The Journal of Supercomputing (2021) 77:2416–2457
https://doi.org/10.1007/s11227-020-03352-5

Elastodynamic full waveform inversion on GPUs with time-space tiling and wavefield reconstruction

Ole Edvard Aaker[1] · **Espen Birger Raknes**[1,2] · **Børge Arntsen**[3]

Published online: 17 June 2020

Abstract
Full waveform inversion (FWI) is a procedure used to determine the elastic parameters of the Earth by reducing the misfit between observed elastodynamic wavefields and their numerically modelled counterparts. The numerical solution of the elastodynamic wave equation is computationally expensive, and its performance is typically bandwidth bound. Computing the gradient of the FWI misfit functional adds further complexity as it involves computing the zero-lag cross-correlation of two wavefields propagating in opposite temporal directions. In this paper, we utilize graphics processing units (GPUs) for their high memory bandwidth and combine two principal optimizations in order to compute FWI gradients on large models and for long simulation times. Wavefield reconstruction methods allow efficient gradient computations with minimal memory requirements and interconnection transfers. Time-space tiling techniques permit us to transcend the limited amount of GPU memory while avoiding dramatic slowdowns due to the low interconnection bandwidth. The implementation considers a task-oriented, hybrid usage of explicitly managed and Unified Memory in order to satisfy the requirements. Benchmarks demonstrate that the proposed approach is able to preserve 78–90% of the original performance, when oversubscribing the amount of physical memory available on GPUs. Comparison with existing methods highlights the benefits of the method.

Keywords Graphics processing unit · Wave propagation · Full waveform inversion · Adjoint state method · Tiling

1 Introduction

Full waveform inversion (FWI) is a modern seismic inversion procedure used for quantitative estimation of subsurface parameters, such as wavefield velocities or elastic parameters. The method can be considered a nonlinear optimization

✉ Ole Edvard Aaker
ole.edvard.aaker@akerbp.com

Extended author information available on the last page of the article

method that minimizes the misfit between observed seismic data and the data computed by numerical solution of a suitable wave equation [56]. From its original mathematical framework introduced by Lailly [21] and Tarantola [49], FWI has attracted significant interest within geophysics and has been widely adopted by both academia and the hydrocarbon exploration industry. Several excellent resources on the theory and practical applications of FWI exist in the present literature. The work in [51] presents important aspects such as the conventional FWI workflow, along with successful applications ranging from offshore exploration settings to regional and global scale earthquake seismology. An overview of the main technical aspects of FWI can be found in [56]. Similarly, [9] constitutes an in-depth resource on FWI and numerical modelling of seismic waves. The high computational demand of FWI, induced by solving the wave equation on large grids in three dimensions and for many independent shots, currently limits the resolution and robustness achievable with the present compute capabilities[40, 51].

Popular time-domain solvers, including finite-difference (FD) methods and the spectral element method, are iterative stencil operations with achievable performances typically bounded by the memory bandwidth of the hardware system [57]. The last decade has witnessed a widescale adoption of graphics cards for this type of computations, largely due to their advantage in memory bandwidth. Specifically, the seismic community has seen forward modelling implementations such as Michéa and Komatitsch [24] and Komatitsch et al. [20] for the elastodynamic wave equation. Beyond wave equation simulation capability, many local and certain global optimization methods require computation of the gradient of the misfit functional with respect to the subsurface parameters. Even with the adjoint state method [38], this is computationally challenging to perform with time-domain wave equation solvers, as the gradient is computed by a zero-lag crosscorrelation between two wavefields propagating in opposite directions in time. Implementations on graphics processing units (GPUs) are even more affected than central processing units (CPUs), primarily due to the limited size of device memory and slow interconnection bandwidth to system memory. Streaming entire wavefield states across the interconnection at each timestep can be avoided with the use of wavefield reconstruction methods [41], at the cost of one extra wave equation simulation. Fabien-Ouellet et al. [8] utilized a direct stencil reconstruction method [43, 61] for wavefield reconstruction in elastodynamic finite-difference simulations on GPUs. They furthermore provided an open source implementation for heterogenous compute systems written in OpenCL. As the name suggests, stencil-based reconstruction methods record and inject the direct dependendencies of the stencil scheme. This leads, however, to the undesirable property that the inherent memory requirements scale linearly with the half-length of the stencil operator. In contrast, the memory size and interconnection transfers required by wavefield reconstruction methods formulated from conservation of elastodynamic power [1, 26, 52] are constant across discretization accuracies. For realistic half-lengths of the stencil operator, such as the interval four to sixteen, considering the latter reconstruction method can decrease the memory requirements for wavefield reconstruction by an order of magnitude, compared to stencil-based reconstruction methods.

Even so, the limited memory capacities of graphics cards can present challenges for performing FWI in scenarios such as wide aperture aquisition and high-resolution simulation grids. The low bandwidth of the interconnection to system memory then represents a performance bottleneck. Iteration-space tiling techniques like time-space tiling [59, 60] can alleviate the slowdown in these *out-of-core* scenarios. Time-space tiling increases the temporal locality of values loaded to the device by partitioning the iteration space, consisting of nested loops over time and space, into tiles that optimally exploit the dependency diagram of the numerical algorithm. The work of Strzodka et al. [48] utilized this technique on various iterative stencil computations for optimal cache utilization. Nguyen et al. [28] combined it with classical blocking techniques for improved cache friendliness, a concept which Yount et al. [62] utilized in order to exploit the HBM present on Xeon Phi coprocessors. Specifically, seismic finite-difference modelling has seen the time-space tiled out-of-core implementation presented in [53].

We consider the combined application of the reconstruction method in Aaker et al. [1] in conjuction with time-space tiling for a fast FWI implementation on GPUs, also beyond the device memory size. A key feature is that the memory-efficient reconstruction method significantly reduces the working set, whilst tiling allows efficient computations even when the (reduced) working set oversubscribes GPU memory. In order to realize this, we utilize the Compute Unified Device Architecture (CUDA) [6, 34, 47] application programming interface (API) for implementation on Nvidia-based graphics cards. We assume basic familiarity with the basic concepts of the API, common features of CUDA enabled GPUs and their execution model throughout this article. Modern Nvidia GPUs support Unified Memory (UM), allowing one to implement CUDA compute kernels with data in the same address space as on the CPU. Along with a page migration engine that enables memory oversubscription and on-demand page migration, UM significantly reduces the implementational effort required to develop CUDA code. The results of paper [19] demonstrate that the success of UM, in terms of performance, depends on the combination of the computational algorithm, the exact GPU hardware and, optionally, the usage of UM prefetching instructions. Particularly interesting to our case, the usage of UM shows comparable performance to explicitly managed memory for an iterative stencil computation, as long as copying across the interconnection is kept at a minimum [19]. With this in mind, we designed our implementation to utilize a hybrid of Unified Memory and explicitly managed memory for a suitable balance between readability, maintanability and performance. Throughout this article, we consistently utilize the nomenclature used for Nvidia GPUs.

The main novelties of our work can be summarized as: We combine two key optimizations targeted at the memory system in order to maximize the benefit and applicability of FWI on GPUs. Within this, we discuss how to accomodate source and receiver functionality, along with wavefield reconstruction methods, within time-space tiling. Furthermore, we give attention to the efficient implementation of the wavefield reconstruction method in [1] within staggered-grid FD methods. This type of wavefield representation method is ubiquitous in the seismic community[42], with uses including separation of wavefield constituents [3] and wavefield redatuming [5, 14]. Minor novelties arise in, e.g. the hybrid utilization for of explicitly managed

and Unified Memory in order to solve specific tasks. This work also presents extensive benchmarks, including a comparative analysis to existing methodologies.

The remainder of the article is organized as follows: We first give a brief introduction to modern GPUs and the CUDA programming model. The mathematical background of elastodynamic FWI is briefly reviewed, before introducing the corresponding staggered-grid FD discretization. We provide a brief algorithmic analysis aided by the roofline model [57] in order to motivate our two key optimizations, namely wavefield reconstruction and time-space tiling. Furthermore, we discuss the code design strategy and common low-level optimizations for the stencil scheme, certain optimizations required for implementation of the reconstruction method and the role of UM prefetching. The computational benchmarks assess the performance of the wavefield reconstruction method and time-space tiled gradient computations and forward modelling. We provide a realistic gradient computation example in which we benchmark our implementation on several Nvidia graphics cards and compare the performance to our in house CPU implementation. Finally, we present benchmarks against a modern, open-source implementation of FWI on GPUs.

2 Background

2.1 Introduction to modern GPU computing

GPUs are massively parallel computational devices originally designed to rapidly create and manipulate images for output to a display device. The introduction of unified graphics and compute architectures along with modern parallel computing APIs has enabled a broad adoption of GPUs in scientific computing [29, 37]. Applications that benefit from mapping onto GPUs typically exhibit a large computational demand, with substantial parallelism and where high throughput is more important than low latency [37]. Modern Nvidia GPUs are built as scalable arrays of *streaming multiprocessors* (SMs), along with elements such as device-specific dynamic random-access memory (DRAM), memory controllers and caches [6, 31, 34]. Communication to system memory is performed through an interconnection, e.g. PCI-Express [4] or NVLink [16]. Important components of an SM include several functional units for arithmetic and memory operations, along with an L1 cache, on-chip shared memory and a register file. Certain architectures such as Kepler, Volta and Turing feature a merged shared memory and L1 cache [6, 32, 33]. Furthermore, recent architectures include special purpose tensor cores in the SM design, built for deep learning matrix arithmetic [32]. SMs are designed to execute hundreds of threads concurrently according to the paradigm of the Single Instruction Multiple Threads (SIMT) execution model [34]. Here, thread management and execution is performed in scheduled groups of 32 threads, termed *warps*. The threads within a single warp are commonly and interchangeably denoted as *lanes* [34]. Warps execute instructions in a lockstep fashion, meaning that all lanes must execute the same instruction at any instance. This also entails that divergence due to conditional branches yields serialized execution of all branches taken. The independent thread scheduling introduced in Volta [32] assigns per-lane program counters and

call-stacks, whereas these resources were previously shared across the entire warp. The possibility to synchronize and communicate between the diverged threads of a warp enables the execution of algorithms with even finer parallelism patterns than previously possible.

The introduction of UM has constituted an important step in simplifying the developer's view of memory, and, therefore, the effort to develop applications on Nvidia GPUs. It offers a single pointer for data and a consistent view of memory across the CPU and GPU, where data transfers across the interconnection are automatically managed by the CUDA driver [6]. The limited form of UM introduced with Kepler [15] was significantly improved in Pascal [31], enabling oversubscription of Unified Memory and on-demand page migration [44]. Certain improvements of the page migration for multiple devices were introduced with Volta [32].

The CUDA programming model commonly refers the GPUs as the *device* and the CPU as the *host*, where programs running on the device are termed *kernels* [6, 34]. Kernel launches involve generation of a large number of threads on the device, organized in a two-level hierarchy. The collective group of threads initiated by a single kernel launch is referred to as a *grid*. The grid is composed of many *thread blocks*, where each block is mapped to a single SM during execution. All threads of the same grid share the same *global memory space*, which physically resides in device DRAM. The threads of a block may cooperate by sharing data through the shared memory local to each SM. In addition, lanes within a warp may cooperate by sharing register data through warp shuffle instructions [6, 34].

2.2 Full waveform inversion and gradient computations

Throughout this article, we consider the coordinate system formed by the Cartesian product between the spatial variable $\mathbf{x} = (x_1, x_2, x_3)^{\mathrm{T}}$, defined on a spatial domain Ω with boundary $\partial\Omega$, and the temporal variable $t \in [0, T]$.

Seismic experiments yield recordings of the induced wavefield response observed at selected receiver locations, constituting the data $\mathbf{d}(\mathbf{x}, t)$. Similarly, synthetic wavefield responses can be numerically computed for an assumed earth model, say $\mathbf{m}(\mathbf{x})$, by solving a *forward problem*. Solution of the forward problem entails solving a suitable wave equation. The *inverse problem* is to determine the earth model which, by solving the forward problem, gives the synthetic data $\overrightarrow{\boldsymbol{\psi}}(\mathbf{x}, t)$ most closely reproducing the observed data. This particular inverse formulation is commonly termed full waveform inversion, in which deviations between the observed and synthetic data are quantified by a misfit functional [9], denoted by $\chi(\overrightarrow{\boldsymbol{\psi}};\mathbf{d})$. The optimal earth model, $\hat{\mathbf{m}}$, minimizes the misfit according to

$$\hat{\mathbf{m}} = \arg\min_{\mathbf{m}} \chi(\overrightarrow{\boldsymbol{\psi}};\mathbf{d}), \tag{1}$$

with $\overrightarrow{\boldsymbol{\psi}}(\mathbf{x}, t) = \overrightarrow{\boldsymbol{\psi}}(\mathbf{x}, t;\mathbf{m})$ being the solution to the wave equation for a given model $\mathbf{m}$. The wave equation can be considered a specific example of a state equation, and we refer to the wavefield $\overrightarrow{\boldsymbol{\psi}}(\mathbf{x}, t)$ as the state wavefield. Furthermore, we consider

elastodynamic waves in lossless isotropic media. The velocity-stress formulation of the elastic wave equation reads [2]

$$\rho \partial_t v_i(\mathbf{x}, t) = \sum_{j=1}^{3} \partial_j \tau_{ij}(\mathbf{x}, t) + f_i(\mathbf{x}, t), \tag{2a}$$

$$\partial_t \tau_{ij}(\mathbf{x}, t) = \sum_{l,k=1}^{3} \Big(\lambda(\mathbf{x}) \delta_{ij} \delta_{kl} + \mu(\mathbf{x}) (\delta_{il} \delta_{jk} + \delta_{ik} \delta_{jl}) \Big) (\partial_l v_k(\mathbf{x}, t) - h_{kl}(\mathbf{x}, t)), \tag{2b}$$

where the wavefield variables are the particle velocities $v_i(\mathbf{x}, t)$ and the stress tensor $\tau_{ij}(\mathbf{x}, t)$ and the medium parameters are described by the mass density $\rho(\mathbf{x})$ and the two Lamé stiffness parameters $\lambda(\mathbf{x})$ and $\mu(\mathbf{x})$. The source functions consist of the sources of body force and deformation rate, namely $f_i(\mathbf{x}, t)$ and $h_{kl}(\mathbf{x}, t)$. Furthermore, the indices i, j, k, l take on the values 1, 2, 3, the Kronecker delta is denoted by $\delta_{\cdot,\cdot}$ and the operators ∂_i and ∂_t are shorthand notations for $\partial/\partial x_i$ and $\partial/\partial t$, respectively. For completeness, we expand the elastic wave equation as

