Real-Time GPU Based Video Segmentation with Depth Information

Nilangshu Bidyanta
Department of Electrical and Computer Engineering
University of Arizona
Tucson, AZ
nbidyanta@email.arizona.edu

Ali Akoglu
Department of Electrical and Computer Engineering
University of Arizona
Tucson, AZ
akoglu@email.arizona.edu

***Abstract*—In the context of video segmentation with depth sensor, prior work maps the Metropolis algorithm, a simulated annealing based key routine during segmentation, onto an Nvidia Graphics Processing Unit (GPU) and achieves real-time performance for 320x256 video sequences. However that work utilizes depth information in a very limited manner. This paper presents a new GPU-based method that expands the use of depth information during segmentation and shows the improved segmentation quality over the prior work. In particular, we discuss various ways to restructure the segmentation flow, and evaluate the impact of several design choices on throughput and quality. We introduce a scaling factor for amplifying the interaction strength between two spatially neighboring pixels and increasing the clarity of borderlines. This allows us to reduce the number of required Metropolis iterations by over 50% with the drawback of over-segmentation. We evaluate two design choices to overcome this problem. First, we incorporate depth information into the perceived color difference calculations between two pixels, and show that the interaction strengths between neighboring pixels can be more accurately modeled by incorporating depth information. Second, we pre-process the frames with *Bilateral* filter instead of Gaussian filter, and show its effectiveness in terms of reducing the difference between similar colors. Both approaches help improve the quality of the segmentation, and the reduction in Metropolis iterations helps improve the throughout from 29 fps to 34 fps for 320x256 video sequences.**

Keywords-cuda; gpu; kinect; depth; video segmentation;

I. INTRODUCTION

Video segmentation is one of the fundamental tasks in computer vision. It deals with clustering neighboring pixels in a video frame with similar visual cues into different spatial regions, while maintaining their temporal coherence in appearance and motion. As such, video segmentation is the basis for many higher-level vision applications, including object tracking, video summarization, event detection, etc.

Despite extensive studies of the problem for decades, video segmentation remains as a challenge due to several reasons. For example, segmenting homogeneous spatial regions relying only on visual cues is still difficult and error prone. It is especially true when two regions that are adjacent or overlap have similar colors. On the other hand, maintaining temporal coherence of segmented regions can be easily made complicated when object occlusion or deformation occurs. Finally, the sheer volume of pixels in a video often makes the processing speed a bottleneck, unsuitable for many applications with real-time requirement.

Recent advances in several different technological fronts, however, have brought new hopes to address the above issues. One such notable breakthrough is the availability of capable but inexpensive depth sensors, such as Microsoft Kinect and ASUS Xtion Pro. These sensors usually have two cameras placed side-by-side: one RGB camera for capturing color information, and one Infrared (IR) camera for capturing scene depth. By providing information in RGB-D domain jointly, those depth sensors have spawned many interesting applications ranging from interactive learning to remote healthcare. Another development forefront worth noticing is the General Purpose GPU (GPGPU) computing platform. In the past decade, the computational power of GPGPUs has been growing steadily that they have become legitimate alternatives to CPUs in some cases, especially for data-intensive applications with a high degree of inherent parallelism. In fact, the parallel processing capabilities of GPGPUs nowadays are often orders-of-magnitude higher than those of more expensive CPUs.

In this paper, we explore the application of depth sensors in video segmentation on a GPU-CPU integrated computing platform. In this framework, the inclusion of depth information in addition to RGB color cue helps improve the segmentation accuracy and temporal consistency. And by designing highly parallelized segmentation algorithm, we are able to take advantage of the GPU's massive computational power to achieve real-time processing speed for videos up to 320x256 resolution.

We next present our work in detail in the following order. In Section 2, we first give an overview about the related work in the field and at its end highlight our contributions. Section 3 explains our proposed video segmentation flow and detailed GPU implementation of the Metropolis algorithm. In Section 4, we evaluate the performance of our segmentation framework and present several case studies on further improving its performance. Section 5 concludes the paper and points out future directions.

II. RELATED WORK

Several methods are available for object detection and tracking in images and video. Popular modern algorithms include the Scale-Invariant Feature Transform (SIFT) [2] as well as the more recent Tracking-Learning-Detection (TLD) [3], which both rely on prior knowledge of the region of

978-1-5386-9120-5/18/$31.00 ©2018 IEEE

interest. The segmented visual data can be used for higher level vision tasks which require spatial and temporal relations between objects to be established [10][11].

Traditionally, segmentation algorithms have been implemented on CPUs using different approaches [7][8][9]. For example, authors of [7] use the medoid shift algorithm to obtain the clusters in the feature space of the image. The authors of [9] use the mean shift algorithm, a non-parametric clustering technique which does not require prior knowledge about the number and shape of clusters. In [8], the authors treat image segmentation as a graph partitioning problem. The image is converted into a connected weighted graph. The partitioning is based on calculating the eigenvalues and eigenvectors of the matrix representation of this graph.

Since image segmentation is a computationally intensive algorithm, it is an excellent candidate for parallelization on the GPU. This opens up the possibility of segmenting frames of a video sequence in real time. Authors of [6] implement the mean shift algorithm on the GPU. Their implementation can process 5 - 10 frames per second (fps). The authors of [16] implement an image segmentation algorithm on the GPU based on fuzzy connectedness. They achieve a frame rate of 4 fps. Because a smooth motion observation requires a frame rate of at least 16 fps, neither of the above implementations is suitable for real-time video segmentation.

