Asynchronous Computations for Solving the Acoustic Wave Propagation Equation

Kadir Akbudak¹, Hatem Ltaief¹, Vincent Etienne², Rached Abdelkhalak¹, Thierry Tonel|ot² and David Keyes¹

Abstract

We design and implement an asynchronous computational scheme for solving the acoustic wave propagation equation with Absorbing Boundary Conditions (ABCs) in the context of seismic imaging applications. While the Convolutional Perfectly Matched Layer (CPML) is typically used for ABCs in the oil and gas industry, its formulation further stresses memory accesses and decreases the arithmetic intensity at the physical domain boundaries. The challenges with CPML are twofold: (1) the strong, inherent data dependencies imposed on the explicit time stepping scheme render asynchronous time integration cumbersome and (2) the idle time is further exacerbated by the load imbalance introduced among processing units. In fact, the CPML formulation of the ABCs requires expensive synchronization points, which may hinder parallel performance of the overall asynchronous time integration. In particular, when deployed in conjunction with the Multicore-optimized Wavefront Diamond Temporal Blocking (MWD-TB) approach for the inner domain points, it results into a major performance slow down. To relax CPML's synchrony and mitigate the resulting load imbalance, we embed CPML's calculation into MWD-TB's inner loop and carry on the time integration with fine-grained computations in an asynchronous, holistic way. This comes at the price of storing transient results to alleviate dependencies from critical data hazards, while maintaining the numerical accuracy of the original scheme. Performance and scalability results on various x86 architectures demonstrate the superiority of MWD-TB with CPML support against the standard spatial blocking on various grid sizes. To our knowledge, this is the first practical study that highlights the consolidation of CPML ABCs with asynchronous temporal blocking stencil computations.

Keywords


1 Introduction

The end of Dennard scaling and Moore’s law, both of which have driven the hardware technology evolution for the last two decades, have caused the scientific community to invest massively in software development and to redesign legacy codes at the dawn of the exascale era. Single-node hardware specification of current and future supercomputers is as follows: hundreds of cores heading towards a thousand and a memory subsystem with a deep hierarchy getting deeper and more heterogeneous. The performance on shared-memory computational nodes is no longer solely governed by the cores’ clock frequency, especially for memory-bound applications. Their versatility in performing floating-point operations (FLOPs) on locally cached data in strong scaling mode of operation is the new norm for high performance software development. The aforementioned trends, i.e., extreme thread concurrency for parallel performance and data locality for traffic reduction, portend the end of the ubiquitous bulk synchronous programming (BSP) model.

For decades, many applications in the oil and gas industry relied on solving the acoustic wave equation using a Finite-Difference Time-Domain (FDTD) scheme (Levander 1988; Virieux 1986). Such computational kernels, also referred to as stencil computations in this paper, constitute one of the main computational phases during forward and backward modeling in Reverse Time Migration (RTM) (Baysal et al. 1983; Chan and McMechan 1990) or Full-Waveform Inversion (FWI) (Virieux and Operto 2009). In fact, these kernels have enjoyed a free lunch throughout the emergence of previous hardware generations. Their BSP model, often expressed with the OpenMP parallel for pragma, in conjunction with spatial blocking optimizations, has made them amenable to various hardware transformations (e.g., SIMD vectorizations), while ensuring a high performance efficiency. However, the BSP model for stencil computations has started showing performance degradation on current x86 hardware systems, due to a number of factors, including dynamic frequency scaling, non-uniform memory access, memory bandwidth starvation, load imbalance, and limited data locality. As a result, the cores lagging behind may result

¹Extreme Computing Research Center, King Abdullah University of Science and Technology, Jeddah, KSA
²Exploration and Petroleum Engineering Center - Advanced Research Center, Saudi Aramco, Dhahran, KSA

Corresponding author:
Kadir Akbudak
Email: kadir.akbudak@kaust.edu.sa
The remainder of the paper is as follows. In Section 2, the related work in stencil computations is reviewed and then we highlight our main contributions. Section 3 describes the acoustic wave equation in the context of seismic imaging applications and Section 4 shows its discretization using the FDTD method. Section 5 presents the CPML ABCs, which are the crux of the paper. High performance implementation details are given in Section 6. Section 7 reports performance results and we conclude in Section 8.

2 Prior Work and Current Contributions

Optimizing stencil kernels on various hardware architectures is a well-covered topic in the literature. Performance engineering techniques and compiler-based approaches represent two of the main trends in stencil kernel design for high performance computing. The cache oblivious methodology (Frigo and Strumpen 2005) has been an essential step in enhancing user-productivity for stencil implementations on modern systems. This approach has been applied with success in the context of industrial seismic applications (Etienne et al. 2014; Imbert et al. 2011; Sena et al. 2011). With the development of code generations using compiler technologies (Bondhugula et al. 2008; Grosser et al. 2014a; Henretty et al. 2013; Holewinski et al. 2012; Orozco et al. 2010; Wonnacott and Strout 2013), formulating stencil kernels has been further facilitated. Most of these references rely on spatial blocking with an implementation based on the BSP model. To increase data locality, temporal blocking through tiling techniques (Bandishti et al. 2012; Grosser et al. 2014b; Malas et al. 2015; Orozco and Gao 2009; Strzodka et al. 2011; Wellein et al. 2009; Wonnacott 2000; Yuan et al. 2017; Zhou 2013) has been widely considered using various advanced programming models to favor asynchronous execution. Performance tuning using roofline models (Datta 2009; Etienne et al. 2017; Nguyen et al. 2010; Titarenko and Hildyard 2017) remains an important assessment step for stencil computations to ensure a good utilization of the underlying hardware resources. Some of these efforts have translated into software releases, e.g., (Christen et al. 2011; Malas 2015; Tang et al. 2011). More thorough related work can be found at (Malas et al. 2017; Yuan et al. 2017).

The aforementioned references present a striking similarity. None of them deals with boundary conditions in a systematic way. In fact, most of them do not even support them. And for the few that support them, they use either Dirichlet or periodic boundary conditions. These boundary conditions are usually straightforward to integrate in the execution flow of a broad class of stencil formulations, since they may not require contributions from the neighborhood of the solution grid points located at the domain physical boundary.

In this paper, we consider solving the acoustic wave equation as a proxy for seismic imaging application using explicit time integration based on stencil computations. As such, we employ challenging CPML (Komatitsch and Martin 2007), as it is the most effective ABC for seismic wave modeling. We then integrate it into the MWD-TB asynchronous tiling approach (Malas et al. 2015, 2017) and
strive to maintain its performance enhancements against spatial blocking with the BSP model.