$$\rho \partial_t v_1 = \partial_1 \tau_{11} + \partial_2 \tau_{12} + \partial_3 \tau_{13} + f_1, \tag{3a}$$

$$\rho \partial_t v_2 = \partial_1 \tau_{12} + \partial_2 \tau_{22} + \partial_3 \tau_{23} + f_2, \tag{3b}$$

$$\rho \partial_t v_3 = \partial_1 \tau_{13} + \partial_2 \tau_{23} + \partial_3 \tau_{33} + f_3, \tag{3c}$$

$$\partial_t \tau_{11} = (\lambda + 2\mu)(\partial_1 v_1 - h_{11}) + \lambda(\partial_2 v_2 + \partial_3 v_3 - h_{22} - h_{33}), \tag{3d}$$

$$\partial_t \tau_{22} = (\lambda + 2\mu)(\partial_2 v_2 - h_{22}) + \lambda(\partial_1 v_1 + \partial_3 v_3 - h_{11} - h_{33}), \tag{3e}$$

$$\partial_t \tau_{33} = (\lambda + 2\mu)(\partial_3 v_3 - h_{33}) + \lambda(\partial_1 v_1 + \partial_2 v_2 - h_{11} - h_{22}), \tag{3f}$$

$$\partial_t \tau_{23} = \mu(\partial_2 v_3 + \partial_3 v_2 - h_{23} - h_{32}), \tag{3g}$$

$$\partial_t \tau_{13} = \mu(\partial_1 v_3 + \partial_3 v_1 - h_{13} - h_{31}), \tag{3h}$$

$$\partial_t \tau_{12} = \mu(\partial_1 v_2 + \partial_2 v_1 - h_{12} - h_{21}), \tag{3i}$$

in which the stress tensor is uniquely described by six elements rather than nine, due to the symmetry property $\tau_{ij}(\mathbf{x}, t) = \tau_{ji}(\mathbf{x}, t)$ [2]. We define the synthetic data to be

$$\overrightarrow{\boldsymbol{\psi}}(\mathbf{x}, t) = (v_1, v_2, v_3, \tau_{11}, \tau_{22}, \tau_{33}, \tau_{23}, \tau_{13}, \tau_{12})^{\mathrm{T}}(\mathbf{x}, t), \tag{4}$$

and the model parameters we wish to determine by performing FWI are some formulation of the medium parameters of the earth. For simplicity, we define the model parameters to be

$$\mathbf{m}(\mathbf{x}) = (\rho, \lambda, \mu)^{\mathrm{T}}(\mathbf{x}). \tag{5}$$

The gradient of the misfit functional with respect to the model parameters is ubiquitous in local optimization methods used for solving Eq. 1, see, e.g. [30]. In order to utilize these methods, we must hence be able to compute

$$\frac{\partial \chi(\overline{\boldsymbol{\psi}};\mathbf{d})}{\partial \mathbf{m}} = \Big(\nabla_{\rho}\chi(\overline{\boldsymbol{\psi}};\mathbf{d}), \nabla_{\lambda}\chi(\overline{\boldsymbol{\psi}};\mathbf{d}), \nabla_{\mu}\chi(\overline{\boldsymbol{\psi}};\mathbf{d})\Big)^{\mathrm{T}}. \tag{6}$$

The gradient of the misfit functional is computed with the adjoint state method, in which the state variables are combined with solutions of the so-called adjoint state equation [38].

Following the approach in Vigh et al. [54], the adjoint state equation reads

$$\rho\partial_{t'} v_i^{\dagger}(\mathbf{x}, t') = -\sum_{j=1}^{3} \partial_j \tau_{ij}^{\dagger}(\mathbf{x}, t') + f_i^{\dagger}(\mathbf{x}, t'), \tag{7a}$$

$$\partial_{t'} \tau_{ij}^{\dagger}(\mathbf{x}, t') = -\sum_{l,k=1}^{3} \Big(\lambda(\mathbf{x})\delta_{ij}\delta_{kl} + \mu(\mathbf{x})(\delta_{il}\delta_{jk} + \delta_{ik}\delta_{jl})\Big)(\partial_l v_k^{\dagger}(\mathbf{x}, t') - h_{kl}^{\dagger}(\mathbf{x}, t')), \tag{7b}$$

with

$$t' = T - t, \tag{8}$$

and $v_i^{\dagger}(\mathbf{x}, t')$ and $\tau_{ij}^{\dagger}(\mathbf{x}, t')$ the adjoint particle velocities and the adjoint stress tensor, respectively. The superscript † denotes that the given variable is an adjoint quantity. The adjoint source functions $f_i^{\dagger}(\mathbf{x}, t')$ and $h_{kl}^{\dagger}(\mathbf{x}, t')$ depend on the specific choice of the misfit functional. The adjoint sources inject some form of a residual between the true and predicted data. For the standard least-squares misfit functional, we for example have

$$f_i^{\dagger}(\mathbf{x}, t') = \sum_{\mathbf{x}_r} \delta(\mathbf{x} - \mathbf{x}_r)(v_i^{\mathrm{obs}}(\mathbf{x}_r, t') - v_i(\mathbf{x}_r, t')), \tag{9}$$

in which $v_i^{\mathrm{obs}}(\mathbf{x}_r, t')$ is the i'th particle velocity observed at receiver position $\mathbf{x}_r$ of the seismic experiment and $\delta(\cdot)$ is the Dirac delta distribution. The specific form of the adjoint source in Eq. 9 therefore represents the arithmetic difference between observed and modelled particle velocities, to be injected as point sources at the receiver positions. The adjoint field can be compactly denoted as

$$\overline{\boldsymbol{\varphi}}(\mathbf{x}, t') = (v_1^{\dagger}, v_2^{\dagger}, v_3^{\dagger}, \tau_{11}^{\dagger}, \tau_{22}^{\dagger}, \tau_{33}^{\dagger}, \tau_{23}^{\dagger}, \tau_{13}^{\dagger}, \tau_{12}^{\dagger})^{\mathrm{T}}(\mathbf{x}, t'). \tag{10}$$

The numerical cost of solving the adjoint state equation is the same as one solution of the state equation.

The gradient with respect to the first Lamé parameter reads [54]

$$\nabla_{\lambda}\chi(\overrightarrow{\boldsymbol{\psi}};\mathbf{d}) = -\int_{\Omega}\int_{0}^{\mathrm{T}} \frac{1}{3\lambda(\mathbf{x})+2\mu(\mathbf{x})}\Big(\tau_{11}^{\dagger}(\mathbf{x},t)+\tau_{22}^{\dagger}(\mathbf{x},t)+\tau_{33}^{\dagger}(\mathbf{x},t)\Big) \times\Big(\partial_1 v_1(\mathbf{x},t)+\partial_2 v_2(\mathbf{x},t)+\partial_3 v_3(\mathbf{x},t)\Big)\,\mathrm{d}t\,\mathrm{d}^3\mathbf{x}, \tag{11}$$

in which the integration over time has the form of a zero lag cross-correlation, involving two wavefields propagating in opposite temporal directions. Similar relations hold for the other model parameters, as shown in, e.g. [9, 54]. A graphical overview of the workflow of gradient-based FWI can be found in, e.g. [51].

2.3 Discretization of the state equation

The initial condition of the state field is

$$\overrightarrow{\boldsymbol{\psi}}(\mathbf{x},t) = \mathbf{0} \qquad t \le 0, \tag{12}$$

and suitable boundary conditions must be prescribed on the boundary $\partial\Omega$. Absorbing boundary conditions are used on one or more boundaries in order to avoid artificial reflections due to truncating the computational domain. For this purpose, we utilize the perfectly matching layer (PML) formulation of [39]. It augments Eq. 2 with two sets of memory variables acting as energy sinks in a small zone beyond the domain of interest. Equation 2 is discretized using the explicit FD method of Virieux [55], which involves staggering of wavefield quantities in both time and space. The particle velocities are updated at half-time steps by utilizing

$$\partial_t v_i^n(\mathbf{x}) \approx \frac{v_i^{n+\frac{1}{2}}(\mathbf{x}) - v_i^{n-\frac{1}{2}}(\mathbf{x})}{\Delta t}, \tag{13}$$

and the stress tensor is updated at integer time steps

$$\partial_t \tau_{ij}^{n+\frac{1}{2}}(\mathbf{x}) \approx \frac{\tau_{ij}^{n+1}(\mathbf{x}) - \tau_{ij}^{n}(\mathbf{x})}{\Delta t}, \tag{14}$$

where the integer superscript n refers to the discretized time $n\Delta t$ and Δt is the timestep increment. Staggered forward, $\mathcal{D}_i^+$, and backward, $\mathcal{D}_i^-$, differential operators of order $2L$ are used to approximate the spatial derivatives, *viz*

$$\partial_i f(x_i + \frac{1}{2}\Delta x_i) \approx \mathcal{D}_i^+ f(x_i) = \frac{1}{\Delta x_i}\sum_{\ell=0}^{L-1}\alpha_\ell[f(x_i+(\ell+1)\Delta x_i) - f(i-\ell\Delta x_i)], \tag{15a}$$

$$\partial_i f(x_i - \frac{1}{2}\Delta x_i) \approx \mathcal{D}_i^- f(x_i) = \frac{1}{\Delta x_i}\sum_{\ell=0}^{L-1}\alpha_\ell[f(x_i+\ell\Delta x_i) - f(x_i-(\ell-1)\Delta x_i)], \tag{15b}$$

where Δx_i is the spatial grid size in direction, i and α_ℓ are the differentiator coefficients. In this notation, f is an arbitrary function and L is commonly denoted as

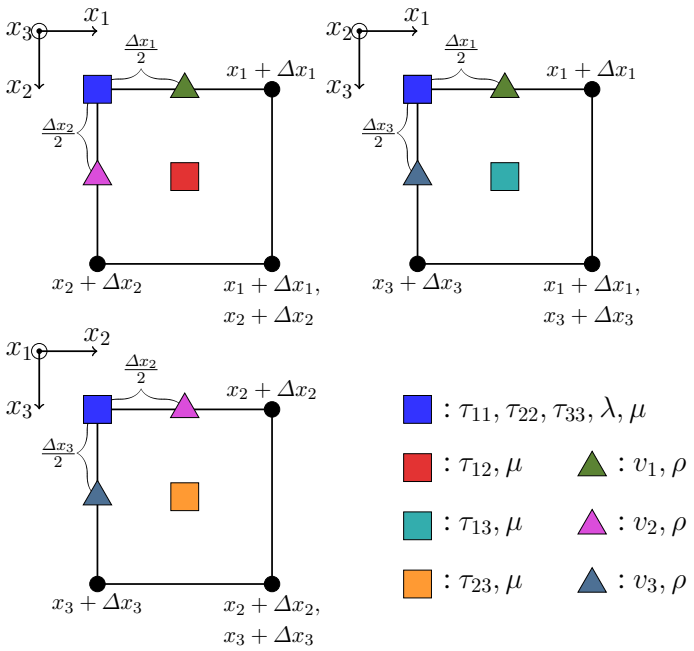


Fig. 1 Spatial cell structure showing the node location for all wavefield variables and medium parameters of the staggered-grid FD discretization (color figure online)

the *half-length* of the operators $\mathcal{D}_i^+$ and $\mathcal{D}_i^-$. The forward operator is used when the output of the differentiation is required to be on the staggered grid in direction i, and the backward operator is used when the output is required on the reference grid. The elementary cell of the discretized wave equation is shown in Fig. 1. For example, the explicit update formula for v_2 is

$$v_2^{n+\frac{1}{2}} = v_2^{n-\frac{1}{2}} + \frac{\Delta t}{\rho}\left(\mathcal{D}_1^- \tau_{12}^n + \mathcal{D}_2^+ \tau_{22}^n + \mathcal{D}_3^- \tau_{23}^n\right), \tag{16}$$

whereas the update of, e.g. τ_{33} reads

$$\tau_{33}^{n+1} = \tau_{33}^{n}\, ^{n-1} + \Delta t\left(\lambda \mathcal{D}_1^- v_1^{n+\frac{1}{2}} + \lambda \mathcal{D}_2^- v_2^{n+\frac{1}{2}} + (\lambda + 2\mu)\mathcal{D}_3^- v_3^{n+\frac{1}{2}}\right). \tag{17}$$

Injection of source functions can be considered a separate step in the modelling scheme due to the spatial sparsity of sources. When v_2 and τ_{33} have been updated according to Eqs. 16 and 17, the respective source functions can be injected according to

$$v_2^{n+\frac{1}{2}} = v_2^{n+\frac{1}{2}} + \frac{\Delta t}{\rho} f_2^n, \tag{18}$$

$$\tau_{33}^{n+1} = \tau_{33}^{n+1} - \Delta t\left(\lambda h_{11}^{n+\frac{1}{2}} + \lambda h_{22}^{n+\frac{1}{2}} + (\lambda + 2\mu) h_{33}^{n+\frac{1}{2}}\right). \tag{19}$$

A complete list of the discretized update scheme can be found in, e.g. Graves [13].

2.4 Discretization of the adjoint state equation

The numerical implementation of the adjoint state equation requires only minor modifications of the implementation of the state equation. The former is a time reversed wave equation to be solved forward in time, with respect to the variable $t' = T - t$. The initial condition for this field is

$$\overleftarrow{\boldsymbol{\varphi}}(\mathbf{x}, t') = \mathbf{0} \qquad t' \leq 0 \quad \Leftrightarrow \quad \overleftarrow{\boldsymbol{\varphi}}(\mathbf{x}, t) = \mathbf{0} \qquad t \geq T, \tag{20}$$

and its boundary conditions are time reversed variants of the boundary conditions of the state field. The most prominent implementational difference is that the update order between the particle velocities and the stresses is reversed when simulating backwards in time.

