Real-Time Video Processing for Multi-Object Chromatic Tracking

Cristina N. Vasconcelos*, Asla Sá+, Lucas Teixeira+,
Paulo Cezar Carvalho$^{\Delta}$ and Marcelo Gattass+
PUC-Rio *, Tecgraf/PUC-Rio +, IMPA $^{\Delta}$
crisnv@inf.puc-rio.br, pcezar@impa.br
{aslasa,lucas,mgattass}@tecgraf.puc-rio.br

Abstract

This paper presents **MOCT**, a **m**ulti-**o**bject **c**hromatic **t**racking technique for real-time natural video processing. Its main step is the MOCT localization algorithm, that performs local data evaluations in order to apply a multiple output parallel reduction operator to the image. The reduction operator is used to localize the positions of the object centroids, to compute the number of pixels occupied by an object and its bounding boxes, and to update object trajectories in image space. The operator is analyzed using three different computation layouts and tested over several reduction factors.

1 Introduction

Object localization is a well studied topic in early vision research. Countless possibilities are opened when such information is extracted in real-time, especially for the development of augmented reality applications.

Motivated by current graphics hardware processing power, Brunner et al. [1] presented a GPU-based technique for finding the location (centroid position and mass) of a single, uniquely colored object in a scene. This technique is attractive for its simplicity, but has serious limitations: besides dealing with a single object, in the composition step it assumes that the object has a square bounding-box (for instance, it does not treat adequately long, thin objects).

In this paper we present **MOCT**, a GPU-based technique for **m**ulti-**o**bject **c**hromatic **t**racking. Our main contribution is a reduction operator that works over the video frames for localizing a set of *n* distinct objects, each with a unique color (in parallel programming, a *reduction* is a computation that produces a smaller stream from a larger one [7, 5, 6]).

The proposed reduction operator is analyzed using three different computation layouts: row, column and tile oriented. We also consider several reduction factors: traditionally, reductions are computed halving each dimension at each step but we conclude that computing time can be reduced by adjusting the reduction factor. The results of this analysis can be used to make more efficient use of GPU resources, by optimizing its memory access pattern.

Additional contributions are: the extension of the reduction operator for computing bounding boxes, thus extending its applicability to objects having non-square bounding

boxes; and the storage of the obtained results into a *trajectory history texture*, allowing the tracking in time of the centroid position and bounding box coordinates for each object. The *trajectory history texture* can be useful for computing movement predictions or producing temporal video effects in GPU.

The paper is structured as follows: In Section 2 we briefly describe the foundations of the parallel programming pattern used and its *General-Purpose Computation on GPU*(GPGPU) version, in order to clarify the structure adopted in MOCT. The processing steps of the MOCT technique are described in Section 3. In Section 4, the efficiency of the MOCT localization algorithm is compared to the technique presented in [1] and also compared against a CPU implementation. The comparison shows that our algorithm is faster than both of them, obtaining real-time rates for a set of several markers. We also show that for the MOCT the ratio between the computing time and the number of objects gets smaller as the number of tracked objects increases. The operator layout analysis is presented in Section 5. Finally conclusions are drawn and future work is discussed in Section 6.

Figure 1: Multi-Object Real-Time Chromatic Tracking of Natural Images: (left) Original Video Frame, (center) Composite and (right) Trajectories

2 Background and Related Work

The MOCT localization algorithm takes advantage of a parallel programing pattern called *reduction operator*. Such parallel programing pattern, also known as semigroup or fan-in operator, is defined by Parhami in [7] as follows: given an associative binary operator $\otimes$, a reduction is simply a pair $(S, \otimes)$, where S is a set of elements on which $\otimes$ is defined. Reduction computation is defined as: given a list of n values $x_0, x_1, ..., x_{n-1}$, compute $x_0 \otimes x_1 \otimes x_2 ... \otimes x_{n-1}$. Common examples for the operator $\otimes$ include +, $\times$, $\vee$, $\wedge$, $\otimes$, $\cup$, $\cap$, max and min.