We propose to tackle an open research problem, which consists in consolidating CPML integration with temporal blocking stencil computations, as implemented in the MWD-TB tiling approach. These ABCs are key for simulating subsurface seismic data. Their severe data dependencies impede the benefits of asynchronous executions brought by the MWD-TB tiling mechanism. A tight coupling of CPML and MWD-TB stencil formulations is then necessary to alleviate this performance burden and reinstate the superiority of CPML-enabled MWD-TB tiling approach over spatial blocking with the BSP model. To our knowledge, this is the first practical study, which highlights the consolidation of CPML ABCs with asynchronous temporal blocking stencil computations, in the context of seismic imaging applications.

3 The Acoustic Wave Equation

In this study, we consider the 3D acoustic wave equation. When the density is assumed homogeneous (with \( \rho = 1 \) kg/m\(^3\)), this equation is written as:

\[
\frac{1}{V_P^2(x)} \frac{\partial^2 P(x,t)}{\partial t^2} = \frac{\partial^2 P(x,t)}{\partial x^2} + \frac{\partial^2 P(x,t)}{\partial y^2} + \frac{\partial^2 P(x,t)}{\partial z^2} + \delta(x - x_s)s(t),
\]

where \( V_P \) is the velocity of the P-wave that propagates in the medium and \( P \) is the pressure. \( V_P \) is time-independent and space-dependent and specified for the forward problem we consider in this paper. In the inverse problem that runs loops over the forward problem, \( V_P \) is the desired output. \( P \) depends upon both time and space. In conventional seismic acquisition, receivers (geophones for land and hydrophones for marine surveys) are distributed on the surface and record displacement variations in time. These time series, or seismic traces, contain the interactions of an acoustic wave emitted from a source with geological structures in depth. In Eq. 1, the position of the source is \( x_s \), the source time function is \( s(t) \) and \( \delta \) is the Kronecker delta function to impose a point source. The main challenge in exploration geophysics is to extract the information embedded in the seismic traces to retrieve an image of the subsurface or formations. Therefore, Eq. 1 is found in geophysical wave equation based methods such as RTM (Baysal et al. 1983; Chan and McMechan 1990) or FWI (Virieux and Operto 2009). Eq. 1 is the second-order acoustic wave equation which can be re-written in the first-order as follows:

\[
\frac{1}{V_P^2(x)} \frac{\partial P(x,t)}{\partial t} = \frac{\partial v_x(x,t)}{\partial x} + \frac{\partial v_y(x,t)}{\partial y} + \frac{\partial v_z(x,t)}{\partial z} + \delta(x - x_s)s(t)
\]

where \( v_x, v_y \) and \( v_z \) denote the particle velocities along the three spatial axes. In order to solve Eq. 1 and the system of equations in Eq. 2, initial conditions and boundary conditions are required. Initial conditions are that all unknowns are set to 0 at \( t = 0 \). The boundary conditions will be discussed in Section 5.

It is worth mentioning that Eq. 1 and the system of equations in Eq. 2 are equivalent and one can retrieve Eq. 1 by substituting the last three expressions (velocities) of Eq. 2 into the first expression (pressure). However, the numerical complexities of both formulations are different. Since the system of equations in Eq. 2 contains four unknowns while Eq. 1 contains only one, it is straightforward to infer that the cost of solving the former is higher than the latter. A comparison of the numerical complexity of both equations can be found in (Etienne et al. 2014). If only the pressure wavefield is required then it is advisable to devise a numerical scheme based on Eq. 1 to speed-up the computations. For some applications, velocity unknowns are also required and therefore the first-order acoustic wave equation has to be used. Below we consider only the second-order acoustic wave equation.

4 The Finite-Difference Time-Domain Method

4.1 Background

The FDTD method is the most popular numerical method within exploration geophysics for modeling seismic data. Its large success mainly comes from two factors: its ease of implementation and its computational efficiency. FDTD relies on evaluation of the wave equation on points generally distributed on a Cartesian grid. The FDTD method is largely used for geophysical applications that require an accurate solution of the wave equation such as RTM or FWI.

FDTD requires discretization of Eq. 1 or the system of equations in Eq. 2 on regular Cartesian grids both in time and space. We use the notation \( u_{i,j,k}^n \), where \( u \) denotes either \( P, v_x, v_y \) or \( v_z \). \( n \) is the time index, and \( i,j,k \) represent the spatial indices. The accuracy of the numerical scheme depends on the considered stencil, defined by the number of grid points used to approximate the derivatives along each Cartesian direction. The computation of the first-order derivative along the index \( i \) with a centered finite-difference scheme is given by

\[
\frac{\partial u_{i,j,k}^n}{\partial x} \approx \sum_{l=1}^{m} a_l u_{i+l,j,k}^n - u_{i-l,j,k}^n, \quad (3)
\]

and for the second-order derivative

\[
\frac{\partial^2 u_{i,j,k}^n}{\partial x^2} \approx b_l u_{i,j,k}^n + \sum_{l=1}^{m} b_l u_{i+l,j,k}^n - u_{i-l,j,k}^n, \quad (4)
\]

In Eqs. 3 and 4, \( h \) denotes the distance between grid points. The order of accuracy of the method is \( 2m \), where \( m \) is the number of grid points in each direction of the stencil. The coefficients \( a_l \) and \( b_l \) can be determined following (Fornberg 1988).

We can recognize the similarity between the \( i,j,k \) notation introduced above and some programming languages like...
Algorithm 1 The Spatial Blocking (SB) algorithm.

