Performance Analysis of an Ultrasound Reconstruction Algorithm for Non Destructive Testing

Antoine PEDRON [a], Lionel LACASSAGNE [a], Victor BARBILLON [a], Franck BIMBARD [b], Gilles ROUGERON [a] and Stéphane LE BERRE [a]

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

Abstract. The CIVA software platform developed by CEA-LIST offers various simulation and data processing modules dedicated to non-destructive testing (NDT). In particular, ultrasonic imaging and reconstruction tools are proposed in the purpose of localizing echoes and identifying and sizing the detected defects. Because of the complexity of data processed, computation time is now a limitation for the optimal use of available information. In this article, we present performance results on parallelization of one computationally heavy algorithm on general purpose processors (GPP) and graphic processing units (GPU). Although GPU implementation makes an intensive use of atomic intrinsics, it gave the highest performances. Only a dual 6-core GPP catches up its performances. Both architectures have been evaluated with different APIs : OpenMP, CUDA and OpenCL.

Keywords. non-destructive testing, ultrasonic reconstruction, parallelization, general purpose processors, graphic processing units

Introduction

Non destructive testing (NDT) consists in examining specimen or material integrity without damaging it. Ultrasonic Testing is one of the most used method, because of its sensitivity, its depth penetration, and its accuracy for positioning and dimensioning internal defects as well as its simplicity of implementation.

Ultrasonic testing consists in scanning the component under examination and acquiring signals at each probes positions. Acquired data are processed for visualization and analysis purpose, and different views are proposed. Besides these views visualizing raw data, different reconstruction algorithms can be proposed to give a representation of the data in real space.

This work is done in the context of the development at CEA LIST of the CIVA software platform dedicated to simulation and processing of NDT data [1]. Because of the complexity of the data to be processed, computation time remains a limitation for an optimal use of reconstruction algorithms. Acceleration of reconstruction and visualization algorithms would be a great step forward for the tools of analysis and automatic diagnosis. In this perspective, parallelization and exploitation of GPU are investigated. An ultrasound reconstruction algorithm called True Cumulated Views (TCV) has been chosen for optimization and parallelization with the two most common hardware platforms used nowadays, general purpose processors (GPP) and graphic processing units (GPU). GPU are well known to give high speedups, particularly when computations are highly parallel [2].

This article is organized as follow: first of all, section 1 presents TCV algorithm. Then, parallel implementations are detailed in section 2. After that, sections 3 deals with implementations analysis and finally 4 give experimental and performance results.

1. True Cumulated View Algorithm

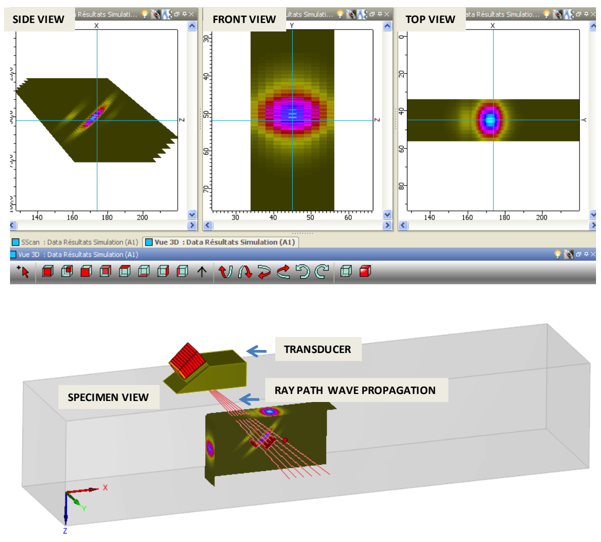


Figure 1. In the top part : views computed by True Cumulated View algorithm. In the bottom part : views assembled on a 3D model.

The *True Cumulated View* algorithm (TCV) offers a complete visualization of the data through top, side and front views of the specimen (figure 1). This algorithm postulates that both the specimen geometry and material properties are known. In complex cases, the geometry can be given as a CAD description.

First, a description of the algorithm will be given in subsection 1.1 and then, initial implementation will be presented in subsection 1.2.

