International Journal of Multimedia and Ubiquitous Engineering
Vol.9, No.1 (2014), pp.29-48
http://dx.doi.org/10.14257/ijmue.2014.9.1.04

State-Based Gauss-Seidel Framework for Real-time 2D Ultrasound Image Sequence Denoising on GPUs

Banpot Dolwithayakul[1], Chantana Chantrapornchai[2] and Noppadol Chumchob[3*]

[1]Department of Computing, Faculty of Science, Silpakorn University, Thailand
[2]Department of Computer Engineering, Faculty of Engineering, Kasetsart University, Bangkok, Thailand and
Department of Computing, Faculty of Science, Silpakorn University, Thailand
[3]Department of Mathematics, Faculty of Science, Silpakorn University and
Centre of Excellence in Mathematics CHE, Si Ayudthaya Rd., Bangkok, Thailand

*chumchob@gmail.com**

Abstract

The ultrasound image sequences are not only majorly contaminated by multiplicative noises but they are also usually contaminated with additive noises. As in the past few decades, there were some works, which had focused on removing the noises from ultrasound images, such as in the JY model [1] and in the variational model, which were able to remove both types of noises. However, denoising these noises from the ultrasound image sequence is a time-consuming process that occurred from using fixed-point iterative method. From our investigation, the most time-consuming process part of the denoising process is the Gauss-Seidel. By parallelizing these processes with modern multi-core and many-core processor, the denoising ultrasound image in real-time is possible. In this study, we propose the modified strategy from [2] for managing threads and propose the modified state-based Gauss-Seidel method from [16] for GPUs. Our proposed model can retain the frame order, and get the satisfactory frame rate (about 23.33 fps). The proposed strategy boosts the speedup of the frame denoising to 13.80 times compare to the sequential computation.

Keywords: *Real-time image sequence denoising; Parallel computing; OpenMP; Graphic Processing Units (GPUs); multi-core; CUDA; Image processing; Ultrasound image sequence*

1. Introduction

In the real-world, the usage of image sensor, such as image sequence camera, sonar, and ultrasound usually incurs the noises to the media. The noises may cause degrading in image quality or image sequence quality; therefore, this can lead to losing some important information in the media. In the past decades, there were a lot of studies aimed to restore images or image sequences from noise.

In the mathematical area, image noises are categorized into two categories: additive and multiplicative noises. The additive noise can be written as

$$z = u + \eta . \qquad (1)$$

* *Corresponding Author*

ISSN: 1975-0080 IJMUE

where z is the noisy image, u is the noise-free image and η is the noise in the image. The multiplicative noise can be expressed as

$$z = u\eta \tag{2}$$

The ultrasound image and image obtained from Synthetic Aperture Radars (SARs), such as radars and satellites [12] contains both additive and multiplicative noises as suggested by Hirakawa and Parks [4]. These noises can be expressed as (3):

$$z = u + (k_0 + k_1 u)\eta \ , \tag{3}$$

where k_0 and k_1 are parameters indicating how many additive and multiplicative noises are in the image. These noises require more complicate model to remove them.

However, the ultrasound image has different noise model [1] as in Equation (4)

$$z = u + (k_0 + k_1\sqrt{u})\eta \ , \tag{4}$$

In this study, we assume that each frame has the same distribution of noises and the ratio(k_0/k_1) of additive and multiplicative noises are the same in every frame, and there is no noise dependency between each frame.

The image sequence denoising is more complicated than single-image denoising. There are complications due to the following aspects.

1. **Frame rate**. Normally, human eyesight can process about 10-12 frames per second [5]. As suggested by [5], the real-time image sequence frame rate should be normally higher than 15 frames per second, in order that the latency will not be noticed.

2. **Frame order**. The denoised image sequence frames order must be retained. In the process, the output frames need to be merged in the correct order.

3. **Frame rate control**. The output image sequence should have a stable frame rate for the entire image sequence playback to guarantee the quality of service.

With these challenges, image sequence denoising usually cannot be done in real-time due to the extensive computation. With the current multi-core and many-core technology, such as multi-core processor and graphic processing units (GPU), it makes the image sequence denoising possible.

In this paper, we extend the previous work from [2] which used the ROF model [8] for denoising image sequences in real-time using GPUs. The work in [2] can only remove additive noise with GPUs in real-time. Nevertheless, this study aims to remove noises from the streaming ultrasound image sequences which contain both additive and multiplicative noises by using both GPU and OpenMP technology.

2. Backgrounds

This section consists of two parts: the first subsection shows the denoising model used in this work and the next subsection shows the CUDA architecture for GPU computing [6, 7].

2.1. New Variational Model Noise Removal Algorithm

The variation model for restoring an image that is contaminated with both additive and multiplicative noises can be modified from Equation (3) as described by Equation (5),

$$z = u + k_0\eta + k_1\sqrt{u}\eta, \tag{5}$$

where η is the additive noise and multiplicative noise, respectively.

Due to the independence of additive and multiplicative noise, we can measure these noises using Equation (6)

$$D[u] = \frac{\alpha_1}{2}\int_{\Omega}(u-z)^2 d\Omega + \alpha_2\int_{\Omega}\frac{(z-u)^2}{u}d\Omega \tag{6}$$

Here $\alpha_1 > 0$ and $\alpha_2 > 0$ are the regularized fitting parameters for the additive noise and the multiplicative noise removals, respectively, Ω is the domain of the image, u_x and u_y is the differential on the x-axis and y-axis correspondingly. By using Euler-Lagrance equations.

The variation model and JY Model[1] for removing both additive and multiplicative noises is given by Equation (7)

$$\min\{J_{\alpha_1,\alpha_2}(u) = D[u] + R[u]\} \tag{7}$$

where D[u] is the total variation term and R[u] is the regularization term described as

$$R[u] = \int_{\Omega}|\nabla u|_{\beta}\, d\Omega = \int_{\Omega}\sqrt{u_x^2 + u_y^2 + \beta}d\Omega, \beta > 0 \tag{8}$$

According to the calculus of variations, the Euler-Lagrange equation from Equation (7) is given by:

$$-\nabla\frac{\nabla u}{|\nabla u|_{\beta}} + \alpha_1(u-z) + \alpha_2\left(1-\frac{z^2}{u^2}\right) = 0 \tag{9}$$

where $\mathrm{K}(u) = \nabla\cdot\left(\frac{\nabla u}{|\nabla u|_{\beta}}\right), |\nabla u|_{\beta} = \sqrt{|\nabla u|^2 + \beta}$, and $\beta > 0$ is a small constant to avoid the divide-by-zero. By using the finite difference method for discretization Ω to the discrete domain Ω_h, where h is the distance between each grid point, we discretize the domain into $n_x \times n_y$ grid cells. Each cell has the size of 1×1 ($h_x = h_y = 1$). The discrete equation on (x_i, y_j) on Ω_h is obtained by Equation (10)

$$\underbrace{-\mathrm{K}^h(u^h)_{i,j} + \alpha_1((u^h)_{i,j} - (z^h)_{i,j}) + \alpha_2\left(1-\frac{z^2}{u^2}\right)}_{\mathrm{N}(u)} = (g^h)_{i,j}, \tag{10}$$

where

$$\mathrm{K}^h(u^h)_{i,j} = \left[\frac{\delta_x^-}{h_x}\left(\frac{D(u^h)_{i,j}\delta_x^+(u^h)_{i,j}}{hx} \right) + \frac{\delta_y^-}{h_y}\left(\frac{D(u^h)_{i,j}\delta_y^+(u^h)_{i,j}}{h_y} \right) \right]. \quad (11)$$

From Equation (10), there are several methods to solve it; for example, the Time Marching technique is a simple iterative technique using a synthetic time variable [13]. However, this method converged slowly to the solution and, therefore, is not suitable for the parallel computing because of the data dependency in each iteration.

