Performances of a Parallel CUDA Program for a Biorthogonal Wavelet Filter

Antony Scivoletto
University of Catania
Viale A. Doria 6, 95125 Catania, Italy
antonyscivoletto@gmail.com

Nella Romano
University of Catania
Viale A. Doria 6, 95125 Catania, Italy
nromano919@gmail.com

***Abstract*—Parallel high-performance computing technologies have encountered tremendous growth, especially in the last decade, and they have made a strong impact in a variety of areas concerning mathematical and engineering fields.**

In this work we analyze the behavior of a parallel algorithm developed to compute a random signal processing through a biorthogonal discrete wavelet filter using CUDA and we compare the parallel implementation against similar sequential CPU code. The simulations are conducted using two computers: the first one is a conventional host; the second one is a server. This comparison allows us to underline the differences between hardware architectures with different specifications in terms of time performance. The experimental results show that if the signal is decomposed into a rather substantial number of samples, parallel code timings on server and a usual host GPU win on the corresponding sequential code on the same machines.

***Index Terms*—GPU programming, performances, wavelet**

I. Introduction

The last decades have been featured by large motions in the perceived value of parallel computing. In such years high-performance computing technologies have seen substantial growth and this is the case for general purpose computing on GPU, or rather GPGPU. The intensive use of GPU parallelization techniques has given the possibility to implement complex simulation and hard computational tasks [1], [2], [3], [4], [5], [6]. For much time, one of the most important methods used to improve the computational capability of devices has been increasing the processor clock speed. Unfortunately, because of various fundamental limitations in the manufacturing of integrated circuits, it is not possible to extract much additional computational power from existing architectures by increasing the processor clock speed. Then, supercomputer manufactories were able to introduce many changes in performance by strongly increasing the number of processors: they could have tens or hundreds of thousands of processor cores working concurrently. CPU manufacturers have announced plans for 12 or 16 cores CPUs. In this way, a big number of simple multi-thread cores in GPU cards could offer the potential for strong speedups regarding several general purpose applications, if compared with the CPU similar sequential computation, confirming parallel computing has arrived for good [7].

Nevertheless, the architecture of GPU cards gives rise to some programming issues, such as *sharing resources*, which are managed through a lock or token. In shared memory machines, a single address space and global memory are shared between multiple processors; each processor owns a local cache, and its values have to be made coherent with the global memory by the operating system. Data can be exchanged among processors simply by placing the values in a predefined location and synchronizing appropriately [8]. Other issues involved, which complicate development and have no counter part in the sequential world, can include: finding and expressing concurrency, managing data distributions, managing communication among processors, balancing computational load, and simply implementing the parallel algorithm correctly. Therefore, implementing parallel programs is more difficult than implementing sequential ones [8], [9]. In fact, in sequential programming, the programmer has to implement a code correctly, and efficient to execute. Parallel programming involves the same issues, and adds several additional challenges, some of which are described above.

In this paper we implement a parallel code by CUDA architecture, subduing a random signal to a biorthogonal discrete low pass wavelet filter. In particular, this work is structured as follows. In section 2 we present some essential concepts for introducing wavelet theory and we refer to specific techniques, such as subband coding, in order to process a given signal with the biorthogonal wavelet filter in discrete time case. In section 3 we describe the main features of CUDA architecture, used in this work for the parallelization procedure. In section 4 some basic concepts on the definition of a digital filter and its implementation are given through mathematical convolution operation. Section 5 examines serial and parallel algorithms, with which wavelet filter implementation and signal processing are achieved. In section 6 we show the results obtained by experimental simulations and compare time performance of two codes executed in two machines with different technical specifications. Finally, in the last two sections we deduce consequent results, conclusions and we also report all those works related to subject matter described in this paper.

II. Wavelet theory

A. Preliminaries

Only in recent years, wavelet theory has been developed as a unifying framework for a large number of techniques thought

for wave signal processing applications, such as multiresolution analysis, subband coding and wavelet series expansions. The idea of treating a signal at various scales and analyzing it with different resolutions has emerged independently in many mathematics, physics and engineering fields [10]. The interest of the signal processing community became strong when Doubechies and Mallat, giving a further contribution to the wavelet theory, established connections to discret signal processing results. Since then, a number of theoretical as well as pratical contributions have been made on various aspects of this topic and nowadays it is growing rapidly [10], [11].

B. Wavelets in continuos domain

Wavelets are continuos basic functions constructed mainly in order to satisfy certain mathematical properties. A wavelet is defined as a function with zero mean value which is both above and below the x-axis, looking like a "wave" [12]. It is possible to give a more rigorous definition of a wavelet. In fact, if we consider a multiresolution decomposition of $L^2(\mathbb{R})$, that is

$$\emptyset \subset V_0 \subset \cdots \subset V_j \subset V_{j+1} \subset \cdots \subset L^2(\mathbb{R})$$

and we call W_j the orthogonal complement of V_j, then we can define a wavelet as a function $\psi(x)$ such that the set of $\{\psi(x-l), l \in \mathbb{Z}\}$ is a Riesz basic of W_0 and also must be subject to the following constrains [13]:

1) $\int_{-\infty}^{+\infty} \psi(x)\, dx = 0$

2) $\| \psi(x) \|^2 = \int_{-\infty}^{+\infty} \psi(x)\psi^*(x)\, dx = 1$

The idea behind wavelets is that by stretching and translating one such main wavelet function $\psi(t)$ (called mother wavelet), we can represent finer parts of a function or a signal by simple linear combinations:

$$f(t) = \sum_{j,k} b_{j,k}\, \psi_{j,k}(t) \tag{1}$$

where $b_{j,k}$ are called *wavelet coefficients* of the function f in the wavelet basis given by the inner product of $\psi_{j,k}$. Moreover, a whole family of wavelet functions can be obtained by just shifting and scaling the mother one by the law [12]:

$$\psi_{j,k} = \sqrt{2^j}\, \psi(2^j t - k), \quad i, j \in \mathbb{N} \tag{2}$$

In the procedure of wavelet dilatation and translation, we can see that for a large j, $\psi_{j,k}(t)$ is short and of high frequency; smaller values for j, instead, give long wavelet functions of low frequency. This offers us the possibility to analyze a function in different scales [10]. Typical examples of wavelet functions are shown in figure 1.

