This is a previous version of the article published in Concurrency and Computation: Practice and Experience. 2017, 29(4): e3880. doi:10.1002/cpe.3880

A Performance Analysis of a Mimetic Finite Difference Scheme for Acoustic Wave Propagation on GPU Platforms

B. Otero*[1], J. Francés[2 3], R. Rodríguez-Cruz[4], O. Rojas[5 6], F. Solano[7] and J. M. Guevara-Jordan[7]

[1] *Dpto. d'Arquitectura de Computadors, Universitat Politècnica de Cataluña, Barcelona, Spain*
[2] *Dpto. de Física, Ingeniería de Sistemes y Teoría de la Señal, E-03080, Alicante, Spain*
[3] *Instituto Universitario de Física aplicada a las Ciencias y las Tecnologías Universidad de Alicante, E-03080, Alicante, Spain*
[4] *Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona, Universitat Politècnica de Cataluña, Barcelona, Spain*
[5] *Escuela de Computación, Facultad de Ciencias, Universidad Central de Venezuela, Caracas, Venezuela*
[6] *Department of Computer Applications in Science and Engineering, Barcelona Supercomputing Center, Barcelona, Spain*
[7] *Escuela de Matemática, Facultad de Ciencias, Universidad Central de Venezuela, Caracas, Venezuela*

SUMMARY

Realistic applications of numerical modeling of acoustic wave dynamics usually demand high performance computing because of the large size of study domains and demanding accuracy requirements on simulation results. Forward modeling of seismic motion on a given subsurface geological structure is by itself a good example of such applications, and when used as a component of seismic inversion tools or as a guide for the design of seismic surveys, its computational cost increases enormously. In the case of finite difference methods (or any other volumen-discretization scheme), memory and computing demands rise with grid refinement which may be neccesary to reduce errors on numerical wave patterns and better capture target physical devices. In this work, we present several implementations of a mimetic finite difference method for the simulation of acoustic wave propagation on highly dense staggered grids. These implementations evolve as different optimization strategies are employed starting from appropriate setting of compilation flags, code vectorization by using SSE and AVX instructions, CPU parallelization by exploiting the OpenMP framework, to the final code parallelization on GPU platforms. We present and discuss the increasing processing speed-up of this mimetic scheme achieved by the gradual implementation and testing of all these performance optimizations. In terms of simulation times, the performance of our GPU parallel implementations is consistently better than the best CPU version.

Received ...

KEY WORDS: Acoustic waves; mimetic finite differences; SIMD extensions; GPU; CUDA programming

*Correspondence to: C/ Jordi Girona 1.3, Edf. C6-204, 08034, Spain. E-mail: botero@ac.upc.edu

1. INTRODUCTION

Modeling of compressional P wave propagation in seismic Earth models is probably the application of acoustic motion with the highest demand of high performance computing. Explicit finite differences (FD) are still the most widely used discretization technique in this field because of its simple formulation and implementation even on complex geological media (see [12] for a review of multiple formulations). FD acoustic solvers are also amenable for domain decomposition and code parallelization, and particularly, GPU-CUDA implementations have recently increased mainly on
3-D domains (for instance, [1, 16, 17]). However, these GPU-based methods are still being developed on 2-D domains given their potential use on computationally intensive applications for seismic imaging [2], inversion of source location [11], parametric studies on water immersed bodies [18], and also represent useful prototypes in anticipation to their generalization to 3-D as well [30].

Modern seismic FD methods perform the spatial discretization of the acoustic wave model on staggered grids (SG). On these grids, each wave field is computed at a separate nodal grid displaced by half of the grid spacing (in one or more directions) from the remaining individual grids. The advantage of such geometrical distribution of discrete values is that each scalar field is located at the center of those it depends upon, and numerical differentiation gains accuracy by halving the grid step [9, 19, 31]. Grid staggering has been also sucessfully applied for FD simulations of wave propagation in the case of more general elastic or viscoelastic media [10, 13, 32]. Regardless the material rheology, accuracy and computational efficiency of FD modeling of seismic motion depend on the implementation of free surfaces (FS) and absorbing boundary conditions (ABC). The former techniques are used to model the Earth-Atmosphere interface, and low dispersive and efficient modeling of surface waves have been achieved by using lateral FD discretization of null pressure (or traction free) boundary conditions. On planar FS, this type of implementation was employed by Kristek and collaborators by means of Taylor-based FD stencils [14], and then by Rojas and co-workers in the case of second- and fourth-order accurate mimetic formulas [22, 25]. The concept of mimetic FD discretization is breifly introduced in the next paragraph, and applied to the acoustic model in the following section. On the other hand, ABC are differential operators designed to avoid artificial wave reflections at the boundaries of a finite simulation domain. Clayton and Enquist [3] propose FD implementations of ABC based on paraxial approximations of the two-way wave equation that only model the outward-moving energy impinging domain boundaries. Later, Reynolds [20] constructs a family of splitting operators for acoustic and elastic media. The application of Reynolds' ABC proceed by compositions of first order differential operators that progressively increase their effectiveness and implementation complexity. Recently, Solano and co-authors in [27] implement a second-order FD method for the acoustic wave equation on 1-D and 2-D rectangular SG, where lateral mimetic discretization of FS and ABC result crucial for the global accuracy. This scheme incorporates the first order Reynolds' ABC operators that show enough efficiency on standard seismic tests.

The extense work of Samarsky [26] on constructing conservative FD has evolved into two modern methods on SG. The first one is based on two discrete support operators, a Divergence

D and a Gradient G, both providing second-order accuracy at interior grid points, but reduced to first order at domain boundaries. This method has been extensively applied to diffusion, electromagnetic, and viscoelastic problems even on non uniform meshes (see [15] for a general review, and also [10] for the latter application). The second family of conservative D and G, exhibits second-, fourth-, and
sixth-order accuracy along all grid locations including boundaries and were proposed by Castillo and collaborators in [5] and later reformulated in [4]. For a rigorous numerical treatment of Neumann and Robin boundary conditions, Castillo and Yasuda [7] introduce an operator B to approximate boundary fluxes of a given vector field. These new operators D, G, and B, are referred as mimetic because in combination to the numerical solution to a boundary value problem, satisfy a particular discrete version of Gauss' Divergence theorem. Applications of mimetic D and G to 2-D wave propagation reduce to works on cartesian SG by Rojas and co-authors focused on modeling shear ruptures [23, 24], and surface Rayleigh waves [25]. Recently, De La Puente in [8] employs mimetic D and G on 3-D curvilinear structured meshes to study seismic motion acounting for irregular topographic effects. To our knowledge, the full set of mimetic D, G, and B has been only applied to wave propagation problems by Solano and collaborators in [27] and [28].

This paper focuses on the progressive optimization and parallelization of the 1-D and 2-D mimetic Solano's methods on available multi-core (CPU) and many-core (GPU) architectures. The parallel CPU code version takes advantage of the Streaming SIMD Extensions (SSE) found in modern microprocessors to exploit built-in capabilities. Then, this code incorporates parallel strategies to use all available cores in a given CPU by means of OpenMP directives. This fine-tuned CPU version was later compared to a massively parallelized CUDA code that also implements the same mimetic methods on a GPU platform. Thus, we carry out a comparative performance analysis of both parallel versions and report speed up of global simulation times. These implementations correspond to the first parallel versions of Castillo type mimetic FD schemes for acoustic wave propagation.

The remainder of this paper is organized in the following way: Section 2 gives a brief review of the mimetic discretization framework and presents Solano's algorithms for 1-D and 2-D acoustic wave propagation, that we use in our parallel implementations. In section 3, we describe available architecture and chosen application sceneries, in addition to our CPU and GPU parallel code versions. Section 4 compares the speed up of the execution time spent by both parallel implementations. Finally, Section 5 summarizes our conclusions and points out new directions for future works.

2. MIMETIC METHODS FOR SOLVING THE ACOUSTIC WAVE EQUATION

This section presents an overview of the mimetic acoustic solvers proposed by Solano in [27] for 1-D and 2-D spatial domains. We make a special emphasys on the different FD stencils that conform both numerical schemes giving their crucial implications on the parallel implementations discussed in this paper. In [27], acoustic wave propagation is modeled by the second order differential equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla \cdot (\nabla u) + f, \tag{1}$$

where c is the wave speed, f is a source term, and the physical interpretation of the unknown scalar function u depends on the spatial dimension. Symbols $\nabla\cdot$ and ∇ represent the divergence and gradient operators, respectively, widely used in the field of Mathematical Physics. Particular solutions of (1) require appropriate initial and boundary conditions. In seismic applications, the former usually correspond to the homogeneous case $u = \frac{\partial u}{\partial t} = 0$, while the latter result a combination of FS and ABC conditions. In the 1-D case, Solano in [27] considers u as the particle displacement, and implements the following boundary conditions on the interval $x \in [0, 1]$

$$\frac{\partial u}{\partial x}(0,t) = f_1(t), \tag{2}$$

$$\left[\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x}\right](1,t) = f_2(t). \tag{3}$$

In the case that $f_1(t) = f_2(t) = 0$, above equations represent a FS condition at $x = 0$, and the first order Reynolds' ABC operator at $x = 1$. For 2-D domains, formulation in [27] identifies u with the acoustic pressure experienced by particles on the plate $(x, y) \in [0, 1] \times [0, 1]$. Boundary conditions at edges $x = 0, 1$, and $y = 0$, correspond to the extension of Reynolds's ABC to 2-D rectangular domains, and a FS condition is imposed at the top boundary $y = 1$. That is,

$$\left(\frac{\partial u}{\partial t} - c\frac{\partial u}{\partial x}\right)(0,y,t) = 0, \qquad \left(\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x}\right)(1,y,t) = 0 \tag{4}$$

$$\left(\frac{\partial u}{\partial t} - c\frac{\partial u}{\partial z}\right)(x,0,t) = 0, \qquad u(x,1,t) = 0. \tag{5}$$

