Elsevier Editorial System(tm) for Signal Processing
Manuscript Draft

Manuscript Number: SIGPRO-D-08-00879R1

Title: Parallel Implementation of Wavelet-based Image Denoising on Programmable PC-grade Graphics Hardware

Article Type: Special Issue: High-Dimensional Masses

Keywords: Discrete Wavelet Transform, Image denoising, Graphics accelerator, General-purpose Computing on Graphics Processing Unit

Corresponding Author: Dr Zhijie Xu, PhD

Corresponding Author's Institution: University of Huddersfield

First Author: Yang Su, Master of Computing and Communication

Order of Authors: Yang Su, Master of Computing and Communication; Zhijie Xu, PhD

Abstract: The intensive computation of Discrete Wavelet Transform (DWT) due to its inherent multilevel data decomposition and reconstruction operations brings a bottleneck that drastically reduces its performance and implementations for real-time applications when facing large size digital images and/or high-definition videos. Although various software-based acceleration solutions, such as the lifting scheme, have been devised and achieved a higher performance in general, the pure software accelerated DWT still struggle to cope with the demands from real-time and interactive applications. With the growing capacity and popularity of graphics hardware, personal computers (PCs) nowadays are often equipped with programmable Graphics Processing Units (GPUs) for graphics acceleration. The GPU offers a cost-effective parallel data processing mechanism for operations on large amount of data, even for applications beyond graphics. This practice is commonly referred as General-purpose Computing on GPU (GPGPU). This paper presented a GPGPU framework with the corresponding parallel computing solution for wavelet-based image denoising by using off-the-shelf consumer-grade programmable GPUs. This framework can be readily incorporated with different forms of DWT by customising the parameter of the wavelet kernel. Experiment results show that the framework gains applicability in data parallelism and satisfaction performance in accelerating computations for wavelet-based denoising.

Parallel Implementation of Wavelet-based Image Denoising on Programmable PC-grade Graphics Hardware

Yang Su[a,b], Zhijie Xu[a,*]

[a]School of Computing & Engineering, University of Huddersfield
Queensgate, Huddersfield HD1 3DH, UK

[b]School of Communication & Information Engineering, Xi'an University of Science & Technology, Xi An 710054, China

Abstract — The Discrete Wavelet Transform (DWT) has been extensively used for image compression and denoising in the areas of image processing and computer vision. However, the intensive computation of DWT due to its inherent multilevel data decomposition and reconstruction operations brings a bottleneck that drastically reduces its performance and implementations for real-time applications when facing large size digital images and/or high-definition videos. Although various software-based acceleration solutions, such as the lifting scheme, have been devised and achieved a higher performance in general, the pure software accelerated DWT still struggle to cope with the demands from real-time and interactive applications. With the growing capacity and popularity of graphics hardware, personal computers (PCs) nowadays are often equipped with programmable Graphics Processing Units (GPUs) for graphics acceleration. The GPU offers a cost-effective parallel data processing mechanism for operations on large amount of data, even for applications beyond graphics. This practice is commonly referred as General-purpose Computing on GPU (GPGPU). This paper presented a GPGPU framework with the corresponding parallel computing solution for wavelet-based image denoising by using off-the-shelf consumer-grade programmable GPUs. This framework can be readily incorporated with different forms of DWT by customising the parameter of the wavelet kernel. Experiment results show that the framework gains applicability in data parallelism and satisfaction performance in accelerating computations for wavelet-based denoising.

Keywords: Discrete Wavelet Transform; Image denoising; Graphics accelerator; General-purpose Computing on Graphics Processing Unit

[*] Corresponding author. Email: z.xu@hud.ac.uk, Tel: +44(0) 1484 472156.

1. Introduction

The wavelet transforms are usually classified into two categories, Continuous Wavelet Transform (CWT) and Discrete Wavelet Transform (DWT). It has become an important tool in image denoising due to its abilities in obtaining multi-resolution analysis results with localized features both in the frequency and the time domain. When the DWT applied in image denoising, implementation involves the following three processing phases [1]:

1) Decomposition
   Select a suitable base wavelet and a decomposition level to generate the approximation and detail coefficients of a noisy image at the chosen level.
2) Thresholding
   For each level, to generate a threshold and apply it through hard/soft thresholding to the detail coefficients.
3) Reconstruction
   Compute for reconstructions using the modified coefficients of various levels.

Various kind of base wavelet, such as Haar, Daubechies, CDF biorthogonal, Coiflet, Daubechies' Symlet, can be employed by the above procedures [2]. Since the thresholding strategy directly determines the quality of wavelet-based denoising, some methods have been proposed for improvement on performance by Birgé and Massart [3, 4], Donoho and Johnstone [5,6,7], and many others [8,9]. For example, the Birgé-Massart strategy, in which the numbers of detail coefficients are kept for the process of reconstruction, is dependent on the decomposition level and the length of the coarsest approximation coefficients of the noisy signal [3]. In comparison, Donoho and Johnstone devised an adaptive thresholding procedure, named as SureShrink, for adapting to unknown smoothness via wavelet shrinkage [5]. In the SureShrink process, a threshold level is assigned to each dyadic resolution level by the principle of minimizing the Stein Unbiased Estimate of Risk (SURE) for the threshold estimates. In addition, Donoho and Johnstone also developed a minimax model for nonlinear estimation of noisy data in wavelet domain [6]. Donoho and Kerkyacharian further developed a universal threshold strategy in which the threshold determination is related to the signal length and the noise standard deviation. For a two dimensional signal such as an image, this method can be extended to incorporate to the image size [7].

Although a rich and advanced body of work on wavelet-based denoising theories evolved in the last decade, in contrast, the practical implementation of their counterparts on computers have limited success so far due to the computational intractability caused by the large amount of computation brought by the transform. It is well acknowledged that the intensive computation of DWT through multilevel decomposition and reconstruction will often introduce serious computational bottlenecks, especially when the data size is large. To solve this problem, Sweldens [10] proposed an accelerated implementation method of DWT, known as the lifting scheme through reusing the intermediate values from previous calculation steps. The lifting scheme achieves a higher performance than conventional filter bank scheme (FBS). However, its pure software realization is still facing the harsh challenges from dealing with large data sets and to achieve real-time or interactive rate performance. Other solutions through employing hardware accelerators, such as field programmable gate array (FPGA) and very large scale integration (VLSI) were proposed by some researchers [11,12]. Unfortunately, these implementations require extra computer hardware and accessories which are often costly and difficult to set up.

In recent years, consumer-grade graphics processing unit (GPUs) – initially designed for computer game enthusiasts -- have evolved into powerful parallel processors with various degree of programmability and precisions [13]. Except for graphical intensive computation through specially designed data formats and process flows, other more general purpose computing using GPU (GPGPU) have also been attempted ranging from numeric computing operations such as dense and sparse matrix transform [14], solving partial differential equations, linear algebraic operations [15], to physical simulations such as fluid mechanics solvers, as well as signal processing through fast Fourier Transform (FFT) [16] and DWT [17,18,19].

In this paper, we propose a GPGPU framework for supporting wavelet-based denoising and process acceleration. Through careful balancing, most of the DWT computations are performed on GPU rather than CPU. Hence it improves the efficiency when facing large data streams, such as from high resolution digital images or high definition videos. Furthermore, this framework is so designed to support wavelet-based denoising that employs different forms of DWT and thresholding strategies through using GPU textures for updating the input parameters. The flexibility and effectiveness of the proposed framework are tested and evaluated by comparing with the performance measured from software based solutions.

2 GPGPU Review

General Purpose Computation on GPUs (GPGPU) as a reincarnated concept can be traced back to the early 1990s when the Pixel Machine was of the state-of-the-art. However, it was not till 2002 when consumer-grade graphics cards became truly "programmable", the concept was becoming widely accepted. Almost all of today's commodity GPUs and their computational flows follow a similar infrastructure called the graphics pipeline which is depicted in Fig.1.

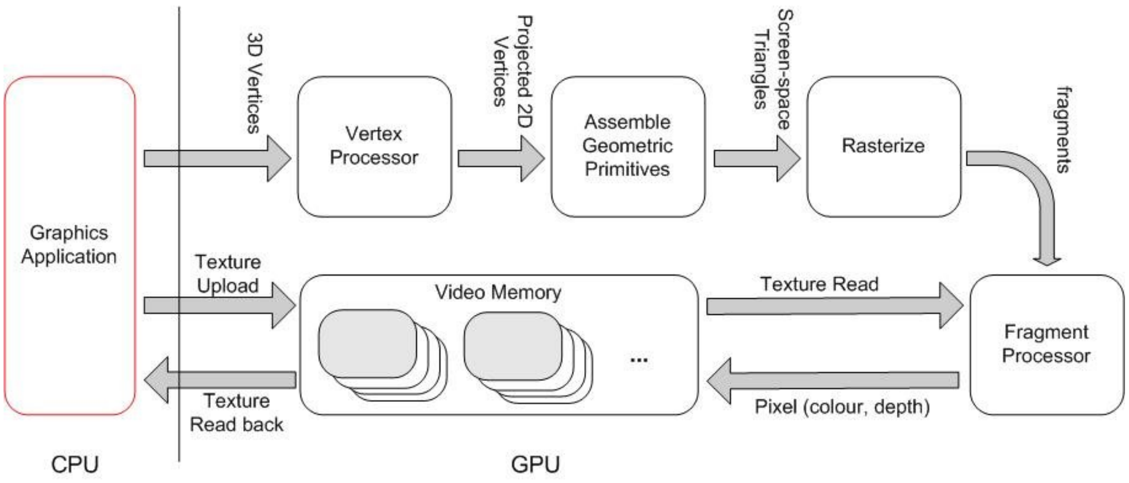


Fig.1 Overview of the 3D Graphics Pipeline.

The inputs of this pipeline are vertices from a 3D polygonal mesh defined by their spatial and appearance information such as coordinates, colors, and texture mapping values, and the output is a 2D array of colored pixels to be displayed on the screen. The process of the pipeline mainly consists of three stages that are vertex processing, rasterization, and fragment processing [20]. In chip design and component layout, each stage is implemented as a separate piece of hardware on the GPU card in a so-called task-parallel machine organization. This hardware structure was historically a fixed-function pipeline (FFP), where limited numbers of operations available at each stage of the graphics pipeline were hardwired for specific tasks. In the last decade, major graphics vendors such as ATi and Nvidia have transformed the fixed-function pipeline into a more flexible programmable one. This effort has been primarily concentrated on two stages of the pipeline: the vertex processing and the fragment processing. The fixed-function operations in these two stages are replaced by the user-defined vertex program and fragment program respectively [21]. Furthermore, the support inherited from existing offline rendering systems, especially the introduction of high level shading languages, has made the general-purpose computing on GPU an accessible and cost-effective platform for application developers, in which GPU acts as a parallel processor that adopts single instruction multiple data (SIMD) architecture to provide data parallelism [22]. The latest development trend driven by Microsoft's Direct3D10 and unified shading language such as CUDA is trying to insulate GPGPU developers from the distinctive vertex, pixel or even the geometry shading stages.

The data parallelism offered by the modern GPUs stems back to the early super-computers equipped by the SIMD technology. The SIMD architecture distinguishes GPGPU's programming paradigm from the traditional sequential programming model of CPUs. In the common GPGPU framework, the fully programmable vertex and fragment processors perform the roles of the computational kernels while the video memory (i.e., frame-buffers, textures) provides runtime data access services. An operation referred as texture mapping on the GPU is analogous to the random read-only memory interface on the CPUs, while the ability to render directly into texture (off-screen rendering) available on most modern GPUs provides a memory-write mechanism. However, by default of its specialized design, commodity GPU has a more restricted memory model when compared to a CPU (i.e. random memory write is not allowed). In addition, texture memory caches on GPUs are designed for access speed, and general prohibit concurrent read and write into the same memory address. Thus distinctive read and write texture phases must be applied so they can be swapped after each rendering pass in a so-called ping-pong mode.

In order to implement an algorithm on a GPU, different computational steps are often manually mapped to various vertex and fragment programs as a common practice prior to the emerging of unified GPU languages such as CUDA from Nvidia. For each computational step, the appropriate vertex or fragment program are bound to the corresponding processors and invoked by various rendering operations. The rasterization engine on GPU generates a stream of fragments and can also provide a fast way of interpolating numbers. Most GPGPU applications execute multiple vertex and/or fragment programs in a series of successive off-screen rendering passes. Some specialized render-to-texture schemes, such as the pixel-buffers (pBuffers) [23] and the framebuffer objects (FBOs) [24] were introduced by GPU vendors to provide a simple and efficient off-screen rendering mechanism. Further details about GPGPU programming techniques and know-hows were discussed in Pharr's text published in 2005 [25].