Some of the works that natively process video for segmentation include [12], [17] and [18]. The authors of [17] use a spatio-temporal tree based segmentation approach. The algorithm requires the granularity of the segmentation as an input. This needs to be set according to the content of the scene being segmented. Owing to the complexity of the algorithm and its implementation on the CPU, they achieve a frame rate of 1 fps. Of particular interest with regards to our work is [12], where a depth map is used in conjunction with color information to refine a segmentation obtained through motion detection. While the authors achieve real-time performance, all moving objects are segmented as one large object. The authors also admit that the algorithm is sensitive to noise in the depth map.

The work in [15] builds up on a method similar to ours, in which the authors apply a contour detection algorithm based on [14] and then apply the watershed transform to fill in the segment boundaries formed by the detected contours. These correspond to border detection and connected components labeling, respectively, in our algorithmic flow. While the quality of segmentation is state-of-the-art, the algorithm is not real-time. It is interesting to note that a GPU accelerated version of [14] was worked on in [13]. However, with a processing time of 1.8 seconds per image of size 0.15 megapixels, it is not applicable to a real-time scenario.

Recently, depth information was incorporated by Abramov et al. [4] to improve the segmentation quality, where the depth is captured using a Kinect sensor. The complete flow of the segmentation process executes on a CPU-GPU integrated platform. The simulated annealing process based on the work by Metropolis et al. [1] is the computationally challenging process in this flow, which is the primary target of the parallelization effort. Compared to the related work summarized earlier, Abramov's depth supported segmentation is the leading implementation in terms of throughput and segmentation quality. This work shows that segmentation quality improves by the inclusion of depth information.

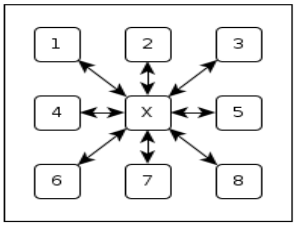


Figure 1. Energy state by pairwise swapping of the center pixel with its 8 neighbor pixels: The swap with the highest energy reduction is accepted.

In this paper, we implemented a video segmentation framework based on Abramov's work. However, we have made the following important changes that can further improve the throughput and segmentation quality:

1) Include depth information in the calculation of the interaction strengths between a pair of pixels;
2) Introduce a scaling factor to amplify the bond between a pair of pixels and to increase the strength of object boundaries; and
3) Replace the Gaussian filter with the more powerful Bilateral filter [19].

III. METROPOLIS PROCEDURE

Our algorithm is based on the description of the approach given in [4]. In [4], the authors propose using the Potts model with the Metropolis method [1] for segmentation. The Potts model is taken from solid state physics and is a way to represent the interactions of particles in a crystalline lattice. In the context of segmentation, each pixel in an image is treated as a particle in a lattice, and assigned a "spin state" value. The spin state of each pixel is thus a representation of the final label that the pixel is assigned to. Each pixel is capable of being in any of the "q" states, where "q" is taken as 256 (i.e. pixels may be in states 0 to 255). This allows the spin state to be represented by a single byte. Note that the value of "q" does not affect the performance and the computation time of the Metropolis kernel itself. However, limiting the value allows us to represent it as a byte and thus reduce the data transfer times between CPU and GPU. The Metropolis algorithm is an energy minimization process for determining the equilibrium states of the particles in a system, or the pixel values in an image in our case. This is done by computing the interaction strength (coupling constants) between a pixel and the surrounding pixels as illustrated in Fig. 1, and then updating the pixel state such that the system energy reduces over time.

$$E = -\sum_{<ij>} J_{ij}\,\delta_{ij} \tag{1}$$

The energy of the system is calculated using (1), where E is the total system energy, J_{ij} is the coupling constant, δ_{ij} is the *Kronecker* delta function. It takes the value of 1 when the

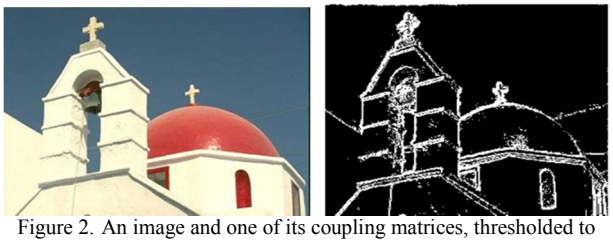


Figure 2. An image and one of its coupling matrices, thresholded to show only the weak (negative) coupling constants in white.

state of pixel *i* is equal to the state of pixel *j*, and 0 otherwise. The indices <*ij*> indicate the region of pixels neighboring pixel *i*, where the distance between pixel *i* and pixel *j* is equal to 1. So the region <*ij*> is simply the pixels immediately adjacent to *i*. J_{ij} is calculated as shown in (2), where g_i and g_j represent the color states of pixels *i* and *j*, and $\bar{\Delta}$ represents the average color difference over all pixel pairs throughout the image.

$$J_{ij} = 1 - \frac{|g_i - g_j|}{\bar{\Delta}} \tag{2}$$

$$J_{ij} = \begin{cases} J_{ij} \; if \; |z_i - z_j| \leqslant \tau \\ \Theta \; otherwise \end{cases} \tag{3}$$

