Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

OPEN ENGINEERING (Belgium): “RF Solver on multi GPUs”

S. Costarelli[a], P. De Vincenzo[a], R. Dubois[b], K. Hasnaoui[b*]

[a]*Open Engineering, Belgium*
[b]*IDRIS – Orsay, France*

Abstract

The Open Engineering is a high-tech SME which was created in 2001 and located in Liege (Belgium). The company is active in the Computer-Aided Engineering (CAE) market. They design, develop, and sell the OOFELIE::Multiphysics software. OOFELIE::Multiphysics is used to conceptualize, design, analyze, and optimize various types of systems before starting time-consuming and costly build-and-test cycles. The goal of this SHAPE project is to develop in partnership with the IDRIS team, a multi GPUs version of the CUDA RF Solver in order to address problems that do not fit in the main memory of a single GPU; and to analyse its performance on an HPC multi-GPUs host.

1 Introduction

The Open Engineering Company is a high-tech SME from Belgium that provides Computer-Aided Engineering (CAE) software tools and services in the field of multiphysics. This has applications in Sensors and actuators, including MEMS (Micro-Electro-Mechanical Systems) and Microsystems, Optomechanical systems including MOEMS (Micro-Opto-Electro-Mechanical Systems) and Multidisciplinary systems involving interaction between a fluid medium and other structures.

Radio Frequency (RF) devices constitute a major class of components in the MEMS landscape as they are very important for wireless communications (IoT, M2M) and 5G.
The modelling and simulation of RF-MEMS involves several types of physics and couplings: coupled electrostatics-mechanics; electromagnetic wave (EM) propagation; coupled EM-thermal; thermo-mechanics; stiction; humidity; just to mention a few, and these effects have an impact on the reliability of the device.
Open Engineering has already developed and validated an electromagnetic wave propagation coupled with temperature solver for RF MEMS applications. Its core algorithm is based on the FDTD method. Two versions exist respectively written in C++ and CUDA. In the framework of this PRACE SHAPE (10th call application) project, with the cooperation of IDRIS (France), a multi-GPU version of the CUDA solver was developed to study pure EM problems. Indeed, working together with IDRIS, Open Engineering wants to have a multi-GPU version of its code based on a single-GPU and an OpenMP version that they already have. The plan was then to analyse the performance of the new code. Developments carried out during this project should improve performance and reduce memory usage which should then lead to faster, more accurate results thus improving the services provided to customers. The plan was also to investigate potential MPI version of the code.

The main partners for this project are the following:
- Pascal De Vincenzo (Open Engineering – General Manager),
- Santiago Costarelli (Open Engineering - Engineer/Technical contact),
- Rémy Dubois (Idris – HPC Engineer),
- Karim Hasnaoui (Idris/MdlS – HPC Engineer).

* Corresponding author email address: karim.hasnaoui@idris.fr

2 Available resources

All the development has been done at the Jean Zay machine located at Idris. Jean Zay is an HPE SGI 8600 computer composed of two partitions: a partition containing scalar nodes, and a partition containing accelerated nodes which are hybrid nodes equipped with both CPUs and GPUs. All the compute nodes are interconnected by an Intel Omni-PAth network (OPA) and access a parallel file system with very high bandwidth.

Illustration 1: The Jean Zay supercomputer located at Idris (C.Frésillon/ IDRIS /CNRS Photothèque)

The scalar partition is composed by 1528 pure CPU nodes based on:

- 2 Intel Cascade Lake 6248 processors (20 cores at 2.5 GHz), namely 40 cores per node,
- 192 GB of RAM memory per node,

and the accelerated partition is composed by 643 GPU nodes based on:

- 261 four-GPU accelerated compute nodes with:
    - 2 Intel Cascade Lake 6248 processors (20 cores at 2.5 GHz), namely 40 cores per node,
    - 192 of memory per node,
    - 4 Nvidia Tesla V100 SXM2 GPUs (32 GB),
- 31 eight-GPU accelerated compute nodes, currently dedicated to the AI community with:
    - 2 Intel Cascade Lake 6226 processors (12 cores at 2.7 GHz), namely 24 cores per node,
    - 20 nodes with 384 of memory and 11 nodes with 768 of memory,
    - 8 Nvidia Tesla V100 SXM2 GPUs (32 GB),
- 351 four-GPU accelerated compute nodes with:
    - 2 Intel Cascade Lake 6248 processors (20 cores at 2.5 GHz), namely 40 cores per node,
    - 192 of memory per node,
    - 4 Nvidia Tesla V100 SXM2 GPUs (16 GB).

For this project accounts have been created for all the collaborators at the Jean Zay machine. 1000 GPUs cores hours and 50000 CPU cores hours have also been provided for the full project duration.

