HAL
open science

Analysis of Multicore CPU and GPU toward Parallelization of Total Focusing Method ultrasound reconstruction

Jason Lambert, Antoine Pédron, Guillaume Gens, Franck Bimbard, Lionel Lacassagne, Ekaterina Iakovleva

To cite this version:

Jason Lambert, Antoine Pédron, Guillaume Gens, Franck Bimbard, Lionel Lacassagne, et al.. Analysis of Multicore CPU and GPU toward Parallelization of Total Focusing Method ultrasound reconstruction. DASIP 2012 - Conference on Design and Architectures for Signal and Image Processing, Electronic Chips & Systems design Initiative, Oct 2012, Karlsruhe, Germany. hal-01093274

HAL Id: hal-01093274

https://inria.hal.science/hal-01093274v1

Submitted on 12 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ANALYSIS OF MULTICORE CPU AND GPU TOWARD PARALLELIZATION OF TOTAL FOCUSING METHOD ULTRASOUND RECONSTRUCTION

Jason Lambert[1], *Antoine Pédron*[1], *Guillaume Gens*[2], *Franck Bimbard*[2], *Lionel Lacassagne*[2] *and Ekaterina Iakovleva*[1]

[1] CEA, LIST, F-91191 Gif-sur-Yvette, France
[2] Institut d'Electronique Fondamentale, UMR 8622, Université Paris-Sud 11, F-91405 Orsay, France

ABSTRACT

Ultrasonic imaging and reconstruction tools are commonly used to detect, identify and measure defects in different mechanical parts. Due to the complexity of the underlying physics, and due to the evergrowing quantity of acquired data, computation time is becoming a limitation to the optimal inspection of a mechanical part. This article presents the performances of several implementations of a computational heavy algorithm, named Total Focusing Method, on both Graphics Processing Units (GPU) and General Purpose Processors (GPP). The scope of this study is narrowed to planar parts tested in immersion for defects.

Using algorithmic simplifications and architectural optimizations, the algorithm has been drastically accelerated resulting in memory-bound implementations. On GPU, high performances can be achieved by profiting from GPU long memory transactions and from hand managed memory. Whereas on GPP, computations cost are overrun by memory access resulting in less efficient performances compared to the computing capabilities available.

The following study constitutes the first step toward analyzing the target algorithm for diverse hardware in the non-destructive testing environment.

Index Terms— non-destructive testing, ultrasonic reconstruction, parallelization, general purpose processors, graphic processing units, total focusing method

1. INTRODUCTION

Non Destructive Testing (NDT) regroups numerous techniques that are designed to evaluate the properties of a mechanical part without causing damage. Multiple industries are concerned, such as: aeronautics, petrochemical, transports, energy, etc. Because security is always more important, the analysis needs are growing. The control tools have evolved drastically, so as the data storage allowing the acquisition of large volumes of data. Linear ultrasonic transducers are one of the evolutions of the ultrasound transducers and allow the NDT expert to use the Full Matrix Capture (FMC) to get the maximum information to be post-processed. Reconstructions from the FMC acquisition are usually very expensive and consequently avoided in real situations because of the computation time.

Until a decade ago, the hardware evolution - on clock speed, execution optimization and caches - was enough to compensate the volume growth, but the frequency wall hit by hardware manufacturers has set a new deal. Up to now, parallel chips have grown, reaching nowadays a point of maturity. General Purpose computations on GPU (GPGPU) is obviously the domain that is in spotlight, with numerous articles in the literature showing great accelerations with GPU compared to sequential GPP implementations. But multicore GPP have also evolved, increasing the SIMD possibilities and enhancing cache memory, more particularly because of the need to link memory for multicore systems.

The CIVA software platform has been developing the Total Focusing Method (TFM) within its ultrasonic reconstruction module [1]. This approach has been firstly introduced in ultrasonic NDT field with the well-known SAFT algorithm [2] [3]. This article presents a study of the parallelization of TFM on GPU and multicore SIMD GPP. Parallelization of such method has already been realized on GPU by Romero *et al.* [4]. However, this work focuses on immersion transducers, implying more complex computations and significantly longer execution time than the contact transducer version of the algorithm, as described by Stepinski *et al.* [5].

This paper is organized as follows. The TFM algorithm is presented in section 2 with the computation specificities added by immersion transducers. Then, GPP and GPU implementations are detailed in section 3 and 4 respectively. Benchmarks performed on GPP and GPU architectures using datasets from real control cases are analyzed in section 5. Finally, the last section concludes the paper and discusses future works.

2. TOTAL FOCUSING METHOD

This section deals with the presentation of the Total Focusing Method for contact and immersion transducers. Subsec-

tion 2.1 presents the core process and subsection 2.2 the computation differences between contact and immersion transducers. Then, the algorithm is shown in subsection 2.3.