The coupling constant between each pixel pair needs to be computed only once for each frame being processed. Since the sum in (1) is negative, the energy of the system is lowered when particles that are alike in color are in the same state. Identical pixels have a coupling constant of 1. Pixels whose delta is equal to the average delta have a coupling value of 0. Pixels whose delta is greater than the average have negative coupling [5], and thus increase the energy of the system in the summation. Hence the coupling constants are a measure of how strong the bonds are between pixels. For example, pixels lying on the real borders of a segment will have negative coupling constants as illustrated in Fig. 2. With the depth information added, the segmentation can be even more useful. The objects that are labeled together can be properly segmented, especially when objects of similar color are separated in depth. On a single frame, the depth value of a pixel is compared against a set threshold. The coupling constants are changed according to (3), where z_i and z_j are the depth measures of neighboring pixels and Θ is the maximum value of 250 for a coupling constant.

The Metropolis algorithm iteratively operates on each pixel of the image, checking if changing the state of a pixel to that of one of the surrounding pixels would lower the energy state of the system. If one or more switch option is found which will move the system to a lower energy state, the algorithm selects the change that would lower the energy the most. It is worth noting that the energy contributed to the system by a single pixel is only a function of that pixel and its immediately surrounding pixels. So each pixel can be operated on independently, and thus a single metropolis update can be performed nearly simultaneously on the GPU, making it ideal for GPU acceleration. If no state change is found that can lower the energy of the system (i.e. all state switches will cause the energy of the system to increase), the

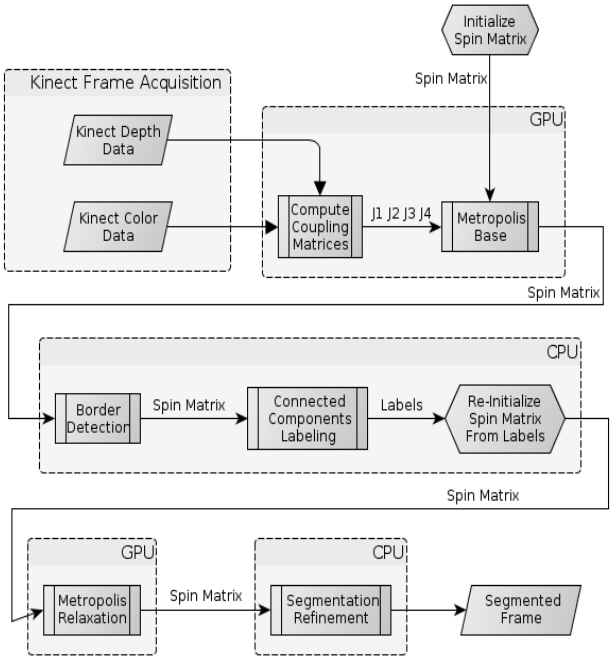


Figure 3. The flow of the proposed segmentation process executing on the CPU-GPU integrated platform.

algorithm selects an unfavorable state switch with the probability calculated by (4), where ΔH is the change in energy before and after the state switch and T_n is the "temperature" at this iteration.

$$P(select\ increase) = \exp\left(-\frac{|\Delta H|}{T_n}\right) \tag{4}$$

The temperature value is a part of the simulated annealing aspect of the Metropolis procedure. Simulated annealing is a preferable method for approximating the global minimum in a large search space that is not manageable though combinatorial methods. The Metropolis method was chosen for its effectiveness in maintaining consistent segmentation labels between frames. Other segmentation methods may be more efficient or effective in some ways but may produce different segmentation results for two slightly different frames. Metropolis routine is executed in two stages in our proposed flow as illustrated in Fig. 3. The purpose of the initial run of Metropolis is to find the "real boundaries" of the objects in a frame. After the initial run of the algorithm, a boundary detection algorithm marks the boundary of objects in the frame.

Fig. 4 shows the state of the execution flow based on Fig. 3. After the first set of the Metropolis iterations are run, real segmented boundaries in the spin matrix have noisy neighborhoods. A boundary detection algorithm is run on this intermediate spin matrix, following the steps in Fig. 3. Next, connected components are labeled and the regions of the spin matrix that coincide with the unlabeled sections of the image (represented as black in Fig. 4) are reinitialized to random values. Another set of Metropolis iterations is run on this spin matrix to give the segmentation that is ready for refinement.

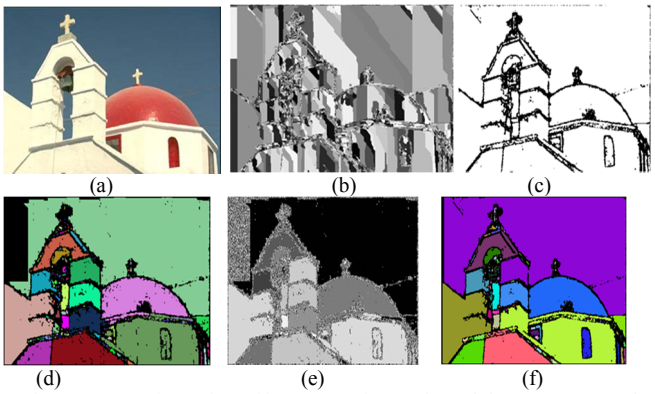


Figure 4. Transformation of image to spin matrix and then to segmented image ready for refinement: (a) Original image (b) Intermediate spin matrix (c) Detected boundaries Connected component labeling and segmentation (e) Reinitialized spin matrix (f) Segmented image after Metropolis II.