Parhami [7] shows that a binary-tree architecture is ideally suited for this computation. Each inner node of the binary-tree receives two values from its children, applies the operator to them, and passes the result upward to its parent, and after $O(\lg_2 p)$ steps, the root processor will have the computation result.

The reduction operator pattern characteristics are extremely well suited for graphics hardware architecture as they offer task balancing design across the processors into independent kernels, and are widely used in GPGPU applications in cases where it is required to generate a smaller stream (usually a single element stream) from a larger input stream [3, 6, 5, 8].

Its design attends to GPGPU challenges as each one of its nodes is responsible for computing partial computations, in a manner that can be seen as an independent processing kernel with gather operations on previously computed values, i.e, by reading the corresponding values from an texture where the previous results have been saved. Thus, while a reduction is computed over a set of n data elements in $O(\frac{n}{p} \log n)$ time steps using the parallel GPU hardware (with p elements processed in one time step), it would cost $O(n)$ time steps for a sequential reduction on the CPU [6].

Traditionally, the reduction operator is used over a 2D texture by reducing by one-half in both vertical and horizontal directions. In this case, instead of a binary-tree, its structure is a pyramid representing a tree whose nodes have four children each. The input image corresponds to the pyramid base, which is reduced in multiple passes creating higher levels that correspond to intermediary computations until reaching its root, with 1×1 pixel dimension, corresponding to the final desired result.

The proposed reduction operator used by MOCT is generally classified as a multiple parallel reduction, as it can run many reductions in parallel ($O(log_2 N)$ steps of $O(MN)$ work [4, 2, 6, 5]). This class of reduction operators, despite of its applicability, has not been widely explored yet.

An interesting example of a multiple parallel reduction operator is presented by Fluck et al. [2]. In that work, it is used for computing a histogram over a entire input image or over selected regions of it. The input image is initially subdivided into square tiles. During the processing, each texel within a tile represents a counter for the occurrence of the bin that it represents. Such subdivision prepares the input image for the reduction by computing partial histograms within the area covered by each tile. Finally, the partial histograms are then reduced by multiple halving steps into a single square tile, as proposed in [4].

In the field of chromatic tracking, Brunner et al. [1] use a pyramidal reduction operator on GPU to find the centroid position and pixel mass of a uniquely colored object in a scene. The obtained information is used for overlaying a textured square over the tracked object, creating video composition effects. Their technique starts by creating a binary image mask that indicates whether or not the pixel falls within the object, by comparing each pixel with the target object color in a thresholding procedure. In a second pass, the mask is reduced to a 1×1 image that stores the object mass and its centroid x and y coordinates (scaled by the mass), by means of a multi pass reduction operator that computes the sums of the positions of the pixels in the mask that are considered as belonging to the object. Finally, the data obtained is used for image composition.

Bruner et al.'s algorithm can be used for tracking multiple objects, each with a unique color, but this requires multiple executions of the algorithm, one for each object. Also, the overlaid object does not adjust itself to the shape of the object being tracked. Our algorithm, described in the next section, addresses these shortcomings.

3 MOCT: Multi-Object Chromatic Tracking

This section details MOCT - multi-object chromatic tracking - in GPU. The core of our tracking proposal is a n-object localization procedure via a multiple parallel reduction operator. The routine consists of two steps: a local evaluation (described in section 3.1) and a multiple parallel reduction (in section 3.2). The operator can be extended to objects

having non-square bounding boxes, by means of bounding box extraction (in section 3.3). The thresholding sensitiveness can be reduced by transforming the input video frames color space as shown in section 3.4. Finally, the MOCT routine cycle is completed by storing a trajectory history of object movement as detailed in 3.5.