1.1. Algorithm description

The inputs are organized as follow:

- N signals (N is the number of ultrasonic shots) : amplitude in function of time. The signals are results of an experimental acquisition or a simulation calculation.
- N ray-paths : geometrical polylines, which represent the path of theoretical beam deviation inside the specimen. At each vertex of the polyline is associated the time of flight value of the wave to reach this point. Time of flight between two

vertex are obtained through simple linear interpolation. These ray-paths are pre-calculated with CIVA simulation code [3].

- Area of reconstruction : 2D regular grid (that can be confused with the final image).

The Output is:

- An image : amplitude is assigned to each point of the reconstruction area.

The algorithm proceeds as follows. First, a projection of the signal to the ray-path is done. Each value (amplitude, time) of the signal is localized on the ray-path and converted to a (amplitude, (x,y,z)) value. Then the value is projected on a pixel. Second step consists in applying a mathematical morphology dilation filter to the output image in order to fill the holes between pixels with values.

The holes are the consequence of the gap between two ray-paths and also from signal sampling. The dilation filter is commonly used in image processing. TCV algorithms resemble to Maximum intensity projection algorithms which use this filter to refine the projections [4]. Moreover, this type of filter is well-known and benefit of many optimizations such as kernel separability [5] or Van Herk algorithm [6].

This algorithm has an **irregular workload** whose intensity depends on the data. Ray-paths can be either well aligned or disorganized, short or long, with a few or many segments and in most cases can be source of memory conflicts in case of parallelization.

1.2. Initial implementation

The initial implementation is a direct mapping of the *physical* algorithm. Sequential implementation samples each ray-path along. In the inner loop, each sample is projected onto a pixel. Then, dilation filter is applied. Usually, the dilation step is very fast and do not take more than 5% of the total time.

2. Parallel Implementations

Ultrasonic image reconstruction has to deal with important amount of data. Numerous elements are processed with the same operation. This pattern can be mapped onto GPU computing pattern which is based on large data sets with high parallelism and minimal dependency between data elements. Unlike GPP, GPU has to run a large number of threads to be efficient. GPU threads are not like GPP ones : launching a kernel with ten thousands threads takes about 12 μs with synchronization [7]. As a result, parallelization strategies are usually different. Two different strategies are discussed in this section, one on GPP and one on GPU. Post-processed dilation step, as discussed in subsection 1.2, will not be taken into account in this benchmark.

In subsections 2.1 and 2.2, GPP and GPU implementations will respectively be discussed.

2.1. GPP implementations

The GPP approach is straight-forward and, if using a p-core GPP, ray-paths are distributed among p threads. These threads are created by using the OpenMP library that automatically distributes a loop computation without modifying the existing code. Each thread runs the TCV algorithm and generates a local image. Then the images are merged by extracting the maximum value in each pixel (reduction).

2.2. *GPU implementations*

GPU implementation strategy is different. Assuming that GPU need to launch a very large number of threads (thousand of them), multiple image reconstruction cannot be used. Instead, one unique output image is used. Since different threads can potentially access the same address at the same time, max operations has to be done synchronously to be correct. These operations exist on Nvidia GPU since *compute capability 1.1* hardware and are called atomic operations. They allow to create a mutual exclusion section to enforce serial access to memory. At the moment, sections are restricted to basic operators like cas, max, add, or a few other unitary operators. *Atomics* operations were slow with 1.x hardware, but Nvidia announced with the latest generation of GPU (2.x) that they have improved their *atomic* instructions up to a factor $\times 20$ [8][9].

Like CPU, one implementation have been realized. TCV kernel parallelize the upper loop of algorithm so as to run one thread per ray-path. Each thread computes its own ray-path projection, using the same image as all threads, with the use of `atomicMax`.