C. Discrete Time Case

Now we focus our attention on discrete time signals, since it is the case of our interest for the implementation of a wavelet filter as a parallel program which exploits the computational capabilities of GPUs. In the discrete time case two methods were developed independently, namely *subband coding* and*multiresolution signal analysis*. Two notions which are important for wavelet theory are *scale* and *resolution*. Scale is related to the size of the signal, while resolution refers to the amount of details present in the signal. In order to understand better these concepts, we can note that the scale parameter in discrete wavelet analysis behaves as follows [10]:

- for large scales delated wavelet take a global view of a subsampled signal;
- for small scales, reduced wavelets analyze small details of the signal.

1) Multiresolution analysis: With this method we can derive a lower resolution signal by lowpass filtering with a half-band low-pass filtering having impulse responce $g(n)$. This results in a signal $y(n)$ where

$$y(n) = \sum_{k=-\infty}^{k=+\infty} h(k) * x(2n-k) \tag{3}$$

Now based on subsampled version of $x(n)$ we want to find an approximation, $a(n)$, to the original: this is done by inserting a zero between every sample, because we need a signal at the original scale for comparison. In general, $a(n)$ is not going to be equal to $x(n)$; therefore we compute the difference between $a(n)$ and $x(n)$ with the following formula:

$$d(n) = x(n) - a(n)$$

however there is some redundancy, since a signal sampling ratef_s is mapped into to signals $d(n)$ e $y(n)$ with sampling ratesf_s and $f_{s/2}$, respectively [10].

2) Subband coding schemes: The method of multiresolution analysis described in [10] is characterized by a redundant set of samples. We now look at a different scheme where no such redundancy appears. This method is called subbang coding scheme, first widely used in voice compression. The idea behind this technique is based on the following methodology: a pair of filters are used where low pass and a high pass filter partecipate; we decompose a sequence $X(n)$ into two subsequences at half rate, or half resolution, and this by means of orthogonal filter. This process can be iterated on either or both subsequences. In particular to obtain finer frequency resolution at lower frequencies, we iterate the scheme on the lower band only. Each iteration halves the width of the low band (in fact it increases its frequency resolution by two), but because of the subsampling by two, its time resolution is halved as well. Schematically, this is shown in figure 2.

An important feature of this discret algorithm is its relatively low complexity. Actually, the following somewhat surprising result emerges: regardless the depth of the tree in 2, the complexity is linear in the number of input samples, with a constant factor which depends on the length of the filter [10].

D. Biorthogonality

The wavelet filter implemented in this work is the 3.7 biorthogonal one. A biorthogonal wavelet is a wavelet where the associated wavelet transform is invertible but not necessarily orthogonal. In the biorthogonal case there are two scaling

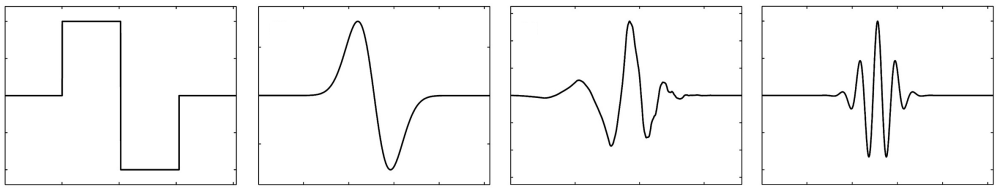


Fig. 1: Example of wavelet functions [14]

functions ϕ and $\tilde{\phi}$, which may generate different multiresolution analysis, according to different wavelet functions ψ and $\tilde{\psi}$, having $\tilde{\psi}$ used in the analysis and ψ used in the syntesis. In addition, the scaling functions ϕ and $\tilde{\phi}$ and wavelet function ψ and $\tilde{\psi}$ are related by duality in the following sense [15]:

$$\int \psi_{j,k}(x)\ \tilde{\psi}_{j',k'}(x)\,dx = 0$$

as soon as $j \neq j'$ or $k \neq k'$ and even

$$\int \phi_{0,k}(x)\ \tilde{\phi}_{0,k'}(x)\,dx = 0$$

as soon as $k \neq k'$ [16]. About biorthogonal wavelet filter, by using two wavelets, one for decomposition and the other for reconstruction, interesting properties are derived. Unlike orthogonal wavelets, biorthogonal ones require two different filters: one for the analysis and other one for syntesis of an input. The first number indicates the order of the syntesis filter while the second number refers to the order of the analysis filter [17]. Unfortunately, frequency responses from biorthogonal filters may now not show any symmetry and the energy of the decomposed signal may also not equal the energy of the original signal [12]. Nevertheless, there are some reasons behind using biorthogonal wavelets related to some requirements to satisfy. One such condition is that they should have compact support: if this constrain were not satisfied, then we would need to use an IIR filters to approximate the function but IIR filters are more difficult to treat and they also require extra computing power.

III. GPU PARALLEL COMPUTING WITH CUDA

The advent of multicore CPUs and manycore GPUs has allowed the development of applications that transparently scale its parallelism to take advantage of the increasing number of processor cores [18]. The launch of the Nvidia CUDA technology has opened a new era for GPGPU computing allowing the design and implementation of parallel GPU oriented algorithms without any knowledge of OpenGL, DirectX or graphics pipeline [19]. CUDA (Compute Unified Device Architecture) is a novel technology of general purpose computing on the GPU, which makes users develop general GPU programs easily. CUDA GPU has the following advantages [18], [19]:

- general programming environment: CUDA uses C programming tools and C compiler, which make programs have better compatibility and portability;
- more powerful parallel computing capability: CUDA graphic card employs more transistors for computing;
- higher bandwidth: it can be strongly affected by the choice of memory in which data is stored, how the data is laid out and the order in which it is accessed;
- instruction operation: CUDA GPU supports integer and bit operation.

According to these considerations, programmable GPUs have evolved into a highly parallel, manycore processor with considerable computational power.

Comparing the performance of GeForce GTX 680, GeForce GTX 580 and GeForce GTX 480 among Sandy Bridge and Bloomfield Intel CUPs around about 2009, we can notice since 2001 the theoretical floating point capability on GPUs has increased more rapidly than in CPUs; furthermore, the memory bandwidth follows a similar trend, mainly looking at years since 2003 [20]. The main difference in floating point capability between the CPU and GPU is due to the specialization of the second one for compute-intensive highly parallel computation. A CUDA-enabled GPU is composed of several MIMD (multiple instruction multiple data) multiprocessors that contain a set of SIMD processors (single instruction single data). Each multiprocessor has a shared memory that can be accessed from each of its processors, and also shares a bigger global memory common the global multiprocessors [21]. Shared memory buffers reside physically on the GPU as opposed to residing in off-chip DRAM, so because of this, the latency to access shared memory tends to be far lower than typical buffers [7].