2.1. Method principle

Consider a linear ultrasonic array of N_t transceivers in which each transceiver is successively used as the transmitter, while all other transceivers are used as receivers. The data are organized in a symmetric matrix that contains all the acquired $N_t \times N_t$ signals. This acquisition process is called Full Matrix Capture (FMC).

The TFM is a technique used to post-process the data from FMC to produce a scalar image, $I(P)$, of the inspected region, where the array is focused in transmission and reception at every point P in the image. The intensity of the TFM image $I(P)$ is given by:

$$I(P) = \sum_{i,j=1}^{N_t} S_{ij}(T_{ip} + T_{jp}) \tag{1}$$

where S_{ij} is the acquired temporal signal, T_{ip} and T_{jp} are the time delays relative to the point P for i'th and j'th transceivers, respectively. Thus, the TFM algorithm can be summarized as follows:

1. Discretization of the inspected region into a grid.
2. Calculation of $I(P)$ given by equation 1 for each grid point P.

2.2. Delay law computation

This subsection describes the computation of the time delay $T_{ip}, i = 1 \ldots N$, for contact and immersion setups used in NDT for ultrasonic inspection of solid components.

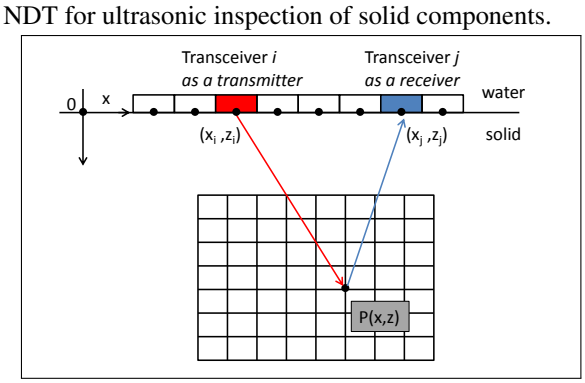

Fig. 1. 2D-configuration representing TFM in contact mode

In contact mode, the ultrasonic transducer is directly coupled to the surface of the mechanical part using a thin layer of contact agent. This inspection setup is schematized in figure 1. Following to [6], the propagation time T_{ip} is determined by dividing the geometric distance from i'th transceiver to the imaging point P by the velocity of sound c:

$$T_{ip} = \frac{\sqrt{(x_i - x)^2 + (z_i - z)^2}}{c} \tag{2}$$

where (x_i, z_i) is the position of the i'th transceiver, x and z are the coordinates of the imaging point P.

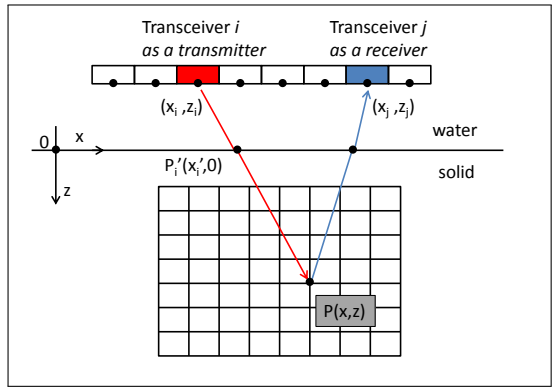

Fig. 2. 2D-configuration representing TFM in immersion mode

In immersion mode, the transducer is separated from the mechanical part by a coupling liquid medium, which is generally water. For this inspection mode, propagation of ultrasonic waves in two media with different sound velocities results in refraction at the interface. As is shown in figure 2, ultrasound beam is refracted at the point $P_i'(x_i', 0)$ at the planar liquid/solid interface given by $z = 0$. Analogously to the previous case, the propagation time T_{ip} can be expressed as follows:

$$T_{ip} = \frac{\sqrt{(x_i - x_i')^2 + z_i^2}}{c_1} + \frac{\sqrt{(x_i' - x)^2 + z^2}}{c_2} \tag{3}$$

where c_1 and c_2 are the velocity of ultrasonic waves in liquid and solid, respectively, (xi, zi) is the position of the i'th transceiver, $(x_i', 0)$ is the positions of the refraction point P' and (x, z) are the coordinates of the point P.

The x-coordinate of the refraction point P' can be determined using Snell's law which leads, as detailed in [7], to the following nonlinear equation:

$$\frac{x_i - x_i'}{c_1\sqrt{(x_i - x_i')^2 + z_i^2}} = \frac{x_i' - x}{c_2\sqrt{(x_i' - x)^2 + z^2}} \tag{4}$$

The solution of the equation 4 can be approximated by using some iterative methods in order to find the roots of the associated polynomial of fourth degree.