Alternatively, the fast and robust method for solving Equation (10), called Fixed-Point iterative method, was proposed by Vogel and Oman [9, 10]. This method works by freezing some coefficients and converting the problem into a system of linear equations, which can be solved by using an iterative solver such as Gauss-Seidel, a modern solver technique like multi-grid(MG), or a preconditioned conjugate gradient (PCG). However, our previous researches showed that using Gauss-Seidel method can obtain a satisfactory convergence rate with an acceptable accuracy. Thus, in this work, the local fixed point iterative method is used because it is highly parallelizable and it is easy to implement on both the multi-core CPU and the GPUs.

2.2. Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture (CUDA) is an architecture for Single Instruction Multiple Data(SIMD) from NVIDIA®. Despite of the graphic processing, this render graphic card with CUDA can be used as a general-purposed processor, which is called General Purpose Graphic Processing Unit (GPGPU).

CUDA has 4 levels of memory. The first level is called "global memory". It is the slowest memory accessed by the GPU. Hundreds of clock cycles are needed to access the global memory. The next level is called "shared memory," which is the fastest memory that a user can allocate and manage on the GPU device. Reading and writing through the shared memory uses approximately 40 clock cycles. Another two levels are local memory and texture memory. Both are large memories and can be allocated by users. To access them, more cycles are needed when compared to accessing the shared memory. However, this still uses the same number of cycles as the global memory. The CUDA memory model can be shown as in Figure 1.

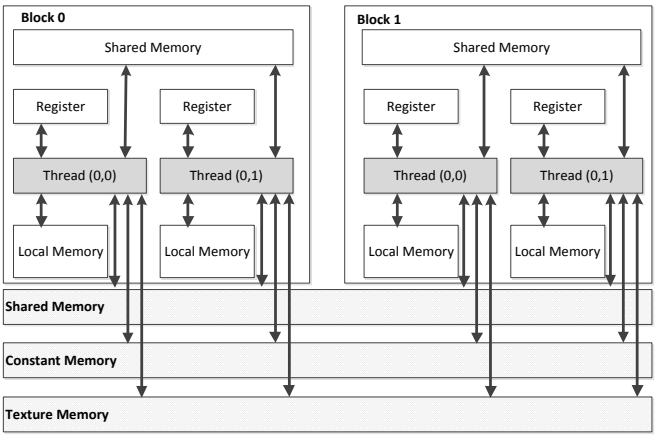


Figure 1. CUDA memory model [6,7]

For programming on the CUDA platform, the developer has to specify the number of threads for computation. Threads that will be executed in a kernel must be managed as groups of threads with shared data, called thread blocks. A group of blocks forms a grid. Creating, organizing, and destroying threads on the GPU consume only a few resources. This allows the developers to manage hundreds of threads very fast and effectively.

2.3. Sliding Window Gauss-Seidel

The approach is to parallelize Gauss-Seidel from the fixed-point method as we have described in Subsection 2.1. At the best of our knowledge, the latest and most efficient parallel approach for Gauss-Seidel named "Sliding Window Gauss-Seidel" works [14] by dividing the matrix row into blocks and windows. Each thread executes the Gauss-Seidel method on its window. After executing the method, all threads slide down to the next block and execute again in the same manner until the last thread executes the job in the last block. This parallelization is illustrated in Figure 2.

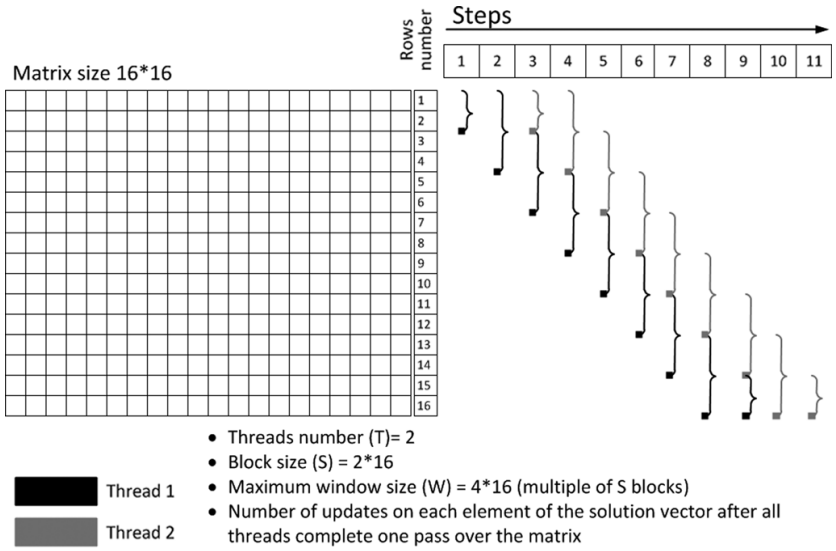


Figure 2. Sliding Window Gauss-Seidel (SWGS) example for 2-threads computation [16]

The work in [14] is based on the multicore platform. For the GPU platform, the appropriate window size and block size may be different. In our work, we need to consider this issue since utilizing the GPU platform, which has massive number of threads, as many threads as possible should be executed.

To explain this algorithm, consider Figure 2. It shows how SWGS works with 4×6 domain size with the window size of 4 and the block size of 2. The algorithm divides each main iteration into substeps. For the substep in Figure 3(a), first thread enters its job first, and then works one block while other threads wait to enter its job. After the first thread finishes its computation, it will slide one block as in substep in Figure 3 (b). When it slides more than 1 window, another thread can enter its job as substep in Figure 3 (c) and will keep doing this until the last thread finishes its job.

