Fast Implementation of the FRAME Algorithm using a GPU Gibbs Sampler

Baruch Lubinsky
School of Electrical Engineering
University of Cape Town
Cape Town, South Africa
Email: baruchlubinsky@gmail.com

Fred Nicolls
School of Electrical Engineering
University of Cape Town
Cape Town, South Africa
Email: fred.nicolls@uct.ac.za

***Abstract*—The FRAME (Filters, Random fields and Maximum Entropy) algorithm for visual textures is presented. This paper describes an approach to implementing the algorithm aimed at decreasing its execution time. This is done by decreasing the complexity of some of the operations and by running the most time consuming component — the Gibbs sampler — on a GPU. Specific details on how the Gibbs sampler is coded to run on the GPU are given. The parallel implementation allows the Gibbs sampler to run more quickly than on a serial processor thereby reducing the time taken to run the FRAME algorithm. This implementation is designed such that more complex problems lead to more threads. The more threads that are run, the greater the advantage over serial execution. Consequently the size and complexity of the visual texture that is processed can be scaled up with minimal increases to the execution time.**

I. Introduction

Visual textures are images that are made up of patterns [1]. Texture information greatly enriches human perception. The patterns that comprise a visual texture are not necessarily strictly regular so it is difficult to model them mathematically. There are many approaches to the problem of creating computer models of visual textures. The work in this paper is based around the FRAME algorithm of Zhu et al. [2].

The power of modern computers has made the computation of many algorithms feasible that were previously considered too complex to implement. One area that has gained a lot of popularity due to this new computational power is Bayesian inference [3]. The integration required to marginalise the posterior can be of very high dimension, and theoretically intractable [4]. Monte Carlo approximations can be used to estimate the required integral in reasonable time. This allows powerful statistical algorithms to be used in practical applications.

This paper considers the use of massively parallel computer architecture to implement a Gibbs sampler. The Gibbs sampler is the innermost function of the FRAME algorithm. The algorithm requires many iterations of a Gibbs sampler to be executed and therefore benefits greatly, in terms of execution speed, when it is optimised. The next section describes the FRAME algorithm in broad terms and discusses the Gibbs sampler. Section III gives the details of the GPU implementation followed by testing details and analysis in section IV. Concluding remarks are given in section V.

II. Background

Zhu et al. develop an algorithm for statistical modelling of visual textures — Filters, Random fields and Maximum Entropy (FRAME) [2]. The algorithm simultaneously confronts the two main problems associated with visual textures, namely synthesis and classification, by building statistical models of the texture [1]. The models produced by this algorithm are attractive because the information they contain is easily interpreted.

A. *FRAME algorithm*

An overview of the algorithm is discussed to place the Gibbs sampler [5] in context. Figure 1 shows a basic flow diagram of the steps in the algorithm. The full details of the FRAME algorithm are too complex to present here, refer to the original paper of Zhu et al. for a complete explanation [2].

Images of a certain visual texture are considered to be instances drawn from a distribution $f(\mathbf{I})$. All samples from the distribution have a similar appearance. The goal of the algorithm is to make inferences about $f(\mathbf{I})$ based on the observed samples. The texture is modelled by a set of filters and the statistics related to the observation's response to those filters.

The algorithm is initialised with a data observation and a filter bank of all the filters that may be used. The selection of filters is very important as a texture with features that cannot be detected by any of the filters in the bank will be impossible to model. A variety of common linear filters are used at different scales and rotations. The algorithm does not require that the filters be linear but this limitation is assumed as it allows for faster computation [6].

The observation is filtered by each filter in the bank and histograms of the responses are calculated. The filter whose response has a histogram with the maximum entropy is selected. The entropy is calculated by,

$$S = -\int H(\mathbf{I}) \log H(\mathbf{I}) d\mathbf{I}, \tag{1}$$

so the histogram, $H(\mathbf{I})$, with the most information is chosen. The filter is removed from the filter bank and added to the model.

