# GPU Acceleration : OpenACC for Radar Processing Simulation

Maxime Martelli, Cyrille Enderli, Nicolas Gac, Antoine Vermesse, Alain Mérigot

# GPU Acceleration : OpenACC for Radar Processing Simulation

Maxime MARTELLI*†‡, Cyrille ENDERLI‡, Nicolas GAC*, Antoine VERMESSE‡, Alain MÉRIGOT†
*Laboratoire des Signaux et Systèmes, CentraleSupélec, CNRS, Université Paris Sud,
Université Paris-Saclay, FRANCE
†Laboratoire des Systèmes et Applications des Technologies de l'Information et de l'Énergie,
ENS Cachan, CNRS, Université Paris Sud, Université Paris-Saclay, FRANCE
‡Thales DMS France, Elancourt, FRANCE

***Abstract*—This article gives a methodological approach to accelerating an environment of a RADAR (RAdio Detecting And Ranging) simulation, from a single-core CPU implementation to a multi-core GPU implementation. We focus our attention on the most common tools for GPU programming like CUDA [2], but more specifically on OpenACC [6], a directive based parallel programming language. One of its promises is, with minimal modifications, to transform a CPU code to take advantage of many-core architectures, CPUs or GPUs alike.**

**Radar systems rely on many layers of testing, one of them being software validation. As technology moves forward, systems become increasingly complex, thus increasing the required processing power to simulate those systems. With CPU performance stalling, it is imperative to switch to alternative architectures. Our contribution is providing key steps for accelerating a software simulation of a radar algorithm on a GPU, with a particular focus on performance but also on the ease of programming. Maximum achieved execution time speedup on GPU architecture for our typical use case of radar processing is of 8.2 for CUDA and of 4.56 for OpenACC compared to the reference implementation on CPU.**

***Index Terms*—RADAR, GPU, Simulation, OpenACC, CUDA**

## I. Introduction

With the knowledge of our world constantly expanding, physical and mathematical modelling are scaling, both in precision and complexity. Those models require an ever-growing processing power, and, due to the lack of expertise or available resources, they are most of the time simplified, and therefore less accurate.

Technological advances in computer science relies on new specialized architectures, pushed by hot trends, like Artificial Intelligence [13], Self-driving Automobiles [12], or Data Storage. Leaders of the semiconductor industry have been pushing for years those new architectures, just like Apple in its most recent mobile custom chip. For example, the Apple A11 Bionic [1], which powers the iPhone 8, 8 Plus, and X, contains a six-core Central Processing Unit (CPU), with two high-performance cores, and four energy-efficient cores, as well as a Graphic Processing Unit (GPU), a motion co-processor, an image processor, and a neural engine. Such heterogeneity in a single chip showcases the need for specialized architectures in a single platform, to delegate specific processing to the most efficient system.

The democratization of heterogeneous computing is correlated with the desire to find a new growth driver for computer hardware, because Moore's Law is coming to a halt in 2021 [9]. Therefore, instead of increasing the raw power of an architecture, more focus is given to algorithm architecture adequacy to get a more efficient heterogeneous system. This is achieved by building suitable processors for different algorithms categories.

Radar processing is a computing intensive domain where there is a real need for more processing power. In the airborne field, both for civilian and military systems, radars are of paramount importance, and to ensure the reliability of sensors, it is mandatory to test them upstream. An essential validation step is to simulate as accurately as possible the targets that these sensors might encounter and need to spot in their final environment. Thus, radar simulation, whether it concerns its environment or its processing, is strategic to validate algorithms before designing the embedded system.

In many testing scenarios, target emulation can be achieved by connecting the emission and the reception of the radar with a link, for example an optical fibre, which induces a temporal delay proportional to the length of that fibre. While it is possible to emulate moving targets, by adding Doppler shifts, the emulation of several targets in parallel requires a specific optical fibre length per target, making the emulation even more complex. Therefore, important testing, with many targets and specific trajectories can mostly take place during radar simulation at the software level, or during real test flights, that have the disadvantage of being costly and restrictive.