IV. GPU IMPLEMENTATION

In keeping with the implementation in [4], for each frame we compute a set of four arrays with the same size as the frame. They separately contain the horizontal, vertical, left-diagonal, and right-diagonal coupling constants. These arrays are computed on the GPU based on the color values of the frame as well as the associated depth data captured by Kinect. Notice that both data have to go through some pre-processing steps before they can be used to calculate coupling constants. Specifically, as RGB and depth cameras are placed at different positions on Kinect, color and depth data need to be first calibrated to be spatially aligned. This is done by utilizing OpenNI API functionality during frame acquisition and registration. We also scale both data dimensions to 320x256 to make efficient use of the threads on the GPU.

Once these four arrays are computed, they are passed to the GPU segmentation kernel, along with the spin matrix, which has the same size as the image and contains the states for all the pixels. Initially the spin matrix is filled with random values. The arrays containing the coupling constants of the four directions are represented as linear arrays of floats. The 2-dimensional data derived from the frame is mapped into a linear array. Normally, linearization of a 2D array is done in the row major format. However, an important aspect of the GPU optimization is to coalesce loads and stores to global memory. This step aligns all the loads and stores for a given thread block, so that all the threads from a warp (typically 32 threads) in a given thread block can access global memory with contiguous memory locations. Thus, we ensure that all threads in a thread block can access global memory in parallel.

For the parallel Metropolis algorithm we rearrange the array indexing such that global loads and stores are coalesced. Coalescing is achieved by encoding the data in the following way as shown in Fig. 5. For example, pixels to be managed by Thread_11 are grouped into one packet of data, in row major order. This is repeated for all subsequent threads and all packets are concatenated to form a new linearized array. To decode the data, the process is reversed. Encoding and decoding are carried out for the four coupling matrices

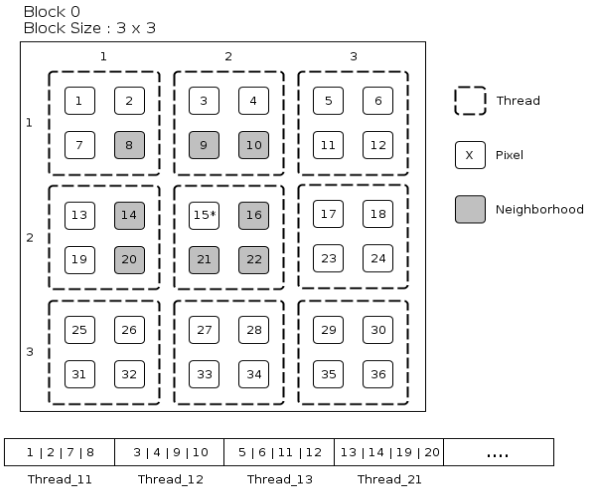


Figure 5. Thread configuration in the case of a 3x3 neighborhood of Pixel 15.

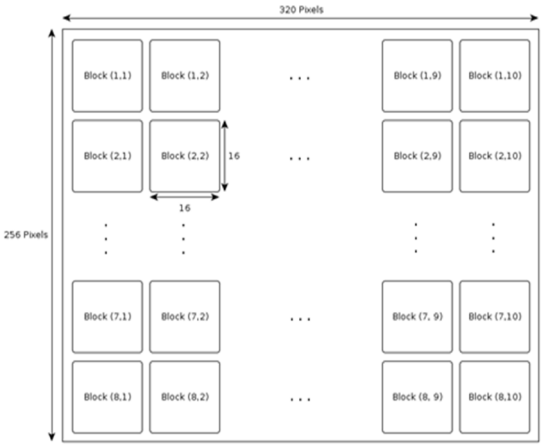


Figure 6. Minimum number of blocks for 320x256 frame. Illustration is based on no overlapping between different blocks

and the spin matrix.

Each thread in a block loads the associated data from the global memory into the shared memory. Furthermore, we arrange thread blocks to have a two-pixel wide overlap between adjacent blocks. With that, when a given thread block updates the inner pixels within the overlap region, its neighboring blocks can update the outer pixels. As illustrated in Fig. 5, each thread updates 4 pixels. In the case of "pixel 15", neighbor 8 is loaded by Thread_11,neighbors 9 and 10 are loaded by Thread_12, neighbors 14 and 20 are loaded by Thread_21, and the rest of the neighbors of 15 are loaded by 15's thread, Thread_22. Threads at the edge of a thread block also load the data for the pixels adjacent to them into the local shared memory.

In our implementation, each thread covers a 2x2 pixel region and operates serially on the four pixels assigned to it.