2.3. TFM algorithm

This subsection describes the TFM algorithm accordingly to the principle exposed in subsection 2.1. Supposing that the algorithm produces the scalar image corresponding to a rectangular region beginning at (x_b, z_b) (top left corner) and ending at (x_e, z_e) (bottom right corner) with a linear ultrasonic array of N_t transceivers. For a scalar image with a resolution of $K \times L$, the previous rectangular region is discretized in $K \times L$ points $P_{i,j}(x_b + i.\Delta_x, z_b + j.\Delta_z)$ where $i \in [0, K[$, $j \in [0, L[$, $\Delta_x = (x_e - x_b)/(K-1)$ and $\Delta_y = (z_e - z_b)/(L-1)$.

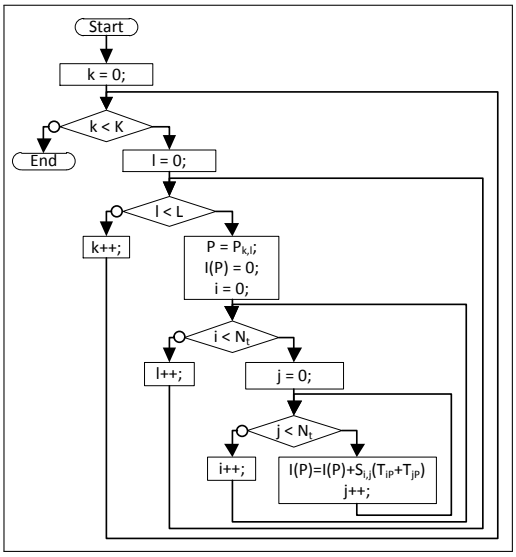

Fig. 3. *TFM: Classical algorithm*

Thus, figure 3 presents the classical algorithm, where for a given point P, each time delay T_{iP} $(i \in [0, N_t[)$ is computed $N_t + 1$ times: once as transmitter and N_t times as receiver.

For a given point $P_{i,j}$ and a given transceiver, the time delay between them depends only on their positions which do not change during the computation. Thus, this time delay is the same whether the transceiver acts as a transmitter or as a receiver. That is why, it is possible to say that there is a symmetry regarding to the role of each transceiver. Then, the previous algorithm can be optimized by computing each time delay only once. As exposed in figure 4, for a given point P, the N_t time delays are all computed once and then are used to produce the pixel $I(P)$.

2.4. Numerical considerations

Iterative root finding methods refine over an initial guess or approximation of one root of the studied polynomial. The selected root finding method is Laguerre's because of its capacity to always converge for any initial guess and with a cubical rate of convergence for simple roots. For each root found, polynomial deflation is applied using Horner's Method, dividing $P(X)$ by $(X - x_{root})$ to reduce the polynomial degree.

Implementations of these methods imply an important number of floating point operations. In preparation for future implementation, computations accuracy and numerical stability of single-precision floating-point implementations of those algorithms have been validated using both the Mathematica software and the GNU MPFR library.

3. GPP IMPLEMENTATIONS

In this section is explained how the TFM algorithm (see subsection 2.3) has been implemented on a GPP. First of all, the subsections 3.1 and 3.2 respectively explains the principle of SIMD instructions and the principles of the OpenMP library. Then, in the subsection 3.3, the use of these two technologies is exposed in order to accelerate the TFM algorithm.

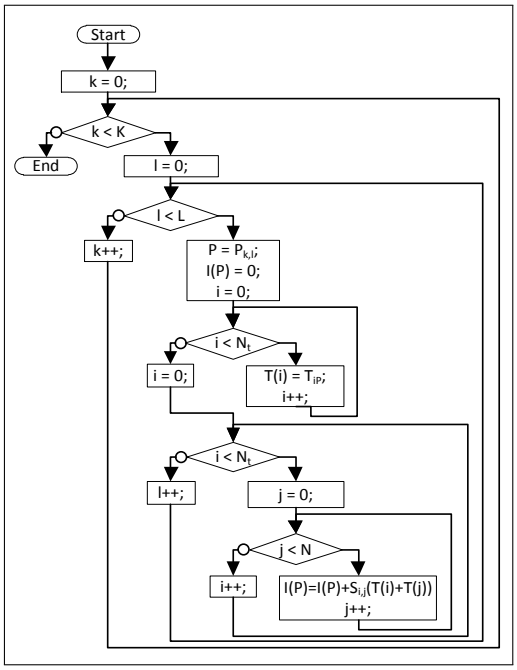

Fig. 4. *TFM: Algorithm using symmetries*

3.1. Principle of SIMD instructions

SIMD instructions currently work on 128-bit and 256-bit vectors. Because of the length of the vectors, each SIMD instruction is able to perform a given operation simultaneously on multiple data [8] [9]. For example, if using 128-bit vectors, a SIMD instruction can perform simultaneously four additions on four single-precision floating point numbers (32-bit).