Having a reliable software radar simulation is crucial to reduce costs and improve the reliability of the system, and the use case presented in this paper is part of an industrial simulation and validation radar processing tool. This simulator includes many possible environment, like for example clouds, motorways, seas or lakes, land, and moving targets (Air, Land, Sea). This is one of the upstream stages in the development of a radar, which consists of simulating both the environments and the radar to test its algorithms and, to a certain extent, its final analogue sensors.

As those radar processing algorithms gain in complexity, there is a real need of software acceleration for radar simulations.

As mentioned earlier, one solution is to evaluate the strength of other architectures than CPU. Most of modern parallel computing systems are heterogeneous, often including at least a CPU and another accelerator, such as a GPU, a FPGA, or a Manycore CPU. To program such systems, one can use programming frameworks such as OpenMP [8], OpenCL [7], OpenACC or CUDA, that allow expressing the inherent parallelism of each architecture and handle memory transfers. Some of those frameworks are focused on portability from one architecture to another, while also providing a good level of performance, like OpenACC. It is a directive based language that enables a CPU code to be adapted to GPU using preprocessing directives. One of its main advantage is that this ported program can still be compiled for both GPU and CPU.

In this paper our contribution consists in providing an acceleration methodology for OpenACC on GPU applied to a radar simulation environment and compare it to both the CPU reference and to an optimized CUDA implementation. GPU implementation of radar algorithms has been evaluated before for software radar [14] and real radar processing execution [10], and the results show an efficient speedup compared to a standalone CPU implementation.

The reminder of this paper is organized as follows : in Section II, we quickly present some radar concepts. Then, we introduce in Section III the Radar Simulation tool and the considered environment, as well as the tools and concepts used for its acceleration. Section IV describes our methodology for OpenACC acceleration, while showcasing the taken optimization steps. Finally, Section V consists of discussing obtained results, in raw performance as well as in programming productivity.

## II. Radar : preliminary concepts

At the start of the 20th century, the development of the radio and wireless communication paved the way for the radar. Now used in a wide range of applications, from meteorology to warfare and defence systems, it is nowadays a crucial technology.

Basic Doppler radar uses the Doppler-Fizeau effect [3]. The microwave signal issued by the antenna has a precise frequency, and the echo received from the reflection by a target has another frequency. By correlating both frequencies, the radial speed of the target can be calculated. In the case of a Doppler radar, it is possible to construct an **A**mbiguous **R**ange **D**oppler (**ARD**) map as illustrated in Fig. 1. Its goal is to detect the targets and accurately measure their range and speed.

This map is a two-dimensional graph, with the speed of the target on the X axis, and its range on the Y axis. Each rectangle on this map corresponds to a target detected by the radar. Actual ARD maps also contain additional echoes from the radar environment.

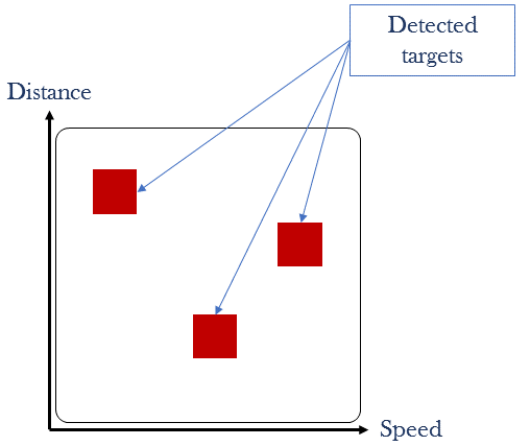


Fig. 1. Ambiguous Range Doppler Map

## III. Experimental setup

### A. Digital Echo Simulator

The **D**igital **E**cho **S**imulator (DES) is an industrial simulation and validation tool in support to the development of innovative signal processing algorithms for airborne radars. With the DES, radars are accurately modelled in order to simulate in a very representative fashion the echoes received from an environment that may include clouds, dense forests, deserts, oceans, targets and jammers, as illustrated in Fig. 2. While analogue components and filters are taken into account in the simulation, the echoes produced by the DES are located at the radar I/Q demodulated output. It thus provides a relevant means to validate signal processing algorithms without the need of costly flight tests.

