image segmentation, *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 731–737.

Sinop, A. & Grady, L. (2006). Accurate banded graph cut segmentation of thin structures using laplacian pyramids, *International Conference on Medical Image Computing and Computer-Assisted Intervention*, Vol. 9, pp. 896–903.

Takaya, Y., Garrett, Z. & Saito, H. (2009). Player segmentation in sports scenes using graph cuts with a playing field constraint, *Meeting on Image Recognition and Understanding*.

Ueda, E., Matsumoto, Y., Imai, M. & Ogasawara, T. (2003). A hand-pose estimation for vision-based human interfaces, *IEEE Transactions on Industrial Electronics* 50(4): 676–684.

Vineet, V. & Narayanan, P. J. (2008). Cuda cuts: Fast graph cuts on the gpu, *IEEE Conference on Computer Vision and Pattern Recognition*, IEEE Computer Society, pp. 1–8.