3 Test cases

For the study, 2 tests cases and 3 benchmark cases have been provided by the SME:

- antenna (nx=62, ny=102, nz=50),
- antennaUHF (nx=152, ny=90, nz=49),
- case 10 10 (nx=136, ny=99, nz=278),
- case 20 10 (nx=171, ny=104, nz=265),
- case 30 10 (nx=335, ny=210, nz=248).

where nx, ny and nz are the mesh sizes without PML along the x, y and z axes respectively.

4 Dynamical and static analyses

Before doing any development, the first step was to work on the portability and to identify which parts of the given code can be optimized by doing static and dynamic analyses. The three main tools used for this study were:

- Cppcheck v1.87,
- Valgrind v3.14,
- ARM Forge v 20.1.2.

4.1 Static analysis

The static analysis has been performed by using Cppcheck [1][2]which is a static code analysis tool for the C and C++ programming languages. Cppcheck supports a wide variety of static checks that may not be covered by the compilers themselves. These checks are static analysis checks that can be performed at a source code level in order to get a code in line with the C++11 standard. Cppcheck has been used with the following command in order to detect all the potential deviations from the standard:

```
cppcheck --enable=all --xml . 2> output.xml
```

In order to complete this study, the code has always been compiled with the GCC compiler v8.3.1 by activating all possible warnings by using the “-Wall” and “-Wextra” options.

By using Cppcheck and the GCC compilers, **14 warnings have been detected and fully corrected**. This strategy was followed throughout the development duration. The code can now be compiled with a wide range of compilers without showing any warnings, and it ensures a good portability of the code.

The tested compilers are the following:

- GCC v8.3.1
- Clang v10.0.1
- Intel Compiler v19.4

4.2 Dynamical analysis

Dynamic Program Analysis (DPA) is a form of program analysis that requires its execution. It aims to find problems in programs by detecting whether or not the program is accessing prohibited memory areas, or even reveal bugs in a program using fuzzers. It allows also for a program to be debugged in real time, giving the possibility of looking at what is happening in the memory and in the processor at any time during its execution. The DPA also aims to profile the code in order to give information about how resources (CPU, RAM & Cache memory, etc…) are being used during the program execution and to identify potential optimizations. **The dynamic analysis has been performed by using the test cases described in the previous section.**

4.2.1 Memory leak study

Valgrind [3] is a tool for memory debugging, memory leak detection and profiling. Valgrind contains several modules. The Memcheck module has been used for this study. Memcheck makes possible to flush out the leaks in a program at the memory level usage. Memcheck checks among other things:

- Variables and pointers are initialized,
- Allocated memory gets freed after use,
- No access is attempted to freed or unallocated memory areas,
- Memory zones are not freed twice,
- Valid arguments are passed to standard library functions like the memcpy() function.

By using Valgrind, **no memory leak has been detected**, that means all the memory is released after its usage.

4.2.2 Profiling

ARM forge [4] is a graphical performance analysis and parallel debugger tool for parallel applications on CPU (OpenMP, MPI), GPU (CUDA C/C++/Fortran, OpenACC) and hybrid applications (MPI+CUDA/OpenACC). Its usage on the test cases doesn't emphasize threads bottlenecks in the CUDA implementations. Data are kept on the GPU, except for IO operations. ARM forge reports a huge fraction of runtime being spend on device synchronization. However, this is arises from the limited size of the tests cases and should not be considered as a limitation because the multi-GPU implementation requires asynchronism to provide a speed-up.

5 Multi-GPU implementations

The multi-GPU implementation is based on asynchronous kernel launches on each available device to reduce latencies. The grid is split on each GPU by using ghost cells inside each sub domain and asynchronous communications between GPUs.

5.1 Benchmarks

3 cases were considered, namely "case_10_10", "case_20_10" and "case_30_10" and their memory footprint are reported in *Table 5.1*:

	case 10 10	case 20 10	case 30 10
memory usage [MiB]	2997	3569	11277

Table 5.1: Memory footprint for the 3 benchmark cases.

Computing times for the 3 cases, which are performed on nodes with 4 GPUs of 16 GB, are reported in *Table 5.2* and *Figure 5.1*, 20000 time steps were considered. It shows the computing times for the multi-GPU implementation (reported as 1, 2, 3 and 4 GPU), the original implementation on GPU (reported as mono-GPU) and the CPU implementation simulated on a full scalar node with 1 process and 40 threads.

elapsed time [s]	CPU	monoGPU	1 GPU	2 GPU	3 GPU	4 GPU
case 10 10	1345,99	66,14	66,38	41,91	34,84	32,91
case 20 10	1653,39	80,82	80,90	49,37	39,75	36,39
case 30 10	5597,96	264,79	264,07	143,50	104,73	85,42