CUDA hardware architecture has the following novel features [19]:

- general write/read global memory: GPU can acquire data from any location or the global memory and also put data to any location, almost as supple as CPU;

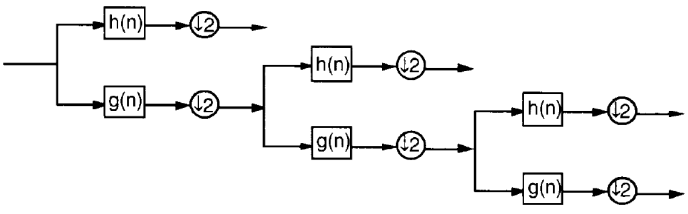


Fig. 2: Filter bank tree of Discret Wavelet Transform implemented with two discret time filters (low-pass and high-pass filter, respectively) and subsampling by two [12]

- shared memory placed on chip: it can make threads in the same multiprocessor getting data immediately available and avoiding accessing global memory frequently;
- thread synchronization: threads in a thread group can be synchronized to each other, so they can communicate and collaborate to resolve complex problems.

In CUDA programming model, an application consists of a *host* program that executes on the CPU and other parallel *kernel* programs executing on the GPU. A kernel program is executed by a set of parallel threads, where a thread is a subdivision of a process in two or more sub-processes, which are executed concurrently by a single processor computer system (multithreading). The host program can dynamically allocate device global memory to the GPU and copy data to/from such a memory from/to the memory on the CPU. Moreover, the host program can dynamically fix the number of threads used for running of a kernel program. A set of threads creates a block, and each block has its own shared memory, which can be accessed only by each thread on the same block [21]. Note that interactions between CPU and GPU should be minimized in order to avoid communication bottlenecks and delays due to data transfers.

IV. Signal processing by wavelet filter

As already widely stated, the main target of this paper is the implementation and performance analysis of an algorithm built for emulating a randomic signal processing through a 3.7 biorthogonal wavelet filter. However, before entering in the description and performance analysis of different algorithms used for both sequential and GPU parallel computing versions, we want to give the theoretical basis about mathematical meaning of a digital FIR filter.

A. Digital filter and Convolution

It is extremely useful to observe that any type of FIR digital filter can be interpreted mathematically as the convolution product between a function f, which represents the signal to scan, and a function h, which is the impulse response of LTI system through which the signal passes, in our case represented by the filter. In the case, such as that of our interest, in which two discrete time functions S and h are given, then it is possible to define the convolution operation as follows:

$$Y(j) = \sum_{i=0}^{N} S(j-i) * h(i) \tag{4}$$

In other words, this operation can also be seen as the application of a function h, which represents the filter, to a function S that instead symbolizes the signal to filter. In particular, the terms $h(i)$ are called *filter coefficients*, while N is the order of this filter.

The convolution of a discrete time signal may be described by a sequence of steps achieved in the following way:

1) function S is reflected: $S(i) \rightarrow S(-i)$;
2) S is scrolled over h with a shift of k: $S(k-i)$;
3) for each shift j all products between S and h are added.

B. 3.7 biorthogonal wavelet filter

Applications of wavelet decomposition techniques in numerical analysis of a signal seems very promising because it highlights the "zooming" property, which allows a very good representation of critical points, such as irregularity, discontinuos, and so on [20]. Wavelet transform can be realized simply and effectively through a filter bank. In particular, a decomposition phase, adopted in this work, provides a low pass and high pass filtering, followed by decimation by two; the reconstruction phase, instead, processes separately inputs, through expansion and filtering, and then they are summed. These operations have to be iterated according to the number of decomposition levels with which we want to achieve wavelet transform [22].

The following table shows the sixteen 3.7 biorthogonal decomposition wavelet filter coefficients taken into account in this work [23].

Filter coefficient	*Value*
h_0	0.0030210861
h_1	-0.0090632583
h_2	-0.0168317654
h_3	0.0746639851
h_4	0.0313329787
h_5	-0.3011591259
h_6	-0.0264992409
h_7	0.9516421219
h_8	0.9516421219
h_9	-0.0264992409
h_{10}	-0.3011591259
h_{11}	0.0313329787
h_{12}	0.0746639851
h_{13}	-0.0168317654
h_{14}	-0.0090632583
h_{15}	0.0030210861

V. Algorithms

Once understood how we can build a digital filter by a mathematical point of view, we see now how to have it in C language. We propose two different versions of code whose target is the translation of convolution operation, previously described, between the signal and the wavelet filter. The first version refers to a programming strategy that uses a parallel CUDA architecture, where the heart of the program, called *kernel*, is defined in *host* memory as a *global* function, and it is executed in GPU memory, or rather in *device*. Later we will analyze the temporal performance of the two programs and compare them, eventually to highlight differences and observe if it might be advantageous using an implementation strategy or another one.

A. Parallel code in CUDA

The first implementation we want to present is related to the construction of a wavelet filter with a parallel programming model. The parallel code is implemented in C language; however we introduce changes to a conventional C code and take advantage of all typical CUDA structures by executing

certain functions in GPU device. Once loaded C libraries and `<cuda.h>` library for device functions, we fix the maximum number of threads that draws each block in device memory. Obviously, this amount of threads is linked to hardware features of GPU cards. As regards the simulations presented in this paper, we use two graphics cards: a 480-core GeForce and a 1536-core Tesla Kepler. The number of threads discends by these specifications, calculating it as the ratio between the total amount of data to process and the number of threads per block.

We are now ready to describe the global main function which, as such, is called in host, and that is after performed in device: this function is exactly the *kernel*.

```
// kernel definition
__global__ void wavelet(float *sign, float *fil, float*y,
     int m) {
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  if (id>=sample) return;
  int P = m*sample;
  for (int i=0; i<filter; i++)
    if ((id-i) > 0) y[P+id]+= sign[id-i]* fil[i];
    else y[P+id]+= sign[i-id]* fil[i];
  int div = (int) powf(2,m);
  sign[id] = sign[id-(id%div)];
  return;
}
```

Listing 1: kernel definition

In the kernel, an instruction which performs the convolution operation between the signal S and h filter is implemented. In fact, if *filter* represents the number of wavelet filter coefficients (in our case $filter = 16$) and *sample* is the amount of randomic signal samples taken into account, then each id-th component of output signal y is obtained as the sum of the products between the i-th component of the filter and the $(id - i)$-th component of signal; id is the thread index with which we exploit computational capability of various thread in GPU memory and whose dimension should cover the whole length of signal. In other words, a wavelet filter can be graphically thought as a floating window, initially aligned with the first sample of signal $(id = 0)$, and comes slid up to cover all samples.

