University of Huddersfield Repository

Su, Yang, Xu, Zhijie, Jiang, Xiang and Pickering, Jonathan

Discrete wavelet transform on consumer-level graphics processing unit

Original Citation

Su, Yang, Xu, Zhijie, Jiang, Xiang and Pickering, Jonathan (2008) Discrete wavelet transform on consumer-level graphics processing unit. In: Proceedings of Computing and Engineering Annual Researchers' Conference 2008: CEARC'08. University of Huddersfield, Huddersfield, pp. 40-47. ISBN 978-1-86218-067-3

This version is available at http://eprints.hud.ac.uk/id/eprint/3682/

The University Repository is a digital collection of the research output of the University, available on Open Access. Copyright and Moral Rights for the items on this site are retained by the individual author and/or other copyright owners. Users may access full items free of charge; copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational or not-for-profit purposes without prior permission or charge, provided:

- The authors, title and full bibliographic details is credited in any copy;
- A hyperlink and/or URL is included for the original metadata page; and
- The content is not changed in any way.

For more information, including our policy and submission procedure, please contact the Repository Team at: E.mailbox@hud.ac.uk.

http://eprints.hud.ac.uk/

Discrete Wavelet Transform on Consumer- Level Graphics Processing Unit

Yang Su, Zhijie Xu, Xiangqian Jiang and J. Pickering
University of Huddersfield, Queensgate, Huddersfield HD1 3DH, UK

ABSTRACT

The discrete wavelet transform (DWT) has been extensively studied and developed in various scientific and engineering fields. The multiresolution and local nature of the DWT facilitates applications requiring progressiveness and the capture of high-frequency details. However, the intensive computation of DWT caused by multilevel filtering/down-sampling will become a significant bottleneck in real-time applications when the data size is large. This paper presents a SIMD-based parallel processing framework as a commodity solution to this problem, that is based on the consumer-level programmable graphic processing unit (GPU) on personal computers. Simulation tests show that, in contrast to those CPU-based solutions for DWT, this GPU-based parallel processing framework can bring a significant performance gain on a normal PC without extra cost.

Keywords discrete wavelet transform, filtering, parallel processing, graphic processing unit

1 INTRODUCTION

The discrete wavelet transform (DWT) has been extensively applied in engineering, especially in image classification, characteristic extraction, image de-noising, and image compression. The main reason for DWT's popularity is the fact that DWT can extract the maximum energy from a signal just using several transform coefficients, which is also the reason why DWT was adopted as the core engine in JPEG2000 [1], the second generation of the popular JPEG still image encoding standard. The traditional DWT employs Filter Bank Scheme (FBS) that is a recursive process that uses high-pass and low-pass filtering iteratively and inevitably brings intensive computation. To reduce the computational size of the DWT, Sweldens proposed an algorithm called Lifting Scheme (LS) that has been a great success [2]. In contrast to FBS, LS has two remarkable advantages in actual applications:

1. the filtering series of LS are less than FBS since LS just uses a half of filtering series of FBS;
2. LS can be used in Integer Wavelet Transform (IWF), which is of great importance for image compression with non-distortion.

Although LS decreases the number of filtering series in the DWT, for high-definition image or video with enormous number of pixels, LS still can't achieve real-time or interactive performance. Vishwanath et al. presented a hardware implementation of the DWT that employs VLSI architecture to further promote computational efficiency [3]. Unfortunately, the VLSI architecture has proved an expensive solution since many more memory units are needed. In recent years, the rapid improvement in the performance of PC-grade graphics processing units (GPU), their natural data parallelism and improved programmability, have made GPU a competitive platform for computationally demanding tasks in a wide variety of application domains. Many researchers and developers have become interested in harnessing the power of commodity graphics hardware for general-purpose computing, known as General Purpose Graphics Processing Unit (GPGPU). As a powerful and inexpensive solution, GPGPU has been applied in the area such as linear algebra, differential equation solving, physically based simulation, signal and image processing [4]. The main characteristic of GPGPU is its adoption of the data parallel computation that is achieved by the stream processing, which was the foundation for GPGPU's superior computational efficiency over CPU-based approach.