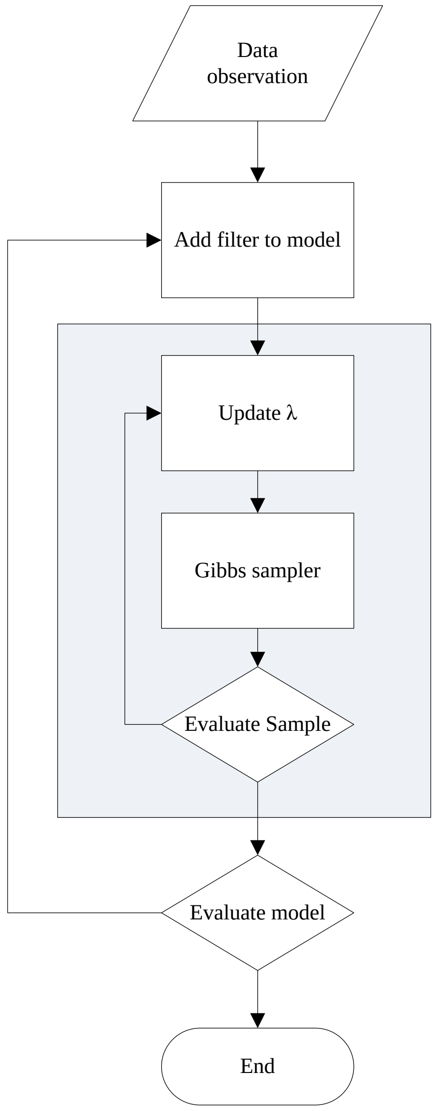


Fig. 1. Basic flowchart of the FRAME algorithm

The model is defined by a set of filters and an array of Lagrangian multipliers associated with each filter:

$$p(\mathbf{I}) = \frac{1}{Z(\Lambda_K)} e^{-\sum_{\alpha=1}^{K} < \lambda^{(\alpha)}, H^{(\alpha)} >} \tag{2}$$

for K filters. The Lagrangian multipliers λ, which act like weights to the histogram bins, are calculated numerically by iterative gradient descent. At each iteration a sample is drawn from $p(\mathbf{I})$ and histograms are calculated for each filter in the model. The differences between those histograms and the histograms from the observation are used to push the Lagrangian multipliers towards values which make $p(\mathbf{I})$ better represent the true distribution underlying the texture.

The algorithm alternates between steps of updating λ and drawing samples from $p(\mathbf{I})$ until the values converge. Then another filter is added from the filter bank. Subsequent filters are selected based on the difference between the histograms obtained from filtering the observation and from filtering the latest sample of $p(\mathbf{I})$. The filter with the largest sum of absolute differences is chosen. So that at each step of the outer loop the filter which contains the most new information about the texture is added to the model.

Samples are drawn from the proposal distributions using a Gibbs sampler [5]. It takes many iterations for the λ values to converge so this process must be carried out many times in order to obtain a good model. Decreasing the time taken to draw each sample greatly improves the performance of the algorithm. The Gibbs sampler is also extremely parallelisable in this application. It is for these reasons that that function is implemented on a GPU.

B. *Gibbs sampler*

The Gibbs sampler is a Markov chain Monte Carlo algorithm and a special case of the Metropolis-Hastings algorithm [7]. It is used to produce samples from probability distributions that are either unknown (but form part of a known joint distribution) or too difficult to sample directly [8]. This application is an example of the latter; the image space is of very high dimension — equal to the number of pixels — so it is impractical to sample the proposal distribution directly.

The algorithm creates a sequence of possible samples that form a Markov chain; each sample depends only on the previous one. It is initialised with a white noise image and at each step, the colour of a single pixel is updated under the current proposal distribution $p(\mathbf{I})$. The probability of the pixel's intesity being each colour is calculated from the distribution and a new colour is selected with likelihood in proportion to those probabilities. After enough steps in the chain the samples are being drawn from the target distribution. However, knowing the exact number of steps required before the Markov chain has converged is not straightforward [9].