Table 5.2: Elapsed time for the 3 benchmarks cases using 20000-time steps.

speed up	1 GPU	2 GPU	3 GPU	4 GPU
case 10 10	1	1,58	1,91	2,02
case 20 10	1	1,64	2,03	2,22
case 30 10	1	1,84	2,52	3,09

Table 5.3: Speed up for the 3 benchmarks cases using 1 GPU as reference.

parallel efficiency	1 GPU	2 GPU	3 GPU	4 GPU
case 10 10	100%	79,20%	63,51%	50,42%
case 20 10	100%	81,93%	67,83%	55,57%
case 30 10	100%	92,01%	84,05%	77,29%

Table 5.4: Parallel efficiencies for the 3 benchmarks cases using 1 GPU as reference.

The evaluation of speed up and parallel efficiency (*Figure 5.2* and *Figure 5.3*) make use of the computing time of 1 GPU as reference. Communication overhead is not present in the reference case; therefore, limiting this overhead will be of primary importance to achieve a good speed up and parallel efficiency.

The loss in parallel efficiency as the number of GPU increases arises from the fact that communications between GPUs gets larger as the number of sub domains that each GPU should resolve grows. The significant increase of parallel efficiency along with the memory footprint shows that system's size remains the limiting factor for the benchmarks under study. The parallel efficiency of the multi-GPU implementation considering the largest case remains between 77 to 92 %, with a speed up of 3 when running on 4 GPUs with an average memory allocation of 3159 MiB per GPU.

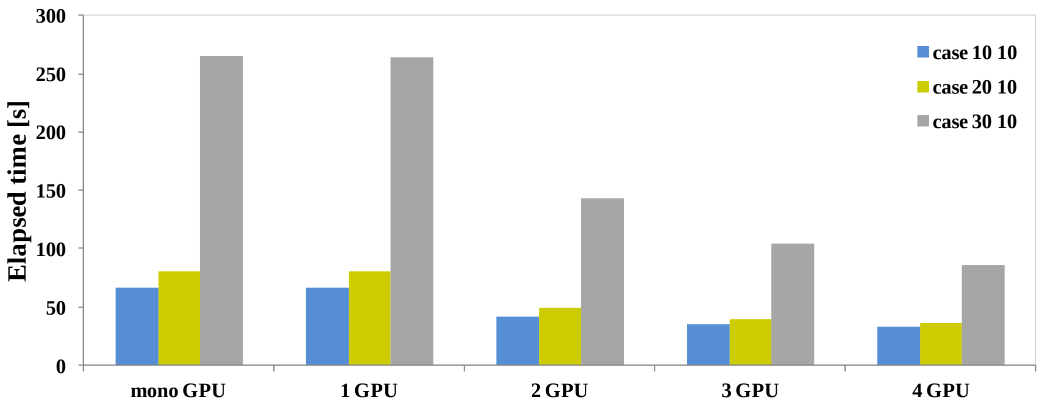

Figure 5.1: Evolution of the elapsed time with increasing numbers of GPU for the 3 benchmark cases.

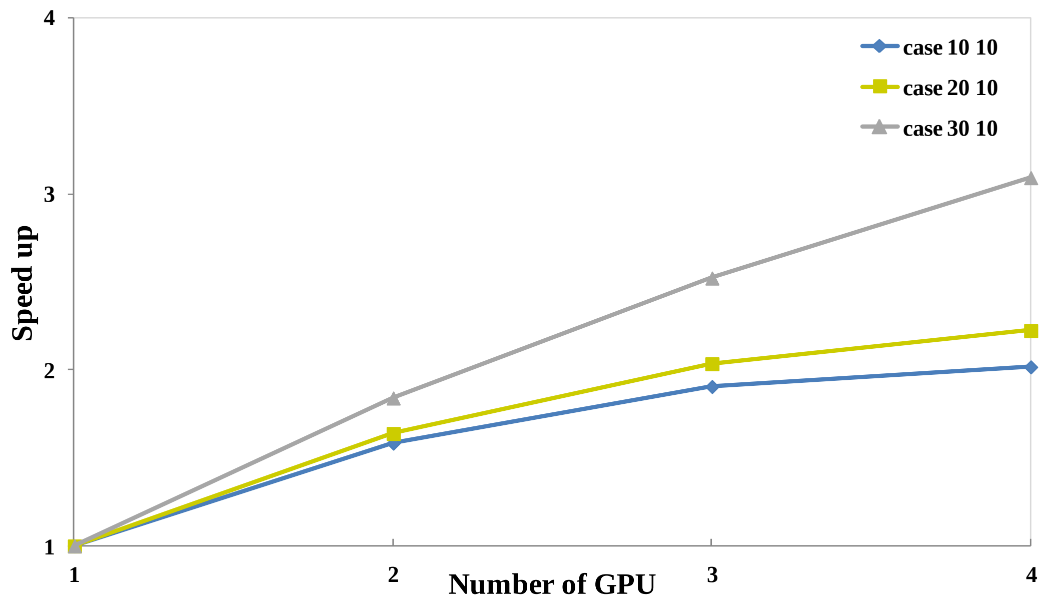

Figure 5.2: Speed up evolution with increasing number of GPU.

