The Journal of Supercomputing (2023) 79:7648–7664
https://doi.org/10.1007/s11227-022-04975-6

Strategies to parallelize a finite element mesh truncation technique on multi-core and many-core architectures

Jose M. Badia[1] · Adrian Amor-Martin[2] · Jose A. Belloch[3] · Luis Emilio Garcia-Castillo[3]

Accepted: 22 November 2022 / Published online: 2 December 2022
© The Author(s) 2022

Abstract

Achieving maximum parallel performance on multi-core CPUs and many-core GPUs is a challenging task depending on multiple factors. These include, for example, the number and granularity of the computations or the use of the memories of the devices. In this paper, we assess those factors by evaluating and comparing different parallelizations of the same problem on a multiprocessor containing a CPU with 40 cores and four P100 GPUs with Pascal architecture. We use, as study case, the convolutional operation behind a non-standard finite element mesh truncation technique in the context of open region electromagnetic wave propagation problems. A total of six parallel algorithms implemented using OpenMP and CUDA have been used to carry out the comparison by leveraging the same levels of parallelism on both types of platforms. Three of the algorithms are presented for the first time in this paper, including a multi-GPU method, and two others are improved versions of algorithms previously developed by some of the authors. This paper presents a thorough experimental evaluation of the parallel algorithms on a radar cross-sectional prediction problem. Results show that performance obtained on the GPU clearly overcomes those obtained in the CPU, much more so if we use multiple GPUs to distribute both data and computations. Accelerations close to 30 have been obtained on the CPU, while with the multi-GPU version accelerations larger than 250 have been achieved.

Keywords Parallel computing · CUDA · OpenMP · Finite elements · GPU

Adrian Amor-Martin, Jose A. Belloch and Luis Emilio Garcia-Castillo contributed equally to this work.

✉ Jose M. Badia
badia@uji.es

Extended author information available on the last page of the article

1 Introduction

Achieving maximum performance of an application is a complex and multifactorial problem that depends on the characteristics of the application as well as the architecture where it is executed. Decades of development of parallel algorithms applied to a huge number of very different applications (on diverse and continuously evolving parallel architectures) have led us to understand the main factors that determine the performance of parallel algorithms. These factors include, for example, the number of tasks that can be executed in parallel, their granularity and load balance, the need to synchronize them or the ratio between the cost of the calculations, the communications and the memory accesses.

On the other hand, most of the current fastest supercomputers are built from nodes containing several multi-core CPUs and one or several high-performance GPUs. Many computationally expensive problems can be tackled by appropriately leveraging both kinds of processing element or a combination of them. A good example of robust but computationally expensive tool is the finite element method (FEM), used for the numerical solution of partial differential equations in a wide variety of engineering disciplines and physics. When FEM is used to model electromagnetic wave propagation in open region domains, for example, in antenna analysis or in radar cross-sectional (RCS) prediction, the mesh generally needs to be truncated at some distance of the device/structure under analysis [1]. This kind of problems can also be found in other applications such as microwave tomographic imaging [2], geophysics [3], and nanotechnology [4]. Since a volumetric mesh is needed, the number of finite elements (and hence unknowns of the problem) increases rapidly with this truncation distance, exerting a severe impact on the computational resources required to solve the problem. On the other hand, the mesh truncation technique also affects the accuracy of the FEM solution [1].

Different approaches for mesh truncation in electromagnetic wave propagation have been proposed in the literature. One of the authors has proposed a non-standard mesh truncation technique called FE-IIEE (Finite Element-Iterative Integral Equation Evaluation), which was introduced in the context of hybridization with asymptotic high-frequency techniques (see [5–7]).

The FE-IIEE technique comprises convolutional type operations that can be implemented as three main nested loops and whose iterations are almost independent and involve massive computations on different data. This loop structure is also found in other widely used numerical methods (e.g., boundary element methods [8]) and applications related to electromagnetism such as, radiated/scattered field, and other fields. There are other fields where we can find the same structure: convolutional neural networks [9], face recognition [10], image processing [11], data spectroscopy [12], among others. These operations are a good example of many algorithms whose parallelism can be naturally exploited both on multi-core CPUs and on many-core GPUs [11, 13, 14].

This type of application offers us an excellent opportunity to explore what we can call the problem execution space. That is, the possibility of studying the effect of choosing a certain level of parallelism, associated with the granularity of the tasks to

be executed, and modifying the number of tasks to be carried out. In this paper, we will compare the effect of parallelizing each of the three nested loops of the application and choosing the number of iterations of each of them. Depending on the specific problem to solve or the architecture to be employed, some restrictions could be imposed regarding the size of some of those loops.

The main contribution of this work is the comparison of the behaviour of multi-core CPU and many-core GPU architectures in applications that offer the possibility of leveraging data parallelism with different levels of granularity. A mesh truncation algorithm that can be used as part of FEM has been used as a case study. A total of six parallel algorithms implemented using OpenMP for the CPU and CUDA for the GPU have been used to carry out the comparison leveraging the same levels of parallelism on both types of platform. Three of the algorithms are presented for the first time in this paper, and two others are new redesigned versions of algorithms previously developed by some of the authors [15, 16], whose performances are now significantly improved. To tackle the thorough experimental analysis, we used an Intel Xeon CPU with 40 cores and four Nvidia Tesla P100 GPUs with 3584 cores each. We have used different optimization techniques to improve the performance of the OpenMP and CUDA algorithms. For example, we have leveraged the registers and shared memory of the GPUs and we have modified the way to perform the reduction of partial results to deal with very large vectors in OpenMP.

