GPU Accelerated Multimodal Background Subtraction

Peter Carr

RSISE
Australian National University
Canberra, ACT, Australia
peter.carr@anu.edu.au

NICTA
Canberra, ACT, Australia

Abstract

Although trivial background subtraction algorithms (such as temporal averaging) can execute quite quickly, they do not give useful results in most situations. More complex algorithms usually provide better results, but are typically too slow for widespread use. Here, we examine the architecture of the GPU and describe how a multimodal background subtraction algorithm can be implemented on graphics hardware to provide useful results in real-time.

1. Introduction

Background subtraction provides a quick way to identify foreground objects in an image. The difference between any image and its corresponding background image must be due to foreground objects. More generally, background subtraction is binary classification. A background appearance model is assembled by observing the temporal history of each pixel. The information is then used to determine whether pixels acquired at some future point in time can be explained by the model or not—i.e., segmenting the image into foreground and background pixels.

The major difficulty of background subtraction is, of course, estimating the background. In most scenarios, the background is not static, and dynamic models that allow objects to transition from foreground to background and vice-versa are needed. Traffic surveillance, for instance, must be able to handle stationary vehicles correctly. Vehicles parked on the side of the road should be considered background, and those temporarily stopped at traffic lights should continue to be classified as foreground objects, even though they are not moving.

1.1. Graphics Processing Units

Graphics hardware (described in more detail in Section 3) is specifically optimized for rendering images, and is able to achieve high performance by executing independent calculations across parallel hardware simultaneously [7]. The environment is very different from the more common CPU architecture, and shares many similarities with stream programming [6]. Algorithms which adhere to this paradigm have a good chance of achieving dramatic performance increases on the GPU relative to their CPU equivalent. However, they must be able to work within the limitations of the actual hardware (such as local memory). Details of a GPU's inner workings are proprietary, making it difficult to optimize an implementation. Instead, we employ a restricted GPU programming framework, which automatically applies established compiler optimization techniques to the code, and minimizes reliance on temporary memory. As a result, the challenge investigated here is to find a robust background subtraction algorithm which can be implemented within this reduced GPU instruction set.

1.2. Background Subtraction Algorithms

Elementary background subtraction uses an *a priori* image of the empty scene, and then computes the difference between this and each new video frame. Although easy to construct, this scheme is not useful in practice, as it does not dynamically update. Moreover, it requires controlled circumstances to capture an image of just the background. Temporal averaging allows the *a priori* image to be updated dynamically, but this method often produces ghost artefacts around moving objects (see Figure 1). Stauffer and Grimson were the first to demonstrate how a Gaussian mixture model could be employed for robust background sementation of a dynamic scene [8]. Furthermore, the memory and computational requirements of this parametric method are much lower than the eqivalent kernel density estimation ap-

proach [2].

In this work, we use Stauffer's and Grimson's Gaussian mixture model [8] to approximate the temporal history of a pixel and describe how a parallel implementation can be realized on graphics hardware.

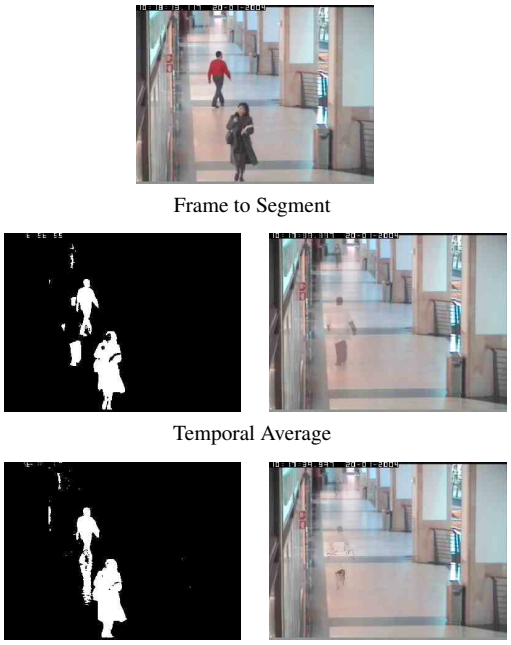

Figure 1. *The background of the scene is estimated using temporal averaging (middle row) and a three mode Gaussian mixture model (bottom row). The segmentation of the input image (top row) is shown in the left column for each model. The corresponding background estimate is shown in the right column. Whereas temporal averaging produces a ghost artefact of the person present during initialization, pixels in the GMM have either re-initialized to the true background or are slowly converging over time.*

