GPU-accelerated MRF segmentation algorithm for SAR images

Haigang Suia, Feifei Penga,b, Chuan Xua, Kaimin Suna, Jianya Gonga

a The State Key Laboratory of Information Engineering in Surveying, Mapping and Remote Sensing, Wuhan University, Hubei 430079, China
b Institute of Survey and Mapping of Tianjin, Tianjin 300381, China

Abstract

Markov Random Field (MRF) approaches have been widely studied for Synthetic Aperture Radar (SAR) image segmentation, but they have a large computational cost and hence are not widely used in practice. Fortunately, parallel algorithms have been documented to enjoy significant speedups when ported to run on a graphics processing units (GPUs) instead of a standard CPU. Presented here is an implementation of graphics processing units in General Purpose Computation (GPGPU) for SAR image segmentation based on the MRF method, using the C-oriented Compute Unified Device Architecture (CUDA) developed by NVIDIA. This experiment with GPGPU shows that the speed of segmentation can be increased by a factor of 10 for large images.

1. Introduction

SAR is capable of high-resolution imaging, even from space-borne platforms (Ulaby et al., 1982). Since the radar system is active, it does not depend on natural illumination. Furthermore, the frequencies used are capable of penetrating cloud cover. SAR is thus an all-weather sensor with day and night capability, and has a unique advantage over other systems. For these reasons, SAR systems are growing in importance in the area of Earth’s observation. Their main drawback is that the imagery invariably suffers from a high level of “speckle” noise caused by back-scattering (Lee, 1989). This form of noise is generally non-Gaussian and multiplicative in nature (regions of high intensity are subject to more noise), making statistical treatment difficult (Oliver and Quegan, 1998; Horritt, 1999).

Image segmentation is an important technique in SAR applications. The idea is to divide an image into several blocks from which different objects of interest can be extracted, then segment the image by concatenating areas with the same classification. Segmentation accuracy is a key index in the SAR image analysis. However, speckle noise reduces the spatial resolution of the image and contaminates the detailed structures.

The information contained in SAR images mainly depends on the local statistical properties rather than the value of the intensity, so it is difficult to apply conventional image segmentation methods. Thus, several statistical models and segmentation methods for SAR images have been proposed (Smits and Dellepiane, 1999; Xu et al., 2003; Freitas et al., 2005; Yu and Clausi, 2006; Frey et al., 2007; Li et al., 2009) and the MRF algorithm is one of the most commonly segmentation method used for SAR images. By making full use of statistical features and textures, approaches based on MRF have achieved high accuracy on several types of images (Geman and Geman, 1984; Derin and Elliott, 1987; Li and Lee, 1993; Bouman and Shapiro, 1994; Dong et al., 1997; Venkatachalam and Choi, 1998; Cheng and Bouman, 2001; Zribi, 2007; Li et al., 2010). The main idea of the MRF method is to calculate a new set of labels for the image by estimating a parametric, statistical model of the data. Typically, the MAP estimation method is employed. However, the MAP estimation requires a large amount of calculation, and as the size of the image increases the computational load quickly becomes unbearable. Thus, the MRF method needs to be improved for practical applications.

Nowadays, most desktop computers are equipped with fully programmable graphics processing units. These chips contain many powerful Single Instruction Multiple Data (SIMD) processors, so can support parallel data processing and high-precision computation (Bai et al., 2009) – a practice known as GPGPU. The rapid advances in GPU performance and programmability have made it possible to parallelize digital image processing on personal computers, and a large number of methods have been proposed for this purpose (Luo and Duraiswami, 2008; Yang et al., 2008; Pan et al., 2008). For SAR images, many applications, such as interactive visual analysis of SAR images (Lambers et al., 2007; Lambers and Kolb, 2008), SAR image formation (Hartley et al., 2009; Rubin and Sager, 2010), SAR simulation (Balz and Stilla,
2. CUDA technology for GPGPU applications

GPGPU is a young area of study that has attracted the attention of many research groups in recent years. Although graphics hardware has been used for general purpose computation since the 1970s, the flexibility, processing power, and the low cost of modern GPUs have extended its uses to signal processing, computer vision, computational geometry and scientific computing (Owens et al., 2007).

NVIDIA’s CUDA, AMD Stream Computing technology and OpenGL (Open Graphics Library) are the most popular GPGPU applications at present (Segal and Akeley, 2004; Horn, 2005; Luo and Duraiswami, 2008). In addition, Intel’s Larrabee structure and Microsoft’s DirectX 11 Computer Shade are still being developed. A major advantage of CUDA compared to other GPU programming systems is that it uses a C dialect, so that the C functions originally written for a CPU can often be ported to a CUDA kernel with little modification. Furthermore, NVIDIA provides developers with C libraries that expose all the device functionalities needed to integrate CUDA into a C program. For these reasons, we use CUDA in this paper to test the high-performance SAR image segmentation process.