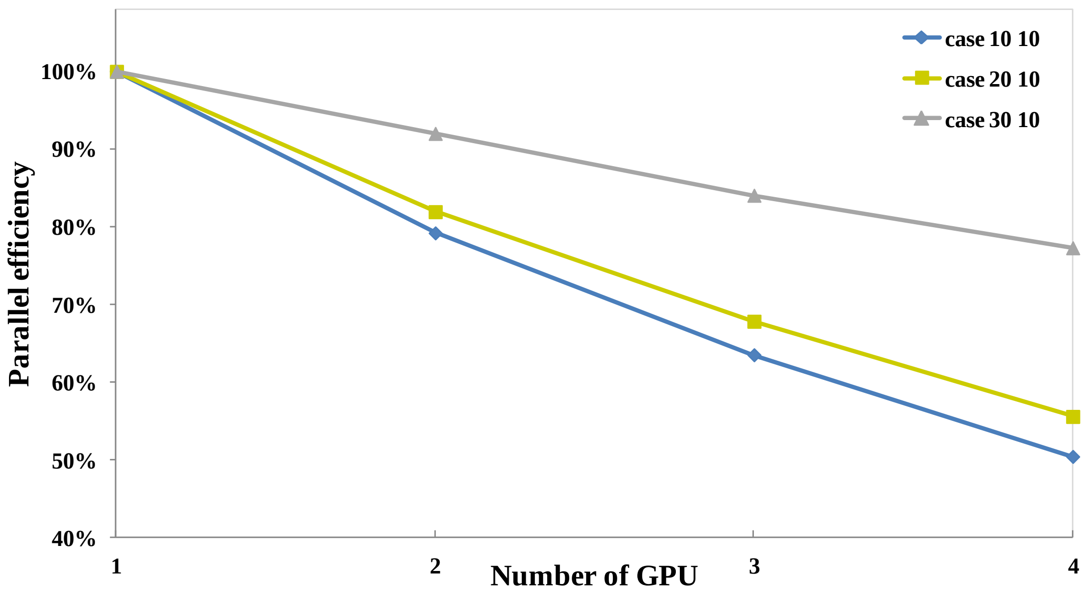

Figure 5.3: Parallel efficiencies of the 3 benchmark cases.

Improvements on speed up and parallel efficiency is observed for every system while increasing the size of system. This observation suggests that in order to determine the upper limit of both speedup and parallel efficiency larger benchmark cases must be considered. Assessing the weak scalability can also be a point of interest.

6 Benefit for the SME

The work performed in the framework of this project has confirmed the quality of the initial code provided by Open Engineering. The implementation was indeed based on modern C++ and CUDA with few warnings and no memory leaks. This code proved to be a good foundation for the multi-GPU development foreseen in this project.

In this context, IDRIS has indeed proposed and developed a new version of the EM FDTD solver adapted to multi-GPU servers. Communication between Idris and Open Engineering team was excellent during the project.

Performance evaluation of the work needs to be pursued. We have to remember that this project was performed during the COVID-19 pandemic. It has not been possible to provide large meshes by the end of the project and thus continue the scalability analysis further. Investigation of a MPI version has not been conducted either.

However, the Open Engineering RF solver is now compatible with modern HPC infrastructure. With the work done in this project, the code can now benefit from the speedup provided by several GPUs on a single compute node. That was the major goal of the project and it has been achieved. With the growing number of HPC infrastructure in Europe, many opportunities exist for Open Engineering to run its solver.

Opportunities exist to further develop the solver: the first and most obvious one consists in developing a MPI version of the RF solver to make it compatible with clusters and HPC infrastructures that are not equipped with GPUs.

7 Lessons learned

During the project, more resources from Open Engineering would have been needed to provide larger size problems or to include the modifications of the EM code into the complete version of its solver by the end of the project.
As already stated, Open Engineering is now in a better position to participate in other EU projects in the field of HPC or to access HPC resources in Europe.

References:

[1] http://cppcheck.sourceforge.net/
[2] https://github.com/danmar/cppcheck/
[3] http://valgrind.org/
[4] https://www.arm.com/products/development-tools/server-and-hpc/forge

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU’s Horizon 2020 Research and Innovation program (2014-2020) under grant agreement 823767.