1.3. Related Work

A GPU version of Stauffer's and Grimson's algorithm was first implemented by Lee and Jeong in 2006 using NVIDIA's *Cg* library [5]. Few technical details were divulged in the publication, so it is difficult to determine exactly what was done. The work investigated the specific case of three modes, and appears to have made slight deviations from the original algorithm. For instance, Stauffer and Grimson specify the modes to be stored in sequential order. When a new mode is discovered, the last (least probably) background mode is discarded. Conversely, Lee and Jeong elect to store the modes unordered and resolve any necessary sorting conditions (such as finding the least probable mode) only when necessary. Although this can improve the speed of the implementation, it also requires approximating other portions of the algorithm. When determining if a matched mode is transient or not (see Section 2.2), Stauffer and Grimson base their decision on cumulative temporal history, whereas Lee and Jeong only check the individual duration of each mode. Such an approximation may break-down in scenes where repetitive background modes are common.

2. Stauffer's and Grimson's Algorithm

The expressions used in Stauffer's and Grimson's algorithm are identical for all pixels in the image and do not contain any inter-pixel dependencies. For clarity, the following review will omit the row and column subscripts from all per-pixel symbols. We also assume a greyscale image and address the necessary modifications for colour images in Section 4.

The Gaussian mixture model stores K normal distributions for each pixel (parameterized by μ_k and σ_k, where k = 1, 2, ..., K) with K typically between 3 and 5 (depending on the complexity of the background). An additional weighting parameter w_k is associated with each mode, and represents the frequency with which this mode has occurred in the past. The modes are stored in descending order, with the likelihood, L_k, determined by the mode's proportion of temporal history and estimated variation.

$$L_k = \frac{w_k}{\sigma_k} \tag{1}$$

2.1. Matching

At time t, a new image is acquired and each pixel is assigned to one of the K modes associated with the same image location. The association, $M_t \in [0, K]$, is performed by traversing the list of modes until the Mahalanobis distance between the pixel X_t and the k^{th} mode is less than a pre-determined tolerance D:

$$M_t = \arg\min_k \frac{|X_t - \mu_{k,t-1}|}{\sigma_{k,t-1}} \leq D \tag{2}$$

In their implementation, Stauffer and Grimson use $D = 2.5$ and note that "this threshold can be perturbed with little effect on performance" [8]. If no mode is sufficiently close to X_t, then $M_t = 0$.

2.2. Segmenting

If the pixel X_t can not be explained by any of the existing background models (i.e., $M_t = 0$), it is classified as a foreground pixel. However, even if it is matched to a mode, further analysis needs to be conducted to ensure X_t has not been matched to a transient mode.

In a dynamic scene, some foreground objects may become stationary, and should be incorporated into the background model. During this transient period, the pixel X_t (now representing the appearance of the stationary foreground object) should continue to be classified as foreground. However, once a suitable amount of time has passed, it should be classified as background. Stauffer and Grimson incorporate a user-defined threshold, T, which determines how much temporal history is attributed to the background. Therefore, the first B_{t-1} modes are classified as background, and the others transient.

$$B_{t-1} = \arg\min_{b} \sum_{i=0}^{b} w_{i,t-1} > T \tag{3}$$

As a result, the classification rule becomes:

$$S_t = \begin{cases} \text{background} & \text{if } 0 < M_t \leq B_{t-1} \\ \text{foreground} & \text{otherwise.} \end{cases} \tag{4}$$

2.3. Updating

If the pixel is not matched to any of the modes, the last mode in the list (which is the least probable) is re-initialized to the observed value, along with a "high variance and low prior weight" [8]. Re-initialization allows the algorithm to adapt to rapid changes in the scene.