The chain can take many iterations to converge making the Gibbs sampler very computationally expensive. The computational complexity can be reduced by a factor linear to the number of threads available by updating multiple components at once. Doing so does not violate any of the theoretical basis of the algorithm as each sample still depends only on the previous one, so the Markov property of the chain is preserved. As the dimensionality of the sample can easily be in the thousands, it makes sense to use a graphics programming unit (GPU) capable of running large numbers of threads simultaneously.

III. Implementation details

When the FRAME algorithm is executed, the Gibbs sampler is required to produce a sample of the proposal distribution each time the λ values are adjusted. It takes hundreds of iterations for those values to converge. The number of pixels to be updated at each step is chosen to be four times the total number of pixels in the image, following the original work of Zhu et al. [2]. For each of these pixels, the Gibbs sampler requires a probability of the pixel being each colour. That probability depends on the histogram produced by each of the filters in the current model. For a simple case of a 32×32 pixel image with eight colours and three filters that equates to producing ten thousand histograms per iteration. This is extremely inefficient so a number of measures are taken to optimise the process.

All the filters in the filter bank are linear and smaller than the entire image. Consequently changing the colour of a single pixel affects a small region of the filter response and this change can be calculated by a single addition [6]. The change to the filter response when adjusting the intensity of a pixel is simply the response plus the impulse response of the filter scaled by the change in intensity. If the intensity pixel p is changed from C_0 to C_1 the filter response $\mathbf{F}$ becomes:

$$\mathbf{F}_{n+1} = \mathbf{F}_n + (C_1 - C_0)\mathbf{K}_p \tag{3}$$

where $\mathbf{K}_p$ is the filter kernel centred on p. This calculation is far simpler than computing the filter response each time and only requires the convolution be computed once per filter.

Extracting a histogram from the filter response is also an expensive operation. This can be obviated in a similar way. The histograms of all the responses for the current sample are stored. As the pixels are flipped to create new samples a count is kept of the changes to each bin in the histogram. For each element of the response that changes its value, 1 is added to the bin containing the value and subtracted from the bin that previously contained it.

This is the operation that is run on the GPU. The pixels to be considered are passed to the GPU code and it returns the changes in histogram bin counts for each colour for each filter. The data required by the GPU are the current sample; the filters in the model; the current sample's responses to each filter; the histograms of those responses; and the set of colours (or intensities) that may be used. One thread is created for each set of pixel, colour and filter. Each thread is required to perform the addition of the filter kernel and to count the changes to the histogram. It is a very simple computational task, requiring very few instructions to execute. There is an inefficiency caused by the fact that all the threads must wait for the thread handling the largest filter to complete. However, none of the filters is very large so this is not a serious bottleneck.

A. Hardware considerations

The GPU hardware used is a NVIDIA GeForce GTX 470 [10]. Programming for a GPU is sensitive to the specific details of the hardware. Some specifications of the card are given in Table 1.

TABLE I
GTX 470 SPECIFICATIONS

CUDA cores per SM	32
Maximum threads per SM	1536
Maximum threads per block	1024
32-bit registers per SM	32 K
Global memory	1280 MB
Shared memory	48 KB
Constant memory	64 KB

These are significant because in order to get the best performance the program must run within the constraints of the hardware while using as much of the power available in each streaming multiprocessor (SM) as possible.

B. GPU program

The GPU code is written in CUDA C [10]. In this platform threads are grouped into blocks with each block containing an array of threads. The fastest type of memory to use is the registers but the space available is limited. Threads from the same block have access to the same block of "shared memory". Shared memory is similar to cache memory on a CPU; it is much faster than the global memory and it has much higher bandwidth to the SM. In addition there is a region of memory called "constant memory" which is available globally, to all threads. Access to data stored in constant memory is cached to increase efficiency.