1: procedure SB($P$, $nt$, $lx$, $ly$, $lz$, $bsx$, $bsy$, $th$)
2:     $n$ ← $nt$; number of time steps
3:     $l$ ← $lx$, $ly$, $lz$; length of X, Y and Z dimensions
4:     $bs$ ← $bsx$, $bsy$; block size for X and Y dimensions
5:     $th$ ← $th$; number of threads
6:     for $n$ ← 1 to $nt$ do
7:         parallel for $x$ ← 1 to $lx$ step $bsx$ do
8:             parallel for $y$ ← 1 to $ly$ step $bsy$ do
9:                 for $i$ ← $(x - 1)$ $bsx$ to $x$ $bsx$ do
10:                    for $j$ ← $(y - 1)$ $bsy$ to $y$ $bsy$ do
11:                        for $k$ ← 0 to $lz$ − 1 do
12:                           update $P[i][j][k]$

Fortran or C/C++ that support array structures. Hence, the implementation of FDTD is straightforward in such languages. In this work, we focus on the well established 8th-order in space and 2nd-order in time FDTD scheme. We choose this numerical scheme because it offers a good compromise between accuracy and computational cost. The discrete system relative to Eq. 1 can be expressed as:

$$\frac{P_{i,j,k}^{n+1} - 2P_{i,j,k}^n + P_{i,j,k}^{n-1}}{\Delta t^2} V_{i,j,k}^{\Delta x} [O_{i,j,k}^n(P_{i,j,k}^n + O_{i,j,k}^n(P_{i,j,k}^n + O_{i,j,k}^n(P_{i,j,k}^n)),$$

where $\Delta t$ denotes the temporal sampling and $O_{i,j,k}^n$ the 8th order spatial operator to evaluate the spatial second derivative along index $i$ as given in Eq. 4. Discretized system in Eq. 5 is used to advance the solution in time. From an algorithmic point of view, the FDTD algorithm requires the storage of three 3D-arrays: two arrays to store the pressure at current and previous time steps ($P_{i,j,k}^n$ and $P_{i,j,k}^{n-1}$) and one array for the medium velocity $V_{i,j,k}$. It is not necessary to keep all the time samples since at each iteration, the previous time step is overwritten by the new one $P_{i,j,k}^{n+1}$ following the expression in Eq. 5. If we consider a grid size $N \times N \times N$ and $N_t$ time steps, then the FDTD numerical (without boundary conditions) is given by $C(n) \times N^3 \times N_t$, where $C(m)$ is the number of FLOPs per grid point considering an $m^{th}$ spatial order. For an 8th order in space, we get $C(8) = 48$.

4.2 Spatial Blocking

A simple FDTD implementation would sequentially access all grid points via three nested loops over indices $i$, $j$, $k$ and perform computations following the expression in Eq. 5. To enable parallelism on multi-core architectures, one may include OpenMP directives to parallelize these loops and take advantage of the shared memory. On such architectures, it is beneficial to implement cache blocking to work around memory bandwidth limitations and to increase performance. Algorithm 1 presents an implementation of the Spatial Blocking (SB) scheme. The concept of spatial blocking is also illustrated in Figure 1. The aim is to divide the computational grid into subdomains that can fit into the CPU cache (the last level cache on chip is often sufficient when it is large enough) for efficient data reuse. Each thread is then associated with a subdomain characterized by its size along the three directions. The algorithm contains two nested parallel loops over the blocks (lines 7 and 8), and three nested loops over the grid points $i$, $j$, $k$ of the block (lines 9-12). In our implementation, the inner index (memory is contiguous along this index) is $k$, followed by indices $j$ and $i$. In the C++ language, $P_{i,j,k}$ is then translated into $P[i][j][k]$. To provide a map between the computing indices and the spatial dimensions, we adopt the following convention: index $i$ (slow index) corresponds to the X-axis, index $j$ to the Y-axis, and index $k$ (fast index) to the Z-axis.

4.3 Temporal Blocking

As highlighted in the literature from Section 2, temporal blocking (TB) enables FDTD reference schemes with spatial blocking to further improve their performance throughput. TB allows immediate reuse of the already, locally-cached data from spatial decomposition, before getting evicted from the current cache level. Besides a high data reuse, TB has profound implications on the granularity of the computation as well as in the programming model. Indeed, fine-grained computation becomes paramount to maintain freshly computed data solutions in cache. Given the neighboring data dependencies implied by Eq. 1, TB breaks down the entire time domain integration into diamond shapes to ensure data dependencies are not violated. A single diamond then represents the building block for all the stencil computations of the FDTD method. Herein, we use the Multicore-optimized Wavefront Diamond TB (MWD-TB) approach (Malas et al. 2015, 2017), as implemented in the software library Girih (Malas 2015).

Algorithm 2 presents a very high level sketch of the MWD-TB approach. In this approach, the main driver...
Algorithm 2 The Multicore-optimized Wavefront Diamond Time Blocking (MWD-TB) algorithm.

1: procedure MWD-TB-MAIN($P$, $nt$, $lx$, $ly$, $lz$, $th$, $th_x$, $nwf$, $th_{nwf}$)
2:     ▶ $nt$: number of time steps
3:     ▶ $l\{x,y,z\}$: length of X, Y and Z dimensions
4:     ▶ $th$: total number of threads
5:     ▶ $th_x$: number of threads in $x$ dimension
6:     ▶ $nwf$: size of wavefront block
7:     ▶ $th_{nwf}$: size of wavefront per thread
8:     in parallel with $th/th_x$ threads do
9:         for each block blk in Y dimension do
10:            if dependent blocks are finished then
11:                Update $P$ to setup the wavefront in X using SB
12:            MWD-TB$(P)$ ▶ process blk and proceed in time
13:        Update $P$ to complete the wavefront in X using SB
14:     ▶ Processes one diamond-shaped block in time and Y dimension domain.
15: procedure MWD-TB$(P$, $tb$, $te$, $xb$, $xe$, $yb$, $ye$, $zb$, $ze$, $th_x$, $nwf$, $th_{nwf}$)
16:     ▶ $tb$, $te$: beginning&end of time loop
17:     ▶ $\{x,y,z\}\{b,e\}$: beginning&end indices of $x$, $y$ and $z$
18:     ▶ $th_x$: number of threads in $x$ dimension
19:     ▶ $nwf$: size of wavefront block
20:     ▶ $th_{nwf}$: size of wavefront per thread
21:     in parallel with $th_x$ threads do
22:         for $xi \leftarrow xb$ to $xe$ by $nwf$ do ▶ blocking in wavefront
23:             for $nt \leftarrow tb$ to $te$ do ▶ time loop
24:                 for $i \leftarrow xi$ to $xi + nwf$ do
25:                     if $(i/th_{nwf})\%\ th_x = tid_x$ then
26:                         for $j \leftarrow yb$ to $ye$ do
27:                             for $k \leftarrow zb$ to $ze$ do
28:                                 update $P[i][j][k]$

procedure MWD-TB-MAIN starting at line 1 of Algorithm 2 marshals the diamond-based computational tasks defined as the MWD-TB procedure starting at line 14 of Algorithm 2 using the asynchronous task-based programming model associated with the OpenMP dynamic runtime system.