Obviously, when probing the entire signal, there will be situations in which, according to the values assumed by index id and based on i-th coeffiient position of the filter, the difference $id - i$ will be negative; this would require the translation of the signal for "negative" istants, i.e. points where the signal does not exist: this is known as *boundary problem*. Therefore, it is necessary to extend the signal at instants in which it is undefined; this can be done in several ways. The strategy adopted here is the *symmetrical extension*, particularly suitable for biorthogonal filters [22].

For this reason, in wavelet function there is a conditional instruction which chooses the $(id - i)$-th sample of S or $(i-id)$-th one, to avoid "boundary problem". As we can see by code 1, wavelet function takes as input not only the point-to-float variables *sign*, *fil*, *y*, but also another parameter, m, that is the number of current iteration. In fact, as we shall soon see, the purpose of this paper is adopting subband coding technique in a tree-structure, iterating the kernel a number of times equal to *num_iter* in order to isolate step by step residuals and details of signal and get different levels of resolution. Therefore we achieve also the scaling procedure organizing step by step samples in groups of 2^m elements such that they assume the same value in each group. This operation is implemented as

```
int div=(int)powf(2,m);
sign[id]= sign[id-(id%div)];
```

Listing 2: Scaling istructions

Inside the `int main ()`, actual variables corresponding to formal variables are defined and, for each of them, memory allocation is carried out. Also we introduce other three variables *dev_sign*, *dev_fil*, *dev_y* that play a similar task in GPU device

```
// allocation in host memory
  sign_host = (float *) malloc(sample*sizeof(float));
  fil_host = (float *) malloc(filtro*sizeof(float));
  y_host = (float *) malloc(sample*sizeof(float));

// allocation in device memory
  cudaMalloc((void**)&dev_sign, sample*sizeof(float));
  cudaMalloc((void**)&dev_fil, filtro*sizeof(float));
  cudaMalloc((void**)&dev_y, sample*sizeof(float));
```

Listing 3: Malloc and CudaMalloc

Later, after loading the samples of the signal and filter coefficients, respectively on the arrays *sign_host* and *fil_host*, we proceed with data transfer on device using *cudaMemcpy* commands. In fact, they constitute a bottleneck in the performance of a code and therefore should be used as sparingly as possible. For this reason we decide to accumulate the data of num_iter iterations in a single output array y whose size is equal to $num\_iter * sample$. After we carry out another cudaMemcpy to return output data from host to device

```
//Data transfer from host to device
cudaMemcpy(dev_sign, sign_host, sample*sizeof(float),
     cudaMemcpyHostToDevice);
cudaMemcpy(dev_fil, fil_host, filtro*sizeof(float),
     cudaMemcpyHostToDevice);
```

Listing 4: Cudamemcpy istructions

Finally, we save output data for each of single iterations and we can free host and device allocate memory through `free(...)` and `Cudafree(...)` commands. Function `clock(...)`, called from C library `<time.h>`, is useful to measure the time of this and the other program described in the following section.

B. Sequential program

Another version of code that performs the same task refers to a sequential programming model. By this we mean the transposition of signal filtering in a conventional program written in C language. In this case the first thing to do is defining the function which achieves convolution product and corresponds to the parallel code kernel, as shown below. As we can observe, such a function is similar to the kernel described above. Here, thread index disappears and is replaced by an index with the same name but referring to a for loop whose element $y(id)$ is computed in a sequential way.

```
void wavelet(float *sign, float *fil, float*y, int m) {
  for (int id=0; id<sample; id++) {
    int P = m*sample;
    for (int i=0; i<filtro; i++)
      if ((id-i) > 0) y[P+id]+= sign[id-i]* fil[i];
      else y[P+id]+= sign[i-id]* fil[i];
    int div = (int) pow(2,m);
    sign[id] = sign[id-(id%div)];
  }
  return;
}
```

Listing 5: sequential function

Clearly all typical CUDA instructions have disappeared; nevertheless, in order to compare fairly this code with parallel one, we need to replace them with similar instructions that emulate the same behavior. This explains why instead of $cudaMalloc$ we find three additional variables*d_sign_host, d_fil_host, and d_y_host* allocated in host memory which substitute arrays *dev_sign, dev_fil and dev_y*. Then, after execution finishes, cudaMemcpy are replaced by assignments listed below, unnecessary for the results, but useful to take into account the evaluation of time (and performance)

```
d_sign_host = sign_host;
d_fil_host = fil_host;
d_y_host = y_host;
```

Finally, in this model we use `clock( )` function in order to measure execution time of the program. The goal now is making benchmark of two codes and comparing results.

VI. Experimental results

Now we analyse the performance for sequential and parallel codes. In order to pursue this target, we change the number of samples that makes up our examined signal and we evaluate the time trend which results increasing gradually the number of samples. The simulations of both codes will be conducted on two different technical specification computers. In fact the first machine is used as a normal PC for student laboratory applications; the other one is dedicated to specific applications as a server, which has got high reliability, increased performance and additional features. For this reason, in order to distinguish them, we denote the first computer as *PC*, the second one as *Server*.

A. Technical specifications

We describe the main features of two machines and their respective CPU and graphics card.

1) PC: It has got an AMD Athlon 64X2 with frequency CPU clock up to 2.9 GHz and NVIDIA GeForce GTX 480 GPU. This GPU shows the following engine and memory specifications [24]: it has a number of CUDA cores egual to 480, with a graphic clock of about 700MHz and a fill rate texture of 42 billion per second. The interfaces are designed for PCI Express 2.0x16 accelerator, with a peak bandwith of up to 2 GByte per second and with a 1536 MByte GDDR5 standard memory, where memory locations are connected through 384 bit memory interface.

2) Server: In server we find a Intel Xeon CPU having a clock frequency up to 3.4 GHz. About GPU, here is a NVIDIA Tesla Kepler K10. This card offers a total of 8 GByte GDDR5 on-board memory, of which 4 GByte for GPU, and supports PCI Express Gen 3 bus interface.