2.5 Discretization of the gradient kernels

The model parameters of the inverse problem are often discretized on the same grid as the reference FD grid. The support of each discretized model parameters is then reduced to one grid node and the spatial integrals in the gradient expressions vanish. The gradient expression for the first Lamé parameter at location $\mathbf{x}'$ therefore becomes

$$\begin{aligned} \nabla_{\lambda}(\mathbf{x}')\chi(\overleftarrow{\boldsymbol{\psi}};\mathbf{d}) = -\sum_{n=0}^{N_t-1} \frac{1}{3\lambda(\mathbf{x}') + 2\mu(\mathbf{x}')} \left(\tau_{11}^{\dagger,n}(\mathbf{x}') + \tau_{22}^{\dagger,n}(\mathbf{x}') + \tau_{33}^{\dagger,n}(\mathbf{x}')\right) \\ \times \left(\mathcal{D}_1^- v_1^n(\mathbf{x}') + \mathcal{D}_2^- v_2^n(\mathbf{x}') + \mathcal{D}_3^- v_3^n(\mathbf{x}')\right) \Delta t, \end{aligned} \tag{21}$$

with N_t being the number of timesteps. In similarity to the forward modelling scheme, gradient computations involve calculations of partial derivatives. The implementation of Eq. 21 requires the two distinct wavefields to be available on the same timestep. This is computationally challenging to achieve with time domain solvers, as the adjoint state field propagates backwards in time. The propagation behaviour of the two fields is shown in Fig. 2. Moreover, the adjoint wavefield can only start its simulation *after* the state wavefield has been simulated to the end time $t = T$. A straightforward solution to this problem is to save snapshots of the state wavefield during forward propagation and subsequently read the snapshots when simulating the adjoint field. For three-dimensional problems, there is typically not enough memory available to accomodate several thousands of timesteps. Relying on the low bandwidths of disks and/or interconnections might decrease program performance dramatically. As a part of the next section, we will introduce the motivation for utilizing so-called wavefield reconstruction methods to ameliorate this issue.

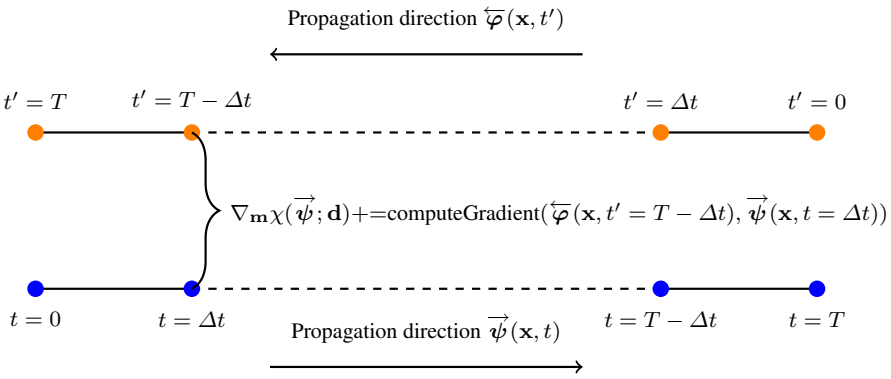


Fig. 2 Graphically visualizing the propagation directions of the wavefields contributing to the FWI gradient. The state and adjoint wavefields must be simultaneously available at all timesteps in order to compute the gradient, which is an algorithm depicted by the pseudocode computeGradient()

Table 1 Probable characteristics of a modern graphics card, utilized for our algorithmic analysis

Peak arithmetic performance (π)	DRAM memory bandwidth (β)	Interconnection bandwidth
8064 FLOP/s	448 GB/s	64 GB/s

3 Method

The implementation of the forward modelling scheme is key to achieving sufficiently fast propagation of the state and adjoint fields separately. A second challenge is to be able to provide the two wavefields at the same timestep for the gradient computation kernels. We initiate the analysis and motivation by assuming that all required quantities can be kept in memory, including the snapshots of the state wavefield. We will underways consider scenarios in which this assumption cannot be met. For our algorithmic analysis, we consider the performance and bandwidth specifications of a hypothetical, modern graphics card given by Table 1. These specifications correspond roughly to a mid- or upper-range GPU of the Turing architecture, with the bidirectional interconnection bandwidth being equivalent to 16 lanes of the PCI Express 4.0 standard [4].

3.1 Algorithmic analysis

Performance models enable inference of achieved performance relative to hardware capabilities, identifying bottlenecks and limitations in both implementations of numerical algorithms and the available hardware. The roofline model [57] is an insightful, yet simple, performance model which provides intuitive visual

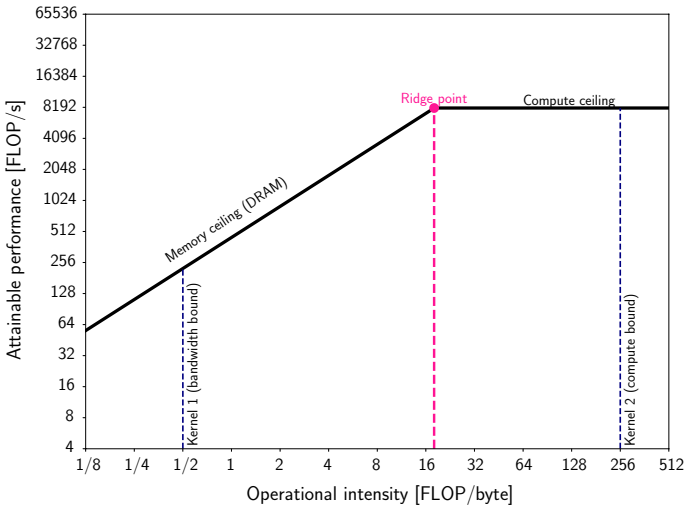


Fig. 3 The roofline performance model for the hypothetical system. The ridge point separates the range of operational intensity in which computations are bandwidth-bound from the range in which computations are compute-bound

description along with accurate performance bounds. According to the roofline model, the maximum achievable performance for an input of size N is given by

$$P_{\max}(N) = \min \left\{ \begin{array}{l} \pi, \\ \beta I_O(N), \end{array} \right\}, \tag{22}$$

in which π is the peak arithmetic performance of the hardware and β is its memory bandwidth. $I_O(N)$ is the operational intensity of the numerical algorithm at the given input size, defined as the quotient of the arithmetic instruction count and the memory traffic in bytes. The algorithm-hardware combination is either memory bound, occurring when $\beta I_O(N) < \pi$, or compute bound, when $\beta I_O(N) > \pi$. The crossing at $\beta I_O(N) = \pi$ is termed the *ridge point* of the roofline model, in which the algorithm is equally dependent on memory bandwidth and arithmetic performance. The graphical visualization of this model is a plot of performance against operational intensity on logarithmic axis scales, and a visualization for the hypothetical graphics card is shown in Fig. 3. In this display, the two computational kernels, 1 and 2, are, respectively, located in the bandwidth-bound and the compute-bound regions of the roofline model. We now wish to gain insight into which factors may limit the performance of the two main types of computational kernels in the FWI algorithm. We regard the two kernel types, the FD simulation kernels and the gradient computation kernels, as separate algorithms.

The FD method is a stencil type operation, as it involves updating one point of the grid based on its neighbours' values. The stencil formulas of our forward modelling scheme are highly similar to one another, and for brevity the stencil

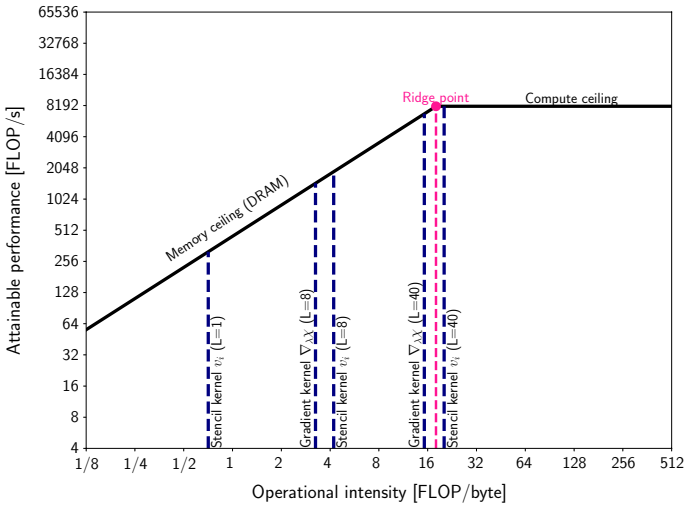


Fig. 4 The roofline model of the hypothetical system with the stencil kernels of particle velocity and gradient computations plotted for a range of differentiator half-lengths

formula of v_2 in Eq. 16 is considered. The arithmetic instruction count for one timestep of this update formula is given by

$$C_A[v_2](N) = N\left\{\begin{array}{ll} 3 \times 2L + 1 & \text{Fused multiplication and addition (FMA) operations,} \\ 3 & \text{Addition (ADD) operations,} \\ 1 & \text{Division (DIV) operations,} \end{array}\right\}, \tag{23}$$

with

$$N = N_{x_1} \times N_{x_2} \times N_{x_3}, \tag{24}$$

being the number of spatial grid nodes. We specify the lower bound for the required memory traffic by assuming that there is no redundancy in loads nor stores. This assumes optimal re-use of the values used by the numerical differentiation operators, and that these values can be kept in on-chip resources such as caches or registers. Assuming four byte sized floating point numbers, the minimum required memory traffic, $B(N)$, reads

$$B[v_2](N) \geq 4 \times N \times 6 \quad \text{bytes}. \tag{25}$$

Combining the number of arithmetic operations in Eq. 23 and the lower bound for the memory traffic in Eq. 25 gives an operational intensity

$$I_O[v_2](N) = \frac{C_A[v_2](N)}{B[v_2](N)} = \frac{N(12L+5)}{N \times 24} = \frac{L}{2} + \frac{5}{24} \propto \mathcal{O}(1), \tag{26}$$

where we for simplicity have grouped the different arithmetic operations together and the FMA instructions count as two operations each. Equation 26 reveals that the operational intensity for our FD scheme is independent of the number of grid cells, under the assumption that the entire working set fits in memory. Along with Fig. 4, we can observe that the achievable performance of each of the particle velocity kernels on the hypothetical GPU is bandwidth bound for realistic values of L.

Under the very same assumptions, it can be shown that one timestep of the gradient computation formula in Eq. 21 has an arithmetic operations count given by

$$C_A[\nabla_\lambda \chi](N) = N(12L + 8), \tag{27}$$

and a minimum required memory traffic according to

$$B[\nabla_\lambda \chi](N) \geq 4 \times N \times 8 \quad \text{bytes}. \tag{28}$$

This gives an upper-bound operational intensity

$$I_O[\nabla_\lambda \chi](N) = \frac{N(12L + 8)}{4 \times N \times 8} = \frac{12L}{32} + \frac{1}{4} \propto \mathcal{O}(1), \tag{29}$$

from which it is deducible that also this computational kernel is bandwidth bound in realistic scenarios.

The analysis of the FD stencil scheme has thus far assumed that the corresponding working set fits into device memory. Similarly, the analysis of the gradient computation has assumed that all snapshots of the state wavefield fit into device memory and are directly available. In reality, the second assumption is particularly unlikely to be satisfiable. When either of these conditions cannot be met, one must consider the host to device interconnection bandwidth, in order to accurately predict the upper performance bound.

The roofline model accomodates memory hierarchies by adding separate memory ceilings for each member of the hierarchy. The attainable performance is dictated by the most limiting roofline, i.e. the lowest roofline in the plot. The roofline model for the particle velocity stencil formula is given by Fig. 5. The lower bandwidth of the interconnection shifts the ridge point towards higher operational intensities, thereby extending the bandwidth-bound region. Moreover, the maximum attainable performance within the bandwidth bound region is limited by the bandwidth of the interconnection. The roofline model therefore suggests that an efficient FWI computation minimizes the memory traffic across the interconnection. The gap between the DRAM and interconnection ceilings is the potential *performance* reward for such optimizations. Certain optimizations may, however, increase the computational load performed, and the runtime is determined by the balance between the computational load and the achieved arithmetic performance. We consider the usage of wavefield reconstruction methods in order to eliminate the need for storing and loading wavefield snapshots of the state wavefield. The state wavefield is then propagated backwards in time alongside its adjoint counterpart, at the cost of one extra FD modelling. The working set is thereby reduced to only two wavefield snapshots, along with terms such as the elastic medium parameters and the misfit gradient terms. In conjuction, we consider the use of time-space tiling methods in order to maintain the performance of our FWI implementation when the reduced working set exceeds the device memory capacity.

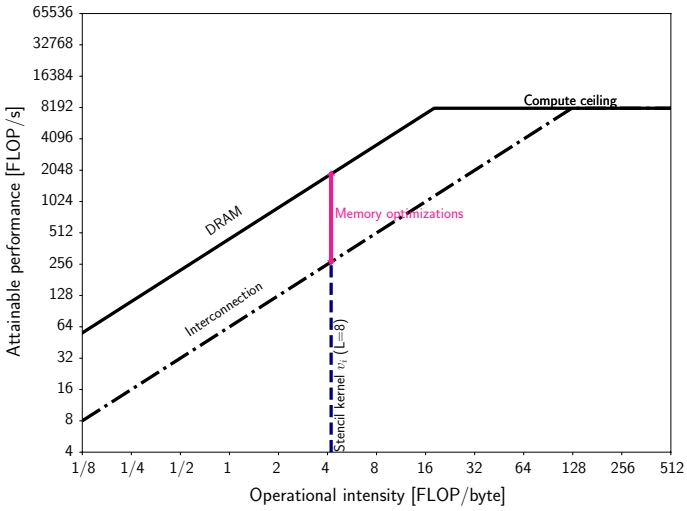


Fig. 5 The roofline model of the hypothetical system, simultaneously considering the DRAM and interconnection bandwidths

3.2 Wavefield reconstruction

Reconstruction methods allow reverse-time propagation of a wavefield by injecting suitable source functions that compensate for the energy dissipated through the attenuating boundaries. Reconstructing the state wavefield eliminates the need for wavefield snapshots in FWI gradient computations, thereby significantly reducing the amount of data transferred across the interconnection.