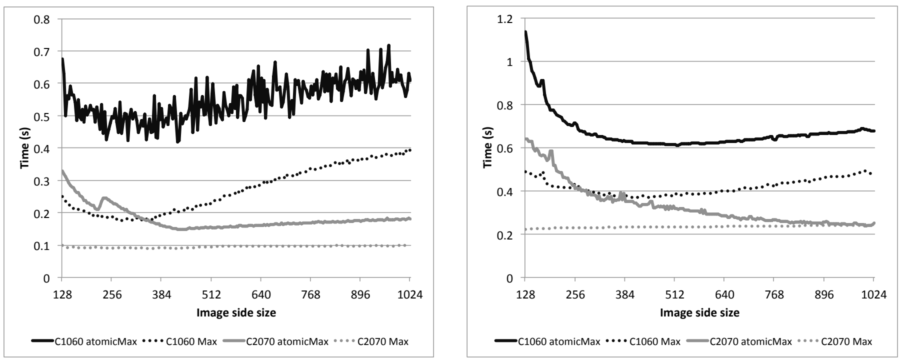
3. Implementations Analysis

Benchmarks in this section have been carried out with two Nvidia GPU : Tesla C1060 with 240 cores running at 1.3 GHz and Tesla Fermi C2070 with 448 cores running at 1.5 GHz.

One ultrasound defect response dataset has been selected as a representative subset of data that TCV use to deal with. PMF is a 150k ray-paths with 1k sample per signal set representing a 150M point projection for a full reconstruction.

Subsection 3.1 will present a benchmark of atomic performance on TCV computations. Then subsection 3.2 will be discussed.

3.1. *Impact of using atomic instructions*



(a) Random accesses (b) TCV projection

Figure 2. (a) Pseudo-random `atomicMax` and max operation evaluation for an image size from 128×128 to 1024×1024. (b) TCV GPU computation time for a fixed number of pixel accesses (PMF dataset, 150M point projection) with a variation on image size between 128×128 and 1024×1024

Given that GPU implementation is using *atomics* intrinsics and pseudo-random accesses depending on projection and control data, one can wonder about the performance of these operations.

On C1060, global memory is not cached [10] whereas C2070 includes two levels of cache. L1 is local to a streaming multiprocessor whereas L2 is global. L1 cache is not useful in our case because TCV has an intense atomics use whose memory transactions are done in L2. But since concurrent access intensity is low (*i.e.* compared to a histogram view), one can wonder how caches impact on TCV and more generally what cost has to be paid.

Figure 2 (a) presents the results of a pseudo-random access computation of `atomicMax` and max operators (ignoring concurrent accesses) with C1060 and C2070. 150M points are written in a 1D array so as to be able to compare with the experimental results of TCV. The results from the two computations are not equal, but this test allows to better understand the cost of the atomic operation compared to a lock-free one.

C1060 `atomicMax` curve is very noisy and basically slow. When compared to a simple max operator, `atomicMax` is about $2.5\times$ slower. With C2070, synchronization is still expensive when access intensity is high but the gap between `atomicMax` and max has been reduced with a slowdown lower than 50% for images larger than 350×350 meaning that *atomic* instructions latency has been greatly improved.

Figure 2 (b) shows TCV side view projection time on C1060 and C2070 with `atomicMax` and max operators. For images larger than 192×192, C1060 curves are outspread of a $1.5\times$ factor meaning that lock latency is basically free when access intensity is high enough. Lock latency increase with smaller images with up to a $3\times$ factor for 128×128 images. That show how expensive atomics intrinsics are on GT200 hardware.

On C2070, both curves are different than C1060's. Max curve is very stable with a 10% compute time increase between 128×128 and 1024×1024. The main difference come from `atomicMax` curve which is deceasing over time like C1060's but instead of following max curve with a factor, it reaches max performance for images greater than 800×800. This means `atomicMax` computation is not more expensive than a simple max operation when concurrency does not impact. Moreover, performance gain is significative between C1060 and C2070 with a $1.5\times$ to $2.8\times$ factor between `atomicMax` computation for 1024×1024 images.

In TCV algorithm, the use of *atomics* is mandatory to avoid errors of computations. But TCV projection show that with Fermi GPU, *atomics* can be used with a low performance decrease almost like others non-atomic operators for random accesses as soon as concurrency intensity is not too high.