In this paper, a Single Instruction Multiple Data (SIMD) based parallel processing framework and its corresponding prototype for implementing hardware-accelerated DWT is presented. This prototype

makes use of programmability of consumer-level GPU to carry out the DWT on a GPU. We evaluated the performance of our proposed prototype by comparing with the results obtained by Wavelet toolbox in MATLAB.

The remainder of this paper is organized as follow: Section 2 gives a brief introduction to DWT, Section 3 presents the framework and prototype of the new GPU-accelerated 2D DWT technique that is developed by employing shading language -- Cg, Section 4 evaluates the performance of the proposed prototype with Section 5 concludes and highlights the future works.

2 TWO- DIMENSIONAL DWT

Given a signal s of length N, the 1D-DWT produces a pyramidal decomposition of s which consists of $\log_2 N$ levels at most. Level j generates a pair of approximation coefficients cA_j and detail coefficients cD_j from the approximation band of the previous level j–1, i.e. cA_{j-1}, where cA_j is a coarse-grained representation of its predecessor cA_{j-1}, and cD_j contains the high-frequency details that have been removed [5]. So the decomposition of signal s analyzed at level j has the following structure: [cA_j, cD_j, ..., cD_1]. As shown in Fig.1, this structure contains for j = 3 the terminal nodes of the following tree.

For 1D-DWT, cA_j and cD_j are produced by convolving cA_{j-1} with a low-pass filter (denoted by Lo_D) and a high-pass filter (denoted by Hi_D) respectively. The detailed procedure of decomposition at level j is depicted as Fig.2.

The 2D-DWT is usually obtained by applying a separate 1D transform along the horizontal and vertical directions. The most common approach, known as the square decomposition, alternates between computations on image rows and columns [6]. This process is applied recursively to the quadrant containing the coarse scale approximation in both horizontal and vertical directions. This way, the data on which computations are performed is reduced to a quarter in each decomposition step. It is noted that there are four coefficients, approximation coefficients cA_j, horizontal detail coefficient cH_j, vertical detail coefficient cV_j, and diagonal detail coefficient cD_j, are produced in square decomposition at level j, which is shown in Fig.3.

The detailed procedure of producing cA_j, cH_j, cV_j, and cV_j is shown in Fig.4. Fig.5 shows the decomposition tree of 2D-DWT where j = 2.

3 GPU COMPUTING MODEL

In this section, we give a brief review on the GPU architecture and programming model, which outlines a high-level description of the traditional rendering pipeline and illustrates how DWT can be mapped to the GPU using a stream programming model.

The rendering pipeline of a GPU contains three parts: vertex processing, rasterization, and fragment processing. The inputs to this pipeline are vertices from a 3D polygonal mesh together with attached information such as their colors and texture coordinates, and the output is a 2D array of pixels to be displayed on the screen.

During vertex processing, the geometry engine in the pipeline operates on incoming stream of vertices, mainly computing linear transformations, such as translation, rotation and projection of the vertices. After perspective transformation, which is the final step of the vertex processing stage, rasterization converts geometric data into fragments. Each fragment corresponds to a square pixel in the resulting image. These fragments produced by rasterization are written into the frame buffer(color buffer). Then in the stage of fragment processing, the color of each fragment is computed using texture mapping and other mathematical operations. the output from the fragment processing stage is combined with the existing data stored at the associated 2D locations in the frame buffer (color buffer) to produce the final colors [7].

Until only a few years ago, commercial GPUs were implemented using a fixed-function rendering pipeline. However, most GPUs today include fully programmable Vertex and Fragment stages. The programs that they execute are usually called vertex and fragment programs (or shaders), respectively, and can be written using C-like high-level languages such as Cg. This is the feature that allows for the implementation of nongraphics applications on the GPUs. In contrast to CPU

programming model that is based on instruction, programmable GPU model is based on processing data streams [8]. The core of the latter processing style is that the processor is first configured by the instructions and then a data stream is processed by a number of depth-parallel units in a pipeline. Given sufficient memory bandwidth and multiple processing pipelines, data processing can also be parallelized in breadth by distributing the execution among several pipelines. Commodity GPUs adopt breadth parallelism using Single Instruction Multiple Data (SIMD) processing units with multiple-component vectors. Their pipeline arrangements are similar to vector processors with multiple pipelines.