3.1 Local Evaluation

As in Fluck et al. [2] proposal for histogram computation, before applying the reduction operator we prepare each video frame using local evaluations, producing what we call a *base texture*. The basic idea is to build a texture that contains localization information regarding the objects in the scene. However, since we are tracking multiple objects, a masking texture where texels are in 1-1 correspondence with the pixels of the original image, as the one used in [1], does not work. Instead, following the ideas in [2], our base texture is subdivided into cells of size n, where n is the number of objects being tracked. Each cell contains tracking information concerning the corresponding region in the imput image. More precisely, the i-th texel of a given cell stores information regarding the count and localization of the pixels in the corresponding image region that are identified with object i. We investigate three different layouts for the cells in the base texture (Figure 2): a *vertical layout*, where cells are sets of n consecutive texels on the same column; a *horizontal layout*, where the n texels are on the same row; and a *square layout*, where the cells are squares with sides equal to $\lceil \sqrt{n} \rceil$ (this last one is the arrangement used in [2]). Note that in the last layout some of the texels of the cell may be unused for storing object information. Observe, also, that the size of the base texture may be slightly larger than that of the input image, since each of its dimensions must be an integer multiple of the corresponding cell dimension.

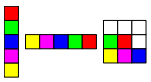

Figure 2: Operator Layouts: Vertical, Horizontal and Square

Whatever the layout chosen, local evaluation is done in a fragment shader that, for each fragment being produced, stores information regarding the occurrence, in the image region associated with the cell to which the fragment belongs, of pixels identified with the object corresponding to the fragment position within the cell. More precisely, the shader counts how many of the input image pixels are considered to belong to the corresponding object and also stores information regarding the position of their centroid. In order to do that, the fragment shader sweeps the region in the input image associated with the current cell, keeping track of the number of pixels classified as belonging to the corresponding object and of the sums of their x and y coordinates in image space. At the end of the local evaluation, the data is saved in the R, G and B channels of the *base texture* (the alpha channel can optionally be used to save the object ID).

Figure 3 illustrates a local evaluation for the vertical layout, showing that the counter indicates an evaluation over the area in the input image associated to the cell and not only over the corresponding pixel (observe the zoomed area). In the figure, positions that are not numbered have zero-valued counters.

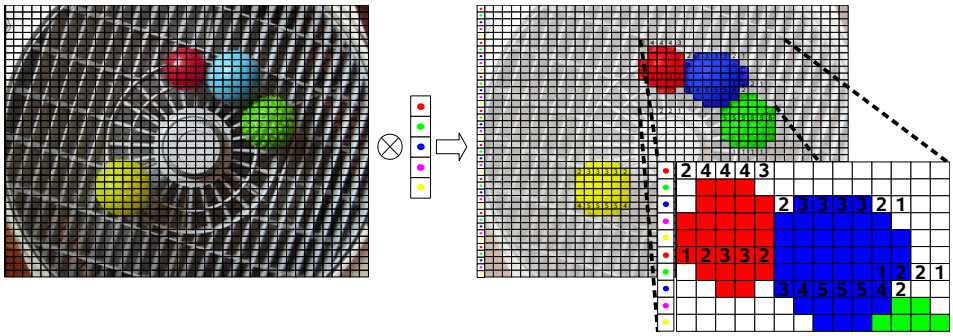

Figure 3: Base Preprocessing Using the Vertical Layout

3.2 Multiple Parallel Reduction

Now that local evaluators have set apart each object data into well defined cells within the *base texture*, the goal of the step presented in this section is to assemble such data from the texture generated into a single storage space for each object. Thus, the goal of this procedure is to reduce the *base texture* into a new texture, by gathering the data corresponding to each object. As in any reduction, each new fragment computed within a level must read the appropriate samples from the previous level and gather their representative data into the newly generated fragment. Usually, reductions are designed in such a way as to group information regarding a set of $2 \times 2 = 4$ texels into a same texel, but in section 5 we discuss different reduction factors.

In our case we produce, at each level, a texture subdivided in cells, according to the layout chosen for the base texture. Each texel in the newly generated texture stores information regarding a specific object, obtained by simply adding the values in each R, G, B channel of the texels in the same position in the cells being aggregated.

When the reduction process is completed, a texture composed of a single cell is produced. Each texel of this cell corresponds to one of the objects and stores the number of pixels belonging to that object and the sums of their x and y coordinates from the input image. Thus, centroid position for each object may be obtained by simply dividing those sums by the number of pixels.