The rest of the paper is structured as follows: The following section reviews the FE-IIEE methodology and formulation together with a description of the data structures to be considered. Section 3 describes the parallelization strategies that are carried out in both multi-core CPU and GPU. Experimental evaluation is devoted to Section 4. Finally, conclusions are drawn in Section 5.

2 FE-IIEE method

The FE-IIEE Method [5] is based on a decomposition of the infinite domain into two overlapping subdomains limited by S and S' respectively: one "interior" domain encloses the volume limited by S, while the other "exterior" domain takes the space from S' to infinity (see Fig. 1a illustrating the analysis of a radiation problem-antenna). FE-IIEE provides an (asymptotically exact) radiation boundary condition on S by iterative interaction between the solution of both domains. Thus, we need first to use the FEM formulation shown in [5] to solve for the electric field solution in the interior domain. For that purpose, a Cauchy boundary condition on the exterior boundary S is used. Therefore, the values of the field (and its curl) on S must be given. At the first iteration, these values are initialized to appropriate values (see [5] for further details). Next, the field solution in the exterior domain (and specifically on S) is obtained using an integral equation representation of the electromagnetic field based on the "equivalent currents" (basically, the FEM electric field solution and its curl evaluated on S') and the Green's function of the exterior domain. Actually, the kernel behind the representation of the integral equation of the field is of convolutional type (see [5] for further details). The field (and its curl) on S is used to update the boundary condition for the FEM interior problem and start a new

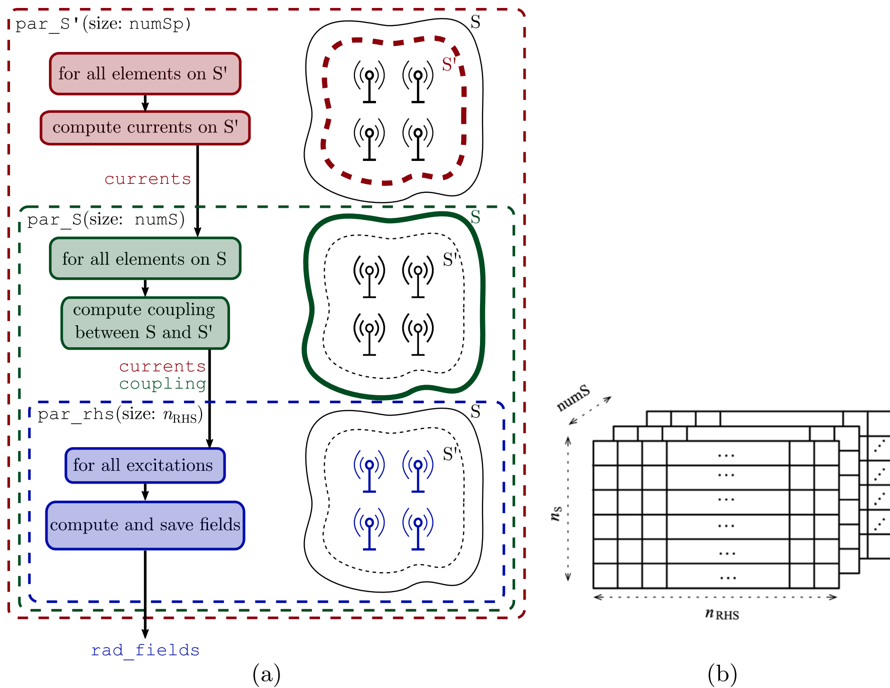


Fig. 1 **a** Flow diagram of the parallel code. **b** Logical structure and size of `rad_fields` computed by the algorithm

iteration cycle. The code devoted to the computation of the field solution on S due to the radiation of the equivalent currents on S' is precisely the target for parallelization in this work.

The process is repeated until a threshold error (or a maximum number of iterations) is achieved. The outcome of the method is that the truncation boundary S may be close to the radiating structure without losing accuracy. This establishes a trade-off between the size of the FEM domain and the number of iterations needed to achieve a given accuracy. An important advantage for the computer code implementation is that each iteration does not interfere with the workflow of the FEM code since it can be considered as a post-process of the final solution, which allows us to parallelize independently this part of the code.

2.1 Data structures and algorithm considerations

Figure 1a shows a flow diagram of the section of the code to be parallelized. It is illustrated with an antenna problem. The code consists of three nested loops. The outermost (red) loop, labelled as `par_S′` (where `par` will be `omp` or `cuda` depending on the framework used) covers all the elements located on S', where we need to compute equivalent electric and magnetic currents at some sample points (used in the numerical integration of a convolution-like operator). This leads to an

array `currents` of dimension $3 \times 2 \times n_{S'} \times n_{RHS}$, where $n_{S'}$ is the number of sample points in each element of S', and n_{RHS} is the number of inputs (or excitations, i.e. antenna elements, waveports, or planewaves falling from outside the domain, included in the problem as different right-hand sides) in the FEM problem. The value 3 corresponds to the number of components of each current, and the value 2 is associated with the fact that we need electric and magnetic currents (related to the electric field and its curl). The intermediate loop (green), labelled `par_S`, is associated with the outer surface S, and involves the computation of the coupling between S and S'. The variable `coupling` is a set of values that relate the sample points on S to the equivalent currents on each sample point on S' and include the evaluation of the Green's function on the corresponding integration points of S' and S. Finally, the innermost (blue) loop, `par_rhs`, covers the n_{RHS} inputs present in the problem. The output `rad_fields` of this loop uses the intermediate output variables `currents` and `coupling` and is the final output of the code to be parallelized. Note that a more detailed pseudo-code is available in our previous work [17].

