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

Jose M. Badia (✉ badia@uji.es)  
Universitat Jaume I

Adrian Amor-Martin  
Universidad Carlos III

Jose A. Belloch  
Universidad Carlos III

Luis Emilio García-Castillo  
Universidad Carlos III

Research Article

Keywords: Parallel Computing, CUDA, OpenMP, Finite Elements, GPU

Posted Date: September 14th, 2022

DOI: https://doi.org/10.21203/rs.3.rs-2048751/v1

License: ☇️ 🛡️ This work is licensed under a Creative Commons Attribution 4.0 International License.  
Read Full License
Strategies to parallelize a finite element mesh truncation technique on multi- and manycore architectures

José M. Badía · Adrian Amor-Martin · Jose A. Belloch · Luis Emilio García-Castillo

Received: date / Accepted: date

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 Section 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 acceleration larger than 250 have been achieved.

Keywords Parallel Computing, CUDA, OpenMP, Finite Elements, GPU.

José M. Badía
Depto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I de Castellón, Spain,
E-mail: badia@uji.es, Tel: +34 964 72 82 95, Fax: +34 964 72 84 86

Adrian Amor-Martin
Depto. de Teoría de la Señal y Comunicaciones, Universidad Carlos III de Madrid, Spain.
E-mail: aamor@ing.uc3m.es

Jose A. Belloch, Luis Emilio García-Castillo
Depto. de Tecnología Electrónica, Universidad Carlos III de Madrid, Spain
E-mail: {jbelloc,legcasti}@ing.uc3m.es
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 synchronise 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 Section (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 hybridisation 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. 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, [9–11].

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.

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 [12,13], 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 optimisation 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 III describes the parallelization strategies that are carried out in both multi-core CPU and GPU. Experimental evaluation is devoted to Section IV. Finally, conclusions are drawn in Section V.

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 left-hand side of Figure 1 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 initialised 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 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

Left-hand side of Figure 1 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 to 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.

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
Fig. 1 Flow diagram of the parallel code (left) and logical structure and size of \texttt{rad\_fields} computed by the algorithm (right).

problem and start a new iteration cycle of FE-IIEE. Thus, the size of the matrix \texttt{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., $N_S = n_S \times \text{numS}$ being $n_S$ the number of sample points in each finite element of $S$ and \text{numS} the number of elements on $S$ (see green loop of Figure 1). The number $N_S$ is the main factor affecting the size of \texttt{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 \texttt{rad\_fields} is depicted in the right-hand side of Figure 1. The first dimension corresponds to $n_{RHS}$, the second dimension to $n_S$, and the third dimension to \text{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 the right-hand side of Figure 1 represents $3 \times 2$ 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 \texttt{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 \( \text{rad fields} \) 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 manycore GPU, we can use CUDA or OpenCL to implement kernels associated to 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 one of the three main loops of the method.

- \( \text{omp}\_S' \): loop on the elements of the interior boundary.
- \( \text{omp}\_S \): loop on the elements of the exterior boundary.
- \( \text{omp}\_\text{rhs} \): loop on the right-hand sides.

The algorithm \( \text{omp}\_S' \) parallelizes the outermost loop of the algorithm. A first approach to this parallelization was previously introduced by some of the authors in [12], 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 \( \text{rad fields} \) calculated by each thread at the end of the loop. We cannot use the OpenMP built-in reduction clause over all the elements of \( \text{rad fields} \), 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 vector we use a parallel loop to distribute the addition of its elements to the global reduced \( \text{rad fields} \).
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 the right-hand side of Figure 1 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, 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 pre-
liminatory straightforward GPU-based implementation of (cuda$_{rhs}$) that did
not consider optimisation strategies associated to the GPU was proposed by
some of the authors in [13].

The algorithm cuda$_S$, introduced in [14], is based on a kernel that im-
plements all computations involved in each iteration of the loop on $S$. The
algorithm tries to optimise 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 glob-
al 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.

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
the right-hand side of Figure 1. Obviously, this optimisation 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 to 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 2.60 GHz, each with 20 cores, and a total of
512 GB of DDR4-2666 main memory. Each of the four GPUs is a Tesla P100 SXM2 GPU powered by the NVIDIA Pascal architecture, [15]. 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, [12]. 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, C compilers, and the CUDA version 10.1. A customisation layer over GiD is used to obtain FEM meshes and to plot the results, [16].