On the other hand, if the observed pixel is matched to a mode, then the information is incorporated into the model. This allows the algorithm to adjust to subtle changes in the scene appearance, such as outdoor lighting conditions, as well as refining the estimated variance.

Given $t-1$ unknown observations $\{X_1, ..., X_{t-1}\}$, the revised estimates of the mode parameters after observing and matching X_t become:

$$\mu_{M_t,t} = \frac{t-1}{t}\mu_{M_t,t-1} + \frac{1}{t}X_t \tag{5}$$

$$\sigma^2_{M_t,t} = \frac{t-1}{t}\sigma^2_{M_t,t-1} + \frac{t-1}{t^2}|X_t - \mu_{M_t,t-1}|^2 \tag{6}$$

The above expressions assume the underlying probability conditions for generating X_t do not change. As a result, the estimated values of $\mu_{M_t,t}$ and $\sigma_{M_t,t}$ are asymptotic as $t \to \infty$. Since a background model needs to be able to adapt to slow temporal changes, Stauffer and Grimson instead use a fixed temporal history window (determined by the *learning rate*, α). Each data point is assigned a weight, ρ, which is a combination of the learning rate and a modifier to reduce the influence of outliers. Stauffer and Grimson employed probability, whereas we use the likelihood ratio (relative to $\mu_{M_t,t-1}$):

$$\rho = \alpha\, L\left(X_t | \mu_{M_t,t-1}, \sigma_{M_t,t-1}\right) = \alpha \exp\left(-\frac{|X_t - \mu_{M_t,t-1}|^2}{2\sigma^2_{M_t,t-1}}\right) \tag{7}$$

The probability expression is not bounded in magnitude and has an additional dependency on $\sigma_{M_t,t-1}$. As a result, the rate at which new information is incorporated into a mode is dependent on its variance. The likelihood ratio, on the other hand, is bounded in [0,1] and only depends on the Mahalabonis distance. As a result, the revised update equations become:

$$\mu_{M_t,t} = (1-\rho)\mu_{M_t,t-1} + \rho X_t \tag{8}$$

$$\sigma^2_{M_t,t} = (1-\rho)\sigma^2_{M_t,t-1} + \rho|X_t - \mu_{M_t,t-1}|^2 \tag{9}$$

Once the Gaussian parameters of a particular mode have been updated, the weights of all modes must be recalculated. Stauffer and Grimson use a linear interpolation between the previous and current weight distributions, scaled by the learning rate, α:

$$w_{k,t} = (1-\alpha)w_{k,t-1} + \alpha\delta(k - M_t) \tag{10}$$

After interpolating, the weights are normalized so that they sum to one.

2.4. Sorting

The values of $u_{k,t}$, $\sigma_{k,t}$ and $w_{k,t}$ have changed since time $t-1$, so the corresponding likelihood of each mode (1) must be re-calculated. There is a good chance that the list of modes will have to be re-arranged to ensure that the modes are ordered from most likely to least likely.

3. Graphics Processing Units

GPUs are built for the specific task of rendering a 2D image from a collection of geometric, lighting and appearance information. The process is performed in a pipeline architecture, allowing the multiple stages to operate simultaneously (see Figure 2). Here, we present a generalized overview of graphics hardware, and refer the reader to [3, 7] for more complete descriptions.

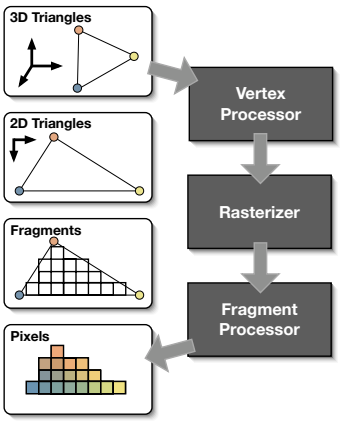

Figure 2. *The graphics pipeline transforms geometric, lighting and appearance information into an image. The vertex processor maps 3D scene geometry into 2D screen co-ordinates. The rasterizer converts these into discrete fragments, which are then transferred to the fragment processor, and eventually become the pixels of the output image.*

