International Journal of Computer Applications (0975 - 8887)
Volume 183 - No.42, December 2021

GPU Implementation of Faber Schauder Discrete Wavelet Transform using CUDA

Assma Azeroual
Computer Science Department
Higher School of Education and Training
Ibn Zohr University

Karim Afdel
Computer Science Department
College of Sciences
Ibn Zohr University

ABSTRACT

Faber Schauder discrete wavelet transform (FSDWT) has many interesting advantages in image and video processing owing to its simplicity and its multiscale-based theory. It preserves the pixel ranges, has arithmetic operations, and detects edges in multiscale representation. With the increase of image size and the real-time requirement of many applications, the FSDWT computation becomes complex and needs other techniques to deal with it. To solve this problem, the FSDWT is implemented in parallel on a Graphics Processing Unit (GPU) using Compute Unified Device Architecture (CUDA) code. The results demonstrate that the GPU-based FSDWT exceedingly outperforms the CPU FSDWT.

Keywords

Image processing, FSDWT, GPU, CUDA, Multiscale transform

1. INTRODUCTION

Wavelets are the foundation of powerful approaches in signal processing and multi-resolution the [1, 2, 3, 4]. This later includes techniques from a variety of disciplines such as subband coding from signal processing, quadratic mirror filtering from digital speech, and image processing. The appeal of the multi-resolution theory is the detection of features at many levels of resolution [5].
Various generalizations of wavelets were introduced in [6, 7], many techniques for constructing wavelet transforms were proposed such as the lifting scheme [8, 9]. Since computers have just finite precision, floating point transforms are less interesting from the viewpoint of numerical computation. That is why integer wavelets have been widely used in image processing. Faber Schauder wavelet transform (FSWT) is a simple multiscale transformation that has many interesting properties in image processing such as preservation of pixel ranges, arithmetic operations, multiscale edge detection.
A Faber Schauder discrete wavelet transform (FSDWT) was proposed by Douzi et al. [10] for image characterization and edge detection, Amar et al. [11] had proposed similarly a FSDWT technique for edge detection. Furthermore, FSDWT was used in multiple techniques of image watermarking [12, 13, 14, 15, 16, 17]. In [18, 19] FSDWT was used for video processing and may also be used for image and video compression [20].

With the growth of technologies, computers operate on large data sets such as HD images or videos and real-time requirement becomes more demanded in many image and video processing tasks. Thus, the use of parallel processing becomes necessary to provide high-performance with a reasonable cost. As discussed before, various techniques used and defined the FSDWT, however, we are not aware of any work addressing the FSDWT optimization in Graphics Processing Unit (GPU).
In this paper, the work on implementing FSDWT in GPU using Compute Unified Device Architecture (CUDA) code is presented, the obtained performance is shown, and the execution time to find the optimal number of threads per block is analyzed. In addition, a results comparison between CPU and GPU is done. The rest of the paper is organized as follows. A small introduction of GPU is presented in section 2, then the FSDWT is detailed in section 3. The proposed method is described in section 4. The experimental results are described in details in section 5. Conclusions and future perspectives are drawn in section 6.

2. GPU

GPUs have been widely used in recent years as tools to process a large amount of information in parallel. As opposed to CPUs, GPUs execute multiple concurrent threads slowly, instead of executing a single thread very quickly. The GPU is especially well-suited to applications having data-parallel computations with high arithmetic intensity, hence, GPU is designed such that more transistors are devoted to data processing rather than data caching and flow control as illustrated by figure fig.1. In other words, the ratio of arithmetic operations must be higher than the memory operations. Otherwise, these applications might run slower on GPU than on CPU [21].
CUDA was introduced by NVIDIA as a general purpose parallel computing platform and programming model to solve efficiently various complex computational problems compared with CPU. The CUDA parallel programming model partitions a task into many sub-tasks that can be solved independently in parallel and each subtask into finer pieces that can be solved cooperatively in parallel. Every elemental task is running by a thread and all cooperating threads run parallelly in a thread block, which allows many blocks to run in parallel. Blocks are organized into one dimensional, two dimensional, or three dimensional grid of thread blocks as shown in figure fig.2. Like C functions, CUDA has its own functions called

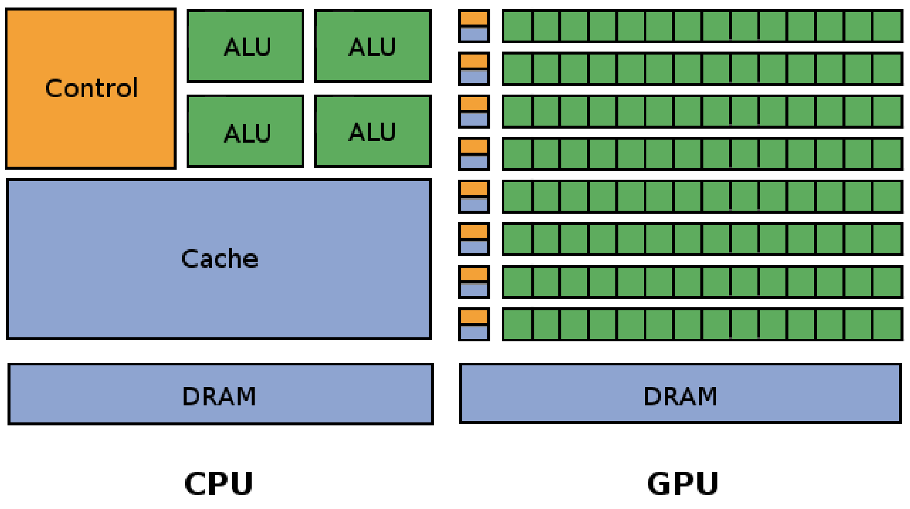

Fig. 1. More transistors are devoted to data Processing in GPU.