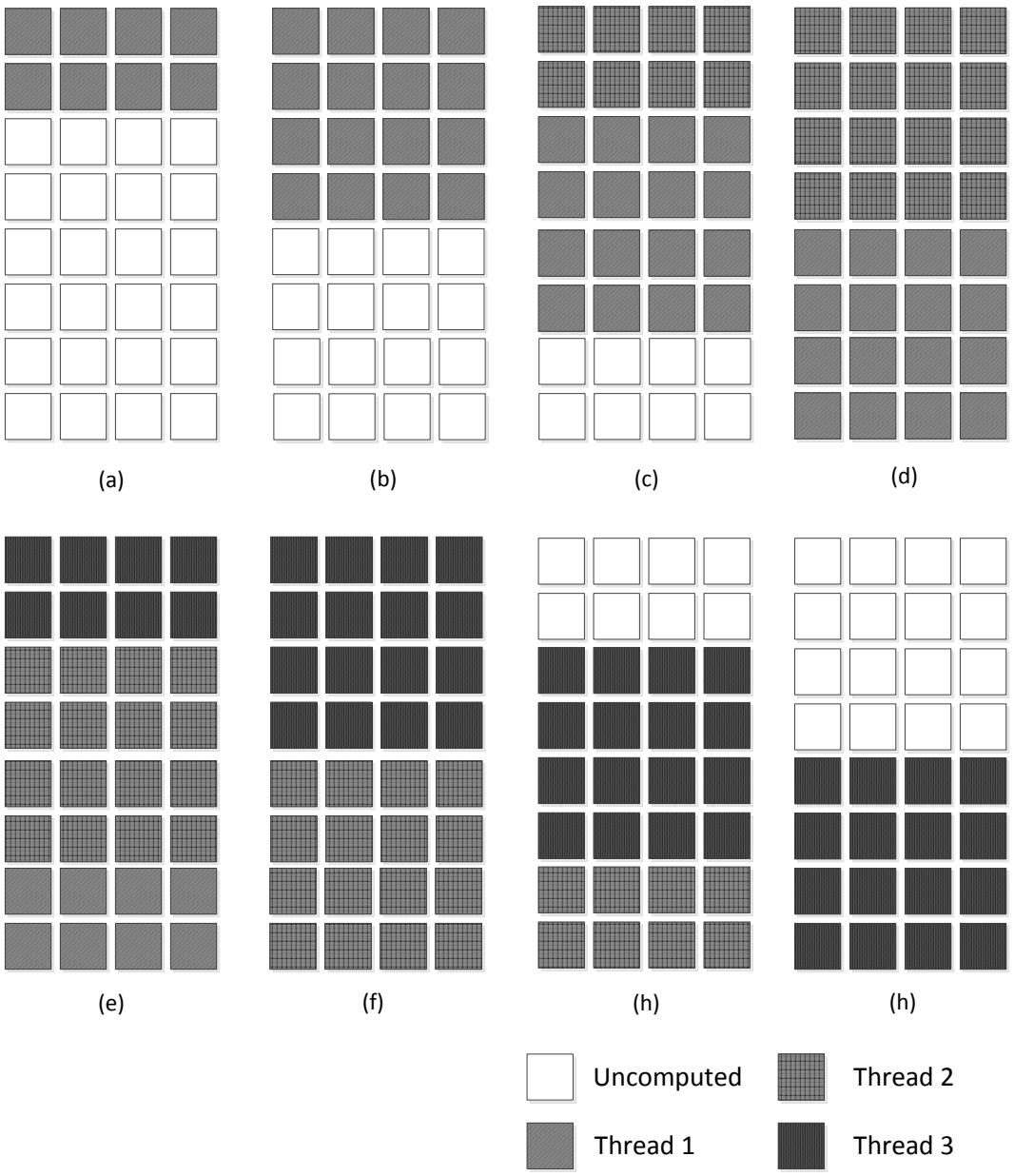


Figure 3. Example of Sliding Window Gauss-Seidel (SWGS) for the first 6 substeps on the domain size of 4×6.

The original SWGS works in the producer-consumer model in [14] which performs greatly on the multi-core computer because not many threads are used in the computation. This keeps the thread synchronization and the memory transfer overheads very low compared to the whole computation time. On the contrary, in Graphic Processing Units (GPU), there are a lot of threads which can incur lots of overheads.

In our previous works, we had proposed a new State-Based Gauss-Seidel method [16] which works efficiently on the multi-core processor. However, to use this method on the GPU, which differs very much in architecture, the proper modification is required. Therefore, we redesign the data structure and management as described Section 3.2.

2.4. State-based Gauss-Seidel

In the previous work by [14], there are two main bottlenecks on the algorithm due to the pipeline computation style. First, there is a significant amount of thread synchronization since it uses the producer-consumer model, which the synchronization occurs at the end of each substep. Next, the pipeline computation style makes a lot of idle threads at the early stage of computation. To alleviate these problems, in our method, we introduce a state-based framework with an asynchronous-style communication to parallelize the Gauss-Seidel computation.

2.4.1. Data Structure: In the proposed approach, a novel data structure design is used to keep track of the progress in states as follows: matrix U represents the domain with initial values, an integer matrix, called an Iteration_Matrix, with the same size of U for storing the current iteration number on each matrix cell. We use a list, called Job Table to store available jobs and a constant (Max_Iteration) to specify the maximum iteration.

Mark_As_Read is a one-dimensional array of integers which stores the *Mark_As_Read* signal for each thread. As an example, in Figure 4 assuming the 3×3 domain size with 4 threads, we assign the first thread (Thread #1) to work on the cell (3,1). For the sake of clarity, we use this example throughout this section.

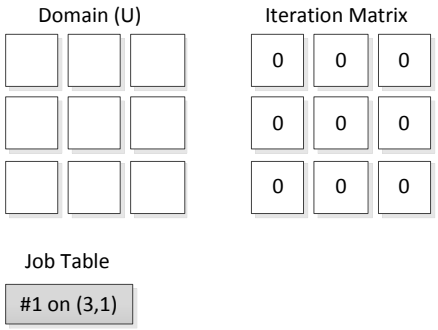


Figure 4. Initial state on each matrix *U* and *Iteration Matrix.*

After applying the first job on the first cell, each thread will move to the following states.

2.4.2. Thread States: In the following explanation, we use the legend to indicate the cell occupied by each thread shown in Figure 5.

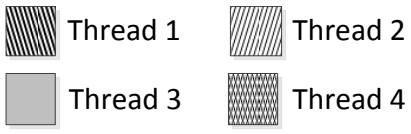


Figure 5. Legend for each thread illustrated in the proposed method.

The thread has four states: Waiting State, Working State, Validation State, and Shifting State.

(i) Waiting State

Every threads start at this state. While a thread is in the Waiting State, it keeps scanning *Job Table* to check if there is an available job for it. If it finds the job, it will delete the job list from the *Job Table* and will change its state to Working State.

(ii) Working State

The thread starts working on the cell number given by *Job Table* and performs the following actions.

1. Read the data from current and its neighboring cells. For the right and upper cells, the thread sends *Mark_As_Read* signal to threads that compute neighboring cells.
2. Start working on the current cell but does not update the data on the current cell yet.
3. Increase the iteration number by 1 on *Iteration_Matrix* in the corresponding working element and enter the Validation State.

The results after the first thread computes on its job are shown in Figure 6.