Many matured and widely applied algorithms for image processing map well into GPU's parallel stream processing model and hence opening a new front for real-time image or even video processing. Image processing tasks which can be applied on multiple pixels simultaneously (eg. convolution) can be performed efficiently by fragment programs through exploiting the parallelism provided by multiple fragment streams. Since 2002, there has been a growing interest in the image processing community to solve important and computationally expensive imaging problems using this new found computer power, for example, using wavelet transform for image compression. Hopf and Ertl at the University of Stuttgart in Germany, first implemented a 2D-DWT on graphics hardware [17]. Wong at the Chinese University of Hong Kong also developed a GPU implementation for a 2D-DWT, which has been integrated into an open-source JPEG2000 codec called "JasPer" [18]. These pilot projects experienced various degree of success but were all restricted by the functions of the graphics hardware and shading languages at the time. This drawback will be further discussed in Section 4 following the review of wavelet-based denoising in the next section.

3 Wavelet-based signal denoising

3.1 Analysis of wavelet transform

Wavelet transform is used to construct a time-frequency representation of a signal, which offers excellent time and frequency localization. For a continuous, square-integrable function *f*(*t*), its continuous wavelet transform (CWT) is defined as the sum over all time of the signal multiplied by scaled, shifted versions of the wavelet function *ψ*:

$$C(scale, translation) = \int_{-\infty}^{+\infty} f(t)\psi(scale, translation, t)dt \qquad (1)$$

The results of the CWT are a series of wavelet coefficients, which are functions of scale and translation. In mathematical terms, the accurate definition of CWT on *f*(*t*) at a scale *a*>0 and a translation value *b* (where *b* is a real number) is expressed by the following integral

$$C_W(a,b) = \frac{1}{\sqrt{a}}\int_{-\infty}^{+\infty} f(t)\psi^*(\frac{t-b}{a})dt \qquad (2)$$

, where ψ* represents complex conjugation of *ψ*.

To recover the original signal *f*(*t*) from reserved wavelet coefficients, the inverse CWT can be exploited as

$$f(t) = \int_{0}^{+\infty}\int_{-\infty}^{+\infty} \frac{1}{a^2} C_W(a,b)\frac{1}{\sqrt{|a|}}\varphi(\frac{t-b}{a})dbda \qquad (3)$$

, where *φ* (*t*) is the dual function of *ψ*(*t*).

CWTs operate on every possible scale and translation over a signal spectrum. However, the calculation of coefficients at every scale and translation is a substantial work that often generates a huge amount of data, In addition, any signal processing operations performed on a computer using real-world data must be carried out on a discrete signal that is, a signal measured at discrete times. The discrete wavelet transform (DWT) uses a specific subset of scale and translation values where the chosen scale and translation are based on the powers of two, which is the so-called dyadic scales and translations. In this case, the wavelet analysis is much more efficient but just as accurate. When it is implemented, the DWT of *f*(*t*) is calculated by passing *f*(*k*) that is the discrete expression of *f*(*t*) through a series of low-pass (LP) and high-pass (HP) filters respectively. The bandwidth of the filter outputs are half the bandwidth of the input signal, which allows the downsampling of the output signals by the factor of two without losing any information according to the Nyquist theorem. The downsampled signals from the LP and HP filters are referred to as the approximation and the detail coefficients, respectively. The prior is the high-scale and low-frequency component of the signal, and the latter is the low-scale and high-frequency component. The decomposition process can be iterated, with successive approximation coefficients being generated in turn so that a signal can be broken down into many lower resolution components.

The inverse discrete wavelet transform (IDWT) is used for reconstructing the original signal. It involves two distinctive operations of upsampling and filtering. Upsampling is the process of lengthening a signal component by inserting zeros between samples. The filtering part of the reconstruction process also consists of a series of LP and HP filters which are associated with the decomposition filters in DWT. These form a system of what is called the quadrature mirror filters to guarantee reproducing the original signal accurately. Fig. 2 illustrates a multi-level DWT and IDWT of a signal with bandwidth *F*.

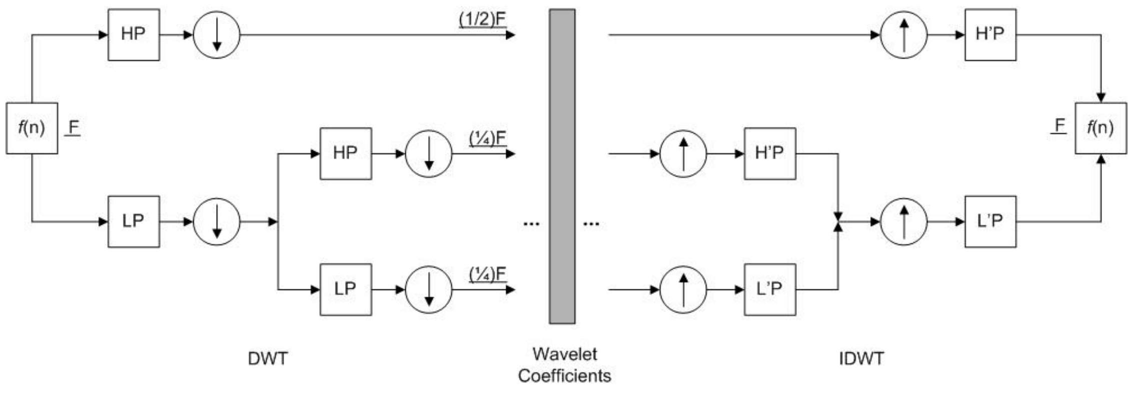


Fig.2 Multi-level DWT and IDWT

A noisy signal *f*(*k*) is commonly modeled as the following form:

$$f(k) = s(k) + e(k) \qquad (4)$$

where *s*(*k*) is the helpful one which is often a low frequency or stationary component in the practical implementation. *e*(*k*) is the actual noise, which is usually of a high frequency domain that contains high frequency details. As stated by [1], the general wavelet denoising procedure consists of three steps, forward transformation of the signal to the wavelet domain, modifying the wavelet coefficients, and inverse transformation to the native domain. The following section discusses the thresholding strategy and other related issues.

3.2 Thresholding strategy

As widely cited by many publications in various application domains, the most practical thresholding methods were mainly initiated by the work of Birgé and Massart [3,4], and Donoho and Johnstone [5,6,7].

Based on the work of Birgé and Massart, the thresholding methods used in practice can be classified into the following two categories:

- Scarce High, Medium, and Low (SHML)
- Penalized High, Medium, and Low (PHML)

The SHML methods work as the following: for a noisy signal that is decomposed to a level *J*, the approximation coefficients at level *J* are kept; for a random level *i* from 1 to *J*, the n_i largest coefficients are kept in the form stated as formula (5).

$$n_i = \frac{M}{(J+2-i)^a} \qquad (5)$$

In the above equation, the value of parameter *a* and *M* are determined by the practical applications. The SHML methods can be further classified by the value of parameter *a*.

For the PHML, a threshold *T* applied to the detail coefficients for the wavelet case can be generalized as:

$$T = |c(t^*)| \qquad (6)$$

with

$$t^* = \arg\min[-sum\{c^2(k), k < t\} + 2vt(a + \log(\frac{n}{t})); t = 1,...,n] \qquad (7)$$

In equation (6) and (7), $c(\cdot)$ is all the detail coefficients of DWT, the coefficients $c(k)$ are sorted in a decreasing order of their absolute values, where *v* is the noise variance. The value of *a* that corresponds to the method of PHML are in the range of $2.5 \leq a < 10$, $1.5 < a < 2.5$, and $1 < a < 2$ respectively.

Regarding the issue of denoising, Donoho and Johnstone have devised four different thresholding options [5,6,7]:

1. *Rigrsure*

   Rigrsure is an adaptive threshold selection approach using the Stein's unbiased risk estimate criterion. The Rigrsure method defines the threshold level T by

   $$T = \sigma\sqrt{2\log_e(N\log_2 N)} \qquad (8)$$

   Where N is the number of signal samples; and σ is the standard deviation of the noise.

2. *Sqtwolog*

   The Sqtwolog method defines the universal threshold slightly different from the Rigrsure method in a fixed form

   $$T = \sigma\sqrt{2\log_e(N)} \qquad (9)$$

3. *Heursure*

   Heursure is a synthesis version of the aforementioned two rules resulting in an optimal forecasting variable threshold.

4. *Minimaxi*

   Minimaxi is a threshold selection scheme using the minimax principle, in which a fixed threshold is selected to obtain the minimum of the maximum mean square error that is obtained for the worst function in a given set, when compared against an ideal procedure.

All the above thresholding criteria is based on a simplified model that suppose a noise is a Gaussian white noise with standard deviation σ =1. For the general cases that noises are unscaled or nonwhite ones, the threshold level should be rescaled according to the aforementioned thresholding criteria. The actual level is commonly obtained by multiplying a rescaling factor by the thresholding value determined by the *Sqtwolog* method. Two rescaling options have been proposed. The first one is to rescale the noise based on coefficients in the first level of the wavelet decomposition. In this option, Daubechies (Db) 1 wavelet is used to obtain the detail coefficients of decomposition level

1, then the rescaling factor is made to equal to the median values of all absolute values of the detail coefficients. If the median absolute value is equal to 0, the actual threshold value T_s is expressed as:

$$T_s = 0.05 \times max(abs(c)) \qquad (10)$$

where $abs(c)$ represents a set of absolute values of detail coefficients at decomposition level 1 of the Db1 wavelet. The first rescaling option then treats the T_s as a global rescaling factor for the whole reconstruction. The second rescaling option, which is best used for nonwhite noise, determines different rescaling factors at various reconstruction levels.

In fact, there are a variety of noises in practical engineering and computer science applications. It is almost impossible to adopt a uniform thresholding strategy to achieve the best performance of denoising for all applications when facing noises with various characteristics. Actually, there are many other thresholding methods specially designed to deal with various forms of noise in specific fields. The performance evaluation of different denoising methods are often carried out by means of Mean Square Error(MSE), Signal to Noise Ratio (SNR), and Peak Signal-to-Noise Ratio (PSNR) with many past publications being focused on.

Except the aforementioned precision performance evaluation measures, another vital but often omitted factor also determines the perspective of successful implementation – computational cost. Extremely high computational cost (slow process and long delay to users) will constrain the application of denoising methods that demand a large pool of computer resources. This problem can become very serious when wavelet-based denoising are used for large size noisy images or high-definition videos, for example, satellite image processing and real-time surveillance video processing, or even Augmented Reality applications, in which enormous number of pixels need to be processed in a fraction of a second. In this research, a hardware accelerated solution for wavelet-based denoising has been proposed for alleviating the problem of computational cost and process speed.

4. Wavelet-based denoising on GPU

The amount of computation of wavelet-based denoising are mainly originated from the recursive operations of wavelet decomposition and reconstruction. With the constant increasing horse-power of commodity GPUs, extensive researches on implementing DWT on GPU have been carried out for image processing. The most relevant contributions are works from Hopf and Ertl [17] at the University of Stuttgart in Germany, and Wong at the Chinese University of Hong Kong [18]. Hopf and Ertl developed an OpenGL-based model of the filter bank scheme (FBS) for implementing DWT on a Silicon Graphics workstation by using high-level OpenGL routines, for example, OpenGL convolution filters. The project had experienced a degree of success on process acceleration. However, the solution has no direct mapping on hardware, which limits the efficiency of the implementation with some of the GPU resources left to spare. For the works of Wong, the convolution, downsampling, and upsampling operations were performed in sequence on a GPU's fragment processors (FPs). Due to the restrictions on GPU programmability at the time and coding facilities, the texture mapping prior to the convolution process was issued by establishing texture lookup tables in which every single texture coordinate is pre-defined in advance by separate CPU programs. The potential benefit of hardware-driven acceleration by using the GPU's hardware interpolators for generating texture coordinates and texture fetch were not fully exploited, and in turn hampers the performance of the consequent FP programs.

Based on the existing GPU-based denoising work, the GPGPU framework proposed in this research aimed at seeking further hardware empowered process acceleration for wavelet-based denoising through directly implementing texture fetching using hardware interpolators. When issuing filtering, kernels for downsampling and upsampling in the stages of decomposition and reconstruction, there is no need to employ any pre-defined values issued by separate CPU routines in advance. Furthermore, filtering and down-sampling operations can be carried out on GPU simultaneously, for instance, to implement the two operations on a single FP to exploit the performance gain from GPU's intrinsic functions.

However, against media over-exposure and many confident proclaims, GPGPU is hardly a computational panacea. There are still many issues regarding the hardware structure and programming paradigm to be tackled before a proper match against its CPU counterparts becoming a reality. In this project, the task partitioning in the proposed framework that decides which part of the work will be conducted on the GPU and which part should be left to the CPU for the current generation of hardware will also be discussed in the remaining sections.