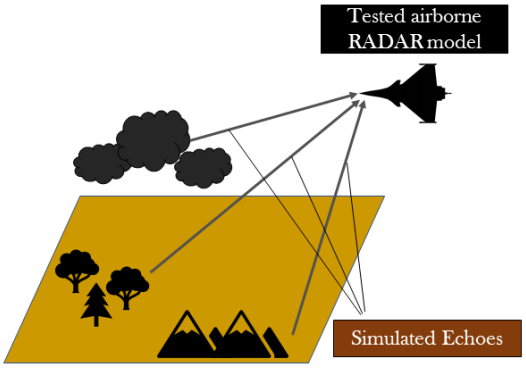


Fig. 2. Digital Echo Simulator

The variety and precision of those simulations are crucial for the reliability of the overall system, and this approach is mainly used to improve its robustness. Amongst the vast library of configurable environment the DES contains, our acceleration methodology focused on one in particular, a radar jammer which we refer to as the **L**ocalized **N**oise **J**ammer (**LNJ**).

### B. Localized Noise Jammer

As explained in section II, one step of radar processing alorithm is to construct from radar echoes an ARD map, that reflects and characterizes in range and speed the targets spotted by the radar. The goal of an LNJ is to jam the radar in a specific

way that it will mask a target, by saturating a localized zone around this target on the map.

Fig. 3 illustrates this principle, and shows what incidence an LNJ would have on the ARD map compared to Fig. 1.

The goal of the Localized Noise Jammer environment is to blur a rectangle zone on the final ARD map with specific ranging, to prevent the radar from accurately detecting a possible target inside. To achieve that, it must calculate what echoes the radar is supposed to receive for its radar processing to generate such a map. This jamming algorithm was already implemented on CPU in the DES, and our objective was to speed it up on GPU, while evaluating OpenACC programming.

### C. GPU parallelism

For a GPU developer, the co-processing platform usually consists of one host, usually a CPU and one or more GPUs. In modern applications, some calculations can easily be parallelled. Applications that handle large amounts of data can take a lot of time to execute and this time could be reduced by parallelling operations: physical phenomena can be calculated independently of each other, images to be analysed can be cut in portions and a video stream can be cut image by image. The parallelization of the data refers to the capability of a program to handle in parallel and independently these arithmetic instructions. For example, a multiplication of two square matrices of one thousand rows and columns needs one million elementary multiplication, independent from one another, which can therefore be easily parallelled.

When it comes to OpenACC implementation, Listing 1 illustrates some directives to port a CPU code to a GPU architecture. The main advantage of this framework is to allow backward compatibility of the transformed code. For example, when compiling the program for a CPU execution, all **#pragma** directives will be ignored. Line 4 illustrates the mechanism to create, from a host pointer, a corresponding memory object in the GPU address space. This prevents, contrary to CUDA for example, the need to add more lines to create another memory object specific to the device. Line 5 allow the OpenACC compiler to understand that the function3_par will be launched by the host on the GPU, using *v1* and *v2* as data on the GPU. Finally, Line 16 underlines some parallelism directive to tune the granularity of the following loop parallelization.

However, this transition to a more abstract description prevents more complex memory optimizations.

**Listing 1:** OpenACC directives for GPU execution

```
1  void main() {
2    float2 v1[S1], v2[S2];
3    ... some processing ...
   /* Device creation and copy from host
      pointer                                */
4    #pragma acc enter data copyin(v1[S1], V2[S2])
   /* Launching GPU execution and
      linking memory                         */
5    #pragma acc host_data use_device(v1, v2)
6    function3_par(v1, v2) ;
7  }
   /* Sequential function on GPU             */
8  #pragma acc routine seq
9  inline float2 function1_seq(float2 a, float s) {
10   float2 c;
11   c.x = s * a.x;
12   c.y = s * a.y;
13   return c;
14 }
   /* Parallel function on GPU               */
15 void function3_par(float2 * restrict a, float s) {
   /* Tuned loop parallelism                 */
16   #pragma acc parallel loop deviceptr(a)
17     num_gangs(GANGS) vector_length(SIZE)
18     num_workers(NB_WORKER) independent
19   for (int i = 0; i < size; i++ ) {
20       function1_seq(float2 a[i], s);
21       ... some processing ...
22   }
23 }
```

### D. CPU-GPU Platform

Our co-processing platform consists of an Intel Xeon E5-2667 CPU [4] and a Nvidia GTX 1080Ti GPU [5]. The CUDA toolkit version is 8.0, OpenACC version is 15.10. The profiling tools used in this article are the nvcc visual profiler and the ones included in the PGI OpenACC compiler. Specifications are shown at Table I.