2.1. Mimetic discretization

Generally speaking, the mimetic discretization of boundary value problems stated above proceeds on a SG by substituting the continuous differential operators gradient ∇, divergence $\nabla\cdot$, and the normal derivative $(\frac{\partial}{\partial \overline{n}})$ at boundaries by the discrete FD approximations G, D, and BG, proposed in [4] and [7]. The operator $(\frac{\partial}{\partial \overline{n}})$, also known as flux operator in diffusion applications, corresponds to any of the spatial derivates appearing on boundary conditions (2), (3), (4), and (5). In Appendix, we briefly present the matrix representation of 1-D second-order accurate G, D, and B operators that produce the whole set of FD stencils of Solano's mimetic acoustic schemes, in one and two dimensions. For the sake of completeness, we also transcribe these mimetic stencils on this Appendix, but refer the interested reader to original references [4] and [7] for construction details and properties of G, D, and B, as well as to Solano's work in [27] for the complete formulation and convergence analysis of acoustic solvers subject of this paper. An important property of these schemes is the use of explicit numerical integrations for time derivates present in the acoustic equation and boundary

conditions, and therefore computer implementations do not require linear solvers or costly matrix inversion strategies. In this section, we limit ourselves to illustrate the grid distribution of unknown u values on the 1-D and 2-D mimetic grids, each of which must be computed by an individual FD stencil from approximations for previous times. This whole set of different stencils is also shown in a programmer friendly fashion on grid illustrations.

Figure 1 depicts the 1-D mimetic grid on the interval $[0,1]$ with cells $[x_i, x_{i+1}]$ and nodes $x_i = ih$, $i = 0, \ldots, n$ for a constant step $h = \frac{1}{n}$. This grid becomes staggered after including the cell centers $x_{i+\frac{1}{2}} = \frac{x_i + x_{i+1}}{2}$. Discrete values of target function u are considered at cell centers $u_{i+\frac{1}{2}}$, in addition to both boundary values u_0 and u_n, all of which are collected in the computer vector $\vec{u}$. A standard FD solution of (1) on uniform grids would require the approximation of the laplacian $\frac{d^2u}{dx^2}$ at some interior points of this (or any other) grid. In the case of a mimetic scheme, numerical differentiation of u is computed by using the gradient operator $G\vec{u}$, that renders estimates to $\frac{du}{dx}$ at each grid node. Next, this vector is transformed by the divergence operator by computing $D(G\vec{u})$. Components of last vector are approximations to $\frac{d^2u}{dx^2}$ at all cell centers, and can be used by a standard FD method to update u values at those locations upon a numerical integration of (1). However, the mimetic discretization of the Laplacian operator at cell centers is actually given by the interior components of vector $(DG + BG)\vec{u}$. Thus, non zero interior rows of B (see Appendix) add the mimetic contribution to the standard Laplacian approximation GD at the two nearest cell centers to each grid boundary. This discretization process gives rise to three different Laplacian approximations at centers $x_{\frac{1}{2}}, x_{\frac{3}{2}}$, and $x_{i+\frac{1}{2}}$ for $i = 2, \ldots, n-3$, and depicted with distinct symbols by figure 1. Same symbols are used at right centers $x_{n-\frac{1}{2}}$ and $x_{n-\frac{3}{2}}$ as a way to remark that computing FD stencils at those locations are minor variations of their symmetrical counterparts on the left side. The remaining first and last rows of matrix operator $DG + BG$ represent the particular boundary discretizations of the spatial terms on conditions (2) and (3), and then are shown by a new symbol in this figure.

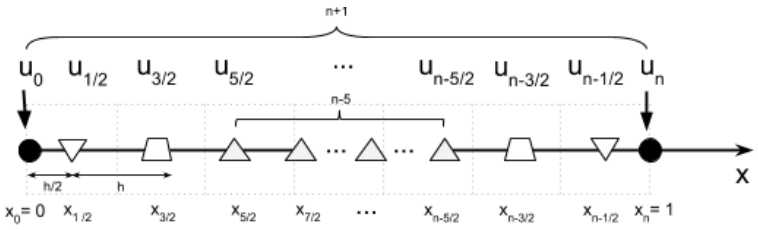

Figure 1. 1-D grid distribution of u values and mimetic FD stencils

The mimetic discretization grid for the acoustic model (1), (4) and (5), in the rectangular domain $[0,1]$x$[0,1]$ is shown in figure 2. Coarsely speaking, this 2-D grid can be thought of as the Cartesian product of two one-dimensional grids, each one defined along a particular axis x or y, and the final exclusion of corners $(0,0), (0,1), (1,0)$, and $(1,1)$. Classical mimetic FD discretization on multidimensional rectangular domains excludes corner points (see e.g., [6]), and Solano's acoustic method falls into this family. In figure 2 one can see that the target discrete values of u are defined at the center of all rectangular cells, and also at the middle of each boundary cell edge. Similar to the grid geometry, the 2-D FD stencils of the mimetic scheme result from a cross combination of 1-D stencils that creates a non-standard set of nine

2-D stencils at each $3x3$ grid corner. These grid corners along with the different mimetic stencils within are illustrated in figure 2. Similar to the 1-D case, there are minor sign variations among these nine-stencil sets belonging to different corners, but we only display on this figure the most different computing stencils. Once again, we repeat that Appendix lists all explicit FD computing equations in a amenable coding way for both 1-D and 2-D mimetic schemes.

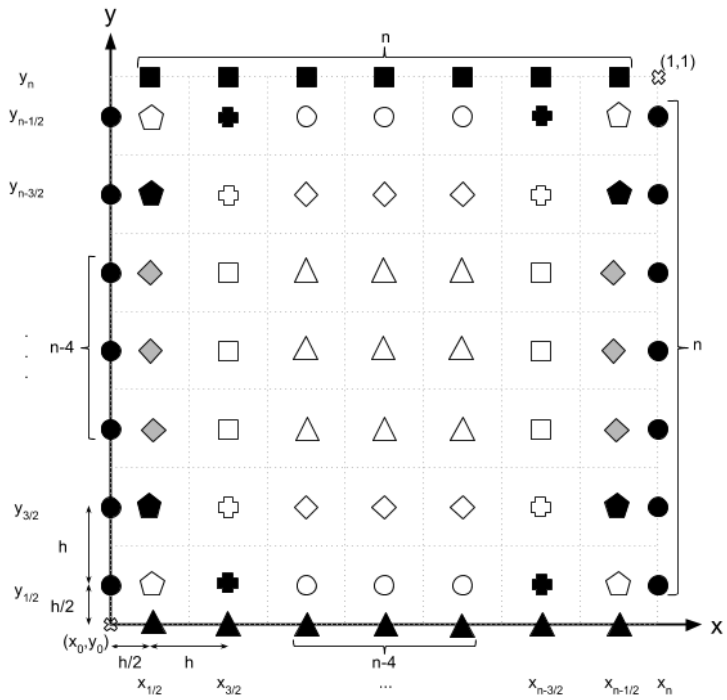

Figure 2. 2-D rectangular grid distribution of u values and mimetic FD stencils

2.2. New acoustic mimetic algorithms

Figure 3 shows the FD mimetic algorithms for solving the acoustic wave equation on 1-D and 2-D rectangular spatial grids. In this diagram, target u wavefield values at all grid nodes are collectively stored in separate computer vectors U^{k+1}, U^k and U^{k-1}, according to the time level of computation which is denoted by the superscript iteration index. The process starts with the allocation of initial conditions on these vectors. Next, time updating of discrete wavefields is carried out by a k-indexed loop until a final simulation time is achieved on k_{steps} iterations. In each loop iteration, the second-order FD computing stencils applied at each non-boundary grid point involves current-time values at same location and at the two adjacent in each direction, all of these are placed on U^k, in addition to the previous-time value at same grid site and stored in U^{k-1}. For this reason, vectors U^k and U^{k-1} must be also updated inside this time loop, in order to make available appropriate values for next iterations. Calculations at boundary nodes are slightly different. On those edges where ABC are applied, the FD stencils

do not involve previous-time values in U^{k-1}, but corresponding components on this computer vector are also updated in the time loop to keep time consistency. Finally, in the case of the 2-D numerical scheme, vector components holding discrete values at boundary $y = 1$ are never updated because of the trivial boundary condition $u = 0$ imposed on it.