We employ a wavefield reconstruction method formulated from conservation of elastodynamic power flux, which utilizes recordings of the modelled wavefield at a boundary enclosing the medium of interest. In order to perform reconstruction, the recorded wavefield is represented as body force and deformation rate sources. Respectively, these sources read [1]

$$f_i^{\mathrm{rec}}(\mathbf{x},t) = \rho(\mathbf{x}) \int_{\partial\mathcal{V}} \delta(\mathbf{x}-\mathbf{x}') \frac{\mathcal{T}_i(\mathbf{x}',t)}{\rho(\mathbf{x}')} \mathrm{d}^2\mathbf{x}', \tag{30}$$

and

$$h_{kl}^{\mathrm{rec}}(\mathbf{x},t) = \int_{\partial\mathcal{V}} \delta(\mathbf{x}-\mathbf{x}') v_k(\mathbf{x}',t) n_l \mathrm{d}^2\mathbf{x}', \tag{31}$$

with

$$\mathcal{T}_i(\mathbf{x}',t) = \sum_{j=1}^{3} \tau_{ij}(\mathbf{x}',t) n_j \tag{32}$$

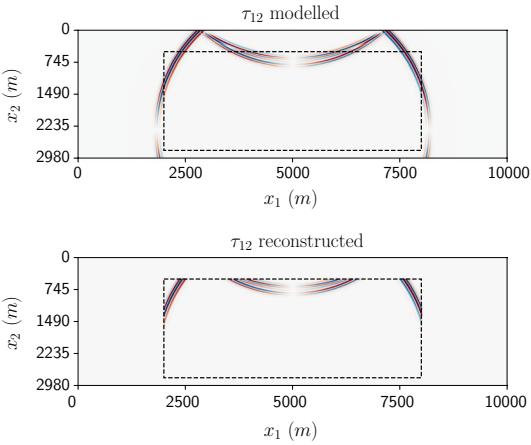


Fig. 6 Wavefield reconstruction within a homogenous region with an elastic free surface at the top. The reconstruction boundary $\partial\mathcal{V}$ is denoted by the dashed lines. The colorscale has been clipped in order to emphasize all events and any errors. This figure is based on Fig. 1 in [1]

being the traction at the boundary $\partial\mathcal{V}$ with normal vector $\hat{\mathbf{n}} = (n_1, n_2, n_3)^{\mathrm{T}}$. Figure 6 graphically demonstrates an application of the reconstruction method, in which a simulated wavefield is reconstructed only inside the subregion bounded by $\partial\mathcal{V}$.

The source functions in Eqs. 30 and 31 require the storage of six unique wavefield quantities at the boundary surface, which is performed during forward simulation. Interpolation is required when the grid nodes of the boundary $\partial\mathcal{V}$ do not intersect with the grid nodes of the required wavefield quantities. Such situations occur when the boundary is curved and/or one considers the usage of spatially staggered FD schemes. The forward and backward interpolation operators for an order $2L$ accuracy read [10, 27]

$$f(x_i + \eta\Delta x_i) \approx \mathcal{I}_i^{+} f(x_i) = \sum_{\ell=0}^{L-1} \beta_\ell^\eta [f(x_i + (\ell+1)\Delta x_i) + f(i - \ell\Delta x_i)], \tag{33a}$$

$$f(x_i - \eta\Delta x_i) \approx \mathcal{I}_i^{-} f(x_i) = \sum_{\ell=0}^{L-1} \beta_\ell^\eta [f(x_i + \ell\Delta x_i) + f(x_i - (\ell-1)\Delta x_i)], \tag{33b}$$

with an interpolation shift $-0.5 \leq \eta \leq 0.5$ and β_ℓ^η being the corresponding interpolation coefficients. The interpolators in Eq. 33 are stencil operators similar to the forward and backward differentiation operators of Eq. 15.

The first step to calculating an FWI gradient is to solve the elastic wave equation with an injection of sources and recording of receivers determined by the acquisition geometry. The pseudocode for this step can be written as in Algorithm 1, where we have included the procedures required for recording the reconstruction sources at the boundary. The physical locations of sources and receivers generally do not coincide with the FD grid, and injection and recording of these quantities require the usage of interpolation operators. The gradient of the misfit functional can be calculated with Algorithm 2. The latter includes an explicit call to the first algorithm, providing the wavefield response at the receiver locations and the required reconstruction sources.

Algorithm 1: Velocity-stress forward modelling with source injection, recording of receiver responses and recording of reconstruction-type sources at the boundaries.

Input: Number of timesteps `nt` and the lengths of the three spatial axes `n1,n2,n3`.

```
for(t = 0; t < nt; t++){
    injectSource(t);                //Inject physical source.
    saveReceivers(t);               //Record receivers at physical locations.
    recordReconstructionSource(t);  //Record reconstruction sources on the boundary.

    //FD stencil kernels for forward propagation.
    {
        for((x1,x2,x3) in (n1,n2,n3))
            updateVelocity(x1,x2,x3);   //Update particle velocities from stress.

        for((x1,x2,x3) in (n1,n2,n3))
            updateStress(x1,x2,x3);     //Update stress from particle velocities.
    }
}
```

Algorithm 2: Velocity-stress gradient computations with wavefield reconstruction methods.

Input: Number of timesteps `nt` and the lengths of the three spatial axes `n1,n2,n3`.

```
call Algorithm 1;          //Generate data, save reconstruction sources.
computeMisfit();           //Compute misfit value for optimization procedure.
computeAdjointSources();   //Create adjoint sources at receiver locations.

for(t = nt-1; t >= 0; t--){

    //Propagate adjoint wavefield backwards in time.
    {
        for((x1,x2,x3) in (n1,n2,n3))
            updateAdjointStress(x1,x2,x3);

        for((x1,x2,x3) in (n1,n2,n3))
            updateAdjointVelocity(x1,x2,x3);

        injectAdjointSource(t);
    }

    //Compute contributions to gradient.
    computeGradient();

    //Propagate state wavefield backwards in time.
    {
        for((x1,x2,x3) in (n1,n2,n3))
            updateStress(x1,x2,x3);

        for((x1,x2,x3) in (n1,n2,n3))
            updateVelocity(x1,x2,x3);

        injectReconstructionSource(t);
    }
}
```

The source functions in Eqs. 30 and 31 can be transferred across the interconnection in a buffered fashion, such that they occupy a minimal amount of device memory at any instant. The interconnection transfers can be performed simultaneously

with, e.g. the finite-difference stencil computations and can in the optimal case be completely hidden. In the following, the working set size for gradient computations refers to the size required when using wavefield reconstruction methods. We consider time-space tiling techniques to maintain program performance in the out-of-core scenario, i.e. when the working set exceeds the physical size of device memory.

3.3 Time-space tiling for out-of-core FWI

At its core, the computations of the FWI method consist of iterative stencils schemes alongside auxiliary routines. The attainable performance in the out-of-core scenario of any naïve implementation drops significantly due to the bandwidth preference of the algorithm and the low interconnection bandwidth, see, e.g. Fig. 5. In this notation, naïve refers to the observation that the loops over the four-dimensional iteration space, consisting of time and space, in the stencil schemes of Algorithms 1 and 2 are oblivious to data reuse in the form of temporal locality. In contrast, time-space tiling exploits the dependency diagram of stencil schemes in order to increase temporal locality [59, 60]. This can be used to improve the caching abilities of multilevel memory hierarchies, which is particularly interesting for bandwidth bound computations. Our motivation is to utilize this method to maintain program performance when oversubscribing the GPU memory.

To this end, we tile the four-dimensional iteration space in the temporal direction and the slowest spatial direction[53]. This amounts to updating the field(s) on a subset of the x_3 axis rather than along the full spatial extent, at any given timestep.

This spatial subset, of length W_{x_3}, is moved and/or modified according to a transformation rule when iterating across a fixed number of timesteps, say W_t. The tile transformation rule, along with the spatial and temporal tile sizes W_{x_3} and W_t, describes an iteration space composed into non-overlapping characteristic shapes, termed tiles, that are preferentially updated in time and space. The preferential update directions in time and space permit an improved reuse of values loaded into a specific memory subsystem, constituting the main attractive property of time-space tiling. Tiling can therefore help overcome the bandwidth gap between device DRAM and the interconnection. One should note that the spatial tile size may vary across timesteps depending on the specific tile transformation rule, yet in this work it can be considered a runtime constant. We first introduce the time-space tiling method to the wave equation stencil scheme, before we consider a generalization in order to include all relevant routines required by the FWI implementation.

The wave equation implementation considers iterative stencil updates of the particle velocity, v_i, and the stress tensor, τ_{ij}, on a spatial grid across several timesteps. The spatial dependency between particle velocity and stress within a given integer timestep is termed the *wavefront-angle*, denoted by A_{x_3}. In FD schemes, this is equivalent to the differentiator half-length L. We present the time-space tiled version of the velocity-stress stencil scheme (present within Algorithm 1) in Algorithm 3. The general form of the iteration space is shown in Fig. 7, having the shape of

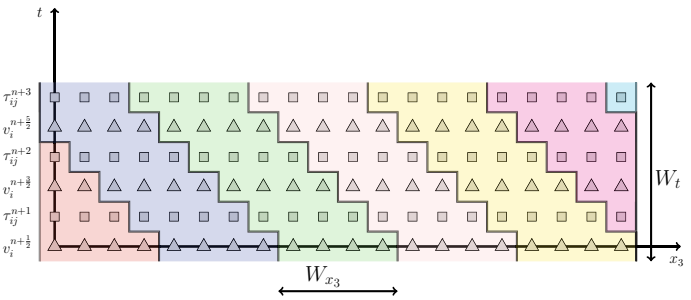


Fig. 7 The iteration space diagram for time-space tiling of the velocity–stress scheme, with the differentiator half-length equal to $L = 1$. Each tile is shaded with a designated colour (color figure online)

sequential parallelograms. The dependencies of a given tile in *parallelogram tiling* are provided by either the previous tile or the boundary conditions.

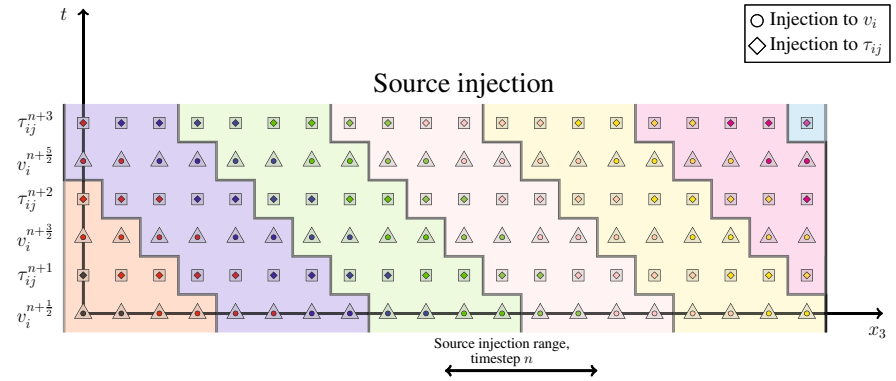
Algorithm 3: Velocity-stress stencil scheme with time-space tiling over the x_3 and t dimensions.

Input: Number of timesteps `nt`, the lengths of the three spatial axes `n1,n2,n3`, the wavefront angle `ax3`, the tile-lengths `wt` and `wx3` and the number of tiles `nbt` and `nbx3`.

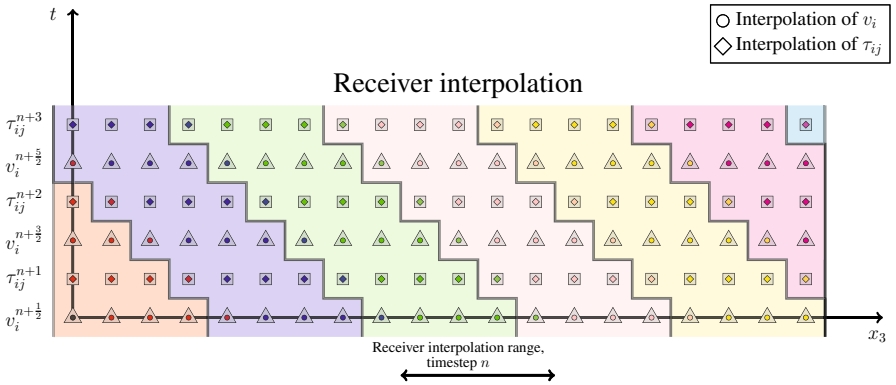
```
for(tt = 0; tt < nbt; tt++){
    for(xx3 = 0; xx3 < nbx3; xx3++){
        ox3 = 0; //Initialize wavefront offset.
        for( t = tt*wt; t < min((tt+1)*wt,nt); t++){
            //Update particle velocity from stress.
            for( x3 = xx3*wx3-ox3; x3 < (xx3+1)*wx3-ox3; x3++)
                for((x1,x2) in (n1,n2))
                    updateVelocity(x1,x2,x3);

            //Update stress from particle velocity, shifted by one wavefront angle.
            for(x3 = xx3*wx3-ox3-ax3; x3 < (xx3+1)*wx3-ox3-ax3; x3++)
                for((x1,x2) in (n1,n2))
                    updateStress(x1,x2,x3);

            ox3 += 2*ax3; //Add to wavefront offset due to dependencies in the next timestep.
        }
    }
}
```

The update range of the stress tensor is shifted by one wavefront angle relative to that of the particle velocities, and the entire tile is shifted two wavefront angles when both the velocities and the stress have been updated. The coupled dependency between the two principal wavefield quantities is hence respected. The wavefield is updated on the entire domain at every W_t'th timestep, leading to a memory which generally holds wavefield values located at different timesteps. The time-space tiled algorithm reduces to the naïve algorithm when specifying $W_t = 1$ and $W_{x_3} = N_{x_3} + A_{x_3}$, leading to only one spatial tile which updates the wavefield on the entire spatial domain by one timestep.

Our FWI implementation requires more functionality than the FD stencil computations. The routines related to recording of receivers, injection of sources and the gradient computation kernels interact with the values of the FD scheme and must therefore respect the dependencies of the tiled iteration space. In addition, the



(a) Source injection in the parallelogram tiled velocity-stress scheme.



(b) Receiver interpolation in the parallelogram tiled velocity-stress scheme.

Fig. 8 Source injection (**a**) and receiver interpolation (**b**) iteration space structure in the parallelogram tiled version of Algorithm 1. The tiles are shaded according to the stencil iteration space in Fig. 7 and the node colouring denotes which tile the injection or interpolation operation belongs to. The black nodes belong to a tile which is not observed in Fig. 7. The half-length for differentiation and interpolation is equal to $L = 1$ (color figure online)

same rules apply to recording and injection of the sources for wavefield reconstruction. Based on Algorithm 1, we demonstrate the evaluation ranges for source injection and receiver interpolation in Fig. 8a and b, respectively. Injection of sources is performed prior to evaluation of the stencil update in Algorithm 1, and the sources therefore inject to the stencil update ranges of the previous timestep. Receiver interpolation is performed prior to the stencil kernels but succeeding the source injection, and is therefore shifted by one wavefront angle compared to the source injection update range. Evidently, the update range for source injection to either v_i or τ_{ij} is shifted by minus two wavefront angles compared to that of its respective stencil kernel. According to this notation, the wavefront angle shift for receiver interpolation is minus one wavefront angle. The intra-tile shift of two wavefront angles across

timesteps is preserved, as none of the extra routines have a *coupled* dependency with the wavefield quantities. The introduction of negative dependency shifts requires more tiles to be introduced, as can be seen from the black nodes for source injection and receiver interpolation in Fig. 8a and b, respectively. The black nodes cannot be satisfied by the tiles belonging to the stencil scheme, but introducing an extra, shifted tile will alleviate the issue. No FD stencil operations are performed in the extra tile of this specific example, as the wavefield values in its domain of support are completely determined by the boundary conditions. Recording and interpolation of the reconstruction sources share the same update range as the receiver interpolation, c.f. Algorithm 1. Injection of the reconstruction sources in Algorithm 2 is performed succeeding (reverse-time) stencil updates, and therefore have update ranges shifted by positive multiples of the wavefront angle.