The GPU is a self-contained unit within the computer and has its own dedicated video memory. Direct access to the underlying graphics hardware, such as obtaining a pointer to a location in memory, is not permitted—at least, in in most APIs. Instead, one must interact with the GPU using OpenGL, and explicitly transfer data between video memory and regular system memory (see Figure 3). Modern GPUs have programmable vertex and fragment processors, allowing the programmer to transfer "shader programs" to the GPU, in addition to the traditional information such as geometric and image ("texture") data.

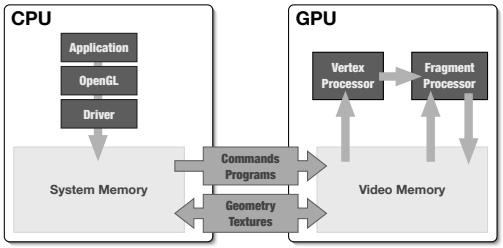

Figure 3. *Access to the GPU is controlled by the OpenGL API, as well as proprietary drivers. Image and geometric data can be transferred between the two memory units. Modern GPUs allow vertex and fragment programs to be transferred as well.*

Since the stages of the graphics pipeline are chained together, and the vertices and fragments are processed individually, the concept of a GPU is similar to that of a *stream processor* [6]. In the stream programming paradigm, the data is organized into sequences of individual elements (all of the same type) known as *streams*. The functionality of the program is achieved using a chain of *kernels*. Each kernel produces one or more output data streams by consuming one or more input streams. Kernels, however, are not arbitrary stream functions. They must ensure that "computations on one stream element are never dependent on computations on another element" [6]. This restriction allows the program to be implemented on parallel hardware quite easily, since each process is independent. The constraint also means most kernels implement one of the following generic operations:

map A single output stream is produced by applying a function to each element of an input stream.

scatter An input stream is used to produce multiple output streams.

gather Multiple input streams are reduced to a single output stream.

filter A single input stream is used to produce a single output stream with potentially fewer elements.

Although GPUs are similar to stream processors on a conceptual level, the actual operation is quite different. Fragment shader programs are only able to write to a single memory location (determined by the rasterizer [7]). As a result, scatter-type operations are not possible in fragment shader programs. On graphics hardware, pixels are an encapsulation of four values (three for colour, one for transparency), and the fragment processors are 'single instruction multiple data' (SIMD) units—meaning the same operation is applied to the four elements of a pixel simultaneously (discussed further in Section 3.2). Support for branching and looping statements is limited [7]. Unless every element of the vector evaluates to the same condition, the system falls back on 'predication': both branches are evaluated and only the results from one branch are kept depending on how the branching condition was evaluated for each element. Finally, the fragment units have a limited number of registers to store temporary results during execution. However, a complex task can be split into multiple passes by writing the partial results to global memory, and then reading these back at the next iteration.

3.1. Core Image

Apple's *Core Image* framework [1] is designed to simplify the task of accelerating image processing operations

using the GPU. Similar to NVIDIA's *Cg* or Microsoft's *HLSL*, Apple's framework abstracts the GPU environment so that fragment shader programs can be written in a more familiar C-style syntax.

However, *Core Image* also provides a high-level interface to combine multiple fragment shaders into a single program (much like AMD's *Ashli* [7]). The framework applies established compiler optimization techniques to the resulting code, and minimizes the memory requirements of both shader programs and temporary storage between iterations [1]. Finally, *Core Image* transparently handles data transfer between the CPU and the GPU, and automatically manages its use of available memory on the graphics card.

Unlike the other APIs, *Core Image* stays very close to the stream programming paradigm. Shader programs are referred to as kernels, and a program is implemented by chaining multiple kernels together. Since *Core Image* only provides access to the programmable fragment processor, its kernels are limited to producing a single output stream.