TABLE I
HARDWARE SPECIFICATION

| Specifications | CPU | GeForce GPU |
|---|---|---|
| Type | Intel Xeon E5-2667 | GTX 1080Ti |
| Core Frequency | 2.9 GHz | 1.48 GHz |
| Physical Cores | 6 | 3584 |
| Global Memory | 96 GB | 11 GB |
| Memory Bandwith | 51.2 GB/s | 484.4 GB/s |
| FP32 Performance | 500 GFLOPS | 11.34 TFLOPS |

## IV. Methodology

This section showcases the different implementations on the GPU plaform with OpenACC. The methodology is split in three main categories, Algorithm analysis, Memory considerations, and Parallelism expression, and each OpenACC optimization is explained in the corresponding category. The achieved execution time speedup compared to the reference execution time on CPU is detailed in Fig. 4.

Our platform includes an Nvidia GPU, and, to give an exhaustive comparison of GPU acceleration, we implemented an optimized CUDA version, also included in Fig. 4.

For all versions, the achieved *optimization speedup* include memory transfer time.

### A. Algorithm analysis

**Code analysis**

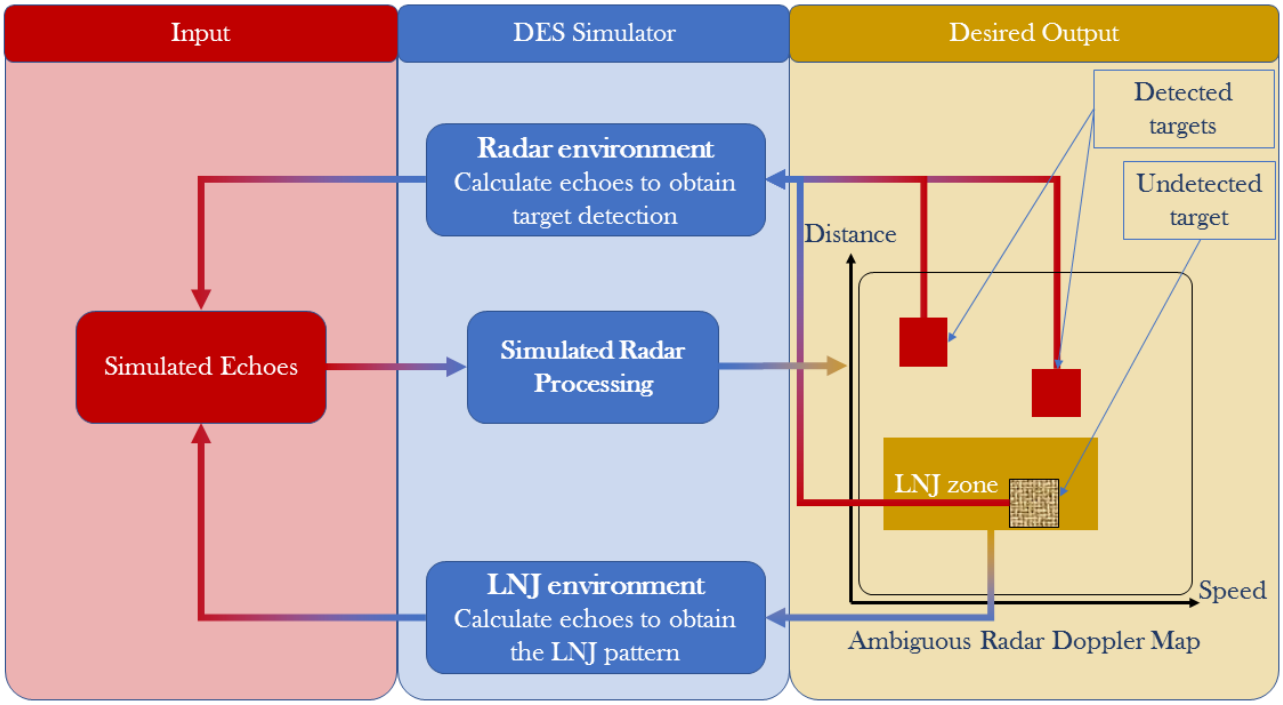


Fig. 3. Localized Noise Jammer and the DES Simulator

The first step, which conditions the rest of our acceleration methodology, is to analyze the intrinsic properties of our algorithms. Therefore, it consists of verifying that the algorithms performed on the reference architecture can be implemented on the target architecture. This includes, for example, looking for specifics such as recursiveness. Generally, recursive functions cannot be effectively paralleled on GPU. Thus, they should be avoided as much as possible.