4 Implementation

We utilized object-oriented C++ as our host programming model in order to manage the codebase with a clear inheritance pattern. The classes of the GPU implementation inherit from CPU classes and implement their computational kernels by virtually overriding specific member functions. The overridden member functions act as wrappers for CUDA-C compute functions, and all usage of polymorphism is entirely determined at compile time. The inheritance-oriented design allowed us to minimize code duplication of auxiliary routines, which constitute a significant amount of code in any FWI implementation. The CPU code furthermore served as verification of program correctness.

We chose to utilize UM for many variables involved in the forward modelling and gradient computations. UM offers a single pointer for data and a consistent view of memory across the CPU and GPU. In the Unified Memory model, data transfers between host and device are automatically managed by the CUDA driver [6]. The Pascal generation of graphics cards introduced a page migration engine enabling oversubscription of Unified Memory and on-demand page migration [44].This enabled us to write an out-of-core FWI implementation in the same address space as on the CPU, without having to write explicit transfers for the UM-allocated variables. The ability to logically access the main wavefield and model variables the same way as on the host proved particularly advantageous for the correctness and maintainability of the implementation when introducing time-space tiling. Explicit memory management and transfers were, however, used for the variables stored with an explicit time dependence, with the respective transfers organized in a buffered fashion. Such quantities include source and receiver functions, along with the source functions for wavefield reconstruction. Explicit management of these variables avoided unnecessary allocation of page-locked memory on the host. Perhaps more importantly, it enabled efficient, intermittent writes and reads of the reconstruction sources to/from secondary storage for long simulations.

We launch two-dimensional grids of two-dimensional threadblocks in order to cover one $x_1 - x_2$ slice of the computational domain. The launched threads are allowed to loop over the x_3 dimension, permitting the time-space tiling

implementation to re-use all computational kernels of the baseline (i.e. non-tiled) implementation. Furthermore, the implementation uses several CUDA streams in order to overlap computations and data transfers. Sychronization is performed on the thread block level for correct shared memory usage within kernels, implicitly within CUDA streams and across streams with the usage of CUDA events [22]. At runtime, the mainly utilized SM units are the memory load and store units along with the (unified) 32-bit CUDA cores for floating point multiplication, addition, subtraction and FMA operations. On architectures without dedicated units for integer operations, the memory address calculations are also performed on the unified 32-bit CUDA cores. In addition, the special functions units are minorly used for computing floating point reciprocals [58]. Beyond the higher-level algorithmic optimizations involving the usage of reconstruction and tiling methods, the GPU code also contains several implementational optimizations. In the following subsections, we briefly review some of these.

4.1 Shared memory and register pipelining for efficient differentiation

Implementing the numerical differentiations of the finite-difference scheme with low memory redundancy is of high importance. The approach of Micikevicius [25] aims to reduce the amount of redundant memory operations in the global memory space, and we briefly review it here. The implementation of the horizontal derivatives utilizes two-dimensional blocks of shared memory in the $x_1 - x_2$ plane, allowing a greater data reuse on the thread block level. For realistic differentiator half-lengths, there is not enough shared memory available per streaming multiprocessor to extend this approach to the vertical derivative, i.e. along the x_3 direction. Instead, a per-thread register pipeline stores the values of the previous and next $x_1 - x_2$ slices that are required for calculating the vertical derivative at a single point (x_1, x_2, x_3). One new value is loaded and another is omitted when moving to the next $x_1 - x_2$ slice, giving a re-use of $2L - 1$ values in the pipeline. By reducing the memory operation redundancy, we move the performance of the implementation towards the upper bound set by the roofline model.

4.2 Prefetching of unified memory

Utilization of more Unified Memory than the physical size of device memory leads to frequent page faulting when computational algorithms traverse across the allocated pages. In order to prevent excessive page faulting and improve data locality, the CUDA driver includes certain algorithm-oblivious page migration heuristics[45]. Because time-space tiling is a structured algorithm with a predictable memory access pattern, it is possible to improve on the default behaviour of the driver. Our FWI implementation can therefore override the heuristics by utilizing the asynchronous prefetching functionality of the runtime API. Prefetching can potentially reduce the number of warp stalls and hide the latency induced by page faulting. We allow the least recently used (LRU) policy of the page migration engine to control the eviction of pages from the GPU.

4.3 Wavefield reconstruction

The source functions for wavefield reconstruction can in principle be utilized with an arbitrarily shaped boundary $\partial\mathcal{V}$. However, interpolation of wavefield quantities quickly becomes expensive when the boundary points do not coincide with any finite-difference grid nodes. In this case, interpolation of one wavefield quantity at a single boundary point requires $(2L)^3 \sim \mathcal{O}(L^3)$ FMA and memory operations. For comparison, the FD stencil formulas are distributed according to $\mathcal{O}(L)$ operations at each grid point.

Placing the reconstruction boundary along the grid nodes of the reference FD grid leads to a reduction in the operation count from $\mathcal{O}(L^3)$ to $\mathcal{O}(L)$ for interpolating staggered wavefield quantities. Furthermore, the non-staggered wavefield quantities can be serviced by only one operation. Interpolation is required for both recording and injection of the reconstruction sources. These interpolation operations are implicit in the operations at lines 4 and 29 of Algorithms 1 and 2, respectively. In the following, we first consider the interpolation related to recording of the reconstruction sources before relating this to their injection.

The interpolation of any particle velocity onto $\partial\mathcal{V}$ reads

$$v_i(\mathbf{x}, t)|_{\mathbf{x}\in\partial\mathcal{V}} = \mathcal{I}_i^- v_i\Big(\mathbf{x} + \mathbf{e}_i \frac{1}{2}\Delta x_i, t\Big), \tag{34}$$

when $\partial\mathcal{V}$ is aligned with the reference grid and $\mathbf{e}_i$ is a unit vector in direction x_i. Per definition, this operation requires $2L$ FMA and memory operations.

The diagonal terms of the stress tensor, τ_{ii}, are located on the reference grid and require no interpolation, only a load operation. The traction in Eq. 32 furthermore requires the off-diagonal quantities τ_{ij} $(i \neq j)$, which are staggered in two spatial dimensions. The corresponding interpolation onto the boundary $\partial\mathcal{V}$ is given by

$$\tau_{ij}(\mathbf{x}, t)|_{\mathbf{x}\in\partial\mathcal{V}} = \mathcal{I}_i^- \mathcal{I}_j^- \tau_{ij}\Big(\mathbf{x} + \mathbf{e}_i \frac{1}{2}\Delta x_i + \mathbf{e}_j \frac{1}{2}\Delta x_j, t\Big), \tag{35}$$

which in a naïve implementation requires $(2L)^2$ operations for each point on the boundary. However, the reconstruction boundary is the surface of a cuboid volume when aligned with the reference FD grid. Such surfaces can be seen as the union of six piecewise continuous parts, *viz* $\partial\mathcal{V} = \partial\mathcal{V}_1 \cup \partial\mathcal{V}_2 \cup \cdots \cup \mathcal{V}_6$. Each boundary piece is a rectangular region in which the interpolation of equation 35 can be performed in a two-pass approach

$$\tau_{ij}(\mathbf{x}, t)|_{\mathbf{x}\in\partial\mathcal{V}_k} = \mathcal{I}_i^- \Big(\mathcal{I}_j^- \tau_{ij}(\mathbf{x} + \mathbf{e}_i \frac{1}{2}\Delta x_i + \mathbf{e}_j \frac{1}{2}\Delta x_j, t)\Big), \tag{36}$$

where the wavefield is first interpolated in direction x_j on all nodes of $\partial\mathcal{V}_k$, including all required halo points for the subsequent interpolation in direction x_i. The two-pass interpolation of τ_{ij} requires a number of operations for all N by M grid points on $\partial\mathcal{V}_k$ according to

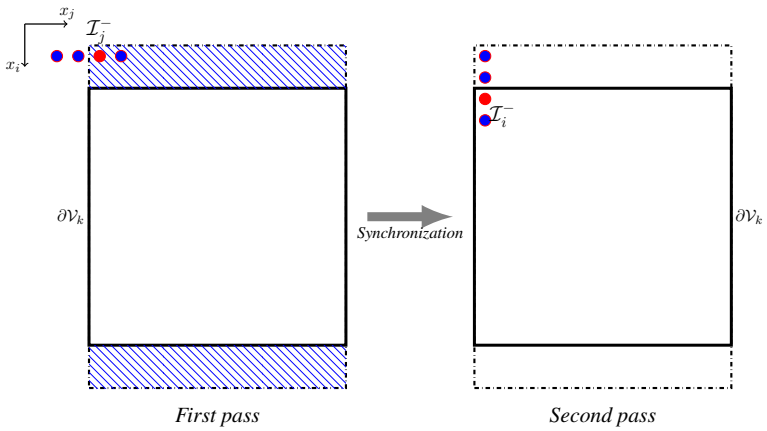


Fig. 9 Two-pass interpolation of doubly staggered wavefield quantities onto the rectangular boundary region $\partial\mathcal{V}_k$, denoted by the solid black lines. In the first pass, the wavefield is interpolated in direction x_j on $\partial\mathcal{V}_k$ and on a halo zone, marked in striped blue between the solid and the dash-dotted lines. The second pass performs interpolation in direction x_i only on $\partial\mathcal{V}_k$, with some data dependencies located in the halo zone. The respective interpolation stencils are shown for a half-length equal to 2 with the output point denoted as solid red node. The stencil dependencies for the output point in each interpolation procedure include itself and three adjacent nodes (color figure online)

$$\text{OPS} = 2L \times (2NM + 2LN), \tag{37a}$$

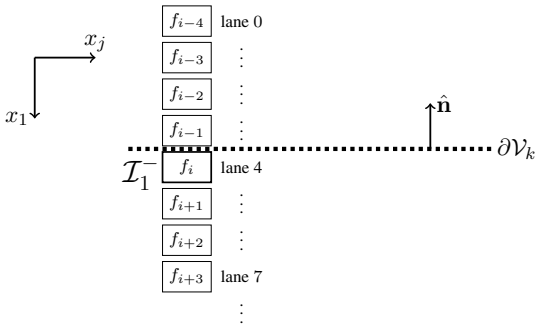
or

$$\text{OPS} = 2L \times (2NM + 2LM), \tag{37b}$$

depending on the exact orientation of the staggering of τ_{ij} with respect to the orientation of $\mathcal{V}_k$. According to Eq. 37, the number of operations for a single point is distributed according to $\frac{\text{OPS}}{NM} \sim \mathcal{O}(L)$, provided that the extents of the boundary is significantly larger than the interpolation half width. Our proposed rearrangement therefore reduces the number of operations by a factor L. The two-pass interpolation approach is shown in Fig. 9 for a staggering parallel to the plane of $\mathcal{V}_k$. The staggering direction orthogonal to the boundary should always be considered in the first step, if present in a considered wavefield quantity. Adherence to this rule minimizes the size of the memory buffers required by the two-pass method, as well as the amount of memory operations. We utilize memory buffers in the global memory address space and utilize two separate kernel launches for implementing the two-step approach.

A two-dimensional grids of two-dimensional threadblocks is launched on all boundary sections, except the two sections with normal vectors parallel to the x_1 direction. This direction is in memory represented with the fastest varying index. The standard thread setup would serialize every memory access required by $\mathcal{I}_1^-$ for all lanes across any warp, leading to a substantial amount of wasted bandwidth and serialization. Special considerations are therefore required in order to maintain an

Step 1: Warp load at boundary



Step 2: Intra-warp reduction

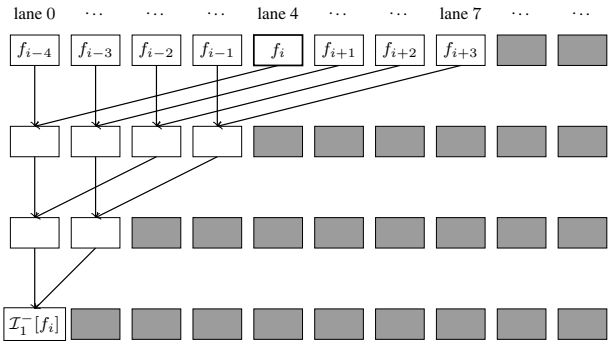


Fig. 10 Intra-warp reduction for efficient interpolation at a boundary with normal vector $\hat{\mathbf{n}}$ aligned with the x_1 axis. In the first step, the valid lanes load the required parts of f for computing the backwards interpolation with $\mathcal{I}_1^-$ at grid point i. In the second step, an intra-warp reduction operation is performed to compute the result at lane 0. The contributing lanes at each substep of the reduction are shaded white, whereas the inactive lanes are shaded in a gray colour. For this display, an interpolation half-length equal to 4 has been used

optimal memory access pattern on these boundaries. We launch one *warp* per point (x_2, x_3) on the x_1 aligned boundaries in order to utilize the interpolator $\mathcal{I}_1^-$ efficiently. All values required for the interpolator are then provided by *one* global load instruction on the warp level. The output of the interpolation is implemented with an intra-warp reduction operation, utilizing warp shuffle instructions [23]. This procedure is demonstrated in Fig. 10. All threads with lane indices greater than $2L-1$ are masked out during the reduction operation, and the number of threads participating is halved after each step. The proposed approach can be extended to cases where $2L$ is greater than the warp size by considering an intra-block reduction operation. Partial reduction would then be performed within warps prior to distributing partial sums to shared memory, from which the first warp of the block would perform the final reduction, see, e.g. [23].

The interpolation routines for injection of the reconstruction-type source functions are highly reminiscent of those used for recording. The body force source f_i^{rec} must be interpolated with the operator $\mathcal{I}_i^+$ in order to be injected to the particle velocity v_i. The deformation rate source h_{kl}^{rec} $(k \neq l)$ is injected to τ_{ij} and must be interpolated with $\mathcal{I}_i^+$ and $\mathcal{I}_j^+$. In similarity with the recording procedure, this is done with a two-pass approach. The minimalistic memory buffers are herein re-used. Finally, the source functions h_{kk}^{rec} are injected to the elements τ_{ii} and therefore require no interpolation. The grid and threadblock structure for injection utilizes the same structure used for recording. An optimal memory access pattern on all boundary sections is hence provided. Injection of the reconstruction sources is thread safe due to the highly regular structure of the boundary and by design of the injection procedures. We therefore avoid expensive atomic operations or, even worse, serialization through the usage of locks.