Thus, SIMD instructions are useful for optimizing algorithms but can only be used for regular computations. Among other things, these instructions do not support test jumps but if-then-else structure can be implemented through the use of comparisons and selection (i.e. masking).

3.2. Principles of the OpenMP library

The OpenMP library is designed for shared-memory parallel programming.This library is useful to easily parallelize existing codes without doing heavy modifications by creating as many threads as the number of GPP logical cores. Each thread has its own stack in which its local variables are located. Thus, any local variable is private and can be read and/or written only by the thread to which it belongs. But the OpenMP library also allows developers to share some variables among the threads. They are located in a shared memory and can be read and written by all threads.

3.3. TFM implementation with using SIMD instructions and OpenMP library

In the particular case of the TFM algorithm, the OpenMP library is used to parallelize the external loop. According to this algorithm, the FMC containing all the acquired signals, the output image and its position, the number N_t of transceivers in the linear ultrasonic array must be shared among all threads.

Other variables are private and can be read only by the thread to which they are belonging. Then, the OpenMP library automatically breaks the loop and creates threads to execute its code among all the cores in the GPP.

Concerning SIMD instructions, 128-bit vectors are actually used (SSE2). Each vectors contains four simple-precision reals. Thus, four time delays are computed simultaneously following the equations described in the subsection 2.2.

4. ADAPTATION TO THE GPU

This section deals with the specificities of the GPU architecture and the CUDA model. Subsection 4.1 concisely explores the CUDA model, then in subsection 4.2 the implementations of the TFM developed to benefits from a GPU computational power are detailed.

4.1. GPU architecture considerations

NVIDIA released a new programming architecture in 2006 to ease GPGPU on its GPU. This architecture comes with both a programming paradigm and a C-like interface to program GPU. A CUDA task is called "kernel" and it is represented by a C-like function which will be executed once by all threads.

With CUDA, GPU are used as coprocessors to their host systems. They dispose of their own memory, off-chip DRAM, called *global memory*, which is slow to access memory (comparing to GPU frequency), but it is kept alive for the duration of the application allowing communication between kernels without back-and-forth data movement to the host. Furthermore, on recent GPU (Fermi architecture), access to global memory can be made through L1 and L2 caches of 16 to 48KB and 768KB respectively.

GPU consist, too, of severals *streaming multiprocessors* (SM, 16 on current high end cards). Each SM is composed of multiple elements, whose numbers depend on hardware generation, for example :

- *CUDA cores* for integer and floating-point arithmetic operations;
- *Special Function Units* for floating-point transcendentals functions;
- one or two *schedulers* to manage threads.

Beside these computing elements, each SM disposes of :

- a rather small (16KB, up to 48KB on recent devices) but fast *shared memory* on the SM, for exchanges between its cores;
- a limited, per SM, set of *registers* divided between the threads of a kernel at compile-time. If too many registers are required by a kernel, the compiler decides which registers to "spill" to a portion of global memory call *local memory*, this portion is exposed to the kernel only by a per thread basis.

CUDA multiprocessors use the *Single Instruction Multiple Thread* (SIMT) architecture : each cycle, one instruction is executed by all the cores of a single SM. Threads are grouped into packs of 32 threads, called *warps*, to be scheduled by each SM. A SM can host multiple warps up to its physical limits, depending of the required configuration form the kernel. This model is most efficient when each thread in a warp executes the same instruction. In case of divergence, each branch is consecutively executed by the warp, with threads following the other branch doing nothing. Each thread disposes of its own program counter and of its own registers. Thread contexts of warps stay on the multiprocessor for the duration of their execution enabling a cheap and fast hardware scheduling of ready warps in spite of execution time and/or latency of instructions.

To fit to this architecture, the CUDA model subdivides on two, more coarse, levels than threads :

- It defines a *block* as a group of contiguous threads.All the threads of a block are divided into warps.The CUDA model imposes that all the warps of a block will be executed by the same SM. This way, multiple blocks may reside on a single SM and share its resources. The *shared memory* allows the threads of a block to communicate together.
- Finally, to organize these blocks, the model represents blocks as a *grid*.

An important notion when designing parallel algorithms is how sub-tasks can synchronize among themselves. CUDA provides two ways to synchronizing threads : at block level by establishing a barrier for all the threads in the block and at grid level CUDA ensures the programmer that all the block have been executed once a kernel finishes.

4.2. GPU Implementations

In this subsection, different steps leading toward refinement of parallelization implementation of the TFM algorithm on GPU are detailed : first the straigh forward implementation of the basic algorithm, then the implementation of the algorithmical optimization using symmetries and last its architectural optimization. In these implementations, computations are done using single precision floating point numbers, as previously validated in subsection 2.4.