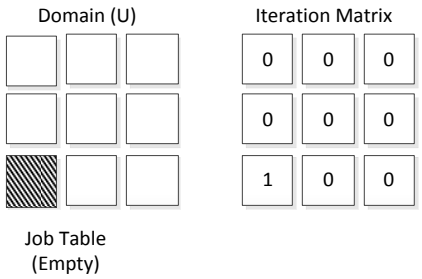


Figure 6. *Job Table, Iteration_Matrix,* and *U* after the first thread finishes its Working State

(iii) Validation State

To maintain the correctness of the Gauss-Seidel method and data integrity, when the thread finishes its computation, the following checking steps are performed.

1. If the current cell is in the first cell, say cell number (3,1) in the example, the thread can update the data in the current cell without any validation.

2. If the thread computes the element in the last row or in the first column, it must receive a *Mark_As_Read* signal from the thread computing the left cell before it can update the data. This is to ensure that the left cell uses the correct data to compute.

3. Other threads must have two *Mark_As_Read* signals from the left and lower cells before it can update the data. This means the previous cell's computation has been finished.

In particular, this validation is to ensure that the right and upper cells of the current working cell are not updated by the new data from the next iteration if the threads on the right and upper cells finish the computation before the data is read by the current cell. After updating the data, the thread is turned into Shifting State.

(iv) Shifting State

The thread which enters this state must decide what to do next. The thread makes a decision based on the following conditions:

1. If the thread is working on the last column, it will enter the Waiting State again.

2. If the current iteration number in *Iteration_Matrix* on its right cell reaches *Max_Iteration*, it will stop the computation and return to Waiting state.

3. If the thread is working on the first column and on *Max_Row*, it will pick two threads which is in Waiting State and assign new threads to work on its upper and current cells. If there is no thread available in the Waiting State, it will pick the first thread found from the *Job Table* and increase *Max_Row* value by 1, if it doesn't reach *Max_Row*. It will pick one for its current cell only . The current thread checks *Iteration_Matrix*. If the right cell has *Iteration_Matrix* value lower than that of the current cell by 1, the current thread will check for the iteration number of the right cell in *Iteration_Matrix,* if its value is greater than that of the current cell, then it shifts itself to work on its right cell and turns itself to Working State.

4. If the thread does not work on the first column, it will compare the *Iteration_Matrix* between its cell and its right cell. If the iteration number of the right cell is greater than or equal to that of its cell, then the thread moves to compute the right cell and turns itself to Working State.

From our example, we show the working of threads and the computing data *U* after this stage in Figure 7 and the next working state of Thread #1 in Figure 8. Thread #1 starts its Working State and Thread #2 and Thread #3 have found their jobs from *Job Table*. Then, they delete their jobs from *Job Table* and start working.

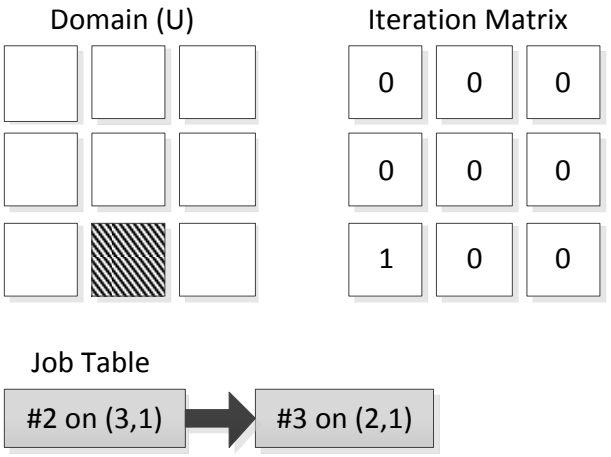


Figure 7. Working of threads and data after Shifting State

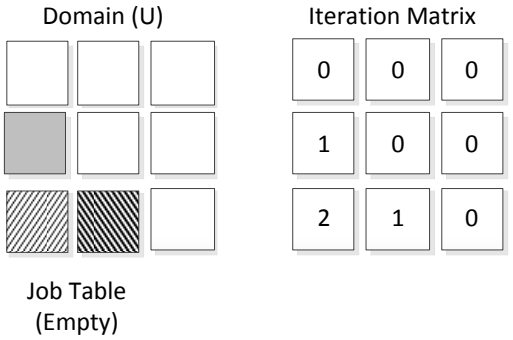


Figure 8. *Job Table* and *Iteration_Matrix* after thread #1 enters its working state again

Figure 9 shows the state diagram of the approach. It shows each thread state transition where *S1*, *S2*, *S3* and *S4* denote the *Waiting State*, *Working State*, *Validation State* and *Shifting State* respectively. A thread in *S1* will change to *S2* when it finds a job in the job table and then starts working on it. After the thread finishes computation, it will enter *S3* for the *Validation State*. Once the validation is complete, the thread will update the data and enter the last state (*S4*).

After the thread enters *S4*, it will decide by the following condition: If there are more non-computed cells on that row and the right cell (if any) has finished its computation, it will shift itself to the right cell and back to *Working State*.

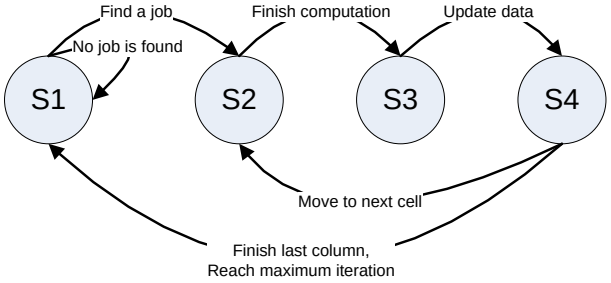


Figure 9. State diagram for each thread

3. Proposed Strategy

We divide this section into 3 parts. First, we investigate time used in each parts of the denoising algorithm. Next, we make State-Based Gauss-Seidel from [16] suitable for GPU computing by redesigning the data structure and hierarchy. Finally, we put the Gauss-Seidel and other techniques together by using OpenMP and CUDA, so denoising ultrasound image sequence is done in real-time.

3.1. Investigation of time used in frame denoising.

To help designing our strategy for denoising the ultrasound image frame; first, we measure the time used in each step in percentage as in Figure 10.

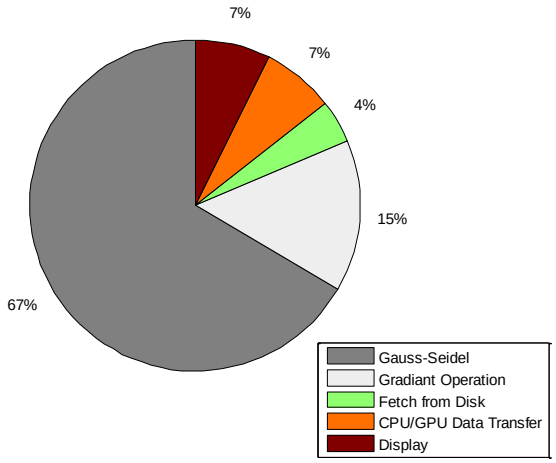


Figure 10. Time used in each part in percentage of denoising process of 256×256 frame size

From Figure 10, it clearly shows that Gauss-Seidel is the most time consuming part of the denoising process. Additionally, we tried to vary the frame size to 1024×1024 as in Figure 11.