The variable `rad_fields` is a huge array that contains the values of the three spatial components of the fields (the electric field and its curl) at each sample point on S for each of the n_{RHS} excitations of the problem. The array `rad_fields` is used to update the boundary condition for the FEM interior problem and start a new iteration cycle of FE-IIEE. Thus, the size of the matrix `rad_fields` is $3 \times 2 \times N_S \times n_{RHS}$. The value 3 is because of the three spatial components of the field (or its curl). The value 2 is because we have to store the field itself and its curl. The value N_S is the total number of sample points on the surface S (and not only in each element of S), i.e., $N_S = n_S \times$ `numS` being n_S the number of sample points in each finite element of S and `numS` the number of elements on S (see green loop of Fig. 1a). The number N_S is the main factor affecting the size of `rad_fields` and that will play an important role on the different parallelization strategies as it will be clear later.

For the sake of clarity, the structure of the matrix `rad_fields` is depicted in Fig. 1b. The first dimension corresponds to n_{RHS}, the second dimension to n_S, and the third dimension to `numS`. The elements are stored sequentially in memory, but we adopt a 3D representation because it shows the different dimensions of the problem that we can parallelize. Each small square in Fig. 1b represents 3×2 complex elements of the vector modified by one iteration of the innermost loop corresponding to a single right-hand side of the problem to solve. Every iteration of that loop involves dozens of products and additions of complex and real numbers that modify different elements of `rad_fields`, and that can be implemented as a kernel to run by different threads on a GPU platform. Note that the field contributions generated by the `currents` in each sample point of S' are accumulated into the corresponding small squares shown in the figure.

3 Strategies of parallelization

FE-IIEE method is very attractive from a computational point of view, because it offers many options of parallelization, since most of its operations can be performed independently. Those options can be associated with different loops of the algorithm. The loop we choose to parallelize determines the granularity of the computations performed by each thread and also the size of the data used by that thread.

If we choose to parallelize the loop for S or the loop for RHS, all the computations of each iteration can be performed fully in parallel. However, if we want to parallelize the loop for S', then each iteration can be performed in parallel by storing a local copy of the vector `rad_fields` for every thread. A final reduction must then be performed to add the contents of those copies.

We can take advantage of the different options of parallelization on different types of parallel platform. For example, if we have a multi-core CPU, we can use OpenMP to parallelize the loops. Alternatively, if we have a many-core GPU, we can use CUDA or OpenCL to implement kernels associated with the same loops.

3.1 OpenMP parallelization on CPU

We have implemented and evaluated three parallel versions of the FE-IIEE method using OpenMP. Each version parallelizes only one of the three main loops of the method.

- `omp_S'`: loop on the elements of the interior boundary.
- `omp_S`: loop on the elements of the exterior boundary.
- `omp_rhs`: loop on the right-hand sides.

The algorithm `omp_S'` parallelizes the outermost loop of the algorithm. A first approach to this parallelization was previously introduced by some of the authors in [15], but this version has been redesigned from scratch. The most challenging step of this algorithm is the reduction of the partial values of the vector `rad_fields` calculated by each thread at the end of the loop. We cannot use the OpenMP built-in `reduction` clause over all the elements of `rad_fields`, because we are dealing with very large complex vectors and this kind of vector reduction is limited by the size of the stack. Therefore, we implemented the reduction by parallelizing the addition of the elements of the vectors computed by each thread. That is, for each copy of vector computed by each OpenMP thread, we use a parallel loop to distribute the addition of its elements to the global reduced `rad_fields` vector. The following pseudo-code shows how this reduction is performed.

```
for th in all OpenMP threads
    #pragma omp parallel for
    for i in all elements of the rad_fields vector
       global_rad_fields[i] += local_rad_fields[th][i]
```

Parallelizing the loop on the right-hand sides to obtain the algorithm `omp_rhs` is straightforward, as all its iterations are fully independent. However, this parallelization can only be efficient if the problem involves thousands of right-hand sides (e.g., very large antenna arrays), which is not the usual case.

Alternatively, we can parallelize the loop on S, whose iterations are also fully independent. This idea can be leveraged even for only one right-hand side (e.g. a single antenna). This loop corresponds to one layer in Fig. 1b and can consist of thousands of iterations, allowing us to exploit multi-core CPUs and many-core GPUs.

We have tested different scheduling schemes for the three parallel algorithms and they offer very similar experimental results. As we are dealing in all cases with iterations with a well-balanced cost, we use in all the experiments showed in this paper the default static scheduling, where equal-sized blocks of consecutive iterations are computed by each thread.

3.2 CUDA parallelization on GPU

Efficient GPU programming requires taking into account the particular characteristics of this type of architectures. The most commonly used metric to assess GPU performance is occupancy, which is related to the degree of utilization of the cores. The architectural factors that limit the performance of the GPUs are related to the resources available on each Streaming Multiprocessor (SM). Specifically, the limiting factors include the registers and shared memory available, or the maximum number of warps or thread blocks that can be launched on each multiprocessor. Other important factors to have into account are the reduction of data transfers between CPU and GPU, the preferential access to the fastest available memories (registers and shared memory) and the possibility of coalesced access to data.

Specifically, the main limiting factor for the efficient implementation of the FE-IIEE algorithm on GPUs is the large amount of local data that needs to be handled on each thread. GPU cores are much simpler than CPU cores, and their maximum performance is usually obtained when running regular low-granularity computations that handle a limited amount of local data. However, in our case study, even the granularity of the computations carried out in each iteration of the innermost loop is very high. Each thread must perform a large number of floating point operations on a large amount of different data, the values of which need to be stored locally. Therefore, every thread requires a very large number of registers to perform its computations, and this is the factor that considerably limits the maximum number of threads that can be executed in parallel and take advantage of the thousands of GPU cores. This problem is exacerbated when we parallelize the intermediate loop on S, which implies even higher granularity computations and more local data.