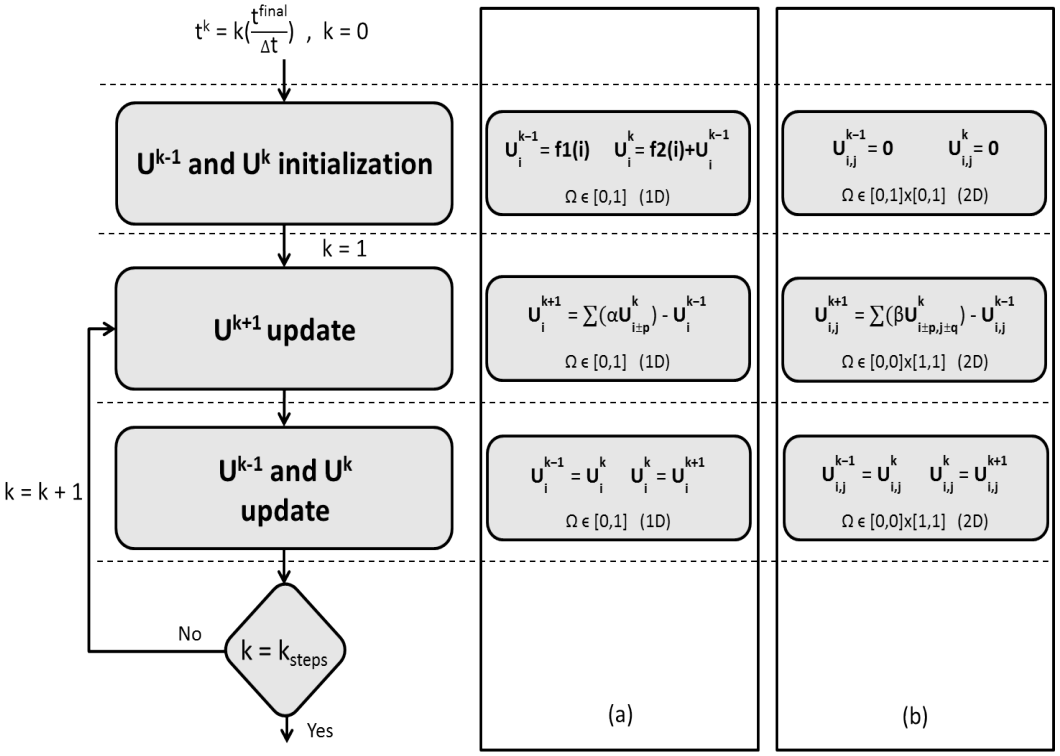

Figure 3. Flowchart of mimetic methods for solving the acoustic wave equation in a: (a) One-dimensional spatial domain; (b) Two-dimensional rectangular domain

3. COMPUTATIONAL OPTIMIZATION

3.1. Hardware

Here, we briefly describe the computer architecture used to implement and speed up our mimetic acoustic methods. This architecture is an Intel Xeon E5-2630 CPU with 6 cores working at a 2.3 GHz clock speed, with 15 MB of cache memory, and access to two GTX 670 GPU cards. On the architecture, the operating system is the 7.1 version of CentOS.

3.2. CPU implementations

3.2.1. A sequential C code. Original codes of the mimetic acoustic schemes were developed in MATLAB without using of any special built-in function. To better use the capabalities of our CPU architecture and apply some code optimization techniques, we migrate original mimetic codes to C language, and use the 4.8.3 version of the *gcc* compiler in this process. The first optimization step was using different compilation flags, among which the $-O3$ flag significantly

improved our C code and achieved the best performance in our numerical tests [21, 29]. This $-O3$ flag automatically introduces the following software optimization techniques: loop unrolling, function inlining and automatic vectorization.

3.2.2. A vectorized code. To fully exploit the benefits of CPU vectorization, we implement a new code with both SSE and AVX intrinsic functions, explicitly coded in our schemes' programs. An appropriate use of these intrinsic functions require a specific memory alignment (32 bits) and data independence in the updating process of floating-point vector $\mathrm{U}^{\mathrm{k+1}}$ components. This is naturally offered by the 1-D scheme implementation, but in the 2-D case the memory allocation of grid values uses a single aligned vector following the ordering shown in figure 4. Let us explain this updating by taking an interior grid location (i, j) denoted with a triangle mark on this figure. The computation of $\mathrm{U}_{i,j}^{k+1}$ requires several aligned loads of nearby values in space and time, and available in the vector registers $\mathrm{U}_{i,j}^{k-1}$, $\mathrm{U}_{i,j}^{k}$, $\mathrm{U}_{i-1,j}^{k}$, $\mathrm{U}_{i+1,j}^{k}$, $\mathrm{U}_{i,j-1}^{k}$, $\mathrm{U}_{i,j+1}^{k}$. For the specific case of AVX functions, eight float values can be loaded into a vector register (the register length is 256 bits). Once all values required for $\mathrm{U}_{i,j}^{k+1}$ update have been loaded, the stencil arithmetic operations is performed by using these vector registers. Some of intrinsic functions involved in this process are `_mm256_mul_ps(_m256 a, _m256 b)`, `_mm256_add_ps( m256 a, m256 b)` and `_mm256_sub_ps(m256 a, m256 b)`. In the SSE case, the number of float values stored by a register reduces from eight to four, and also slightly change the names of some intrinsic functions names with respect to AVX. Figure 5 illustrates $\mathrm{U}_{i,j}^{k+1}$ values whose computation have been vectorized according to detailed capabilities. It is worthy to note that there is an alternative set of vector instructions for larger registers known as AVX2 (the register length is 512 bits). However, this instruction set has been omitted in this work due to hardware limitations. It is expected that results derived from current analysis could be easily extrapolated to the AVX2 instruction set.

3.2.3. Vectorized and parallelized implementation. The next step after explicit SSE/AVX vectorization is loop parallelization by using OpenMP. OpenMP allows us to take full advantage of Intel multi-core CPU architectures. In our OpenMP loop parallelization, we have taken into account two intrinsic aspects of our numerical discretization: (i) data dependencies, and (ii) each loop collects the common set of updating instructions applied on a particular grid zone (boundary, near boundary, or interior nodes). We keep using SSE/AVX vectorization, so we actually update 4/8 grid nodes in each loop iteration, which are processed in parallel along the available microprocessor cores. As an example, we below present a generic parallel loop incluiding these ideas

```
1 #pragma omp parallel for schedule(auto) \
2   private(xmm0,xmm1,xmm2,xmm3)
3   for (i=3;i<(n-1);i=i+8){ //AVX vectorized loop
4      ...
5   }
```

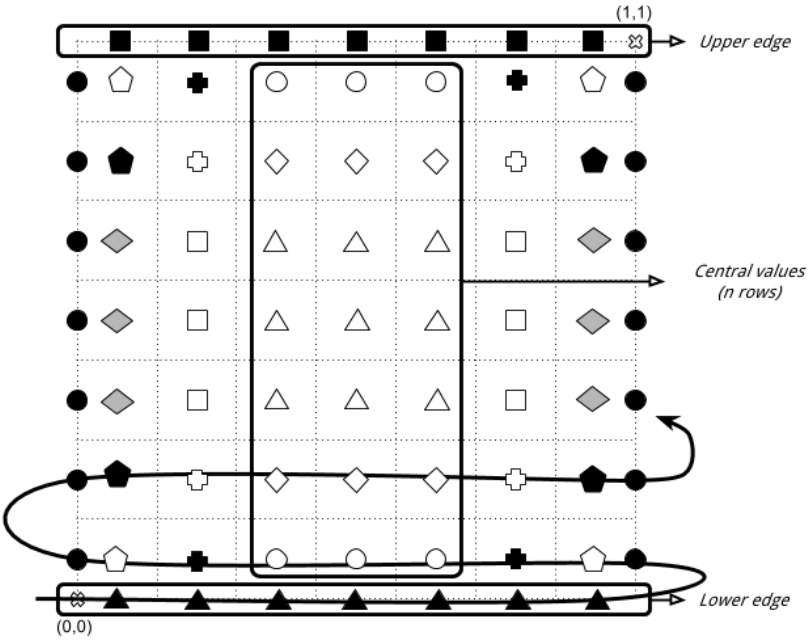

Figure 4. Allocation ordering of 2-D grid values into single aligned vectors U^{k+1}, U^k and U^{k-1} (given by the arrow)

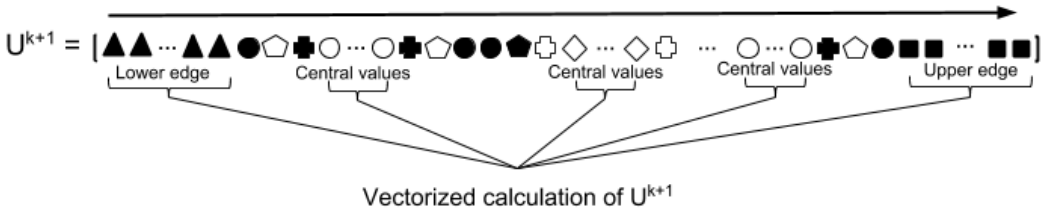

Figure 5. Array allocation of 2-D grid values subject to vectorized optimization

In above code, line 1 sets a parallel loop with automatic load allocation for each thread, and line 2 defines the thread-local variables. Scheduling of OpenMP threads is established by the flag 'auto', which invokes as many threads as the number of available CPU cores. We observed in our experiments that this option provides the best performance for a wide range of scenarios. Increasing the number of threads over the number of cores usually entails an overhead due to communication conflicts, whereas using less threads than available cores might lead to an underuse of a multi-core platform.