kernels, that, when launched, are executed N times in parallel by N different CUDA threads.
CUDA threads can access data from different memory spaces during their execution as shown in figure fig.3. Local and global memories are expensive to access compared with shared memory, the use of these three kinds of memory depends on the application needs. To obtain optimal results from GPU based computing, the data transfer between Host and Device must be minimized [22].

3. FSDWT

The algorithm of Faber Schauder transform is more simple to express based on lifting scheme [9].
Let $f^0 = (f_k^0)_{k \in Z}$ be a real sequence of a one dimensional signal. The transform can be done in three steps:

—Splitting : the sequence f^0 is decomposed into two disjoint sets of samples
$f^{1,0} = (f_{2k}^0)_{k \in Z}$ and $g^{1,0} = (f_{2k+1}^0)_{k \in Z}$

—Predicting: to use the correlation between odd and even coefficients ($f^{1,0}$ and $g^{1,0}$), the odd coefficients are predicted from a linear combination of the neighboring even coefficients, and the prediction residual is token as the sequence g^1:
$g_k^1 = g_k^{1,0} - \frac{1}{2}(f_k^{1,0} + f_{k+1}^{1,0})$

—Updating: the aim of this step is to preserve some original signal properties in the sequence f^1, for Faber Schauder transform the updating is a simple interpolation of:
$f^1 = f^{1,0}$

The advantage of the lifting scheme is the possibility to construct an integer version of the Faber Schauder transform by rounding off the prediction and updating operations. In addition, the range of pixel values is preserved. Moreover, Faber wavelet transform can be generalized easily to two dimensional signals. The lifting scheme of the transformation with n scales is given by:

$$\begin{cases} f^0 = f_{ij} \quad for\; i,j \in Z \\ for \quad 1 \leq k \leq n \\ f_{ij}^k = f_{2i,2j}^{k-1} \\ g_{ij}^k = (g_{ij}^{k1}, g_{ij}^{k2}, g_{ij}^{k3}) \\ g_{ij}^{k1} = f_{2i+1,2j}^{k-1} - \frac{1}{2}(f_{2i,2j}^{k-1} + f_{2i+2,2j}^{k-1}) \\ g_{ij}^{k2} = f_{2i,2j+1}^{k-1} - \frac{1}{2}(f_{2i,2j}^{k-1} + f_{2i+2,2j+2}^{k-1}) \\ g_{ij}^{k3} = f_{2i+1,2j+1}^{k-1} - \frac{1}{4}(f_{2i,2j}^{k-1} + f_{2i,2j+2}^{k-1} + \\ f_{2i+2,2j}^{k-1} + f_{2i+2,2j+2}^{k-1}) \end{cases} \qquad (1)$$

3.1 Implementation of FSDWT on CPU

In the case of discrete signals such as images, the FSDWT of an image I of dimensions $h \times w$ have been implemented by the algorithm 1.
where $fsdwt$ is the FSDWT of image I, $nbScales$ is the number of scales and m is an intermediate matrix for calculation. This algorithm gives a mixed-scales representation of an image and for each scale k it computes the FSDWT coefficients $m_{i,j}$ by adding 2^k to i and j in every iteration. Inside the first loop, there are three loops, the first one represents the g^{k1} vector, the second one represents g^{k2} vector and the third one represents the vector g^{k3}. The computation of f^k is done by dividing the sample by 2.
The FSDWT contains simple arithmetic operations, however, it becomes more complex when the image has a high resolution, and when we deal with videos.

4. THE PROPOSED APPROACH

The implementation of our proposed method consists of Host part and Device part. In the Host part, the image is just uploaded, copy it to Device and call the function FSDWT_host that will launch the Device computation. The result is then copied back to Host after the Device has completed its work. In the Device part, all the computations are done in three levels of parallelization. As mentioned before, the Host function FSDWT_host is responsible for launchin