3.2. Core Image Kernel Language

The Core Image Kernel Language (CIKL) is a subset of the OpenGL shading language [4], and does not support complex functions and data types (such as matrices and arrays). Dynamic operations are also quite limited. For instance, elements within a vector can only be accessed by constant expressions. Similarly, conditional expressions which can not be evaluated at compile time are also not allowed—essentially making `if`, `for` and `while` statements unavailable. However, the inline `if` ternary operator `( ? : )` is permitted and is able to evaluate run-time expressions.

SIMD commands perform the same operation on one or more sets of data (depending on the number of input parameters). In CIKL, a set of data is a float vector of one to four elements. For example, the following code will compute the element-wise maximum between two four-vectors `x` and `y`:

```
vec4 x = vec4(1.0,2.0,3.0,4.0);
vec4 y = vec4(7.0,5.0,3.0,1.0);
vec4 z = max(x,y);
  // z = {7.0,5.0,3.0,4.0}
```

The maximum element in a four-vector can be realized by cascading multiple one-vector SIMD operations.

```
float z = max(max(x[0],x[1]),
              max(x[2],x[3]));
   // z = 4.0
```

The forthcoming section describing how multimodal background subtraction can be implemeted within *Core Image* will make extensive use of the `max()` and `min()` operations. The following functions will also be employed:

`equal(x,y)` Produces a boolean vector containing the result of the element-wise equality operator. We will also employ its variants, such as `lessThan(x,y)`.

`dot(x,y)` The dot product of `x` and `y`.

4. Implementation

Stauffer and Grimson extended their framework to colour images by assuming that the variances of red, green and blue values within a colour image are the same. As a result, a colour mode is represented as four floating point measurements: $\mu_{\text{red}}, \mu_{\text{green}}, \mu_{\text{blue}}, \sigma$. Since GPU pixels consist of four elements, the alpha channel (which stores transparency) can be used for σ, as background pixels are always opaque. Unfortunately, there is no remaining space in the pixel buffer to store w_k (the weight assigned to that mode). Instead, additional pixel buffers must be allocated to store the mode weights (up to four can be stored in a single pixel buffer).

Our prototype system models the temporal history of the scene using three modes per pixel. As a result, four buffers are used to store the statistical characteristics of the three modes. Although a four mode implementation would be more memory efficient (in terms of allocated bytes per mode), the added algorithmic complexity severely hinders performance, and doesn't necessarily increase the usefulness of the results for our applications.

For computational efficiency, the variance of the mode (and not its standard deviation) is stored in the alpha channel. Furthermore, any distance based calculations, such as (2), are implemented using the square of the expression, as this avoids calculating square roots (a relatively slow operation).

Since *Core Image* automatically optimizes the cumulative result from a chain of kernels, this discussion will only cover how each stage of the algorithm is implemented as one or more kernels.

4.1. Matching

The matching stage is responsible for producing M_t, the index of the mode which best explains X_t (if any). Since *Core Image* does not support dynamic indexing, we encoded M_t as an indicator vector, so that the equivalent of dynamic indexing can be achieved using the `dot()` product of M_t and an arbitrary vector.

Unfortunately, the matching task does not simply reduce to implementing a parallel version of (2), as the recently acquired pixel X_t can only be matched to at most one mode. However, since the modes are stored in sorted order, the better of any two matches will occur towards the beginning of the vector [8]. One possible approach is to chain