CUDA exposes the GPU (here called a device) to the CPU (here called the host) as a coprocessor. The GPU executes a function (the kernel), which permits a number of threads to run the same program on different data. From the host’s point of view, kernel invocations are synchronous function calls. The synchronization can be performed explicitly by calling a synchronization function or implicitly when the host tries to access memory on the device. In both cases, it takes the form of a barrier that blocks the calling host thread until all previously calls to the kernel have been completed.

The device is designed to process tens of thousands (or even more) threads at the same time. It hides the costs of memory access by efficient thread scheduling; i.e., when a thread is waiting for a read from memory to complete, other threads could do data processing in the same time. The memory at the device is global memory, meaning that it can be accessed by the host and all processors of the device.

The so-called host runtime component of the CUDA library offers functions similar to malloc and free to manage global memory. In order for a kernel to access memory allocated with a CUDA malloc call, a pointer to the memory must be passed as a parameter. Furthermore, the host runtime component offers a function to copy data from main memory to global memory or vice versa. As the device cannot access main memory, only memory located on the graphics board, global memory must be used to exchange data between the host and device.

A typical CUDA computation follows the steps shown in Fig. 1. The host function starts by allocating one or more buffers in the GPU global memory and transfers the data from CPU to GPU to process. Then the CUDA computations are launched one or more times by addressing thread function. Pointers to data and result buffers are passed as parameters to the thread function. Finally, after the computation has completed, results are copied back to the CPU memory and the GPU buffers are disposed.

In this paper we explore applications of CUDA technology to SAR image segmentation, considering its advantages and limitations. We focus on alleviating the high computational cost of the MRF algorithm, and propose a CUDA solution suitable for SAR image segmentation. Our aim is to show that CUDA can take advantage of GPUs to accomplish complex, time-consuming tasks that until recently required large-scale parallel computers.

3. CUDA algorithms for MRF segmentation method

3.1. The principle of MRF segmentation algorithms

The MRF segmentation problem can be expressed in the Bayesian framework. Bayesian image segmentation can be described as a labeling process: it assigns a label to each pixel in the image. The label identifies the region or class to which the pixel belongs. Naturally, the outcome of this phase is a layered model (see Fig. 2): the first (visible) layer is the image field and the second (invisible) layer is the field of labels.
Assume that the visible image $Y$ and label image $X$ are both defined by a $M \times N$ rectangular grid $S = \{i|j| 1 \leq i \leq M, 1 \leq j \leq N\}$. For every point on the grid, $s = (i,j)$ have two distinct values: $y_s$ is the gray scale intensity of the image, and $x_s$ is the region or class of the pixel. Thus, a segmentation algorithm tries to find the best $X$ that agrees with the known data $Y$. In other words, it tries to obtain the posterior distribution $P(X|Y)$.

Neighborhood concept used to describe whether the grid points in the space grid are adjacent. Assume that $N_s$ represent the neighborhood of grid point $s$, and $N = \{N_s, s \in S\}$. Fig. 3 demonstrates the second-order neighborhood system used in this paper and the corresponding cliques. In the second-order neighborhood system, also called the 8-neighborhood system, there are eight neighbors for every (interior) site, as shown in Fig. 3(a). Fig. 3(b) show clique types for the second-order neighborhood systems for a lattice. And $x_1$ represents the single-site clique, $\beta_1$ and $\beta_2$ represent the horizontal and vertical pair-site cliques, $\beta_3$ and $\beta_4$ represent the diagonal pair-site cliques, $\gamma_1$, $\gamma_2$, $\gamma_3$ and $\gamma_4$ represent the triple-site cliques and $\zeta_1$ represents the quadruple-site cliques. As the order of the neighborhood system increases, the number of cliques grows rapidly and so do the computational expenses involved.