When such processing is essential, and can only happen recursively, it can be faster to stop the calculations on the GPU, transfer the needed data back to the CPU, and perform the recursive processing on the CPU instead. Afterwards, new data can be copied to the GPU, and following processing can resume. Time lost in transferring data can still be not too much compared to the possible execution slowness on the GPU.

**Algorithm conversion**

The second step consists in adapting unsuitable processing on the target architecture, while making sure the precision of both versions remains comparable, for example that the output data are identical, or in accordance with the specifications.

The LNJ environment uses Fast Fourier Transforms, and the CPU algorithm is from the recursive kissFFT library. According to previous remarks, there were two possible options. The first one was to do all the data-parallel friendly computation on the GPU, and execute the original kissFFT algorithm on the CPU, while switching contexts between the two architectures. The second option was to find another FFT algorithm, this time optimized for GPU execution. In our case it was the cuFFT Nvidia algorithm.

Considering the FFT computation alone, we opted for the second option, the first one being slower (even without the data transfer) than the computation with cuFFT. Because we modified the original algorithm, we had to make sure we did not interfere with the specifications. The measured absolute error difference between the kissFFT and the cuFFT in our algorithm were lesser than 0.01%, and well below the specification threshold (***Optimization 1***).

**CPU/GPU Code partition**

The point here is to partition the code according to previous remarks. Co-processing has the disadvantage of adding transfer time between architectures, but latency can be hidden by executing concurrently calculations and memory transfers, to take advantage of the strength of the different architectures. Usually, one can rely on profiling tools, like the ones included in the PGI Compiler for OpenACC. By analysing the memory access in the loops of a program, it will give information on possible parallelization of some portion of the source code. Of course, automatic tools can not be used exclusively, and the programmer must push the analysis deeper to spot more intricate execution patterns.

### *B. Memory considerations*

The common difficulty in co-processing is the handling of memory objects. Because GPUs need to hide the latency by having memory transfer and compute units working simultaneously, it is crucial to take time for memory optimization.

**Data Structure**

Data distribution will greatly impact performance, because the layout of a memory object will dictate its coalescence [11]. For example, Array of Structures are the most conventional

layout (Listing 2) and the most used in traditional computing. Data for different fields are intertwined. This is often more intuitive and supported directly by most programming languages, but with poor performances in parallel programming. On the other hand, Structure of Arrays is a layout where data are gathered per element (Listing 3). Access is faster for data parallel programs (***Optimization 2***).

**Listing 2:** AoS

```
1 typedef struct Point {
2     int x;
3     int y;
4 } tPoint;
5 tPoint AoS[N];
```

**Listing 3:** SoA

```
1 typedef struct Point {
2     int x[N];
3     int y[N];
4 } tPoint;
5 tPoint SoA;
```

**Data locality**

Another optimization can be achieved by delimiting memory regions, thanks to the directives *enter data* and *exit data*, and, with the clauses *copyin* and *copyout* to choose the data to be transferred to the GPU. Other directive like *update* will refresh the CPU data from the GPU (***Optimization 3***).

**Non-intersecting data**

For optimal parallelism, the programmer can specify to the compiler two keywords, *const* and *restrict*. The first one is to indicate that the data is read-only, while the *restrict* indicates that the corresponding pointer is the only one pointing in its memory zone during the instance of our application. This limits aliasing effects, thus allowing the compiler to implement deeper optimizations (***Optimization 4***).

### C. Parallelism expression

**Vectorisation**

OpenACC defines three levels of parallelism, the finest scale being *vector* parallelism. It works like SIMD parallelism, and operations are carried out for all indexes of a vector simultaneously. For example, using *int8* types (i.e. a vector of 8 integer values) for vector addition will reduce execution time by a factor of 8. Depending on the problem, the programmer must choose the granularity to achieve best performance.