4.1 Implementation strategies

The GPGPU framework for wavelet-based denoising developed in this project has synthesized OpenGL graphics library and the C for Graphics (Cg) shading language from Nvidia for processing 2D signals such as digital images. In the framework, a 2D-DWT was implemented by applying separate 1D-DWTs along the horizontal and vertical directions respectively. The decomposition process has adopted the common square decomposition method which is depicted as in Fig.3, where cA_j, cH_j, cV_j , and cD_j represent approximation coefficients (cA_0 represents original 2D signal), and the detail coefficients along horizontal, vertical, and diagonal orientations.

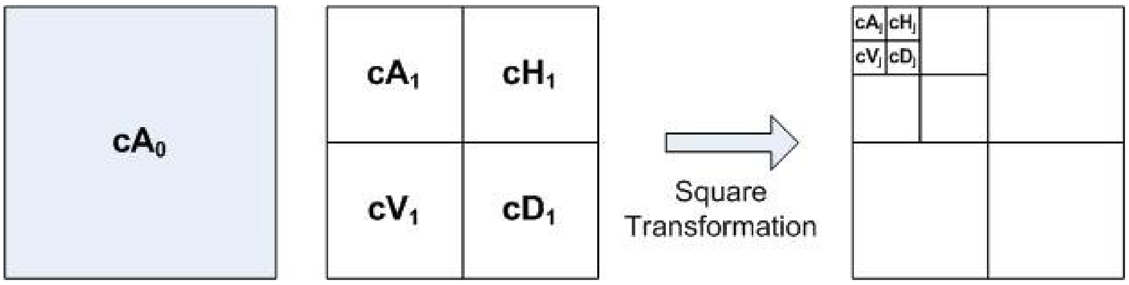


Fig.3 The square decomposition scheme

The thresholding approach chosen in this project has employed *Sqtwolog* method introduced in Section 3.2 to integrate with the global rescale options. As discussed earlier, the global rescaling factor is normally determined by the median absolute values of the detail coefficients obtained by the Db1 wavelet process, in which a sort operation on the absolute values of detail coefficients is essential. The sort operation requires random memory write accessibility, which is often not available from fragment processors on today's GPU in the so-called "scatter" memory operations (i.e., indexed-write array operations). The GPGPU framework devised in this research then assigned the task of thresholding to a CPU while concentrating GPU resources on issuing the operations of decomposition and reconstruction solely on the GPU. The entire framework can be summarized as in Fig.4.

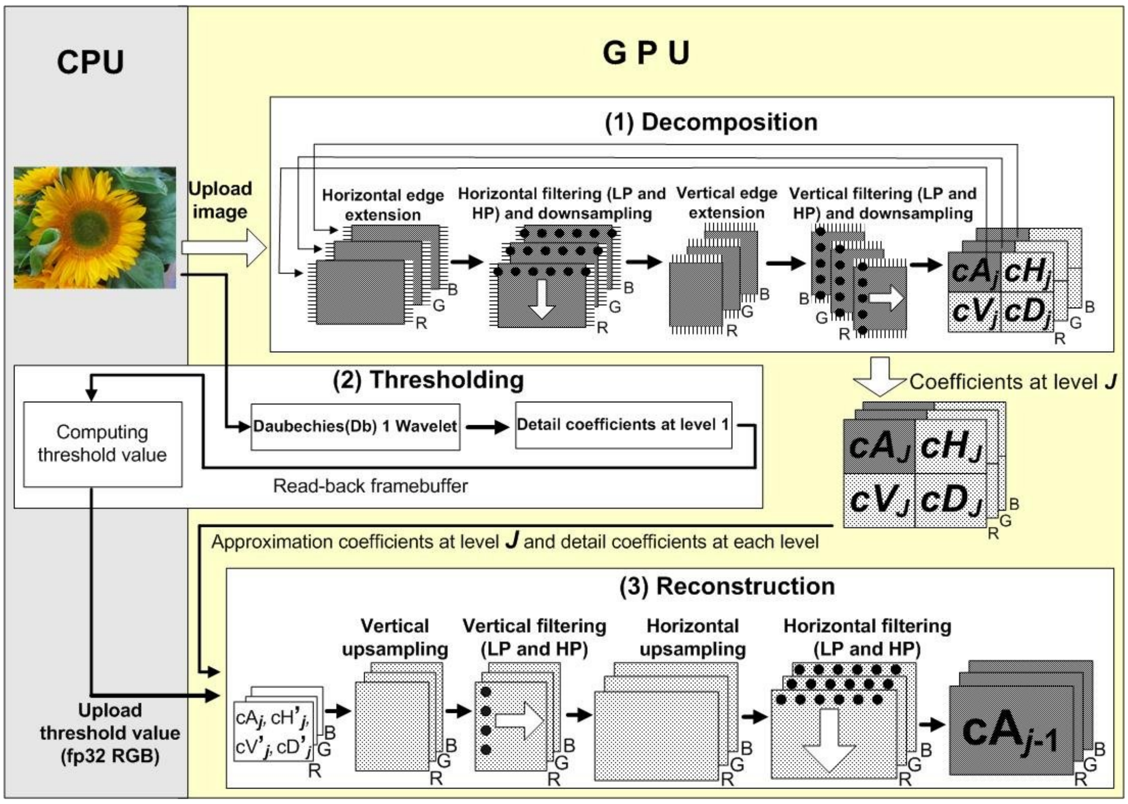

CPU
GPU
(1) Decomposition
Upload image
Horizontal edge extension
Horizontal filtering (LP and HP) and downsampling
Vertical edge extension
Vertical filtering (LP and HP) and downsampling
cAj cHj cVj cDj
Coefficients at level J
(2) Thresholding
Computing threshold value
Daubechies(Db) 1 Wavelet
Detail coefficients at level 1
Read-back framebuffer
cAJ cHJ cVJ cDJ
Approximation coefficients at level J and detail coefficients at each level
(3) Reconstruction
Vertical upsampling
Vertical filtering (LP and HP)
Horizontal upsampling
Horizontal filtering (LP and HP)
cAj, cH'j, cV'j, cD'j
cAj-1
Upload threshold value (fp32 RGB)
R G B

Fig.4 Overview of the framework of the GPGPU-based wavelet-based denoising

The following section will discuss the flows and processes hosted by the structure.

4.2 Technical notes of the GPU implementation

As a standard practice for GPU-based operations, the Red-Green-Blue-Alpha (RGBA) floating point vectors were used for storing pixels of an image. All the approximation coefficients (cA_j) and the detail coefficients (cH_j, cV_j, cD_j) obtained by deploying the same base wavelet were also stored in the same texture with RGBA four channels. A back buffer technique – the Framebuffer Objects (FBOs) – was employed as an off-screen rendering mechanism for storing intermediate computation results.

4.2.1 Decomposition

There are three main steps concerning decomposition being integrated into the framework including image edge extension, filtering and sampling. After investigating common extension schemes that include periodic padding, symmetric padding, and zero padding summarized by Strang and Nguyen[26], this research has applied the symmetrical periodic extension for its simplicity as shown in Fig.5. In the diagram the extension length *L* is determined by the kernel length of a filter employ in decomposition.

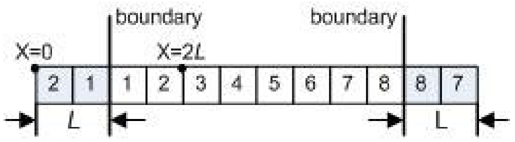


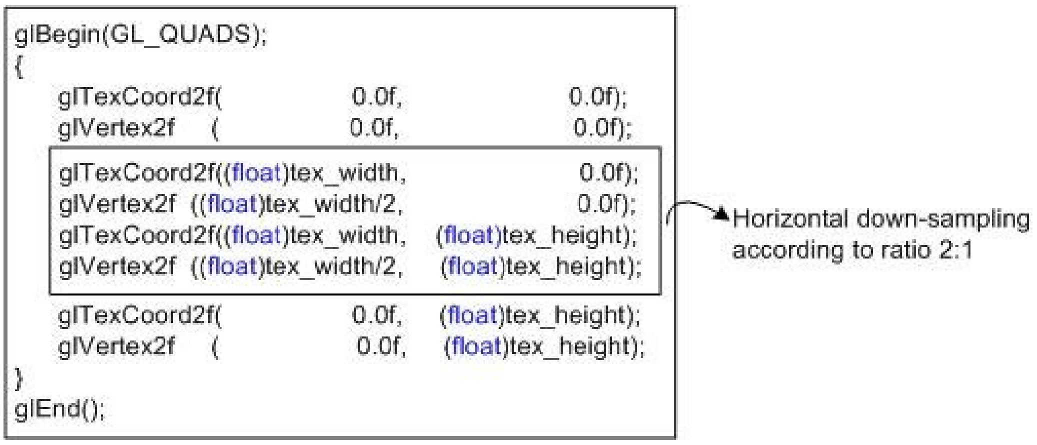
Fig.5 The symmetrical periodic extension scheme

Fig.6 shows a GPU program snippet for extending the left edge of an image on a GPU. The extended edge consists of the part outside the left boundary as indicated in Fig.5. The computational area is specified by an intrinsic OpenGL instruction *glBegin*(*GL_Quads*) for defining an off-screen quad canvas with specified vertex coordinates. The left edge extension was then issued with the following fragment program (FP).

```
fragout_float main(vf30 IN,
              uniform samplerRECT image,  //image texture
              uniform float L     //extension length)
{
  fragout_float OUT;
  OUT.col =f4texRECT(image, float2(2L-IN.TEX0.x, IN.TEX0.y));
  OUT.col.a=0.0;
  return OUT;
}
```

Fig.6 FP for edge extension

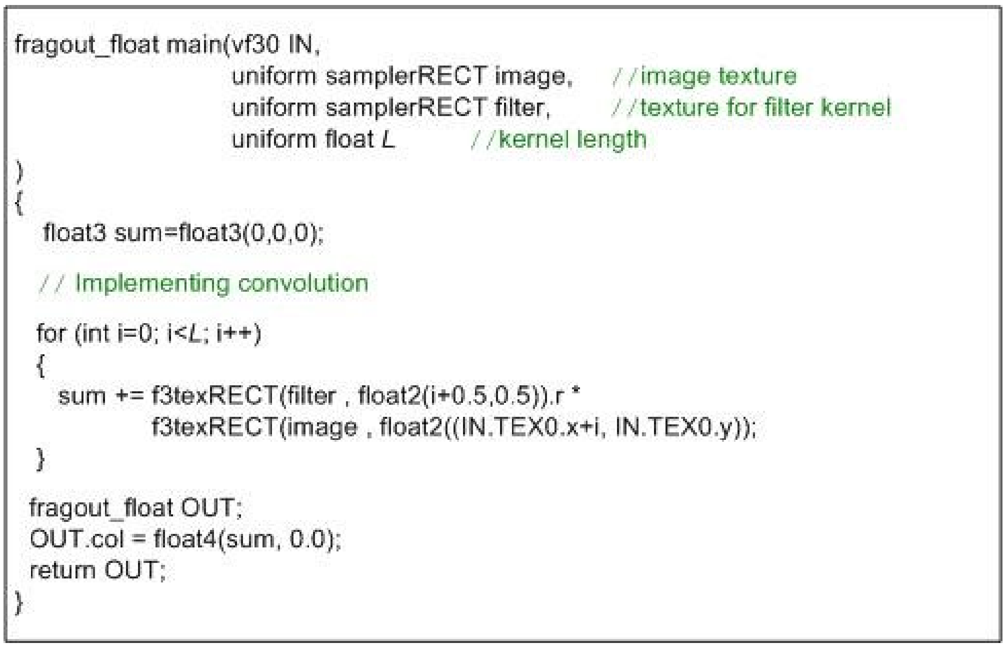
Two separable 1D-DWTs were issued following the edge extension, to enable convolutions between the image texture and the filter kernel for downsampling along the horizontal and vertical dimensions. In this project, the downsampling was issued by using functions from OpenGL library to control the actual sample intervals in the texture fetching operations. For example, if using the variables *tex_width* and *tex_height* to represent the width and height of an image texture, the convolution between the image texture and the filter kernel along the horizontal dimension for dowsampling can be combined into the following OpenGL instruction sets and FP process, as shown in Fig.7 and Fig.8.



```
glBegin(GL_QUADS);
{
    glTexCoord2f(              0.0f,              0.0f);
    glVertex2f    (            0.0f,              0.0f);
    glTexCoord2f((float)tex_width,                0.0f);
    glVertex2f  ((float)tex_width/2,              0.0f);
    glTexCoord2f((float)tex_width,   (float)tex_height);
    glVertex2f  ((float)tex_width/2, (float)tex_height);
    glTexCoord2f(              0.0f, (float)tex_height);
    glVertex2f    (            0.0f, (float)tex_height);
}
glEnd();
```

Horizontal down-sampling according to ratio 2:1

Fig.7 OpenGL instructions for controlling filtering and downsampling



```
fragout_float main(vf30 IN,
                   uniform samplerRECT image,    //image texture
                   uniform samplerRECT filter,   //texture for filter kernel
                   uniform float L        //kernel length
)
{
   float3 sum=float3(0,0,0);

  // Implementing convolution

  for (int i=0; i<L; i++)
  {
    sum += f3texRECT(filter , float2(i+0.5,0.5)).r *
           f3texRECT(image , float2((IN.TEX0.x+i, IN.TEX0.y));
  }

  fragout_float OUT;
  OUT.col = float4(sum, 0.0);
  return OUT;
}
```

Fig.8 Corresponding fragment program for filtering in horizontal dimension

When implemented in the proposed GPGPU denoising framework, the filter kernel was stored in the *R* channel of a texture. As shown in Fig.8, a factor of 0.5 for addressing the pixel center when fetching a texture has been adopted.