3.3. GPU implementations

3.3.1. One-GPU: A parallel CUDA code on a 1 GPU card. Once CPU parallelization has been appropriately exploited, we address a new GPU implementation of our numerical schemes. On the architecture, the programming language is CUDA C based on the *nvcc* compiler. We then migrate our original sequential C code to the CUDA programming paradigm, and the new algorithm is shown in figure 6. To exploit the potential of a CUDA kernel, it is

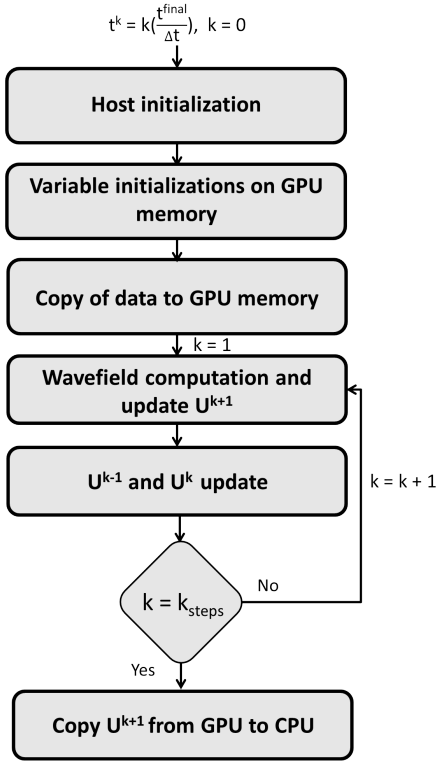

Figure 6. General flowchart of a One-GPU CUDA code for mimetic acoustic wave simulations

necessary to split the whole computation process into several kernels, each one carrying out the update of a particular group of locally closed nodes on the spatial grid, and by means of a serialized instruction set with enough computational load [21]. Below, we also present a code with example lines to write a CUDA kernel that updates U^{k+1} values at interior grid nodes (shown with triangle marks in figure 2).

```
1 __global__ void kernel_red_central_values(float *u_new, float *u_old, float *u, float
      dt, float h, int n, int k){
2   int id = (blockIdx.x)*blockDim.x + threadIdx.x;
3   if( id < (n*n - 8*n +16 )){ //Barrier
4   int pos = (3*n +7) + id + 6*roundf(id/(n-4)); // Position in the grid
5   u_new[pos] = 2*u[pos] - u_old[pos] + (dt*(c/h)*dt*(c/h))*( u[pos-1] + u[pos+1] -
        4*u[pos] + u[pos+n+2] + u[pos-(n+2)]);}
6 }
```

Above kernel begins by assigning an identifier to each thread launched in line 2. There is a barrier condition in line 3 to prevent the access of some extra threads (this programming device is commonly used when the number of blocks per grid is dynamically adjusted in execution time). Once this barrier has been overcome, an array position corresponding to an interior mimetic grid node is assigned to each thread in line 4, and then a new value of U^{k+1} is computed in line 5 and stored on that array location. Additionally, figure 6 shows how input data (initial conditions and other parameters) has been uploaded to GPU memory before

invoking the kernels. This action improves the overall performance by reducing overhead due to continuous CPU - GPU transfers. Next step implements the time iteration where updating kernels, previously declared and appropriately sized, are launched in each cycle. Lastly, the vector with final values after simulation time is spent are transfered to host (CPU) memory.

3.3.2. Two-GPU: A CUDA parallel code on 2 GPU cards. Our main idea when programming on this system is equally splitting the total amount of grid nodes into both GPUs. In this way, each GPU works on updating half of the solution values in parallel to the other one.

Due to the spatial dependencies of FD stencil computation and considering that there is no a direct visibility between memories of both GPUs, a synchronization step must be implemented at each time iteration. This synchronization can be carried out by a Peer-to-Peer communication between both GPUs, which avoids involving the CPU in the process and a potential bottleneck. Figure 7 shows this Two-GPU scheme. It is obvious, that by halving the whole computation load an important reduction of the computational load is experienced by each GPU, in comparison to the One-GPU CUDA implementation. This reduction load leads naturally a decay of each GPU utilization and computational cost, as we discuss in next section.

4. RESULTS

Before presenting the performance results achieved by our CPU and GPU codes, we would like to point out that in the CPU case, performance improvements are cumulative and only the final explicitly vectorized and parallel implementation is considered for comparison against GPU results.

As an interesting reference for numerical developer colleagues, we first report on improvements with respect to original non-optimized MATLAB versions of our acoustic solvers. In the cases of finest grids, the new vectorized and parallel CPU version achieves time reductions up to 730 times (grid size $n = 60000$) and 1504 times (grid size nxn for $n = 5000$) for 1-D and 2-D spatial domains, respectively. On the other hand, new GPU implementations are even faster than the CPU ones since new execution times correspond to fractions $\frac{1}{6237}$ and $\frac{1}{17597}$ of their MATLAB counterparts under same maximum grid resolutions, in 1-D and 2-D spatial domains, respectively. We observe that the vectorized serial codes based on SSE or AVX statements are not competitive with respect to the C version optimized by the -O3 flag compiler. Among other software optimizations, the superiority of the former is mainly due to the $-O3$ automatic vectorization. However, this C version is way much slower than both parallel and vectorized (C+SSE+OpenMP and C+AVX+OpenMP) codes that yield speed up about 12.52 and 12.12, respectively. These performance advantages are mostly attributed to the OpenMP multi-threading, and results are shown in figure 8, using *log* scale, along with speed up observed on vectorized SSE- or AVX-based simulations. Results on this figure are relative to execution times spent by the C code. Concerning the amount of floating point operations per second (FLOPS), we would like to comment about our measurements in the finest mesh test with $n = 60000$. For this grid, the C code performs about 0.47 GFLOPS, while

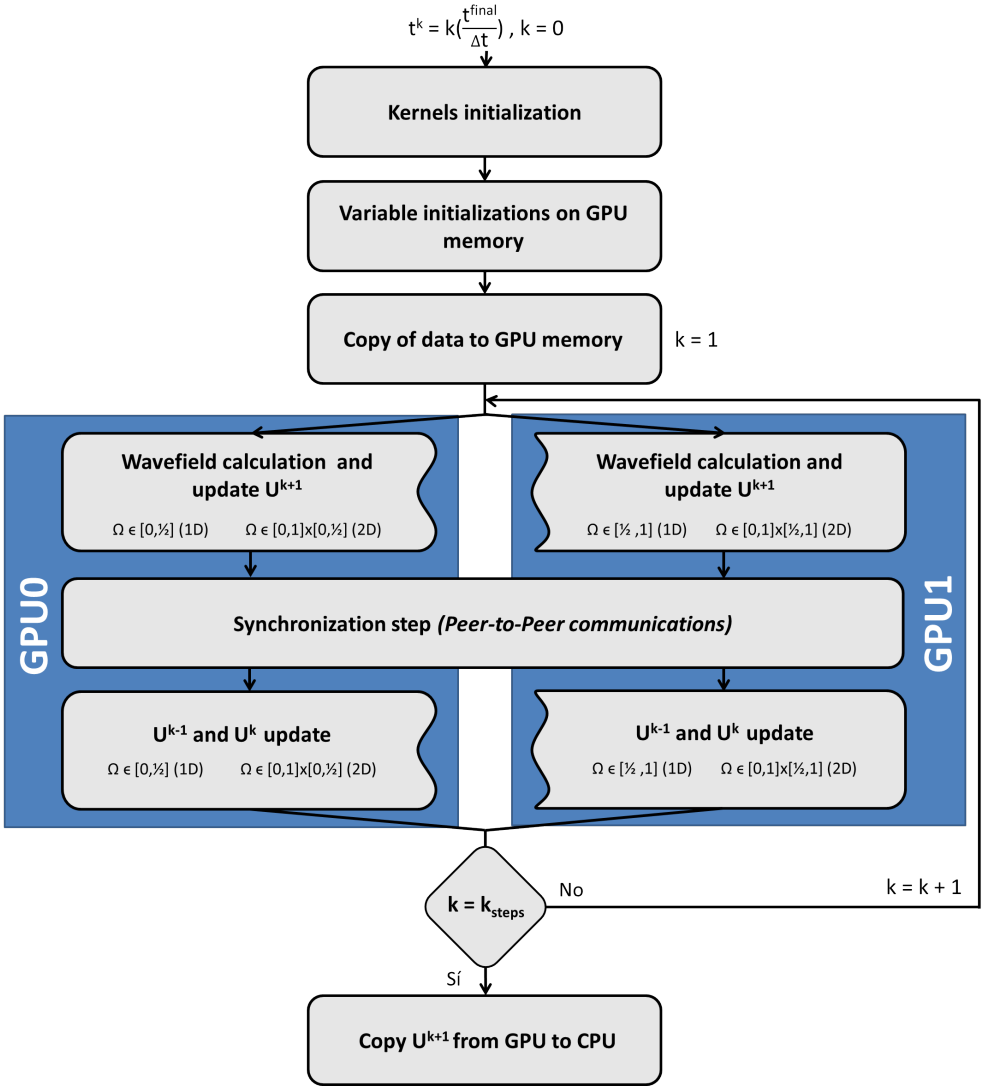

Figure 7. General flowchart of a Two-GPU CUDA parallel code for mimetic acoustic wave simulations

the SSE+OpenMP and AVX+OpenMP parallel and vectorized C versions execute 5.86 and 5.67 GFLOPS, respectively. For the same mesh, our One-GPU CUDA implementation achieve up to 50.04 GFLOPS, while the Two-GPU CUDA code performs nearly 44.96 GFLOPS. The memory bandwidth reached by both GPU implementations is 5.6 GB/s with an average throughput of 480 Kb. Additionally, the memory usage corresponds to 1.41 MB on the same highly-resolved grid. All these performance metrics has been reported by the NVIDIA Visual Profiler. In the particular case of our CUDA implementations, we have verified that the amount of GFLOPS given by thus profiler agree with our theoretical estimates.