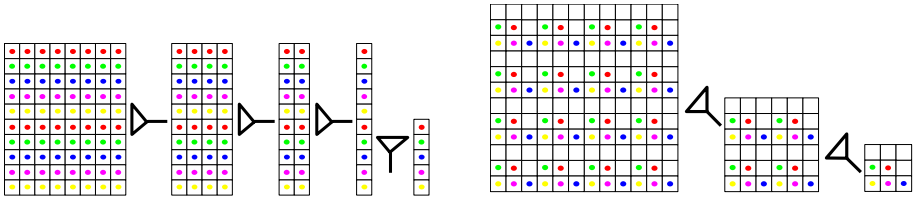

Figure 4: Reduction Processing for (left) Vertical and (right) Square Layouts

Figure 4 illustrates the reduction for the vertical and square layouts. A reduction for the horizontal layout is a trivial variation of the vertical case presented. It is important to notice here that, while in vertical and horizontal layouts the data gathered will be read from neighboring positions, in the square layout case they will be separated by a distance

corresponding to the square side. A more thorough analysis of the layouts efficiency is presented in Section 5.

3.3 Bounding box

The use of the object centroid and mass for video composition suggested by Brunner et al. [1] yields good results for objects having a square bounding-box, such as spheres, but not for objects having other shapes. It is known that minimal bounding boxes can be used to give the approximate location of an object, offering a very simple descriptor of its shape, ideally suited for the composition effects desired.

Our operator can also be used to compute the objects bounding boxes. For that goal, during the *base texture* creation, the pixels classified as belonging to a given object should have their coordinates compared to local minimum and maximum coordinates. Thus, after this computation the *base texture* will contain local bounding boxes over the cells. During the reduction, the gathering is done again by choosing the minimum and maximum coordinates. Then, after gathering all the data contained in the *base texture* into a single cell, each object data will represent the minimum and maximum x and y coordinates, thus, its axis-aligned minimum bounding box.

3.4 Color Space

When colors similar to the objects' colors occur in other parts of the scene, it is difficult to calibrate the colors representative of each object. In some cases, better results can be obtained by working in a different color space. For instance, we may apply a preprocessing step to convert the input image from RGB to HSV, using a simple fragment shader, and then put more weight in the hue information. Figure 5 illustrates a situation, featuring a yellow background with a similar yellow object, where classification using a RGB color space fails, but classification in the HSV space succeeds.

Figure 5: Original Frame (left); When Normalized RGB Goes wrong (center); HSV processing (right)

3.5 Trajectory History

For each video frame processed, after the localization procedure computed the objects centroids localization and bounding boxes, the MOCT processing organizes such data in order to maintain a sequential history of the trajectory of the objects. For that purpose, we create what we call the *trajectory history texture* by using a new fragment shader that

receives the reduced texture and a time counter indicating the frame from which the data was extracted from. Then, the shader updates the trajectory by adding the object data to the next appropriate position in a sequential order, thus fulfilling the *trajectory history texture*. Figure 6 shows an illustration of the configuration for saving the data collected by the vertical layout. When the frame number is larger then the *trajectory history texture* width, the data is saved in a new row of texels as shown in figure 6 (center)).

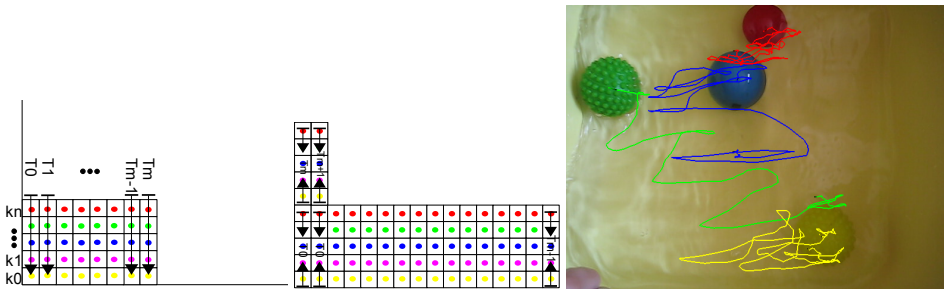

Figure 6: Trajectory History Textures (left and center); Trajectory Drawing (right)