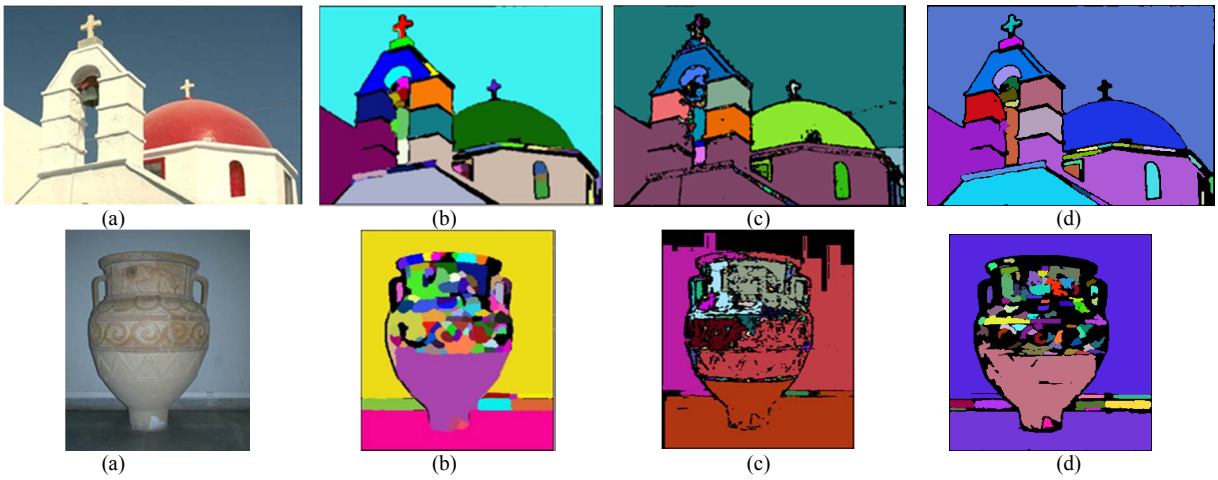


Figure 7. Functionality verification of the CPU-GPU based implementation with two images: (a) source image, (b) reference implementation, (c) our serial implementation, (d) our cpu-gpu implementation. Image for reference implementation was taken from Abramov[4]

All the threads work almost in parallel. The synchronization among thread blocks (for outer overlap pixels) takes place after all the threads finish their processing. Hence for *n* iterations of the Metropolis algorithm, there are *n* synchronizations. We organize 256 threads into a thread block of 16x16, hence the dimension of each thread block is 32x32 in pixels. As each block processes a 32x32 block of pixels, the frame size is set to 320x256 to be multiples of 32. In total, this allows us to launch a minimum of 80 thread blocks, each with 256 threads as illustrated in Fig. 6. However as we allow two-pixel wide columns and rows to be shared between adjacent thread blocks, our true thread block count is 99.

V. TESTBED AND EVALUATION STRATEGY

As a first step, we implemented the serial version of the segmentation flow running on a general purpose processor (Intel Xeon, 3GHz, 2GB RAM) with the OpenCV library. Our objective at this stage was to establish a working flow in the context of still images without depth information. We then partitioned the workload and implemented parallelizable routines in CUDA and implemented on the NVIDIA K40 GPGPU that led to the CPU-GPU based flow illustrated in Fig. 3. We verified both implementations by comparing their output using still images with those from Abramov's implementation, as illustrated in Fig. 7.

After incorporating depth information and introducing the scaling factor into the flow, we implemented two versions with depth-based interaction strength calculation and bilateral filtering. In the experiment setup, the host computer captures video using a Kinect sensor and feeds it into the GPU. A short video clip is recorded that has various objects of different shapes and colors (a white plate, colorful tower blocks and a white kettle with texture) placed in front of a white background. And in the video, someone brings these objects into the scene. We evaluate the performance of each version based on the quality and processing speed in frame rate. The speed is calculated by including the entire process shown in Fig. 3, starting from the data acquisition till the generation of segmented frames.

VI. RESULTS AND ANALYSIS

The final Metropolis iteration speed achieved by our CPU and GPU implementations are 3.199 and 0.013 sec/frame respectively, which is a 241x speed-up. These results demonstrate just how well the Metropolis algorithm is suited for running on the GPU. The fact that thread blocks can operate independently of one another, as well as individual threads within a thread block, is what makes this an ideal solution. Meanwhile, depth supported segmentation runs at 34 fps.

Apart from the Metropolis core, the auxiliary tasks and refinement are performed on the CPU. When the complete flow (Fig. 3) is taken into account, based on the execution times reported in Table 2, our implementation achieved an overall speedup of 60x over the serial only version.

A. Scaling factor for amplifying the interaction strength between the two pixels

The most time-consuming part of the framework is the Metropolis core. An obvious way for throughput improvement is to reduce the number of base and relaxation iterations in the algorithm. We first investigated the impact of the number of Metropolis iterations on the segmentation quality. Fig. 8(a) shows an input image before processing, and Fig. 8(b) illustrates the output with the default numbers of iterations. In Fig. 8(c), we show the output image using only one less iteration in both Metropolis base and relaxation stages. It has many small segments in the scene, with a large portion occupied by boundary pixels (black in color). That implies that the interaction strength for most pixel pairs is low. In fact, by using less number of iterations, we effectively prevent the particles in the Potts model (pixels in our case) from reaching their steady states. This implies that the interaction strength for most pixel pairs is low; hence more pixels form the borders. Therefore, we observe a decrease in

the segmentation quality. In order to reduce the boundary pixel percentage, there is a need to increase the interaction strength between adjacent pixels.

$$J_{ij} = 1 - \alpha \frac{|g_i - g_j|}{\bar{\Delta}} \quad (5)$$

For this purpose, we introduced a scaling factor (α) to (2) and show in (5). By sweeping α value for the baseline (α_{base}) and relaxation ($\alpha_{relaxation}$) stages between 1 and 10 with the increment of, we identified the scaling factors that give the best quality. The default numbers of Metropolis iterations for the "Base" and "Relaxation" are 10 and 25 respectively. When we set the pair of numbers to 5 and 10, we identified α_{base} as 5 and $\alpha_{relaxation}$ as 4 to produce the best segmentation quality, shown in Fig. 8(d). The scaling factors helped us improve the object boundaries, and the throughput increased from 29fps to 44fps. However, it also has the drawbacks of over-segmentation and unstable boundaries. In the following subsections we present two different approaches to overcome them.