From these results, we next isolate execution times of all four 1-D parallel implementations: the (C and C+SSE+OpenMP) CPU codes, and (One-GPU and Two-GPU) CUDA codes. Relative to the C code, One-GPU results competitive after $n = 400$ and advantageous for grids with 1600 cells and finer with a speed up that exceeds 10, and finally reaches a notorious

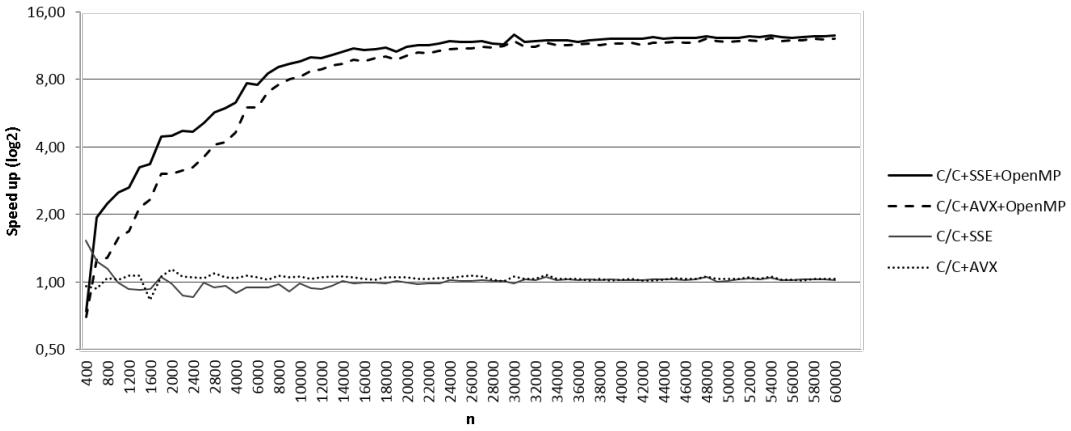

Figure 8. Speed up *log*2 values of the C implementation versus the vectorized and parallel CPU codes on a 1-D numerical test

improvement of 106.9 when $n = 60000$. Now, the relative speed up achieved by Two-GPU CUDA code is about 1.42 on grids with $n = 1000$, and steadily increases to nearly 96.04 for $n = 60000$. Figure 9 depicts these time ratios for all grid sizes considered in our tests, and also compares the speed up in *log* scale of both CUDA codes using the best CPU implementation C+SSE+OpenMP as reference. In this case, One-GPU reaches a modest improvement of 8.54 when $n = 60000$, while Two-GPU CUDA implementation starts being competitive for $n = 6000$ with a speed up of 1.55 than later improves to 7.67 for $n = 60000$.

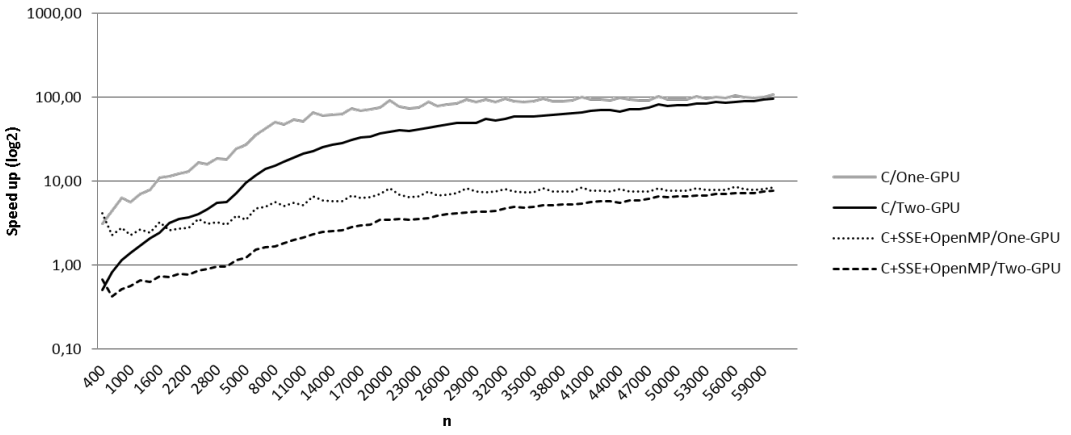

Figure 9. Speed up *log*2 values of the GPU implementations versus the CPU (C and C+SSE+OpenMP) codes on a 1-D numerical test

We additionally compare the execution times of both CUDA implementations on our 1-D experiments, and time ratios of Two-GPU using One-GPU as reference are shown in *log* scale in figure 10. Unfortunately, Two-GPU results do not improve One-GPU times, and actually are quite similar. The computational utilization is a metric related to the efficiency of a CUDA implementation, and measures the throughput achieved by a GPU platform when available resources are occupied to the maximum extent. In our case, this metric depends on the volume

of computational load, and when this volume is not enough to fully exploit GPU resources, the utilization is not optimal. In the case of our Two-GPU CUDA implementation, we have divided the whole computation load between two GPUs, but find that even on those highly refined grids where numerical errors almost reach machine precision, each load is not big enough to fully utilize each GPU resources. In fact, the NVIDIA visual profiler showed how the Two-GPU CUDA code attains a computing utilization under 23% for a grid size of $n = 60000$, while One-GPU reaches a higher 91% on same grid.

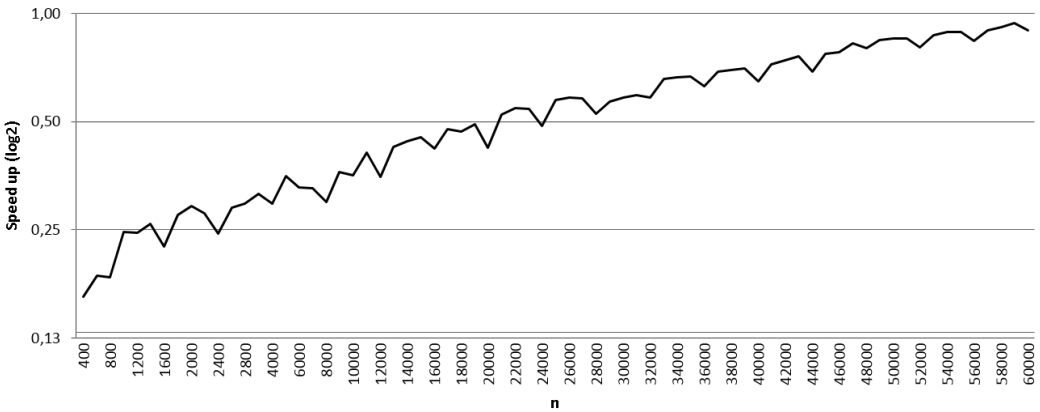

Figure 10. Speed up $log2$ values of the Two-GPU CUDA implementation relative to the One-GPU CUDA code on a 1-D numerical test

Now, we focus on numerical experiments on 2-D spatial domains where a higher volume of computational data and more complex data dependencies, with respect to the 1-D case, make results significantly different.

Figure 11 shows the speed-up reached by all CPU vectorized implementations relative to the simple C solver. Similar to the 1-D case, we also observe here that the performance of vectorized SSE and AVX codes is very similar to one on C $-O3$ simulations on most grids. Again, this C solver is benefit from the automatic vectorization enabled by the compilation flag O3. On the other hand, for $n = 1200$ the C+AVX+OpenMP implementation achieves a significant speed up of 8.86 respect to the C solver, and the C+SSE+OpenMP also reaches a gain little above 8. Data associated to these tests take nearly half of the capacity of the CPU L3 cache memory. Under this condition, using intrinsic functions and multi-threading allow taking full advantage of all resources available at the microprocessor. For $n \geq 1400$, relative performances of both OpenMP-based implementations steadily decay mainly due to memory saturation (Cache misses) until reach a speed up of 2 on simulations with $n = 2000$. As memory allocation increases with n beyond the L3 cache memory capacity, the performance of these vectorized codes is affected and steadily follows same speed up level. Similar to the 1-D vectorized CPU codes, figure 11 also illutrates that C+AVX+OpenMP and C+SSE+OpenMP execution times are comparable, and there is a minor speed-up gain showed by the former on coarse grids, $n \leq 1400$. For a grid size of $n = 5000$, calculations of the One-GPU CUDA implementation spends about 42.36 GFLOPS, while the Two-GPU CUDA code performs nearly 77.07 GFLOPS. The memory usage is close to 300 MB on these highly-resolved

simulations. Similar to previous cases, these performance metrics have been reported by the NVIDIA Visual Profiler.

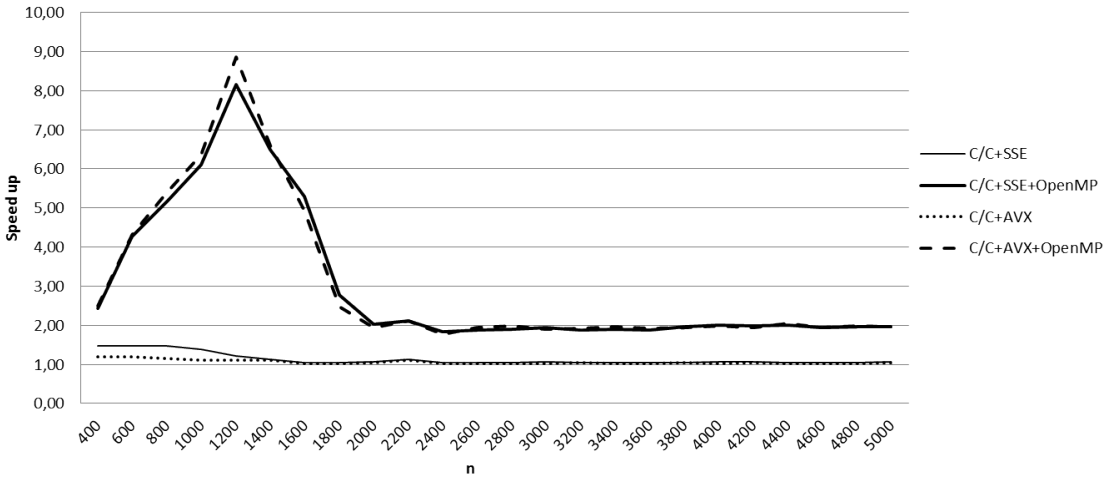

Figure 11. Speed up of the C implementation versus the vectorized and parallel CPU codes on a 2-D numerical test