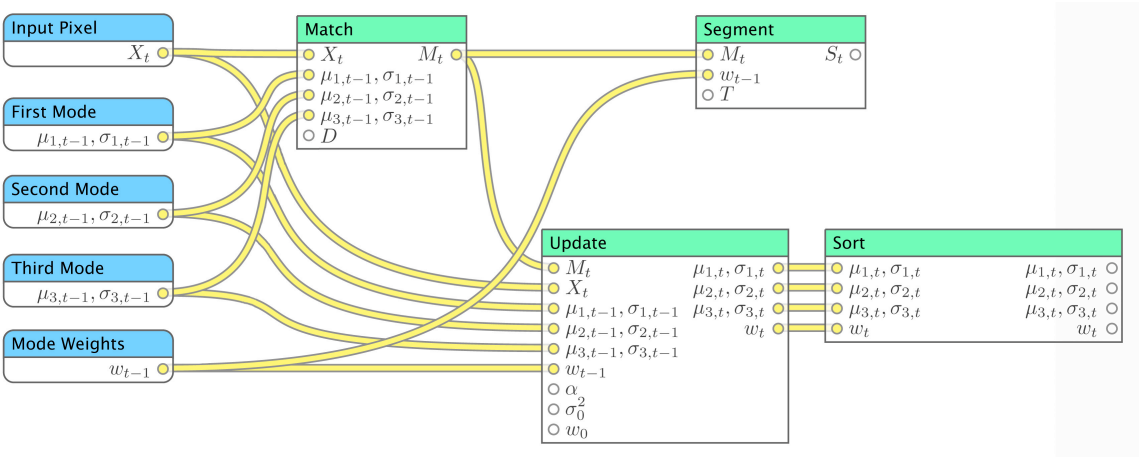

Figure 4. *Stauffer's and Grimson's algorithm expressed in the stream programming paradigm. Each mode is stored in a pixel buffer (identified using rounded corners), with the weights of the three modes stored in an additional pixel buffer. The algorithm is implemented using multiple kernels (identified using square corners), with inputs listed on the left, and outputs on the right (details of the multi-kernel "update" and "sort" routines are shown in Figures 5 and 6). Core Image automatically generates the five GPU fragment shader programs to produce the output segmentation mask S_t, as well as the updated mode parameters (which are copied back into the pixel buffers for the next iteration). The program is executed in parallel for all pixels in the image.*

ternary operators together and test the suitability of each mode sequentially. Instead, we can calculate the distance to all modes simultaneously, and then apply data-dependent masking operations to ensure that the first `true` value is the only occurrence within the indicator vector version of M_t (and avoid the overhead of branching):

```
vec3 M = lessThanEqual(dist,D);
M[1] *= (1.0-M[0]);
M[2] *= (1.0-M[0]-M[1]);
```

4.2. Segmenting

Once a match has been found, the mode's status (either background or transient) must be known for proper segmentation. To avoid the dynamic operations required by (3), we compute a vector, Y, indicating whether the k^{th} mode is non-transient. The `dot()` product of Y and M_t will determine the segmentation (4).

In the specific case of three modes, there are three possible configurations of Y, determined by (3).

Condition	Y
$0 \le T < w_1$	[1 0 0]
$w_1 \le T < w_1 + w_2$	[1 1 0]
$w_1 + w_2 \le T < 1$	[1 1 1]

Only two expressions need to be evaluated, since the first element of Y is always `true`.

```
vec2 cond;
cond[0] = weight[0];
cond[1] = weight[0] + weight[1];
vec3 Y = vec3(1.0,lessThan(cond,T));
float seg = 1.0-dot(M,Y);
```

4.3. Updating

The matched mode is updated using (7) through (9). If a suitable match was found, the weights of all modes are revised using (10). Here, there are no additional restrictions to enforce beyond the mathematical expressions, so the equations can be executed simultaneously.

The three kernels used for updating the μ and σ^2 parameters of each mode (see Figure 5) are quite similar. If the k^{th} element of the matching vector M_t is equal to one, the values of $\mu_{k,t}$ and $\sigma^2_{k,t}$ are updated. However, if the pixel was not matched to this mode, the values are returned unchanged. The kernel for the third mode contains one additional complication: before returning the unaltered parameters $\mu_{3,t}$ and $\sigma_{3,t}$, the matching vector M_t is examined to see if all elements are zero. If that is the case, $\mu_{3,t}$ is initialized to the pixel value X_t, and $\sigma^2_{3,t}$ assigned a user-defined initial variance σ_0^2.