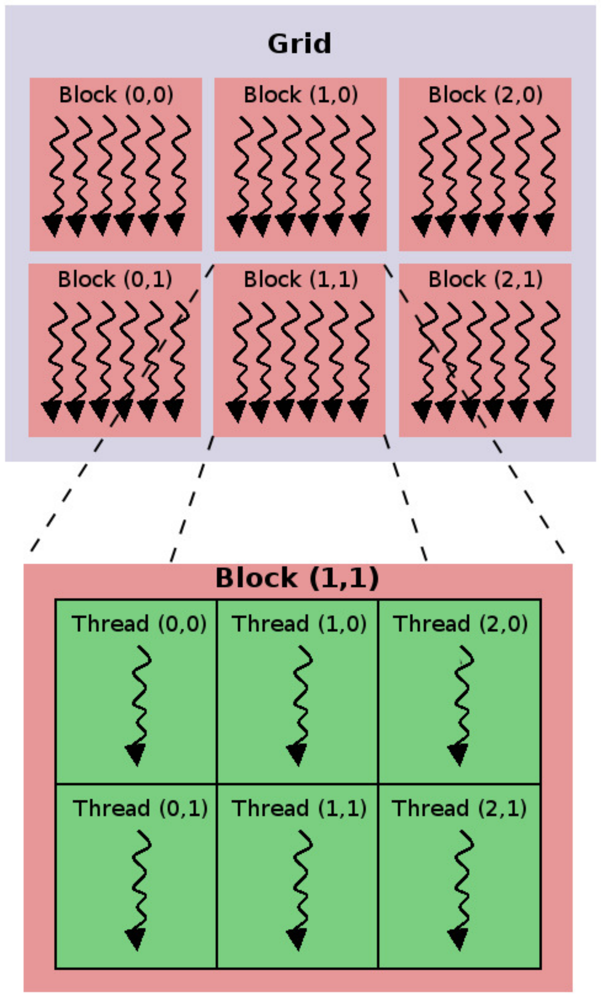

Fig. 2. Grid of thread blocks.

g the Device computation, which is done by starting the kernel FSDWT and executing it in parallel by $nbScales$ threads. Every instance of the kernel FSDWT launch three child kernels g_i^{k1}, g_i^{k2}, and g_i^{k3} which are executed by $b \times t$ threads in parallel for each child kernel. Finally, each instance of these child kernels launch other kernels g_{ij}^{k1}, g_{ij}^{k2}, and g_{ij}^{k3} using $b \times t$ threads for every kernel. The figure fig.4 shows an overview of the proposed method.

4.1 Host part

The Host part runs sequentially on the main CPU. All the basic information required are defined and fetched here. This includes reading the image, getting its height and width, then compute the image number of scales. After that, the image is copied from Host to Device as one dimensional vector. The next step is to launch the device first kernel and give it all the required information. As a last step, after the device computation, the Host get the image Faber Shauder discrete wavelet transform from the Device.

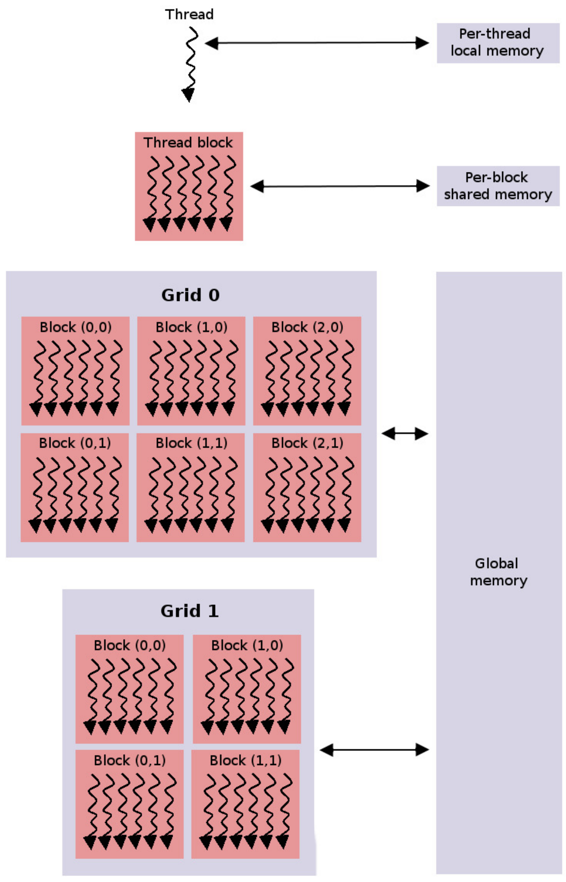

Fig. 3. Hierarchy of Memory.

4.2 Device part

let X be an image of dimensions $h \times w$. To simplify the description of this part the example $h = w = 9$ is token, the image X is shown in figure fig.5, where f_{ij} are the image pixels values in gray scale representation.
The number of scales for X is already computed in Host and is equal to $nbScales$, where $nbScales$ is the maximum integer verifying $2^{nbScales} < min(h-1, w-1)$, hence in our example $nbScales = 2$. Let us consider the equations system (1). In the first scale $k = 1$, the positions of the vectors g^{11}, g^{12}, g^{13}, f^1 elements are shown in figure fig. 6, the vector f^1 will have the form shown in figure fig. 7 and will be considered as the input data for the next scale. In the second scale $k = 2$, the positions of the vectors g^{21}, g^{22}, g^{23}, f^2 elements are shown in figure fig. 8. The mixed scales representation of all the coefficients is given in the figure fig. 9. By these examples, it is clear that the positions of scales coefficients do not coincide with each other, and the computation of a scale coefficients depends just on the original image pixels values and does not depend on the previous scales coefficients. Hence the FSDWT can be performed in parallel.
The first kernel which is called in the Device part is FSDWT, it is executed by $nbScales$ threads in parallel. The role of this kernel is to launch for each scale the child kernels g_i^{k1}, g_i^{k2}, g_i^{k3} that will compute the corresponding coefficients. Let us consider the previous example of the matrix X and let us take the figure fig. 9. Here,