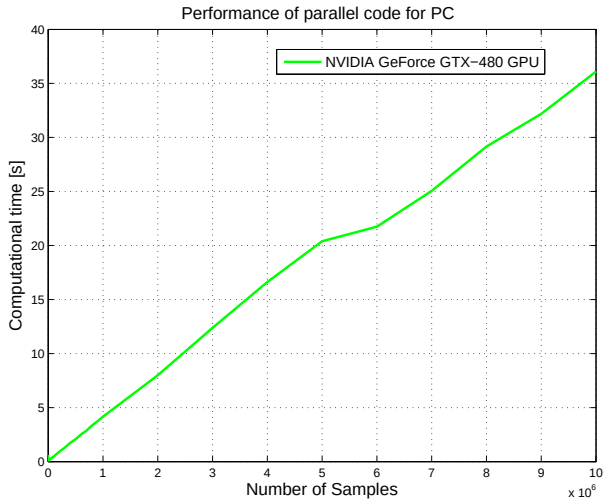
B. Simulations

Since technical features of PC GPU are very different than Server one, it is clear that for the parallel code we have to adjust the number of threads per block, according to the type of used GPU: if we work with GeForce, this parameter will have to be equal to 480; if the code is executed on server with Tesla Kepler K10 card, then we should put it equal to 1536. The simulation of the parallel code on PC offers the result represented graphically in Figure 3a.

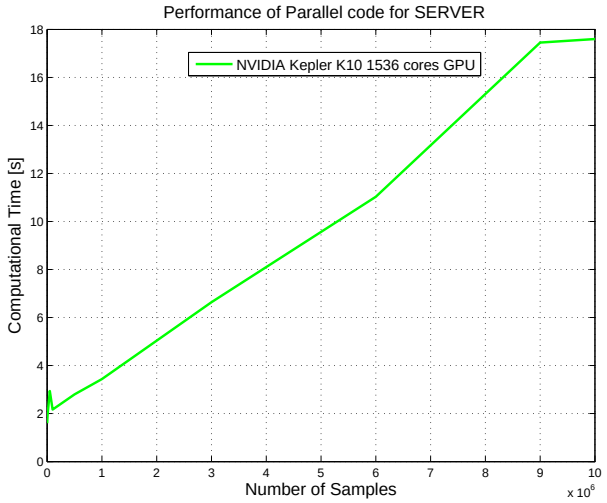
Figure 3a shows an almost linear trend throughout the range from 1000 to 10 milion of samples, with a computation time varying from 150 to 300 ms for the first 10000 samples. Start from 5 milion of samples, the computing time increases of about 20 seconds; in the range of equal amplitude to previous one, from 5 to 10 milion of samples, the computing time increment is about 15 seconds compared to the theoretical expected $20s$ that we should obtain if the curve had not undergone the level-shift down. A similar trend is given in Figure 3b for the simulations of parallel code on the server. In this case, for 10 milion of samples computing time is initially larger of about one order of magnitude, growing from there linearly up to 9 milion of samples, where the execution time of the program is $17.4s$. Subsequently, from 9 to 10 milion of samples, the computing time increases of only 200 ms. A comparison between PC and server runs for parallel and sequential version is shown in 4a and 4b:

We can see that, for PC for a number of samples less than 15000, sequential code timing is below the curve relative to the parallel code; in the range between 20000 and 70000 samples the two curves intersect several times, showing an almost swinging behavior; instead from 70000 samples the CPU line is above GPU one and increasing the number of samples their distance is larger and larger, and at the 10 million samples it reaches 25 seconds. Server simulations show, instead, parallel code starts having performance better than sequential one only when they have reached 3 milion of samples, a value much higher than the 70000 seen for the server. We should note in this case the gap between two curves is very small: in fact in the range up to 3 milion samples the sequential is faster than parallel of about $2s$; once crossed the point where the two codes have similar performance, the GPU starts to work better, gaining a time advantage of about $4.5s$ in the neighborhood of 10 million samples.

Figure 5 gives a global overview of the CPUs and GPUs behavior in PC and Server. As clearly shown in this graph, reaching an amount of samples high enough (almost 3 million), the best performance in terms of saved time for program execution is when we implement and execute the parallel code using server GPU: this result was quite predictable because with this card we use 1536 cores compared to 480 of GeForce

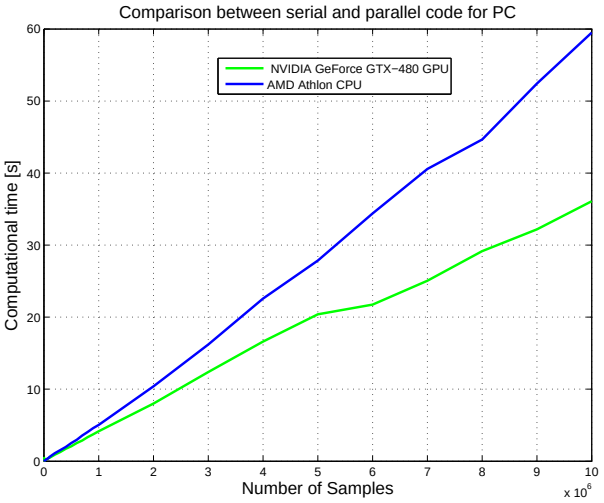


(a) Computing time of parallel code in PC for varying amount of samples

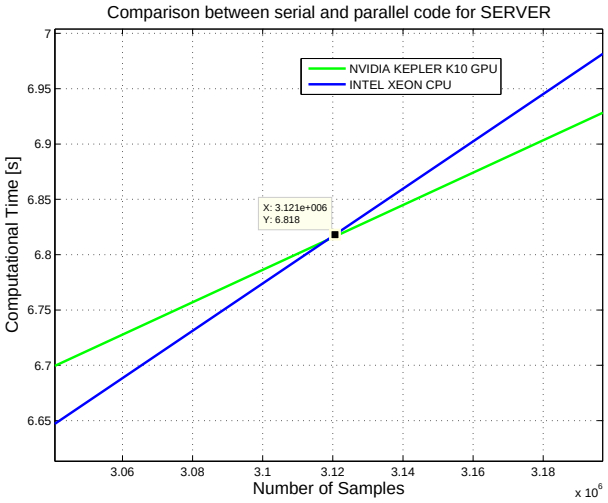


(b) Computing time of parallel code in Server for varying amount of samples

Fig. 3: Computing time of parallel code in PC and Server



(a) Parallel and sequential program in PC



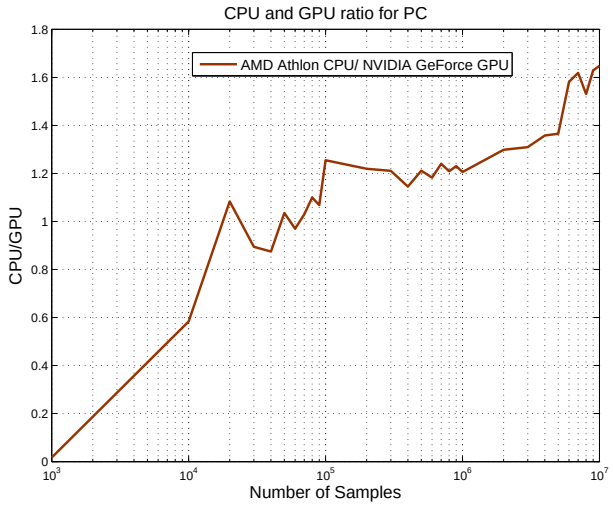
(b) Parallel and sequential program in Server