Throughout this paper, we assume that the distributions of feature data are Rayleigh functions with parameter $\mu_r$. That is

$$P(y_s|x_s) = (y_s/\mu_r^2)\exp(-y_s^2/2\mu_r^2))$$

where $\sqrt{2/\mu_r}$ is the mean for class $r$ in the corresponding region and can be estimated as follows:

$$\mu_r = \sqrt{2/\pi} \left( \sum_{i \in X_r} y_i / N_r \right)$$

where $N_r$ is the pixel number of class $r$ in the corresponding region and the conditional density function can be estimated as follows:

$$P(Y|X) = \prod_s P(y_s|x_s) = \prod_s ((y_s/\mu_r^2)\exp(-y_s^2/2\mu_r^2))$$

In our work we have opted for the deterministic method, which is the Iterated Conditional Modes (ICM) algorithm (Besag, 1986), because our goal is to process images of large sizes and with this method the convergence is guaranteed with a reasonable time of execution. The key to the ICM method is the following equation of proportionality for the probability of the label at pixel $s$, given the observed image $y$ and the current estimates $x_{\neq s}$ of the labels of all pixels in the neighborhood of pixel $s$. The notation $x_{\neq s}$ refers to the labels of all pixels in the image, excluding the label at site $s$ itself.

$$P(X_s = x_s|y, x_{\neq s}) \propto P(y_s|x_s)P(X_s = x_s|X_{\neq s} = x_{\neq s})$$

According to the Hammersley–Clifford theorem, a Markov random field $X$ equivalent to a Gibbs random field, then

$$P(X = x) = \frac{1}{Q} \exp(-U(x))$$

where $Q$ is the normalization constant and $U(x)$ is the energy function, the energy

$$U(x) = \sum_c V_c(x)$$

is a sum of clique potentials $V_c(x)$ over all possible cliques $C$.

In this paper, Multi-level logistic (MLL) Model (Derin and Elliott, 1987) is used to describe the invisible layer, in addition, a second-order neighborhood system and pair pixels clique (that is to say, parameters $\beta_1$, $\beta_2$, $\beta_3$ and $\beta_4$ are the only one we considered, and we make $\beta_1 = \beta_2 = \beta_3 = \beta_4 = 0.88$, take experience value instead of $\beta$) are considered in our paper, please see Fig. 3.

The clique potential-energy function can be described as below

$$V_c(x_i, x_j) = \begin{cases} \beta & x_i = x_j \\ -\beta & x_i \neq x_j \end{cases}$$

where $x_i$ means the label value of current pixel, $x_j$ means the label value of its eight neighboring pixels.

According to Bayes theorem and Eqs. (1)–(7), we have

$$P(X|Y) \propto P(Y|X)P(X)$$

$$\propto \left\{ \exp \left( -\sum_{s \in S} \frac{y_s^2}{2\mu_r^2} \prod_{r \in N_s} y_r / \mu_r^2 \right) \right\} \frac{1}{Q} \exp \left( -\sum_{s} \sum_{c \in N_c} V_c(x_i, x_j) / T \right)$$

$$\propto \left\{ \frac{1}{Q} \exp \left( -\sum_{s} \left( \frac{y_s^2}{2\mu_r^2} + \sum_{c \in N_c} V_c(x_i, x_j) / T \right) \right) \right\} \prod_{s \in S} \frac{y_s}{\mu_r^2}$$

where $T$ is a constant and represent the value of temperature, then the best segmentation can be acquired by calculating the maximum the posteriori probability distribution and it can be express as follows:

$$\hat{x} = \arg \max \{ P(X|Y) \} = \arg \max \left\{ \exp \left( -\sum_{s} \left( \frac{y_s^2}{2\mu_r^2} + \sum_{c \in N_c} V_c(x_i, x_j) / T \right) \right) \prod_{s \in S} \frac{y_s}{\mu_r^2} \right\}$$

Maximizing the conditional probability in Eq. (8) is equivalent to the following expression:

$$\hat{x} = \arg \min \left\{ \left( \prod_{s \in S} \frac{y_s}{\mu_r^2} \right)^{-1} \exp \left( \sum_{s} \left( \frac{y_s^2}{2\mu_r^2} + \frac{1}{T} \sum_{c \in N_c} V_c(x_i, x_j) \right) \right) \right\}$$

From the above proposed MRF segmentation algorithm and the ICM algorithm can be described in Algorithm 1. The observed image $y$ is given and estimated pixel labels $\hat{x}$ are computed, two regions are being considered in this paper.
Algorithm 1. Void CPU_MRFProcess ( )
1: Get the initial value $\hat{x}$ using algorithm of fuzzy C-means clustering;
2: Calculate the area mean value $\mu_r$ using Eq. (2);
3: Search from the first pixel, traversing the whole image, calculate the pixel-energy $E(x_1)$ using Eq. (8), and the global energy $E(\text{global})$ is the summation of each $E(x_i)$;
4: Update the label $\hat{x}$ using the Eq. (10), that is to say, for pixel $s$, if $E(x_1) \geq E(x_2)$, then $\hat{x}_s = \hat{x}_s$, if $E(x_1) < E(x_2)$, then the value of $\hat{x}_s$ remain unchanged;
5: For the value of labels are changed, compute a new global energy, named $E(\text{global new})$, then the global energy change $\Delta E = E(\text{global}) - E(\text{global new})$;
6: Assign the value of $E(\text{global new}) - E(\text{global})$;
7: Repeat step 2–step 6 until the global energy change $\Delta E$ tend to recede, in other words, the energy difference before and after is less than a given threshold.
8: Output the segmentation result.