As stated in Section 2, the DWT is practically done by an input signal filtering and a down-sampling step. The down-sampling step need to locate texture coordinates correctly when issuing texture mapping instructions. For this problem, a direct solution is to issue position-dependent down-sampling by means of predicated execution and indirect texture lookup with pre-computed texture coordinates. Wang et al. adopted this approach in their GPU implementation of the Filter Bank Scheme (FBS) of DWT [9]. Besides down-sampling, they also used this approach to implement image edge expansion that is essential in the DWT. The shortcoming of this approach is that it does not efficiently exploit all the hardware resources available in current GPUs such as the hardware interpolators of texture coordinates. Furthermore, predicated execution limits the effective performance of the GPU's fragment programming. Except filtering, modern graphics hardware also supports resampling for image transfer operation, which we will utilize for down-sampling in DWT. Thus our solution for hardware based DWT will implement image edge expansion, filtering and down-sampling on GPU without pre-computing texture coordinates or establishing texture lookup tables in advance.

In common GPU programming model, the GPU performs computations through the use of streams and kernels. A stream, that is abstracted as texture, is an ordered collection of elements requiring similar processing. A kernel, that is expressed as texture, is a data-parallel function that processes input streams and produces new output streams. The fragment programs are coded using a high-level shading programming language such as Cg. However, we must still use a 3D graphics API such as OpenGL to organize data into streams, transfer those data streams to and from the GPU as 2D textures, upload kernels, and perform the sequence of kernel calls dictated by the application flow. Based on these notations, the flow of hardware based DWT can be summarized as follow.

Step 1: Upload the raw data/image and parameters of the high and lowpass filters into GPU's memory to form textures respectively;

Step 2: Invoke the boundary extension kernel to extend the edge of raw data/image, which includes the left, right, top, and bottom edge extension. The actual extension length is determined by kernel length of the decomposition filter. Suppose the kernel length of the decomposition filter is j then the extension length should be j –1. Fig.6 shows the results of symmetrical periodic extension where j=3.

Step 3: Invoke the filtering and down-sampling kernel to obtain approximation and detail coefficients,i.e., cA_j, cH_j, cV_j, and cV_j , at level j. Fig.7 shows the OpenGL commands and Cg code that issue horizontal filtering and down-sampling (Vertical filtering and down-sampling are analogous with those in horizontal direction.

Step 4: Store the approximation and detail coefficients of the various decomposition levels in corresponding location of a colour buffer for displaying.

4 PERFORMANCE EVALUATION

We evaluated our framework of hardware based DWT with Daubichies 4 wavelet on a PC equipped with Nvidia's GeForce 7900 GTX. The test image has a size of 800×432 that is shown in Fig.8. Fig.9-11 show the decomposition results at level 1, 2, 3 by using our proposed framework.

The execution time of our hardware based DWT was compared with the software based DWT. We used two images with the size of 800×432 and 1280×960 for comparison respectively. Table 1 lists the execution time and accelerating factor at decomposition level 1, 2 and 3. It can be seen from Table 1 that the hardware based DWT has a better accelerating factor when the data size is larger and/or the decomposition level is lower. As the decomposition level increases, the image size is reduced in a

ratio of a quarter, thus the acceleration performance of hardware based DWT is not as great as that at lower levels.