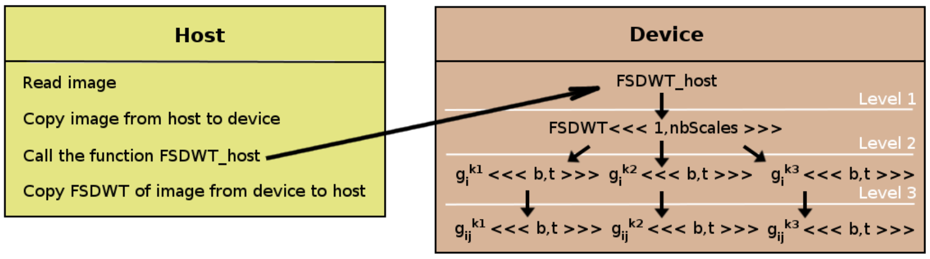
Host
Read image
Copy image from host to device
Call the function FSDWT_host
Copy FSDWT of image from device to host
Device
FSDWT_host
Level 1
FSDWT<<< 1,nbScales >>>
Level 2
g_i^k1 <<< b,t >>> g_i^k2 <<< b,t >>> g_i^k3 <<< b,t >>>
Level 3
g_ij^k1 <<< b,t >>> g_ij^k2 <<< b,t >>> g_ij^k3 <<< b,t >>>

Fig. 4. Overview of the proposed method.

```
$dim \leftarrow max(h, w)$
$nbScales \leftarrow 0$
while $2^{nbScales} \leq min(h-1, w-1)$ do
    $nbScales \leftarrow nbScales + 1$
end while
m $\leftarrow$ Initialize with $I$ values
for each $k$ from 1 to $nbScales$ do
    for each $i$ from $1 + 2^{k-1}$ to $dim - 2^{k-1}$ do
        for each $j$ from 1 to $dim$ do
            $m_{i-1,j-1} \leftarrow$
            $m_{i-1,j-1} - \left[\frac{1}{2}(m_{i-2^{k-1}-1,j-1} + m_{i+2^{k-1}-1,j-1})\right]$
            $j \leftarrow j + 2^k$
        end for
        $i \leftarrow i + 2^k$
    end for
    for each $i$ from 1 to $dim$ do
        for each $j$ from $1 + 2^{k-1}$ to $dim - 2^{k-1}$ do
            $m_{i-1,j-1} \leftarrow$
            $m_{i-1,j-1} - \left[\frac{1}{2}(m_{i-1,j-2^{k-1}-1} + m_{i-1,j+2^{k-1}-1})\right]$
            $j \leftarrow j + 2^k$
        end for
        $i \leftarrow i + 2^k$
    end for
    for each $i$ from $1 + 2^{k-1}$ to $dim - 2^{k-1}$ do
        for each $j$ from $1 + 2^{k-1}$ to $dim - 2^{k-1}$ do
            $m_{i-1,j-1} \leftarrow m_{i-1,j-1} - [\frac{1}{4}(m_{i-2^{k-1}-1,j-2^{k-1}-1} +$
            $m_{i+2^{k-1}-1,j-2^{k-1}-1} + m_{i-2^{k-1}-1,j+2^{k-1}-1} +$
            $m_{i+2^{k-1}-1,j+2^{k-1}-1}$ ]
            $j \leftarrow j + 2^k$
        end for
        $i \leftarrow i + 2^k$
    end for
    $fsdwt \leftarrow m$
end for
```

$$X = \begin{array}{c|ccccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline 0 & f_{00} & f_{01} & f_{02} & f_{03} & f_{04} & f_{05} & f_{06} & f_{07} & f_{08} \\ 1 & f_{10} & f_{11} & f_{12} & f_{13} & f_{14} & f_{15} & f_{16} & f_{17} & f_{18} \\ 2 & f_{20} & f_{21} & f_{22} & f_{23} & f_{24} & f_{25} & f_{26} & f_{27} & f_{28} \\ 3 & f_{30} & f_{31} & f_{32} & f_{33} & f_{34} & f_{35} & f_{36} & f_{37} & f_{38} \\ 4 & f_{40} & f_{41} & f_{42} & f_{43} & f_{44} & f_{45} & f_{46} & f_{47} & f_{48} \\ 5 & f_{50} & f_{51} & f_{52} & f_{53} & f_{54} & f_{55} & f_{56} & f_{57} & f_{58} \\ 6 & f_{60} & f_{61} & f_{62} & f_{63} & f_{64} & f_{65} & f_{66} & f_{67} & f_{68} \\ 7 & f_{70} & f_{71} & f_{72} & f_{73} & f_{74} & f_{75} & f_{76} & f_{77} & f_{78} \\ 8 & f_{80} & f_{81} & f_{82} & f_{83} & f_{84} & f_{85} & f_{86} & f_{87} & f_{88} \end{array}$$

Fig. 5. The example of a 9×9 image.

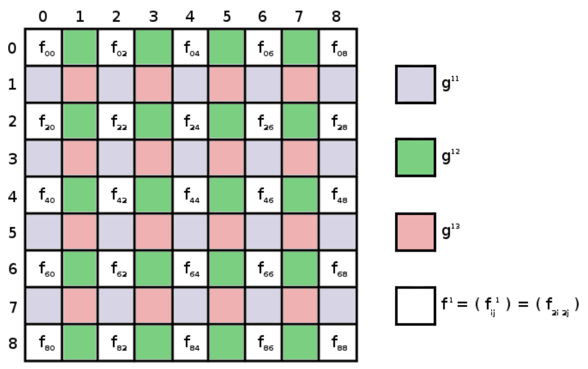

Fig. 6. The first scale coefficients in the example of a 9×9 image.

the FSDWT will be executed by two threads in parallel, one will compute the first scale coefficients colored in yellow and the second one will compute the second coefficients colored in brown. The kernels g_i^{k1}, g_i^{k2}, and g_i^{k3} will be run in parallel by $b \times t$ threads for each one of them, b and t correspond respectively to the number of blocks per grid and the number of threads per block. The role of these kernels is to compute the coefficients for each line of a given matrix by launching other kernels g_{ij}^{k1}, g_{ij}^{k2}, and g_{ij}^{k3} respectively.