In the subsequent figures, we omit the Z dimension since it carries the fastest index, similar to the SB approach. Before looking into 3D representation of the MWD-TB approach, Figure 2 showcases its 2D representation associating the time-Y axes (top) for the diamond interlocking and the time-X axes (bottom) for the wavefront thread parallelism. In addition, the former displays the data dependencies between diamond and the latter reveals the pipeline of the wavefront. Figure 3 depicts the 3D representation of a $128 \times 128 \times 128$ domain integrated over 144 time steps. This figure highlights the diamond-based time/space domain decomposition in 3D, along time and Y dimension, processed by four thread groups of eight threads each. Each thread group executes concurrently each diamond, while the OpenMP runtime system orchestrates the inter-diamond parallelism.

Figure 2. We show the time-Y (top) and time-X (bottom) views of interlocking diamonds and their corresponding data dependencies. In the time-Y view, the thread parallelism can simply be expressed by splitting the diamond (shown with two colors/threads) along the Y space dimension, without any data dependencies between sub-diamonds. However, the loop-carried data dependencies exist along the time dimension. Once the stencil computation within a diamond finishes, the computations of up to two neighboring diamonds may be launched. In the time-X view, each subdomains is processed by multiple threads using wavefront parallelism. The data dependencies are not only along the time dimension but also along the X space dimension (shown with four colors/threads). We use SB stencil computations to start and finalize the pipeline of the wavefront in the time-X view.

Figure 3. The full complexity view of the MWD-TB approach showing how the diamonds cover space and time. The size of the whole spatial domain is $128 \times 128 \times 128$ and the number of time steps is 144. Different colors represent different threads that are dynamically assigned to subdomains of the whole time-space domain. The spatial domain is partitioned along Y axis. Each diamond-shaped block is scheduled dynamically by the MWD-TB-MAIN procedure declared at line 1 of Algorithm 2 upon satisfaction of data dependencies and processed by the MWD-TB procedure declared at line 14 of Algorithm 2. The subdomains at corners are processed with SB in order to setup and finalize the wavefronts as depicted in lines 11 and 13 of Algorithm 2. The intra-diamond parallelism can be seen along the X axis.
The 3D isotropic acoustic wave equation is rewritten formulation and notations described in (Pasalic and McGarry Martin 2007). In this work, we adopt the same CPML Matched Layer (CPML) formulation (Komatitsch and Martin 2007).

A major concern in seismic modeling is the ability to evaluate the first and second spatial derivative along index $k$. The 3D isotropic acoustic wave equation is rewritten in a stretched coordinate space as:

$$\frac{1}{c^2} \frac{\partial^2 P}{\partial t^2} = \frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} + \frac{\partial^2 P}{\partial z^2} + \frac{\partial \Psi_y}{\partial y} + \frac{\partial \Psi_x}{\partial x} + \zeta_x + \zeta_y + \zeta_z,$$

where $\Psi_x$, $\Psi_y$, $\zeta_x$, $\zeta_y$, $\zeta_z$ are auxiliary variables with time evolution for the Z dimension (without loss of generality) given by:

$$\begin{align*}
\Psi_x^n &= \alpha_z \Psi_x^{n-1} + \beta_z \left( \frac{\partial P}{\partial x} \right)^n, \\
\zeta_x^n &= \alpha_z \zeta_x^{n-1} + \beta_z \left[ \frac{\partial \Psi_y}{\partial x} \right]^n + \left( \frac{\partial \Psi_x}{\partial x} \right)^n.
\end{align*}$$

$$\alpha_z$$ and $\beta_z$ are the time-independent and space-dependent CPML damping coefficients for the Z dimension (Pasalic and McGarry 2010).

After discretization on a staggered grid (Yang et al. 2014), the system to update the pressure wavefield with the Z CPML contribution becomes:

$$\begin{align*}
\Psi_x^{n+1/2}_{i,j,k} &= \alpha_k \Psi_x^{n-1/2}_{i,j,k} + \beta_k O_k^8(P^n_{i,j,k}); \\
\zeta_x^{n+1}_{i,j,k} &= \alpha_k \zeta_x^{n-1}_{i,j,k} + \beta_k (O_k^8(P^n_{i,j,k}) + O_k^8(\Psi_x^{n+1/2}_{i,j,k})); \\
P^{n+1}_{i,j,k} &= P^{n+1}_{i,j,k} + \Delta t^2 \left[ \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right] \psi^{n+1/2}_{i,j,k} + \zeta_x^{n+1}_{i,j,k},
\end{align*}$$

where $O_k^8$ and $O_k^8$ denote the 8th order spatial operator to evaluate the first and second spatial derivative along index $k$, respectively.
The CPML formulation in Eq. 7 damps the seismic wavefield in the absorbing layers via the introduction of additional variables (called memory variables) into the original wave equation. A total of six convolution memory variables $\Psi$ and $\zeta$ (two for each of the three dimensions) and six coefficients $\alpha$ and $\beta$ (three for each) are required to implement the CPML for the system of equations in Eq. 6. Even though $\alpha$ and $\beta$ are not time dependent, we precompute and store these coefficients. Therefore, twelve additional arrays need to be allocated. For large 3D models, this might lead to memory issues. In order to reduce the required memory, we allocate the memory variables only for the grid points located in the absorbing layers.

5.2 CPML Pseudo-Code

Algorithm 3 presents the first and second derivative formulations for the Z dimension. These derivatives and their respective formulations emphasize the complexity of the data dependencies involved during CPML calculations for all dimensions without loss of generality. This algorithm also presents another procedure for computing the first derivative, DZFIXEDSIZEARRAY($A$), which computes the derivative on a fixed-size array $A$, used as a buffer to eventually weaken data dependencies. This procedure is also used in the subsequent algorithms.

Indeed, this first-order CPML formulation reveals the leap-frog staggered grid with two-levels of imbricated data dependencies, i.e., $\Psi$ and $\zeta$, before eventually adding the final contribution to the lattice updates located at $P^{n+1}$, as shown in Eq. 8. The first-order derivative translates into a discretization involving eight lattices of $P$ and $\Psi$ to update the transient lattices located at $P$ and $P^{n+1}$, respectively. These severe data dependencies may impede parallel performance, due to the resulting synchronization points.

Figure 5 shows snapshots of the wave propagation during the time integration. Once the wave reaches the physical boundary domain, the energy is absorbed thanks to the CPML boundary conditions. The amplitude of reflections (order of $10^{-3}$) at time step 1808 is much lower than the wave before it hits the boundaries of the physical domain at time step 1408.