Fig. 4: Comparison between GPU and CPU performance in PC and Server

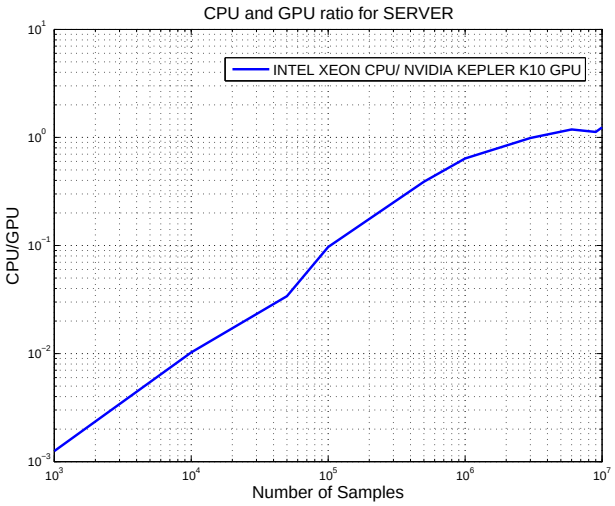
GPU. Nevertheless, by a careful observation, though GPU server performance are better than GeForce one, being able to earn between 10 and $25s$ ahead of the second one, it is equally true that there aren't considerable differences from what it happens by exploiting the server CPU potential than Kepler GPU: in this case the separation between the Intel Xeon CPU curve and Tesla Kepler relating one is only a few seconds. This result is due to the server, because of its features and applications, has got hardware equipment more advanced than a normal PC; in fact Xeon CPU, with its 6 cores per socket and a clock frequency up to 3 GHz, can provide very satisfactory results which are not very different than ones offered by Tesla Kepler GPU. On the other hand the difference between the use of a parallel code and a sequential one is more pronounced when we look at a PC with an average CPU and we compare it with GeForce GTX 480 GPU, which belongs to a high-end market and with its 1536 Mbytes of installed RAM, its GDDR5 video memory and high operating frequencies, it is certainly one of the most important products made by NVIDIA corporation [24].

According to what argued so far, some significant information can be derived by plotting all the different ratio combinations between GPU and CPU timing for PC and server. From Figure 6a we observe clearly that for the first 20000 samples the sequential code on the CPU is faster than GPU program; the situation is reversed past this value of samples, from which parallel code becomes approximately 1.5 times faster than sequential one.

As shown in figure 6b for the server, we have a similar situation. However, here the GPU shows its advantages



(a) Ratio between AMD Athlon CPU and NVIDIA GeForce GPU times in PC



(b) Ratio between Intel Xeon CPU and NVIDIA Tesla K10 GPU times in Server

Fig. 6: Ratio CPU/GPU for PC and Server

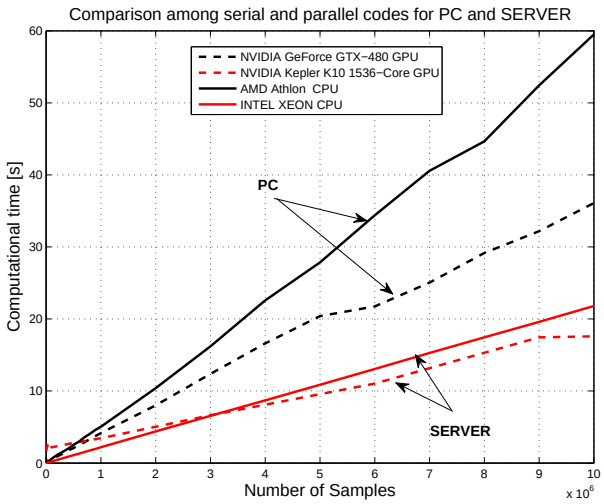


Fig. 5: Comparison among parallel and sequential programs on PC and Server

after passing 3 milion of samples. Moreover, in this graph the performance of parallel code are better than about 1.5 times, compared with sequential one in correspondence of the maximum limit of our observation range. The situation is a bit different when we make mixed ratios among GPU performance of a machine and CPU performance of the other one.

If we take into account figure 7a, we see a downward trend, where in correspondance of an amount of samples equal to 1000, the times of server GPU are greater than PC CPU, even over two orders of magnitude. The performance of examined GPU and CPU are equal when we reach 500000 samples. Going forward, this ratio becomes thiner and thiner, until it settles around 0.3 in the range from 9 to 10 million samples.

A monotonically similar trend is shown in the graph 7b. However, for reason seen previously, the ratio between GPU time of PC and CPU time of server is always maintained higher than one, demonstrating GeForce performance are always inferior if we compare them with the Intel Xeon CPU. Finally, if we compare to each other the execution times of two GPUs, we observe that initially the curve remains below the unit in the range from 1000 to about 800000 samples, indicating PC GPU works better than Server one, with values 10 times lower. Instead, in the remaining part of observation range, GeForce execution time is about twice than Tesla Kepler K10 (see figure 8).