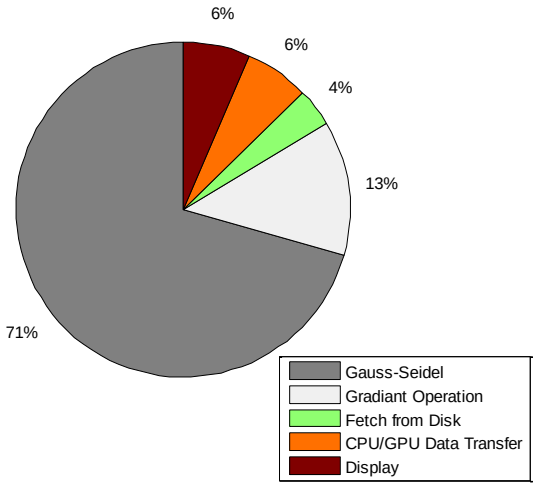


Figure 11. Time used in each part in percentage of denoising process of 1024×1024 frame size

The results in Figure 10 and Figure 11 are very similar. We found that the larger frame size, the more dominant of Gauss-Seidel part. On the 256×256 frame size, Gauss-Seidel part consumes 67% of the total time whereas when the frame size is larger, 1024×1024, the Gauss-Seidel consumes 71% of the total computational time.

Since the Gauss-Seidel part is the major part of the denoising process, to denoise the ultrasound image sequence in real-time, the acceleration of Gauss-Seidel process is necessary. In our previous works, we parallelize Gauss-Seidel by using Sliding Window Gauss-Seidel from [14]. However, this method has a lot of overhead when working with many threads. We proposed the state-based method in [16], but the data structure was not suitable for GPU. In

this work, we also make some modifications of the data structure and memory usage to fit the GPU platform in Section 3.2.

3.2. Modification for State-based Gauss-Seidel for GPU

In the GPU implementation, the GPU programming paradigm has some limitation on the pointer data type. We employ queue data structure instead to represent *Job Table*. Each thread has its own queue structure to avoid the single point of synchronization and to guarantee the mutual exclusion. Notice that we may use the queue data structure in the previous version; however, the queue approach usually takes more memory space because in practice, array memory allocation for each thread is required at the beginning of computation while using the list structure, we can allocate and deallocate each element dynamically. On the GPU, each thread has its own queue. The job queue will be allocated on shared memory on GPU to reduce access time. Each block of thread will have its own job queue as shown in Figure 12.

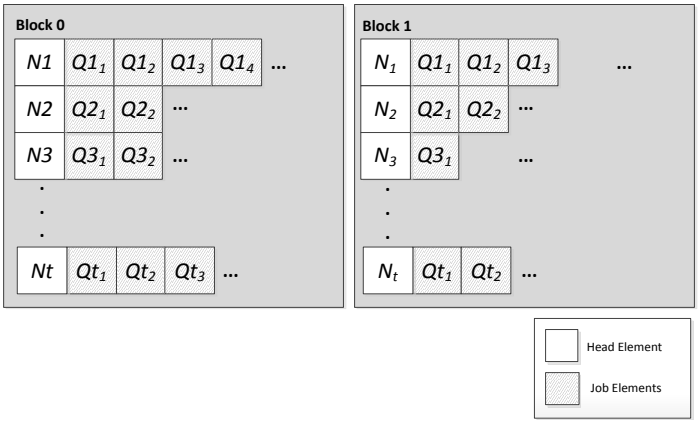


Figure 12. The example use of new data structure for GPU version

In Figure 12, we use two-dimentional array for storing the job queue. Each thread will use a row of array to store and search for their job. There are two types of array elements:

First, the head element contains the number of job element in its row. Every time a new job is inserted (enqueued), the head element will increase its value by 1, and when the job is fetched (dequeued), the head node value will decrease by 1. $N_1..N_t$ indicate the head node for thread number 1..*t* in Figure 12.

Unlike the CPU, which consists of few threads, the GPUs have massive number of threads. There will be an issue that may cause our algorithm to be less efficient. Many threads will keep searching on the *Job Table* and *Iteration Matrix; consequently*, the single point of synchronization occurs which cause only one thread to be able to search and update *Job Table* and *Iteration Matrix*, while the others need to wait to guarantee the mutual exclusion.

To overcome this problem, we make these synchronization points become distributed. For the *Job Table*, we divide threads into blocks and use a shared memory, which is the fastest usable memory for storing *Job Table*. In the Fermi® device, there is 48kB of shared memory per processor. The shared memory on each block of the GPU is allocated by a queue array and will be used only for the threads of each block. When a thread is going to add a job for another thread in *Job Table*, it will pick a free thread in its own block only. This will reduce the single point of synchronization issue from many threads accessing the *Job Table* at one time. In this experiment, we use up to 32 threads per block.

3.3. Thread Management Strategy

The aim of our strategy is to make the denoising process to be able to be done in real-time by utilizing all the computation resource both CPUs and GPUs efficiently.

On the CPU side, we create 4 types of threads as follow:

1. Fetching Thread is used to fetch and pre-fetch the frame and convert the image into 8-bit grayscale image, create the label for each frame and store them in the main memory and label.

2. Compute Thread is used to compute gradient operation (K(*u*)) and transfer the computed frame to GPU's memory.

3. Merging Thread is used to transfer denoised frame data back from GPUs and sort the denoised frame and discard expired frame.

4. Display Thread is an optional thread to display the denoised image sequence in real-time.

Our strategy is illustrated in Figure 13.

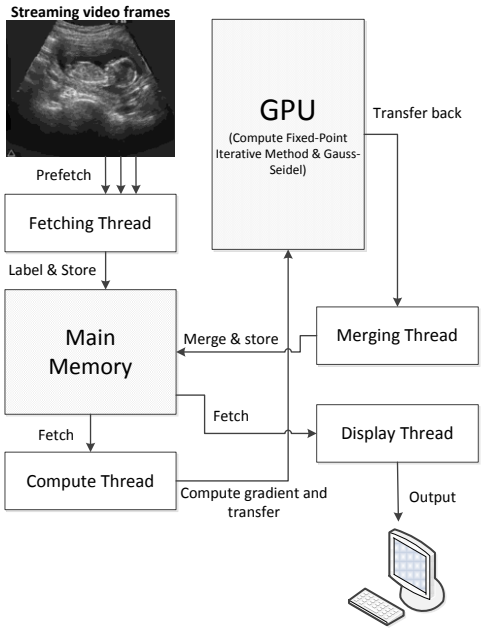


Figure 13. Our propose strategy for denoising image sequence in real-time

The explanation of the denoising process is presented step-by-step as follow:

1. *Fetching Thread* fetches and uses the prefetching technique by fetching the frames ahead, stores them into the main memory and labels them. Each frame's label contains (i) frame sequence, (ii) frame expiration time.