4.4 Flow chart overview of time-space tiled gradient computations

The flowchart in Fig. 11 demonstrates the higher level overview of our implementation of the loops in Algorithm 2, yet with time-space tiling. For brevity, there are minor differences compared to the exact lines of Algorithm 2: The explicit call to Algorithm 1 is assumed to have been performed, and the residual sources have been created. The loop *structure* in this figure is exactly the same as that of the tiled velocity-stress stencil scheme, found in Algorithm 3. In Fig. 11, the outermost loop moves backwards in time from $t = T$ to $t = 0$ in batches of W_t timesteps. At the start of each batch, the source functions for wavefield reconstruction and the adjoint sources are copied to the GPU in a buffered fashion, i.e. W_t timesteps of these quantities are transferred. The implementation then loops over all tiles. If desired, the variables on the given tile are prefetched with UM prefetching. Otherwise, the UM page migration system handles all page faults on demand. The forward and adjoint fields are propagated backwards in time, injected sources to and combined to compute the gradient contributions at all timesteps within the tile.

5 Results

We consider a selection of benchmarks in order to assess the performances of the wavefield reconstruction method and the out-of-core performance of both the wave equation simulation and gradient computations. Ultimately, a realistic FWI gradient computation example is presented, in which we compare the performance of our GPU implementation on a selection of devices, and compare it to our in-house CPU-based implementation. We consider the hardware selection in Table 2, in which our main development and benchmark platform is the Geforce GTX 1070. The compilers and main compiler flags used for the benchmarks are given in Table 3.

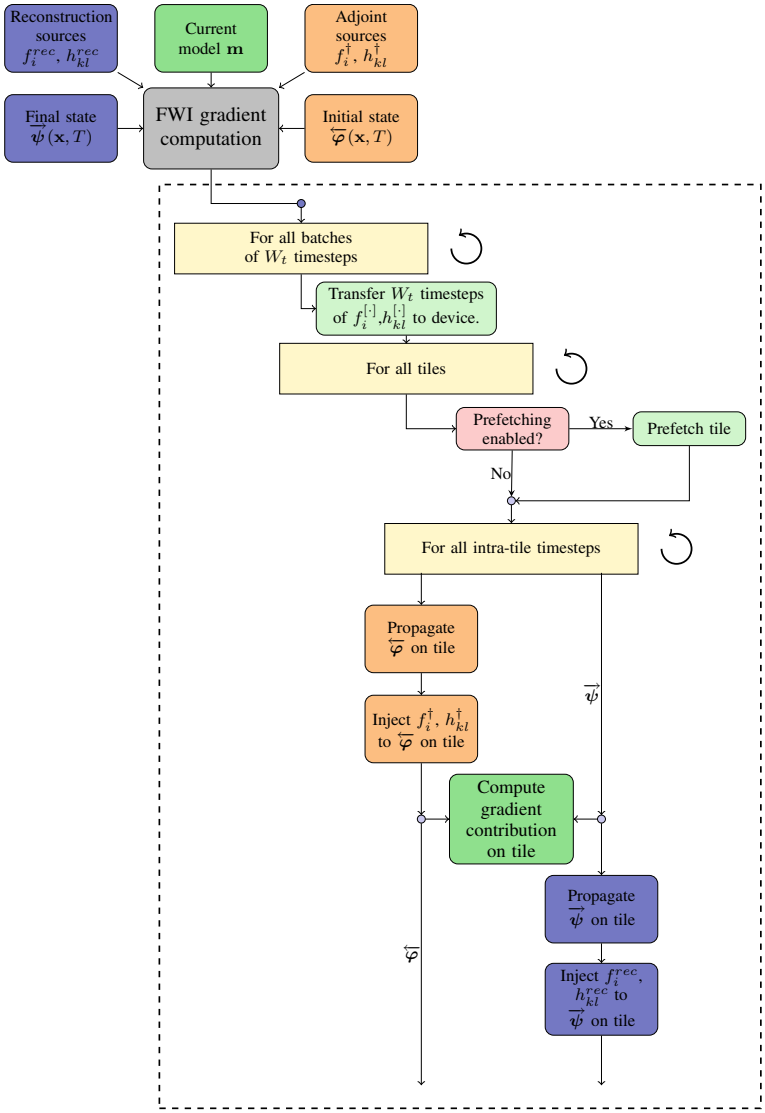


Fig. 11 Flowchart of the implemented FWI gradient computation with time-space tiling and wavefield reconstruction. This constitutes a graphical display of the main computations of Algorithm 2 with time-space tiling, also including information of where explicit transfers and, optionally, UM prefetching is performed. W_t refers to the temporal tile size. Refer to the main text for further explanation

The roofline model suggests that the maximum achievable performance in terms of *floating point operations* is proportional to the differentiator half-length in the bandwidth bound region. An equivalent observation is that increasing the differentiator half-length permits running FWI on a coarser grid, which reduces runtime for a fixed differentiation accuracy. This analysis assumes that all values of L can be

Table 2 The compute units used for benchmarking

Computational unit	Memory capacity	Memory bandwidth	Release	MSRP
Nvidia GeForce GTX 1070	8 GiB	256 GB/s	2016	$379
Nvidia GeForce GTX TITAN X	12 GiB	336.5 GB/s	2015	$999
Nvidia Tesla P100	16 GiB	732 GB/s	2016	$5699 [50]
2 × Intel Xeon E5-2660V3	128 GiB	2 × 68 GB/s	2014	2 × $1445

The Intel CPU is a dual-socket system where the denoted memory capacity is the *installed* amount of memory

Table 3 The compilers used for benchmarking

Compiler	Version	Compiler flags

`nvcc`	9.1/9.1.85/10.2	`–x=cu -arch=sm_52 -lineinfo -Xptxas –opt-level=3 -Xptxas \`
		`–warn-on-spills -Xptxas -allow-expensive-optimizations=true \`
		`-Xcompiler -fopenmp -std=c++11`

`icpc`	18.0.1	`–std=c++11 -O3 -march=native -diag-disable 161 \` `-diag-disable 3180 -qopenmp`

The host compilers refer to the CPU benchmarks

met with sufficient on-chip hardware resources, such that the redundancy in memory traffic does not increase. A suitable value for L in terms of *runtime* therefore achieves a good balance between numerical accuracy and the hardware resources required in the (efficient) implementations of the differentiators. Obscenely large half-lengths are not desirable as the required spatial grid sampling only asymptotically approaches the Nyquist sampling criterion for large values of L [17]. We consistently chose to utilize a half-length $L = 8$ with coefficients minimizing the numerical dispersion error [17] for our benchmarks.

5.1 Wavefield reconstruction performance

We measure the performance of the wavefield reconstruction method by evaluating the slowdown introduced by recording the reconstruction sources during forward simulation. Similarly, the time-reversed simulation with injection of the reconstruction sources is also compared to the plain forward simulation. The problem size considered is a cubic domain of 256^3 grid cells, the reconstruction boundary is the union of six quadratic pieces of size 200×200 grid cells and 1000 timesteps are performed for each run. The cells outside the reconstruction region are considered part of a PML and padding zone. The relative performances shown in Table 4 demonstrate that the recording and injection procedures induce performance reductions of slightly more than 3% and 7%, respectively. The differences in performance

Table 4 Performance of the wavefield reconstruction implementation relative to a plain forward simulation, with GCU abbreviating grid cell updates

Experiment	Performance (GCU/s)	Relative performance (%)
Forward simulation	4.0681×10^8	100
Forward simulation with recording	3.9369×10^8	96.77
Reverse simulation with injection	3.7811×10^8	92.94

The results have been averaged over three independent runs

Table 5 Performance of extracting snapshots relative to a plain forward simulation, with GCU abbreviating grid cell updates

Experiment	Performance (GCU/s)	Relative performance (%)
Forward simulation	3.7872×10^8	100
Forward simulation with snapshots to system memory	3.3384×10^7	8.77

The results have been averaged over three independent runs

across the two procedures are unsurprising as the recording procedure is a reduction operation, whereas the injection procedure is a scattering operation. The injection procedure trades load operations for store operations, which are satisfied by two, rather than one, memory operations. It should be noted that the two simulations with reconstruction procedures include the effect of (asynchronous) transfers of the reconstruction sources across the interconnection. The interconnection transfers are here performed every 10'th timestep.

Extracting snapshots of the state wavefield at each timestep in Algorithm 1 represents a naïve alternative to the utilization of (wavefield) reconstruction methods. We consider the effect of writing snapshots of all components of the state field to system memory. Even for this relatively small example, the size required for the snapshots is 26.82 GiB per 100'th timestep. In order to comfortably fit the snapshots in memory, the number of timesteps is reduced to 100, although it should be noted that considering longer simulation times might require the usage of secondary storage rather than system memory. For a fair comparison, the snapshots are extracted within a region of the same size as that bounded by the reconstruction boundary. This region was chosen to be contiguous in memory in order to minimize any potential bias introduced by the granularity present in UM page migration.

We measure performance in terms of grid cell updates per time and average the results over three independent runs. The results are shown in Table 5, in which the writing snapshots to system memory features a slowdown of approximately a factor 12.

No prefetching has been utilized in this example, hence writing snapshots induces page faults on the host and subsequently on the device when moving to the next timestep.

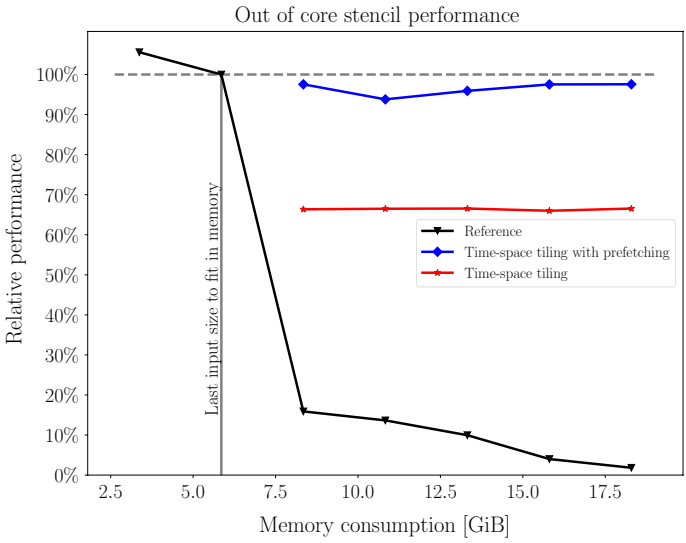


Fig. 12 Out-of-core stencil performance. The grey vertical line separates the in-core (left) and out-of-core (right) regions

5.2 Time-space tiling

In this section, we benchmark the effect of time-space tiling for the wave equation discretization and FWI gradient computations. We consistently refer to the reference implementation as the implementation *without* time-space tiling enabled.

The first tiling example is concerned with the out-of-core performance of the wave equation discretization. Successive simulations were performed on a grid monotonically increasing in size, and performance was measured relative to that achieved at the last input size to fit completely in device memory. We chose the metric grid cell updates per time in order to quantify performance. All measurements are performed in a 'warm-cache' scenario, meaning that the stencil updates belonging to either one timestep or one temporal block were performed prior to the benchmark measurement. In the in-core scenario, the interconnection transfers were therefore provided at the benchmark start. The black curve in Fig. 12 demonstrates that the performance of the reference implementation drops dramatically when the working set exceeds the physical memory. At the start of the out-of-core region, the achieved performance is approximately 16% but decreases monotonically towards 2.5% when the input size is more than twice the size of device memory. The baseline time-space tiling implementation is able to sustain around 66% of the original performance in the out-of-core region. This is a considerable improvement over the reference implementation, yet not entirely satisfactory. With prefetching, we are able to increase the out-of-core performance by approximately 25 percentage points, achieving at least 92% of the original performance.

The second time-space tiling example evaluates the out-of-core performance of the FWI implementation by considering successive gradient computations

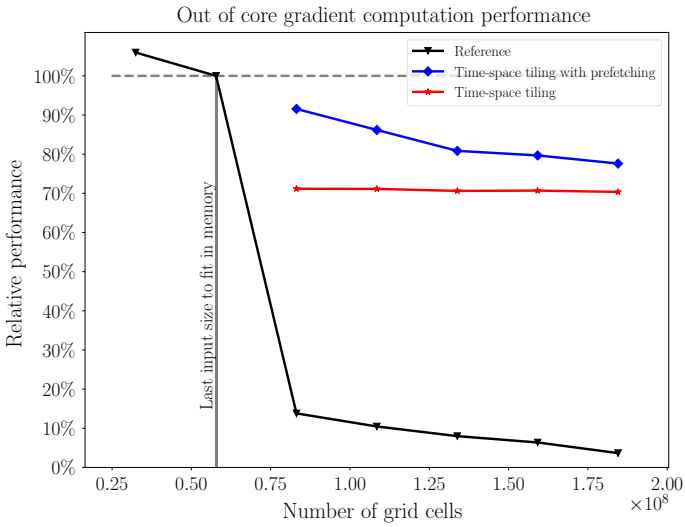


Fig. 13 Out-of-core gradient computation performance. The grey vertical line separates the in-core (left) and out-of-core (right) regions

performed on a grid increasing in size. This represents a benchmark of Algorithm 2 excluding the explicit call to Algorithm 1. We utilize the same relative performance metric as the last example, and all measurements are performed in a 'warm-cache' scenario. Figure 13 demonstrates that the reference implementation takes a severe performance hit in the out-of-core region. The baseline time-space tiling implementation is able to sustain circa 70% of the original performance, whereas including prefetching raises the performance up to 90% of the in-core performance. A slight decrease is noted for larger input sizes, dipping down to about 78%.

5.3 SEG/EAGE overthrust example

We consider a realistic, large-scale gradient computation example in order to demonstrate the applicability and feasability of GPU-based FWI with wavefield reconstruction and time-space tiling. A modified subset of the SEG/EAGE Overthrust model is considered, with a physical size of 8000 m × 14,000 m × 4000 m uniformly discretized on a 20.0 m grid spacing. Including padding zones for the attenuating boundary conditions results in a working set of size 10.05 GiB[1] for FWI gradient computations. The size of the working set is well beyond the device capacity of the GTX 1070, but within the capacity of the GTX Titan X, which does not support Unified Memory oversubscription. We consider a simulation time of 2500 timesteps, place 73 shots along the line $x_1 = 4000$ m and place 13,650 receivers throughout

[1] Measured with the tool `nvidia-smi`.