5.3 CPML Data Dependencies

The first challenge when dealing with CPML is the severe data dependencies imposed on the MWD-TB approach. Even though CPML computations affect only lattice updates located on the physical domain boundary, they may ultimately impact the time integration of the lattices located in the inner domain. For simplicity purposes and without loss of generality, we explain the CPML data dependencies along the Z dimension.

Figure 6 displays the CPML two-level imbricated data dependencies located on the outer domain: (1) the initial pressure component $P^{n-1}$ is obtained by applying once the standard MWD-TB across all lattices of the inner and outer domains, and (2) $\Psi^{n-1/2}$ is then formed successively from the first space derivative of the pressure $P^{n-1}$, but only for the outer domain lattices, using the $P^{n-1}$ values demarcated by the two dashed lines. The final components of $\zeta^n$ and $P^n$ for lattices located at the domain boundary are then calculated from $\Psi^{n-1/2}$ and the second space derivative of $P^{n-1}$, respectively. In these calculations, it is clear that $\Psi$ is known at time $n - 1/2$ and such calculations at half time steps constitute the main problem. Figure 6 visualizes this sequence of computations via a red point, which corresponds to the final lattice computed from $\Psi^{n-1/2}$ and $P^{n-1}$. These CPML two-level imbricated data dependencies have to be satisfied before carrying on the time integration. For an asynchronous MWD-TB scheme deployed at the core of the
stencil computation, this appears to be a major showstopper. The MWD-TB approach can not proceed in time for the physical boundary lattices until and unless the CPML contributions have been applied and interleaved between subsequent time steps. This Write After Read (WAR) type of data hazard is also well-known in the compiler community for limiting instruction-level parallelism. This issue related to WAR data hazards is not as critical for SB as for TB, since SB has to synchronize at each time step regardless.

5.4 CPML Load Imbalance

The second challenge resides in the load imbalance brought by the CPML integration into the overall MWD-TB approach. Diamonds containing lattices from the outer domain have to perform more floating-point operations, as opposed to diamonds containing only lattices from the inner domain. This may create situations, where groups of threads have to wait until outer domain diamonds have finished. The asynchronous execution of the MWD-TB may mitigate this load imbalance, thanks to its task-based programming model. In fact, if enough fast inner domain computations are available to overlap the slow outer domain computations, the MWD-TB approach may be amenable to efficiently pursue the critical path, while reducing the overall idle time. Here again, the load imbalance issue is not as decisive for TB as for SB. In SB, the successive computational phases of inner and outer domain lattices require all threads to participate in a holistic way. As a result, the workloads at each time step are always equally distributed among threads, at the cost of global synchronization between each phase in addition to limited data reuse.

6 Implementation Details

In this section, we explain how CPML may be consolidated with MWD-TB by introducing three incremental optimization techniques and providing their corresponding implementation details.

6.1 Fusing MWD-TB with CPML Loops

The first, natural optimization technique consists in fusing the CPML calculations into the MWD-TB inner loops. In fact, this is not an optimization technique but rather a mandatory code restructuring to ensure the integrity of CPML data dependencies and numerical accuracy. This permits to maintain ubiquitous asynchronous execution but with two major drawbacks. Algorithm 4 describes this reference implementation with the pipelined wavefront parallelism across the X dimension. Moreover, this baseline algorithm shows how all CMPL ABCs are calculated at the innermost loop level. For instance, in line 10 of Algorithm 4, an if conditional is now inserted in the innermost loop of MWD-TB for the Z-boundary region, right after the update of the inner domain lattices. This conditional statement permits to decide whether this updated lattice belongs to the physical domain and, in that case, should receive in addition the CPML contribution. This conditional statement is then repeated for the other dimensions. Unfortunately, these additional conditional statements disable the compiler SIMD vectorization and dramatically slow down the performance throughput. The second drawback corresponds to redundant operations, which further exacerbate the overall performance. Figure 7 illustrates these extra operations for the CPML region along the Z dimension. A single red point in $P^n$ requires a radius of eight red points to be computed in $\Psi^{n-1/2}$, as explained in the first-order CPML formulation (see Section 5.1). For the subsequent red points in $P^n$, the corresponding red points in $\Psi^{n-1/2}$ have to be recalculated, including those which may have been computed previously. This pattern continues until all red points in $P^n$ have been updated by their CPML contributions. Algorithm 5 summarizes the challenges related with redundant operations, using CPML.$Z[+()]$ as a proxy for the other CPML dimensions. For MWD-TB to be competitive again, it is paramount to restore SIMD vectorizations and remove these extra operations.

6.2 Re-enabling SIMD Vectorizations

To enable compiler SIMD vectorizations of the innermost loop, we instead remove the if conditional statements from Algorithm 4 and replace them with CPML loops at the appropriate outer loops’ levels. Algorithm 6 describes the pseudo-code of the SIMD-enabled MWD-TB. Since these various conditional statements are done at the outer levels (e.g., see lines 13, 18 and 25 for the Z, Y, and X CPML boundary regions, respectively), the performance impact of the conditional becomes limited. Indeed, the compiler is now able to vectorize the inner domain lattice updates, which account for most of the physical domain. It is true that there is a performance penalty for not vectorizing the CPML lattice updates due to the imbricated data dependencies (as shown in the pseudo-codes of Algorithm 3), but it is commensurate to the ratio between the number of CPML lattices on each dimension and the actual 3D domain size. We refer to this optimization as opt1 in the subsequent sections.

6.3 Removing Redundant Computations

To remove the redundant computations, it is important first to note that between two subsequent red points in $P^n$, only
Algorithm 4 The baseline algorithm ([noopt]).

1: procedure MWD-TB-[noopt](P, th, te, xb, xe, yb, ye, zb, ze, th_x, nwf, th_xw_f)
2: in parallel with th_x threads do
3: for xi ← xb to xe by nwf do \(\triangleright \) blocking in wavefront
4: for ni ← th to te do \(\triangleright \) time loop
5: for i ← xi to xi + nwf do
6: if ((i/th_xw_f) \% th_x == tid_x) then
7: for j ← yb to ye do
8: for k ← zb to ze do
9: update \(P[i][j][k]\)
10: if lattice is in Z- boundary region then
11: \[P[i][j][k] ← P[i][j][k] + CPML-Z()\]
12: if lattice is in Z+ boundary region then
13: \[P[i][j][k] ← P[i][j][k] + CPML-Z()\]
14: if lattice is in Y- boundary region then
15: \[P[i][j][k] ← P[i][j][k] + CPML-Y()\]
16: if lattice is in Y+ boundary region then
17: \[P[i][j][k] ← P[i][j][k] + CPML-Y()\]
18: if lattice is in X- boundary region then
19: \[P[i][j][k] ← P[i][j][k] + CPML-X()\]
20: if lattice is X+ boundary region then
21: \[P[i][j][k] ← P[i][j][k] + CPML-X()\]
22: return

Algorithm 5 Definition of CPML-Z+() ([noopt]).