**Loop unrolling**

Loop parallelization is at the centre of parallel programming. The first step is to make sure there is no loop-carried dependency between each iteration, otherwise the compiler will not be able to fully extract its parallelism. It is possible, using *reduction* and *atomic* clauses to fix data dependency through a loop (***Optimization 5***).

## V. Results and Discussions

**Execution time**

As illustrated in Fig. 4, we compare the execution time speedup to the reference CPU mono-core execution. For all GPU optimizations, OpenACC and CUDA alike, measured time includes all data transfers between the CPU and the GPU.

The first optimization (*OpenACC Opt1*) achieved to be slower than the reference CPU version. This is due to the fact that there was at this point no memory optimization, and the goal of this step was to have a first functional code running on the GPU.

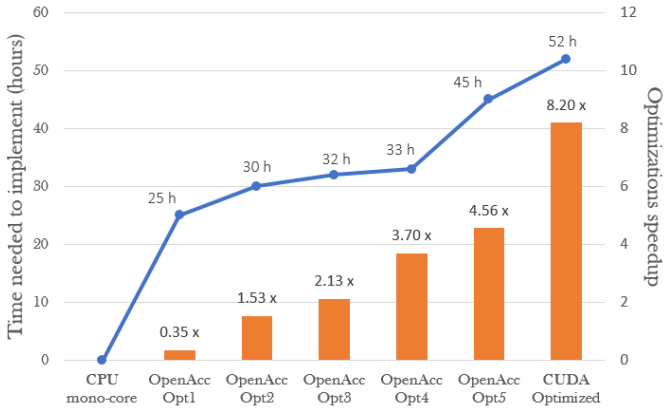


Fig. 4. Execution time speedup & Time needed to implement - CPU - OpenACC GPU - CUDA GPU

*OpenACC Opt4* shows that handling memory objects is crucial to achieve substantial performance, with a speedup of 3.7 compared to the reference execution. Finally, the best achieved OpenACC speedup (*OpenACC Opt5*) still falls short compared to a fully optimized CUDA implementation. This is mostly due to the fact that OpenACC still does not handle optimally some memory transfer. According to Nvidia, this point is going to be improved in a later iteration of the tool.

**Programming time**

Considering programming productivity, Fig. 4 shows the time needed to implement each version, cumulatively.

The slow implementation of the first optimization effort (*OpenACC Opt1*) is independent of the chosen language. However, it depends on the target architecture. This comes from the fact that this step relies on the analysis of the existing code, and in its adaptation to this new architecture.

The three next implementations (*OpenACC Opt2* to *OpenACC Opt4*) are achieved rather quickly, and consist in tuning memory transfers between the host and the accelerator.

*OpenACC Opt5*, on the other hand, is more tedious, as it is the last step, clustering all the other optimization parameters.

Because GPU programming concept is similar whatever tool is used to express it, transition from OpenACC code to CUDA code is fairly easy, and the CUDA optimized was written from the most optimized OpenACC version (*OpenACC Opt5*). If a programmer were to use only CUDA, its optimization methodology (and the time needed for implementation) would have followed a similar pattern as the one described in Fig. 4.

**Code modification**

Table II shows the parallelization effort of our reference application as the percentage of code lines written in OpenACC and CUDA. From the reference code, the *host code* refers to the part that executes on the CPU, including setting up the co-processing platform, whereas the *kernel code* refers to the portion of the code that executes on the GPU.

TABLE II
CODE MODIFICATION

| Version | Host Code [%] | Kernel Code [%] |
|---|---|---|
| OpenACC Opt1 | 10.5 | 27.4 |
| OpenACC Opt2 | 14.7 | 30.5 |
| OpenACC Opt3 | 16.9 | 30.5 |
| OpenACC Opt4 | 20.2 | 30.5 |
| OpenACC Opt5 | 25.7 | 35.8 |
| CUDA Optimized | 35.7 | 59.7 |

We may observe that both host code and kernel code are significantly modified for both the latest OpenACC and CUDA versions. Considering the Host Code modifications, in order to manage the co-processing calculations, a certain number of impressible lines of code must be written, explaining the 10.5% overhead for the *OpenACC Opt1*. The further we tune our memory handling, the more lines are needed in host code. Even then, *OpenACC Opt5* still needs fewer rewriting than the CUDA optimized version.