It is worthy to note that both GPUs are physically connected to a PCIe bus, in such a way that the overall bus performance is just x8 instead of x16. Physical dimensions of GPU cards are incompatible with respect to the gap between PCIe connections in the main board, so there is no way to connect both GPUs to the PCIe slots and ensure a performance of x16. However, our final speed up results and bandwidth profiling measurements indicate that this hardware issue does not affect the overall performance in a significant way. The bus speed in inter-GPU communications is close to 70% of the PCIex8, whereas the bandwidth of Host-to-Device (HtD) transactions reaches a 100% of the total PCIex8 speed. It is worth to note that the transaction Device-to-Device (DtD) in 1-D test were far from the bandwidth achieved for the 2-D case. We mainly attribute the speed reduction in Two-GPU communications on 1-D simulations to the low amount of data transferred in each HtD communication, or the pattern followed by the data. These transactions do not exploit the maximum PCIe bus speed. On the other hand, higher performances observed on 2-D simulations are partially due to communications take place at almost the maximum bus speed, and bigger amount of data are transferred in a single message (variable and matrix initializations, and result retrieval). In these tests, DtD speed is not limited by the x8 setup and HtD transactions only occur at the beginning and the end of our simulations. Since HtD operations represent a small fraction compared to the computation time, the impact of achieving x16PCIe bus performance would not change our global results in great manner.

In figure 12, we show the speed up achieved by both CUDA solvers with respect to C and C+AVX+OpenMP CPU implementations, on our 2-D numerical tests. Relative to the C solver, One-GPU reaches a gain of 12.28 in the most-refined grid case ($n = 5000$), while the speed up accomplished by Two-GPU is 23.06 on same test. Now, taken the vectorized C+AVX+OpenMP simulation as reference on same grid, we observe a speed up of 6.23 and 11.7 on One-GPU and Two-GPU calculations, respectively. These 2-D Two-GPU results represent

a performance gain relative to the implementation on 1-D, which is attributed to an increase of 60% of GPU utilization under a higher data and processing demand. In all these experiments, Two-GPU presents a consistently better performance compared to One-GPU, and particularly for $n \geq 5000$, this gain is about 6 speed up units. Again, the higher occupancy of both GPU's resources (with a peak of 84%) leads to this performance superiority on dense grids. Finally, we additionally compare the performance of this Two-GPU implementation against to the One-GPU CUDA code. The former shows a steady speed up of 1.88 over the latter on dense grids with $n \geq 2000$.

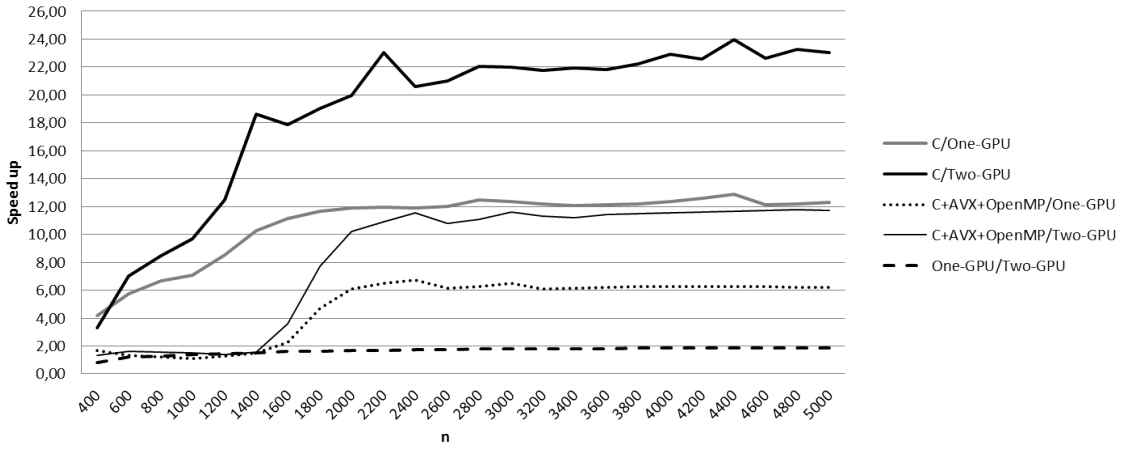

Figure 12. Speed up of: i) The GPU implementations versus the CPU (C and C+AVX+OpenMP) codes;
ii) The One-GPU implementation vs Two-GPU code on a 2-D numerical test

5. CONCLUSIONS

In this work, we present progressive computing accelerations of a mimetic finite difference method for the simulation of wave propagation on 1-D and 2-D acoustic media, on CPU and GPU computer platforms. We start our CPU optimizations by migrating original MATLAB codes to the 4.8.3 *gcc* compiler, test built-in flags for better performance, and observe significant time reductions with the $-O3$ compiling flag. Next, we introduce intrinsic SSE and AVX vectorization statements and OpenMP loop paralelization to this code, which allows taking full advantage of multi-core CPU architectures. Results from these new optimized codes have shown a significant performance improvement by reducing the MATLAB-based codes' execution times to 730 times on 1-D and 1504 times on 2-D tests, respectively. In this process, it has been also exposed that CPU implementations present a cache limitation for high data volumes, and this limits actual improvements to medium-sized grids. In addition, new CUDA implementations have been developed by translating acoustic solver C versions to the NVIDIA CUDA architecture with one GPU card. In both 1-D and 2-D spatial domains, GPU implementations behave faster than their CPU counterparts, and execution times correspond to the fractions $\frac{1}{6237}$ (1-D test) and $\frac{1}{17597}$ (2-D test) relative to original MATLAB computing times. On same experiments, a comparison between CPU-based and GPU-based results show

that our CUDA implementation is up to 8.54 times faster than our best vectorized parallel code in a 1-D scenario, and is up to 6.23 times more rapid in a 2-D scenario. Only in the case of 2-D coarse grids ($n \leq 1200$), the optimal CPU code is nearly competitive to the GPU implementation.

We finally extend our CUDA implementations to a platform with two GPU cards (Two-GPU version). Relative to the best CPU code, the new implementation manages to be 11.7 times faster in 2-D tests with very fine discretizations. In any other case, execution times of the Two-GPU CUDA implementation are comparable or even worse because of the low computational load. Our results confirm the well known fact that CUDA potential is better exhibited under high computational loads, as in the case of 2-D tests with computing utilization up to 90.4%, in comparison to less demanding 1-D tests with 23% utilization, at the most. We finally perform 2-D tests on a broad range of grid sizes, and observe that Two-GPU CUDA is nearly twice faster than the One-GPU solver. These last experiments represent perfect realizations of the theoretical expectation after doubling the hardware GPU capacity.

ACKNOWLEDGEMENTS

Authors from Universidad Central de Venezuela (UCV) were partially supported by: Consejo de Desarrollo Científico y Humanístico de la UCV, Vice-rectorado Académico de la UCV, Coordinación de Investigación de la Facultad de Ciencias UCV, Banco Central de Venezuela (BCV) and Generalitat Valenciana under project PROMETEOII/2015/015.

REFERENCES

1. Abdelkhalek R., Calandra H., Coulaud O., Roman J. and Latu G. 2009. *Fast seismic modeling and reverse time migration on a GPU cluster High Performance Computing and Simulation*. International Conference on HPCS'09, 36-43.
2. Carcione J., Finetti I. and Gei D. 2003. *Seismic modelling study of the earth's deep crust*, Geophysics, 68, 656-664.
3. Clayton R. W. and Engquist B. 1980. *Absorbing boundary conditions for wave-equation migration*, Geophysics, 45, 895-904.
4. Castillo J. E. and Grone R. D. 2003. *A Matrix Analysis Approach to Higher-Order Approximations for Divergence and Gradients Satisfying a Global Conservation Law*, Matrix Analysis Applications, 25, 128-142.
5. Castillo J. E., Hyman J. M., Shashkov M. and Steinberg S. 2001. *Fourth- and sixth-order conservative finite difference approximations of the divergence and gradient*, Applied Numerical Mathematics: Transactions of IMACS, 37, 171-187.
6. Castillo J. E. and Miranda G. F. 2013. Mimetic Discretization Methods Chapman and Hall, 1-260.
7. Castillo J. E. and Yasuda M. 2005. *Linear systems arising for second-order mimetic divergence and gradient discretizations*, Mathematical Modelling and Algorithms, 4, 67-82.
8. De la Puente J., Ferrer M., Hanzich M., Castillo J. E. and Cela J. M. 2014. *Mimetic seismic wave modeling including topography on deformed staggered grids*, Geophysics, 79, T125-T141.
9. Duncan D. B. 1998. *Difference approximations of acoustic and elastic wave equations*, Numerical Methods for Wave Propagation, Eds E F Toro and J F Clarke, Fluid Mechanics and its Applications, 47, 197-210.
10. Ely G. P., Day S. M. and Minster J. 2008. *A support-operator method for viscoelastic wave modelling in 3-D heterogeneous media*, Geophysical Journal International, 172, 331-344.