1: procedure CPML-Z+(P, \(\Psi\), \(\zeta\), \(\alpha\), \(\beta\), i, j, k)
2: \(\triangleright \) t is a temporary array for storing \(\Psi\)
3: \(ta[0] \leftarrow \alpha - \Psi_{i,j,k-3} + \beta \Delta z(P, i, j, k - 3)\)
4: \(ta[1] \leftarrow \alpha - \Psi_{i,j,k-2} + \beta \Delta z(P, i, j, k - 2)\)
5: \(ta[2] \leftarrow \alpha - \Psi_{i,j,k-1} + \beta \Delta z(P, i, j, k - 1)\)
6: \(ta[3] \leftarrow \alpha - \Psi_{i,j,k} + \beta \Delta z(P, i, j, k)\)
7: \(ta[4] \leftarrow \alpha - \Psi_{i,j,k+1} + \beta \Delta z(P, i, j, k + 1)\)
8: \(ta[5] \leftarrow \alpha - \Psi_{i,j,k+2} + \beta \Delta z(P, i, j, k + 2)\)
9: \(ta[6] \leftarrow \alpha - \Psi_{i,j,k+3} + \beta \Delta z(P, i, j, k + 3)\)
10: \(ta[7] \leftarrow \alpha - \Psi_{i,j,k+4} + \beta \Delta z(P, i, j, k + 4)\)
11: \(dz \leftarrow DZFIXEDSIZEARRAY(ta)\)
12: \(\zeta_{i,j,k} \leftarrow \alpha \zeta_{i,j,k} + \beta (D2z(P, i, j, k) + dz)\)
13: \(\Psi_{i,j,k-4} \leftarrow ta[0]\)
14: if last iteration then
15: \(\Psi_{i,j,k-3} \leftarrow ta[1]\)
16: \(\Psi_{i,j,k-2} \leftarrow ta[2]\)
17: \(\Psi_{i,j,k-1} \leftarrow ta[3]\)
18: \(\Psi_{i,j,k} \leftarrow ta[4]\)
19: return \(\Delta t^2 V_j^2 dz + \zeta_{i,j,k}\)

Figure 8. 1D sliding window mechanism to remove redundant computations [opt2]. A temporary array stores \(\Psi\) values recently calculated at half time steps. Only one new \(\Psi\) value calculation happens for each update to \(\zeta^n\) and \(P^n\).

one extra red point in \(\Psi^{n-1/2}\) needs to be recalculated, since previous calculations of red points can be reused. We rely on a sliding window technique that contains the transient lattice updates of \(\Psi^{n-1/2}\) and calculate only a new \(\Psi^{n-1/2}\) lattice every single time we compute a new lattice in \(P^n\), instead of computing all eight lattices of \(\Psi^{n-1/2}\) again. Figure 8 demonstrates this sliding window mechanism for the Z CPML, which ensures an on-demand mechanism by calculating new \(\Psi^{n-1/2}\) lattice updates only when it is necessary. Algorithm 7 highlights this pipelining mechanism, which has an initial overhead for calculating all eight lattices of \(\Psi^{n-1/2}\) in the sliding window, but then most of the transient \(\Psi^{n-1/2}\) lattice updates can be directly cached from the buffer hosting the 1D sliding window. We refer to this optimization as opt2 in the subsequent sections.

Although more challenging, these optimizations are also extended to the X and Y dimensions, which operate on expensive strided memory accesses. A 2D sliding window is then used instead of a 1D sliding window for the Y CPML region to host the necessary housekeeping and avoid redundant operations by the same token (i.e., from line 19 to line 24 of Algorithm 6). Figure 9 pictures the pipelining mechanism and shows how the 2D sliding window moves along the Y axis. Due to the non-contiguous memory.
Algorithm 7 Redefinition of CPML Z+() (opt2).

1: procedure CPML Z+() (opt2)
2: $\triangleright w$ is a sliding window for storing $\Psi$
3: if first iteration then
4: $\implies w[0] \leftarrow \alpha_{k-3} \Psi_{i,j,k-3} + \beta_{k-3} Dz(P, i, j, k - 3)$
5: $\implies w[1] \leftarrow \alpha_{k-2} \Psi_{i,j,k-2} + \beta_{k-2} Dz(P, i, j, k - 2)$
6: $\implies w[2] \leftarrow \alpha_{k-1} \Psi_{i,j,k-1} + \beta_{k-1} Dz(P, i, j, k - 1)$
7: $\implies w[3] \leftarrow \alpha_{k} \Psi_{i,j,k} + \beta_{k} Dz(P, i, j, k)$
8: $\implies w[4] \leftarrow \alpha_{k+1} \Psi_{i,j,k+1} + \beta_{k+1} Dz(P, i, j, k + 1)$
9: $\implies w[5] \leftarrow \alpha_{k+2} \Psi_{i,j,k+2} + \beta_{k+2} Dz(P, i, j, k + 2)$
10: $\implies w[6] \leftarrow \alpha_{k+3} \Psi_{i,j,k+3} + \beta_{k+3} Dz(P, i, j, k + 3)$
11: $\implies w[7] \leftarrow \alpha_{k+4} \Psi_{i,j,k+4} + \beta_{k+4} Dz(P, i, j, k + 4)$
12: else
13: shift values in $w$ to left by one
14: $\implies w[7] \leftarrow \alpha_{k+4} \Psi_{i,j,k+4} + \beta_{k+4} Dz(P, i, j, k + 4)$
15: $\implies dz \leftarrow DZFIXEDSIZEARRAY(w)$
16: $\implies \zeta_{i,j,k} \leftarrow \alpha_{k} \zeta_{i,j,k} + \beta_{k} (Dz(P, i, j, k) + dz)$
17: $\implies \Psi_{i,j,k-4} \leftarrow w[0]$
18: if last iteration then
19: $\implies \Psi_{i,j,k-3} \leftarrow w[1]$
20: $\implies \Psi_{i,j,k-2} \leftarrow w[2]$
21: $\implies \Psi_{i,j,k-1} \leftarrow w[3]$
22: $\implies \Psi_{i,j,k} \leftarrow w[4]$
23: return $\nabla^2 V_{i,j,k}(dz + \zeta_{i,j,k})$

acrosses along the Y dimension, the window can only slide after being filled-out with a surface of transient lattice contributions of $\Psi^{n-1/2}$, required to eventually update the $P^n$ lattices with the final Y CPML individual contributions. For the X CPML region, which corresponds to the slowest index, the window translates into a 3D sliding object because of an even larger stride, i.e., from line 26 to line 35 of Algorithm 6. To overcome the memory footprint overhead of the 3D object, we swap the X and Y loops for the CPML treatment along the X dimension. This loop swapping technique reduces the stride and enables the overall handling of the X CPML computations to fall back to a 2D sliding window, similar to the Y CPML region.