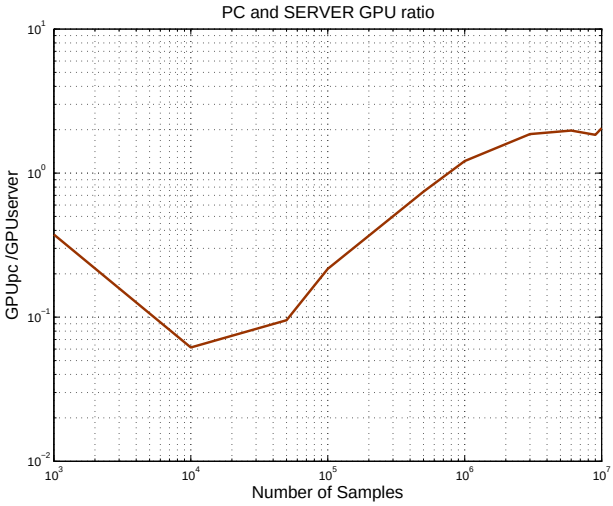
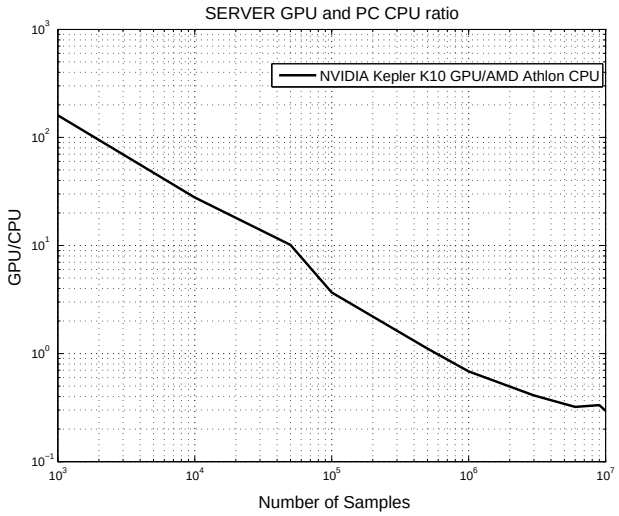


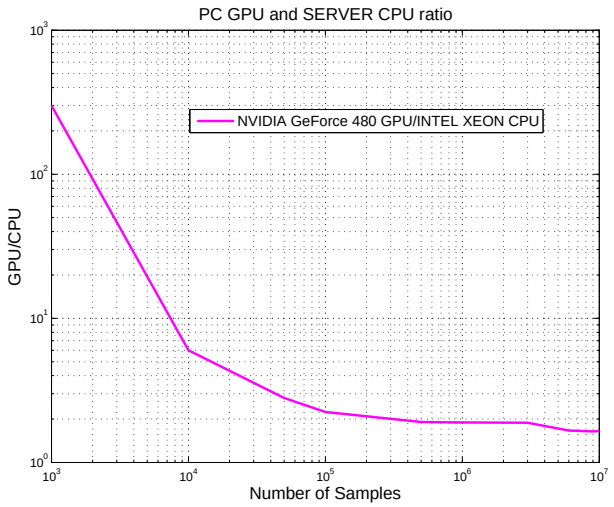
Fig. 8: Ratio between PC and Server GPU

VII. Related Works

Now we remark some works closer to this one, that have been carried out in parallel computing context, in particular



(a) Mixed Ratio between NVIDIA Tesla K10 GPU and AMD Athlon CPU times



(b) Mixed Ratio between NVIDIA GeForce GPU and Intel Xeon CPU times

Fig. 7: Mixed Ratios among GPUs and CPUs

reflecting about GPU card potentials applied to wavelet filters. In [25] the authors deepen the concept based on an algorithm centered on GPU cards to reconstruct 3D wavelet using programs fragments to minimize the transfer of data and processing overhead fragment; for this pourpose they have proposed a novel scheme that uses tile boards as a primary layout to organize 3D wavelet coefficients.

In article [26], the authors present an algorithm that performs SIMD complete discrete wavelet transform (DWT) convolution on a GPU, which brings significant improvement performance on a normal PC, without additional costs. Although the forward and inverse wavelet transforms are mathematically different, the algorithm they proposed unifies them to an almost identical process that can be efficiently implemented on GPU. DWT has a wide range of applications, by signal processing in video and image compression. For this reason, in [27] the authors have shown that this transformation, by means of the lifting system, can be performed in memory mode and efficient computation on GPU, through CUDA computing paradigm of Nvidia. In particular their design exploits three major hardware architectures for 2D DWT (row-column, line-based, block-based) and describes a hybrid method between the row and column-based methods on blocks.

Another work, in [28], presents an improved version of an algorithm suitable to compute the 2D DWT on GPU. In their work the authors have adapted this method for an existing algorithm computing the vertical transform pass. By taking advantage of the computational power of a GPU when implementing a wavelet transform, they demonstrate the time of the computation can be substantially reduced.

Work [29], proposes a method which analyzes the behavior of several parallel algorithms developed to compute the two dimensional DWT, in particular using both openMP over a multicore platform and CUDA over a GPU. They also have found that significant execution time improvements are achieved on both multicore platforms and GPUs. Likewise in the paper [30], the authors explore the potential of OpenCL in accelerating the DWT computation and analyze the programmability, portability and performance aspects of this language; their experimental analysis is done using NVIDIA and AMD drivers that support OpenCL.

The authors in paper [31] present an overview of wavelet based multiresolution analysis, discussing the continuos wavelet transform in its simplest form and looking at orthogonal, biorthogonal and semiorthogonal wavelets. Many authors have applied the strategy of parallel computing using wavelets to image encoding field. In the paper [32] the computation steps in JPEG2000 are examined, particularly in the Tier-1, and novel, GPGPU compatible, parallel processing methods through wavelets for the sample-level coding of the images are developed. In work [33] closer to the previous one, the authors focus their attention on accelerating JPEG2000 encoder by using GPGPU.

Finally, [34] presents a GPGPU framework with the corresponding parallel computing solution for wavelet based image denoising by using off-the-shelf consumer-grade programmable GPUs. Their experiment results show that framework gain applicability in data parallelism and satisfaction performances in accelerating computations for wavelet-based denoising.

VIII. Conclusions

In this work we have analyzed the computational capacity of a parallel code written in CUDA environment which implements the filter process of a randomic signal using a biorthogonal wavelet filter, and we have compared it with a sequential code that performs the same task; we have executed both codes on two computers having different technical specifications.

Simulations make it clear that, except the case in which the number of samples is very low, the adoption of a parallel programming technique is certainly advantageous in terms of execution time. In particular, over a number of 3 million of samples, the GPU Server performance predominates on CPU of the same computer and on both GPU and CPU related to PC, confirming the strong potential of parallel computing in terms of efficiency, computational capacity and execution time.

The potential for future GPU perfomance increases presents great opportunities for demanding applications, including computational graphics, computer vision, and a wide range of high-performance computing applications [35].

ACKNOWLEDGMENT

The authors would like to thank *Giacomo Capizzi* and *Christian Napoli* to lead them in the Parallel GPU computing with CUDA field and for the support in drafting and reviewing this paper.

REFERENCES

[1] G. Pappalardo and E. Tramontana, "Automatically discovering design patterns and assessing concern separations for applications," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, Dijon, France, April 2006, pp. 1591–1596.