Efficient parallelization of each of the loops of the FE-IIEE algorithm poses specific challenges. For example, the computation granularity of iterations of the outermost loop on S' is too large and the data size involved does not fit in the memory even of the most powerful GPUs. Thus, the CUDA parallelization will be only applied to the loops on S (`cuda_S`) and RHS (`cuda_rhs`). A preliminary straightforward GPU-based implementation of (`cuda_rhs`) that did not consider

optimization strategies associated with the GPU was proposed by some of the authors in [16].

The algorithm `cuda_S`, introduced in [17], is based on a kernel that implements all computations involved in each iteration of the loop on *S*. Therefore, we use a grid size equal to the number of iterations of the loop. The algorithm tries to optimize the management of the different kinds of GPU memory by leveraging the register file and the shared memory of the GPU. Specifically, we copy the elements of vector `currents` to the shared memory of each block of threads in order to reduce the number of accesses to global memory. Besides, we are not limited by the size of the problem, because on every iteration of the loop on RHS one of the threads of each block copies only the 6 complex elements (96 bytes) of vector `currents` used during the iteration. In fact, we are only using 0,19% of the 48 KB of static shared memory per block available on the Volta architecture. We have also tested a version of the algorithm in which we used dynamic shared memory to store all the elements of vector `currents` and to read them all at once at the beginning of the kernel. Its performance was very similar to the final proposed implementation.

We have also improved the implementation of `cuda_rhs`. First, we reduce the number of data transfers between the CPU and the GPU. To this end, we transfer only once the entire computed `rad_fields` from the GPU to the CPU at the end of the algorithm, rather than transferring successive blocks of the same vector after each iteration of the loop on *S*, corresponding to a panel in Fig. 1b. Obviously, this optimization implies increasing the use of global GPU memory. Secondly, we replicate some computations on each block of threads, instead of performing them in the CPU. This way, we avoid transferring all these vectors from CPU to GPU. Finally, we reduce the cost of accessing global memory by storing in the shared memory of the GPU some of the vectors used in each iteration of the RHS loop by all the threads.

3.3 Multi-GPU parallelization

We have also implemented a multi-GPU version of the algorithm to distribute some tasks on multiple GPUs. We partition the iterations of the loop on S and distribute them on several GPUs. Each GPU uses the kernel of the `cuda_S` algorithm to perform the computations associated with its iterations. We use OpenMP to launch in parallel on the CPU one thread associated with each GPU. This thread transfers the data to the corresponding GPU, launches the CUDA kernel, and finally receives the elements of `rad_fields` computed by it.

4 Experimental evaluation

4.1 Experimental environment

Performance tests were carried out in a server containing a multi-core CPU and four high-performance GPUs. Specifically, the CPU consists of two Intel Xeon

E5-2698V4 processors, at 2.20 GHz, each with 20 cores, and a total of 512 GB of DDR4-2400 main memory. Each of the four GPUs is a Tesla P100 SXM2 GPU powered by the NVIDIA Pascal architecture [18]. This GPU is composed of 56 Streaming Multiprocessor (SM) units with 64 CUDA cores per unit, i.e. 3,584 CUDA cores in total, a register file with 14,336 KB, an L2 cache with 4096 KB, and a global memory of 16 GB. The GPUs are connected to the CPU host using two PCIe Gen 3x16 switches.

The original FEM code was developed in Fortran 2003 [15]. Thus, a C wrapper was required to launch the CUDA kernels. The experiments were performed with Linux Ubuntu 19.04. The code was compiled with the 2019 version of the Intel Fortran and C compilers, and the CUDA version 10.1. We do not expect that the performance of the algorithms will change significantly if we use other versions of the compilers, such as the ones included in the GNU Compiler Collection. A customization layer over GiD is used to obtain FEM meshes and to plot the results [19].

In order to leverage the persistence of the GPU memory among successive calls from Fortran to the C kernel, we use static memory to allocate the vectors to store in the global GPU memory. The allocation is performed previously to the first iteration of the outermost loop, deallocating the memory at the end of that loop. We update the same allocated vector `rad_fields` among successive calls and only retrieve the final result from the GPU to the CPU after the last iteration of the outermost loop.

4.2 Scattering problem

We have used as a benchmark in all our experiments the computation of the RCS of an F117 plane. Multiple RHS are mandatory in the case of computing monostatic RCS for a sweep of spherical angles, as each angle of incidence corresponds to one right-hand side. The bistatic RCS of an incident wave at $\theta = 0°$ is directed at the tip of the plane in horizontal polarization.

Five meshes, starting from 27,739 tetrahedra and introducing increasing levels of refinement, have been simulated. Cases with $n_{\text{RHS}} = 1, 10, 100$ and 1000 are considered in the evaluation.