3.2. Access sparsity

In order to get a better understanding of the differences between the three views and associated computation times, an histogram of the different memory access has been done on the different views. These histograms shows that PMF front and top views are very different than side view. For each view, the total amount of memory accesses is equal. The histogram focus on the *distribution* of these accesses and the memory footprint. For example, on side view, some pixels are written 3000 times, compared to up to 8000 times on top view and 14000 times on front view.

These differences mean that even with higher access intensity, atomic *intrinsics* can be faster when locality is higher than fully random accesses. Top view is the fastest because of its memory alignment. Given that atomics are passing through L2 cache, some accesses are using this cache to improve performance. On C1060, behavior is different because its L2 cache is read-only compared to Fermi's which is read-write. View differ-

Table 1. Profiling sample of PMF dataset processing for the three views on C2070.

	time (ms)	*# DRAM writes*	*ipc*
View type			
Side View	592	2.56e7	0.60
Top View	457	1.04e7	0.87
Front View	628	1.96e7	0.56

ences can be verified with profiler data where top view implies less *DRAM writes* than other views (table 3.2) which means that many accesses are using cache features.

4. Performance Benchmark

A benchmark has been run on two computers equipped with a quad core Intel Xeon E5472 running at 3.0 GHz, a dual 6-core Xeon X5690 running at 3,47GHz and two Nvidia Fermi graphic cards : a Tesla C2070 and a Geforce GTX 580. Since dilation compute time, fusion compute time or image output transfer time are negligible, it has been chosen to only include projection compute time. An OpenCL implementation has been developed to run on both GPP and GPU. Both Intel 1.1 and AMD 2.4 OpenCL implementations have been targeted. Rewriting from CUDA to OpenCL has been straightforward.

Subsection 4.1 will present OpenMP scalability results. Then, subsection 4.2 will compare OpenMP to OpenCL results on GPP. Subsection 4.3 will compare GPU performances with CUDA and OpenCL. Eventually, subsection 4.4 will conclude on global performances and OpenCL usage.

4.1. OpenMP Scalability

GPP compute time between the three views is very stable. For that reason, only Side View has been tested here. Results are presented in Table 4.1. As for the previous benchmark, OpenMP scheduler is set to *guided* since it gave in all tests cases the best performance.

Table 2. OpenMP scalability on PMF side view projection.

	1 thread	4 threads	12 threads	24 threads	4:1	12:1	24:1
E5472	4296.7	1105.0	n/a	n/a	3.88	n/a	n/a
X5690	2705.8	679.8	266.9	169.3	3.98	10.13	15.98

Both GPP scale almost perfectly. On this GPP, 12 thread parallelization give a good performance with a $10\times$ acceleration. Best performance is reached using all 24 threads with a $16\times$ ratio. An additional 33% performance gain comes from hyperthreading which improves core usage.

4.2. GPP : OpenMP and OpenCL

Table 4.2 present a comparison between OpenMP and OpenCL GPP implementations. On one side, OpenCL implementations run the same kernel using *atom_max* to handle concurrent accesses. On the other side, OpenMP implementation is lock-free and performs between 25% and 30% faster than both OpenCL's. Both OpenCL implementations

Table 3. X5690 projection compute time on OpenMP and OpenCL implementations from AMD and Intel

	Side	Top	Front
OpenMP	169.3	172.0	170.7
OpenCL AMD	215.3	220.0	217.1
OpenCL Intel	217.3	214.7	218.3

are very close. Compared to mono-thread OpenMP implementation, OpenCL gives a rough $12\times$ acceleration compared to the $16\times$ with multi-thread OpenMP. These results show that OpenCL can be a viable alternative to OpenMP. Of course, OpenMP has got a simpler interface when one wants to parallelize loops, but OpenCL brings multi-platform targeting which can be very interesting when an application runs on different hardware.

4.3. GPU : CUDA and OpenCL

Table 4. CUDA and OpenCL PMF side view projection time

	CUDA			**OpenCL**		
	Side	Top	Front	Side	Top	Front
C2070	257.8	109.7	295.6	254.1	107.1	298.7
GTX580	209.2	103.9	227.2	207.0	101.0	225.4

Table 4.3 give CUDA and OpenCL GPU results. Performances differ depending on the view. For a simple application as TCV can be, we observe that CUDA or OpenCL are very close in terms of performances. In terms of development complexity, CUDA and OpenCL are quite the same. These observations make OpenCL also a good alternative to compute on graphic processing units.