The role of these later kernels is to calculate the coefficients in a given column in the line where the parent kernel launched the call. Let us consider again the example of the matrix X, and the figure fig. 6 for the first scale, here g_i^{k1} will be executed by four threads each one will take the computation of a blue line (ie a line of g^{11} by launching another kernel g_{ij}^{11} wich will compute -in the specified blue line- every coefficient by using one thread for it, the figure fig. 10 shows the application of our proposed approach on the example of the matrix X .

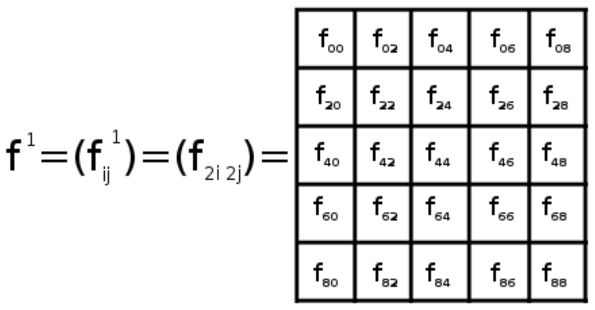
$f^1 = (f^1_{ij}) = (f_{2i\,2j}) =$
f_{00} f_{02} f_{04} f_{06} f_{08}
f_{20} f_{22} f_{24} f_{26} f_{28}
f_{40} f_{42} f_{44} f_{46} f_{48}
f_{60} f_{62} f_{64} f_{66} f_{68}
f_{80} f_{82} f_{84} f_{86} f_{88}

Fig. 7. The form of f^1 in the example of a 9×9 image.

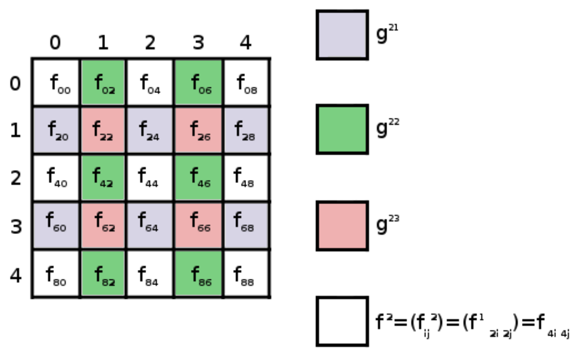

Fig. 8. The second scale coefficients in the example of a 9×9 image.

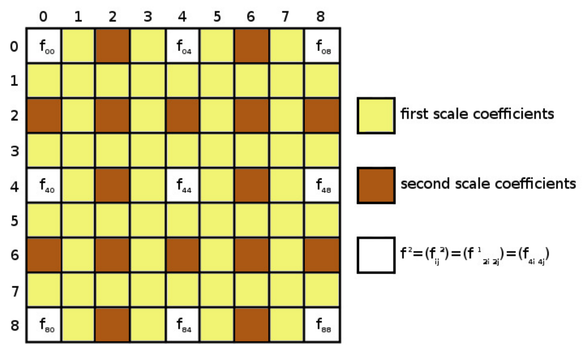
first scale coefficients
second scale coefficients
$f^2=(f^2_{ij})=(f^1_{2i\,2j})=(f_{4i\,4j})$

Fig. 9. The mixed scales representation of FSDWT coefficients in the example of a 9×9 image.

5. EXPERIMENTAL RESULTS

We analyze in this section the results obtained with the GPU implementation of FSDWT and compare them with the results obtained with the CPU implementation.
The experimental results are obtained using an Intel Core i7-4510U CPU and 2GB RAM with an NVIDIA GeForce 840M graphics card. These results have been fully implemented and optimized in OpenCV C++ and for the parallelized FSDWT using CUDA C++. We used one dimensional structure of grids and blocks of threads. We used five different images for the results performance, these images are of size 256×256, 512×512, 1024×1024, 2048×2048, 4096×4096.

5.1 CUDA implementation of FSDWT

In the CPU part, the image is firstly uploaded , getting its information, then copying the image from Host to Device by using the following code:

```
int dim = max(h,w);
int size = dim*dim*sizeof(int);

int *h_m, *d_m;

h_m = (int*)malloc(size);
cudaMalloc((void**)&d_m,size);

for(int i=0;i<dim;i++)
 for(int j=0;j<dim;j++)
  h_m[i*w+j] = image.at<uchar>(i,j);

cudaMemcpy(d_m,h_m,size,cudaMemcpyHostToDevice);
```