Table 1 shows the time corresponding to the sequential execution on the CPU using meshes with increasing number of exterior boundary elements `numS`. The number of interior boundary elements `numSp` is always 1,272. We can observe that the sequential time grows linearly not only with `numS`, but also with the number of

Table 1 Sequential times in seconds for the RCS problem

`numS`	n_{RHS}			
	1	10	100	1000
1572	7.28	44.66	416.68	4150.27
3432	15.78	97.30	905.87	9133.59
8012	36.99	228.90	2135.05	22,124.53
14,696	67.92	422.14	3890.59	41,198.88
32,576	149.89	936.58	8906.99	91,669.12

RHS. Note that we use these values as reference since the algorithms are the same for both OpenMP and GPU architectures. Regarding computational complexity, Table 2 shows the amount of floating point operations associated with different parts of the code: first, the loop `par_rhs`; second, the code executed from the start of the loop `par_S` to the start of the loop `par_rhs`; and, finally, the operations realized from the start of the loop `par_S′` to the start of the loop `par_S`. These parts of the code are indicated in Table 2 as (1), (2), and (3), respectively. We skip two intermediate values of `numS` for space purposes and to show the trend. We used a value of 4 and 8 flops for the square root and the exponential of real numbers, respectively. As expected, (1) grows linearly with n_{RHS}, whereas (2) remains the same for the different n_{RHS}, and grows linearly with `numS`. Regarding (3), its complexity changes with n_{RHS}, since we calculate the variable `currents`, which is used in (1) and depends linearly on n_{RHS}. Note that the computational load of (1) is much higher from $n_{\mathrm{RHS}} = 10$, whereas the load in (3) is small compared to (1) and (2) even for large n_{RHS}.

4.3 OpenMP results on CPU

We have compared the different parallel versions of the OpenMP algorithm evaluating the effect of modifying the number of iterations of the parallelized loops and also the number of CPU cores. Recall that we are running one OpenMP thread on each CPU core. Obviously, increasing the number of iterations of one of the loops increases the parallelism that can be leveraged by the algorithm that parallelizes it and also increases the granularity of the computations of the outer loops, if any.

Table 2 Number of floating point operations (in millions) for the RCS problem

	n_{RHS}	`numS`		
		1572	8012	32,576
(1)	1	1.12	5.70	23.16
(2)		4.38	22.32	90.79
(3)		0.08	0.08	0.08
(1)	10	11.18	56.97	231.62
(2)		4.38	22.32	90.79
(3)		0.33	0.33	0.33
(1)	100	111.77	569.65	2316.15
(2)		4.38	22.32	90.79
(3)		2.89	2.89	2.89
(1)	1000	1117.69	5696.53	23,161.54
(2)		4.38	22.32	90.79
(3)		28.4	28.4	28.4

(1) refers to the operations within the innermost loop, (2) is the number of operations between the intermediate loop and the innermost loop, and (3) covers the operations between the outermost loop and the intermediate loop

However, this growth of iterations also increases the cost of the algorithms both in terms of time and space. In order to compare the performance of the parallel algorithms running on the CPU an GPU we use as baseline the execution time of the sequential algorithm on one core of the CPU. We compute the acceleration of the different parallel algorithms with respect to this baseline. For OpenMP results, note that this *acceleration* is equivalent to the so-called *speedup*.

For example, Fig. 2a shows how increasing the number of elements in the interior boundary S' improves the performance of algorithm `omp_S'`. If we use less than 8 cores, we get almost optimal accelerations with all the problem sizes, but as we increase the number of cores the acceleration gradually moves away from optimum for the smallest problem sizes. Therefore, in the rest of the experiments, when comparing the parallel algorithms, we always run the `omp_S'` algorithm with 1,272 elements on the interior boundary. We carried out the same analysis for the algorithm `omp_S` by increasing the number of elements on the exterior boundary, obtaining a similar behaviour.

Figure 2b compares the algorithms `omp_S'` and `omp_S` with one and ten right-hand sides and using all 40 CPU cores. We can see that if our problem involves only one right-hand side, it is better to parallelize the outermost loop `omp_S'` as every iteration has a larger computational granularity. This effect is specially noticeable with the smallest problem, where `omp_S` gets an acceleration smaller than 20. If we increase the number of right-hand sides, the performance of both parallel algorithms tends to be equal, especially when we increase the number of elements in the exterior boundary.

Regarding the algorithm `omp_rhs`, we need to use more than 1000 right-hand sides to obtain some acceleration. Figure 3a shows that even in this case we get acceleration much smaller with this parallel algorithm that with the two algorithms that parallelize the outermost loops. Our experiments show that to get acceleration close to 20 using algorithm `omp_rhs` we need to have more than 6000 right-hand sides. This behaviour is mainly due to the relatively small granularity of the

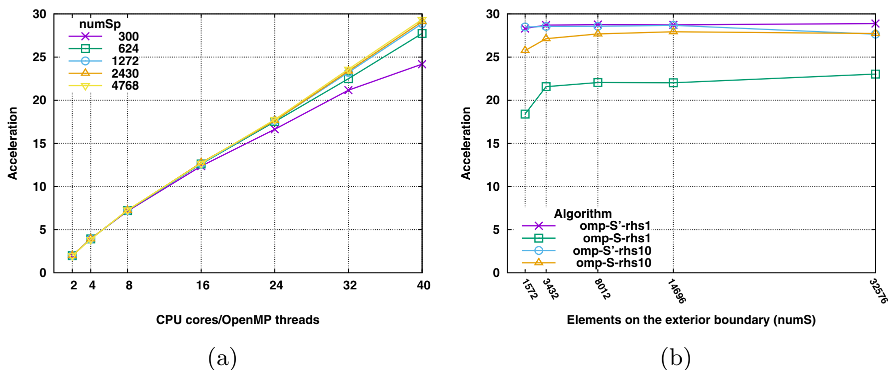


Fig. 2 **a** Acceleration of the `omp_S'` algorithm with different numbers of elements on the interior boundary. Results with one right-hand side ($n_{\text{RHS}} = 1$) and `numS` = 3424. **b** Acceleration of the OpenMP algorithms parallelizing the loops on S and S'. Results with one and ten right-hand sides ($n_{\text{RHS}} = 1$ and $n_{\text{RHS}} = 10$)