4.4. Overall Comparison

For TCV projection, OpenCL has given very interesting performances. The same code, initially developed for CUDA ran both on GPP and GPU without efforts. The dual 6-core X5690 runs between the best and the slowest computations on C2070 and GTX580. Top view being in favor of GPU caches, GPU are almost 60% faster than the X5690, whereas side and front views are more than 50% slower. GPU performances depend a lot on memory alignment and atomic performance. When GPP OpenCL performance is compared to GPU (CUDA or OpenCL), one can see that both GPU run in worse cases at the same speed than the GPP.

Multi-GPU implementation has not been evaluated yet, but adding a second GPU should give a very efficient result. It would only require a merge step as for GPP OpenMP implementation (which is non significative being very fast). This would also apply to a 3 or 4-GPU solution. In these conditions, a GPU solution would be the fastest.

Even if the top OpenMP performance is not reached in OpenCL, the difference can be acceptable. OpenCL is a fast growing standard that would avoid code legacy. Developments and evolutions should be easier. Moreover, the runtime portability allows the application to select the best target platform present on any hardware.

Conclusions and Future Work

This article has presented a study on the optimization and parallelization of a non-destructive evaluation ultrasound reconstruction algorithm. High performance have been reached on both GPP and GPU, more specifically with the latest generation of Nvidia GPU : Fermi. Both architectures have been evaluated on two different parallelization languages: OpenMP and CUDA as *native* API, and OpenCL which is able to target both platforms.

Although GPU performances depend on data, best results were obtained with this architecture. However, a dual 6-core Intel GPP narrows the gap reaching similar results. Even though best performances have been reached in native languages, OpenCL has shown to be profitable allowing a same code to run efficiently on different architectures.

Software optimizations combined to parallelization allow the user to get a very fast reconstruction in human interactive time which is a major step into NDE ultrasound reconstruction. These results are indeed very motivating and other algorithms studies are on the way. GPU are on fast growing curve and multicore GPP are evolving with a core number increasing each year and architecture improvements. Given that at the moment, both allow to reach high performances, the evaluation of future architectures is mandatory. In light of these facts, the usage of API like OpenCL can be very profitable allowing the user to mutualize the effort of optimization and parallelization.

References

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

[2] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov, “Parallel computing experiences with cuda,” *IEEE Micro*, vol. 28, pp. 13–27, 2008.

[3] G.Ribay, C.Poidevin, G.Rougeron, and B.Chassignole L.de Roumilly, “UT Data Reconstruction in Anisotropic and Heterogenous Welds,” *8th International Conference on NDE in Relation to Structural Integrity for Nuclear and Pressurised Components Abstracts*, 2010.

[4] Jos B T M Roerdink, “Multiresolution maximum intensity volume rendering by morphological adjunction pyramids.,” *IEEE Trans Image Process*, vol. 12, no. 6, pp. 653–60, 2003.

[5] T. Saidani, L. Lacassagne, J. Falcou, C. Tadonki, and S. Bouaziz, “Parallelization Schemes for Memory Optimization on the Cell Processor : A Case Study on the Harris Corner Detector,” *HiPEAC journal*, 2008.

[6] Marcel van Herk, “A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels,” *Pattern Recogn. Lett.*, vol. 13, pp. 517–512, July 1992.

[7] Vasily Volkov and James Demmel, “LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs,” technical report, Electrical Engineering and Computer Sciences University of California at Berkeley, May 2008.

[8] David Patterson, “The top 10 innovations in the new nvidia fermi architecture, and the top 3 next challenges,” Tech. Rep., NVIDIA, 2009.

[9] “Whitepaper, NVIDIA’s Next Generation CUDA Compute Architecture : Fermi,” Tech. Rep., NVIDIA, 2009.

[10] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos, “Demystifying GPU Microarchitecture through Microbenchmarking,” *ISPASS*, pp. 235–246, 2010.