3.2. CUDA-supported approach to MRF segmentation algorithms

By analyzing the MRF segmentation method on CPU, we can find that the most time-consuming process are re-compute parameters and calculate new pixel labels using an iterative process (Algorithm 1), and these two process are feasibility to transfer to GPU-based computing. However, the process of parameter calculation is independent of the global cycle. This portion of the algorithm is easier to transfer to GPU-based computing, for example using the block-strip and cycle-strip styles. And the process of calculating new pixel labels is inside the global cycle and therefore is harder to realizing GPU-based computing. The MRF segmentation method based on GPU is shown in Fig. 4. Fig. 4(a) shows the global process. Fig. 4(b) is a more detailed description of the process on the GPU.

Shared storage for parameter computation is based on the parallel protocol. Firstly, each thread reads a data item and at the same time a copy is made from global memory to shared storage. After this, a synchronous operation is carried out to make sure that all data can be accessed by the next thread. Secondly, in the next FOR statement, only half of the threads in the previous FOR statement are used for summation. Subsequent FOR iterations continue to halve the number of threads used for summation. The final result is stored in the first thread, and output is achieved by copying this result to the global memory. The parallel protocol is illustrated in Fig. 5.

In order to fully explore the implementation of parallel computing with GPUs, a method of calculating a new pixel label is illustrated in Fig. 6. We begin with an unlabeled grid, as shown in Fig. 6(a). When a new pixel label is computed, the neighborhood pixels around the current pixel are needed. Given a fixed radius of neighborhood (for example setting the radius to 2), update labels of those pixels have no enough neighborhoods (here they are called edge pixels and are inclined shaded in Fig. 6(b)) would not be considered. Let us begin with the pixel 19, the neighborhood pixels could be seen in Fig. 6(c) as the shaded areas. In our parallel strategy, two conditions must be satisfied for those pixels could implement the process of parallel computing: one is any of those pixels is not in another’s neighborhood and the other is they are not the edge pixels. Thus, such as pixels 19, pixel 20 and pixel 21 could not be computed in parallel, and this calculation should be better accomplished in CPUs. On the contrary, for pixel 22 in Fig. 6(d) and pixel 27 in Fig. 6(e), they met the two conditions mentioned above, so parallel computing can be used. Those pixels computed in parallel can be seen in Fig. 6(f), such as pixel 22 and 27, pixel 30 and 35, pixel 38 and 43.

For an image of $16 \times 16$ pixels, the sequences for calculating new labels are shown in Fig. 7. In the serial process, the total number of pixels that can be labeled is $(\text{width} – 2 \times \text{radius}) \times (\text{height} – 2 \times \text{radius})$. Thus, in Fig. 7(a) there are 144 pixels, each requiring on the order of radius$^2$ calculations. In the parallel process, the total number of steps is $\{\text{radius}\} + 1 + \{\text{height} – 2 \times \text{radius} – 2\} \times (\text{radius} + 1) + 1 + (\text{width} – 2 \times \text{radius} – 1)$. Thus, in Fig. 8(a) there are 45 steps. Compared to the serial process, 99 fewer steps are required.

4. Experimental results

Experiment 1. We have compared the CUDA-based solutions described in Section 3 to the CPU execution described in
Algorithm 1. The CUDA-based approach (Algorithm 2) has been tested in a computer with Processor Inter Core 2 Quad-Core running at 2.66 GHz with 2 GB of RAM and a Windows-based operating system, using a 1 GB NVIDIA GeForce 9800 GT graphics card supporting atomic functions.

The experimental SAR image was acquired by Radarsat-2 with a resolution of 3 m. The segmentation results of the two algorithms are shown in Fig. 8.

Because the accuracy of a single-precision expression is lower in the GPU processor than in the CPU processor, the two algorithms can lead to slightly different results over several iterations. In this example, only the pixel arrowed in Fig. 8(d) is affected.