4.2.1. Basic implementation

The first implementation is a straight forward transposition of the computations from the sequential approach (see figure 3).

Each block will be assigned to a single pixel of the image. In a block, threads will consecutively compute two time delays: the first one, from a transmitter to the physical point corresponding to the pixel and the second one, back from this point to the receiver. Once the round-trip time delay is computed by adding these two values, each thread will retrieve the corresponding sample from the FMC matrix stored in global memory and adds it to the pixel value. Common linear arrays are composed of 64 to 128 transceivers, thus 1024 to 16384 couples of time delays will be computed. Because there are more couples than threads in a block, threads may compute influence of several couples for its pixel. Then, each couple influence is added to the pixel value using an atomic operation to avoid conflicts within the block.

4.2.2. Symmetries on GPU

The second implementation, based on the symmetries approach shown in figure 4, is the result of the following algorithmical optimization. With a linear array of N_t transceivers, this approach reduces the computations from $N_t \times N_t$ computed time delays to N_t. To increase performances over the previous version, adaptations to the CUDA model should be carried on. In this version, a block is still assigned to a single pixel and each block needs enough shared memory to store a time delay for each transceiver. By working simultaneously on a whole pixel, computed time delay can be reutilized by storing these values into shared memory.

This kernel will reconstruct the image in two steps :

- First, the threads will compute and then store into shared memory the time delays between each transceiver and the physical point corresponding to the pixel. Due to the way threads are divided into warps, they have to be synchronized for each pixel to be sure that all time delays of a pixel are computed before combining them to compute the influence of a couple of transceivers.
- After synchronizing with all the threads of the block, each thread assembles the round-trip time delay of each couple from shared memory, then retrieves the corresponding sample from the FMC matrix in global memory, and at last, adds its contribution to the pixel value using an atomic operation.

4.2.3. Multiple Pixels per Block

The last implementation will use symmetries as seen on the previous section and will aim to optimize access to the samples in global memory. To optimize memory access on CUDA devices, threads from the same warp should make coalesced access to the global memory : aligned and sequential access (the latter is no longer required on Fermi GPU). This way, all the threads from a warp can access global memory through a single memory transaction which is subsequently stored into the cache (128-byte long on Fermi GPU).

Due to the nature of the simulated phenomenon, there is no way to predict the distance from a time delay to the next one before computing them. The underlying idea is to rely on the quasi-continuous nature of ultrasound waves : for a given couple of transmitter and receiver and for two physical points that are close to each other, the time delay should be close enough to allow the retrieval of more than one sample by each memory transaction as illutrated in figure 5.

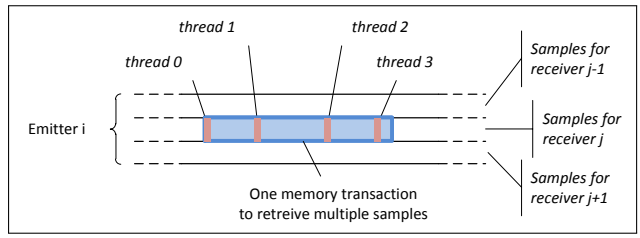

Fig. 5. Retreiving multiple samples by memory transaction

By enforcing spatial locality, this implementation is an architectural optimization of the previous algorithm aimed at producing a cache oblivious algorithm. Packs of N_{wg} pixels will be assign to each block thus threads will be divided into work groups of N_{wg} contiguous threads such that :

- no work group goes beyond the size of a warp;
- threads from each block are equally divided between workgroup.

Block will be assigned to 2D regions of the target image, each will work on N_{wg} pixels. Threads will work, as seen in the previous implementation, in two steps. First, the threads will compute and store into shared memory the time delay between transceivers and physical points corresponding to the processed zone. Then, after synchronizing on the block level, work groups will be assigned to different couples of transceivers. Inside a work group, threads will work simultaneously on N_{wg} different pixels for a set couple of transceiver.

This whole transformation is done by using indexes so the algorithm structure is essentially the same as the previous implementation.

5. BENCHMARKS AND PERFORMANCES ANALYSIS

This section will first outline the computational and the memory access complexities of each presented algorithm in subsection 5.1. Then benchmarking of the different algorithms on two different hardware will be exposed in subsection 5.2.

5.1. Algorithm Complexity

In this subsection, the study will deal with a linear ultrasonic array composed of N_t transceivers and a reconstructed

image of N_p pixels. In the TFM algorithm, the parameters N_t and N_p are crucial to evaluate the performances of the algorithm.

Furthermore, both on GPU and GPPs, two parts of the TFM algorithm are costly toward the performances :

- the number of access to the memory for reading the samples and writing the pixels, as for example the samples, are very large accessing to them is prone to cache miss;
- the number of time delay computations.