The operation of filtering and down-sampling along the vertical direction is an analogue to the horizontal ones.

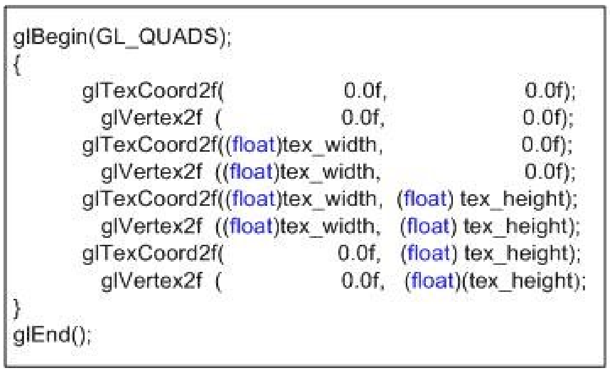
4.2.2 Thresholding

As highlighted in Fig.4, a critical step in the thresholding stage is to implement a Db1 wavelet on a GPU and to retrieve corresponding coefficients from the GPU's framebuffer and to transfer them to a CPU's memory for generating a rescaling factor. The task performed on the CPU is the sorting operation. This back-and-forward process is the most time-consuming step in the entire process for the reasons stated in Section 4.1.

Although some researchers claimed to have developed GPU-based sorting libraries for implementing the sorting algorithms at 16-bit and 32-bit floating precision at a CPU comparable performance, it is noticed that the implementations still struggle to sort arrays with non power-of-two image sizes [27]. For the adaptability, sorting operations in the devised framework in this project are still performed on the CPU. After threshold values being computed, it is then downloaded to GPU for modifying the detail coefficients obtained in the stage of decomposition.

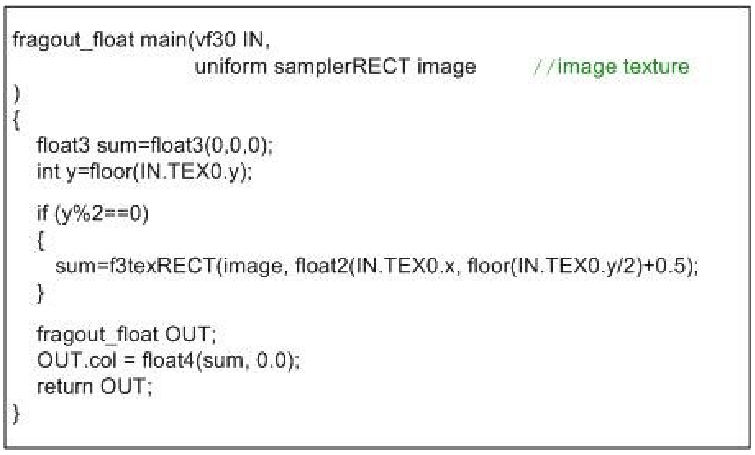
4.2.3 Reconstruction

The reconstruction phase in the framework is an inverse process of the decomposition, which is achieved by applying 1D inverse DWT vertically and horizontally in turn. For reconstruction, the process started from the lowest decomposition level – referred as *J*; and then the approximation coefficients cA_j, and the modified detail coefficients ($cH_j^{'}, cV_j^{'}, cD_j^{'}$) would be upsampled and filtered by corresponding reconstruction filters along vertical and horizontal dimensions respectively. The four computational results originated from cA_j, $cH_j^{'}, cV_j^{'}, cD_j^{'}$ would then be synthesized to form the approximation coefficients of the upper level j-1. After a series of recursive computation, the ultimate denoised image can be obtained. Fig.9 and Fig.10 illustrate the upsampling operations at the image size of *tex_width* and *tex_height*.



```
glBegin(GL_QUADS);
{
        glTexCoord2f(                0.0f,                  0.0f);
          glVertex2f  (               0.0f,                  0.0f);
        glTexCoord2f((float)tex_width,                       0.0f);
          glVertex2f  ((float)tex_width,                     0.0f);
        glTexCoord2f((float)tex_width,  (float) tex_height);
          glVertex2f  ((float)tex_width,  (float) tex_height);
        glTexCoord2f(               0.0f,  (float) tex_height);
          glVertex2f  (              0.0f,  (float)(tex_height);
}
glEnd();
```

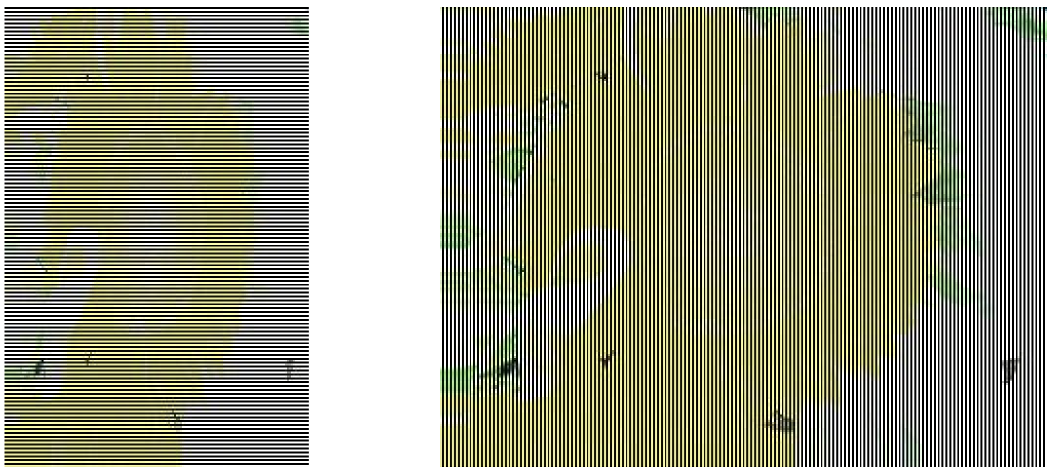
Fig.9 OpenGL commands that implement upsampling along the vertical dimension



```
fragout_float main(vf30 IN,
                   uniform samplerRECT image        //image texture
)
{
   float3 sum=float3(0,0,0);
   int y=floor(IN.TEX0.y);

   if (y%2==0)
   {
     sum=f3texRECT(image, float2(IN.TEX0.x, floor(IN.TEX0.y/2)+0.5);
   }

   fragout_float OUT;
   OUT.col = float4(sum, 0.0);
   return OUT;
}
```

Fig.10 Fragment program for upsampling along vertical direction

The effects of vertical upsampling and horizontal upsampling are display in Fig.11.



(a) Vertical upsampling (b) Horizontal upsampling

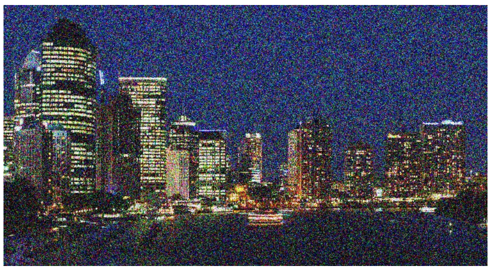
Fig.11 The effect of upsampling

5 Denoising effect analysis and performance evaluation

In this project, Db4 and Sym4 base wavelet have been experimented as the denoising wavelets in the framework respectively to validate the framework's functionality. In addition to the denoising effect to be benchmarked, another key performance is the computational efficiency that is often exponentially linked indicator to a specified base wavelet and the image size. For various base wavelets, the kernel length of the low-pass or high-pass filter is normally less than 20, therefore the image size becomes the dominant factor that influences the computational efficiency of the wavelet-based denoising. Section 5.2 will focus on analyzing the acceleration factor of the devised framework when applied for processing different image sizes.

5.1 Effect of denoising

Two noisy image samples that contain nonzero-mean white noise were tested in this project as shown in Fig.12. The two images formed by distinctive pixel groups with one displaying a night-sky cityscape consisting of synthesis straight lines and corners and the other showing an organic plant with arbitrary curves and edges.


(a) Noisy image *a*

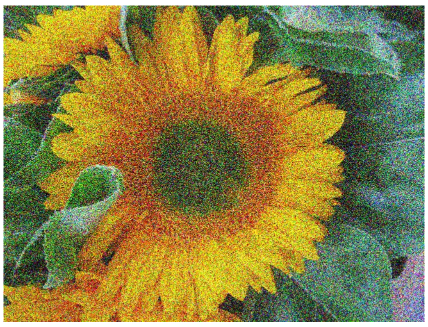
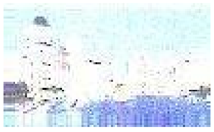
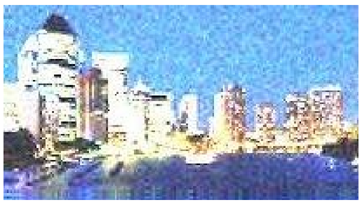
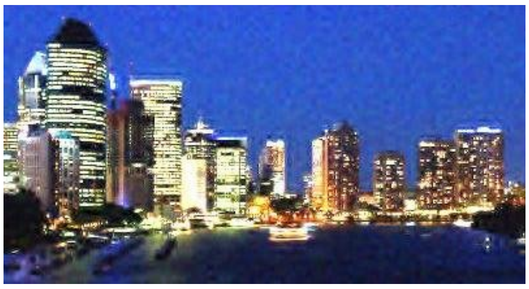

(b) Noisy image *b*

Fig.12 Two samples of noisy image

A Db4 wavelet was employed for de-noise image *a* and a Sym4 wavelet was used for image *b*. The maximum number of wavelet decompositions chosen for the test was 4. The synthesized images at different reconstruction level corresponding to the approximation coefficients (*cAs*) at the reconstruction are illustrated in Fig.13 and Fig.14.


(a) cA_3


(b) cA_2


(c) cA_1

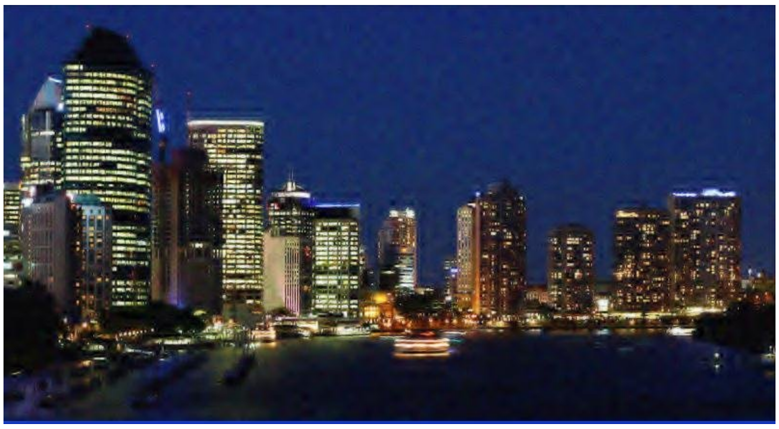
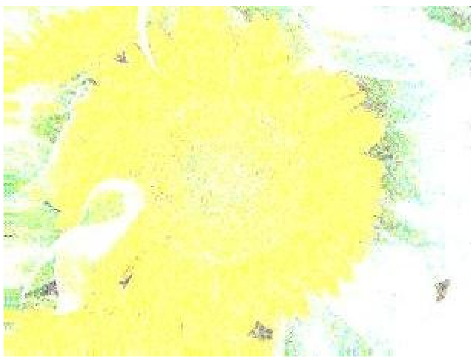
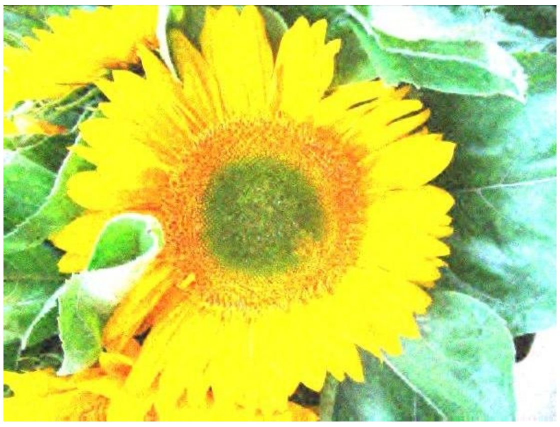