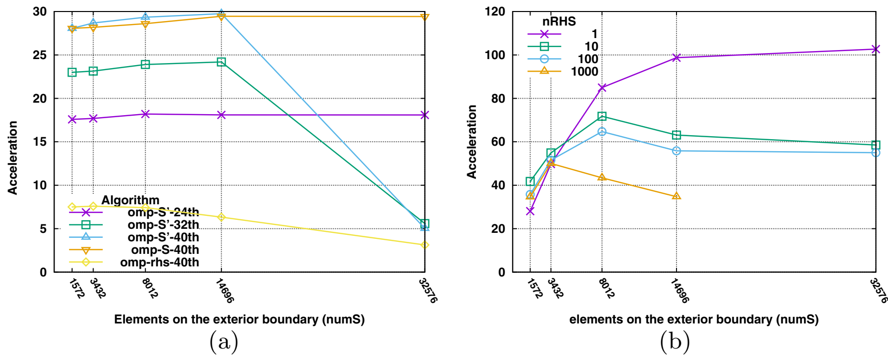


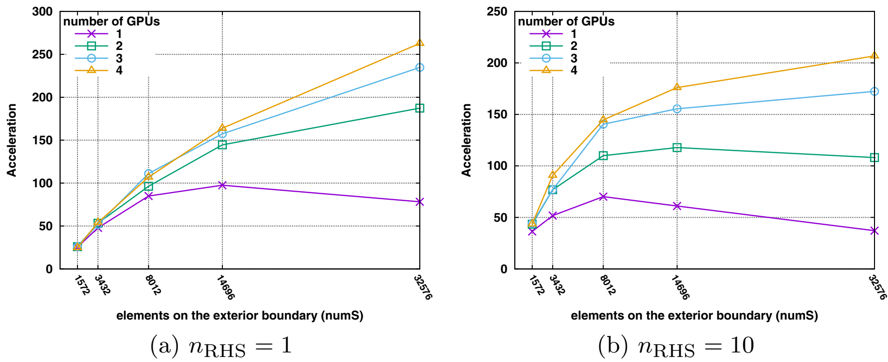
Fig. 3 **a** Acceleration of the OpenMP algorithms parallelizing each of the three main loops. Results with 1000 right-hand sides ($n_{\mathrm{RHS}} = 1000$). **b** Acceleration of algorithm `cuda_S` for different quantities of right-hand sides (right)

computations performed on each iteration of the loop on `omp_rhs` and the fact that we have to create the OpenMP threads on each of the hundreds of thousands or even millions of loop executions during the whole truncation process. Figure 3a also shows that if we use 32 or 40 cores, the acceleration of the algorithm `omp_S'` dramatically drops with the largest problem, which does not happen if we use 24 cores or less. This behaviour highlights the main drawback of this parallel algorithm: its spatial cost. As we have to use a copy of `rad_fields` for each OpenMP thread, the spatial cost of the algorithm grows linearly with the number of threads and we cannot use it with problems involving a large number of right-hand sides and elements on the exterior boundary. Specifically, the total size of 32 or more copies of `rad_fields` is larger than the 512 GiB or RAM memory of the CPU, which involves the use of the much slower secondary memory to store them.

4.4 CUDA results on GPU

We evaluated the behaviour of both CUDA parallel algorithms with the same problems solved with the OpenMP algorithms. Figure 3b shows the performance of the algorithm `cuda_S` when we modify the number of elements on the exterior boundary S and use different numbers of the right-hand sides. First, we can see that the acceleration grows more slowly or even decreases when we increase `numS`. We can also see that we get smaller accelerations as we increase the number of right-hand sides. As we increase both parameters, the spatial cost of the algorithm also increases, and we exhaust the resources available in the GPU. As we pointed out in [17], the main factor that limits the performance of this parallel algorithm is the number of registers available in each streaming multiprocessor. The computations involved by each iteration of the loop on S involve the use of a very large number of small vectors and scalar variables local to every CUDA thread, which use up even the large number of registers available on most modern GPUs.

Our experiments show that one way to mitigate in some cases the negative effect of increasing the size of the problem is to decrease the thread block size. For

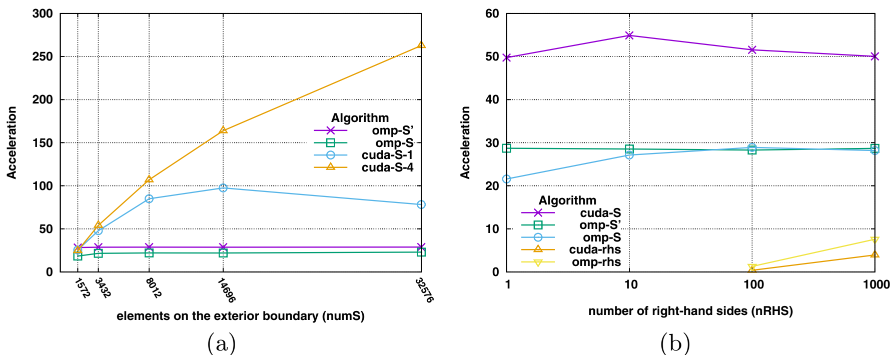


(a) $n_{\mathrm{RHS}} = 1$ (b) $n_{\mathrm{RHS}} = 10$

Fig. 4 Acceleration of the multi-GPU `cuda_S` algorithm

example, instead of using a number of threads multiple to the size of the warp, 32, we use 16 or even 8 threads per block.