Kernel code modifications for OpenACC show, even for the first version, a substantial rewriting of the original source code. This is mainly due to the inadequacy of the original algorithm for data-parallel architecture. Using the cuFFT library is one of the main reasons why the *OpenACC Opt1* has a 27.4 % code modification. On the other hand, kernel code differences between *OpenACC Opt1* and *OpenACC Opt5* is low, consisting only on modifying some directives for different parallel execution methods.

Based on the collected data, using OpenACC still reduces the programming effort compared to CUDA, both for host and kernel code. But, code modification remains significant, mainly because the original code needed substantial rewriting to be adapted for data parallel architecture.

**CUDA or OpenACC : why not both?**

From the results highlighted in this section, one must ask himself what is the best tool to use between OpenACC and CUDA. In this particular use-case, because the parallelism needed significant rewriting of the code, both versions are substantially different from the CPU reference code. However, OpenACC programs are easier to read than CUDA ones.

Superimposing Speed of Implementation and Execution Time show that the most efficient OpenACC version is *OpenACC Opt4*. Afterwards, optimization efforts take much time to achieve a small speedup. At this point, it is best to transition to CUDA for a fully optimized version.

## VI. CONCLUSION

In this paper, we presented how some optimization steps impacted performance of a radar processing algorithm, from a CPU mono-core architecture to a many-core GPU. The maximum achieved speedup with OpenACC was of 4.56, and with CUDA of 8.2. OpenACC is easy to use and has a good readability for a CPU programmer. Its promise to only slightly interfere with the CPU code was, in this use-case, not really achieved, because the initial algorithm was not fully data-parallel friendly. However, OpenACC still needs lesser code modifications than CUDA, and, on use-cases adapted for data-parallelism, it can still run on the CPU, the compiler ignoring the OpenACC directives. Overall, this programming model can be viewed as a good solution for GPU prototyping, but to harness the best performance on GPU architecture, programmers should eventually switch to CUDA.

Because OpenACC and CUDA share most of the same conceptual view of the GPU architecture, one can transition to one to another fairly quickly, easily switching one directive for another. This can be reassuring for programmers as they do not have to commit solely to OpenACC or CUDA. Recently, Nvidia acquired one of the key OpenACC compilers, *PGI Accelerator™*, and communicated on the next improvements of the OpenACC framework, promising to address memory slowness and converging with CUDA in terms of performance.

Future works are focused on porting this code on OpenCL for FPGA implementation, as well as trying a new algorithmic approach to avoid using FFT in the image space, and use convolutions instead.

## REFERENCES


[1] Apple A11 Bionic URL https://en.wikipedia.org/wiki/Apple_A11
[2] CUDA Accelerated Computing URL https://developer.nvidia.com/cuda-zone
[3] Doppler-Fizeau effect URL https://en.wikipedia.org/wiki/Doppler_effect
[4] Intel Xeon E5-2667 specifications URL http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2667.html
[5] Nvidia GeForce 1080Ti specification URL https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877
[6] OpenACC Specification URL https://www.openacc.org/specification
[7] OpenCL standard URL https://www.khronos.org/opencl/
[8] OpenMP Specification URL https://www.openmp.org/
[9] The International Technology Roadmap For Semiconductors 2.0. Semiconductor Industry Association (2015)
[10] Degurse, J.F., Dugrosprez, B., Marcos, S., Savy, L., Molinié, J.P.: Architecture GPU pour radar de surveillance spatial et pour radar aéroporté. Gretsi (2013)
[11] Fauzia, N., Pouchet, L.N., Sadayappan, P.: Characterizing and enhancing global memory data coalescing on GPUs pp. 12–22
[12] Lin, S.C., et al.: The Architectural Implications of Autonomous Driving: Constraints and Acceleration. Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA pp. 751–766 (2018)
[13] Master, P., Furtek, F.: A New Computing Architecture Is Needed for Artificial Intelligence Applications. EE Times (2017)
[14] Zhang, Q., Deng, Y.: Toward of a GPU accelerated software navigation radar. IEEE 4th International Conference on Software Engineering and Service Science (2013)