(d) The ultimate denoised image (cA_o)

Fig.13 Denoising effects using the Db4 wavelet


(a) cA_2


(b) cA_1

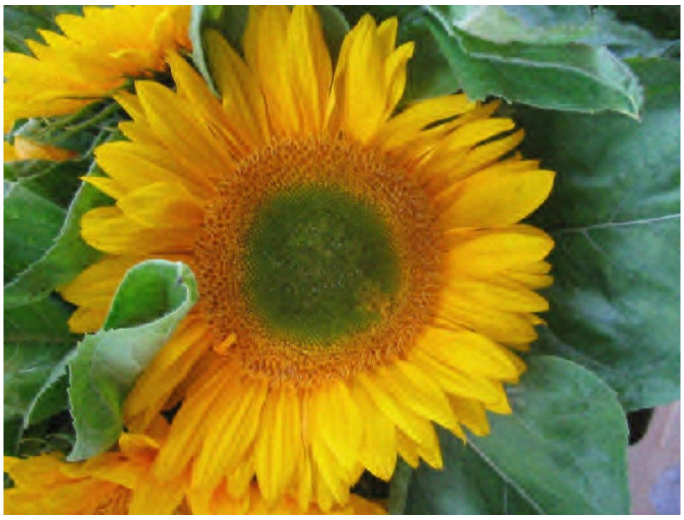

(c) The ultimate denoised image (cA_o)

Fig.14 Denoising effects using Sym4 wavelet

It was observed that during the process of reconstruction, much of the useful image details were resorted with the noise signals in the background region reduced. In real applications, noise rejection and oversmoothing are often a dilemma which is sometimes causing unsatisfying effects such as edge blurring. There exists a tradeoff between these two factors when choosing and balancing a donoising approach. In general, as indicated in Fig.13 and Fig.14 that the wavelet-based denoising achieved a good performance on GPU and restored a substantial percent of strong edges which can be seen from the reconstructed images, which further approves the effectiveness of wavelet for image denoising.

5.2 Evaluation on computational efficiency

5.2.1 Comparison with the software-based wavelet denoising

The computational efficiency of the developed GPGPU framework for image denoising was evaluated against the acceleration factor comparing with software-based wavelet implementations on a Pentium IV 2.6 GHz PC equipped with Nvidia's GeForce 7900 GTX graphics card. Five test images ranging from 512×512 to 2048×2048 were processed. Table 1 lists the comparison results regarding the overall operational time on software-based wavelet denoising and on the GPGPU denoising framework with the accelerating factors computed.

Table 1 Runtime comparisons on different image size (in ms)

Image size	512×512	800×600	1024×1024	1280×1024	2048×2048
Software-based	2125ms	2703ms	6094ms	7562ms	26234ms
GPGPU-based	222ms	348ms	725ms	1275ms	3324ms
Accelerating factor	9.6	7.8	8.4	5.9	7.9

To evaluate the acceleration performance of the framework on the distinctive decomposition and reconstruction stages, a further breakdown of computational time in regard to each stage is listed in Table 2 with a Db4 wavelet as a chosen target. It was envisaged that the framework had a satisfactory performance especially in the decomposition stage. On the other hand, the accelerating factor for the reconstruction is much lower than the decomposition. The reason for that is to obtain the approximation coefficients at level j (cA_j), the approximation and detail coefficients at level j+1 (cA_{j+1}, cH_{j+1}, cV_{j+1}, and cD_{j+1}) need to be upsampled and filtered in sequence in the framework, which increases the computational cost and results in the reduced acceleration performance comparing to the decomposition. In fact, the operations on all coefficients in the reconstruction stage are the same. Therefore a more optimal mechanism for texture mapping in the framework in order to enable all coefficients in the stage of reconstruction to be processed in parallel need to be researched in the future.

Table 2 Breakdown of computational time (in ms)

Image size	512×512	800×600	1024×1024	1280×1024	2048×2048
Software-based decomposition	423ms	658ms	1596ms	1923ms	5862ms
GPGPU-based decomposition	15ms	16ms	31ms	94ms	158ms
Accelerating factor	28.2	41.1	51.5	20.5	37.1

Software-based reconstruction	516ms	798ms	2112ms	2670ms	10968ms
GPGPU-based reconstruction	125ms	171ms	391ms	593ms	2000ms
Accelerating factor	4.1	4.7	5.4	4.5	5.5

Since most of the tasks in the stage of thresholding are actually carried out by CPU, the impact of this workload distribution on the GPGPU framework has also been evaluated. Table 3 lists the runtime of key steps in thresholding operation, which includes issuing the Db1 wavelet decomposition on a GPU, transferring coefficients of the Db1 decomposition at level 1, and sorting the coefficients to compute the median absolute values for generating the rescale factor. It can be observed that most of the runtime latency was caused by the reading of coefficients back from GPU's framebuffer and the sorting operation on CPU. Table 4 lists the proportion of the runtime of these two tasks in the entire GPGPU framework. It can be seen that the runtime of these two tasks dramatically increases along with the image size.

Table 3 Runtime of key steps in thresholding (in ms)

Image size	512×512	800×600	1024×1024	1280×1024	2048×2048
Issue Db1 Decomp.	3ms	5ms	9ms	11ms	36ms
Read-back framebuffer	31ms	47ms	109ms	359ms	500ms
Sort operation	31ms	62ms	125ms	156ms	562ms

Table 4 Proportional benchmarking of GPU-CPU data transfer latency

Image size	512×512	800×600	1024×1024	1280×1024	2048×2048
Latency of GPU-CPU uploading	62ms	109ms	234ms	515ms	1062ms
Total time cost	222ms	348ms	725ms	1275ms	3324ms
Proportion of the cross border delay	27.9%	31.3%	32.3%	40.4%	31.9%

5.2.2 Comparison with another GPU-based solution

The performance of the developed GPGPU framework was also compared with another GPU-based solution devised by Wong's group at the Chinese University of Hong Kong. The core of Wong's solution is to establish lookup tables along horizontal and vertical directions respectively to store the texture coordinates for texture fetching used in the fragment programs for DWT and IDWT at different level. The lookup tables were initialized by a program running on CPU initially.

Adopting the same approach for thresholding operations as explained in Section 4.2.2, a series of experiments for image decomposition and reconstruction that employed Wong's method was also issued. Table 5 lists the runtime performances regarding the sub-stages of decomposition, reconstruction and lookup table initialization.

Table 5 Runtime of sub-stages on various image sizes using Wong's method (in ms)

Image size	512×512	800×600	1024×1024	1280×1024	2048×2048
Decomposition	13ms	25ms	56ms	149ms	248ms
Reconstruction	16ms	31ms	59ms	154ms	251ms
Lookup table initialization	235ms	360ms	901ms	1479ms	3034ms

Comparing with the results shown in Table 2, it is observed that for the GPGPU framework devised in this project, the runtime of image decomposition is less than that of the Wong's method. While using the Wong's solution, the runtime of image reconstruction is faster than the proposed framework. Based on the processing flow of image reconstruction depicted in Fig.4, it can be seen that the processes of upsampling and filtering in IDWT are actually issued by different fragment programs running in multiple passes on GPU. The snippets of the fragment programs have been shown in Fig. 8 and Fig.10 respectively. In comparison, by using Wong's method, the upsampling and filtering can be issued by the same fragment program based on the pre-built texture coordinates lookup tables. In another term, these two processes can be implemented simultaneously. This is the reason why the runtime of image reconstruction using Wong's method is faster than the proposed solution. However, the Wong's approach requires a constant construction of processing phase related lookup tables which can be a time-consuming process to implement. Table 5 also lists the cost of runtime for establishing the texture coordinates lookup tables when using Wong's method, which dominates the application's runtime.

Table 6 lists the comparison results regarding the overall runtime performances of the devised GPGPU framework in this project. It can be seen that the overall processing time of the proposed framework is less than that of Wong's. Another advantage of this solution is that it only allocates textures for image and filter kernels which are essential for the GPU operation. The additional textures to store the lookup tables are unnecessary during the whole operation cycle; hence spare the hosting CPU program's involvement completely. This design further improves the GPU's memory usage when issuing wavelet-based denoising on large size digital images and/or high-definition videos.

Table 6 Runtime comparisons on different image size (in ms)

Image size	512×512	800×600	1024×1024	1280×1024	2048×2048
Wong's solution	284ms	466ms	1103ms	1877ms	4231ms
The new method	222ms	348ms	725ms	1275ms	3324ms

6 Conclusions and Future Works

A GPGPU framework has been devised and evaluated for wavelet-based denoising in this project. It harnesses the parallel processing ability and programmability of modern consume-level graphics hardware for accelerating the image processing speed. Popular signal denoising algorithms and techniques have been integrated to the design with GPU resources mapped to the corresponding processes. The work is particularly effective when the denoising approach is issued on large amount of noisy data. This framework has been focusing on the existing denoising approaches. Its overall performance were assessed on the visual quality and computational efficiency. It has been observed that the framework achieves a great performance increase in both fronts.

Currently, the proposed GPGPU framework is mainly used for 2D image processing. However, in some practical applications, such as video event detection using Closed Circuit Television (CCTV), a series of continuous video images will be processed and the pixel-oriented information are commonly treated as 3D volumes, in which a voxel is the basic volumetric data unit similar to a pixel in a 2D image. Processing of a large amount of voxels can bring in huge challenges to the computational efficiency. It has been envisaged that even using a low-definition video camera for analyzing the frames taken at a relatively short period of time, for example 60 seconds, can cause a big delay for a standard computational platform due to the large data size – a few hundred megabytes without compression. Based on the proposed GPGPU framework for 2D image processing, potentials have been noticed for extending it into the 3D-based data processing domain. A pilot project for accelerating a CCTV surveillance system for low volume crime detection has been started, which aims at developing techniques to generate real-time and automatic classification schemes for identifying events in a video. GPU-based image segmentation, edge/surface detection in a 3D spatial-temporal volume has been investigated.

It was also envisaged that although modern GPUs are fast co-processors, they are not designed to implement all the tasks and replace the CPU. How to optimize the allocation of computational tasks automatically in between CPU and GPU is a continuing research topic, which is tightly related to the evolution of computer hardware. Cell CPU and SLi-GPU present both opportunities and challenges. Further work in this project will focus on these issues aiming to obtain a series of generic cost-effective GPGPU solutions for signal processing. In addition, most of the GPUs nowadays only support floating operations at single-precision, which presents a major drawback when applied on applications requiring higher precision. Techniques to perform integrated extended-precision arithmetic on GPUs will also be a vital part for the future success of GPGPU.

References

[1] A. Bovik, Handbook of image and video processing (Second Edition), Elsevier Academic Press, London, 2005.

[2] K. Amolins, Y. Zhang, P. Dare, Wavelet based image fusion techniques — An introduction, review and comparison, Journal of Photogrammetry and Remote Sensing. 62(4) (2007), 249-263.

[3] L. Birgé, P. Massart, From model selection to adaptive estimation, Festschrift for Lucien Le Cam/ Research Papers in Probability and Statistics, Springer(1997), pp. 55-88.

[4] A. Barron, L. Birgé, P. Massart, Risk bounds for model selection via penalization, Probability Theory and Related Fields, 113(3)(1999) 301-413.

[5] D. L. Donoho, I. M. Johnstone, Adapting to unknown smoothness via wavelet shrinkage, Journal of the American Statistical Association, 90 (1995), pp.1200-1224.

[6] D. L. Donoho, I. M. Johnstone, Minimax estimation via wavelet shrinkage, Journal of Applied Probability, 26 (3) (1998), 879-921.

[7] D. L. Donoho, I. M. Johnstone, G. Kerkyacharian, D. Picard, Wavelet shrinkage: Asymptopia, Journal of the Royal Statistical Society, Series B(*Methodological)*, 57(2) (1995), pp. 301-369.

[8] E. Chicken, T. T. Cai, Block thresholding for density estimation: local and global adaptivity, Journal of Multivariate Analysis, 95(1)(2005), 76-106.

[9] A. Azzalini, M. Farge, K. Schneider, Nonlinear wavelet thresholding: A recursive method to determine the optimal denoising threshold, Applied and Computational Harmonic Analysis, 18(2)(2005),177-185.

[10] W. Sweldens, The lifting scheme:Aconstruction of second generation wavelets, *SIAM J. Math. Anal.*. 29(2)(1998), 511–546.

[11] K. Kuzume, K. Niijima, S. Takano, FPGA-based lifting wavelet processor for real-time signal detection, Signal Processing. 84(10),2004, 1931-1940.