4 Results

This sections presents the timing results comparing the MOCT localization procedure with an CPU implementation and also with the multi-pass Brunner's et al. algorithm ([1]), executed once for each object. The tests were performed using a Intel Core 2 Duo processor E6550 2.33Ghz with 2GB of RAM memory processor and a NVidia GeForce 8800 GTX (768MB) graphics card. The tests with more than 32 objects were done using synthetic images instead of natural ones.

As expected, the CPU implementation is slower than both of the others (left in Figure 7). It is interesting to note that while multi-pass Bruner's et al. algorithm for a set of objects presents constant ratio between the computing time and the number of objects, in our solution such ratio goes down, as the number of objects increases. The only configuration for which the extension of Brunner's et al. presents better performance than MOCT is the case of a single object. In that case the small difference observed in performance is associated with overhead incurred with layout positioning computations. In all cases where more than one object is localized, MOCT achieved significantly better performance. The results are quite expected, as the MOCT localization procedure was developed for an optimized texture access over the graphics card cache policy as detailed in next section. Those results demonstrate the MOCT applicability to real time tracking applications, as it achieves from 500 fps to 40 fps on tests using 1 to 128 objects, respectively.

5 Operator Layout Analysis

The present analysis is inspired by two observations: the number of texture access decreases when the proportion of the reduction increases, and, graphics card texture accesses are faster if the texture was already cached in the corresponding processor.

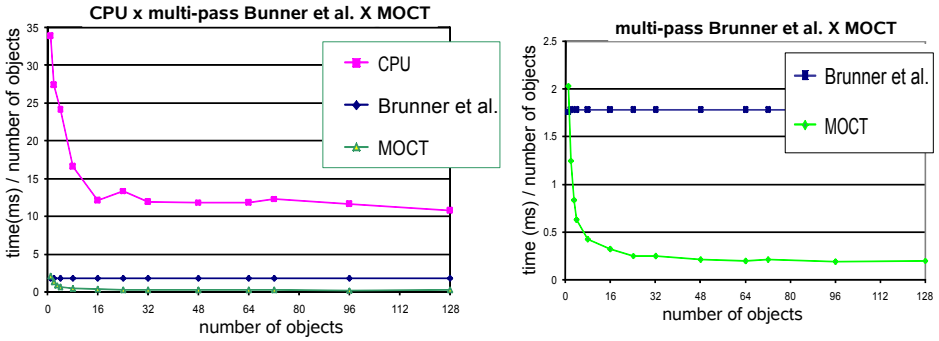

Figure 7: Comparison between CPU, multi-pass Brunner et al. and MOCT

Observe that the number of texture accesses of a reduction operator from a p-pixel input, until reaching the n-sized desired output with a reduction factor of r, can be described by the sum:

$$p + p/r + p/r^2 \ldots + nr^3 + nr^2 + nr \leq nr^i + nr^{i-1} + nr^{i-2} + \ldots + nr^3 + nr^2 + nr \qquad (1)$$

where

$$i = min\left\{k \mid p \leq nr^k\right\} \qquad (2)$$

This means that the higher the reduction factor used, the smaller the total number of texture samples read. In the limiting case, a single step is used during the reduction, making the total number of texture accesses equal to $n * r$, i. e., each sample is read only once. That configuration may not be the better performance result for the parallel computation as it can leave some processors without any work (as long as the number of processors is larger than the number of objects, n) but it shows that increasing the reduction rates induces a smaller number of multiprocessing steps and fewer texture sample readings.

As a second performance analysis factor, we test the three different proposed layouts. Even though graphics card cache memory policy is not open, it is known that memory access pattern does interfere in algorithm efficiency.