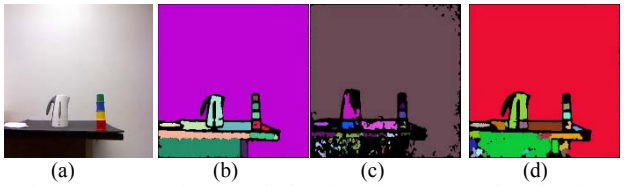
B. *Improve the Segmentation Quality through Depth based Coupling Matrix Computations on the GPU*

We observe that the interaction strengths between neighboring pixels can be more accurately modeled by incorporating depth information.

Recall that the interaction strength between a pair of pixels is defined as a function of their perceived color difference in (2). Although the extended arm in Fig. 9b has a uniform color, owing to difference in lighting, some of the pixel pairs are assigned to a low value of interaction strength. This results with the formation of borders that divide the arm into multiple segments as illustrated in Fig. 9b.

One way to address this issue would be to introduce the concept of "similar colors", realized by adding a threshold (C_T) in the color difference evaluation. Color differences smaller than this threshold will be set to zero, i.e. the two pixels will be marked to have the same color. It should be noted that this scheme assumes the two pixels lie at the same depth. We introduce a second threshold over depth to describe pixel neighborhoods with "similar depth" (D_T). This helps us generalize the approach.

Let ΔC be the difference in color between two pixels and Δ D be the difference in depth between two pixels. In



(a) (b) (c) (d)

Figure 8: Impact of Metropolis iterations for "Base" and "Relaxation" runs on segmentation quality: (a) raw image, (b) default iterations of 10 and 25, (c) iteration reduction by 1, (d) scaling factor based reduction

accordance with the above discussion, we have defined three comparison levels for depth and color as shown in Table 3. Among the nine possible scenarios that combine the different comparison levels for depth and color, we discarded the cases in which at least one of ΔC or ΔD is in the "Different" category, such as different color and same depth, or slightly different color and different depth. We identified four cases that would benefit from the inclusion of depth information for minimizing the color difference, thereby maximizing the interaction strengths of the pixel pairs. Hence we assign them a color difference ($g_i - g_j$) value of zero. These cases are as follows:

- Same depth, same color
- Same depth, slightly different colors
- Slightly different depths, same colors
- Slightly different depths, slightly different colors

TABLE II. EXECUTION TIME (SECONDS)

Task	*Serial*	*CPU-GPU*
Metropolis Core	3.199	0.013
Auxiliary Tasks	0.006	0.006
Refinement	0.035	0.035
Total	3.230	0.054
Speedup		60

Although the thresholds introduced in this approach are dependent on the scene to be segmented, there are some guidelines that help us to arrive at their values. First, the upper limit for the depth threshold is the global depth threshold τ in (3). If the depth threshold exceeds the global depth threshold, any changes made by the proposed algorithm will be over written by (3). Essentially, the new depth threshold grants us a finer control on the usage of depth in segmenting a scene. This is clearly illustrated by the difference between Fig. 9b

a b c

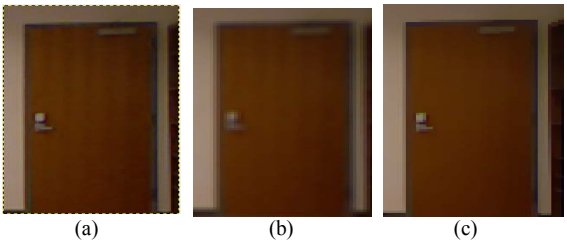
Figure 9: Evaluating the performance of our depth enhanced approach with respect to our CPU-GPU based baseline implementation without using depth information and reference design by Abramov that uses depth: Even indexed columns show raw frames captured during testing. Odd columns show the output frames for: (a) CPU-GPU baseline implementation without depth and with default Metropolis iterations (10,25) operates at 35fps; (b) reference implementation by Abarmov[4] with depth and with its default Metropolis iterations (10,25) operates at 29fps; (c) proposed depth enhanced interaction strength with Metropolis iterations reduced to 5 and 10 for the "Base" and "Relaxation" iterations operates at 34fps.

TABLE III. COLOR AND DEPTH COMPARISON LEVELS

Color Comparison	Definition	Depth Comparison	Definition
Same	$\Delta C=0$	Same	$\Delta D=0$
Slightly different	$0<\Delta C\leq C_T$	Slightly different	$0<\Delta D\leq D_T$
Different	$\Delta C>C_T$	Different	$\Delta D>C_T$

and Fig. 9c, in which the outer rim of the white paper plate is labeled as part of the paper plate when using the newly introduced depth threshold.