[12] M. Vishwanath, R. M. Owens (1995), VLSI architecture for the discrete wavelet transform, IEEE Transactions on Circuit & System. 42(5)(1995), 305-316.

[13] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, Cg: A system for programming graphics hardware in a C-like language, in *ACM Trans. Graphics*, 2003.

[14] J. Bolz, I. Farmer, E. Grinspun, and P. Schreoder, Sparse matrix solvers on the GPU: Conjugate gradients and multigrid, in *ACM Trans. Graphics*, 2003.

[15] J. Krüger and R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, in *ACMTrans. Graphics*, 2003.

[16] K. Moreland and E. Angel, The FFT on a GPU, Proc. ACMSIGGRAPH/EUROGRAPHICS Conf. Graphics Hardware (HWWS'03), pp. 112-119, 2003.

[17] M. Hopf and T. Ertl, Hardware-Accelerated Wavelet Transformations, Proc. EG/IEEE TVCG Symp. Visualization (SisSym '00), pp. 93-103, May 2000.

[18] T. T. Wong, C. S. Leung, P. A. Heng, and J. Q. Wang, Discrete Wavelet Transform on Consumer-Level Graphics Hardware, IEEE Transaction on Multimedia. 9(3)(2007) 668-673.

[19] C. Tenllado, J. Setoain, M. Prieto, L. Pinuel, F. Tirado, Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS. 19(3)(2008) 299-310.

[20] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, et al., Real-Time Volume Graphics, Course Notes 28 in SIGGRAPH2004(2004) 13-18.

[21] J. D.Owens, D. Luebke, N. Govindaraju, M. Harris, et al., A Survey of General-Purpose Computation on Graphics Hardware, Computer Graphics Forum. 26 (1)(2007) 80 -113.

[22] R. Strzodka, M. Doggett, A.Kolb, Scientific Computation for Simulation on programmable Graphics Hardware, Simulation Modelling Practice and Theory. 13(8)(2005) 667-681.

[23] C. Oat. Rendering to an off-screen buffer with WGL_ARB_pbuffer, Technology paper of ATI Inc. pp.1-13. Available on: http://ati.amd.com/developer/ATIpbuffer.pdf.

[24] E. Persson. Framebuffer Objects, Technology paper of ATI Inc. pp.1-12. Available on: http://ati.amd.com/developer/SDK/AMD_SDK_Samples_May2007/Documentations/FramebufferObjects.pdf.

[25] M. Pharr, R. Fernando, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Press, London ,2005.

[26] G. Strang and T. Nguyen, Wavelets and Filter Banks. Cambridge,MA: Wellesley-Cambridge, 1996.

[27] GPUSort: A high performance GPU sorting library. Available on http://gamma.cs.unc.edu/GPUSORT/.

RESPONSE TO REVIEWERS AND LIST OF CHANGES

MANUSCRIPT NUMBER: SIGPRO-D-08-00879

REVIEWER 1

The paper was well presented. This includes the experiments and analysis which was done very thoroughly and in great detail. However, as mentioned in the paper, future experiments needs to be conducted using standard database of images eg Ferret's to cater to all aspects of possible, image features. Maybe, other denoising experiments can be done on other signals for eg audio or video images.

Response: The authors thoroughly agree with the review's suggestion on the adoption of widely recognisable benchmarking images for the future quantitative tests. Measures and experiments have been carried out to cover other image features by deploying the devised GPGPU framework. Results so far are largely satisfactory and will be reported in a separate article focusing on the generic image processing issues. To clarify the main objectives of this investigation and the following phase, the text below has been added at the second paragraph in Section 6 (Conclusions and Future Works).

Currently, the proposed GPGPU framework is mainly used for 2D image processing. However, in some practical applications, such as video event detection using Closed Circuit Television (CCTV), a series of continuous video images will be processed and the pixel-oriented information are commonly treated as 3D volumes, in which a voxel is the basic volumetric data unit similar to a pixel in a 2D image. Processing of a large amount of voxels can bring in huge challenges to the computational efficiency. It has been envisaged that even using a low-definition video camera for analyzing the frames taken at a relatively short period of time, for example 60 seconds, can cause a big delay for a standard computational platform due to the large data size – a few hundred megabytes without compression. Based on the proposed GPGPU framework for 2D image processing, potentials have been noticed for extending it into the 3D-based data processing domain. A pilot project for accelerating a CCTV surveillance system for low volume crime detection has been started, which aims at developing techniques to generate real-time and automatic classification schemes for identifying events in a video. GPU-based image segmentation, edge/surface detection in a 3D spatial-temporal volume has been investigated.

REVIEWER 2

This paper proposed a wavelet based denoising by using the GPU. The novelty of this work is an efficient implementation of DWT arranged for current generation GPUs. I think the paper is well organized and worth to be published. But in simulation, authors only show the comparison with the CPU based denoising. They should show the simulation comparison with the conventional GPU based method such as Wong's method.

Response: The authors appreciate the importance in assessing the performance differences in between different GPU-based image denosing approaches. Through deploying the source code released by Wong's research group on their website, the authors were able to analyse the two distinctive GPU acceleration strategies and carrying out tests on their runtime performances on the same computing platform. The results have been shown in Section 5.2.2 with corresponding analysis.

5.2.2 Comparison with another GPU-based solution

The performance of the developed GPGPU framework was also compared with another GPU-based solution devised by Wong's group at the Chinese University of Hong Kong. The core of Wong's solution is to establish lookup tables along horizontal and vertical directions respectively to store the texture coordinates for texture fetching used in the fragment programs for DWT and IDWT at different level. The lookup tables were initialized by a program running on CPU initially.

Adopting the same approach for thresholding operations as explained in Section 4.2.2, a series of experiments for image decomposition and reconstruction that employed Wong's method was also issued. Table 5 lists the runtime performances regarding the sub-stages of decomposition, reconstruction and lookup table initialization.

Table 5 Runtime of sub-stages on various image sizes using Wong's method (in ms)

Image size	*512×512*	*800×600*	*1024×1024*	*1280×1024*	*2048×2048*
Decomposition	*13ms*	*25ms*	*56ms*	*149ms*	*248ms*
Reconstruction	*16ms*	*31ms*	*59ms*	*154ms*	*251ms*

Lookup table initialization	*235ms*	*360ms*	*901ms*	*1479ms*	*3034ms*

Comparing with the results shown in Table 2, it is observed that for the GPGPU framework devised in this project, the runtime of image decomposition is less than that of the Wong's method. While using the Wong's solution, the runtime of image reconstruction is faster than the proposed framework. Based on the processing flow of image reconstruction depicted in Fig.4, it can be seen that the processes of upsampling and filtering in IDWT are actually issued by different fragment programs running in multiple passes on GPU. The snippets of the fragment programs have been shown in Fig. 8 and Fig.10 respectively. In comparison, by using Wong's method, the upsampling and filtering can be issued by the same fragment program based on the pre-built texture coordinates lookup tables. In another term, these two processes can be implemented simultaneously. This is the reason why the runtime of image reconstruction using Wong's method is faster than the proposed solution. However, the Wong's approach requires a constant construction of processing phase related lookup tables which can be a time-consuming process to implement. Table 5 also lists the cost of runtime for establishing the texture coordinates lookup tables when using Wong's method, which dominates the application's runtime.

Table 6 lists the comparison results regarding the overall runtime performances of the devised GPGPU framework in this project. It can be seen that the overall processing time of the proposed framework is less than that of Wong's. Another advantage of this solution is that it only allocates textures for image and filter kernels which are essential for the GPU operation. The additional textures to store the lookup tables are unnecessary during the whole operation cycle; hence spare the hosting CPU program's involvement completely. This design further improves the GPU's memory usage when issuing wavelet-based denoising on large size digital images and/or high-definition videos.

Table 6 Runtime comparisons on different image size (in ms)

Image size	*512×512*	*800×600*	*1024×1024*	*1280×1024*	*2048×2048*
Wong's solution	*284ms*	*466ms*	*1103ms*	*1877ms*	*4231ms*
The new method	*222ms*	*348ms*	*725ms*	*1275ms*	*3324ms*

REVIEWER 3

This paper proposes a GPU implementation of Wavelet based image denoising using Discrete Wavelet Transform (DWT). The paper proposes a GPU implementation based on the new graphics hardware features and shows improvements as compared to earlier implementation of DWT. From my perspective, this is a useful contribution for people who are interesting in accelerating image processing and filtering applications. Overall, the algorithm is well presented and the snippet shader programs for fragment shader and upsampling operations is quite clear. A couple of minor comments related to the text:

i) Page 2 (and in other places in the paper): instead of mother wavelet, use "base" or "parent" wavelet
ii) Page 8, line 54: instead of "is consisted" use "consists"
iii) Page 10, Line 3: cite the reference related to Donoho and Johnstone
iv) Page 10, Line 45-46: the sentence construction is incorrect
v) Page 12, line 10: "many researches" is not gramatically correct
vi) Page 12, Line 27: replace "in sequential" with "in sequence"
vii) Page 12, Line 43: replace "aiming at seeking" with "aimed at seeking"

Response: The authors would express sincere thankfulness to the reviewer and his/her thorough reviewing. All the errors pointed out have been amended accordingly in this version.