To initialise the GPU code the filters, current sample and its responses are copied from CPU memory to the GPU. The filters are placed in constant memory as they are to be accessed the most often. If the maximum size of a filter is 32×32 and the values are single precision floating point numbers, the amount of constant memory available limits the maximum number of filters to 16. This is an acceptable restriction as most textures can be well modelled using less than ten filters. The current sample and responses cannot fit in constant memory, although they are constant in the scope of the GPU. There is still a performance gain by storing them globally as it reduces the amount of data that needs to be transferred.

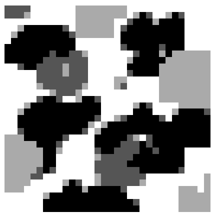
A thread block is created for each pixel under consideration. The thread block contains a two dimensional array of threads — one for each combination of filter and colour. The parameters to the thread are the pixel's coordinates, the set of colours and the borders of the histogram bins. Each thread in the block computes the histogram changes. These can be stored in registers as only one integer is required per histogram bin. Once a thread completes, it waits for all the threads in its block and then the histogram changes are combined into one array to be returned to the CPU.

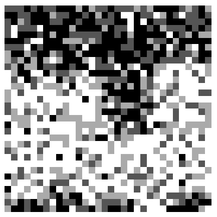
The probabilities of the colours are calculated on the CPU from the histograms. The CPU updates the sample accordingly and changes the responses for the next iteration. These computations are not trivial; they contribute significantly to the overall running time of the algorithm.

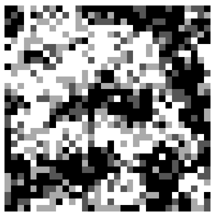
IV. Results and Analysis

The effectiveness of the algorithm is confirmed by viewing a sample image that is produced after the Markov chain has converged. That image should have a similar appearance to the observation. Examples of textures synthesised by the algorithm are shown in Figure 2.

These images are produced by running FRAME with the image in Figure 2(a) as the data observation. The colour palette available to the sampler contains 4 intensities. The algorithm is run for 1000 iterations per filter. The images shown are


(a) Observation


(b) Sample with 1 filter


(c) Sample with 2 filters

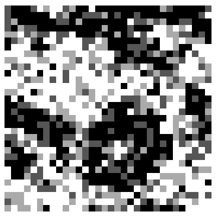

(d) Sample with 5 filters

Fig. 2. Example of a textures synthesised by the FRAME algorithm

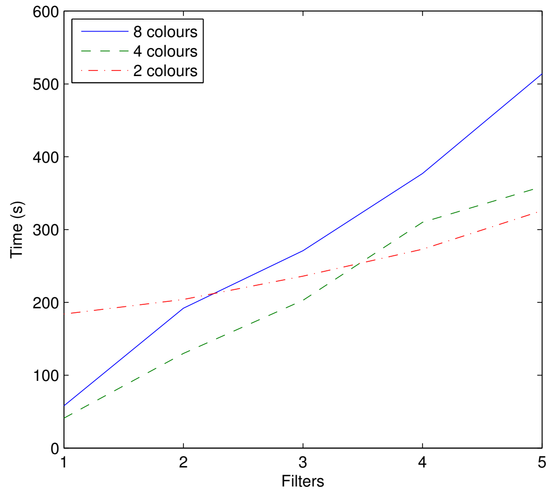
the samples that were drawn having histograms most closely matching those of the observation. Figures 2(b), 2(c) and 2(d) are sampled from models containing 1, 2 and 5 filters respectively. In each case there are artefacts of the initial white noise image remaining. It can be seen that for each filter added to the model the sampled images have more structure to them.