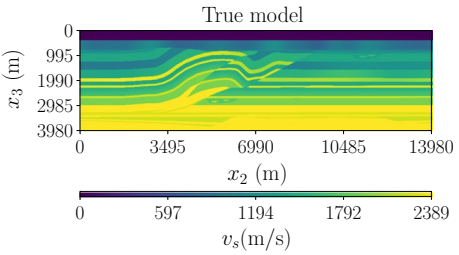


Fig. 14 True shear wave distribution along the slice $x_1 = 4000$ m

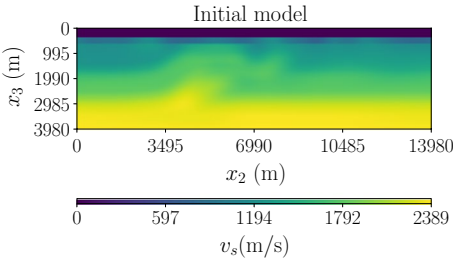


Fig. 15 Initial shear wave distribution along the slice $x_1 = 4000$ m

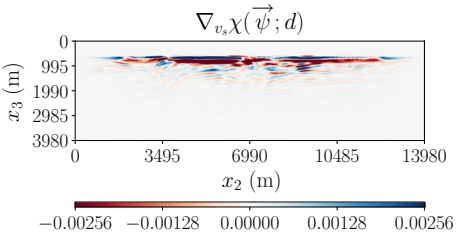


Fig. 16 The calculated gradient for all 73 shots along the slice $x_1 = 4000$ m

the ocean bottom of the model. The total size of the reconstruction sources for all timesteps is in excess of 55 GiB, which we therefore buffer to secondary storage. We calculate the gradient with respect to the shear wave velocity $v_s = \sqrt{\mu/\rho}$, which is a combination of the stiffness gradient terms according to

$$\nabla_{v_s}\chi(\overrightarrow{\boldsymbol{\psi}};\mathbf{d}) = 2\rho v_s \nabla_{\mu}\chi(\overrightarrow{\boldsymbol{\psi}};\mathbf{d}) - 4\rho v_s \nabla_{\lambda}\chi(\overrightarrow{\boldsymbol{\psi}};\mathbf{d}). \tag{38}$$

The true shear-wave velocity model along the shot line $x_1 = 4000$ m is shown in Fig. 14. Computing the misfit gradient in nonlinear optimization methods like FWI requires an initial guess of the subsurface model, which is denoted as the initial model. In our synthetic example, this was generated by applying a Gaussian smoothing kernel to the true model, with the result shown in Fig. 15. For display purposes, we allow the mass density to be constant and the compressional-wave velocity model to be equal to its true model. The calculated gradient is shown along the shot line in Fig. 16, which is essentially an image of the data residuals.

The time required for calculating the gradient scales linearly with the number of shots, and we benchmark the time required to compute the gradient contribution from one shot on all available hardware configurations. For the out-of-core

Table 6 The SEG/EAGE overthrust gradient computation example

Computational unit	Runtime(s)	Speedup
Nvidia GeForce GTX 1070[a]	1.1487×10^4	0.91
Nvidia GeForce GTX 1070[b]	2.4807×10^3	4.24
Nvidia GeForce GTX 1070[c]	2.1027×10^3	5.00
Nvidia GeForce GTX TITAN X	1.5568×10^3	6.75
Nvidia Tesla P100	9.4167×10^2	11.16
2x Intel Xeon E5-2660V3	1.0508×10^4	1.00

[a]Reference iteration space

[b]Time-space tiling

[c]Time-space tiling with prefetching

computations on the GTX 1070, we consider benchmarking without and with time-space tiling, also assessing the effect of prefetching. The CPU implementation features explicit AVX256 vectorization of the x_3 derivatives, but relies on compiler auto-vectorization for the two horizontal derivatives. For this implementation, we utilize the Intel C++ compiler. Parallelization of the host code is performed with the OpenMP compiler directive. The GPU implementation is not a direct port of the CPU code, as the CPU code relies on memory buffers for temporary storing the partial derivatives required for the stencil kernels. On the CPU, this facilitated readability and the implementation of the AVX256 vectorization (without vector folding), as well as the PML absorbing boundary conditions. However, the memory traffic to and from the differentiation buffers raises the number of memory operations required for the stencil kernels, in turn lowering the operational intensity and maximum achievable performance. The GPU version instead utilizes registers, at the cost of a more involved code and decreased occupancy due to an increased hardware resource usage.

At least three runs were performed for each hardware-algorithm combination and we present the benchmarking results in Table 6. Here, it can be seen that the GPU implementation on the most modern and expensive hardware is over an order of magnitude faster than the reference CPU implementation. The desktop-grade GTX 1070 is faster than the CPU implementation *only* when utilizing time-space tiling. Prefetching of Unified Memory also aids the performance on this hardware. It should be noted that the call to Algorithm 1 in the gradient computation algorithm does not require prefetching (nor time-space tiling) for this specific example, as its internal working set fits completely in device memory. The GTX Titan X is of similar age to the CPU configuration and achieves a highly significant speedup. Its main limitation in our implementation is that it does not support oversubscription of UM.

5.4 Comparative analysis

For a comparative analysis, we benchmark our implementation against SeisCL [8], which is freely distributed through [12]. Although being written in OpenCL [18]

and with support for domain decomposition through MPI, there are many comparable features to our work. The elastodynamic forward modelling and FWI gradient computations of SeisCL solves the same state and adjoint-state equation pair, utilizes the same FD discretization method and similarly features PML attenuating boundary conditions. The gradient computation strategy also supports wavefield reconstruction, yet with a different methodology. SeisCL utilizes a direct FD reconstruction method [43, 61] in which all nine fields of the state field are recorded for L values parallel to the boundary, with L again being the FD half-length. This is in contrast to the reconstruction method in our work, which requires only six fields recorded at only the grid cell of the boundary itself. Ignoring minor differences at the corner points, the reconstruction method employed in SeisCL requires a factor $\frac{9L}{6} = \frac{3}{2}L$ more memory than the one implemented in our work.

Another important distinction lies in the way SeisCL performs (efficient) numerical differentiation for the FD implementation. Rather than using a combination of shared memory and registers, SeisCL performs efficient numerical differentiation purely with shared memory, see, e.g. Fig. 4 in [8]. Furthermore, SeisCL relies on multiple GPUs through domain decomposition in order to solve larger problems than can fit in the device memory of a single GPU. In its present implementation, there is no support for transferring the reconstruction sources across the interconnection, such that these quantities must fit in device memory for all simulation timesteps. We benchmark our implementation against SeisCL, although the differences in memory handling greatly restrict the possible benchmarking options to small, in-core problems. Hence, we chose a problem with 200^3 interior grid points along with a width of 12 PML grid points in all directions. The current version of SeisCL supports differentiator half-lengths (L) one through six, although we employ a half-length equal to eight. Provided that any value of L can be met with sufficient on-chip hardware resources, the roofline analysis suggests that runtime is constant for a fixed grid size. We therefore benchmark SeisCL with its default maximum $L = 6$ and with $L = 8$ against our implementation with $L = 8$. Expanding the source code of SeisCL to accommodate longer half-lengths required four minor changes: Expanding the macros used for numerical differentiation (`header_fd.cl`), expanding the table of FD coefficients (`holbercoeff.c`), redefining the macro controlling maximum value of L (`F.h`) and ensuring that the FD symbols were correctly copied when building the OpenCL programs (`F.h`,`clprogram.c`). For this problem size, the maximum number of timesteps which could be simulated for one GTX1070 in SeisCL were only around 80 and 50 for $L = 6$ and $L = 8$, respectively. For both implementations, we consider only the walltime used for computing the FWI gradient and consistently ignore the time required for memory allocation, building of OpenCL programs (SeisCL only) and writing of output.

Table 7 represents the results of the gradient computation benchmark, in which relative performance is simply measured as the quotient of runtimes, with SeisCL as the point of reference. At lower half-lengths and for such small problems, SeisCL is significantly faster. When increasing the half-length to $L = 8$, the runtimes are within a few percentages of one another. We observe that the shared memory usage

Table 7 Relative comparison of *in-core* performance for elastodynamic gradient computations

Implementation	Number of timesteps	Runtime(s)	Relative performance (%)
SeisCL ($L = 6$)	80	6.185	100.00
This work ($L = 8$)	80	10.163	60.86
SeisCL ($L = 8$)	50	6.214	100.00
This work ($L = 8$)	50	6.647	93.49
This work[a] ($L = 8$)	50	5.863	105.98

[a]Average runtime per 50 timesteps for a simulation involving 500 timesteps in total

per thread in SeisCL increases dramatically when increasing the differentiator half-length, leading to a lower achieved parallelism and performance. This behaviour is generally consistent with Fig. 9 in [8]. Furthermore, due to the very few number of timesteps allowed in this benchmark, we observe that the runtime of our code is slightly elevated due to page faults at the startup of certain procedures. Indeed, when considering the average runtime per 50th timestep in a simulation of 500 timesteps, the in-core performance is slightly higher than that of SeisCL. This result is presented in the final row of Table 7.

6 Discussion and possible improvements

6.1 General discussion

Combining the performance results in Table 6 with the pricing guidelines in Table 2, we observe that both consumer and workstation grade GPUs provide a compute platform for FWI economically competitive to workstation grade CPUs. The seismic community largely does not utilize the enhanced features that workstation grade GPUs provide, such as high-performance double precision arithmetic and error-correcting code memory, giving a particular economic advantage for the consumer grade counterparts. We have shown an implementation of FWI highly suitable to GPUs even with limited device memory, as it maintains large parts of its performance in out-of-core scenarios.

When comparing our work to the state-of-the-art for elastodynamic FWI gradient computations on GPUs, in the form of SeisCL [8, 12], the following observations are in order:

- Direct stencil reconstruction methods consume significantly more memory than the reconstruction method employed in our work, e.g. an order of magnitude more. When keeping the reconstruction sources on the GPU, this can severely restrict the achievable simulation times. Due to the differences in memory consumption, it is not obvious whether transferring the reconstruction sources in

SeisCL between device and host can be hidden as well as achieved in our work, particularly in scenarios where buffering to secondary storage would be required.
- The key ingredients of our implementation permit one to compute FWI gradients on significantly larger models and extended simulation times, yet retaining high efficiency, than previously possible with a small number of GPUs per shot. This is attractive as the number of shots in a seismic survey typically outnumber the amount of available GPUs.
- For equivalent FD half-lengths, our FWI implementation shows very comparable *in-core* performance. For shorter half-lengths, SeisCL demonstrates improved (in-core) performance. In our (compiled) code, it appears that register inefficiencies are slightly restricting maximum in-core performance, e.g. see the discussion further below. For SeisCL, the per-thread usage of shared memory increases drastically when increasing the FD half-length, leading to a reduced parallelism.

Unified Memory greatly reduced the implementational effort required for the out-of-core GPU computations as it provides a consistent virtual address space across the physical host and device memory pools. We have shown that prefetching instructions are decently effective in minimizing warp stalls due to page failures on the device. We believe that the slightly reduced prefetching efficiency observable for the largest out-of-core scenarios in Fig. 13 can be avoided by, e.g. optimizing the CUDA stream placement of the prefetching instructions. We attempted to improve the page eviction policy by specifying prefetches of evictable data from device to host. This proved to be challenging as device-to-host prefetch operations will stall the CPU if not deferred by the driver, leading to a reduction in overlapping opportunities [45]. Although not theoretically optimal, we therefore allowed the default LRU policy to control page eviction.

The current CUDA prefetching functionality has at least one caveat in the setting of time-space tiling techniques. In order to hide the interconnection transfers, the spatial tile size W_{x_3} in one-dimensional time-space tiling must be large enough as to provide sufficient data reuse compared to the amount of data loaded and evicted across intra-tile timesteps. This requirement might not be satisfiable if the horizontal dimensions x_1 and x_2 are large and a long differentiator half-length is utilized. An effective solution is to extend the tiling technique to one or both of the horizontal dimensions, which is algorithmically straight-forward. However, the Unified Memory prefetching functionality does not allow a strided access pattern, such as provided by the copying routines for explicitly managed memory. We speculate that the lack of functionality arises because Unified Memory transfers move data at the system page granularity, which might lead to a higher data movement than necessary. The data movement granularity might also present unexpected page ownership conflicts in heterogenous computing systems or when using multiple GPUs. We believe, however, that the data movement redundancy of UM for two-dimensional tiling (x_2 and x_3) in large-scale FWI would be negligible, and particularly so for small system page sizes. Specifying many smaller prefetching operations does not appear as a

feasible solution because the achievable interconnection throughput depends significantly on the message size [46].

We utilized the roofline model in a simplistic fashion to motivate our memory-oriented optimizations, in the sense that we considered all kernels to be independent and without significant cache hits across them. The constituent kernels on GPUs are separated into separate kernel launches in order to enable inter-thread block synchronization and to reduce register pressure. Furthermore, GPUs rely heavily on thread-level parallelism in order to hide latencies to device memory and the cache sizes present are small compared to what is found, e.g. on CPUs. Improving cache hits on GPUs with tiling techniques therefore does not appear as viable as on CPUs. This holds in particular considering our need for longer differentiator half-lengths, which increase the extent at which dependencies propagate in the stencil scheme. These considerations support viewing the constituent kernels of our implementation as separate (in the roofline analysis). The FWI implementation also requires significantly more functionality than merely the wave equation discretization, and providing accurate estimates of operational intensity for all involved routines appears daunting and tedious.

6.2 Possible improvements

A possible improvement would be to consider a multi-GPU implementation of our work. However, as FWI is commonly performed with simulations of many, independent shots, it is attractive to consider task-based parallelism by assigning one GPU to each shot. For this reason, we have not yet considered a multi-GPU implementation. If desired, however, time-space tiling facilitates multi-GPU simulations in an arguably easier formulation than domain decomposition. The latter requires both reads and writes at each shared boundary, but parallelogram tiling performs all reads on one tile boundary and all writes on the other. Furthermore, one can allow greater parallelism in tile execution by changing the shape of the iteration space from parallelograms to diamonds [11, 35, 36]. Diamond tiling has a structured iteration space similar to parallelogram tiling and changing the existing implementation appears straight-forward.

In order to achieve smaller tile sizes and, therefore, better out-of-core performance, one could consider to extend the tiling technique to one or both of the horizontal spatial dimensions. As remarked above, this would be best suited to an implementation relying purely on explicitly managed memory, rather than UM. Alternatively, a worthwhile improvement, while using UM, could be to override the default page eviction policy in order to improve the interconnection transfers.