![Figure 9. 2D sliding window mechanism for the Y CPML region (similar for the X CPML region).](image)

7 Experimental Results

This section describes the performance benchmarking of CPML-enabled MWD-TB on various hardware configurations and application settings.

7.1 Experimental Settings

We conduct a performance evaluation of SB and MWD-TB stencil computations for solving the 3D acoustic wave equation using CPML as ABCs. Two systems are used for performance comparisons between both stencil formulations. The first system has dual-socket 16-core Intel Haswell E5-2698v3 CPUs with AVX256 support. Each Intel Haswell CPU has three levels of cache: L1 (32KB) and L2 (256KB) caches private to each core and a 40MB L3 cache shared per socket. The second system has dual-socket 28-core Intel Skylake Platinum 8176 CPUs with AVX512 support. Each Intel Skylake CPU has three levels of cache: L1 (32KB) and L2 (1MB) caches private to each core and a 38.5MB L3 cache shared per socket. Sub-NUMA Clustering (SNC) is turned off. Both stencil codes are compiled withicc 18.0.1 using -xHost and -o3 flags and rely on the OpenMP programming model. In addition, MWD-TB employs an OpenMP-based dynamic runtime system to schedule the various diamonds as computational tasks presented as MWD-TB() procedure at line 14 of Algorithm 2.

In the experiments, we use three different grid sizes: $520 \times 520 \times 520, 1032 \times 1032 \times 520,$ and $2056 \times 2056 \times 1032$. The width of CPML is set to ten lattices. The percentages of all boundary points in the six layers are 11.1%, 7.5%, and 3.8% for the aforementioned grid sizes, respectively. The number of time steps is set to 2000. The environment variable KMP_AFFINITY=compact,granularity=fine is set for both SB for the largest thread counts on each system. The command numactl --interleave=all is used for MWD-TB since the scheduling algorithm is not yet aware of the NUMA locality, while assigning diamond tiles to thread groups. Single precision arithmetic is used for both codes, as this corresponds to the standard precision arithmetic for seismic imaging applications.

7.2 Comparison of SB and MWD-TB w/o CPML

In this section, we present a detailed performance comparison of SB and MWD-TB excluding the boundary computations. The Skylake system is selected as the test system and small grid of size $520 \times 520 \times 520$ is used. We first assess the performance of the SB scheme, which is used as our baseline method. Similar to (Malas et al. 2015), we employ the SB scheme as well as the 8th-order in space and 2nd-order in time for the stencil computation. This stencil specification is one of the typical configurations for oil and gas industry. We use the analysis based on the roofline model, as explained in Section 3.3.2 of (Malas et al. 2015).

At the $n$th time step of Algorithm 1, we update one entry $P^n_{i,j,k}$ of the pressure array and we read two entries from the pressure array $P^n_{i,j,k}$ and the velocity model $V_q$, as explained at the end of Section 4.1. Hence, the minimum code balance for single precision is 16 bytes/lattice update. The peak DRAM bandwidth reported by the STREAM benchmark (McCalpin 1995) is 175 GB/s (giga bytes per second) on the Skylake system.
Therefore, the predicted upper bound on the SB performance is $175 \text{ GB/s} / \text{byte lattice update} = 10.94 \text{ GPoint/s}$ (the number of giga lattice updates per second). The maximum sustained performance of the SB approach without CPML is 9.54 GPoint/s. This is close to the predicted upper bound of the performance model described in Section 7.2. This demonstrates the validity of our baseline that is used during our performance comparisons in the subsequent figures.

We compare the performances of SB and MWD-TB in terms of the amount of data transferred through the memory hierarchy of the Skylake system. We use likwid-perfctr program provided by likwid (Treibig et al. 2010) and present the data volume (in terms of GB) reported by the L2, L3 and MEM performance groups in Table 1. As seen in the table, MWD-TB incurs significantly less DRAM and L3 traffics than SB. These experimental findings can be attributed to the fact that MWD-TB effectively exploits temporal locality.

Figure 10 shows the hierarchical roofline model (Williams et al. 2009) of SB and MWD-TB on the Skylake system. The roofs for the sustained peak performances of each memory level and the floating point unit are obtained by the Intel Advisor tool and the measurements for the codes are performed by likwid. In this figure, the X axis shows the arithmetic intensity in terms of number of FLOPs per byte (FLOP/Byte) and the Y axis shows the performance in terms of giga FLOPs per second (GFLOP/s). Both axes are in logarithmic scale. Four different traffic measurements for each of SB and MWD-TB schemes are reported. Beside the three performance groups used for Table 1, the DATA performance group is used to approximate the L1 cache traffic (Yang 2018). These four performance groups report the L1, L2, L3 and DRAM bandwidths consumed by the target application. Table 2 explicitly shows these bandwidths in terms of GB/s for the SB and MWD-TB schemes. As seen in Figure 10 and Table 2, The consumed DRAM bandwidth of SB is closer to the sustained peak than MWD-TB. However, the L3 and DRAM bandwidths of SB are closer to each other compared to other levels. This may indicate that SB cannot exploit data locality in L3 so it incurs more DRAM traffic. In case of MWD-TB, the gap between the L3 and DRAM bandwidths are relatively larger than SB, confirming that data locality for L3 is more exploited and, therefore, less DRAM traffic is incurred.

### 7.3 Numerical Validation

Before looking into performance metrics, we assess the numerical accuracy of MWD-TB over SB. We assess the integration of CPML into MWD-TB by qualitatively looking at the lattice-wise difference in magnitude for both grids. As a sanity check, MWD-TB achieves bitwise equality with respect to SB for the values of the final grids in case of both codes are compiled with `-fp-model strict` flag. This compiler flag enforces a strict order of operations involved in the main stencil operator, hence the similar bitwise accuracy result. However, when this compiler flag is turned off, the difference in rounding errors comes to effect.