Figure

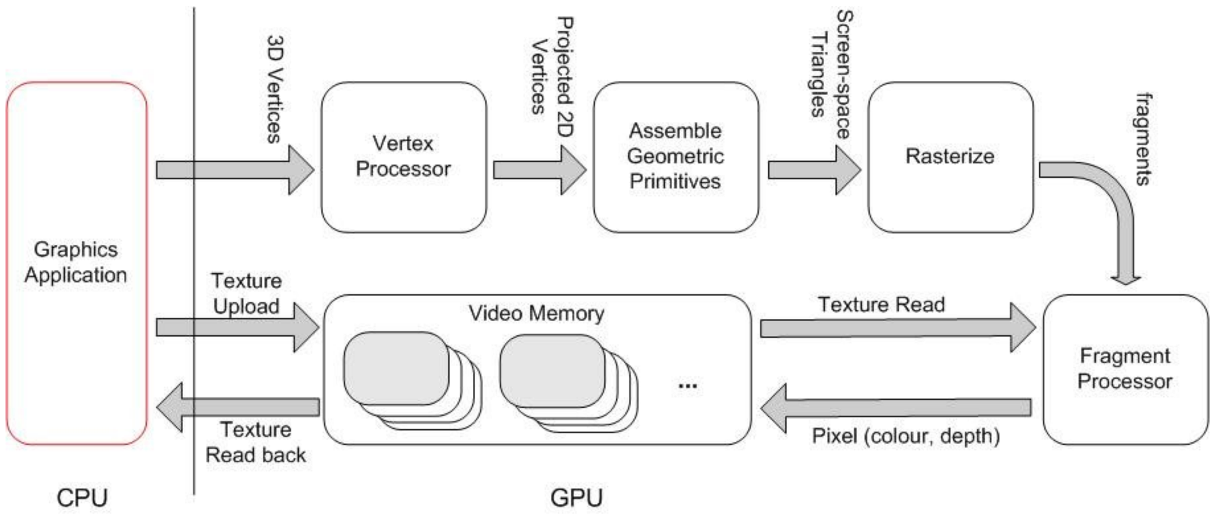


Fig.1 Overview of the 3D Graphics Pipeline

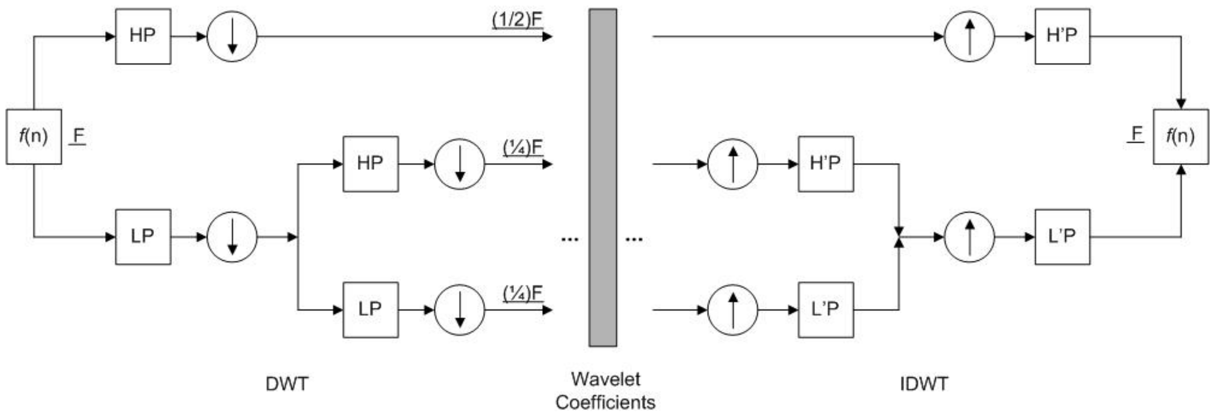


Fig.2 Multi-level DWT and IDWT

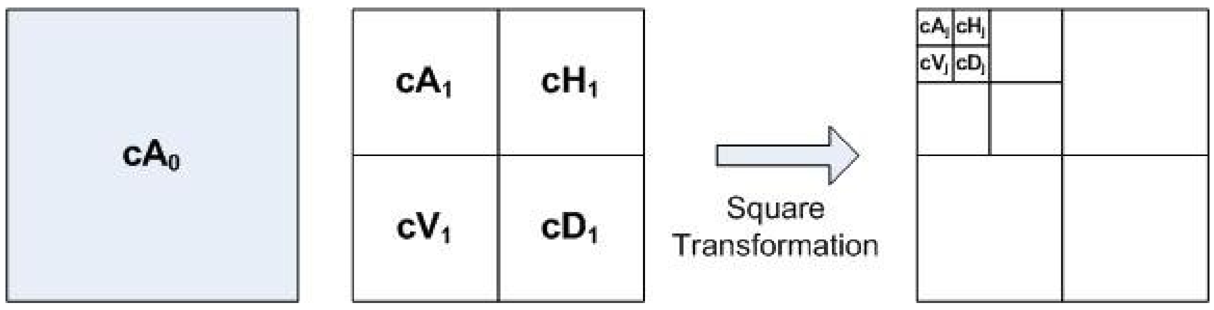


Fig.3 The square decomposition scheme

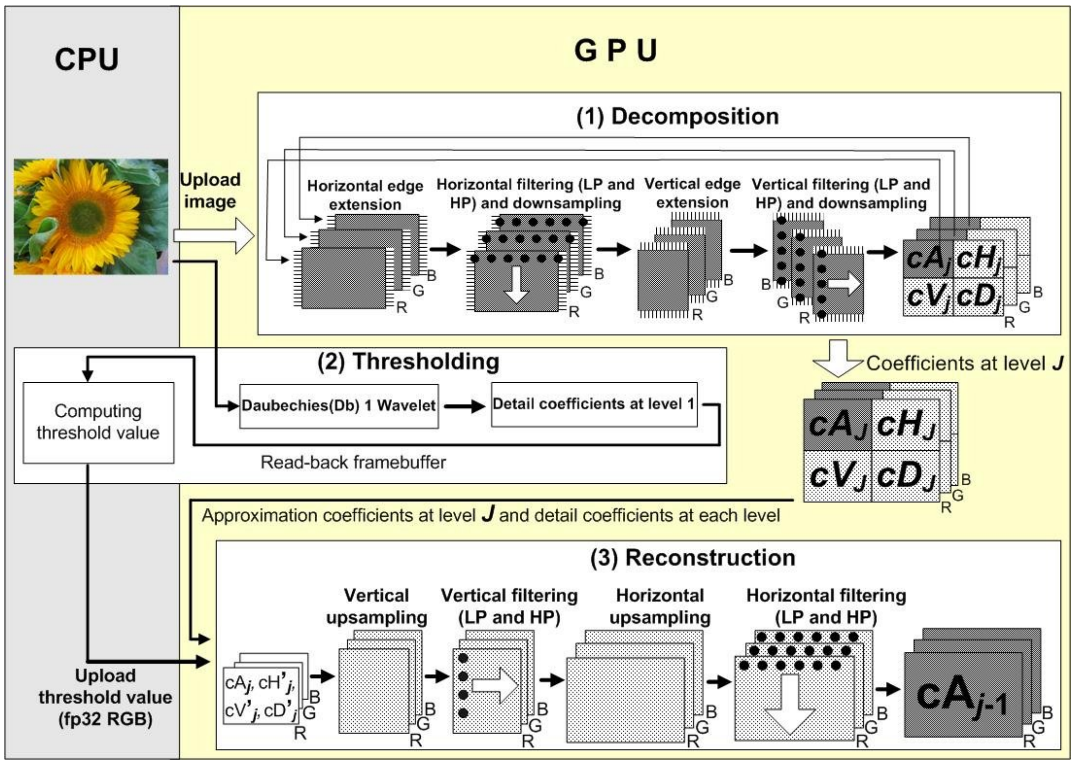

CPU
GPU
(1) Decomposition
Upload image
Horizontal edge extension
Horizontal filtering (LP and HP) and downsampling
Vertical edge extension
Vertical filtering (LP and HP) and downsampling
cA_j cH_j cV_j cD_j
R G B
Coefficients at level J
cA_J cH_J cV_J cD_J
(2) Thresholding
Computing threshold value
Daubechies(Db) 1 Wavelet
Detail coefficients at level 1
Read-back framebuffer
Approximation coefficients at level J and detail coefficients at each level
(3) Reconstruction
Vertical upsampling
Vertical filtering (LP and HP)
Horizontal upsampling
Horizontal filtering (LP and HP)
cA_j, cH'_j, cV'_j, cD'_j
cA_j-1
Upload threshold value (fp32 RGB)

Fig.4 Overview of the framework of the GPGPU-based wavelet-based denoising

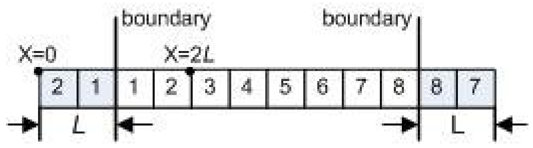
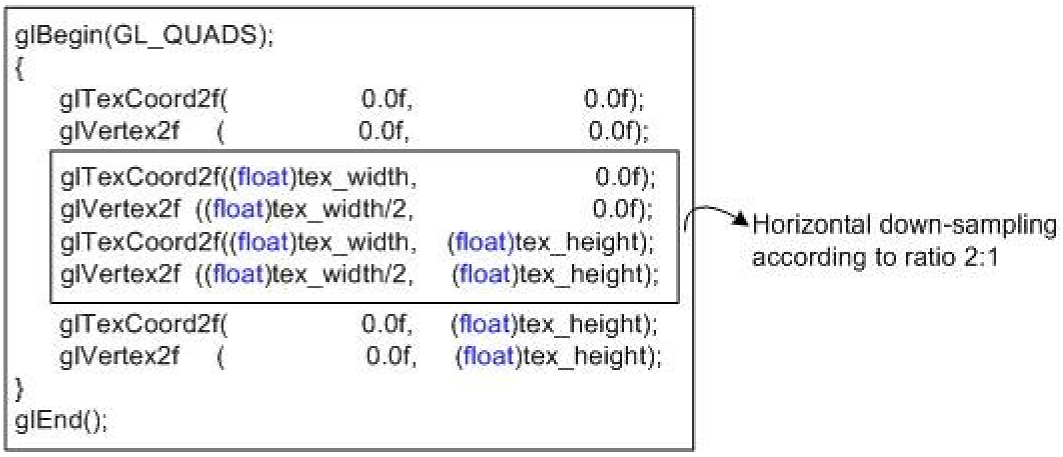


Fig.5 The symmetrical periodic extension scheme

```
fragout_float main(vf30 IN,
                  uniform samplerRECT image,   //image texture
                  uniform float L     //extension length)
{
  fragout_float OUT;
  OUT.col =f4texRECT(image, float2(2L-IN.TEX0.x, IN.TEX0.y));
  OUT.col.a=0.0;
  return OUT;
}
```

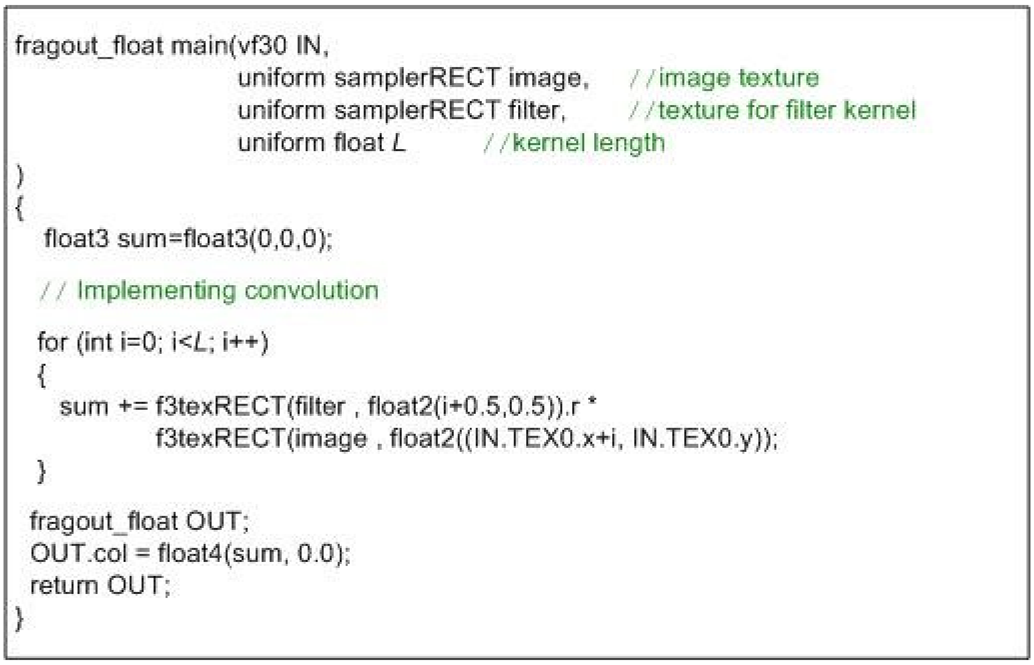
Fig.6 FP for edge extension



```
glBegin(GL_QUADS);
{
    glTexCoord2f(              0.0f,                 0.0f);
    glVertex2f    (            0.0f,                 0.0f);
    glTexCoord2f((float)tex_width,                   0.0f);
    glVertex2f  ((float)tex_width/2,                 0.0f);
    glTexCoord2f((float)tex_width,   (float)tex_height);
    glVertex2f  ((float)tex_width/2, (float)tex_height);
    glTexCoord2f(              0.0f, (float)tex_height);
    glVertex2f    (            0.0f, (float)tex_height);
}
glEnd();
```

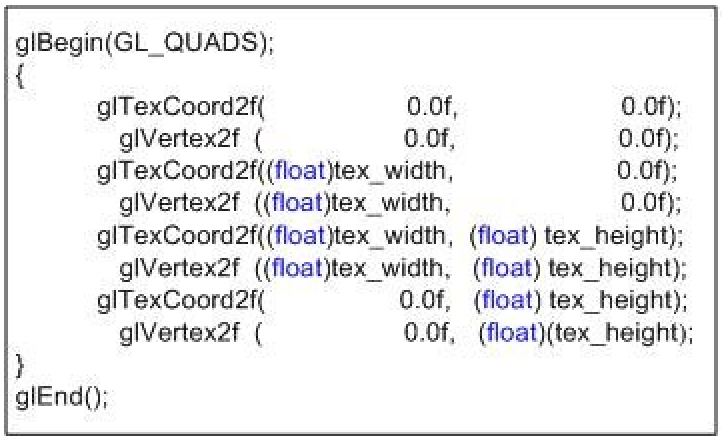
Horizontal down-sampling according to ratio 2:1

Fig.7 OpenGL instructions for controlling filtering and downsampling



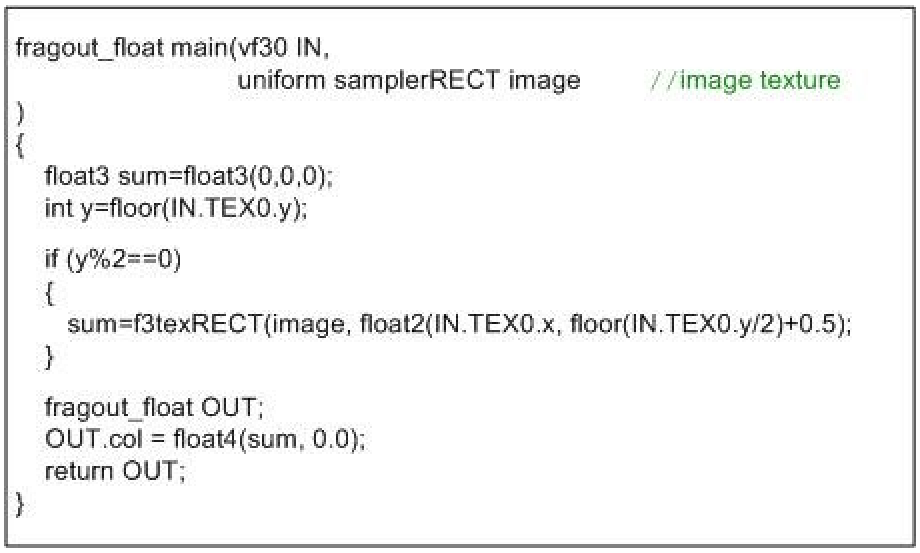
```
fragout_float main(vf30 IN,
                   uniform samplerRECT image,    //image texture
                   uniform samplerRECT filter,   //texture for filter kernel
                   uniform float L        //kernel length
)
{
   float3 sum=float3(0,0,0);

  // Implementing convolution

  for (int i=0; i<L; i++)
  {
    sum += f3texRECT(filter , float2(i+0.5,0.5)).r *
             f3texRECT(image , float2((IN.TEX0.x+i, IN.TEX0.y));
  }

  fragout_float OUT;
  OUT.col = float4(sum, 0.0);
  return OUT;
}
```