The next step is to launch the FSDWT kernel by calling the function FSDWT_host from Host:

```
extern "C" void FSDWT_host(int *A,int *B,int h,int w,int
     dim, int nbScales)
{
 FSDWT <<< 1 , nbScales >>> (A,B, h, w,dim,nbScales);
}
```

where A presents a pointer to the image in GPU memory, B presents a pointer to the image FSDWT in GPU memory. The kernel FSDWT has the following code:

```
__global__ void FSDWT(int *A, int *B ,int h, int w, int
     dim, int nbScales)
{
 __shared__ int threadsPerBlock;
 threadsPerBlock = 256;

 int k = threadIdx.x + 1;
 float p = powf(2,k-1);

 int blocksPerGrid = (dim/(2*p) + threadsPerBlock -1) /
      threadsPerBlock;

 gk1_i<<<blocksPerGrid,threadsPerBlock>>>(A,B,h,w,dim,p);

 gk2_i<<<blocksPerGrid,threadsPerBlock>>>(A,B,h,w,dim,p);

 gk3_i<<<blocksPerGrid,threadsPerBlock>>>(A,B,h,w,dim,p);
}
```

The kernels gk1_i, gk2_i, gk3_i, gk1_ij, gk2_ij, and gk3_ij have the following code:

```
__global__ void gk1_ij(int *A,int *B,int h,int w,int dim,
     float p,int i)
{
 int u = blockDim.x*blockIdx.x + threadIdx.x;
 int j=1+2*p*u;
 if(j<=dim)
  B[(i-1)*w +j-1] = A[(i-1)*w+j-1] - floorf(0.5*( A[(i-(
      int)p-1)*w+j-1] + A[(i+(int)p-1)*w+j-1]));
}
__global__ void gk2_ij(int *A,int *B,int h,int w,int dim,
     float p,int i)
{
 int u = blockDim.x*blockIdx.x + threadIdx.x;
```