A. Execution time

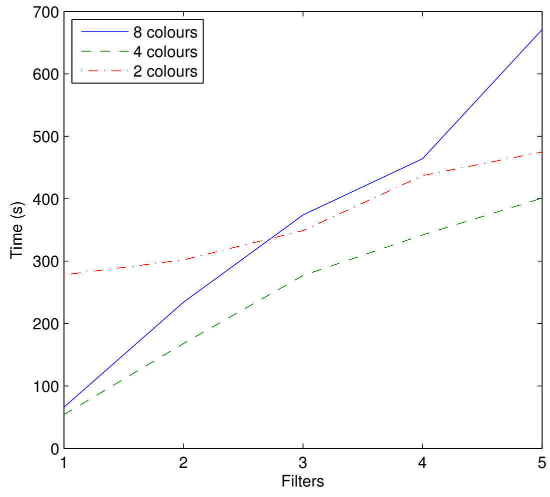
The FRAME algorithm is computationally intensive. In order for it to be useful we must be able to run it in reasonable time. An experiment is conducted to assess the execution speed under different parameters. This is done to test the efficacy of the GPU code and to find the conditions under which the hardware is best utilised.

A small texture patch of 32×32 pixels is used in the tests, at this size a texture can be discerned and the algorithm runs relatively quickly. The image is shown in Figure 2(a). FRAME is run using that image as the observation with a colour palette of 2, 4 and 8 grey levels. The algorithm runs until the model contains 5 filters and 50 iterations are run on each filter. This is a long enough execution to be representative of more extensive tests. The experiment is run three times with the Gibbs sampler updating 16, 64 and 128 pixels at a time, essentially varying the number of simultaneous threads. The results give an indication of the efficacy of the GPU code. One would expect a CPU implementation to have similar performance to the GPU at about 4 threads. Figure 3 shows the time taken, in seconds, to complete 50 iterations in each case.

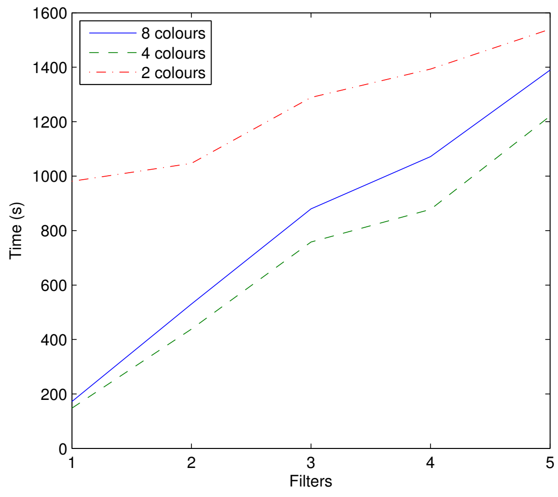
As expected the time taken increases as the model becomes more complex, when more filters or colours are added. However, there are some notable anomalies to this. In each experiment the 2 colour run takes the longest time with one filter. One would expect the 2 colour case to execute in the shortest time because with fewer colours the are fewer calculations. This inconsistency arises from the fact that all the filters are not the same size. When the observation is



(a) 128 pixels



(b) 64 pixels



(c) 16 pixels

Fig. 3. Timing data for different numbers of simultaneous pixels

downsampled to contain only two colours, the features become very large, so the first filter to be selected is larger than that selected for the 4 and 8 colour cases. The greater number of operations required to use this large filter outweighs the time saved by having fewer colours to consider.

The results clearly show that the more pixels which are updated simultaneously, the faster the algorithm runs. The upper bound on that number is set by the hardware and the size of the image itself. If a program attempts to launch more threads then the hardware can handle, some of the threads will be queued, decreasing the performance. This negative impact of this can be avoided by ensuring that the number of threads is a multiple of the capacity of the GPU.

In this application it is incorrect to update too many components at once. Since the filters are larger than a single pixel, changing one pixel's colour has an effect on a region of the filter response. If two or more pixels with overlapping filter regions are flipped simultaneously, the probabilities assigned to their colours will be slightly inaccurate. The set of pixels to be updated at each iteration of the Gibbs sampler is chosen at random. It is possible that some of those will affect overlapping regions. However, since only a small proportion of the pixels is treated simultaneously — about 5 % — this is unlikely and the chain is still able to converge.