The graphs in figure 8 show how processing time changes as a function of the reduction rate, for each type of layout. We observe that better performance is obtained using the vertical and horizontal layouts instead of the square layout, what can be explained by the memory access localization principle. During the reducing, the texture samples read on vertical or horizontal layouts are neighboring texture samples that have a higher chance of representing a cache hit according to a cache policy that corresponds to caching a texture region, while for the square layout, corresponding objects are spaced by the square side, thus increasing the chance of a cache miss.

The different reduction scales and computation layouts presented in figure 8 provide a basis for an attempt of reducing texture already cached waste while maintaining the reduction computation distribution over the multi-processors. The graphs show that the best configuration for the MOCT efficiency was obtained using the vertical layout with a reduction rate of 40, i. e., 40 samples are read per reduction processing kernel.

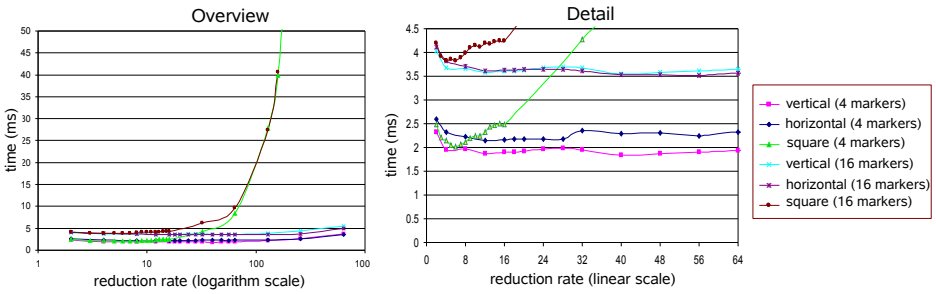

Figure 8: Layout and Reduction Factor Analysis: (right) overview, (left) detail

We observe that the results presented in this section can be extended to other reduction operators such as the histogram computation presented by Fluck et al. [2]. Even though the timing numbers shown here reflect the specific graphics card used they show that reductions other than the usual (simply halving the sizes) should be considered and confirm our assumption that understanding texture samples access patterns is essential to the development of an efficient reduction operator.

6 Conclusion

We presented MOCT, a technique under the *General-Purpose Computation on GPU* paradigm that defines procedures for tracking a set of objects identified by their colors from natural videos. It is composed by a localization procedure and a gathering step for collecting the objects' trajectory data.

The MOCT localization algorithm can be generally classified as a Multiple Parallel Reduction, whose goal is to find object centroids, mass and bounding boxes (allowing its applicability to objects having non-square bounding boxes).

As shown by the timing results, the MOCT localization algorithm is faster than a CPU localization procedure and also faster than applying several times, one for each target object, the technique proposed by Brunner et al. [1], originally devised to track a single object.

As an additional contribution, we compare three different layouts for the reduction operator and several reduction factors. We have shown how those choices can affect the overall efficiency of the reduction operators as they can be used for optimizing the number of texture samples readings, multi-processors occupancy and texture sample access patterns.

In summary, we conclude that the MOCT technique is well suitable for applications requiring the identification and localization of a set of uniquely colored objects at real-time rates.

References

[1] Ralph Brunner, Frank Doepke, and Bunny Laden. Object detection by color: Using the gpu for real-time video image processing. In Hubert Nguyen, editor, *GPU Gems*

3, chapter 26, pages 563–574. Addison Wesley, July 2007.

[2] Oliver Fluck, Shmuel Aharon, Daniel Cremers, and Mikael Rousson. Gpu histogram computation. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Research posters*, page 53, New York, NY, USA, 2006. ACM.

[3] Daniel Horn. *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Stream reduction operations for GPGPU applications, pages 573–589. Addison Wesley, 2005.

[4] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 908–916, New York, NY, USA, 2003. ACM.

[5] John Owens. Data-parallel algorithms and data structures. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 3. ACM, 2007.

[6] John Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[7] Behrooz Parhami. *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, 1999.

[8] David Roger, Ulf Assarsson, and Nicolas Holzschuch. Efficient stream reduction on the gpu. In David Kaeli and Miriam Leeser, editors, *Workshop on General Purpose Processing on Graphics Processing Units*, oct 2007.