11. Eller P., Cheng J.-R. and Albert D. 2010. *Acceleration of 2-D Finite Difference Time Domain Acoustic Wave Simulation Using GPUs*, High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 350-356.
12. Etgen J. T. and O'Brien M. J. 2007. *Computational methods for large-scale 3D acoustic finite-difference modeling: A tutorial*, Geophysics, 72, SM223-SM230.
13. Graves R. W. 1996. *Simulating seismic wave propagation in 3D elastic media using staggered-grid finite differences*, Bulletin of the Seismological Society of America, 86, 1091-1106.
14. Kristek J., Moczo P. and Archuleta R. J. 2002. *Efficient Methods to Simulate Planar Free Surface in the 3D 4th-Order Staggered-Grid Finite-Difference Schemes*, Studia Geophysica et Geodaetica, 46, 355-381.
15. Lipnikov K., Manzini G. and Shashkov M. 2014. *Mimetic finite difference method*, Journal of Computational Physics, 257, 1163-1227.
16. Mehra R., Raghuvanshi N., Savioja L., Lin M. C. and Manocha D. 2012. *An efficient GPU-based time domain solver for the acoustic wave equation*, Applied Acoustics, 73, 83-94.
17. Michea D. and Komatitisch D. 2010. *Accelerating a three-dimensional finite-difference wave propagation code using GPU Graphics cards*, Geophysical Journal International, 182, 389-402.
18. Molero M. and Iturrarn-Viveros U. 2013. *Accelerating numerical modeling of wave propagation through 2-D anisotropic materials using OpenCL*, Ultrasonics, 53, 815-822.
19. Qian J., Wu S. and Cui R. 2013. *Accuracy of the staggered-grid finite-difference method of the acoustic wave equation for marine seismic reflection modeling*, Chinese Journal of Oceanology and Limnology, 31, 169-177.
20. Reynolds A. C. 1978. *Boundary conditions for the numerical solution of wave propagation problems*, Geophysics, 43, 1099-1110.
21. Rodríguez R. 2014. *Optimización y paralelización de un código para la simulación de problemas acústicos*, Proyecto final de carrera, ETSETB, Universitat Politècnica de Cataluña, 1-131.
22. Rojas O. 2007. *Mimetic Finite Difference Modeling of 2D Elastic P-SV Wave Propagation*, Qualifying examination report, Computational Science Research Center, San Diego State University.
23. Rojas O., Day S., Castillo J. E. and Dalguer L. A. 2008. *Modelling of rupture propagation using high-order mimetic finite differences*, Geophysical Journal International, 172, 631-650.
24. Rojas O., Dunham E. M., Day S., Dalguer L. A. and Castillo J. E. 2009. *Finite difference modelling of rupture propagation with strong velocity-weakening friction*, Geophysical Journal International, 179, 1831-1858.
25. Rojas O., Otero B., Castillo J. E. and Day S. M. 2014. *Low dispersive modeling of Rayleigh waves on partly staggered grids*, Computational Geosciences, 18, 29-43.
26. Samarskii A. A., Tishkin V. F., Favorskii A. P. and Shashkov M. Y. 1981. *Operational finite-difference schemes*, Differential Equations,17, 854-862.
27. Solano-Feo F., Guevara-Jordan J., Rojas O., Otero B. and Rodríguez R. 2016. *A new mimetic scheme for the acoustic wave equation*, Journal of Computational and Applied Mathematics, 295, 2-12.
28. Solano F., Guevara-Jordan J. M. and Rojas O. 2012. *An explicit mimetic method for transient beam equations*, SVMNI, Avances en simulación computacional y modelado numérico, MM43-MM48.
29. M. S. R. and the GCC Developer Community. 2010. *Using the GNU Compiler Collection. For GCC version 4.6.4*, GNU Press, 89-137.
30. Weiss R. M. and Shragge J. 2013. *Solving 3D anisotropic elastic wave equations on parallel GPU devices*, Geophysics, 78, 1-9.
31. Virieux J. 1984. *SH wave propagation in heterogeneous media: Velocity-stress finite-difference method*, Geophysics, 49, 1933-1957.
32. Virieux J. 1986. *P-SV wave propagation in heterogeneous media: Velocity-stress finite-difference method*, Geophysics, 51, 889-901.

APPENDIX: MIMETIC OPERATORS AND FD STENCILS IN 1-D AND 2-D DOMAINS

On the 1-D staggered grid described in section 2 with n cells and constant step h, second-order mimetic finite differences can be comprised into the following matrix operators

$$G = \frac{1}{h}\begin{pmatrix} -\frac{8}{3} & 3 & -\frac{1}{3} & 0 & \cdots & 0 \\ 0 & -1 & 1 & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & 0 & 0 \\ 0 & \cdots & 0 & -1 & 1 & 0 \\ 0 & \cdots & 0 & \frac{1}{3} & -3 & \frac{8}{3} \end{pmatrix}_{(n+1)\times(n+2)},$$

$$D = \frac{1}{h}\begin{pmatrix} 0 & 0 & \cdots & \cdots & 0 & 0 \\ -1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 1 & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & 0 & 0 \\ 0 & \cdots & 0 & -1 & 1 & 0 \\ 0 & 0 & \cdots & 0 & -1 & 1 \\ 0 & 0 & \cdots & \cdots & 0 & 0 \end{pmatrix}_{(n+2)\times(n+1)},$$

and

$$B = \begin{pmatrix} -1 & 0 & 0 & \cdots & & & 0 \\ \frac{1}{8} & -\frac{1}{8} & 0 & \cdots & & & \\ -\frac{1}{8} & \frac{1}{8} & 0 & \cdots & & & \\ 0 & 0 & 0 & \cdots & & & \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ & & & \cdots & 0 & 0 & 0 \\ & & & \cdots & 0 & -\frac{1}{8} & \frac{1}{8} \\ & & & \cdots & 0 & \frac{1}{8} & -\frac{1}{8} \\ 0 & & & \cdots & 0 & 0 & 1 \end{pmatrix}_{(n+2)\times(n+1)}.$$

In the gradient operator G, first and last rows correspond to lateral FD stencils to approximate the exact gradient at both boundaries. Remaining G rows coincide with the standard central FD stencil with second order accuracy, that also conforms interior rows of divergence operator D. The first and last zero rows in D respond to the fact that the continuous divergence does not play a physical role at boundaries in most standard applications. The operator B is exclusive of mimetic FD discretizations, and first and last rows of BG allow approximating normal derivates $\frac{\partial}{\partial \hat{n}}$ at boundary nodes, while interior nonzero rows intervene on Laplacian approximations at mid-cell points $x_{\frac{1}{2}}, x_{\frac{3}{2}}, x_{n-\frac{1}{2}}$ and $x_{n-\frac{3}{2}}$. Notice that null D rows also permit a natural superposition of Neumann- or Robin-type boundary conditions with interior Laplacian approximation on the final mimetic discretization operator $(DG + BG)$.

The collection of FD stencils for the mimetic solution of (1), (2), and (3), on the 1-D grid depicted in figure 1 result from combining approximations in $DG + BG$ for the spatial terms to a mixed time integration given by the Leapfrog schemes at grid centers $x_{\frac{1}{2}}, \ldots, x_{n-\frac{1}{2}}$,

and a simple forward difference at boundary x_n, for time steps $t_k = k\Delta t$. These stencils are transcribed below in the special case of $f_1(t) = f_2(t) = 0$ and $c = 1$.

(i) Node x_0

$$\frac{8}{3h}u_0^{k+1} + \frac{3}{h}u_{\frac{1}{2}}^{k+1} - \frac{1}{3h}u_{\frac{3}{2}}^{k+1} = 0$$

(ii) Center $x_{\frac{1}{2}}$

$$u_{\frac{1}{2}}^{k+1} = 2u_{\frac{1}{2}}^{k} - u_{\frac{1}{2}}^{k-1} + \Delta t^2\left[f\left(x_{\frac{1}{2}}, t^k\right) + \left(\frac{8}{3h^2} - \frac{1}{3h}\right)u_0^k + \left(\frac{1}{2h} - \frac{4}{h^2}\right)u_{\frac{1}{2}}^k + \left(\frac{4}{3h^2} - \frac{1}{6h}\right)u_{\frac{3}{2}}^k\right]$$

(iii) Center $x_{\frac{3}{2}}$

$$u_{\frac{3}{2}}^{k+1} = 2u_{\frac{3}{2}}^{k} - u_{\frac{3}{2}}^{k-1} + \Delta t^2\left[f\left(x_{\frac{3}{2}}, t^k\right) + \left(\frac{1}{3h}\right)u_0^k + \left(\frac{1}{h^2} - \frac{1}{2h}\right)u_{\frac{1}{2}}^k + \left(\frac{1}{6h} - \frac{2}{h^2}\right)u_{\frac{3}{2}}^k + \left(\frac{1}{h^2}\right)u_{\frac{5}{2}}^k\right]$$

(iv) Center $x_{i+\frac{1}{2}}$ at grid interior

$$u_{i+\frac{1}{2}}^{k+1} = 2u_{i+\frac{1}{2}}^{k} - u_{i+\frac{1}{2}}^{k-1} + \left(\frac{\Delta t}{h}\right)^2\left[f\left(x_{i+\frac{1}{2}}, t^k\right) + u_{i+\frac{3}{2}}^k - 2u_{i+\frac{1}{2}}^k + u_{i-\frac{1}{2}}^k\right]$$

(v) Node x_n

$$u_n^{k+1} = u_n^k - \left(\frac{\Delta t}{h}\right)\left[\frac{8}{3}u_n^k - 3u_{n-\frac{1}{2}}^k + \frac{1}{3}u_{n-\frac{3}{2}}^k\right]$$

FD stencils at mid-cell points $x_{n-\frac{1}{2}}$ and $x_{n-\frac{3}{2}}$ are symmetric replications of computing formulas at leftmost centers $x_{\frac{1}{2}}$ and $x_{\frac{3}{2}}$, respectively.