In particular, both stencil codes are now compiled with `-xHost` and `-O3` to enable the generation of instructions for the highest SIMD-like instruction set available and to aggressively optimize the resulting code, respectively. We calculate then the mean absolute error (MAE) between the two grids and obtain $2.2e-08$, which is the same order than 32-bit machine precision accuracy. Figure 11 shows the corresponding absorbed energy within the domain solution across time steps for both approaches inside the computational grid of size $520 \times 520 \times 520$ for SB and MWD-TB throughout 2500 time steps. The energy is the sum of the squares of all amplitudes still in the wavefield, which may be looked at an L2 norm. The seismic energy increases as the source is injected. It maintains a plateau until the wave propagation reaches the boundaries and eventually the energy gets damped. The energy evolutionary trend is similar for both SB and MWD-TB approaches, which again highlights the numerical robustness of our approach.

### 7.4 Performance Results

Figure 12 shows the performance impact of the three incremental optimizations introduced in Section 6: (1) fusing CPML and MWD-TB loops (baseline), (2) enabling SIMD (opt1) and (3) removing redundant computations with 1D/2D sliding window (opt2), as described in Section 6.1, Section 6.2, and Section 6.3, respectively. In this figure, we report the performance in terms of the number of giga lattice updates per second (Gpoint/s) on the two Intel architectures for the smallest grid size of $520 \times 520 \times 520$. As seen in the figure, the proposed optimizations achieve up to 3.3X improvements.
and 2.6X speedups against the baseline CPML-enabled MWD-TB implementation on the Intel Haswell and Skylake systems, respectively. In particular, the sliding window technique \textbf{(opt2)} contributes for most of the performance improvements depicted in the figure. This performance breakdown demonstrates also the impact of enabling each CPML dimension during the computation. Since the Z and Y dimensions correspond to indices with short memory strides, turning on their CPML computations presents higher performance gain compared to X CPML ones. When all CPML dimensions are enabled, the overall performance drops down, given the larger ratio of CPML lattices over the inner domain points for such a small grid size.

Figure 13 shows the performance comparisons of MWD-TB against SB in terms of Gpoint/s on both considered architectures. Table 3 summarizes the performance assessment from Figure 13 by reporting the performance comparisons and the percentage change of MWD-TB over SB in case of all CPMLs are enabled.

As seen in Figure 13, CPML-enabled MWD-TB outperforms SB and achieves more than 27% and 33% performance speedup on the Intel Haswell and Skylake systems, respectively, especially for medium (1032 × 1032 × 520) and large (2056 × 2056 × 1032) size grids. As seen in Table 3, MWD-TB shows a performance slowdown for the small grid size (520 × 520 × 520) compared to SB on the Intel Haswell system, due to the high ratio of CPML lattices over the inner lattices. However, this performance slowdown does not appear for the Intel Skylake system for which MWD-TB is actually able to outperform SB on the small grid size. This may be explained not only by the larger L2 capacity per core of the Intel Skylake (1MB) over Haswell (256KB) system but also the significantly higher L2 bandwidth achieved by MWD-TB compared to SB on the Intel Skylake system. Nevertheless, the performance gain of MWD-TB over SB shrinks on both systems when enabling CPML boundary conditions, especially for small and medium grid sizes. This implies that the CPML implementation in MWD-TB seems relatively less efficient than in SB. There is a base cost due to the necessary housekeeping for reducing the number of redundant computations introduced for the optimization technique \textbf{opt2}. One possible solution may be to consider applying MWD-TB only for the inner lattices and to fall back to SB for the CPML lattices. This would necessitate to segregate the boundary domain lattices and to confine them within isolated diamonds so that no interruption may be observed for the asynchronous computations of the inner lattices. This hybrid approach may leverage the best of the two techniques and we plan to assess it in future work.

Figure 14 shows the results of performance scalability and efficiency of SB and MWD-TB for the large grid size (2056 × 2056 × 1032) using the Intel Haswell and Skylake systems. The elapsed time of the corresponding sequential implementation is used for computing performance speedup and efficiency metrics. We adopt the best thread affinity strategy of pinning for SB by using $\text{KMP}_-$\text{AFFINITY}=[\text{compact or scatter}]$, as we increase the number of threads. As for MWD-TB, a similar strategy is inherently employed in the diamond tiling technique to ensure the performance benefits of the thread groups approach are not compromised. All in all, MWD-TB achieves higher relative speedups and efficiencies compared to SB on both systems across various number of cores. While SB performance is limited by the memory bandwidth per core, MWD-TB is instead limited by the caches’ bandwidth per core, and therefore, ultimately outperforms the SB implementation.

8 Conclusions and Future Work

We introduce the design and implementation of Convolutional Perfectly Matched Layers (CPMLs) as Absorbing Boundary Conditions (ABCs) to support the solution of the
3D acoustic wave equation using the asynchronous MWD-TB approach. The new CPML-enabled MWD-TB implementation outperforms its SB counterpart up to 33% on large grid sizes, which are of interest for large-scale simulations using production codes. To restore the performance superiority of MWD-TB over SB, we enable SIMD vectorizations
and remove the extra lattice calculations using a 1D/2D sliding window. To our knowledge, this is the first pragmatic approach in enabling CPML challenging workloads in the MWD-TB approach. This opens up new directions for research in the context of both the reverse time migration and full waveform inversion for seismic imaging and inversion applications.

As future work, we would like to investigate three directions to enhance the framework proposed in this paper. The first one consists in supporting different diamond sizes within MWD-TB to mitigate the load imbalance overhead encountered by the dynamic runtime system. The idea is to favor fine-grained diamonds for handling boundary lattices. The second one is to assess other algorithmic formulations of CPML characterized by simpler data dependencies in order to maintain the asynchrony of the stencil computations. The last one is to simply investigate new ABC formulations for MWD-TB. These friendlier formulations should coexist and be more compatible with MWD-TB, without deteriorating its performance superiority.

Acknowledgements

The authors would like to thank the KAUST Supercomputing Laboratory for computing time and Thomas Gruber at Erlangen Regional Computing Center Erlangen, Germany for his assistance in using likwid. K Akbudak and R Abdelkhalak acknowledge the support of Saudi Aramco through KAUST OSR contract #3226.

References