Tables 1 and 2 present the global complexities of these two parts accordingly to the parameters N_t and N_p for the basic algorithm (see figure 3) and its version using symmetries (see figure 4):

	Memory access	Time delay computations
Basic	$O(N_t^2 \times N_p)$	$O(N_t^2 \times N_p)$
With symmetries		$O(N_t \times N_p)$

Table 1. Complexities of GPP Implementations

	Memory access	Time delay computations
Basic	$O(N_t^2 \times N_p)$	$O(N_t^2 \times N_p)$
With symmetries		$O(N_t \times N_p)$
Multiple pixels per block		$O(N_t \times N_p)$

Table 2. Complexities of GPU Implementations

As the highest complexity of the algorithm is $O(N_t^2 \times N_p)$ both for memory access and for time delay computations, in the subsection 5.2, the execution times are normalized accordingly to this complexity.

5.2. Benchmarks

The following benchmarks were obtained on an Intel i7-2600K GPP and one of the two GPU on a Nvidia GeForce GTX 590, with the announced specifications shown at table 3. GGP implementations use SSE2 SIMD instructions.

	Intel i7-2600K	GTX 590 (1 GPU)
FLoating point Operations Per Second	112 GFLOPS	1244 GFLOPS
Memory bandwidth	21 GB/s	164 GB/s

Table 3. Architecture specifications

As seen in the previous section, performances are closely tied to the parameters N_t (the number of transceivers) and N_p (the number of reconstructed pixels). Thus, this paragraph shows the execution times observed on both GPP and GPU for two scenarios in which only one of the previous parameters varies. These scenarios are detailed in tables 4 and 5.

N_p	80×24	160×48	240×72	320×96	400×120
N_t	128				

Table 4. Scenario with N_t constant

N_p	400×120=48000							
N_t	16	32	48	64	80	96	112	128

Table 5. Scenario with N_p constant

5.2.1. GPU results

First of all, the figure 6 shows the results obtained in the case of the scenario described by the table 4 for each one of the three GPU implementations: basic implementation (see paragraph 4.2.1), implementation with symmetries (see paragraph 4.2.2) and implementation with multiple pixels per block (see paragraph 4.2.3). These results show that, for each one of these implementations, once the image size is sufficiently important, execution time becomes constant. For a small image size, launching a CUDA kernel implies a little overhead that computations cannot balance. Contrary to basic and symmetries implementations, multiple pixel per block is still a bit declining which indicates that the benefits of the method is not at the limits: as the resolution of the reconstruction image increases, more and more contiguous pixels benefit from the same samples.

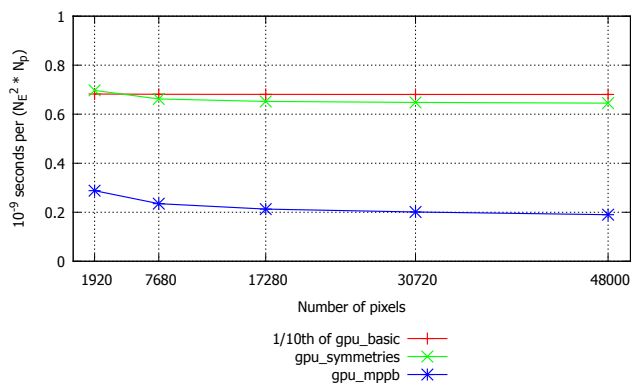

Fig. 6. Execution time of GPU implementations for a sensors of a fixed number of transceivers (128 transceivers)

Then, the figure 7 presents the results obtained in the case of the scenario described by the table 5. The inefficiency of the GPU for very small problem size is noteworthy. Beside, these peaks of inefficiency are more pronounced for the symmetries implementation rather than for the basic one. This highlights the heavy cost of synchronizing threads of a block on GPU between time delay computation and samples fetching. Once the number of transceivers is enough to balance synchronization and kernel launch, execution times become again constant. For implementations with symmetries and with multiple pixels per block, the complexity in terms of time delay computations is equal to $O(N_t \times N_p)$. Knowing that, as said in paragraph 5.1, the execution times have been normalized accordingly to the complexity $O(N_t^2 \times N_p)$. It is possible to say that these implementations are memory-bound

(i.e. memory access are more expensive than time delay computations). Otherwise, for the two previous implementations, execution time would be decreasing.