The mimetic FD discretization of the 2-D wave propagation problem (1), (4), and (5) on the unit square, yields a non-standard stencil set mainly due to computing formulas at interior mid-cell points displaced by either $\frac{h}{2}$ or $\frac{3h}{2}$ from any boundary. These unique mimetic stencils along with FD formulas applied at boundaries and remaining interior points are illustrated in figure 2 and given as follows

(i) Nodes on the bottom boundary, $U_{i,0}$ for $i = \frac{1}{2}, \ldots, n - \frac{1}{2}$:

$$U_{i,0}^{k+1} = U_{i,0}^k + \left(\frac{\triangle t}{h}\right)\left[-\frac{8}{3}U_{i,0}^k + 3U_{i,\frac{1}{2}}^k - \frac{1}{3}U_{i,\frac{3}{2}}^k\right]$$

(ii) Nodes on the left and right edges, $U_{i,j}$ for $i \in \{0, n\}$ and $j = \frac{1}{2}, \ldots, n - \frac{1}{2}$:

$$U_{i,j}^{k+1} = U_{i,j}^k + \left(\frac{\triangle t}{h}\right)\left[-\frac{8}{3}U_{i,j}^k + 3U_{i,j\pm\frac{1}{2}}^k - \frac{1}{3}U_{i,j\pm\frac{3}{2}}^k\right]$$

(iii) Nodes on the top boundary, $U_{i,n}$ for $i = \frac{1}{2}, \ldots, n - \frac{1}{2}$:

$$U_{i,n}^{k+1} = 0$$

(iv) Interior nodes $U_{i,j}$ for $i, j \in \left\{\frac{1}{2}, n - \frac{1}{2}\right\}$:

$$\begin{aligned} U_{i,j}^{k+1} &= 2U_{i,j}^{k} - U_{i,j}^{k-1} + \triangle t^2 \left[\left(\frac{8}{3h^2} - \frac{1}{3h} \right) U_{i\mp\frac{1}{2},j}^{k} + \left(\frac{1}{h} - \frac{8}{h^2} \right) U_{i,j}^{k} + \left(\frac{4}{3h^2} - \frac{1}{6h} \right) U_{i\pm1,j}^{k} + \right. \\ &\left. \left(\frac{8}{3h^2} - \frac{1}{3h} \right) U_{i,j\mp\frac{1}{2}}^{k} + \left(\frac{4}{3h^2} - \frac{1}{6h} \right) U_{i,j\pm1}^{k} \right] \end{aligned}$$

(v) Interior nodes $U_{i,j}$ for $i \in \left\{\frac{3}{2}, n - \frac{3}{2}\right\}$ and $j \in \left\{\frac{1}{2}, n - \frac{1}{2}\right\}$:

$$\begin{aligned} U_{i,j}^{k+1} &= 2U_{i,j}^{k} - U_{i,j}^{k-1} + \triangle t^2 \left[\left(\frac{8}{3h^2} - \frac{1}{3h} \right) U_{i,j\mp\frac{1}{2}}^{k} + \left(\frac{2}{3h} - \frac{6}{h^2} \right) U_{i,j}^{k} + \left(\frac{4}{3h^2} - \frac{1}{6h} \right) U_{i,j\pm1}^{k} + \right. \\ &\left. \frac{1}{3h} U_{i\mp\frac{3}{2},j}^{k} + \left(\frac{1}{h^2} - \frac{1}{2h} \right) U_{i\mp1,j}^{k} + \frac{c^2}{h^2} U_{i\pm1,j}^{k} \right] \end{aligned}$$

(vi) Interior nodes $U_{i,j}$ for $i = \frac{5}{2}, \ldots, n - \frac{5}{2}$ and $j \in \left\{\frac{1}{2}, n - \frac{1}{2}\right\}$:

$$\begin{aligned} U_{i,j}^{k+1} &= 2U_{i,j}^{k} - U_{i,j}^{k-1} + \triangle t^2 \left[\left(\frac{8}{3h^2} - \frac{1}{3h} \right) U_{i,j\mp\frac{1}{2}}^{k} + \left(\frac{1}{2h} - \frac{6}{h^2} \right) U_{i,j}^{k} + \left(\frac{4}{3h^2} - \frac{1}{6h} \right) U_{i,j\pm1}^{k} + \right. \\ &\left. \frac{1}{h^2} U_{i-1,j}^{k} + \frac{1}{h^2} U_{i+1,j}^{k} \right] \end{aligned}$$

(vii) Interior nodes $U_{i,j}$ for $i \in \left\{\frac{1}{2}, n - \frac{1}{2}\right\}$ and $j \in \left\{\frac{3}{2}, n - \frac{3}{2}\right\}$:

$$\begin{aligned} U_{i,j}^{k+1} &= 2U_{i,j}^{k} - U_{i,j}^{k-1} + \triangle t^2 \left[\left(\frac{8}{3h^2} - \frac{1}{3h} \right) U_{i\mp\frac{1}{2},j}^{k} + \left(\frac{2}{3h} - \frac{6}{h^2} \right) U_{i,j}^{k} + \left(\frac{4}{3h^2} - \frac{1}{6h} \right) U_{i\pm1,j}^{k} + \right. \\ &\left. \frac{1}{3h} U_{i,j\mp\frac{3}{2}}^{k} + \left(\frac{1}{h^2} - \frac{1}{2h} \right) U_{i,j\mp1}^{k} + \frac{1}{h^2} U_{i,j\pm1}^{k} \right] \end{aligned}$$

(viii) Interior nodes $U_{i,j}$ for $i \in \left\{\frac{3}{2}, n - \frac{3}{2}\right\}$ and $j \in \left\{\frac{3}{2}, n - \frac{3}{2}\right\}$:

$$\begin{aligned} U_{i,j}^{k+1} &= 2U_{i,j}^{k} - U_{i,j}^{k-1} + \triangle t^2 \left[\left(\frac{1}{h^2} - \frac{1}{2h} \right) \left(U_{i,j\mp1}^{k} + U_{i\mp1,j}^{k} \right) + \left(\frac{1}{3h} - \frac{4}{h^2} \right) U_{i,j}^{k} + \right. \\ &\left. \frac{1}{3h} \left(U_{i,j\mp\frac{3}{2}}^{k} + U_{i\mp\frac{3}{2},j}^{k} \right) + \frac{1}{h^2} \left(U_{i,j\pm1}^{k} + U_{i\pm1,j}^{k} \right) \right] \end{aligned}$$

(ix) Interior nodes $U_{i,j}$ for $i = \frac{5}{2}, \ldots, n - \frac{5}{2}$ and $j \in \left\{\frac{3}{2}, n - \frac{3}{2}\right\}$:

$$\begin{aligned} U_{i,j}^{k+1} &= 2U_{i,j}^{k} - U_{i,j}^{k-1} + \triangle t^2 \left[\left(\frac{1}{h^2} - \frac{1}{2h} \right) U_{i,j\mp1}^{k} + \left(\frac{1}{6h} - \frac{4}{h^2} \right) U_{i,j}^{k} + \frac{1}{3h} U_{i,j\mp\frac{3}{2}}^{k} + \right. \\ &\left. \frac{1}{h^2} \left(U_{i+1,j}^{k} + U_{i-1,j}^{k} + U_{i,j\pm1}^{k} \right) \right] \end{aligned}$$

(x) Interior nodes $U_{i,j}$ for $i \in \left\{\frac{1}{2}, n-\frac{1}{2}\right\}$ and $j = \frac{5}{2}, \ldots, n-\frac{5}{2}$:

$$U_{i,j}^{k+1} = 2U_{i,j}^{k} - U_{i,j}^{k-1} + \triangle t^2 \left[\left(\frac{8}{3h^2} - \frac{1}{3h} \right) U_{i\mp\frac{1}{2},j}^{k} + \left(\frac{1}{2h} - \frac{6}{h^2} \right) U_{i,j}^{k} + \left(\frac{4}{3h^2} - \frac{1}{6h} \right) U_{i\mp 1,j}^{k} + \frac{1}{h^2} \left(U_{i,j+1}^{k} + U_{i,j-1}^{k} \right) \right]$$

(xi) Interior nodes $U_{i,j}$ for $i \in \left\{\frac{3}{2}, n-\frac{3}{2}\right\}$ and $j = \frac{5}{2}, \ldots, n-\frac{5}{2}$:

$$U_{i,j}^{k+1} = 2U_{i,j}^{k} - U_{i,j}^{k-1} + \triangle t^2 \left[\left(\frac{1}{h^2} - \frac{1}{2h} \right) U_{i\mp 1,j}^{k} - \left(\frac{1}{6h} + \frac{4}{h^2} \right) U_{i,j}^{k} + \frac{1}{3h} U_{i\mp\frac{3}{2},j}^{k} + \frac{1}{h^2} U_{i\pm 1,j}^{k} + \frac{1}{h^2} \left(U_{i,j-1}^{k} + U_{i,j+1}^{k} \right) \right]$$

(xii) Interior nodes $U_{i,j}$ for $i, j = \frac{5}{2}, \ldots, n-\frac{5}{2}$:

$$U_{i,j}^{k+1} = 2U_{i,j}^{k} - U_{i,j}^{k-1} + \frac{\triangle t^2}{h^2} \left[U_{i-1,j}^{k} + U_{i+1,j}^{k} - 4U_{i,j}^{k} + U_{i,j-1}^{k} + U_{i,j+1}^{k} \right]$$

Above, subindices with optional sings $\pm$ or $\mp$ must be taken with the top choice at grid points closer to the bottom or left edges, while the bottom sign must be used at grid locations closer to the top or right boundaries.