Regarding algorithm `cuda_rhs`, Fig. 5b shows that parallelizing the innermost loop gets much smaller accelerations that using algorithm `cuda_S`. The poor performance obtained with 100 or even 1000 right-hand sides is mainly due to the small occupancy of the GPU. We do not use the thousands of CUDA cores of the GPU unless we have several thousands of right-hand sides. In addition, as with algorithm `omp_rhs`, the relatively small granularity of the computations of each iteration affects the performance negatively. Even more when we have to transfer some data from the CPU to the GPU just before starting the loop and also replicate some computations on every block of threads. Therefore, algorithm `cuda_rhs` gets even worse performances than algorithm `omp_rhs` if we do not solve problems with more than 6000 right-hand sides.



(a) (b)

Fig. 5 **a** Comparison of the acceleration of the OpenMP algorithms parallelizing the two outermost loops and `cuda_S` algorithm using 1 and 4 GPUs. Results with 1 right-hand side ($n_{\mathrm{RHS}} = 1$). **b** Comparison of the acceleration of the OpenMP and CUDA algorithms parallelizing the different loops. Results with $\mathtt{numSp} = 1,272$ and $\mathtt{numS} = 3423$

4.5 Multi-GPU algorithm results

A straightforward method to increase the performance of CUDA algorithms is to distribute the computations among several GPUs. This way we will not only multiply the number of cores performing the same computations, but also increase the memory available to store the information and solve some of the problems shown in previous section.

Figure 4a shows that using only one GPU the acceleration decreases with the largest problem. However, if we distribute the data among several GPUs, the acceleration increases even with the largest problem. In the best case, using 4 GPUs, we get accelerations larger than 250 over the sequential version of the algorithm running on one core of the CPU.

As expected, if we solve problems involving more than one right-hand side, thus increasing the spatial cost, we get worse accelerations (see the right-hand side of Fig. 4). However, even in this case the multi-GPU version of the algorithm clearly improves the results obtained with only one GPU in all cases. As a matter of fact, in this case, we are obtaining a larger benefit from distributing the data on the memories of several GPUs. This distribution allows us to obtain accelerations larger than the number of GPUs. For example, solving the largest problem with 4 GPUs is 5.57 times faster than solving it with one GPU. This behaviour could be explained by a more efficient usage of the memory when every GPU has to deal with smaller vectors.

4.6 CUDA versus OpenMP parallel algorithms

We finish our experimental evaluation by comparing the OpenMP and CUDA parallel algorithms. Figure 5a shows that we obtain much larger accelerations using algorithm `cuda_S` to solve the scattering problem with one right-hand side on the GPU, than using all 40 cores of the CPU to run algorithms `omp_S′` or `omp_S`. The advantage of the CUDA algorithm is even much larger when we use 4 GPUs to run it. The performances obtained on the CPU and the GPU are only equal with the smallest problem, when we cannot leverage the many-core architecture of the GPU. However, as we increase the number of elements on the exterior boundary, the CUDA algorithm clearly overcomes both OpenMP algorithms.

Table 3 shows the absolute maximum performance in GFLOPS of the parallel algorithms with $n_{\mathrm{RHS}} = 1$. In all cases, this performance is obtained with the maximum value of `numS` used in the experiments, that is, with $\mathtt{numS} = 32,576$. While using the 40 cores of the CPU, we obtain almost 28 GFLOPS, using one GPU we obtain almost 100 GFLOPS and by leveraging the four GPUs we obtain more than 254 GFLOPS. Moreover, Fig. 4a shows that we could obtain even higher

Table 3 Maximum absolute performance of the parallel algorithms with $n_{\mathrm{RHS}} = 1$

	`omp_S′`	`omp_S`	`cuda_S`	multi-GPU
GFLOPS	27.94	22.28	99.35	254.30

performance with the multi-GPU version with larger problem sizes, that is, by increasing numS.

If we increase the number of right-hand sides, the algorithm cuda_S also performs better, as can be seen in Fig. 5b, even when solving one of the smallest problems. We can also see that the algorithms that parallelize the innermost loop on RHS get the worst performance, no matter whether we are using OpenMP on the CPU or CUDA on the GPU.

5 Conclusion

This work shows that the same levels of parallelism can be leveraged in many applications, both on multi-core CPUs and many-core GPUs, by efficiently employing the resources provided by each kind of processing device. Specifically, we have used OpenMP and CUDA to parallelize the three main nested loops of a FEM mesh truncation technique whose code structure (convolutional type) is reproduced also in other widely used methods and applications. When choosing the loop to parallelize, it is necessary to have into account the number of iterations that can be run in parallel, but also the granularity of the computations involved by each iteration, if they are balanced and also the data used by each thread. For example, the granularity of the outermost loop is too large to be tackled as a kernel running on a GPU core.

On the one hand, OpenMP provides a very flexible framework that allows the parallelization of the three loops of the method. On the other hand, CUDA allows only an efficient parallelization of the two innermost loops and leveraging the memory and thousands of cores of GPUs is more challenging.

However, experimental results show that performances obtained on the GPU clearly overcome those obtained on the CPU. The fastest version of the algorithm leverages the massive parallelism provided by modern GPUs. Besides, by distributing the iterations of the middle loop among several GPUs, we can overcome the memory limitations and get accelerations larger than 250 with respect to the sequential version of the algorithm.