The kernel to update the mode weights must also test for all elements of M_t being zero. If this is the case, the values of $w_{1,t}$ and $w_{2,t}$ are unchanged, and $w_{3,t}$ is assigned

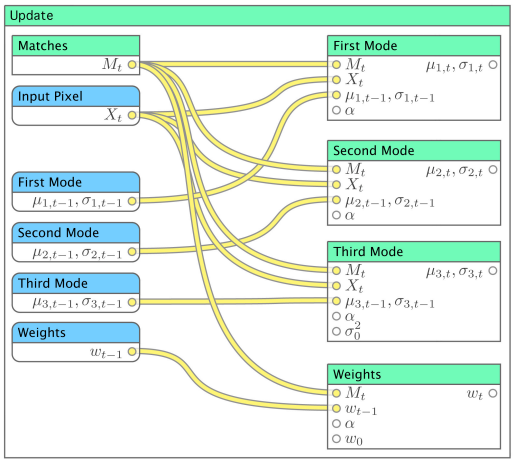

Figure 5. *A streamed version of the "Update" stage. Subject to a branch condition involving $M_{k,t}$, the "n^{th} mode" kernels implement (8) & (9), and the "weights" kernel implements (10).*

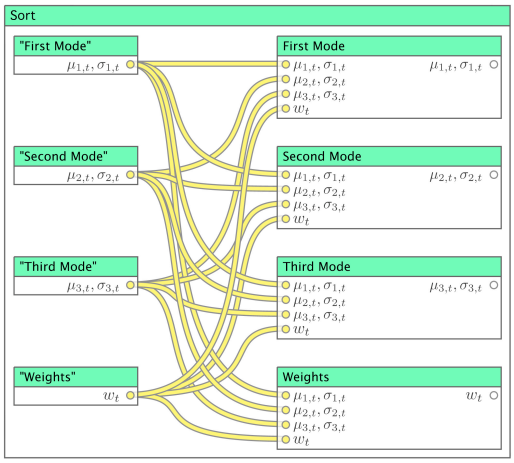

Figure 6. *The unarranged results from the "update" kernel are sorted using multiple n^{th} element kernels. The likelihood (1) is calculated in each kernel, as it determines the sorting order.*

a user-defined initial weight w_0. Otherwise, the weights are updated using (10). In either case, the values of $w_{k,t}$ are then scaled to sum to one.

4.4. Sorting

Determining the largest and smallest elements in a vector is relatively straightforward, and can be implemented by cascading multiple `max()` and `min()` statements across the individual elements (see Section 3.2). In the specific case of a three vector, the sorting task is complete after this stage—the remaining element (which is neither the largest nor smallest) must occupy the middle position when sorted.

As there is no guarantee for unique elements, the `equal()` function may not produce a boolean vector with only one `true` element. However, a boolean vector indicating the location of the element after a stable sort can be achieved by applying data-dependent masks to the result of `equal()`. The mask for the middle element is computed directly from the masks for the largest and smallest elements.

Our sorting stage (see Figure 6) is implemented using three "$n^{\text{th}}element$" kernels. Each calculates L, the vector of mode likelihoods (1), and determines the location indicator vector, I, for the hard-coded value of $n = \{1, 2, 3\}$. The appropriate input is returned using a chain of ternary operators.

```
vec3 I = equal(L,max(L[0],max(L[1],L[2])));
I[1] *= (1.0-I[0]);
I[2] *= (1.0-I[0]-I[1]);
return I[0] ? mode0 :
       I[1] ? mode1 : mode2;
```

4.5. Initialization

Stauffer's and Grimson's publication reveals few details on how the modes were initialized. In this implementation we use the first three video frames to initialize the three modes. Each mode is assigned the user-defined initial variance and a weight of one-third. Alternatively, each mode could be initialized using a random point in the colour space. The drawback of the latter method is that if the random point happens to be close to an actual mode, the model parameters will take a significantly long time to converge, since the influence of each data point is weighted by both its likelihood and the global learning rate parameter α.

The approximate expressions (8) and (9) are only valid once $\alpha \approx \frac{1}{t}$. Therefore, our implementation tracks the total number of frames processed, t, and employs (5) and (6) until $t > \frac{1}{\alpha}$, at which point (8) and (9) are used.