Table 1. Execution times in ms per 2D wavelet step

	800×432 image			1280×960 image		
	Level 1	Level 2	Level 3	Level 1	Level 2	Level 3
Software based DWT	100.8ms	9.25ms	0.77ms	834.6ms	69ms	3.8ms
Hardware based DWT	16ms	3.7ms	0.7ms	78ms	15ms	2ms
Accelerating factor	6.3	2.5	1.1	10.7	4.6	1.9

5 CONCLUSIONS AND FUTURE WORK

We have developed a simple and cost-effective solution to implement 2D DWT on the consumer-level GPU. It can be implemented on any SIMD-based GPU, which are commonly included in PCs. Different wavelet filter kernels and boundary extension schemes can be easily incorporated by modifying the filter kernel values and parameter respectively. We have demonstrated it achieved a great acceleration performance in contrast to the software based DWT.

Reconstruction is an inverse process of decomposition in wavelet transform. The next stage of our work will be on the issue of using a GPU for reconstruction by making use of the existing framework, with the key task of accomplishing up-sampling for reconstruction. We will also try to find an efficient partitioning of the algorithms for wavelet transform. We will investigate which part of the work will be done on GPU and which can be left to CPU. Our goal is to perform the whole transformation on the GPU in order to free up the CPU for other processing tasks. Under this assumption, the partitioning should enlarge the size of the streams to maximize the granularity of the parallelism and leave only the flow-control activity to the CPU.

REFERENCES

[1] Acharya T., Tsai P. JPEG2000 *Standard for Image Compression: Concepts, Algorithms and VLSI Architectures,* Wiley-Interscience, 2004.

[2] Sweldens W., *The Lifting Scheme: A Construction of Second Generation Wavelets*, SIAM J. Math. Analysis, vol. 29, no. 2, pp. 511-546.

[3] Vishwanath M., Owens R. M. (1995), *VLSI architecture for the discrete wavelet transform*, IEEE Transactions on Circuit & System, Vol.42, No.5, pp.305-316.

[4] Owens J. D., Luebke D., Govindaraju N., et al.(2007), *A Survey of General-Purpose Computation on Graphics Hardware*, Computer Graphics Forum, Vol. 26, No. 1,pp. 80 -113.

[5] Antonini M., Barlaud M., Mathieu P. (1992), *Daubechies Image Coding Using Wavelet Transform*, IEEE Transactions on Image Processing, Vol.17, No.1, pp.205-230.

[6] Gonnet C., Torresani B. (1994), *Local frequency analysis with two-dimensional wavelet transform, Signal Processing*, Vol. 37 , No. 3, pp.389-404.

[7] Montrym J., Moreton H.(2005), *The GeForce 6800*, IEEE Micro Magazine, Vol.25, No.2, pp. 41-51.

[8] Strzodka R., Doggett M., Kolb A.(2005), *Scientific Computation for Simulation on programmable Graphics Hardware*, Simulation Modelling Practice and Theory, Vol.13, No.8, pp.667-681.

[9] Wong T. T., Leung C. S., Heng P. A., and Wang J. Q.(2007), Discrete Wavelet Transform on Consumer-Level Graphics Hardware, IEEE Transaction on Multimedia, Vol. 9, No. 3, pp. 668-673.