Regarding the OpenMP versions of the algorithm, we obtain accelerations close to 30 by parallelizing the two outermost loops if they involve enough iterations. However, if we significantly increase the number of cores and right-hand sides, the additional spatial cost of copying the vector rad_fields on every thread is very detrimental to the algorithm that parallelizes the outermost loop (omp_S′). Finally, the parallelization of the innermost loop has the worst performance in both the GPU and the CPU. It only scales in the GPU if the application involves thousands of right-hand sides, which is quite uncommon.

Author contributions All authors contributed equally to this work.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work has been supported by the Spanish Government PID2020-113656RB-C21, PID2019-106455GB-C21 and by the Valencian Regional Government through PROMETEO/2019/109, as well as the Regional Government of Madrid throughout the project MIMACUHSPACE-CM-UC3M.

Data availability No additional data or materials available.

Declarations

Conflict of interest The authors declare that they have no known conflict of interest or personal relationships that could have appeared to influence the work reported in this paper.

Ethical approval Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Jin JM, Ryley DJ (2009) Finite element analysis of antennas and arrays. Wiley, Hoboken
2. Tournier P-H, Aliferis I, Bonazzoli M, De Buhan M, Darbas M, Dolean V, Hecht F, Jolivet P, El Kanfoud I, Migliaccio C et al (2019) Microwave tomographic imaging of cerebrovascular accidents by using high-performance computing. Parallel Comput 85:88–97
3. Um ES, Kim J, Wilt MJ, Commer M, Kim S-S (2019) Finite-element analysis of top-casing electric source method for imaging hydraulically active fracture zones. Geophysics 84(1):23–35
4. Musa SM (2012) Computational finite element methods in nanotechnology. CRC Press, Boca Raton
5. García-Castillo LE, Gómez-Revuelto I, Sáez de Adana F, Salazar-Palma M (2005) A finite element method for the analysis of radiation and scattering of electromagnetic waves on complex environments. Comput Methods Appl Mech Eng 194/2–5:637–655
6. Gómez-Revuelto I, García-Castillo LE, Salazar-Palma M, Sarkar TK (2005) Fully coupled hybrid-method FEM/high-frequency technique for the analysis of 3D scattering and radiation problems. Microw Opt Technol Lett 47(2):104–107
7. Fernández-Recio R, García-Castillo LE, Gómez-Revuelto I, Salazar-Palma M (2008) Fully coupled hybrid FEM-UTD method using NURBS for the analysis of radiation problems. IEEE Trans Antennas Propag 56(3):774–783
8. Silvester PP, Hsieh MS (1971) Finite-element solution of 2-dimensional exterior-field problems. IEE Proc (Microw Antennas Propag) 118(12):1743–1747
9. Mairal J, Koniusz P, Harchaoui Z, Schmid C (2014) Convolutional kernel networks. arXiv preprint arXiv:1406.3332
10. Sharma S, Soni A, Malviya V (2019) Face recognition based on convolution neural network (CNN) applications in image processing: a survey. In: Proceedings of Recent Advances in Interdisciplinary Trends in Engineering & Applications (RAITEA)
11. Saidani T, Lacassagne L, Falcou J, Tadonki C, Bouaziz S (2011) Parallelization schemes for memory optimization on the cell processor: a case study on the Harris corner detector. In: Transactions on high-performance embedded architectures and compilers III. Springer, Berlin, pp 177–200
12. Shi T, Belkin M, Yu B (2009) Data spectroscopy: eigenspaces of convolution operators and clustering. Ann Stat 3960–3984
13. Li X, Zhang G, Huang HH, Wang Z, Zheng W (2016) Performance analysis of GPU-based convolutional neural networks. In: 2016 45th International Conference on Parallel Processing (ICPP). IEEE, pp 67–76
14. Mittal S et al (2021) A survey of accelerator architectures for 3D convolution neural networks. J Syst Archit 115:102041

15. Garcia-Donoro D, Amor-Martin A, Garcia-Castillo LE (2017) Higher-order finite element electromagnetics code for HPC environments. In: International Conference on Computational Science (ICCS), Zurich, Switzerland, pp 819–827
16. Belloch JA, Amor-Martin A, Garcia-Donoro D, Martinez-Zaldivar FS, Garcia-Castillo LE (2019) On the use of many-core machines for the acceleration of a mesh truncation technique for FEM. J Supercomput 75(3):1686–1696. https://doi.org/10.1007/s11227-018-02739-9
17. Badía JM, Amor-Martin A, Belloch JA, García-Castillo LE (2020) GPU acceleration of a non-standard finite element mesh truncation technique for electromagnetics. IEEE Access 8:94719–94730. https://doi.org/10.1109/ACCESS.2020.2993103
18. Nvidia (2016) NVIDIA Tesla P100 Whitepaper. WP-08019-001_v01.1
19. Melendo A, Coll A, Pasenau M, Escolano E, Monros A (2021) GiD software. www.gidhome.com. Accessed Oct 2022

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Jose M. Badia[1] · Adrian Amor-Martin[2] · Jose A. Belloch[3] · Luis Emilio Garcia-Castillo[3]

Adrian Amor-Martin
aamor@ing.uc3m.es

Jose A. Belloch
jbelloc@ing.uc3m.es

Luis Emilio Garcia-Castillo
legcasti@ing.uc3m.es

[1] Depto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I de Castellón, Avda. Sos Baynat, s/n, 12071 Castellón de la Plana, Castellón, Spain

[2] Depto. de Teoría de la Señal y Comunicaciones, Universidad Carlos III de Madrid, Avda. de la Universidad, 30, 28911 Leganés, Madrid, Spain

[3] Depto. de Tecnología Electrónica, Universidad Carlos III de Madrid, Avda. de la Universidad, 30, 28911 Leganés, Madrid, Spain