5. Performance Gains

The performance of the *Core Image* implementation was evaluated on a 2.2 GHz Core 2 Duo MacBook Pro running OSX 10.4.11 with an NVIDIA 8600M GT graphics card. An equivalent CPU version of the algorithm was implemented using the VXL libraries. Since the CPU version

was not multi-threaded, the second core was disabled for all tests to make the comparison fair (as multiple processors could share the tasks of decoding video and communicating with the GPU). The two implementations were tested using a variety of video frame sizes and compression formats, and a summary of typical results is presented in Table 1.

frame size	CPU	GPU	Speed-Up
704×576	3.2 fps	16.7 fps	$5.2\times$
352×288	11.6 fps	58.1 fps	$5.0\times$

Table 1. *Performance gain from GPU background segmentation implementation.*

Although portions of the Gaussian mixture model framework are simple and well isolated, many stages have branching conditions and/or dependencies on previous results. Such aspects will hinder the performance of parallel execution. In this work, we made every effort to minimize the complexity of branching conditions, so that the overhead of predication is minimal.

A raw speed comparison does not illustrate the full performance gain. The load on the CPU is drastically reduced when the background segmentation tasks is offloaded to the GPU. As a result, the main processor becomes available for additional processing, which would not be the case if the background segmentation algorithm was performed in main memory.

Since *Core Image* was designed for processing individual images, some of its optimization choices are not ideal for handling sequences of images. For instance, temporary buffers for "render to texture" feedback operations are allocated and de-allocated as needed. When processing multiple images in succession, the amount of time spent managing video memory can become quite significant. To alleviate this issue, our implementation explicitly allocated buffers to store the matching results and updated (but not sorted) mode parameters.

6. Spatial Cohesion

If the background model is used to generate a foreground segmentation mask, additional spatial filtering is usually conducted. In their work, Stauffer and Grimson use a connected components algorithm to find regions of the image to track. However, the stage is also used as a noise filter. The authors note that "some percentage of the data points 'generated' by a Gaussian will not match (because of a finite D). The resulting random noise is easily ignored by neglecting connected components containing only 1 or 2 pixels" [8].

A similar result can be achieved with morphological operations (which are much easier to implement in a GPU framework). In our application, we use a 3×3 cross-shaped kernel for both erosion and dilation. Although *Core Image* does not have built-in morphological operators, they can be realized quite easily.

7. Conclusions

Since it is a pre-processing stage, background modelling and segmentation needs to be performed quickly. If the algorithm deals with each pixel in isolation, there is a good chance that a parallel GPU implementation will be faster than a CPU equivalent.

In this work, we have described how the temporal history of a pixel can be modelled efficiently as a combination of normal distributions. In the specific case of three modes, we have shown how the algorithm can be implemented on a GPU while achieving a higher processing rate. Moreover, we have implemented a complete version of the algorithm. Previous work [5] appears to have omitted the rather complex sorting stage, and resulted in additional approximations to the algorithm.

References

[1] *Core Image Programming Guide*. Apple Inc., June 2008.

[2] A. Elgammal, D. Harwood, and L. Davis. Non-parametric model for background subtraction. In *Proc. European Conf. on Computer Vision*, pages 751–767, 2000.

[3] R. Fernando, M. Harris, M. Wioka, and C. Zeller. Programming graphics hardware. In *Eurographics*, 2004.

[4] J. Kessenich. *OpenGL Shading Language*. 3Dlabs Inc., September 2006.

[5] S.-J. Lee and C.-S. Jeong. Real-time object segmentation based on GPU. In *Proc. Int. Conf. on Computational Intelligence and Security*, pages 739–742, November 2006.

[6] J. Owens. Streaming architectures and technology trends. In M. Phar, editor, *GPU Gems 2*, pages 457–470, 2005.

[7] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, volume 26, pages 80–113. Blackwell Publishing Ltd., 2007.

[8] C. Stauffer and W. E. L. Grimson. Adaptive background mixture models for real-time tracking. In *Proc. Conf. on Computer Vision and Pattern Recognition*, volume 2, pages 246–252, 1999.