2. *Compute Thread* keeps fetched frames from the main memory, computes K(*u*) and transfers them to the GPU's memory, then invokes the kernel after the data transfer is finished.

3. Next, the GPU denoises the frame in Equation (7) by using the Fixed-point iterative method and Sliding Windows Gauss-Seidel [14].

4. After the GPU finishes denoising, the *Merging Thread* transfers the denoised frame back into the main memory, reads the frame label and then rearranges the denoised frame in the buffer in the main memory and discards the expired frames.

5. *Display Thread* displays the denoised output in real-time.

4. Experiment Results

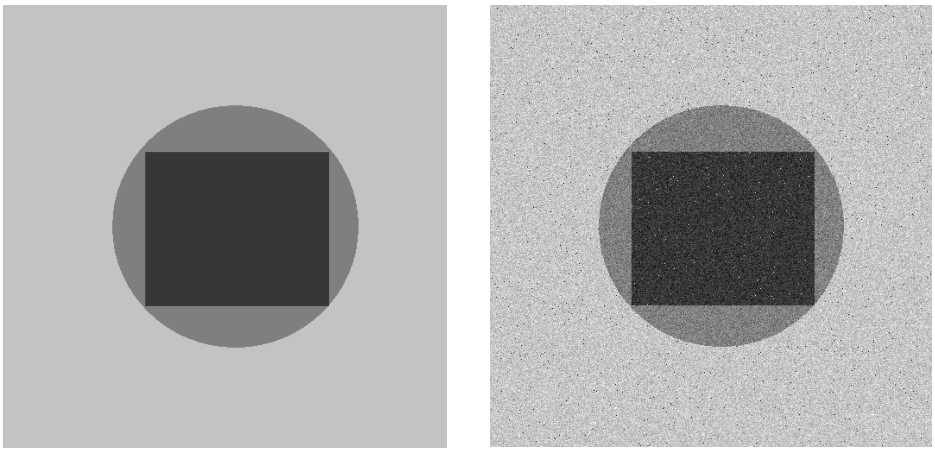
In this section, we divide our results into 3 parts. First, we validate our strategy and the noise removal algorithm. Next, we show the performance gain and the frame rate by using our strategy and finally, the results of some denoised frames are displayed.

The experiments were made on Intel® Core 2 Duo with 2.5 GHz of CPU and 4GB of main memory. The NVIDIA® GeForce GTX-480 with 480 CUDA cores and 1.5GB of the graphic memory. Each core runs at 750MHz.

We use 64-bit Fedora 17 Linux with GNU C Compiler (GCC) 4.7 with GNU debugger (gdb) enabled, OpenCV 2.3 for image and image sequence manipulation.

4.1. Denoising Model Validation

To ensure that the noise removal algorithm and our strategy can efficiently remove both additive and multiplicative noises, we use the sample image sequence and synthesize both additive and multiplicative noises into it. The first frame of sample image sequence is shown in Figure 14.

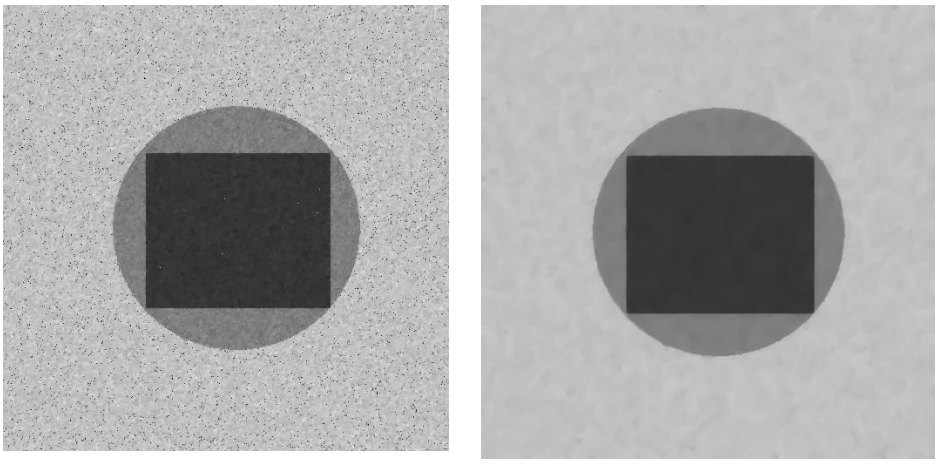


(a) (b)

Figure 14. An original sample picture for testing denoising algorithm (a) and a sample picture with synthesized noises (b)

The synthesized noisy frame has Peak Signal-to-Noise Ratio (PSNR) value of 19.77dB.

After denoising the image sequence with well-selected parameters, the first frame is denoised as shown in Figure 15(b).



(a) (b)

Figure 15. Denoised sample picture from JY Model [1] (a) and from model in [3] (b)

The PSNR value of image with the synthesized noise comparing with the original image is 85.64 while JY Model [1] denoised image PSNR value is 51.54dB. This shows the denoising model and our strategy outperforms the previous JY Model by removing both multiplicative and additive noises efficiently.

In denoising the streaming image sequence, the frame order is also checked to make sure that our strategy is working correctly.

4.2. Performance Results

First, we measure the average frame rate for the denoised image sequence as in Figure 16, while varying the number of GPU threads per frame.

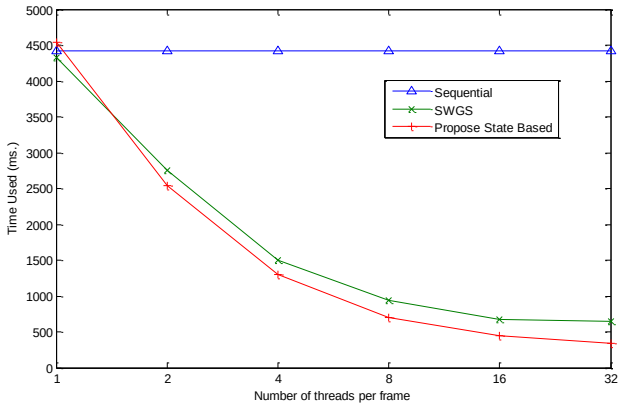


Figure 16. Time used per frame varying threads per frame

We define the average speedup as:

$$s_{avg} = \frac{t_{seq}}{t_{avg}} \qquad (10)$$

Where s_{avg} is the average speedup for each frame, t_{avg} is the average time for each frame including the discarded frames. The t_{seq} is the time used for denoising the frame sequentially. The average speedup varying the number of GPU threads per frame is shown in Figure 17.