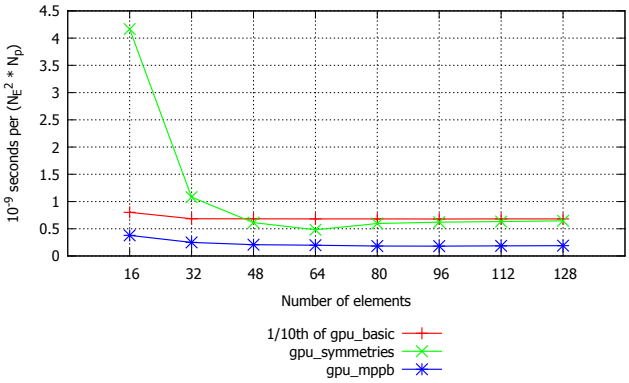

Fig. 7. Execution time of GPU implementations for a fixed image size (400×120 pixels)

5.2.2. *GPP results*

Concerning GPP implementations, as explained at the paragraph 3.3, two methods have been used in order to optimize TFM algorithm: multithreading with the OpenMP libray and SIMD instructions. Thus, in order to be able to analyze the efficiency of these two methods, they have been applied separately and together on the basic TFM algorithm (see figure 3) and its version using symmetries (see figure 4). Thus, eight versions of the TFM algorithm have been benchmarked which are detailed in table 6.

Version	OpenMP	SIMD
gpp_{basic/symmetries}		
gpp_{basic/symmetries}_omp	X	
gpp_{basic/symmetries}_simd		X
gpp_{basic/symmetries}_omp_simd	X	X

Table 6. GPP implementations

The figure 8 presents the performances of GPP implementations in the case of the scenario described by the table 4. As with GPU implementations, normalized execution times are constant once the image size is sufficiently important. Concerning the basic TFM algorithm implementations, as expected, the fastest implementation is the one using both multithreading and SIMD instructions. However, regarding the implementations using symmetries, the fastest implementation is not the one using both multithreading and SIMD instructions but the one using only multithreading. In fact, if using symmetries, the cost of time delay computation is reduced. This implyes more cache pressure on sample loads from the multiple threads when using the OpenMP library. This pressure increases again by using SIMD instructions and leads to a minimization of cache usage and finally to an inefficient implementation.

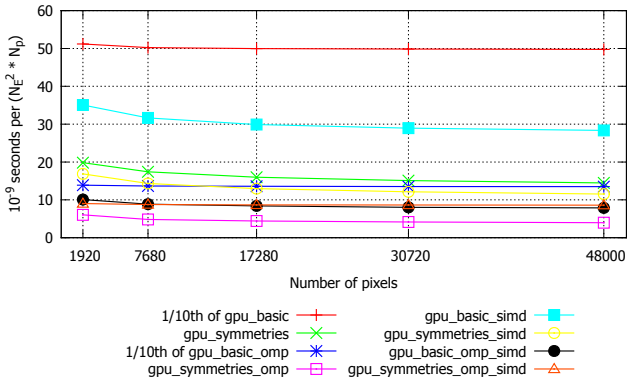

Fig. 8. Execution time of GPP implementations for a linear arrays of a fixed number of transceivers (128 transceivers)

The figure 9 details the performances of GPP implementations in the case of the scenario described by the table 5. The previous results are again observed in this figure. Once the number of transceivers reaches 48, execution times become constant. This figure shows that the GPP implementations using symmetries are also memory bound. The best implementation is still the symmetries algorithm using only multithreading. This indicates that even with slower computations (without using SIMD instructions), this implementation maximizes memory utilization.

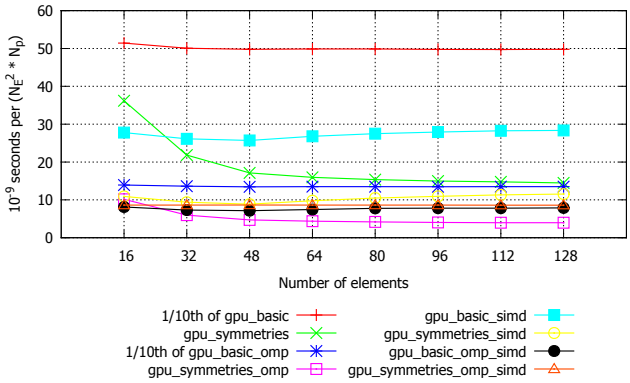

Fig. 9. Execution time of GPP implementations for a fixed image size (400×120 pixels)

5.2.3. *Benchmarks analysis*

Knowing that, in real applications, the TFM algorithm is never used to produce small images or with too small linear ultrasonic arrays, in this paragraph:

- In the case of the scenario described by the table 4, the scope is narrowed to image sizes greater or equal than 240×72.
- In the case of the scenario described by the table 5, the scope is narrowed to linear ultrasonic arrays with at least 48 transceivers.