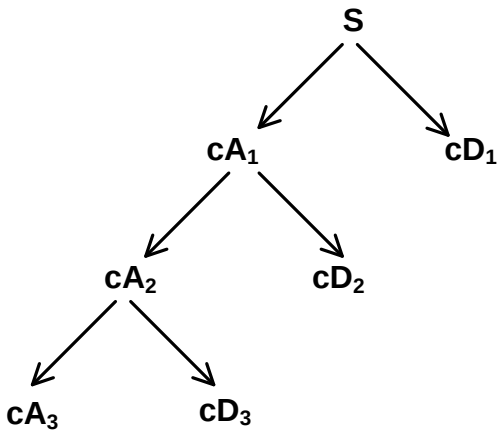

Fig.1 The decomposition tree of 1D-DWT

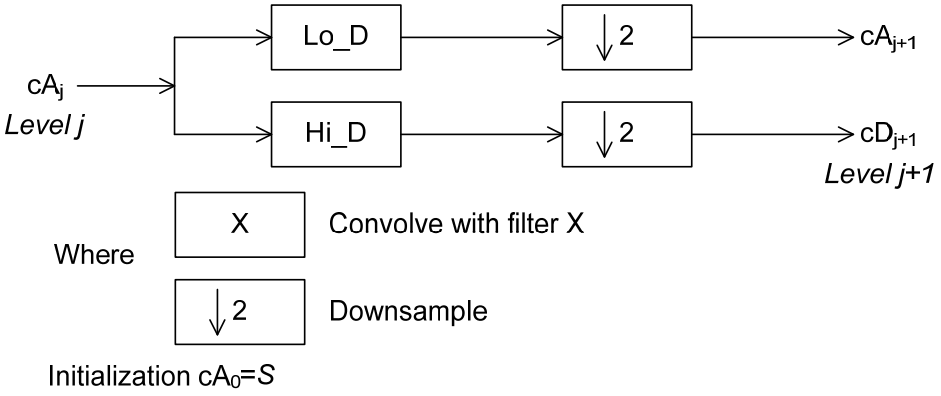

Fig.2 The procedure of decomposition in 1D-DWT

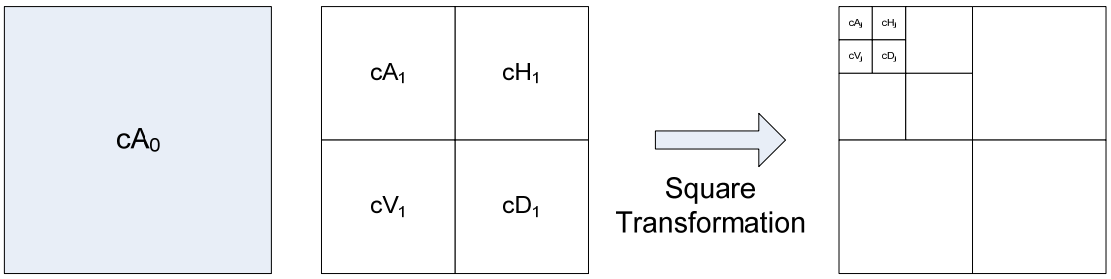

Fig.3 Square decomposition

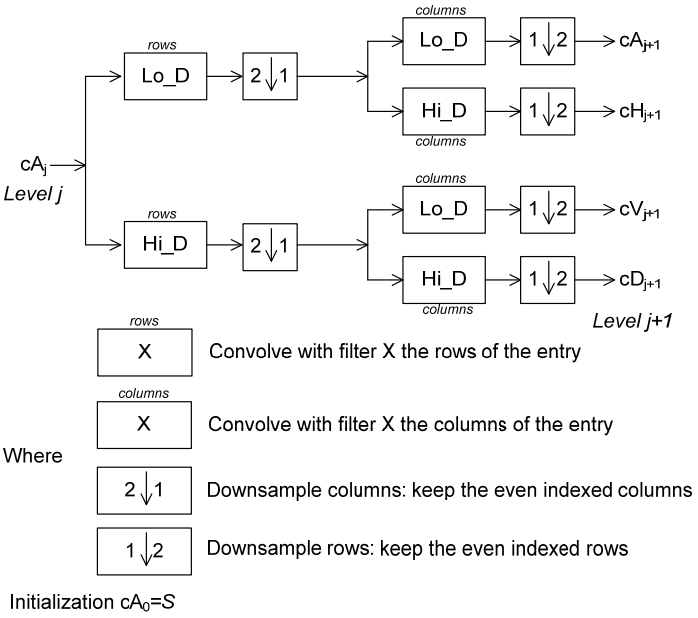

Fig.4 The procedure of decomposition in 2D-DWT

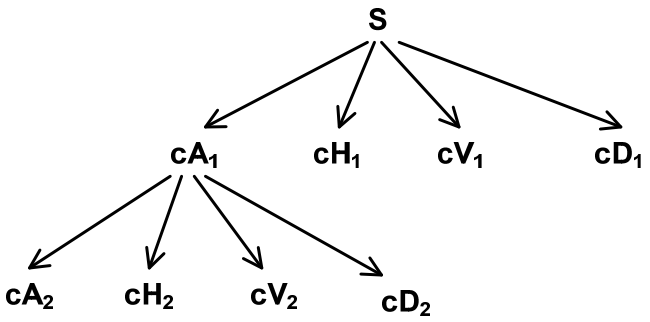

Fig.5 The decomposition tree of 2D-DWT

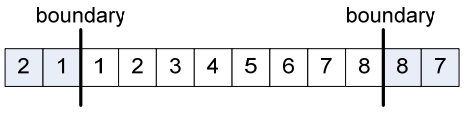

Fig.6 Boundary extension

Fragment program

```
fragout_float main(vf30 IN,
                   uniform samplerRECT image_texture,
                   uniform samplerRECT filter_texture)
{
   float3 sum=float3(0,0,0);

  // Convolving image data with filter kernel

  for (int i=0; i<8; i++)
  {
     sum += f3texRECT(filter_texture , float2(i+0.5,0.5)).r *
            f3texRECT(image_texture, float2((IN.TEX0.x+1.0)+i, IN.TEX0.y));
  }

  fragout_float OUT;
  OUT.col = float4(sum, 1.0);
  return OUT;
}
```

OpenGL commands

```
glBegin(GL_QUADS);
{
    glTexCoord2f(          0.0f,                  0.0f);
      glVertex2f (         0.0f,                  0.0f);

    glTexCoord2f( (float)(2*quadWidth),           0.0f);
      glVertex2f (     (float)quadWidth,          0.0f);
    glTexCoord2f((float)(2*quadWidth),   (float)quadHeight);
      glVertex2f ((float)quadWidth,      (float)quadHeight);

    glTexCoord2f(        0.0f,           (float)quadHeight);
      glVertex2f (       0.0f,           (float)(quadHeight);
}
glEnd();
```