We also note that the achieved occupancy of the FD stencil kernels of our FWI implementation is suboptimal due to high register usage, which was present also before the introduction of register pipelining. A quick analysis of the PTX assembly code generated by the compiler reveals that it performs extensive loop unrolling to split the integer calculations of memory addresses and the FMA operations belonging to the partial derivatives. We believe that this behaviour is in place on the Pascal

and Maxwell architectures due to a lack of dedicated functional units for integer operations. The elevated register usage is also partly due to our implementation of the PML attenuating boundary condition. Limiting the register usage by invocation of compiler options lead to an increased occupancy but decreased performance. This slowdown can be argued from an increased need of flushing the instruction pipeline of the multi-purpose functional units and increased access times due to register spilling. It could be interesting to test our implementation with stencil-aware compilers allowing a greater register reusage, or on GPUs with dedicated functional units for integer arithmetic. Higher occupancy enables better utilization of the device DRAM, which would increase the achieved performance further.

6.3 Applications in other fields

The proposed approach for gradient computations naturally lends itself to closely related algorithms such as the imaging technique known as reverse-time migration (RTM) [7]. RTM builds an image of the scatterers of a medium by computing the cross-correlation between a wavefield modelled from the sources with a backwards-propagating scattered wavefield. The cross-correlation involved in RTM imaging is therefore computationally the same as the adjoint-state gradient computation in FWI. More generally, our approach can be applied to many types of inversion and imaging procedures in lossless media, i.e. for waveforms such as acoustic, quantum-mechanic, elastodynamic and electromagnetic waves in non-conductive media. As the two main optimizations are targeted at the memory hierarchy, we envision that the proposed approach can be successfully applied to other types of accelerators.

7 Conclusion

This paper has presented an implementation of gradient-based elastodynamic FWI on GPUs which utilizes wavefield reconstruction methods and time-space tiling to circumvent the low bandwidth of the device to host interconnection. A hybrid approach utilizing Unified Memory for the wavefields, medium parameters and the absorbing boundary conditions and explicitly managed memory for all auxiliary quantities was implemented. We found this to strike a satisfying balance in simplifying the time-space tiling implementation while retaining full control of the memory transfers of sources, receivers and quantities related to wavefield reconstruction. We have provided a discussion of key optimizations for the reconstruction method within the staggered-grid FD discretization of the elastodynamic wave equation. Our benchmarks demonstrate that the optimized implementation of the reconstruction method has negligible impact on the wave equation simulation performance, in terms of grid cell updates per second. Furthermore, time-space tiling allows one to retain most of the performance in out-of-core scenarios, especially when overriding the UM page-migration engine with prefetching hints.

Acknowledgements O.E. Aaker would like to thank Bart Iver van Blokland for insightful discussions. The authors would like to thank NTNU and Aker BP ASA for the code collaboration through the Codeshare project. This work has been financially supported by AkerBP ASA and by the Research Council of Norway (NæringsPhD Grant 291192).

References

1. Aaker OE, Raknes EB, Pedersen Ø, Arntsen B (2020) Wavefield reconstruction for velocity-stress elastodynamic full waveform inversion. Geophys J Int 222(1):595–609. https://doi.org/10.1093/gji/ggaa147
2. Aki K, Richards PG (2002) Quantitative seismology. University Science Books. https://doi.org/10.1016/S0065-230X(09)04001-9. arXiv:1011.1669v3
3. Amundsen L, Robertsson JO (2014) Wave equation processing using finite-difference propagators, part 1: wavefield dissection and imaging of marine multicomponent seismic data. Geophysics 79(6):287–300. https://doi.org/10.1190/GEO2014-0151.1
4. Anandtech (2017) PCI-SIG finalizes and releases PCIe 4.0, version 1 specification: 2x PCIe bandwidth and more. https://www.anandtech.com/show/11967/pcisig-finalizes-and-releasees-pcie-40-spec. Accessed 13 May 2020
5. Broggini F, Vasmel M, Robertsson JOA, van Manen DJ (2017) Immersive boundary conditions: theory, implementation, and examples. Geophysics 82(3):1MJ–Z23. https://doi.org/10.1190/geo2016-0458.1
6. Cheng J, Grossman M, McKercher T (2014) Professional CUDA C programming. Wiley, New York
7. Etgen J, Gray SH, Zhang Y (2009) An overview of depth imaging in exploration geophysics. Geophysics 74(6):WCA5–WCA17. https://doi.org/10.1190/1.3223188
8. Fabien-Ouellet G, Gloaguen E, Giroux B (2017) Time-domain seismic modeling in viscoelastic media for full waveform inversion on heterogeneous computing platforms with OpenCL. Comput Geosci 100:142–155. https://doi.org/10.1016/J.CAGEO.2016.12.004
9. Fichtner A (2011) Full seismic waveform modelling and inversion. Springer, Berlin. https://doi.org/10.1007/978-3-642-15807-0
10. Fornberg B (1988) Generation of finite difference formulas on arbitrarily spaced grids. Math Comput 51(184):699. https://doi.org/10.2307/2008770
11. Fukaya T, Iwashita T (2018) Time-space tiling with tile-level parallelism for the 3D FDTD method. In: ACM International Conference Proceeding Series. https://doi.org/10.1145/3149457.3149478
12. Gabriel Fabien-Ouellet (2016) SeisCL. https://github.com/gfabieno/SeisCL. Accessed 27 Apr 2020
13. Graves RW (1996) Simulating seismic wave propagation in 3D elastic media using staggered-grid finite differences. Bull Seismol Soc Am 86(4):1091–1106
14. Haime GC, Wapenaar CP (1989) Inverse elastic wave field extrapolation. In: 1989 SEG Annual Meeting. https://doi.org/10.1190/1.1889496
15. Harris M (2013) Unified Memory in CUDA 6. https://devblogs.nvidia.com/unified-memory-in-cuda-6/. Accessed 24 Apr 2020
16. Harris M (2014) How NVLink will enable faster, Easier Multi-GPU Computing | NVIDIA Developer Blog. https://devblogs.nvidia.com/how-nvlink-will-enable-faster-easier-multi-gpu-computing/. Accessed 12 June 2020
17. Holberg O (1987) Computational aspects of the choice of operator and sampling interval for numerical differentiation in large- scale simulation of wave phenomena. Geophys Prospect 35(6):629–655. https://doi.org/10.1111/j.1365-2478.1987.tb00841.x
18. Khronos Group (2009) The OpenCL specification—version 1.0. Khronos Group Specifications
19. Knap M, Czarnul P (2019) Performance evaluation of Unified Memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs. J Supercomput. https://doi.org/10.1007/s11227-019-02966-8
20. Komatitsch D, Erlebacher G, Göddeke D, Michéa D (2010) High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. J Comput Phys 229(20):7692–7714. https://doi.org/10.1016/J.JCP.2010.06.024

21. Lailly P (1983) The seismic inverse problem as a sequence of before stack migrations. In: Conference on Inverse Scattering, Theory and Applications, Society for Industrial and Applied Mathematics
22. Luitjens J (2014) CUDA streams: best practices and common pitfalls. In: GPU Technology Conference
23. Luitjens J (2014) Faster parallel reductions on Kepler. https://devblogs.nvidia.com/faster-parallel-reductions-kepler/. Accessed 24 Apr 2020
24. Michéa D, Komatitsch D (2010) Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. Geophys J Int 182(1):389–402. https://doi.org/10.1111/j.1365-246X.2010.04616.x
25. Micikevicius P (2009) 3D finite difference computation on GPUs using CUDA. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units—GPGPU-2. ACM Press, New York, USA, pp 79–84. https://doi.org/10.1145/1513895.1513905. http://portal.acm.org/citation.cfm?doid=1513895.1513905
26. Mittet R (1994) Implementation of the Kirchhoff integral for elastic waves in staggered-grid modeling schemes. Geophysics 59(12):1894–1901. https://doi.org/10.1190/1.1443576
27. Mittet R, Arntsen B (2000) General source and receiver positions in coarse-grid finite-difference schemes. J Seism Expl 9:73–92
28. Nguyen A, Satish N, Chhugani J, Kim C, Dubey P (2010) 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In: 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, pp 1–13. https://doi.org/10.1109/SC.2010.2. http://ieeexplore.ieee.org/document/5645463/
29. Nickolls J, Dally WJ (2010) The GPU computing era. IEEE Micro. https://doi.org/10.1109/MM.2010.41
30. Nocedal J, Wright S (2006) Numerical optimization, 2nd ed. https://doi.org/10.1007/978-0-387-40065-5. NIHMS150003
31. Nvidia (2016) Whitepaper NVIDIA Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. Accessed 27 Apr 2020
32. Nvidia (2017) Nvidia Tesla V100 GPU architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed 27 Apr 2020
33. Nvidia (2018) Nvidia turing GPU architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. Accessed 27 Apr 2020
34. Nvidia (2020) CUDA C++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed 27 Apr 2020
35. Orozco D, Gao G (2009) Mapping the FDTD application to many-core chip architectures. In: Proceedings of the International Conference on Parallel Processing. https://doi.org/10.1109/ICPP.2009.44
36. Orozco D, Garcia E, Gao G (2011) Locality optimization of stencil applications using data dependency graphs. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pp 77–91
37. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC (2008) GPU computing. In: Proceedings of the IEEE. https://doi.org/10.1109/JPROC.2008.917757
38. Plessix RE (2006) A review of the adjoint-state method for computing the gradient of a functional with geophysical applications. Geophys J Int. https://doi.org/10.1111/j.1365-246X.2006.02978.x
39. Qin Z, Lu M, Zheng X, Yao Y, Zhang C, Song J (2009) The implementation of an improved NPML absorbing boundary condition in elastic wave modeling. Appl Geophys 6(2):113–121. https://doi.org/10.1007/s11770-009-0012-3
40. Raknes EB, Arntsen B (2017) Challenges and solutions for performing 3D time-domain elastic full-waveform inversion. Lead Edge. https://doi.org/10.1190/tle36010088.1
41. Raknes EB, Weibull W (2016) Efficient 3D elastic full-waveform inversion using wavefield reconstruction methods. Geophysics 81(2):R45–R55. https://doi.org/10.1190/geo2015-0185.1
42. Ramírez AC, Weglein AB (2009) Green's theorem as a comprehensive framework for data reconstruction, regularization, wavefield separation, seismic interferometry, and wavelet estimation: a tutorial. Geophysics. https://doi.org/10.1190/1.3237118
43. Robertsson JOA, Chapman CH (2000) An efficient method for calculating finite-difference seismograms after model alterations. Geophysics 65(3):907–918. https://doi.org/10.1190/1.1444787

44. Sakharnykh N (2016) Beyond GPU memory limits with unified memory on Pascal. https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/. Accessed 5 Nov 2019
45. Sakharnykh N (2017a) Maximizing unified memory performance in CUDA|NVIDIA developer blog. https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/. Accessed 3 Dec 2019
46. Sakharnykh N (2017b) Unified memory on pascal and volta. In: GPU Technology Conference (GTC). http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf. Accessed 3 Dec 2019
47. Sanders J, Kandrot E (2011) CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley, Boston
48. Strzodka R, Shaheen M, Pajak D, Seidel HP (2011) Cache accurate time skewing in iterative stencil computations. In: 2011 International Conference on Parallel Processing. IEEE, pp 571–581. https://doi.org/10.1109/ICPP.2011.47. http://ieeexplore.ieee.org/document/6047225/
49. Tarantola A (1988) Theoretical background for the inversion of seismic waveforms including elasticity and attenuation. Pure Appl Geophys PAGEOPH 128(1–2):365–399. https://doi.org/10.1007/BF01772605
50. Techpowerup (2016) NVIDIA Tesla P100 PCIe 16 GB. https://www.techpowerup.com/gpu-specs/tesla-p100-pcie-16-gb.c2888. Accessed 5 Dec 2019
51. Tromp J (2020) Seismic wavefield imaging of Earth's interior across scales. Nat Rev Earth Environ. https://doi.org/10.1038/s43017-019-0003-8
52. Vasmel M, Robertsson JOA (2016) Exact wavefield reconstruction on finite-difference grids with minimal memory requirements. Geophysics 81(6):T303–T309. https://doi.org/10.1190/geo2016-0060.1
53. Venstad JM (2016) Industry-scale finite-difference elastic wave modeling on graphics processing units using the out-of-core technique. Geophysics 81(2):T35–T43. https://doi.org/10.1190/geo2015-0267.1
54. Vigh D, Jiao K, Watts D, Sun D (2014) Elastic full-waveform inversion application using multicomponent measurements of seismic data collection. Geophysics 79(2):R63–R77. https://doi.org/10.1190/geo2013-0055.1
55. Virieux J (1986) P-SV wave propagation in heterogeneous media: velocity- stress finite-difference method. Geophysics 51(4):889–901. https://doi.org/10.1190/1.1442147
56. Virieux J, Operto S (2009) An overview of full-waveform inversion in exploration geophysics. Geophysics 74(6):WCC1–WCC26. https://doi.org/10.1190/1.3238367
57. Williams S, Waterman A, Patterson D (2009) Roofline: an insightful visual performance model for multicore architecture. Commun ACM. https://doi.org/10.1145/1498765.1498785
58. Wilt N (2013) The CUDA handbook: a comprehensive guide to GPU programming. Addison-Wesley, Boston
59. Wolfe MM (1989) More iteration space tiling. In: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing–Supercomputing '89, ACM Press, New York, USA, pp 655–664. https://doi.org/10.1145/76263.76337
60. Wonnacott D (2000) Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In: Proceedings 14th International Parallel and Distributed Processing Symposium, vol 2. IEEE Comput. Soc, pp 171–180. https://doi.org/10.1109/IPDPS.2000.845979. http://ieeexplore.ieee.org/document/845979/
61. Yang P, Gao J, Wang B (2014) RTM using effective boundary saving: a staggered grid GPU implementation. Comput Geosci. https://doi.org/10.1016/j.cageo.2014.04.004
62. Yount C, Duran A (2016) Effective use of large high-bandwidth memory caches in HPC stencil computation via temporal wave-front tiling. In: 2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). IEEE, pp 65–75. https://doi.org/10.1109/PMBS.2016.012. http://ieeexplore.ieee.org/document/7836415/

Affiliations

Ole Edvard Aaker[1] · Espen Birger Raknes[1,2] · Børge Arntsen[3]

Espen Birger Raknes
espen.birger.raknes@akerbp.com

Børge Arntsen
borge.arntsen@ntnu.no

[1] Aker BP ASA, Trondheim, Norway

[2] Department of Electronic Systems, Norwegian University of Science and Technology, Trondheim, Norway

[3] Department of Geoscience and Petroleum, Norwegian University of Science and Technology, Trondheim, Norway