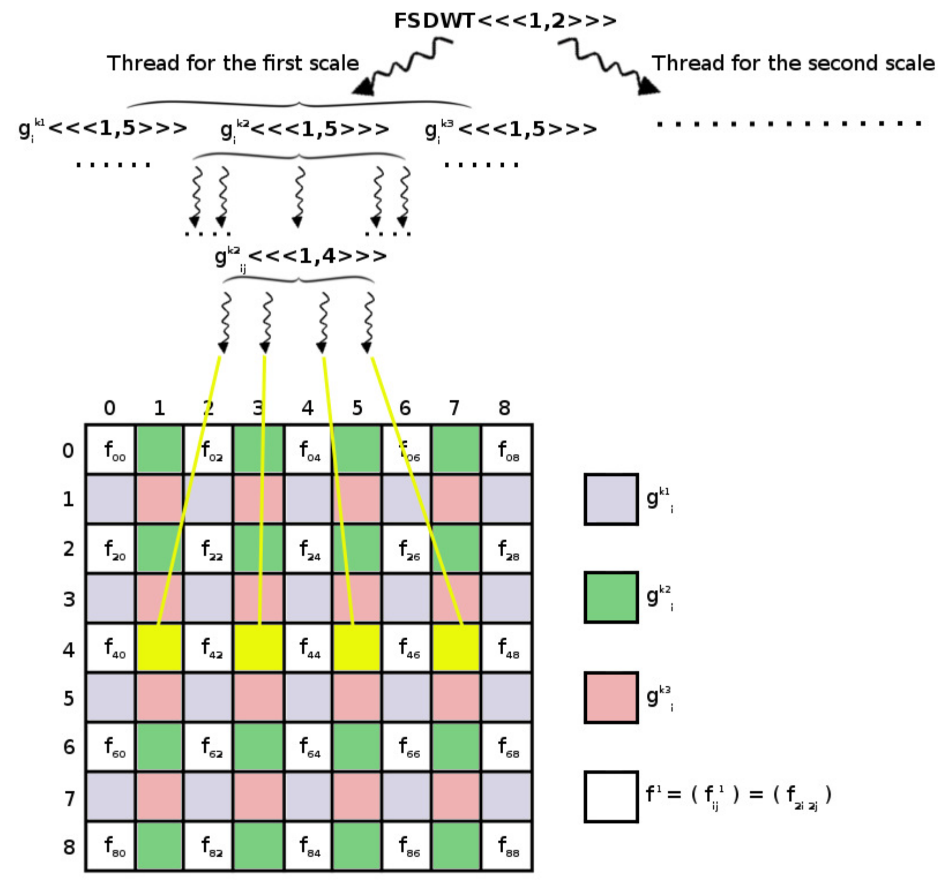

Fig. 10. The parallel FSDWT applied on the matrix X.

```
 int j=1+p+2*p*u;
 if(j<=dim-p)
  B[(i-1)*w +j-1] = A[(i-1)*w+j-1] - floorf(0.5*( A[(i-1)*
       w+j-(int)p-1] + A[(i-1)*w+j+(int)p-1]));
}
__global__ void gk3_ij(int *A,int *B,int h,int w,int dim,
      float p,int i)
{
 int u = blockDim.x*blockIdx.x + threadIdx.x;
 int j=1+p+2*p*u;
 if(j<=dim-p)
  B[(i-1)*w +j-1] = A[(i-1)*w+j-1] - floorf(0.25*( A[(i-(
       int)p-1)*w+j-(int)p-1] + A[(i+(int)p-1)*w+j-(int)p
       -1]
                        + A[(i-(int)p-1)*w+j+(int)p-1] +
                              A[(i+(int)p-1)*w+j+(int)p
                             -1]));
}
__global__ void gk1_i(int *A,int *B,int h,int w,int dim,
      float p)
{
 __shared__ int threadsPerBlock;
 __shared__ int blocksPerGrid;
 threadsPerBlock = 256;
 blocksPerGrid = (dim/(2*p) + threadsPerBlock -1) /
      threadsPerBlock;
```

```
 int v = blockDim.x*blockIdx.x + threadIdx.x;
 int i=1+p+2*p*v;
 if(i<=dim-p)
   gk1_j<<<blocksPerGrid,threadsPerBlock>>>(A,B,h,w,dim,p,
        i);
}
__global__ void gk2_i(int *A,int *B,int h,int w,int dim,
      float p)
{
 __shared__ int threadsPerBlock;
 __shared__ int blocksPerGrid;
 threadsPerBlock = 256;
 blocksPerGrid = (dim/(2*p) + threadsPerBlock -1) /
      threadsPerBlock;
 int v = blockDim.x*blockIdx.x + threadIdx.x;
 int i=1+2*p*v;
 if(i<=dim)
   gk2_j<<<blocksPerGrid,threadsPerBlock>>>(A,B,h,w,dim,p,
        i);
}
__global__ void gk3_i(int *A,int *B,int h,int w,int dim,
      float p)
{
 __shared__ int threadsPerBlock;
 __shared__ int blocksPerGrid;
 threadsPerBlock = 256;
```

Table 1. Execution time (ms) for the FSDWT in CPU and in GPU

Image size	CPU time	GPU time	Speed-up
256 × 256	28.12	88.97	-
512 × 512	96.70	110.95	-
1024 × 1024	329.49	120.74	2.72
2048 × 2048	1264.92	180.74	6.99
4096 × 4096	5037.70	429.50	11.72

Table 2. Detailed execution time (ms) for FSDWT in CPU and in GPU

Operation	256 × 256	4096 × 4096
Copy from Host to Device	83	3244
GPU FSDWT	0.015	0.026
CPU FSDWT	23	4403

Table 3. FSDWT kernel execution time (μs) at various block threads number for 256 × 256 image

# threads	32	64	128	256	512	1024
Time	18.66	19.66	19	16	16	18

```
    blocksPerGrid = (dim/(2*p) + threadsPerBlock -1) /
        threadsPerBlock;
    int v = blockDim.x*blockIdx.x + threadIdx.x;
    int i=1+p+2*p*v;
    if(i<=dim-p)
      gk3_j<<<blocksPerGrid,threadsPerBlock>>>(A,B,h,w,dim,p,
          i);
}
```

5.2 Time evaluation

The comparison between the FSDWT time execution in CPU and in GPU using our proposed method is given in the following table: By analyzing the results shown in table tab.1 we noticed that the proposed method performs well since the image size is large. This due to the time spent to copy the data from Host to Device and from Device to Host, which is a normal issue in GPU computing. The table tab.2 shows the detailed time execution for the images of size 256 × 256, 4096 × 4096 on CPU and on GPU.
We have also investigated the optimum number of threads per block in GPU to implement FSDWT in parallel. Tables tab.3-tab.7 show the execution time of FSDWT kernel for different image sizes using a various number of threads per block. As shown in figure fig.11, threads=256 per block shows the minimum time of execution for any image size. Hence the use of 256 threads per block provides an optimal time for the proposed approach. (see Sect. 4.1).

6. CONCLUSIONS AND FUTURE WORK

This paper implements the FSDWT in parallel on GPU using CUDA code, this implementation outperforms the CPU FSDWT for large size images. Experimental results indicated that threads = 256 per block provides an optimal execution time.
Future work will include the use of GPU FSDWT in many image and video processing such as edge detection, watermarking, and compression. Moreover, another GPU FSDWT implementation will be proposed to work in most GPU architectures using recent techniques of parallel computing.

Table 4. FSDWT kernel execution time (μs) at various block threads number for 512 × 512 image

# threads	32	64	128	256	512	1024
Time	19	20.33	19.33	17.33	20.33	20.4

Table 5. FSDWT kernel execution time (μs) at various block threads number for 1024 × 1024 image

# threads	32	64	128	256	512	1024
Time	22.33	23	22.33	21.33	22.33	23.33

Table 6. FSDWT kernel execution time (μs) at various block threads number for 2048 × 2048 image

# threads	32	64	128	256	512	1024
Time	24.66	25.33	24.66	24.33	25	26.33

Table 7. FSDWT kernel execution time (μs) at various block threads number for 4096 × 4096 image

# threads	32	64	128	256	512	1024
Time	26	28.33	26	24.66	26	31

7. REFERENCES

[1] S.G. Mallat, A theory for multiresolution signal decomposition: The wavelet representation, (IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 11(7), 1989).

[2] S.G. Mallat, Multifrequency channel decompositions of images and wavelet models, (IEEE Trans. on Acoustics Speech and Signal Processing, vol. 37(12), 1989) p. 2091–2110.

[3] S.G. Mallat, A Wavelet Tour of Signal Processing, (Academic Press, New York, 1998).

[4] S.G. Mallat and S.G., Zhong, Characterization of signals from multiscale edges, (IEEE Trans. on Pattern Analysis and Machine Intelligence , vol. 14(7), 1992).