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 \texttt{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 $\theta = 0^\circ$ is directed at the tip of the plane in horizontal polarisation. 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 \texttt{numS}. The number of interior boundary elements \texttt{numSp} is always 1,272. We can observe that the sequential time grows linearly not only with \texttt{numS}, but also with the number of RHS. Note that we use these values as reference since the algorithms are the same for both OpenMP and GPU architectures.

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
Table 1 Sequential times in seconds for the RCS problem.

<table>
<thead>
<tr>
<th>numS</th>
<th>nRHS 1</th>
<th>nRHS 10</th>
<th>nRHS 100</th>
<th>nRHS 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1572</td>
<td>7.28</td>
<td>44.66</td>
<td>416.68</td>
<td>4150.27</td>
</tr>
<tr>
<td>3432</td>
<td>15.78</td>
<td>97.30</td>
<td>905.87</td>
<td>9133.59</td>
</tr>
<tr>
<td>8012</td>
<td>36.99</td>
<td>228.90</td>
<td>2135.05</td>
<td>22124.53</td>
</tr>
<tr>
<td>14696</td>
<td>67.92</td>
<td>422.14</td>
<td>3890.59</td>
<td>41198.88</td>
</tr>
<tr>
<td>32576</td>
<td>149.89</td>
<td>936.58</td>
<td>8906.99</td>
<td>91669.12</td>
</tr>
</tbody>
</table>

Fig. 2 Acceleration of the omp_S’ algorithm with different numbers of elements on the interior boundary. Results with one right-hand side (nRHS = 1) and numS = 3424 (left). Acceleration of the OpenMP algorithms parallelizing the loops on S and S’. Results with one and ten right-hand sides (nRHS = 1 and nRHS = 10) (right).

by the algorithm that parallelizes it and also increases the granularity of the computations of the outer loops, if any. 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 example, the left-hand side of Figure 2 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.

The right-hand side of Figure 2 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
Strategies to parallelize FEM

Fig. 3 Acceleration of the OpenMP algorithms parallelizing each of the three main loops. Results with 1,000 right-hand sides ($n_{\text{RHS}} = 1,000$) (left). Acceleration of algorithm cuda$_S$ for different quantities of right-hand sides (right).

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$_{\text{rhs}}$, we need to use more than 1000 right-hand sides to obtain some acceleration. Left-hand side of Figure 3 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$_{\text{rhs}}$ we need to have more than 6,000 right-hand sides. This behaviour is mainly due to the relatively small granularity of the computations performed on each iteration of the loop on omp$_{\text{rhs}}$ 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. Left-hand side of Figure 3 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$_{\text{fields}}$ 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.

4.4 CUDA Results on GPU

We evaluated the behaviour of both CUDA parallel algorithms with the same problems solved with the OpenMP algorithms. Right-hand side of Figure 3 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 num$_S$. 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 [14], 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 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 $\text{cuda}_{\text{rhs}}$, the right-hand side of Figure 5 shows that parallelizing the innermost loop gets much smaller accelerations that using algorithm $\text{cuda}_S$. The poor performance obtained with 100 or even 1,000 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 $\text{omp}_{\text{rhs}}$, 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 $\text{cuda}_{\text{rhs}}$ gets even worse performances than algorithm $\text{omp}_{\text{rhs}}$ if we do not solve problems with more than 6,000 right-hand sides.

4.5 Multi-GPU 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.

The left-hand side of Figure 4 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.
4.6 CUDA vs OpenMP parallel algorithms

We finish our experimental evaluation by comparing the OpenMP and CUDA parallel algorithms. The left-hand side of Figure 5 shows that we obtain much larger accelerations using algorithm \textit{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 \textit{omp-S}’ or \textit{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 manycore architecture of the GPU. However, as we increase the number of elements on the exterior boundary, the CUDA algorithm clearly overcomes both OpenMP algorithms.

If we increase the number of right-hand sides, the algorithm \textit{cuda-S} also performs better, as can be seen in the right-hand side of Figure 5, 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.
4.7 Conclusions

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 modern 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 similar accelerations close to 30 by parallelizing the two outermost loops if they involve an enough number of iterations. However, if we significantly increase the number of cores and right-hand sides, the additional spatial cost of duplicating the vector \texttt{rad.fields} on every thread is very detrimental to the algorithm that parallelizes the outermost loop (\texttt{omp.S'}). Finally, the parallelization of the innermost loop has the worst performance in both the GPU and the CPU. Scales only in the GPU if the application involves thousands of right-hand sides, which is quite uncommon.

Declarations

Ethical Approval

Not applicable.

Competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.
Authors’ contributions

All authors contributed equally to this work.

Funding

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.

Availability of data and materials

No additional data or materials available.

References