Fig.8 Corresponding fragment program for filtering in horizontal dimension



```
glBegin(GL_QUADS);
{
        glTexCoord2f(               0.0f,                0.0f);
          glVertex2f  (              0.0f,                0.0f);
        glTexCoord2f((float)tex_width,                   0.0f);
          glVertex2f  ((float)tex_width,                 0.0f);
        glTexCoord2f((float)tex_width,  (float) tex_height);
          glVertex2f  ((float)tex_width,  (float) tex_height);
        glTexCoord2f(               0.0f,  (float) tex_height);
          glVertex2f  (              0.0f,  (float)(tex_height);
}
glEnd();
```

Fig.9 OpenGL commands that implement upsampling along the vertical dimension



```
fragout_float main(vf30 IN,
                  uniform samplerRECT image        //image texture
)
{
   float3 sum=float3(0,0,0);
   int y=floor(IN.TEX0.y);

   if (y%2==0)
   {
     sum=f3texRECT(image, float2(IN.TEX0.x, floor(IN.TEX0.y/2)+0.5);
   }

   fragout_float OUT;
   OUT.col = float4(sum, 0.0);
   return OUT;
}
```

Fig.10 Fragment program for upsampling along vertical direction

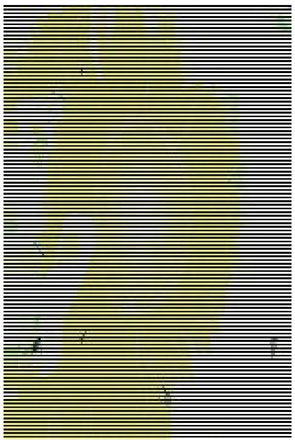


Fig.11 (a) Vertical upsampling

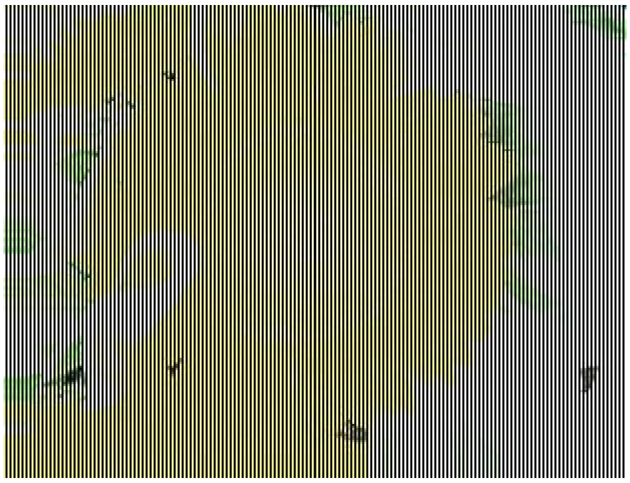


Fig.11 (b) Horizontal upsampling

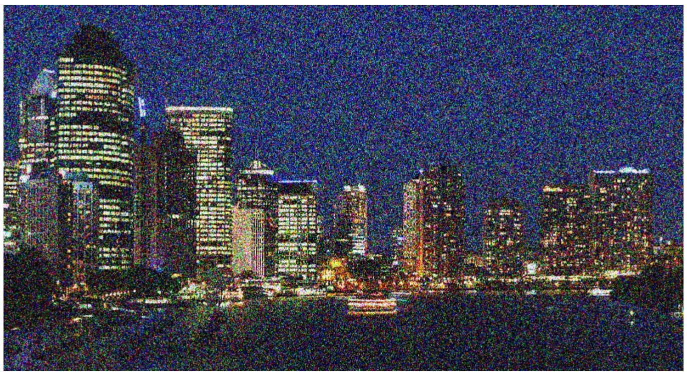


Fig.12(a) Noisy image *a*

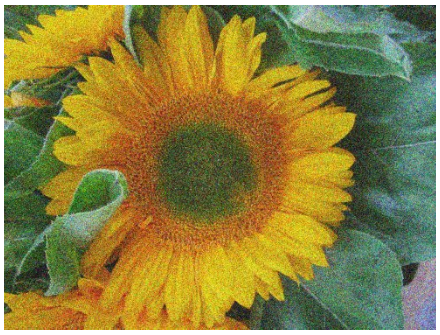


Fig.12(b) Noisy image *b*

Fig.12 Two samples of noisy image

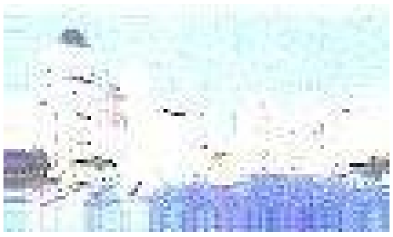


Fig.13(a) cA_3

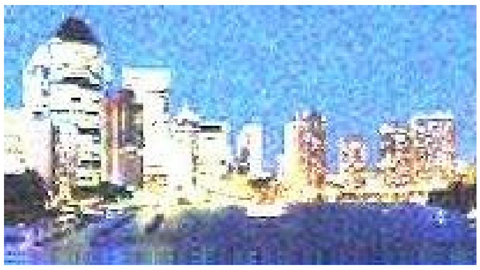


Fig.13(b) cA_2

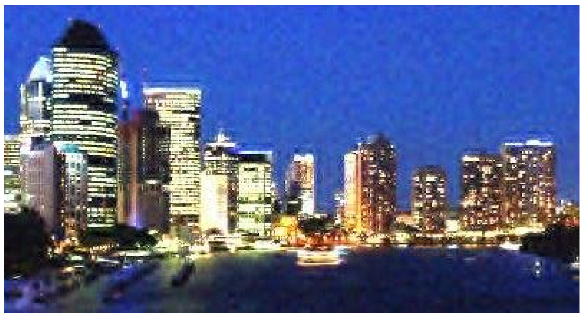


Fig.13(c) cA_1

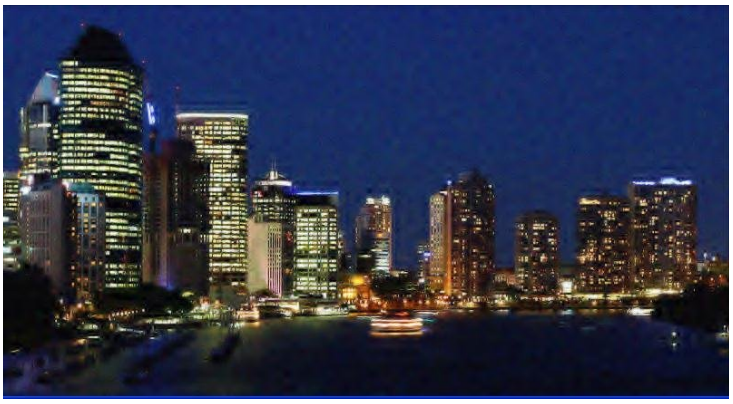


Fig.13(d) The ultimate denoised image (cA_o)

Fig.13 Denoising effects using the Db4 wavelet

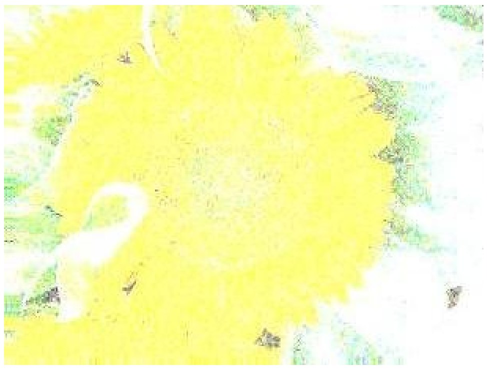


Fig.14(a) cA_2

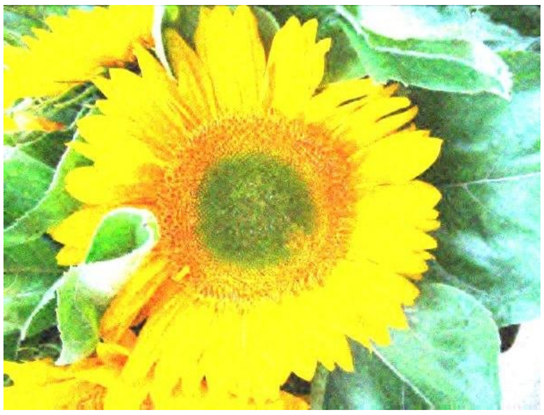


Fig.14(b) cA_1

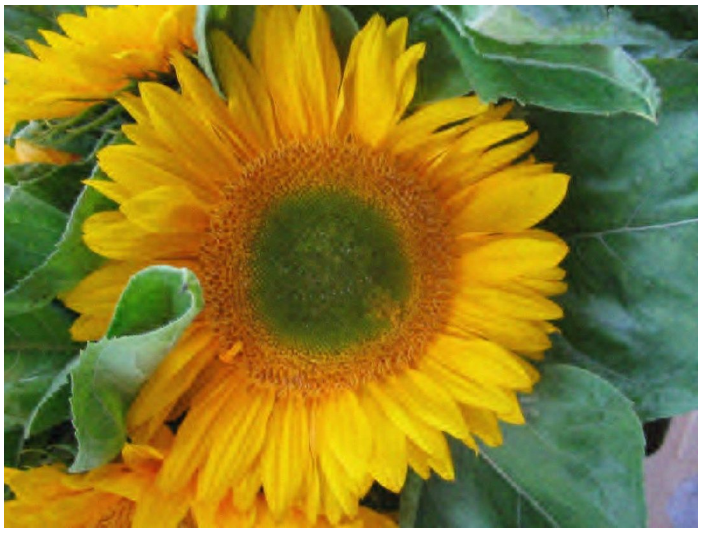


Fig.14(c) The ultimate denoised image (cA_o)

Fig.14 Denoising effects using Sym4 wavelet

Table

Table 1 Runtime comparisons on different image size (in ms)

Image size	512×512	800×600	1024×1024	1280×1024	2048×2048
Software-based	2125ms	2703ms	6094ms	7562ms	26234ms
GPGPU-based	222ms	348ms	725ms	1275ms	3324ms
Accelerating factor	9.6	7.8	8.4	5.9	7.9

Table 2 Breakdown of computational time (in ms)

Image size	512×512	800×600	1024×1024	1280×1024	2048×2048
Software-based decomposition	423ms	658ms	1596ms	1923ms	5862ms
GPGPU-based decomposition	15ms	16ms	31ms	94ms	158ms
Accelerating factor	28.2	41.1	51.5	20.5	37.1
Software-based reconstruction	516ms	798ms	2112ms	2670ms	10968ms
GPGPU-based reconstruction	125ms	171ms	391ms	593ms	2000ms
Accelerating factor	4.1	4.7	5.4	4.5	5.5

Table 3 Runtime of key steps in thresholding (in ms)

Image size	512×512	800×600	1024×1024	1280×1024	2048×2048
Issue Db1 Decomp.	3ms	5ms	9ms	11ms	36ms
Read-back framebuffer	31ms	47ms	109ms	359ms	500ms
Sort operation	31ms	62ms	125ms	156ms	562ms

Table 4 Proportional benchmarking of GPU-CPU data transfer latency

Image size	512×512	800×600	1024×1024	1280×1024	2048×2048
Latency of GPU-CPU uploading	62ms	109ms	234ms	515ms	1062ms
Total time cost	222ms	348ms	725ms	1275ms	3324ms
Proportion of the cross border delay	27.9%	31.3%	32.3%	40.4%	31.9%

Table 5 Runtime of sub-stages on various image sizes using Wong's method (in ms)

Image size	512×512	800×600	1024×1024	1280×1024	2048×2048
Decomposition	13ms	25ms	56ms	149ms	248ms
Reconstruction	16ms	31ms	59ms	154ms	251ms
Lookup table initialization	235ms	360ms	901ms	1479ms	3034ms

Table 6 Runtime comparisons on different image size (in ms)

Image size	512×512	800×600	1024×1024	1280×1024	2048×2048
Wong's solution	284ms	466ms	1103ms	1877ms	4231ms
The new method	222ms	348ms	725ms	1275ms	3324ms