Down-sampling according to ratio 2:1

Fig.7 OpenGL commands and Cg code for horizontal decomposition

Fig.8 The original image (800×432) for testing

Fig.9 Coefficients at decomposition level 1

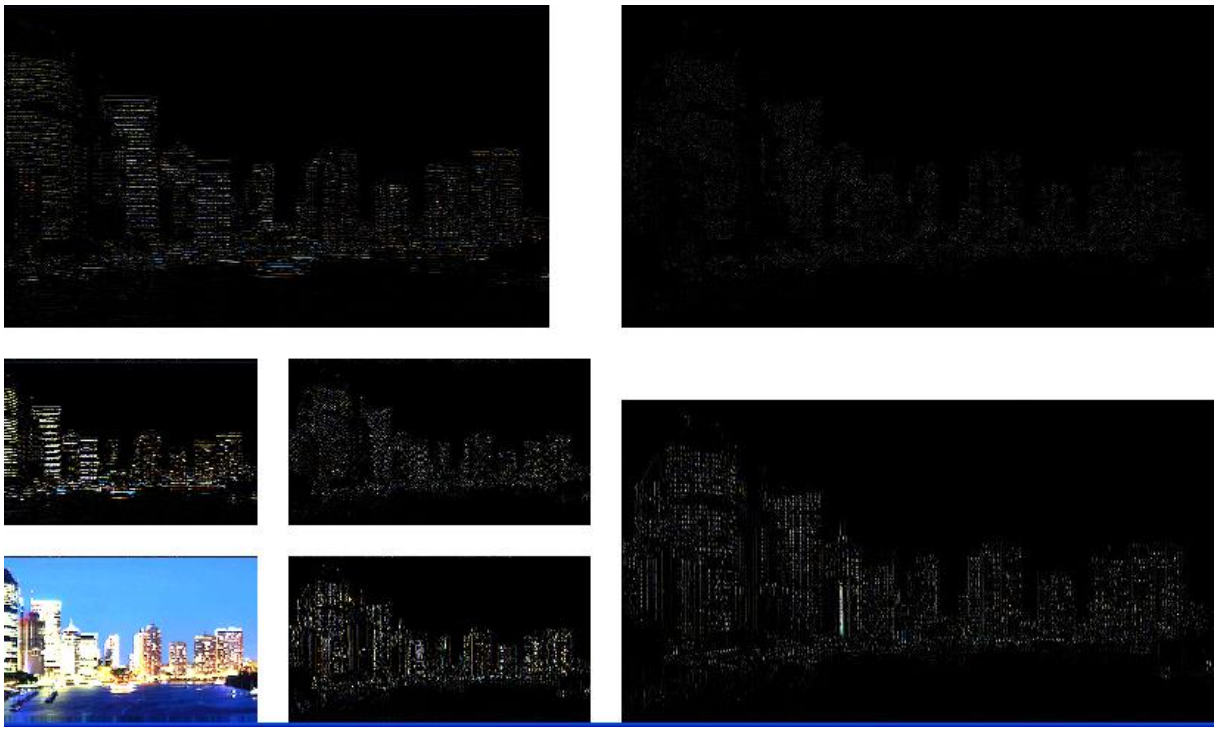

Fig.10 Coefficients at decomposition level 2

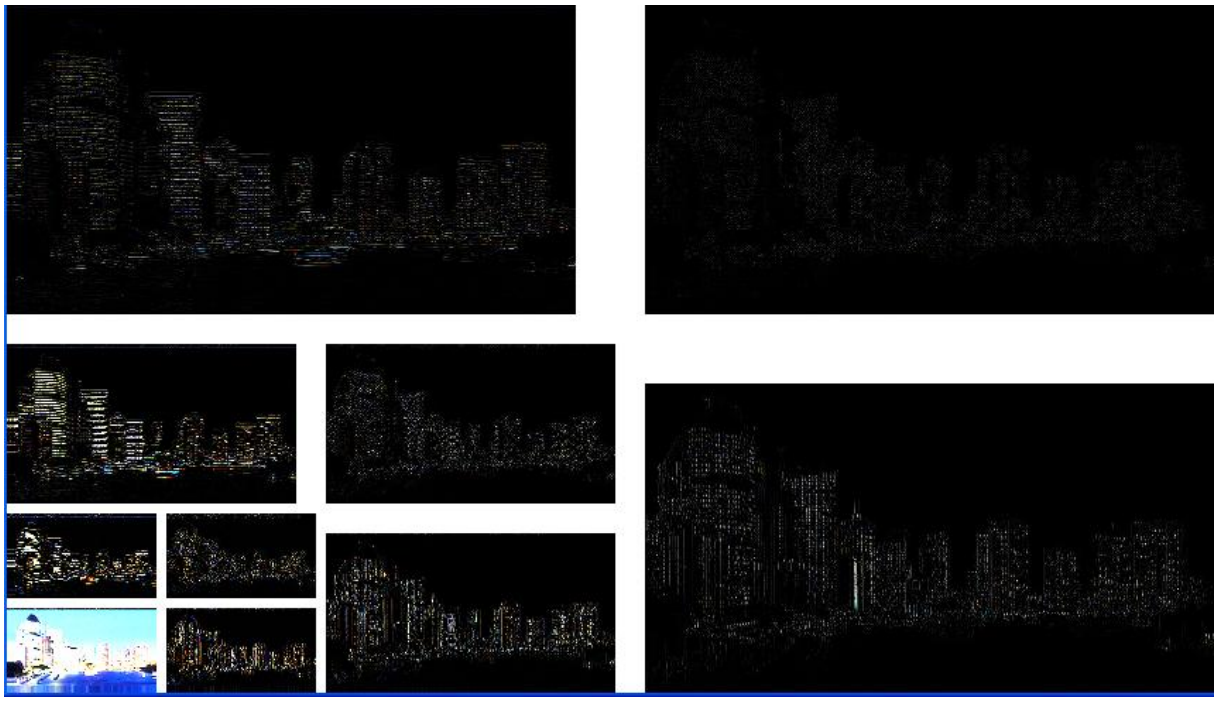

Fig.11 Coefficients at decomposition level 3