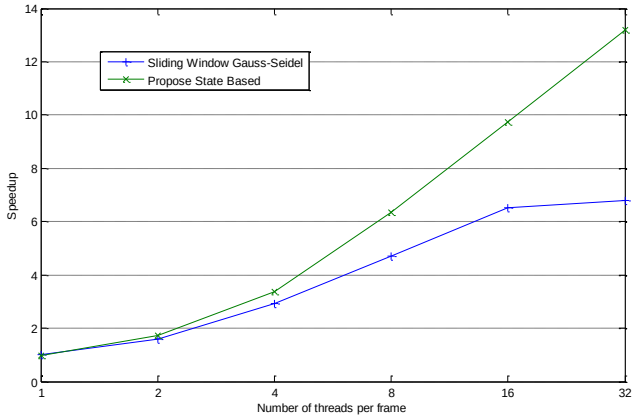


Figure 17. Speedup of proposed strategy varying thread per frame

We fix the frame rate of the output image sequence at 24 frames per second and measure the frame rate of the output varying the number of threads per frame without discarded frames as in Figure 18. Please note that only the entire image sequence average frame rates are shown in Figure 18.

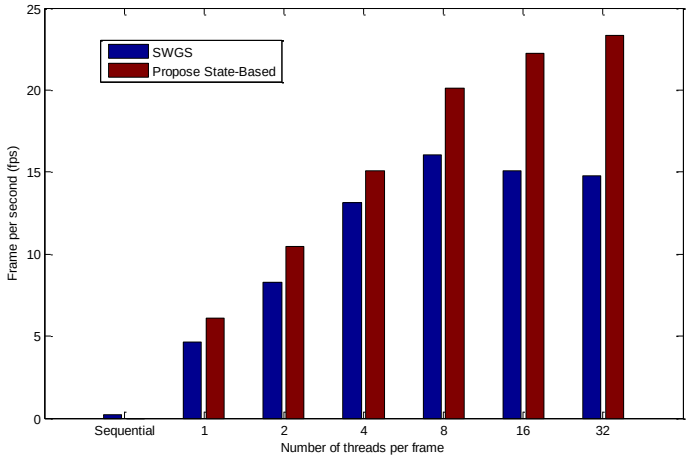


Figure 18. The average frame-per-second vary thread per frame

Figure 18 shows that after utilizing both multi-core CPUs and GPUs asynchronously at the same time, the number of frames per second for processing increased dramatically. This is because multiple kernels are launched and the pipeline latency in the traditional Sliding Window Gauss-Seidel is eliminated in our proposed state-based method. The increase of the number of threads per frame causes the increment of frame rate by allowing more GPU cores to work on the same frame in parallel.

The frame per second is satisfactory (~20-25 fps) when the number of threads per frame is ≥ 8, as we had described in the first section that the output image sequence should have the frame rate at around 12-15 fps.

We investigate the frame drop and between-frame-delay (BFD). The between-frame-latency shows how smooth the image sequence is. We also show the average frame drop per second, the average between-frame-delay and standard deviation of between-the frame-delay is shown in Table 1.

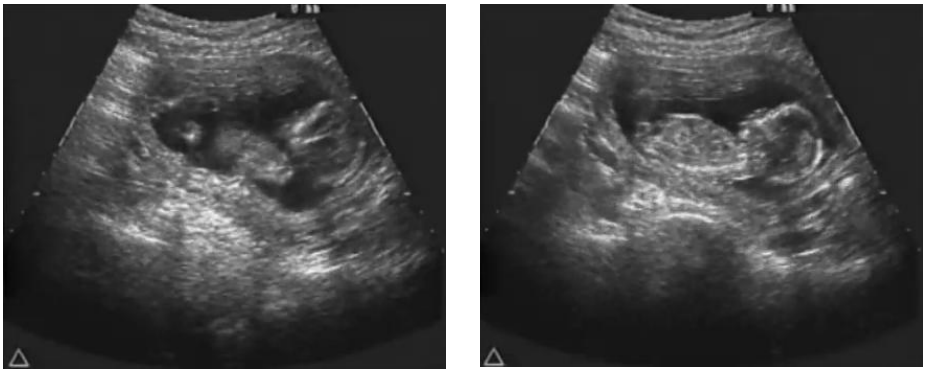
Table 1. Average frame drop per second, average and S.D. of between frame delay

Thread per frame	Avg. Frame Drop (fps)		Avg. BFD (ms.)		S.D. of BFD (ms.)	
	SWGS	Proposed	SWGS	Proposed	SWGS	Proposed
1	19.33	17.89	514	393	61.1	79.3
2	15.73	13.55	289	230	83.5	80.1
4	10.89	8.94	183	159	74.2	82.2
8	7.94	3.90	149	119	90.2	86.8
16	8.96	1.72	159	108	97.9	99.6
32	9.22	0.67	162	103	101.8	99.7

Table 1 shows that our proposed method can greatly decrease the average number of frame drop while the average values between-frame-delay is slightly improved. The output video seems smoother than the output that uses Sliding Window Gauss-Seidel. The frame drop is improved up to 92.73% on 32 threads per frame or 23.72% by average compared to Sliding Window Gauss-Seidel.

4.3. Denoised Image sequence Quality

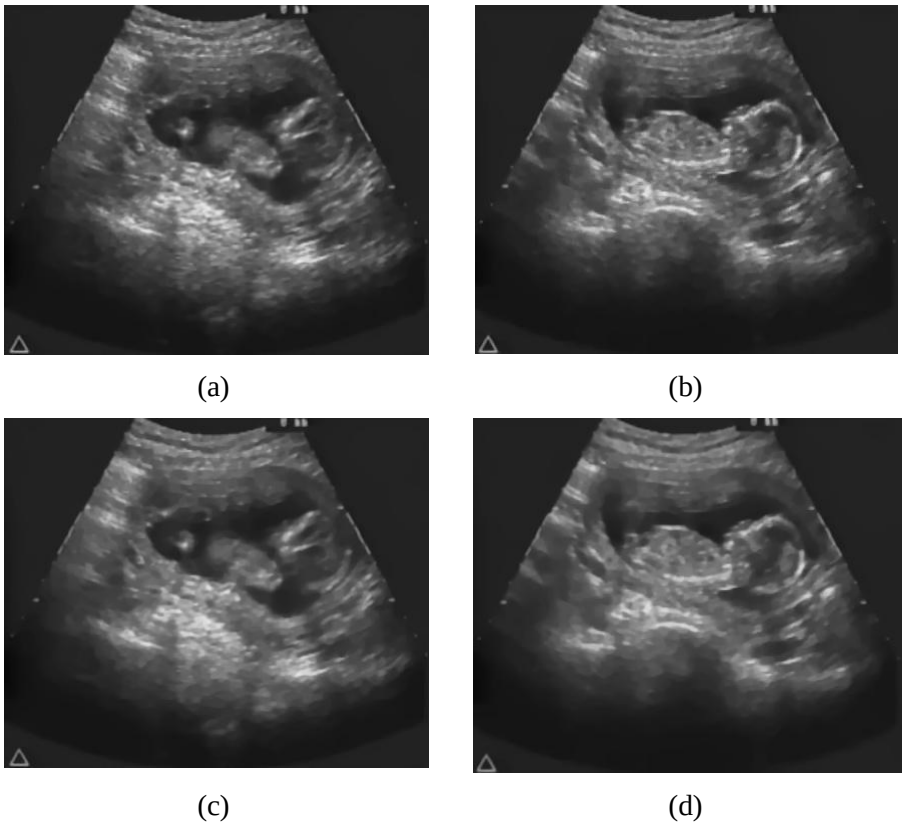
We use sample ultrasound image sequence from public domain image sequence achieve [15]. The image sequence resolution is 480×352 pixel and the frame rate is 29 frames per second. Each frame is converted into the 8-bit grayscale image in the denoising process. The 80^{th} and 300^{th} sample frames are shown in Figure 19.