The time consuming on single core CPU processing and GPU method are shown in Fig. 9 on the log scale. The SAR image is $16,384^2$ (256 M). In order to test the computation speed, the original SAR image was interpolated into different scales of images. It can be clearly seen that in Fig. 9, when the data size is 16 k or smaller, the time consuming on GPU is more than on CPU with single core. Because when we handle the small data, the time consuming on data transfer between CPU and GPU takes a greater proportion in the whole process (the whole process...
include two stages, one is data transfer between CPU and GPU, and the other one is data processing on GPU). Therefore, when we handle the small data, we could do it completely on CPU.

For the same data, the time improvement is defined as the CPU consumption divided by the GPU consumption. The time improvements to the MRF method are shown in Fig. 10. It can be seen that the greater the volume of data, the more important the time improvement. When the image is very small, more time is spent on data transmission between GPU and CPU than data processing in the GPU. As a result, the GPU algorithm consumes more time than the CPU algorithm, and the time improvement is less than 1. With an increase in data processing, time improvement increases, and finally, it stabilizes (in this paper, the stable value is 10.53).
To measure the relationship between an algorithm’s efficiency and image size, we define a metric named the increase in time consumption (ITC for short):

\[ \text{ITC} = \frac{(\text{time}_2 - \text{time}_1)}{(\text{data}_2 - \text{data}_1)} \]

Taking the CPU algorithm as an example, we find that an image with 128 × 128 (16 k) pixels requires 235 ms while an image with 256 × 256 (64 k) pixels consumes 828 ms. The ITC for this step in image size is (828 ms/235 ms)/(64 k/16 k) or 0.88. If the ITC is less than 1, it represents a better result. The ITCs for both algorithms are plotted in Fig. 11.

For the GPU algorithm, as the image size increases to 4096 × 4096 (16 M), the ITC curve increases, then stabilizes at a value close to 1. For the GPU algorithm, this image size is a turning point. This phenomenon occurs because large datasets are processed in blocks of constant size. And it can be obtained that when the data size is greater than 16 M, it is perfect for processing on GPU, at least in this hardware environment.

**Experiment 2.** In order to test the significance speedups on GPU, a multi-core CPU parallel processing experiment without GPU is designed and implemented. The CPU parallel processing algorithm is implemented using OpenMP and on a computer with an Intel Xeon W5580 8 Cores processor running at 3.2 GHz, 16 GB of RAM. The comparison results with eight cores CPU and GPU shows in Fig. 12. It can be obviously seen that the time improvements with GPU is higher than with eight cores CPU, the speedup on GPU with eight cores is about 7.26, and however the speedup on GPU is 10.53. Moreover, the expense of CPU with eight cores is much higher than GPU used in this paper. Thus, the result shows that we run on a low-cost hardware environment but earn a high performance response. For data transfer problem between CPU and GPU mentioned above, when the data size is about 2 M or smaller, the speedup on CPU with eight cores is higher than on GPU.

5. Concluding remarks and discussion

In this paper, we present a GPU-accelerated SAR image segmentation method using the NVIDIA CUDA. We realized significant improvement in runtime (speedup to 10.53), with imperceptible degradations of quality. Although this experiment has verified that CUDA-based parallel computation on a GPU significantly increases the speed of the segmentation algorithm, much work remains to be done.

- The GPU computing power in our hardware environment is 1.1 and cannot support for the double data, high version of the CUDA will provide more performance optimization and handling support, a higher speedup is also can be achieved. The quantitative results are related to the specific hardware environment, such as the number of stream processors and the size of video memory. Thus, the MRF segmentation algorithms should be tested with different hardware environments.
- According to the result of the experiment, although the CPU parallel processing is less efficient than GPU parallel processing, it is still useful for reducing the time consuming of the MRF segmentation, therefore much work can be done by integrating multi-cores CPU and GPUs.
- As CUDA supports parallel processing on multiple GPUs, the experiment can be implemented in GPU clusters for further efficiency improvement.

![Fig. 9. Time consumption curves of single-core CPU and GPU.](image9)

![Fig. 10. Time improvement of the MRF segmentation method with GPU.](image10)

![Fig. 11. The increase in time consumption (ITC) curves of the MRF segmentation method with single-core CPU and GPU.](image11)

![Fig. 12. Comparison with eight cores CPU and signal GPU processing.](image12)
Acknowledgment

This work was supported by National Key Fundamental Research Plan of China (973) (Grant nos.2006CB701304; 2012CB719906), National Natural Fund of China (NSFC) (Grant nos.40901211; 60602013 and 61001187), National High Technology Research and Development Program of China (863 Program) (Grant no. 2007AA120204).

References


Luo, Y.C., Duraiswami, R., 2008. Canny edge detection on NVIDIA CUDA. In: Proceedings of the Workshop on Computer Vision on GPUs, CVPR.