The table 7 presents the average normalized execution times on all the resulting cases for each GPP and GPU im-

Implementation	ANET	Speedup
gpp_basic	4.98E-01	1
gpp_basic_simd	2.79E-02	17.9
gpp_basic_omp	1.35E-01	3.7
gpp_basic_omp_simd	7.80E-03	63.9
gpp_symmetries	1.59E-02	31.4
gpp_symmetries_simd	1.10E-02	45.7
gpp_symmetries_omp	4.36E-03	114.4
gpp_symmetries_omp_simd	8.64E-03	57.7
gpu_basic	6.81E-03	73.2
gpu_symmetries	6.87E-04	726.1
gpu_mppb	1.97E-04	2535.3

Table 7. Synthesis of ANET (average normalized execution times (10^{-9} seconds per $(N_t^2 \times N_p)$)) and speedups

plementations. In this table, all implementations are compared, in term of speedup, to the basic TFM algorithm on GPP (gpp_basic implementation) which serves as reference.

The table 7 shows that, concerning the basic implementation of TFM algorithm, gpu_basic and gpp_basic_omp_simd obtain similar speedups (respectively 73.2 and 63.9). Knowing the specifications of each architecture (table 3), it is possible to say that GPP has less computational power but a more efficient memory architecture than GPU.

Concerning the versions using symmetries, the fastest GPP implementation is gpp_symmetries_omp and obtains a speedup equal to 114.4 whereas the GPU implementation gpu_symmetries obtains a speedup equal to 726.1. As explained in the paragraph 5.2.2, gpp_symmetries_omp_simd is less efficient because of the limitations of GPP memory caches. On this point, the GPU benefits of a shared memory to manually manage local data and share it across threads.

This becomes particularly visible on the GPU implementation gpu_mppb which still use the bases of the symmetries algorithm and which optimizes its memory access to benefit both from this fast shared memory and from caches available on the GPU to access samples. This allows GPU to obtain a speedup equal to 2535.3.

6. CONCLUSIONS AND FUTURE WORK

The study presented in this article is the first step toward offering an efficient parallelized implementation of the TFM algorithm on workstations.

This article have shown that GPU and GPP can both be effective at accelerating the Total Focusing Method. Compared to the GPP scalar execution time, the multithreading allowed a $\times 114.4$ acceleration, whereas GPU reaches a$\times 2535.3$ speedup. It is noteworthy that the fastest GPP implementation is not the one using SIMD instructions. Due to SIMDization, computations become insignificant in comparison to memory access, resulting in too much pressure on the memory cache.

Inspection of planar interfaces has shown to be restricting, thus work is on progress to optimize implementations for more complex surfaces (from cylindrical to torical, and with combinations of different interfaces). These surfaces will require heavier computations to determine the delay laws, thus a shift from memory-bound to compute-bound behavior may be observed.

This way, it will be possible to draw more conclusions on the computing capacities of both architectures and the next step of this study will work out the optimal boundaries for implementing on each of them.

7. ACKNOWLEDGEMENTS

This work has been partly supported by the french *Agence Nationale de la Recherche* OPARUS project (ANR-10-COSI-010-01).

8. REFERENCES

[1] "CIVA : State of the art simulation software for Non Destructive Testing," `http://www-civa.cea.fr/`.

[2] Seydel J., "Ultrasonic synthetic-aperture focusing techniques in ndt," *Research Techniques in Nondestructive Testing*, vol. 6, pp. 1–47, 1982.

[3] I. Trots Y. Tasinkevych, A. Nowicki, "Element directivity influence in the synthetic focusing algorithm for ultrasound imaging," *Proceedings of the LVII Open Seminar on Acoustics*, pp. 197–200, 2010.

[4] D. Romero-Laorden, O.Martnez-Graullera, C.J.Martn-Arguedas, M.Prez, and L.G.Ullate, "Paralelización de los procesos de conformación de haz para la implementación del total focusing method," in *CAEND Comunicaciones congresos, 12 Congreso Espaol de Ensayos No Destructivos*, 2011.

[5] Tadeusz Stepinski and Fredrik Lingvall, "Synthetic aperture focusing techniques for ultrasonic imaging of solid objects," in *8th European Conference on Synthetic Aperture Radar, Aachen, Tyskland*, 2010.

[6] Caroline Holmes, Bruce W. Drinkwater, and Paul D. Wilcox, "Post-processing of the full matrix of ultrasonic transmitreceive array data for non-destructive evaluation," *NDT and E International*, vol. 38, no. 8, pp. 701 – 711, 2005.

[7] Matthias Jobst and George D. Connolly, "Demonstration of the application of the total focusing method to the inspection of steel welds," in *10th European Conference on Non-Destructive Testing*, 2010.

[8] R. Cypher and J.L.C. Sanz, "Simd architectures and algorithms for image processing and computer vision," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. (37) pp 2158-2174, 1989.

[9] D. Etiemble and L. Lacassagne, "Introducing image processing and simd computations with fpga soft-cores and customized instructions," *Workshop on Reconfigurable Computing Education*, vol. 6 pages, 2006.