(a) (b)

Figure 19. The 80^{th} (a) and 300^{th} (b) frames from sample ultrasound image sequence

The denoised 80^{th} and 300^{th} frames are shown in Figure 20.



(a) (b)

(c) (d)

Figure 20. Denoised frame of 80th and 300th frame from the sample ultrasound image sequence by using JY Model (a, b) and the new variation method (c, d)

The PSNR of denoised frame comparing with the noisy image sequence frame is in the range of 78.97 dB – 86.43dB.

5. Conclusion and Future Work

We propose the new thread management strategy and redesign the data structure and memory usage for the state-based Gauss-Seidel for denoising the ultrasound image sequence which contains both additive and multiplicative noises. The GPUs and multi-core processor are used to accelerate the computation to ensure satisfactory frame rate.

Our strategy uses the denoising model and improves image-sequence denoising strategy from [1]. The proposed strategy uses both multi-core advantages and reduces some overhead from the frame distribution while utilizing the GPUs efficiently.

Our results show that our strategy can achieve speedup per single frame computation up to 13.18 times compared to the sequential computation. The output image sequence frame rate is boosted up 106.04 times comparing to the sequential computation. Moreover, the denoising image sequence quality is visually satisfactory. This makes the real-time image sequence denoising possible. However, fine denoising parameter tuning is still essential in practice to make sure that the denoised picture is smooth and retains all necessary information in each frame.

Acknowledgements

This work is supported in part by the Thailand Research Fund through the Royal Golden Jubilee Ph.D. Program., contract no. PHD/0275/2551

References

[1] Z. Jin and X. Yang, "Analysis of a new variational model for multiplicative noise removal", Journal of Math. Anal. Appl., vol. 362, **(2010)**, pp. 415-426.
[2] B. Dolwithayakul, C. Chantrapornchai and N. Chumchob, "Two Parallel Strategies for Real-time Spatial Image sequence Denoising for Multi-core Processors", International Journal of Computer Applications, vol. 48, no. 16, **(2012)**, pp. 28-35.
[3] N. Chumchob, K. Chen and C. Brito-Loeza, "A new variational model for removal of combined additive and multiplicative noise and a fast algorithm for its numerical approximation", International Journal of Computer Mathematics, **(2012)**, pp. 1-12.
[4] K. Hirakawa and T. W. Parks, "Image Denoising usig total least squares", IEEE Trans. Image Process, vol. 15, no. 9, **(2006)**, pp. 2730-2742.
[5] R. Paul and M. Meyer, "Restoration of motion picture film,Conservation and Museology", Butterworth-Heinemann, **(2000)**.
[6] NVIDIA® Corporation, "NVIDIA CUDA compute unified device architecture programming guide version 2.1", **(2008)**.
[7] NVIDIA® Corporation, "NVIDIA CUDA™: NVIDIA C Programming Guide Version 4.2", **(2012)**.
[8] L. Rudin, S. Osher and E. Fatemi, "Nonlinear total variation based noise removal algorithms", Physica D., vol 60, **(1992)**, pp. 130-120.
[9] C. R. Vogel and M. E. Oman, "Iterative methods for total variation denoising", SIAM Journal of Sci. Comput., vol. 17, **(1996)**, pp. 227-238.
[10] C. R. Vogel and M. E. Oman, "Fast, Robust total variation-based reconstruction of noisy, blurred images", IEEE Transaction of Image Processing, vol. 7, **(1998)**, pp. 813-824.
[11] S. S. Al-amri, N. V. Kalyankar and S. D. Khamitkar, "A Comparative Study of Removal Noise from Remote Sensing Image", IJCSI International Journal of Computer Science, vol. 7, **(2010)**, pp. 32-36.
[12] Y. Iikura, "Estimation of noise component in satellite images and its application", Geoscience and Remote Sensing Symposium, **(1995)**, pp. 102-104.
[13] A. Marquina and S. Osher, "Explicit algorithms for a new time dependent model based on level set motion for nonlinear deblurring and noise removal", SIAM J. Sci Comput., vol. 22, pp. 387-405.
[14] H. Moghnieg and D. A. Lowther, "The solution of Electromagnetic Field Problems Using Sliding Windows Gauss-Seidel Algorithm on a Multicore Processor", IEEE Trans. Magnetic, vol. 46, **(2010)**, pp. 3081-3084.
[15] The Sunday Times Online, "Ultrasound Fetal Response to Alcohol Fetal Syndrome", **(2005)** November.
[16] B. Dolwithayakul, C. Chantrapornchai and N. Chumchob, "An efficient asynchronous approach for Gauss-Seidel iterative solver for FDM/FEM equations on multi-core processors", 2012 International Joint Conference onComputer Science and Software Engineering (JCSSE), **(2012)**, pp. 357-361.

Authors

Banpot Dolwithayakul

He received his bachelor's degree of Computer Science (B. Sc.) in 2006 from Silpakorn University, Nakhon-Pathom, Thailand. In 2009, he received M.Sc. in computer science from the same place. Currently, he's working in the Department of Software Development, Computer Centre, Silpakorn University and he is also a Ph.D. student of the Department of Computing, Faculty of Science, Silpakorn University. His research interests are high performance computing, computational science, image restoration and parallel computing.

Chantana Chantrapornchai

She obtained her Bachelor degree from Thammasat University of Thailand in 1991. Her major was Computer Science. She graduated from Northeastern University at Boston, College of Computer Science, U.S.A. in 1993 and University of Notre Dame, Department of Computer Science and Engineering, U.S.A., in 1999, for her Master and Ph.D. degrees respectively. She is an associated professor of Dept. of Computing, Faculty of Science, Silpakorn University, Thailand. She currently is a lecturer at Department of Computer Engineering, Faculty of Engineering, Kasetsart University, Thailand. Her research interest includes parallel and distributed systems, embedded real-time systems, wireless and mobile computing, data mining and artificial intelligence for architecture design.

Noppadol Chumchob

He received his B.Sc. and M.Sc. in mathematics, and Ph.D. in mathematical sciences respectively, from Thaksin University, Thailand, Silpakorn University, Thailand, and the University of Liverpool, U.K. He is currently a lecturer at the Department of Mathematics, Faculty of Science, Silpakorn University, Nakhon-Pathom, Thailand and a member of the Centre for Mathematical Imaging Techniques (CMIT), Department of Mathematical Sciences, The University of Liverpool, U.K. He is a computational mathematician specializing in developing novel mathematical models and fast numerical algorithms for various scientific applications. His research interests are currently centered on developing effective variational models and efficient numerical solvers for image processing applications, including image restoration and registration.