[5] R.C. Gonzalez and R.E. Woods, Digital Image Processing (Third Edition), (976, Prentice Hall, United States Edition, 2007).

[6] A. Aldroubi and M. Unser Families of multiresolution and wavelet spaces with optimal properties, (Numerical Functional Analysis and Optimization, vol. 14(5-6), 1993) p. 417–446.

[7] A. Cohen, I. Daubechies and J. C. Feauveau Biorthogonal bases of compactly supported wavelets, (Comm. Pure Appl. Math., vol. 45, 1992) p. 485–560.

[8] W. Sweldens The lifting scheme: A custom-design construction of biorthogonal wavelets, (Applied and computational harmonic analysis, vol. 3(2), 1996) p. 186–200.

[9] W. Sweldens The lifting scheme: A construction of second generation wavelets, (SIAM J. Math. Anal., vol. 29(2), 1997) p. 511–546.

[10] H. Douzi, D. Mammass and F. Nouboud Faber-Schauder wavelet transform, application to edge detection and image characterization, (Journal of Mathematical Imaging and Vision, vol. 14(2), 2001) p. 91–101.

[11] M. Amar, R. Harba, H. Douzi, F. Ros, M. El Hajji, R. Riad and K. Gourrame A JND Model Using a Texture-Edge Selector Based on Faber-Schauder Wavelet Lifting Scheme, (Image and Signal Processing: 7th International Conference, ICISP 2016, Trois-Rivières, QC, Canada, May 30 - June 1, 2016, Proceedings, 2016) p.328–336.

[12] M. El hajji, H. Douzi and R. Harba Watermarking Based on the Density Coefficients of Faber-Schauder Wavelets, (Image

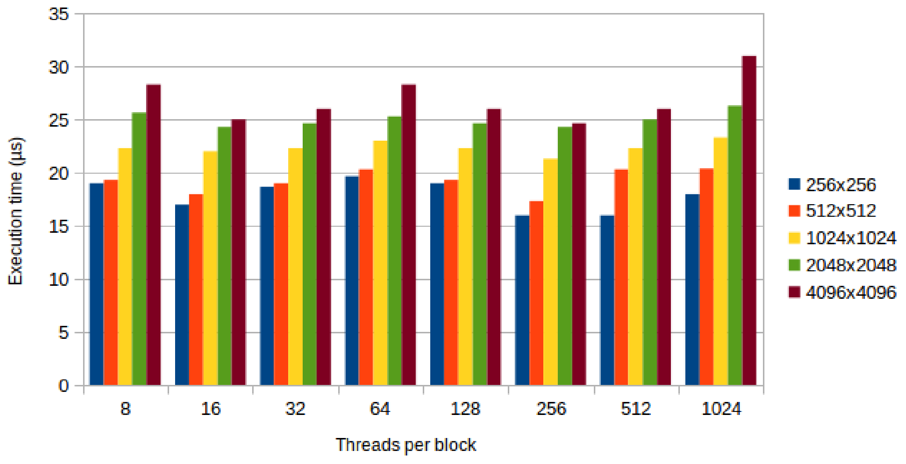

Fig. 11. Execution time of FSDWT kernel for different number of threads per block with different image sizes

and Signal Processing: 3rd International Conference, ICISP 2008. Cherbourg-Octeville, France, July 1 - 3, 2008. Proceedings) p. 455–462.

[13] M. El Hajji, H. Douzi, D. Mammass and R. Harba A robust wavelet-based watermarking algorithm using mixed scales, (Multimedia Computing and Systems (ICMCS), 2011 International Conference on, 2011) p. 1–5.

[14] A. Azeroual and K. Afdel Low Complexity Image Authentication Based on Singular Value Decomposition and Mixed Scales Faber Schauder Wavelet, (International Review on Computers and Software (IRECOS), vol. 10(12), 2015) p. 1209–1215.

[15] A. Azeroual and K. Afdel Image Authentication Based on Faber Schauder DWT, (2016 13th International Conference on Computer Graphics, Imaging and Visualization (CGiV), Beni Mellal), p. 78–83.

[16] A. Azeroual and K. Afdel A Faber Schauder dominant blocks based fragile watermarking scheme for image tamper detection, (2016 International Conference on Electrical and Information Technologies (ICEIT), Tangiers), p. 380–385.

[17] A. Azeroual and K. Afdel A Fragile Watermarking Scheme for Image Authentication Using Wavelet Transform, (International Conference on Image and Signal Processing ICISP: Image and Signal Processing, 2016) p. 337–345.

[18] A. Azeroual, K. Afdel, M. El Hajji and H. Douzi, On line Key Frame Extraction and Video Boundary Detection using Mixed Scales Wavelets and SVD, (International Journal of Circuits, Systems and Signal Processing, 9, 2015) p. 420–426.

[19] A. Azeroual and K. Afdel Key Frames Based Video Authentication Using Fragile Watermarking and Singular Value Decomposition, (International Review on Computers and Software (IRECOS), vol. 11(5), 2016) p. 420–426.

[20] M. Benabdellah, M. Gharbi, F. Regragui and E. H. Bouyakhf, Choice of reference images for video compression, (Applied Mathematical Sciences, vol. 1(44), 2007) p. 2187–2201.

[21] CUDA C Programming Guide, PG-02829-001 v8.0, September 2016.

[22] CUDA C Best Practices Guide, DG-05603-001 v8.0, September 2016.