To arrive at the color threshold, we select an arbitrarily large color difference (say, 200). What we expect to observe is under-segmentation since pixels that lie at the same or similar depths but of very different colors will be labeled as part of the same object. We then decrease threshold value by ten units at a time until the scene does not face the above issue. The threshold can be further fine-tuned by changing its value one unit at a time. For our work, we arrived at a depth threshold of 25 and a color threshold of 20. By introducing this modification into the calculation of interaction strengths, we prevent over-segmentation of an object while retaining different labels for similarly colored objects at different depths.



(a) (b) (c)

Figure 10: Flattening effect on the shades of same color with different filters: (a) raw image (b) Gaussian filter (c) Bilateral filter.

We illustrate the output of this approach in Fig. 9 while comparing it to the output of the baseline approaches. As can be seen in this figure, using only color information to segment a scene with similarly colored objects in front of each other leads to faulty segmentation (Fig. 9a). The paper plate and the white kettle are labeled the same as the background. Note that the bottom of the kettle is dark gray and hence is given a different label.

Outputs in both Fig. 9a and Fig. 9b are obtained by incorporating depth into the flow. With the depth information available to the algorithm, similarly colored objects can be differentiated even when placed in front of each other. This feature is brought out in the Fig. 9 being discussed, where the white plate is in front of the white kettle and the two white objects are in front of a white background. It is clear that both our method (Fig. 9c) and Abramov's method (Fig. 9b) show the benefit of depth compared to segmentation that doesn't utilize depth (Fig. 9a).

It should be emphasized here that while the outputs in Fig. 9a and Fig. 9b were achieved by using the default number of Metropolis iterations, the output of Fig. 9c was achieved with half of that number. Thus, by incorporating depth into the calculation of interaction strengths, we prevent over-segmentation of an object while retaining different labels for similarly colored objects at different depths. The execution time overhead of the additional process we introduce to the Coupling Matrix Computations on the GPU reduces the throughput from 44 (Section VI A) to 34fps. However, there is a substantial improvement in the segmentation quality and throughput compared to Abramov's method that uses depth in a limited manner.

C. *Improving the Segmentation Quality through Bilateral Filtering*

As mentioned above, fewer Metropolis iterations often results in over-segmentation. And this issue can be exacerbated with imperfect lighting conditions or shades, which create artificial false boundaries even on a flat surface.

In Abravmov's work [4], a Gaussian filter is employed as a preprocessing step to smooth the image before segmentation. That has the effect of damping those false edges. However, we found that the Gaussian filter is not strong enough for that purpose. So we propose using the stronger Bilateral filter to replace it. Through experiment, we chose 15 as the filter sigma value and 5 as its size to be a good combination for quality and filtering speed. Additionally, we experimented using bilateral filtering elsewhere too, but application of bilateral filtering right before Metropolis is the most suitable stage.

We applied the Bilateral filter twice to the original image before it is passed on to the Metropolis algorithm, so that false edges can be mostly eliminated or blurred. Fig. 10 shows a comparison between the outcomes of the two filters. In the raw image (a), we can see vertical striations on the door. Since the striations are a part of the door, it would be desirable if they were labeled the same as the door. Gaussian filtered version (b) has more apparent residues than the Bilateral filtered one (c). Additionally, with the Gaussian filter, boundaries become less prominent. Using a Bilateral filter accomplishes our goal - most of striations are removed and the boundaries are retained from the raw image.

By concatenating the Bilateral filter to the Metropolis segmentation, we effectively reduce the values of $|g_i - g_j|$ in Eq. (2). That in turn increases the value of the coupling constant J_{ij}, thereby strengthening the bonds between the pixels with similar colors.

Fig. 11 presents a real-world segmentation example for the cases where the Bilateral filter and the Gaussian filter are used as pre-processing steps to the core algorithm. In this experiment, the total numbers of "Base" and "Relaxation" Metropolis iterations were set to 5 and 10 respectively. The output with Gaussian filter as a pre-processing step (top row) has the expected artifacts, as described in Section VIA. We observe that the region below the table and the black surface on the table are over-segmented. The Bilateral filter is more effective in removing minute differences in color and thus improves the segmentation quality by preventing over-segmentation. Moreover, since it is an edge preserving

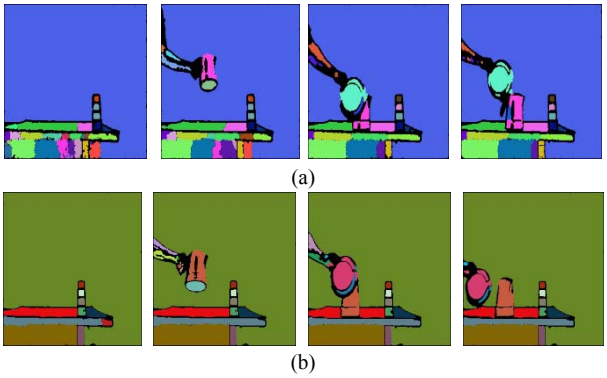


Figure 11: Frames from a video clip shows how the Gaussian (a) and Bilateral (b) filters perform when the number of Metropolis iterations is halved. The frame rates for both cases are 34 fps.

smoothing filter, the boundaries of the segments are sharp and stable.