The issue of overlapping filter regions limits the maximum number of threads that may be used for a certain image size. When the size of the image increases, in width and height, the amount of pixels and therefore calculations required to produce a sample increases quadratically. The number of pixels that can be safely updated simultaneously increases proportionally to the number of pixels. As a result the time taken to run the GPU code increases linearly with the image size within the limits of the hardware. This is a huge advantage as it allows the algorithm to scale up to more complex problems without incurring the penalty in execution time that it theoretically should.

The same applies when the complexity is increased by adding more colours to the palette. When going from 4 colours to 8 (the observation is almost the same in this case) the number of calculations required to produce a sample doubles. The graphs show that time increase between these experiments is minimal. This is because the execution time on the GPU does not change; more threads are added that can run simultaneously. The small increase is due to the linear operations done on the CPU in selecting the new colour to assign to each pixel.

Consequently, the complexity of the problem can be increased with minimal effect on the execution time. This feature of the implementation is extremely useful for studying the algorithm. It allows the code and parameters to be fine-tuned on simple test cases with the knowledge that the same program can be applied to more complex problems.

V. Conclusion

The FRAME algorithm is useful tool for visual texture analysis. The algorithm is computationally complex, requiring many calculations to be performed within hundreds of iterations. Even with the speed of modern CPUs it can take many hours to run the algorithm. The innermost function of the algorithm is the Gibbs sampler. It takes extremely long to run on a serial processor. By adapting the FRAME algorithm to use a Gibbs sampler implemented on a GPU the computation time is greatly reduced. This parallel implementation provides a greater advantage as the problem becomes more complex. The timing data suggests that the running time could be further decreased by optimising the CPU code and by implementing more components on the GPU. The FRAME algorithm is shown to be considerably more feasible with the inclusion of GPU programming.

Acknowledgment

The financial assitance of the National Research Foundation (NRF), Anglo Platinum and the Department of Chemical Engineering towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily attributed to the NRF.

References

[1] A. Materka and M. Strzelecki, "Texture analysis methods - a review," Technical University of Lodz, Institue of Electronics, Brussels, Tech. Rep. COST B11, 1998.

[2] S. Zhu, Y. Wu, and D. Mumford, "Filters, random fields and maximum entropy (FRAME): Towards a unified theory for texture modeling," *International Journal of Computer Vision*, vol. 27, no. 2, pp. 107–126, 1998. [Online]. Available: http://www.springerlink.com/index/H8740U6G0HMJ686U.pdf

[3] W. J. Bolstad, *Introduction to Bayesian Statistics*, 2nd ed. Hoboken, New Jersey: Wiley, 2007.

[4] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter, *Markov Chain Monte Carlo in Practice*. Chapman & Hall/CRC, 1996, ch. Introducing Markov chain Monte Carlo, pp. 1–19.

[5] S. Geman and D. Geman, "Stochastic relaxation, gibbs distributions and the bayesian restoration of images," *Journal of Applied Statistics*, vol. 20, no. 5-6, pp. 25–62, 1993. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/02664769300000058

[6] S. Zhu, X. Liu, and Y. N. Wu, "Exploring Texture Ensembles by Efficient Markov Chain Monte Carlo - Toward a "Trichromacy" Theory of Texture," *Pattern Analysis and Machine*, vol. 22, no. 6, pp. 554–569, 2000. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=862195

[7] C. Andrieu, N. De Freitas, A. Doucet, and M. I. Jordan, "An introduction to mcmc for machine learning," *Machine Learning*, vol. 50, pp. 5–43, 2003.

[8] G. Casella and E. George, "Explaining the Gibbs sampler," *American Statistician*, vol. 46, no. 3, pp. 167–174, 1992. [Online]. Available: http://www.jstor.org/stable/2685208

[9] D. J. Spiegelhalter, N. G. Best, W. R. Gilks, and HazelInskip, *Markov Chain Monte Carlo in Practice*. Chapman & Hall/CRC, 1996, ch. Hepatitis B: a case study in MCMC methods, pp. 21–43.

[10] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide," NVIDIA, Tech. Rep. Version 3.2, 2010.