[2] C. Napoli, G. Pappalardo, E. Tramontana, and G. Zappala, "A Cloud-Distributed GPU Architecture for Pattern Identification in Segmented Detectors Big-Data Surveys," *The Computer Journal*, 2014.

[3] F. Bonanno, G. Capizzi, G. L. Sciuto, C. Napoli, G. Pappalardo, and E. Tramontana, "A novel cloud-distributed toolbox for optimal energy dispatch management from renewables in igss by using wrnn predictors and gpu parallel solutions," in *Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM), 2014 International Symposium on*. IEEE, 2014, pp. 1077–1084.

[4] F. Bonanno, G. Capizzi, G. Lo Sciuto, C. Napoli, G. Pappalardo, and E. Tramontana, "A cascade neural network architecture investigating surface plasmon polaritons propagation for thin metals in openmp," in *Proceedings of International Conference on Artificial Intelligence and Soft Computing (ICAISC)*, ser. Springer LNCS, vol. 8467, Zakopane, Poland, June 2014, pp. 22–33.

[5] C. Napoli, G. Pappalardo, E. Tramontana, R. Nowicki, J. Starczewski, and M. Woźniak, "Toward work groups classification based on probabilistic neural network approach," in *Proceedings of International Conference on Artificial Intelligence and Soft Computing (ICAISC)*, ser. Springer LNCS, Zakopane, Poland, June 2015, vol. 9119, pp. 79–89.

[6] M. Woźniak, D. Połap, M. Gabryel, R. Nowicki, C. Napoli, and E. Tramontana, "Can we process 2d images using artificial bee colony?" in *Proceedings of International Conference on Artificial Intelligence and Soft Computing (ICAISC)*, ser. Springer LNCS, Zakopane, Poland, June 2015, vol. 9119, pp. 660–671.

[7] J. Sanders and E. Kandrot, *CUDA BY EXAMPLE, An Introduction to General-Purpose GPU Programming, NVIDIA*.

[8] *Introduction to parallel programming*. [Online]. Available: courses.cs.washington.edu

[9] E. Tramontana, "Managing Evolution Using Cooperative Designs and a Reflective Architecture," in *Reflection and Software Engineering*, ser. Springer LNCS, 2000, vol. 1826.

[10] O. Rioul and M. Vetterli, "Wavelets and Signal Processing," *IEEE SP magazine*, October 1991.

[11] C. Napoli, G. Pappalardo, and E. Tramontana, "An agent-driven semantical identifier using radial basis neural networks and reinforcement learning," in *XV Workshop "From Objects to Agents" (WOA)*, vol. 1260. Catania, Italy: CEUR-WS, September 2014.

[12] J. Akhtar, *Optimization of biorthogonal wavelet filters for signal and image compression*, February 2001.

[13] C. Napoli, G. Pappalardo, and E. Tramontana, "A Hybrid Neuro-Wavelet Predictor for QoS Control and Stability," in *Proceedings of Advances in Artificial Intelligence (AI*IA)*, ser. Springer LNCS, vol. 9119, Torino, Italy, December 2013, pp. 527–538.

[14] [Online]. Available: http://www.bssaonline.org

[15] M. A. Salem, N. Ghamry, and B. Meffert, "Daubechies versus biorthogonal wavelets for moving object detection in traffic monitoring systems," 2009.

[16] S. Mallat, *A Wavelet Tour of Signal Processing*, October 2008.

[17] M. P. Chaudhary and G. Lalit, "Lifting Schemem Using HAAR and Biorthogonal Wavelets For Image Compression," *Internetional Journal of Engineering Reseach and Applications*, 2003.

[18] "Cuda c programming guide version 4.2," pp. 1–5.

[19] Z. Yang, Y. Zhu, and Y. Pu, "Parallel Image processing Based on CUDA," in *Proceedings of International Conference on Computer Science and Software Engeneering*, 2008.

[20] *NVIDIA CUDA C Programming Guide*, April 2012.

[21] A. Fornaia, C. Napoli, G. Pappalardo, and E. Tramontana, "An AO system for OO-GPU programming," in *Proceedings of the 16th Workshop "From Object to Agents" (WOA15)*, Naples, Italy, June 2015.

[22] L. Verdoliva, "La trasformata wavelet," 2009-2010.

[23] [Online]. Available: http://wavelets.pybytes.com/wavelet/bior3.7

[24] [Online]. Available: http://www.nvidia.it/object/geforce-family-it.html

[25] A. Garcia and H.-W. Shen, "Gpu-Based 3D Wavelet Recostruction With Tileboarding," *The Visual Computer*, vol. 21, no. 8-10, pp. 755–763, 2005.

[26] T.-T. Wong, C.-S. Leung, P.-A. Heng, and J. Wang, "Discrete Wavelet Transform On Consumer-Level Graphics Hardware," *IEEE Transactions on Multimedia*, vol. 9, no. 3, pp. 668–673, 2007.

[27] W. Van Derlaan, A. Jalba, and J. Roerdink, "Accelereting Wavelet Lifting On Graphics Hardware Using Cuda," *IEEE Transactions on Parallel and Distribuited Systems*, vol. 22, August 2010.

[28] H. Moen, "Wavelet transforms and efficient implementation on the GPU," May 2007.

[29] V. Galiano, O. Lopez, M. Malumbres, and H. Migallun, "Parallel strategies for 2D Discrete Wavelet Transform in shared memory systems and GPUs," *Springer Science*, 2012.

[30] B. Sharma and N. Vydyanathan, "Parallel discrete wavelet transform using the Open Computing Language: a performance and portability study," in *Proceedings of IEEE International Symposium on Parallel and Distribuited Processing*, April 2010.

[31] B. Jawerth and W. Sweldens, "An Overview Of Wavelet Based Multiresolution Analyses," *SIAM review*, vol. 36, no. 3, pp. 377–412, 1994.

[32] R. Le, I. Bahar, and J. Mundy, "A Novel Parallel Tier-1 Coder For Jpeg2000 Using Gpus," *Application Specific Processors , IEEE*, june 2011.

[33] M. Ahmadvand and A. Ezhdehakosh, "Gpu-based implementation of jpeg2000 encoder," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2012.

[34] Y. Su and Z. Xu, "Parallel implementation of wavelet-based image denoising on programmable pc-grade graphics hardware," *Signal Processing*, vol. 90, no. 8, pp. 2396–2411, 2010.

[35] S. W. Keckler, W. J.Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Computer Society*, September/October 2011.