VII. CONCLUSION AND FUTURE WORK

Our proposed approach incorporates depth in calculating the interaction strengths along with using it to differentiate similarly colored objects. Using depth supported coupling matrix computations, the segments produced are much cleaner even when the number of Metropolis iterations is halved, thus increasing throughput. We also bring out the clear advantage of the depth information when an object with color similar to the background color, enters the scene. In such situations, without the depth information, the object is given a false label that is the same as the background. On the other hand, with the depth information, our implementation clearly identifies the same object and segments it with a different label. Finally, we show how pre-processing the frames to reduce the difference between similar colors also benefits the throughput of the algorithm without sacrificing the quality.

We believe that incorporating depth information into media content processing and analysis is an interesting future research direction, given the amount of additional information depth can bring besides the traditional 2D colors, and the wide-spread availability of depth sensors. Our work presented in the paper tries to contribute to addressing the problem of segmentation with depth, which is one of the fundamental steps in the processing chain. We believe any meaningful advances in this area will greatly benefit many applications such as object recognition, content retrieval, management, etc.

Despite the results presented in the paper, we still think there is potential in this framework and will focus on further improvement. One direction is to investigate the effect of changes in lighting and scene on various parameters. Just as we discuss approaches to arrive at the values of the color threshold (C_T) and depth threshold (D_T), it would be desirable to develop methods for determining the optimal values of other parameters, such as the scaling factor of (5). Another possible direction is to investigate surface modeling with depth assistance to help object identification.

REFERENCES

[1] N. Metropolis, A. W. Rosenbluth, N. Marshall, A. Teller, H. Augusta and E. Teller (1953). "Equation of State Calculations by Fast Computing Machines". The Journal of Chemical Physics 21 (6): 1087. doi:10.1063/1.1699114

[2] D. G. Lowe (1999). "Object recognition from local scale-invariant features". Proceedings of the International Conference on Computer Vision 2. pp. 1150–1157. doi:10.1109/ICCV.1999.790410

[3] K. Zdenek, M. Krystian, and M. Jiri (2010). "Tracking-Learning-Detection". IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 6, no. 1, January 2010

[4] A. Abramov, K. Pauwels, J. Papon, F. Worgotter, B. Dellen, "Depth-supported real-time video segmentation with the Kinect," wacv, pp.457-464, 2012 IEEE Workshop on the Applications of Computer Vision, 2012

[5] C. Eckes, and J. C. Vorbruggen (1996). Combining data-driven and model-based cues for segmentation of video sequences. In World Congress on Neural Networks, pages 868-875.

[6] B. Fulkerson and S. Soatto. "Really quick shift: Image segmentation on a GPU." Trends and Topics in Computer Vision. Springer Berlin Heidelberg, 2012. 350-358.

[7] A. Vedaldi and S. Soatto. "Quick shift and kernel methods for mode seeking." Computer Vision–ECCV 2008. Springer Berlin Heidelberg, 2008. 705-718.

[8] J. Shi and M. Jitendra "Normalized cuts and image segmentation." Pattern Analysis and Machine Intelligence, IEEE Transactions on 22.8 (2000): 888-905.

[9] D. Comaniciu and P. Meer. "Mean shift: A robust approach toward feature space analysis." Pattern Analysis and Machine Intelligence, IEEE Transactions on 24.5 (2002): 603-619.

[10] S. Gould, Stephen, et al. "Multi-class segmentation with relative location prior." International Journal of Computer Vision 80.3 (2008): 300-316.

[11] B. Fulkerson, A. Vedaldi, and S. Soatto. "Class segmentation and object localization with superpixel neighborhoods." Computer Vision, 2009 IEEE 12th International Conference on. IEEE, 2009.

[12] E. Mirante, M. Georgiev, and A. Gotchev. "A fast image segmentation algorithm using color and depth map." 3DTV Conference: The True Vision-Capture, Transmission and Display of 3D Video (3DTV-CON), 2011. IEEE, 2011.

[13] B. Catanzaro, B. Su, N. Sundaram, Y. Lee, M. Murphy, and Kurt Keutzer. "Efficient, high-quality image contour detection." Computer Vision, 2009 IEEE 12th International Conference on. IEEE, 2009.

[14] M. Maire, P. Arbelaez, C. Fowlkes, and J. Malik. "Using contours to detect and localize junctions in natural images." Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on. IEEE, 2008.

[15] P. Arbelaez, M. Maire, C. Fowlkes, and J. Malik. "From contours to regions: An empirical evaluation." Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on. IEEE, 2009.

[16] Y. Zhuge, Y. Cao, K. J. Udupa and R. W. Miller . "Parallel fuzzy connected image segmentation on GPU." Medical physics 38 (2011): 4365.

[17] M. Grundmann, V. Kwatra, H. Mei, and I. Essa. "Efficient hierarchical graph-based video segmentation." Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on. IEEE, 2010.

[18] D. Zhang, O. Javed, and S. Mubarak, "Video Object Segmentation through Spatially Accurate and Temporally Dense Extraction of Primary Object Regions," Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on , vol., no., pp.628,635, 23-28 June 2013 doi: 10.1109/CVPR.2013.87

[19] C. Tomasi, and R. Manduchi. "Bilateral filtering for gray and color images." Computer Vision, 1998. Sixth International Conference on. IEEE, 1998.